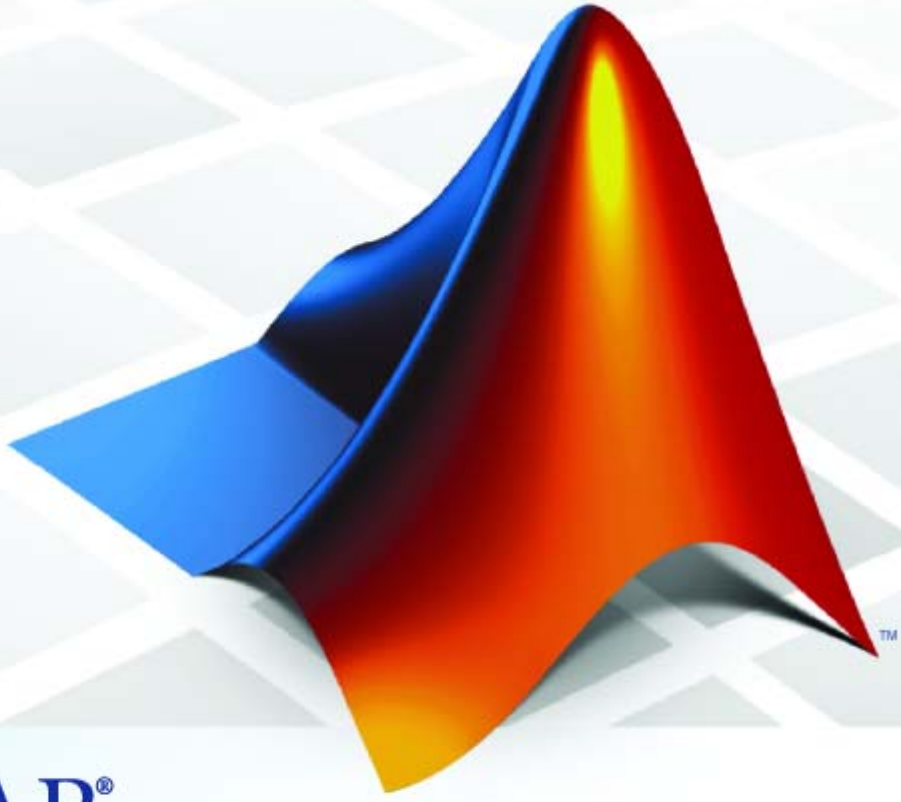
SimMechanics™ 3
User's Guide



MATLAB®
&SIMULINK®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

SimMechanics™ User's Guide

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

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

Revision History

December 2001	Online only	Version 1.0 (Release 12.1+)
July 2002	First printing	Revised for Version 1.1 (Release 13)
November 2002	Online only	Revised for Version 2.0 (Release 13+)
June 2004	Second printing	Revised for Version 2.2 (Release 14)
October 2004	Online only	Revised for Version 2.2.1 (Release 14SP1)
March 2005	Online only	Revised for Version 2.2.2 (Release 14SP2)
September 2005	Online only	Revised for Version 2.3 (Release 14SP3)
March 2006	Online only	Revised for Version 2.4 (Release 2006a)
September 2006	Third printing	Revised for Version 2.5 (Release 2006b)
March 2007	Online only	Revised for Version 2.6 (Release 2007a)
September 2007	Online only	Revised for Version 2.7 (Release 2007b)
March 2008	Online only	Revised for Version 2.7.1 (Release 2008a)
October 2008	Online only	Revised for Version 3.0 (Release 2008b)
March 2009	Online only	Revised for Version 3.1 (Release 2009a)
September 2009	Online only	Revised for Version 3.1.1 (Release 2009b)

Contents

1 Modeling Mechanical Systems

2 Running Mechanical Models

3 Analyzing Motion

4 Motion, Control, and Real-Time Simulation

Index

1

Modeling Mechanical Systems

SimMechanics™ software gives you a complete set of block libraries for modeling machine parts and connecting them into a Simulink® block diagram.

Refer to "Representing Motion" to review body kinematics. If you need more information on rigid body mechanics, consult the physics and engineering literature, beginning with the "Bibliography". Classic engineering mechanics texts include Goodman and Warner [2], [3] and Meriam [8]. The books of Goldstein [1] and José and Saletan [5] are more theoretically oriented.

Representing Machines with Models

In this section...

About Machines

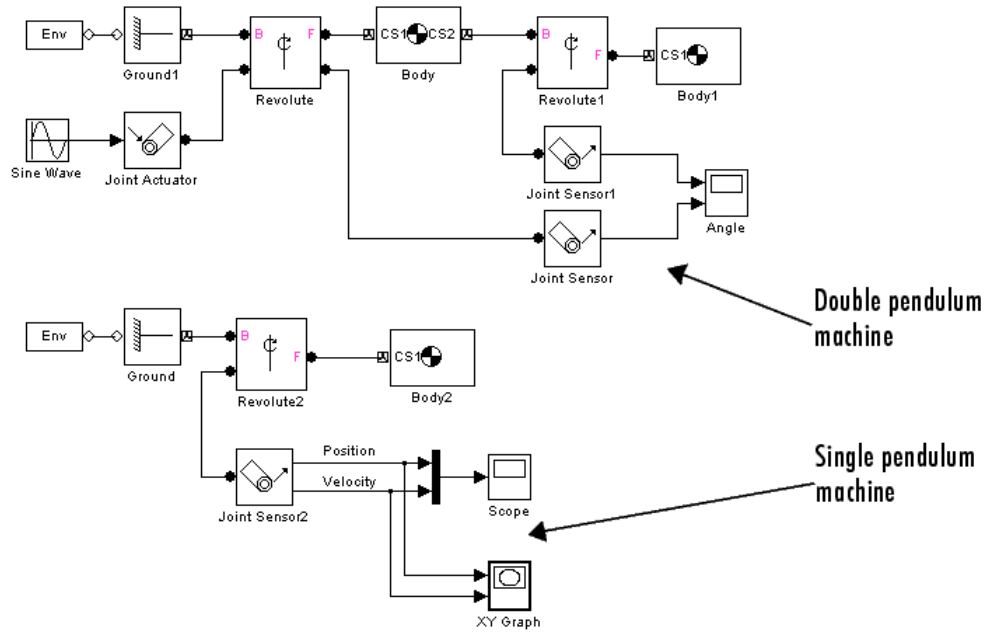
The SimMechanics term *machine* has two meanings.

- It refers to a physical system that includes at least one rigid body. The SimMechanics block library allows you to create Simulink models of machines.
- It also refers to a topologically distinct and separate block diagram representing one physical machine. A model can have one or more machines.

This section explains the nature of machines and SimMechanics models.

About SimMechanics Models

A SimMechanics *model* consists of a *block diagram* composed of one or more *machines*, each of which is a set of connected blocks representing a single physical machine. For example, the following model contains two machines.



Comparison to Other Simulink Models

A SimMechanics model differs significantly from other Simulink models in how it represents a machine.

- An ordinary Simulink model represents the *mathematics* of a machine's motion, i.e., the algebraic and differential equations that predict the machine's future state from its present state. The mathematical model enables Simulink to simulate the machine.
- A SimMechanics model represents the *physical structure* of a machine, the mass properties and geometric and kinematic relationships of its component bodies. SimMechanics software converts this structural representation to an internal, equivalent mathematical model. This saves you the time and effort of developing the mathematical model yourself.

Creating a SimMechanics Model

You create a SimMechanics model in much the same way you create any other Simulink model. First, you open a Simulink model window. Then you

drag instances of SimMechanics and other Simulink blocks from the Simulink block libraries into the window and draw lines to interconnect the blocks (see "Connecting SimMechanics Blocks" on page 1-5).

The SimMechanics block library provides the following blocks specifically for modeling machines:

- Machine Environment blocks set the mechanical environment for a machine. Exactly one Ground block in each machine must be connected to a Machine Environment block. See Chapter 2, "Running Mechanical Models".
- Body blocks represent a machine's components and the machine's immobile surroundings (ground). See "Modeling Grounds and Bodies" on page 1-9.
- Joint blocks represent the degrees of freedom of one body relative to another body or to a point on ground. See "Modeling Degrees of Freedom" on page 1-19.
- Constraint and Driver blocks restrict motions of or impose motions on bodies relative to one another. See "Constraining and Driving Degrees of Freedom" on page 1-38.
- Actuator blocks specify forces, motions, variable masses and inertias, or initial conditions applied to bodies, joints, and drivers. See "Applying Motions and Forces" on page 1-48.
- Sensor blocks measure the forces on and motions of bodies, joints, and drivers. See "Sensing Motions and Forces" on page 1-68.
- Force element blocks model interbody forces. See "Sensing Motions and Forces" on page 1-68.
- Simscape™ mechanical elements model one-dimensional motion and, with certain restrictions, can be interfaced with SimMechanics machines. See "Combining One- and Three-Dimensional Mechanical Elements" on page 1-79.

You can use blocks from other Simulink libraries in SimMechanics models. For example, you can connect the output of SimMechanics Sensor blocks to Scope blocks from the Simulink Sinks library to display the forces and motions of your model's bodies and joints. Similarly, you can connect blocks from the Simulink Sources library to SimMechanics Driver blocks to specify relative motions of your machine's bodies.

Connecting SimMechanics Blocks

In general, you connect SimMechanics blocks in the same way you connect other Simulink blocks: by drawing lines between them. Significant differences exist, however, between connecting standard Simulink blocks and connecting SimMechanics blocks. This section discusses these differences.

Connection Lines

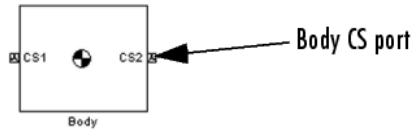
The lines that you draw between standard Simulink blocks, called signal lines, represent inputs to and outputs from the mathematical functions represented by the blocks. By contrast, the lines that you draw between SimMechanics blocks, called *connection lines*, represent physical connections and spatial relationships among the bodies represented by the blocks.

You can draw connection lines only between specialized connector ports available only on SimMechanics blocks (see next section) and you cannot branch existing connection lines. Connection lines appear as solid black when connected and as dashed red lines when either end is unconnected.

Connector Ports

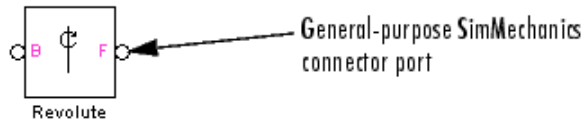
Standard Simulink blocks have input and output ports. By contrast, most SimMechanics blocks contain only specialized *connector ports* that permit you to draw connection lines among SimMechanics blocks. SimMechanics connector ports are of two types: Body CS ports and general-purpose ports.

Body CS ports appear on Body and Ground blocks and define connection points on a body or ground. Each is associated with a local coordinate system whose origin specifies the location of the associated connection point on the body.



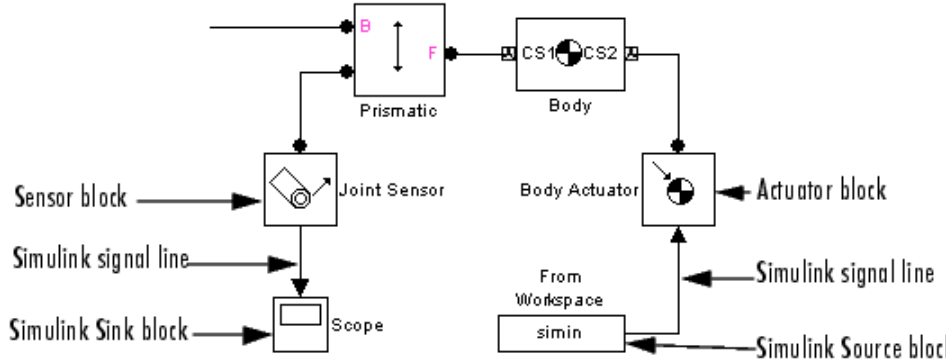
General-purpose connector ports appear on Joint, Constraint, Driver, Sensor, and Actuator blocks. They permit you to connect Joints to Bodies and connect Sensors and Actuators to Joints, Constraints, and Drivers. General-purpose

connector ports appear as circles on the block icon. The circle is unfilled if the port is unconnected and filled if the port is connected.



Interfacing SimMechanics Blocks to Simulink Blocks

SimMechanics Actuator blocks (see "Applying Motions and Forces" on page 1-48) contain standard Simulink input ports. Thus, you can connect standard Simulink blocks to a SimMechanics model via Actuator blocks. Similarly, SimMechanics Sensor blocks contain output ports (see "Sensing Motions and Forces" on page 1-68). Thus, you can connect a SimMechanics model to Simulink blocks via Sensor blocks.



Creating SimMechanics Subsystems

Large, complex block diagram models are often hard to analyze. Enclosing functionally related groups of blocks in subsystems alleviates this difficulty and facilitates reuse of block groups in different models.

You can create subsystems containing SimMechanics blocks that you can connect to other SimMechanics blocks. You do this in two ways:

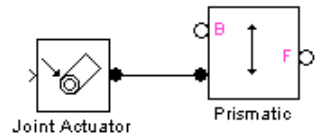
- Automatically
- Manually

The Simulink documentation explains more about creating subsystems.

Creating a Subsystem Automatically

To create a SimMechanics subsystem automatically,

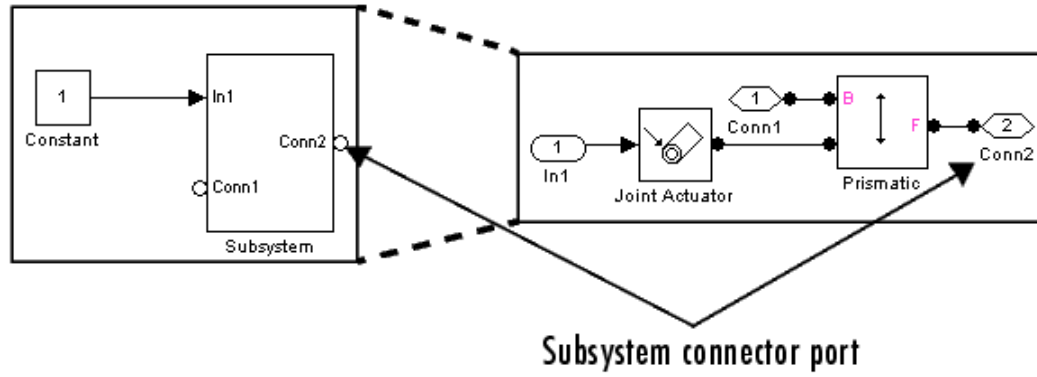
1. Create the subsystem block diagram in your model window, leaving unconnected ports for external connections.



2. Group-select the subsystem block diagram.

3. Select **Create subsystem** from the **Edit** menu of the Simulink model window.

The last step replaces the block diagram with a Subsystem block containing the selected block diagram. It also creates and connects SimMechanics Connection Port blocks for the ports that you left unconnected in the block diagram. The Connection Port blocks in turn create connector port icons on the subsystem icon, enabling you to connect external SimMechanics blocks to the new subsystem.



Creating a Subsystem Manually

Sometimes you need to make a subsystem configured differently from an automatically created one. To create a SimMechanics subsystem manually,

1. Drag a Subsystem block into your model window.
2. Open the Subsystem block.
3. Create the subsystem block diagram in the subsystem window.
4. Drag a Connection Port block from the SimMechanics Utilities library into the subsystem window for each port that you want to be available externally.
5. Connect the external connector ports to the Connection Port blocks.

Creating Custom SimMechanics Blocks with Masks

You can create your own SimMechanics blocks from subsystems, for example, a spring-loaded Joint block or a sphere Body block. To do this, create a block diagram that implements the functionality of your custom block, enclose the diagram as a subsystem, and add a mask (i.e., user interface) to the subsystem. To facilitate sharing your custom blocks across models or with other users, create a Simulink block library and add these masked subsystem blocks to the library. The Simulink documentation explains how to create custom blocks with masks.

Modeling Grounds and Bodies

In this section...

About Bodies and Grounds

The basic components of any mechanism are its constituent rigid bodies. A SimMechanics *body* refers to any point or spatially extended object that has mass. SimMechanics bodies, unlike physical bodies, do not have degrees of freedom. The SimMechanics Bodies library contains two blocks for representing bodies in a Simulink model:

- Ground

  Models a point on an ideal body of infinite mass and extent that serves as a fixed reference point for machines (see "Modeling Grounds" on page 1-9).

- Body

  Models rigid bodies of finite mass and extent, including their attached body coordinate systems (see "Modeling Rigid Bodies" on page 1-11 and "Working with Body Coordinate Systems" on page 1-14).

"Representing Motion" explains, with detailed examples, more about configuring bodies and their coordinate systems in space.

Modeling Grounds

A SimMechanics *ground* refers to a body of infinite mass that acts both as a reference frame at rest for a whole machine and as a fixed base for attaching machine components, e.g., the factory floor on which a robot stands. SimMechanics Ground blocks enable you to represent points on ground in your machine. This in turn enables you to specify the degrees of freedom that your system has relative to its surroundings. You do this by connecting Joint blocks

representing the degrees of freedom between the Body blocks representing parts of your machine and the Ground blocks representing ground points.

Each Ground block has a single connector port to which you can connect a Joint block that can in turn be connected to a single Body block. Each Ground block therefore allows you to represent the degrees of freedom between a single part of your machine and its surroundings. If you want to specify the motion of other parts of your machine relative to the surroundings, you must create additional Ground blocks.

Caution Each machine in a SimMechanics model must contain at least one Ground block connected to a Body block via a Joint block. Each submachine connected by a Shared Environment block must have at least one Ground.

Machine Environment Required for Each Machine

One Ground block in each machine of your model plays a second role, connection to that machine's Machine Environment block, which sets its mechanical environment. See Chapter 2, "Running Mechanical Models".

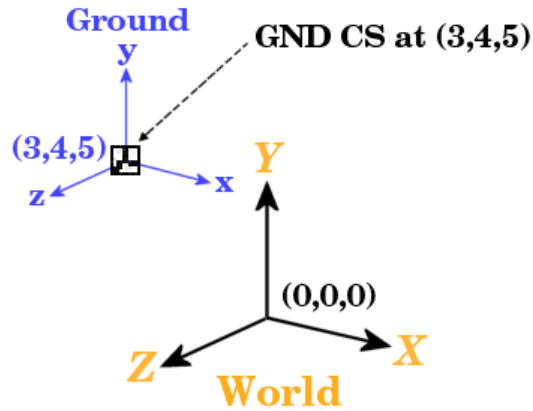
Caution Exactly one Ground block in each machine in your model must be connected to a Machine Environment block.

World and Grounded Coordinate Systems

The SimMechanics master coordinate system and reference frame is called World. All grounds are at rest in World. The connector port of each Ground block defines a grounded coordinate system called GND. The GND coordinate system's axes are parallel to World.

By default the origin of the grounded coordinate system coincides with the origin of the World coordinate system. The **Location** field of a Ground block's dialog allows you to move the origin of GND to some other point in the World coordinate system, as in the example "Creating a Simple Mechanical Model".

The GND coordinate system allows you to specify the positions and motions of parts of your machine relative to fixed points in the machine's surroundings. With a shifted origin, GND remains at rest.



Modeling Rigid Bodies

The SimMechanics Body block enables you to model rigid bodies of finite mass and extent. A body is rigid if its internal parts cannot move relative to one another.

About Body Blocks

A Body block allows you to specify the following attributes of a rigid body.

Mass Properties. These include the body's mass, which determines its response to translational forces, and its inertia tensor, which determines its response to rotational torques.

Body Coordinate Systems. By default a Body block defines three local coordinate systems, one associated with a body's center of gravity, labeled CG, and two others, labeled CS1 and CS2, respectively, associated with two other points on the body that you can specify. You can create additional Body coordinate systems or delete them as necessary.

A Body block's dialog box allows you to specify a Body CS's origin (see "Setting a Body CS's Position" on page 1-14) and orientation (see "Setting a Body CS's Orientation" on page 1-16). The origin and orientation of a body's CG CS specify the body's starting location and orientation. The origins of the other Body coordinate systems specify the initial locations of other points on the body.

The Body block allows flexibility in specifying the origins and orientations of Body coordinate systems. You can specify the origin and orientation of a body CS relative to

- The World CS
- Any other CS on the same body
- The *Adjoining CS*, the CS on the neighboring body or ground directly connected by a Joint, Constraint, or Driver to the selected Body CS you are configuring

This simplifies creation and maintenance of models. The only limitation is that you must specify the origin and location of at least one of a machine's Body coordinate systems relative to the World CS.

Home Configuration. Once you enter all the needed positions and orientations into the Bodies of your model, your machine is in its *home configuration*. The body velocities are zero, and any disassembled joints remain disassembled.

Connector Ports. Any Body CS can display a Body CS Port. A Body CS Port allows you to attach Joints, Actuators, and Sensors to a Body. By default, a Body's CS1 and CS2 coordinate systems each display a Body CS port. You can display a port for any other Body coordinate system as well, including a Body's CG CS.

Creating a Body Block

To create a Body block,

1. Drag a Body block icon from the SimMechanics Bodies Library and drop it into your model window.
2. Open the Body block's dialog box.
3. Enter the mass of the body you are modeling in the **Mass** field.
4. Select the units of mass from the adjacent units list.
5. Enter a 3-by-3 matrix representing the body's inertia tensor relative to its center of gravity coordinate system (CG CS) origin and axes in the **Inertia** field (see "Determining Inertia Tensors for Common Shapes" on page 1-13).
6. Enter the initial positions of the body's CG and coordinate systems in the **Position** tab.
7. Enter the initial orientation of the body's CG and coordinate systems in the **Orientation** tab.
8. Click **OK** or **Apply**.

Determining Inertia Tensors for Common Shapes

The following table enables you to determine the inertia tensors for some common shapes. For each shape of mass `m`, the table lists the shape's principal moments of inertia, I_1, I_2, and I_3, along the x-, y-, and z-axes of the shape's CG coordinate system.

Shape	I_1	I_2	I_3
Thin rod of length L aligned along z	$mL^2/12$	$mL^2/12$	0
Sphere of radius R	$2mR^2/5$	$2mR^2/5$	$2mR^2/5$
Cylinder of radius R and height h aligned along z	$(m/4)(R^2 + h^2/3)$	$(m/4)(R^2 + h^2/3)$	$mR^2/2$

Shape	I_1	I_2	I_3
Rectangular parallelopiped of sides a, b, and c aligned along x, y, z, respectively	$(m/12)(b^2 + c^2)$	$(m/12)(a^2 + c^2)$	$(m/12)(a^2 + b^2)$
Cone of base radius R and height h along z	$(m/4)(3R^2/5 + h^2)$	$(m/4)(3R^2/5 + h^2)$	$3mR^2/10$
Ellipsoid of semiaxes a, b, and c aligned along x, y, z, respectively	$(m/5)(b^2 + c^2)$	$(m/5)(a^2 + c^2)$	$(m/5)(a^2 + b^2)$

The corresponding inertia tensor for the shape is the following 3-by-3 matrix:

$$\begin{pmatrix} I_1 & 0 & 0 \\ 0 & I_2 & 0 \\ 0 & 0 & I_3 \end{pmatrix}$$

Working with Body Coordinate Systems

Every SimMechanics body has Body coordinate systems (CSs) attached to it. The location of a body CS is the origin of that CS. The CS's rectangular x-y-z coordinate axes are rotated at some orientation. You set up body CS origins and orientations before running your model. But once the bodies start to move, the origins and orientations of a body's CSs remain fixed in the body. The elements of a body's inertia tensor also remain fixed in the body. Consult "Representing Motion" for more about orienting bodies and body CSs.

The sections "Managing Body Coordinate Systems" on page 1-17 and "Creating Body CS Ports" on page 1-18 explain how to create custom Body coordinate systems and Body CS ports or delete existing ports.

Setting a Body CS's Position

The **Position** tab of a Body block's dialog box allows you to specify the position of any of a body's local coordinate systems.

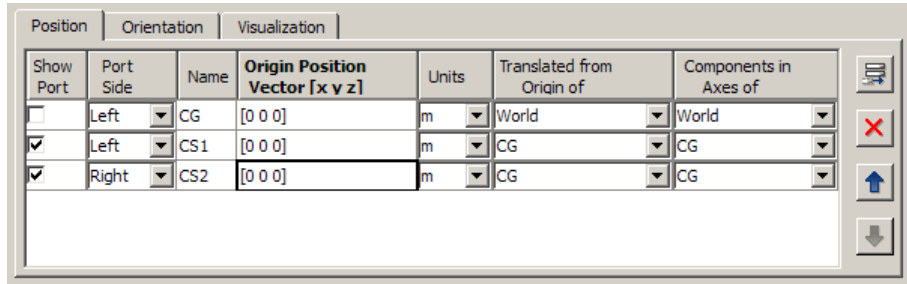
The **Translated from Origin of** and **Components in Axes of** lists in the tab together specify which other of your machine's coordinate systems you

use as reference points and orientations to set up the coordinate systems of the body you are configuring.

To specify the position of a Body CS,

1. Open the Body block's dialog box.

   The dialog box's **Position** tab lists the body's local coordinate systems in a table.

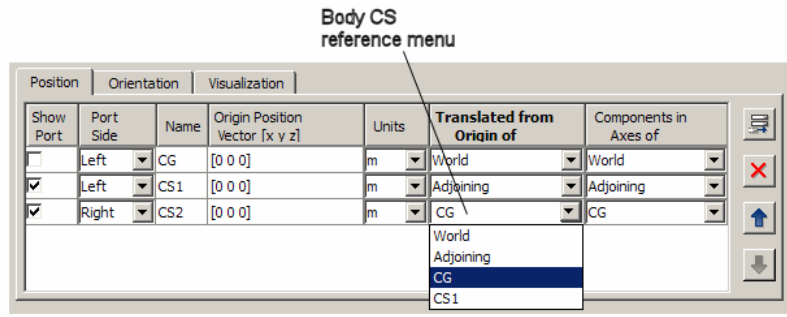


   Each row specifies the position of the coordinate system specified in the **Name** column.

2. Select the units in which you want to specify the origin of the Body CS from the CS's **Units** list.

3. Specify the reference coordinate systems for the Body CS, i.e., the coordinate systems relative to which you want to measure the Body CS origin and the orientation of the Body CS's coordinate axes. The choices are World, the adjoining CS, and other Body CSs on the same Body.

   You must directly or indirectly define all Body CSs by reference to a Ground or to World. Indirect reference means that you specify a Body CS relative to another CS and so on, in a chain of references that ultimately ends in a Ground or World.



You do this by selecting the origin and orientation of the specification CS from the Body CS's **Translated from Origin of** and **Components in Axes of** lists, respectively. For example, suppose that you want to specify the position of CS2 relative to another coordinate system, whose origin is at the origin of CS1 but whose axes run parallel to those of the CG CS. Then you would select CS1 from the **Translated from Origin of** list of CS2 and CG from the **Components in Axes of** list of CS2.

4 Enter a vector specifying the location of the Body CS in the **Origin Position Vector [x y z]** field of the CS.

   The components of the vector must be in the units that you selected and relative to the coordinate system that you selected. For example, suppose that you had selected `m` as the unit for specifying CS2's origin and CS1 and World as the CSs specifying the origin and orientation for CS2. Now suppose that you want to specify the location of CS2 as one meter to the right of CS1 along the World *x*-axis. Then you would enter `[1 0 0]` as CS2's position vector.

5 Click **Apply** to accept the position setting or **OK** to accept the setting and dismiss the dialog box.

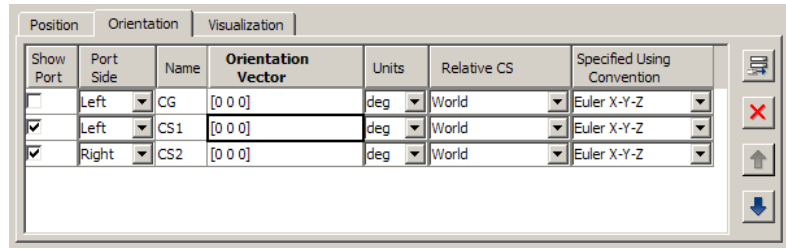
Setting a Body CS's Orientation

The **Orientation** tab of a Body block's dialog box allows you to specify the orientation of any of a body's local coordinate systems.

To specify the orientation of a Body CS,

1 Open the Body block's dialog box.

2 Select the dialog box's **Orientation** tab.



3 Select the units (degrees or radians) in which you want to specify the orientation of the CS from the CS's **Units** list.

4 Select the coordinate system relative to which you want to specify the orientation of the Body CS from the Body CS's **Relative CS** list. The choices are World, the adjoining CS, and other Body CSs on the same Body.

5 Select the convention you want to use to specify the orientation of the Body CS from the CS's **Specified Using Convention** list.

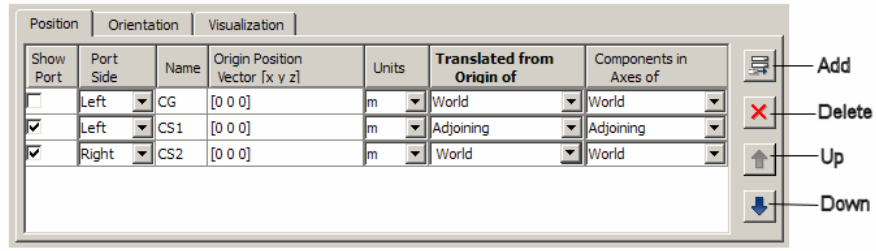
6 Enter a vector that specifies the orientation of the Body CS relative to the CS you choose for that purpose, according to the selected specification convention.

7 Click **Apply** to accept the orientation setting or **OK** to accept the setting and dismiss the dialog box.

Managing Body Coordinate Systems

You will often need to modify the default Body coordinate systems of a Body block. You might want to connect a Body to more than two Joints, in which case you need not only more Body CSs, but their Body CS ports as well. Connecting Actuators and Sensors to Bodies requires a Body CS and Body CS port for each connection.

The Body coordinate systems tab of a Body block's dialog box contains a row of buttons that allow you to add, delete, and reorder a Body's local coordinate systems.



To use these buttons, select a Body CS in the CS table and select

- **Delete** to remove the selected CS from the table
- **Up** to move the CS's entry one row up in the CS table
- **Down** to move the CS's entry one row down in the CS table

Select **Add** to add a new CS.

Creating Body CS Ports

To add or delete a port from a Body block's icon, open the block's dialog box and select or clear the CS's **Show Port** check box in the dialog box's Body CS table. Click **OK** or **Apply** to confirm the change.

Modeling Degrees of Freedom

In this section...

About Joints

A SimMechanics joint represents the degrees of freedom (DoF) that one body (the follower) has relative to another body (the base). The base body can be a movable rigid body or a ground. Unlike a physical joint, a SimMechanics joint has no mass, although some joints have spatial extension (see "Modeling Massless Connectors" on page 1-30).

A SimMechanics joint does not necessarily imply a physical connection between two bodies. For example, a SimMechanics Six-DoF joint allows the follower, e.g., an airplane, unconstrained movement relative to the base, e.g., ground, and does not require that the follower ever come into contact with the base.

SimMechanics joints only add degrees of freedom to a machine, because the Body blocks carry no degrees of freedom. Contrast this with physical joints, which both add DoFs (with axes of motion) and remove DoFs (by connecting bodies). See "Counting Model Degrees of Freedom" on page 1-89 for more on this point.

The SimMechanics Joints Library provides an extensive selection of blocks for modeling various types of joints. This section explains how to use these blocks.

Note A SimMechanics joint represents the relative degrees of freedom of one body relative to another body. Only if a joint is connected on one side to a ground and on the other to a body does the joint represent an absolute DoF of the body with respect to World.

Modeling Joints

Modeling with Joint blocks requires an understanding of the following key concepts:

- Joint primitives
- Joint types
- Joint axes
- Joint directionality
- Assembly restrictions

Joint Primitives

Each Joint block bundles together one or more joint primitives that together specify the degrees of freedom that a follower body has relative to the base body. The following table summarizes the joint primitives found singly or multiply in Joint blocks.

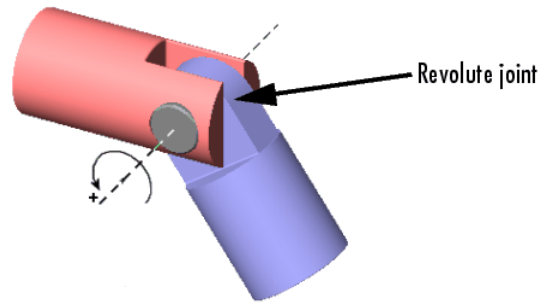
Primitive Type	Symbol	Degrees of Freedom
Prismatic	`P`	One degree of translational freedom along a prismatic axis
Revolute	`R`	One degree of rotational freedom about a revolute axis
Spherical	`S`	Three degrees of rotational freedom about a pivot point
Weld	`W`	Zero degrees of freedom

Joint Types

The blocks in the SimMechanics Joint Library fall into the following categories:

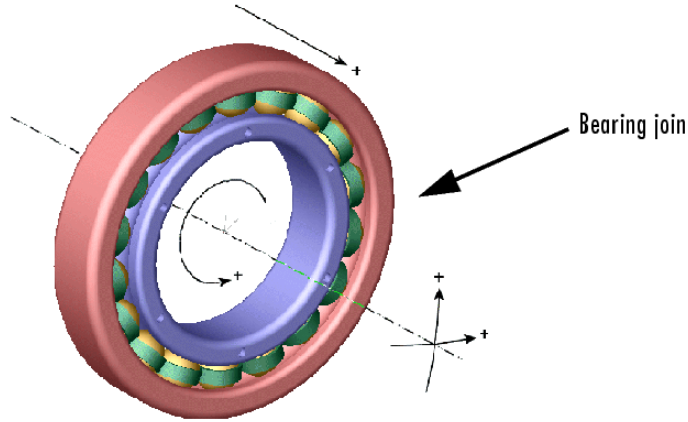
- Primitive joints

  Each of these blocks contains a single joint primitive. For example, the Revolute block contains a revolute joint primitive.



- Composite joints

  These blocks contain combinations of joint primitives, enabling you to specify multiple rotational and translational degrees of freedom of one body relative to another. Some model idealized real joints, for example, the Gimbal and Bearing joints.

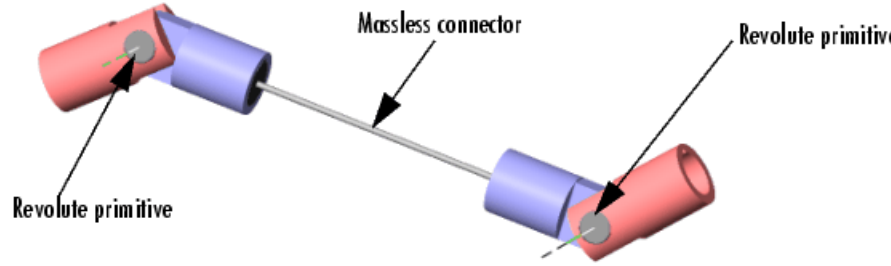


Others specify abstract combinations of degrees of freedom. For example, the Six-DoF block specifies unlimited motion of the follower relative to the base.

The Custom Joint allows you to create joints with any desired combination of rotational and translational degrees of freedom, in any order. The prefabricated composite Joints of the Joints library have the type and order of their primitives fixed. See "Axis Order" on page 1-24.

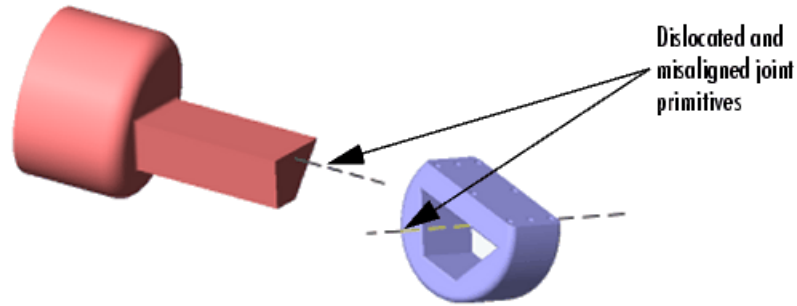
- Massless connectors

  These blocks represent extended joints with spatially separated joint primitive axes, for example, a Revolute-Revolute Massless Connector.



- Disassembled joints

These blocks represent joints not assembled until simulation starts — for example, a Disassembled Prismatic.



See "Assembly Restrictions" on page 1-26 and "Modeling Disassembled Joints" on page 1-34.

Joint Axes

Joint blocks define one or more axes of translation or rotation along which or around which a follower block can move in relation to the base block. The axes of a Joint block are the axes defined by its component primitives:

- A prismatic primitive defines an axis of translation.
- A revolute primitive defines an axis of revolution.
- A spherical primitive defines a pivot point for axis-angle rotation.

For example, a Planar Joint block combines two prismatic axes and hence defines two axes of translation.

Axis Direction. By default the axes of prismatic and revolute primitives point in the same direction as the *z*-axis of the World coordinate system (CS). A Joint block's dialog box allows you to point its prismatic and revolute axes in any other direction (see "Directing Joint Axes" on page 1-28).

Axis Order. Composite SimMechanics Joints execute their motion one joint primitive at a time. A joint that defines more than one axis of motion also defines the order in which the follower body moves along each axis or about a pivot. The order in which the axes and/or pivot appear in the Joint block's dialog box is the order in which the follower body moves.

Different primitive execution orders are physically equivalent, unless the joint includes one spherical or three revolute primitives. Pure translations and pure two-dimensional rotations are independent of primitive ordering.

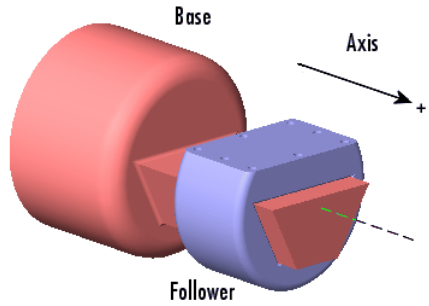
Axis Span. The *span* of the primitive axes is the complete space spanned by their combination. For example, one primitive axis defines a line, and two primitive axes define a plane.

Joint Directionality

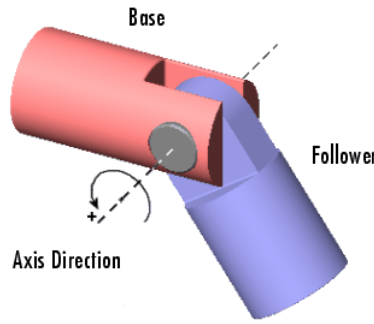
Directionality is a property of a joint that determines the dependence of the joint on the sign of forces or torques applied to it. A joint's directionality also determines the sign of signals output by sensors attached to the joint. Every SimMechanics joint in your model has a directionality. You must be able to determine the directionality of a joint in order to actuate it correctly and to interpret the output of sensors attached to it.

A joint's follower moves relative to the joint's base. The joint's directionality takes into account the joint type and the direction of the joint's axis, as follows.

Directionality of a Prismatic Joint. If the joint is prismatic, a positive force applied to the joint moves the follower in the positive direction along the axis of translation. A sensor attached to the joint outputs a positive signal if the follower moves in a positive direction along the joint's axis of translation relative to the base.



Directionality of a Revolute Joint. If the joint is revolute, a positive torque applied to the joint rotates the follower by a positive angle around the joint's axis of rotation, as determined by the right-hand rule. A sensor attached to the revolute joint outputs a positive signal if the follower rotates by a positive angle around the joint's axis of revolution, as determined by the right-hand rule.



Directionality of a Spherical Joint. Spherical joint directionality means the positive sense of rotation of the three rotational DoFs. Pick a rotation axis, rotating using the right-hand rule from the base Body CS axes. Then rotate the follower Body about that axis in the right-handed sense.

Directionality and Ordering of Composite Joint Primitives. Each joint primitive separately has its own directionality, based on the primitive's type and the direction of its axis of translation or rotation. In each case, the follower body of the composite joint moves along or around the joint primitive's axis relative to the base body.

The order of primitives in the composite Joint's dialog determines the spatial construction of the joint.

The first listed primitive is attached to the base, the second to the first, and so on, down to the follower, which is attached to the last primitive.

- Moving the first listed primitive moves the subsequent primitives in the list, including the follower, relative to the base.
- Moving any primitive moves the primitives below it in the list (but not those above it), as well as the follower.
- Moving the last listed primitive moves only the follower.

Changing the Directionality of a Joint. You can change the directionality of a joint by either

- Reversing and reconnecting the Joint block to reverse the roles of the base and follower bodies.
- Reversing the sign (direction) of the joint axis.

Assembly Restrictions

Many joints impose one or more restrictions, called *assembly restrictions*, on the placement of the bodies that they join. The conjoined bodies must satisfy these restrictions at the beginning of simulation and thereafter within assembly tolerances that you can specify (see "Controlling Machine Assembly" on page 2-12). For example, the Body CSs attached to revolute and spherical joints must coincide within assembly tolerances; the Body CSs attached to a Prismatic joint must lie on the prismatic axis within assembly tolerances; the

Body CSs attached to a Planar joint must be coplanar with Planar primitives, etc. Composite joints, e.g., the Six-DoF joint, impose assembly restrictions equal to the most restrictive of its joint primitives. See the block reference for each Joint for information on the assembly restrictions, if any, that it imposes.

Positioning bodies so that they satisfy a joint's assembly restrictions is called *assembling* the joint.

All SimMechanics Joints except blocks in the Disassembled Joints sublibrary require manual assembly. Manual assembly entails your setting the initial positions of conjoined bodies to valid locations (see "Assembling Joints" on page 1-29). The simulation assembles disassembled joints during the model initialization phase. It assumes that you have already assembled all other joints before the start of simulation. Hence joints that require manual assembly are called *assembled* joints. During model initialization and at each time step, the simulation also checks to ensure that your model's bodies satisfy all assembly restrictions. If any of your model bodies fails to satisfy assembly restrictions, the simulation stops and displays an error message.

Creating a Joint

A joint must connect exactly two bodies. To create a joint between two bodies:

1. Select the Joint from the SimMechanics Joints library that best represents the degrees of freedom of the follower body relative to the base body.

2. Connect the base connector port of the Joint block (labeled B) to the Body CS origin on the base block that serves as the point of reference for specifying the degrees of freedom of the follower block.

3. Connect the follower connector port of the Joint block (labeled F) to the Body CS origin on the follower block that serves as the point of reference for specifying the degrees of freedom of the base block.

4. Specify the directions of the joint's axes (see "Directing Joint Axes" on page 1-28).

5. If you plan to attach Sensors or Actuators to the Joint, create an additional port for each Sensor and Actuator (see "Creating Actuator and Sensor Ports on a Joint" on page 1-28).

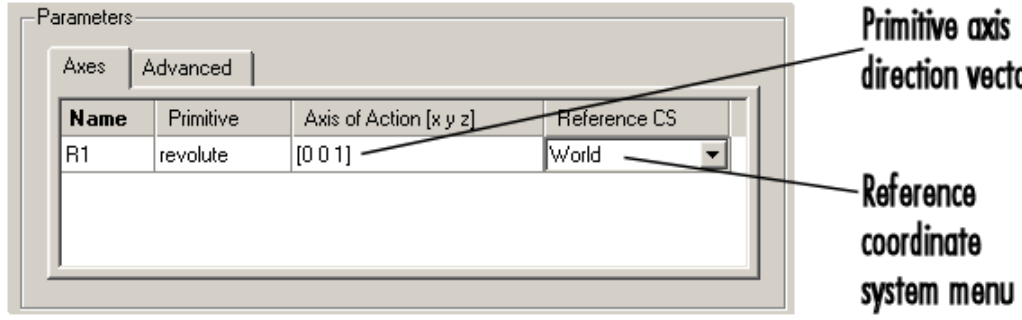
6 If the joint is an assembled joint, assemble the bodies joined by the joint (see "Assembling Joints" on page 1-29).

Directing Joint Axes

By default the prismatic and revolute axes of a joint point in the same direction as the *z*-axis of the World coordinate system. To change the direction of the axis of a joint primitive:

1 Open the joint's dialog box and select a reference coordinate system for specifying the axis direction from the coordinate system list associated with the axis primitive.

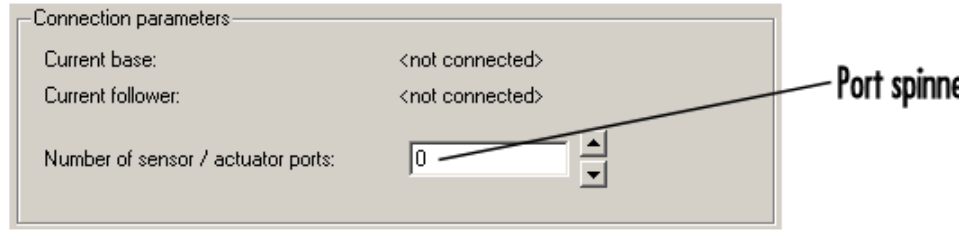
The options are the World coordinate system or the local coordinate systems of the base or follower attachment point. Choose the coordinate system that is most convenient.



2 Enter in the primitive's axis direction field a vector that points in the desired direction of the axis in the selected coordinate system.

Creating Actuator and Sensor Ports on a Joint

To create additional connector ports on a Joint for Actuators and Sensors, open the Joint's dialog box and set the **Number of sensor/actuator ports** to the number of Actuators and Sensors you plan to attach to the Joint.

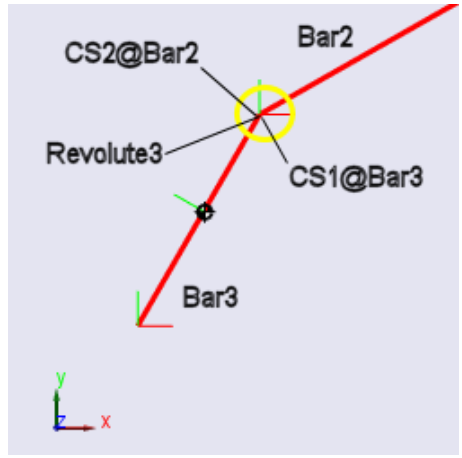


Apply the setting by clicking **OK** or **Apply**.

Assembling Joints

You must manually assemble all assembled joints in your model. Assembling a joint requires setting the initial positions of its attached base and follower Body CSs such that they satisfy the assembly restrictions imposed by the joint (see "Assembly Restrictions" on page 1-26). Consider, for example, the model discussed in "Creating a Closed-Loop Mechanical Model".

This model comprises three bars connected by revolute joints to each other and to two ground points. The model collocates the CS origins of the Body CS ports connected to each Joint, thereby satisfying the assembly restrictions imposed by the revolute joints.

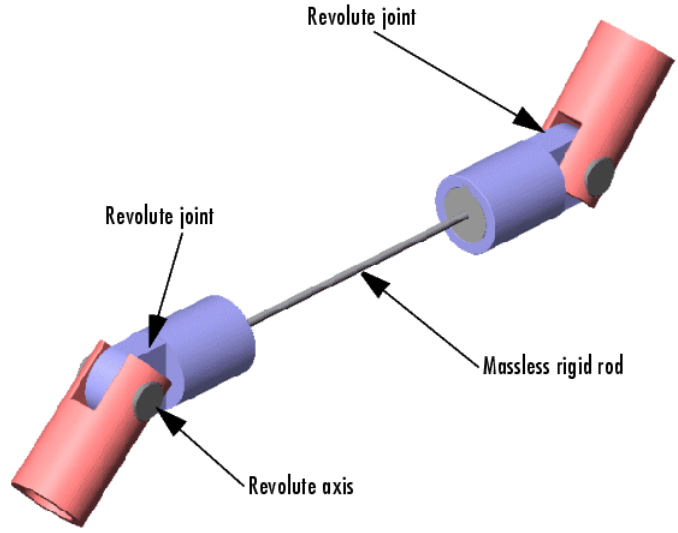


Assembled Revolute Joint in the Four Bar Mechanism

Modeling Massless Connectors

Massless connectors simplify the modeling of machines that use a relatively light body to connect two relatively massive bodies. For example, you could use a Body block to model such a connector. But the resulting equations of motion might be ill-conditioned, because that connecting body's mass is small, and the simulation can be slow or prone to failure. A massless connector also avoids global inconsistencies that can arise if you use a Constraint block to model the connector.

A massless connector consists of a pair of joints located a fixed distance apart. Think of a massless connector as a massless rod with a joint primitive affixed at each end.



The initial orientation and length of the massless connector are defined by a vector drawn from the base attachment point to the follower attachment point. During simulation, the orientation of the massless connector can change but not its length. In other words, the massless connector preserves the initial separation of the base and follower bodies during machine motion.

Note You cannot actuate or sense a massless connector.

The SimMechanics Joints/Massless Connectors sublibrary contains these Massless Connectors:

- Two revolute primitives (Revolute-Revolute)
- A revolute primitive and a spherical primitive (Revolute-Spherical)
- Two spherical primitives (Spherical-Spherical)

Creating a Massless Connector

To create a massless connector between two bodies:

1. Drag an instance of a Massless Connector block from the Massless Connectors sublibrary into your model and connect it to the base and follower blocks.

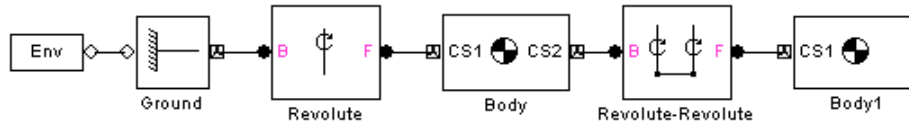
   You can set the direction of the axes of revolute primitives. If necessary, point the axes of the connector's revolute joints in the direction required by the dynamics of the machine you are modeling.

2. Assemble the connector by setting the initial positions of the base and follower body attachment points to the initial positions required by your machine's structure.

During simulation, the massless connector maintains the initial separation between the bodies though not necessarily the initial relative orientation.

Massless Connector Example: Triple Pendulum

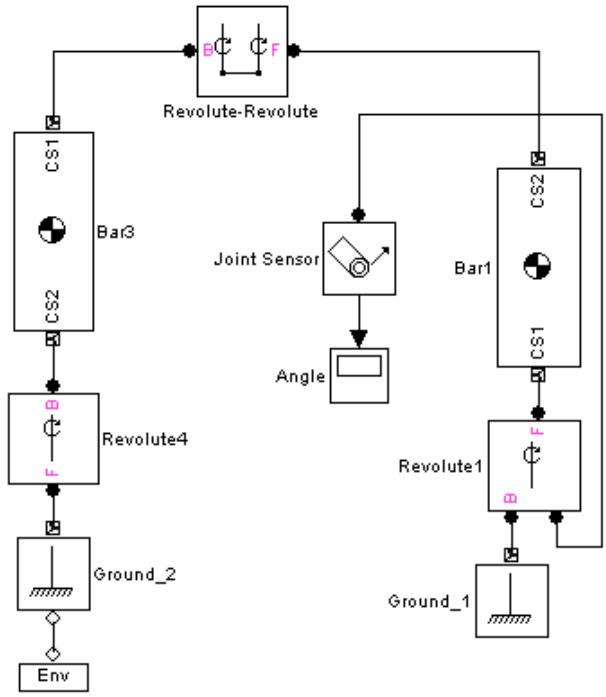
Consider a triple pendulum comprising massive upper and lower bodies and a middle body of negligible mass. The following model uses a Revolute-Revolute massless connector to model such a pendulum.



In this model, the joint axes of the Revolute-Revolute connector have their default orientation along the World *z*-axis. As a result, the lower arm (Body1) rotates parallel to the World's *x*-*y* plane.

Massless Connector Example: Four Bar Mechanism

The following model replaces one of the bars (Bar2) in the `mech_four_bar` model from the Demos library with a Revolute-Revolute massless connector.

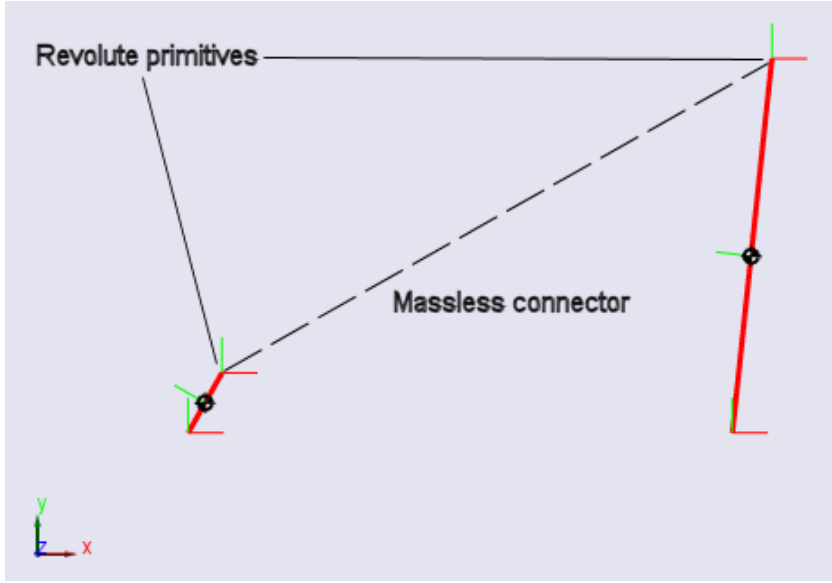


This model changes the Body CS origins of Bar3 to the following values.

Name	Origin position vector	Translated from origin of
CG	[-0.027 -0.048 0]	CS1
CS1	[0.054 0.096 0]	CS2
CS2	[0 0 0]	ADJOINING (Ground_2)

This creates a separation between Bar3 and Bar1 equal to the length of Bar2 in the original model.

Try simulating both the original and the modified model. Notice that the massless connector version moves differently, because you eliminated the mass of Bar2 from the model. Notice also that the massless bar does not appear in the animation of the massless connector version of the model.



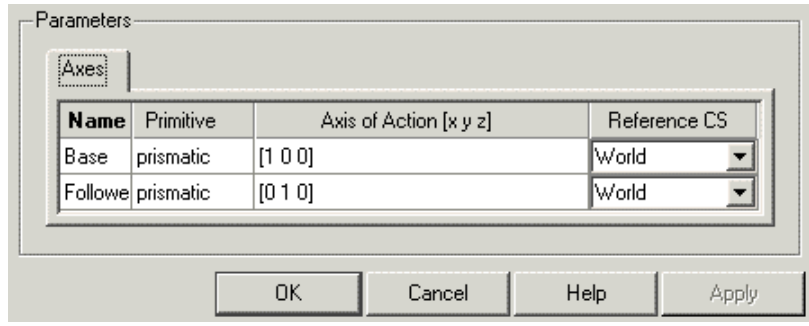
Modeling Disassembled Joints

The SimMechanics Joints/Disassembled Joints sublibrary contains a set of joints automatically assembled at the start of simulation; that is, the simulation positions the joints such that they satisfy the assembly restrictions imposed by the type of joint, e.g., prismatic or revolute. Using these joints eliminates the need for you to assemble the joints yourself.

Disassembled joints differ from assembled joints in significant ways. An assembled joint primitive has only one axis of translation or revolution or one spherical pivot point. A disassembled prismatic or revolute primitive has two axes of translation or rotation, one for the base and one for the follower body. A disassembled spherical primitive similarly has two pivot points.

Caution Disassembled joints can appear only in closed loops. Each closed loop can contain at most one disassembled joint.

The dialog box for a disassembled joint allows you to specify the direction of each axis.



During model assembly, the simulation determines a common axis of revolution or translation that satisfies model assembly restrictions, and aligns the base and follower axes along the common axis.

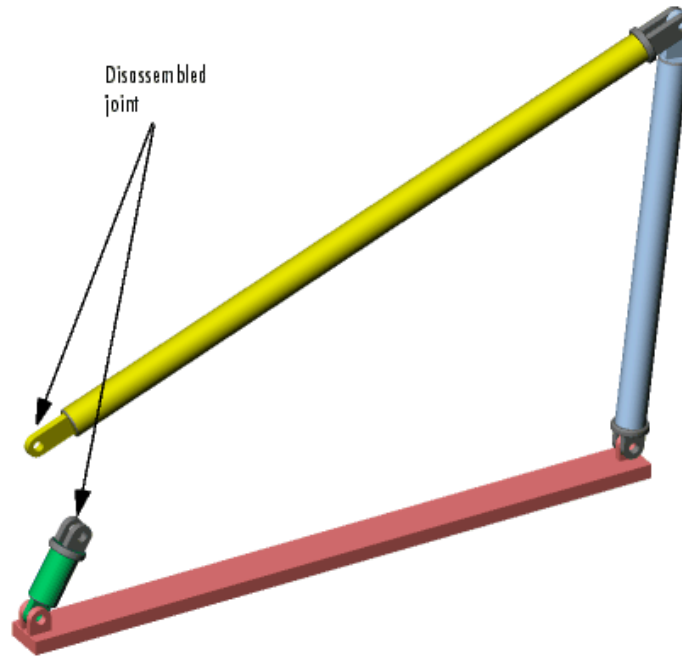
Controlling Automatic Assembly and the Assembled Configuration

If your machine contains Joint Initial Condition Actuator (JICA) blocks, the machine is moved from its home to its *initial configuration* by applying the initial condition information to the machine's joints first. Then any disassembled joints are assembled, leading to the *assembled configuration*.

During model assembly, the simulation might move bodies connected by assembled joints from their initial positions in order to assemble the disassembled joints. The SimMechanics solution to the assembly problem cannot be predicted beforehand, except in simple cases. If you do not want bodies to move during model assembly, use JICA blocks to specify the initial positions of bodies whose positions you want to remain fixed during the assembly process. The resulting assembly will satisfy the initial conditions specified by the JICA blocks.

Disassembled Joint Example: Four Bar Mechanism

This example creates and runs a model of a disassembled four bar machine.



Refer to the tutorial, "Creating a Closed-Loop Mechanical Model", and the `mech_four_bar` demo:

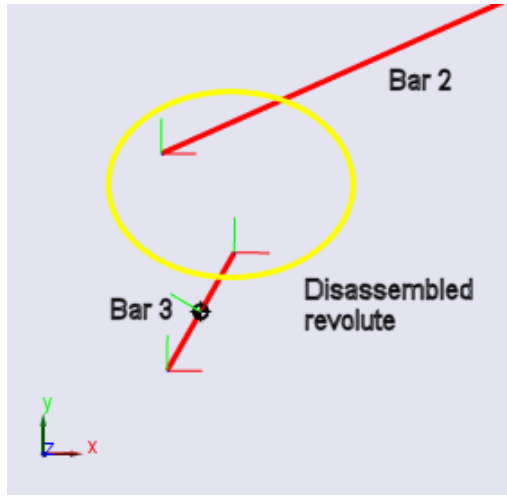
1. Disconnect the Joint Sensor1 block from the Revolute3 block.
2. Replace Revolute3 with a Disassembled Revolute block from the Joints/Disassembled Joints sublibrary.
3. Open the Disassembled Revolute dialog box and, under **Axis of Action** for both `Base` and `Follower` axes, enter `[0 0 1]`. Close the dialog.
4. Open the Bar2 dialog box and dislocate the joint by displacing Bar2's CS2 origin from Bar 3's CS1 origin.

Do this by entering a nonzero vector under **Origin Position Vector [x y z]** for CS2, then changing the **Translated from Origin of** pull-down entry to `ADJOINING`. CS1 on Bar3 is the Adjoining CS of CS2 of Bar2. Close the dialog.

5 To avoid circular CS referencing, you must check the Bar3 dialog entry for CS1 on Bar3. Be sure that CS1 on Bar3 does *not* reference CS2 on Bar2. Reference it instead to CS2 on Bar3, which adjoins Ground_2.

6 Rerun the model.

Note that the motion is different from the manually assembled case.

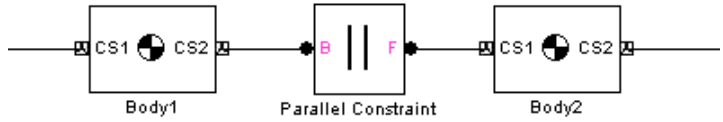


Constraining and Driving Degrees of Freedom

In this section...

About Constraints

The SimMechanics Constraints & Drivers Library provides a set of blocks to model constraints on the relative motions of two bodies. You model the constraint by connecting the appropriate Constraint or Driver block between the two bodies. As with joints, the blocks each have a base and follower connector port, with the body connected to the follower port viewed as moving relative to the body connected to the base port. For example, the following model constrains Body2 to move along a track that is parallel to the track of Body1.



Types of Mechanical Constraints

Constraint and Driver blocks enable you to model time-independent constraints or time-dependent drivers.

- Constraint and unactuated Driver blocks model *scleronomic* (time-independent) constraints.
- Actuated Driver blocks (see "Actuating a Driver" on page 1-62) model *rheonomic* (time-dependent) constraints.

Scleronomic constraints lack explicit time dependence; that is, their time dependence appears only implicitly through the coordinates $\boldsymbol{x}$. Rheonomic constraints have explicit time dependence as well, in addition to implicit time dependence through the $\boldsymbol{x}$.

Holonomic constraint functions depend only on body positions, not velocities:

$$f(\boldsymbol{x}_B, \boldsymbol{x}_F; t) = 0$$

Constraints of the form

$$g(\boldsymbol{x}_B, \dot{\boldsymbol{x}}_B, \boldsymbol{x}_F, \dot{\boldsymbol{x}}_F; t) = 0$$

can sometimes be integrated into a form dependent only on positions; but if not, they are *nonholonomic*. For example,

- The one-dimensional rolling of a wheel of radius R along a line (the x-axis) imposes a holonomic constraint, $x = R\theta$.
- The two-dimensional rolling of a sphere of radius R on a plane (the xy-plane) imposes a nonholonomic constraint, $ds = R \cdot d\theta$, with $ds^2 = dx^2 + dy^2$. This constraint is nonholonomic because there is not enough information to solve the constraint independently of the dynamics.

What Constraints and Drivers Do

Constrained and driven bodies are still free to respond to externally imposed forces/torques, but only in a way consistent with the constraints.

Constraints and drivers can only remove degrees of freedom from a machine. Constraints and unactuated Drivers prevent the machine from moving in certain ways. Unactuated Drivers hold the constrained degrees of freedom between the connected pair of bodies in their initial state. Actuated Drivers externally impose a relative motion between pairs of bodies, starting with the bodies' initial state. See "Counting Model Degrees of Freedom" on page 1-89.

This section discusses modeling constraints and drivers in a general way.

The section ends with two examples, "Constraint Example: Gear Constraint" on page 1-41 and "Driver Example: Angle Driver" on page 1-43.

See the reference pages for information on the specific constraint that a Constraint or Driver block imposes.

Directionality of Constraints and Drivers

Like joints, constraints and drivers have directionality. The sequence of base to follower body determines the directionality of the constraint or driver. The directionality determines how the sign of Driver Actuator signals affects the motion of the follower relative to the base and the sign of signals output by constraint and driver sensors.

Solving Constraints

A SimMechanics simulation uses a constraint solver to find the motion, given the model's Constraint and Driver blocks. You can specify both the constraint solver type and the constraint tolerances used to find the constraint solution. See "Maintaining Constraints" on page 2-12 for more information.

Mitigating Constraint Singularities

Some constraints, whether time-independent (Constraints) or time-dependent (Drivers), can become singular when the constrained bodies take on certain relative configurations; for example, if the two body axes line up when the Bodies are connected by an Angle Driver. The simulation slows down as a constraint becomes singular.

If you find a constrained model running slowly, consider selecting the **Use robust singularity handling** option in the **Constraints** tab of your machine's Machine Environment block dialog. See "Handling Motion Singularities" on page 2-18.

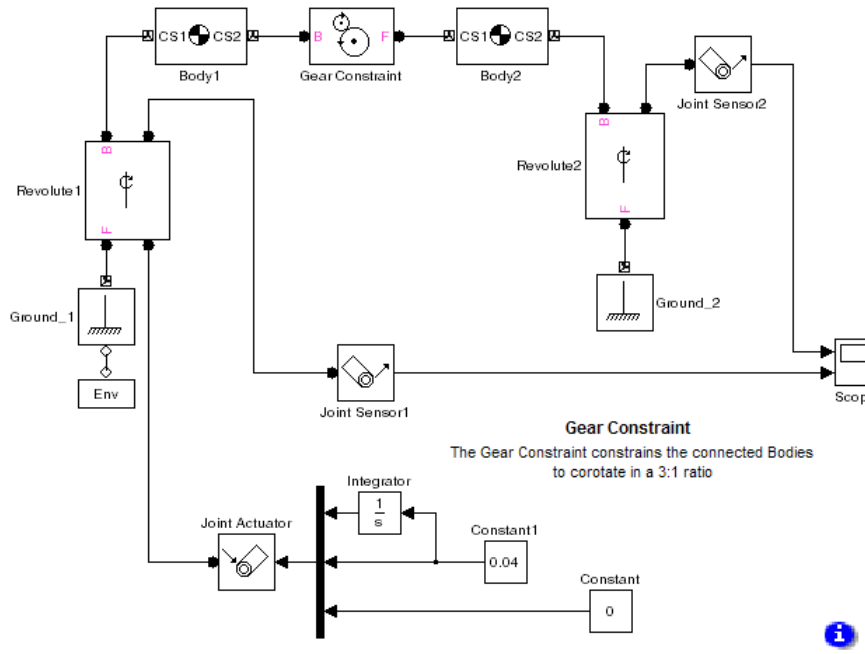
Restrictions on Using Constraint and Driver Blocks

The following restrictions apply to the use of Constraint and Driver blocks in a model:

- Constraint and Driver blocks can appear only in closed loops. A closed loop cannot contain more than one Constraint or Driver block.
- A Constraint or Driver must connect exactly two Bodies.

Constraint Example: Gear Constraint

The `mech_gears` model illustrates the Gear Constraint. Open the Body and Gear Constraint blocks.



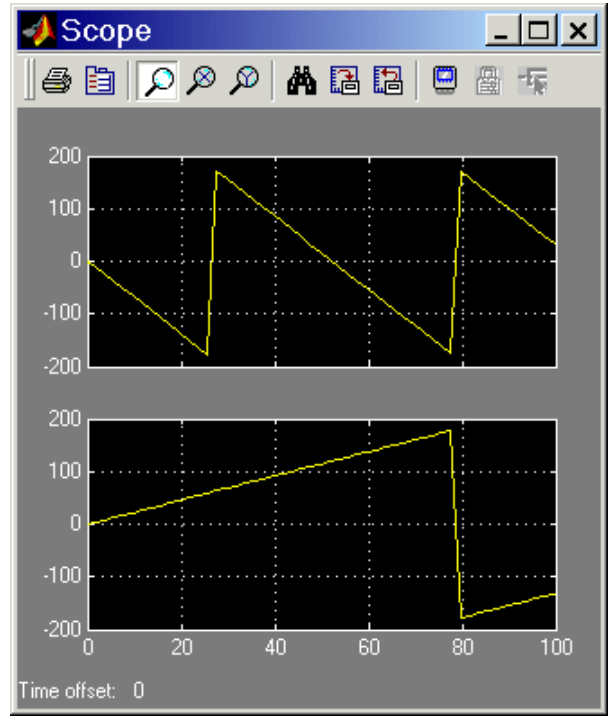
Body1 and Body2 have their CG positions 2 meters apart. CS1 and CS2 on Body1 are collocated with the Body1 CG, and similarly, CS1 and CS2 on Body2 are collocated with the Body2 CG.

The Gear Constraint between them has two pitch circles. One is centered on the CS2 at the base Body, which is Body1, and has radius 1.5 meters. The other is centered on CS1 at the follower Body, which is Body2, and has radius 0.5 meters. The distance between CS2 on Body1 and CS1 on Body2 is 2 meters. The sum of the pitch circle radii equals this distance, as it must.

Visualizing the Gear Motion

The model is set up to open the visualization window automatically upon simulation start, with convex hulls, as explained in the *SimMechanics Visualization and Import Guide*. Start the simulation and watch the CG CS axis triads spin around. The CG triad at Body2 rotates three times faster than the CG triad at Body1, because the pitch circle centered on Body2 is three times smaller.

You can see the same behavior in the Scope. The upper plot shows the motion of Revolute2, and the lower plot the motion of Revolute1. Note that angular motion is mapped to the interval $(-180^o, +180^o]$ degrees.



The Gear Constraint is inside a closed loop formed by

Ground_1–Revolute1–Body1–Gear Constraint–Body–Revolute2–Ground_2

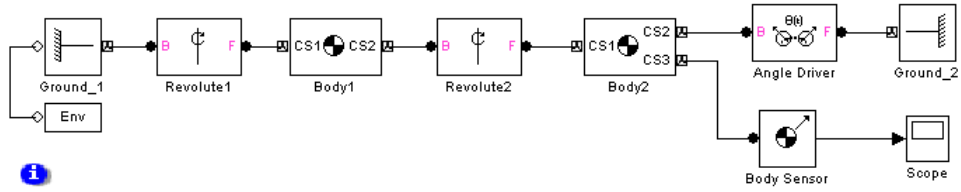
Although Ground_1 and Ground_2 are distinct blocks, they represent different points on the same immobile ground at rest in World. So the blocks form a loop.

Driver Example: Angle Driver

The following two models illustrate the Angle Driver, both without and with a Driver Actuator.

The Angle Driver Without a Driver Actuator

The first is `mech_angle_unact`. Open the Body2 block.



The bodies form a double pendulum of two rods. The Body Sensor is connected to Body2 at CS3 = CS2 and measures all three components of Body2's angular velocity vector with respect to the ground.

The Angle Driver is connected between Body2 and Ground_2. Because the Angle Driver is not actuated in this model, it acts during the simulation as a time-independent constraint to hold the angle between Body2 and Ground_2 constant at its initial value.

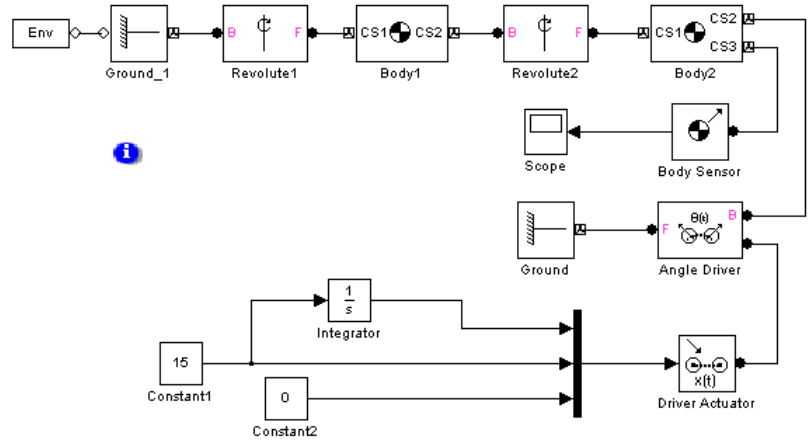
Visualizing the Angle Driver Motion

The model is set up to open the visualization window automatically upon simulation start, with convex hulls, as explained in the *SimMechanics Visualization and Import Guide*.

Start the simulation. The upper body swings like a pendulum, but the lower body maintains its horizontal orientation with respect to the horizontal ground. The Scope measures Body2's angular velocity with respect to ground, and this remains at zero.

The Angle Driver With a Driver Actuator

The second model is `mech_angle_act`. Open the Driver Actuator block.



The Driver Actuator drives the Angle Driver block. Here, the Actuator accepts a constant angular velocity signal from the Simulink blocks. The Actuator also requires the angle itself and the angular acceleration, together with the angular velocity, in a vector signal format. The Angle Driver's angle signal is added to the angle's initial value.

The Body Sensor again measures three components of Body2's angular velocity with respect to the ground. Constant1 drives the angle at 15°/second. While the simulation is running, this angle changes at the constant rate. At the same time, the assembly and the constant length of the two pendulum rods must be maintained by Simulink, while both rods are subject to gravity. As the two axes line up, the mutual constraint between the bodies enforced the Driver becomes singular. The simulation slows down.

As in the Gear Constraint model, the two Ground blocks in these models represent points on the same immobile ground at rest in World, so the Angle Driver is part of a closed loop.

Cutting Machine Diagram Loops

In this section...

Rules for Valid Machine Diagram Loops

In a SimMechanics model, you form a closed loop by the closure of SimMechanics blocks, of any type, on themselves. From a starting point, you can trace a path around a closed loop back to the starting point with no jumps or cuts. A closed loop is valid if it contains:

- At least one Joint block
- No more than one Disassembled Joint block
- No more than one Constraint or Driver block

To simulate a model containing closed loops, the SimMechanics simulation internally converts a closed-loop model to an open-topology tree model. This is accomplished by internally cutting each of the model's closed loops once, at a joint, constraint, or driver block, then replacing each cut by an additional internal constraint.

Rules for Automatic Loop Cutting

A SimMechanics simulation follows these loop-cutting rules.

- If a loop contains a constraint, driver, or disassembled joint, the simulation cuts the loop at one of those blocks. Selecting a preferred cut joint has no effect.

- If the loop does not contain a constraint, driver, or disassembled joint, the simulation cuts the loop at the preferred cut joint if you have specified one.
- If the loop does not contain a constraint, driver, or disassembled joint, and you have not specified a preferred cut joint, the simulation cuts the loop at the joint with the most degrees of freedom.

Note A SimMechanics simulation cuts a closed loop at a Disassembled Joint, Constraint, or Driver block, if one or more of these blocks is present, regardless of other Joints also present in the loop or of your preferred cut choice.

Specifying a Loop Joint for Cutting

You can specify a joint to cut if the loop does not contain a disassembled joint, constraint, or driver. Open the joint's dialog box and select the **Mark as the preferred cut joint** check box on the **Advanced** tab in that joint's dialog **Parameters** area.

Displaying the Cut Joints

To display automatically cut joints in your model, select the **Mark automatically cut joints** check box in the **Diagnostics** area of the **SimMechanics** node of your model's Configuration Parameters dialog. See "Configuring SimMechanics Simulation Diagnostics" on page 2-18.

For More About Disassembled and Cut Joints

Refer to "Modeling Disassembled Joints" on page 1-34 for more on disassembled joints. Consult "Verifying Model Topology" on page 1-85 to learn more about closed loop analysis.

For More About Constraints and Drivers

A SimMechanics simulation represents a cut Joint, Constraint, or Driver as an additional internal constraint. See "Constraining and Driving Degrees of Freedom" on page 1-38 for more about these specialized blocks.

Applying Motions and Forces

In this section...

About Actuators

The SimMechanics Actuators & Sensors Library provides a set of Actuator blocks that enable you to apply time-dependent forces and motions to bodies, joints, and drivers. You can also vary a body's mass and inertia tensor.

Caution You can connect an Actuator to a Ground. But an error results if you attempt to simulate or update a model containing such a connection. This is because ground is immobile and cannot be actuated.

You can use Actuator blocks to perform the following tasks:

- Apply a time-varying force or torque to a body or joint.
- Specify the position, velocity, and acceleration of a joint or driver as a function of time.
- Specify the initial position and velocity of a joint primitive.
- Specify the mass and/or inertia tensor of a body as a function of time.

In general, actuators can apply any combination of forces and motions to a machine provided that

- The applied forces and motions are consistent with each other and with the machine's geometry, constraints, and assembly restrictions.

- The actuation signals representing these forces and motions remain consistent when differentiated or integrated. See "Stabilizing Numerical Derivatives in Actuator Signals" on page 1-49.
- It is possible to find a unique solution for the motion of each actuated degree of freedom (DoF).

Stabilizing Numerical Derivatives in Actuator Signals

To actuate a physical system modeled by blocks, you often need to differentiate an incoming Simulink actuation signal.

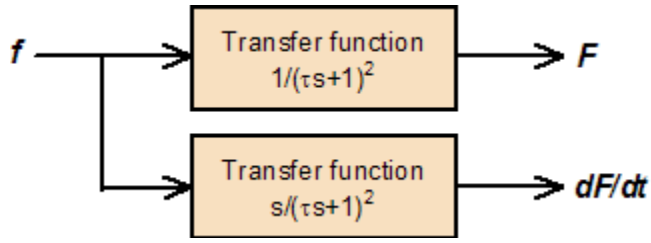
Simulink provides a Derivative block for numerical differentiation of a signal. However, this block's output is sometimes not stable or accurate enough for Physical Modeling purposes. Recommended alternatives to the Derivative block include the following.

Integrating Higher Derivative Signals. Start by specifying the highest derivative signal (such as an acceleration), then integrate this signal to obtain lower derivative signals (such as a velocity) using the Integrator block.

Transforming Signals with Transfer Functions. To differentiate a signal, use a transfer function block (Transfer Fcn). This block actually performs a Laplace transform convolution to smooth the output, which is not exactly the derivative.

You can eliminate this drawback by filtering the original signal f, then defining exact derivatives dF/dt, etc., of the filtered signal F by adding higher orders to the transfer function numerator. The order of the denominator should be equal to or greater than the number of output signals. Use the filtered signal F (instead of f), as well as the filtered derivatives.

In this example, the constant τ represents a smoothing time. The transfer functions define a filtered signal and its first derivative, two signals in all. Therefore, the transfer function denominator should be second order or higher.



Examples of Numerical Derivatives of Actuator Signals

A SimMechanics example requiring numerical derivatives is motion actuation of a joint, which requires position, velocity, and acceleration of each joint primitive as a function of time. You specify this information as a set of Simulink signals, which you can stabilize with one of the previous methods.

The transfer function method is illustrated by the `mech_stewart_control` model. For an example of the derivative-integration method, see the `mech_body_driver` model.

Actuating a Body

You can use the Body Actuator to apply forces and/or torques, but not motions, to bodies. (You can apply motions to a body indirectly, using Joint Actuators. See "Applying Motions to Bodies" on page 1-52.)

To actuate a body,

1. If there is not already an unused connector port available for the Actuator create a Body CS port on the Body for the Actuator. See the Body block reference if you need to learn how.
2. Drag a Body Actuator block from the Sensors & Actuators library into your model and connect its output port to a Body CS port on the Body.
3. Open the Actuator's dialog box.
4. Choose to apply a force or torque to the body:
   - Select the **Applied force** check box if you want to apply a force to the body, and select the units of force from the adjacent list.

- Select the **Applied torque** check box if you want to apply a torque to the body, and select the units of torque from the adjacent list.

5 Select the coordinate system used to specify the applied torque from the **With respect to CS** list.

The list allows you to choose either the World CS or the Body CS of the port to which you attached the Actuator.

6 Create vector signals that specify the value of the applied torque and force at each time step.

You can use any Simulink source block (for example, an Input port block or a Sine Wave block) or combination of Simulink blocks to create the Body Actuator signal. You can also use the output of a Sensor block connected to the Body as input to the Actuator, thereby creating a feedback loop. Such loops are useful for modeling springs and dampers (see "Validating Mechanical Models" on page 1-85).

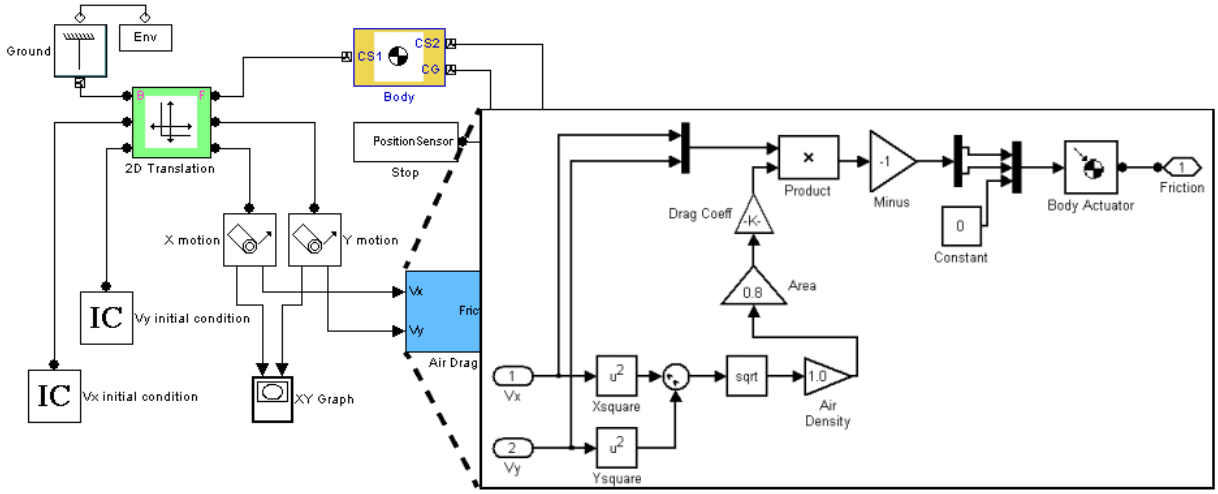
7 Connect the force and/or torque signal to the input port of the Actuator.

If you are applying both a force and a torque to the body, connect the force and torque signals to the inputs of a two-input Mux block. Then connect the output of the Mux block to the input of the Actuator.

Body Actuator Example: Pure Kinetic Friction

The `mech_ballistic_kin_fric` model in the Demos library provides an example of how to implement pure kinetic friction. This type of friction is a continuous force that depends on a body's motion relative to a medium (such as air), as well as on physical characteristics of the body. Kinetic friction, unlike "stiction," involves no "sticking" or locking of motion, and the friction is not discontinuous. While you could use the Joint Stiction Actuator, this is not necessary. This model applies air friction or drag to a projectile with a Body Actuator.

Open the Air Drag subsystem. If you double-click the block, a mask dialog box opens asking for the drag coefficient `Cd`. If you right-click the block and select **Look under mask**, the subsystem itself appears:



The Air Drag subsystem computes the air friction according to a standard air friction model. (See the Aerospace Blockset documentation for more information.) The drag always opposes the projectile's motion and is proportional to the product of the air density, the projectile's cross-sectional area, and the square of its speed.

Run the model with the default drag coefficient (zero). The XY Graph window opens to plot the parabolic path of the projectile. Now open the Air Drag dialog again and experiment with different drag coefficients C_d. Start with small values such as $C_d = 0.05$. For a rigid sphere, C_d is two. The effect of the drag is dramatic in that case.

Applying Motions to Bodies

The Body Actuator block cannot actuate a Body with motion signals. But you can construct such body motion actuators with a combination of other blocks. See "Joint Actuator Example: Body Driver" on page 1-58.

Varying a Body's Mass and Inertia Tensor

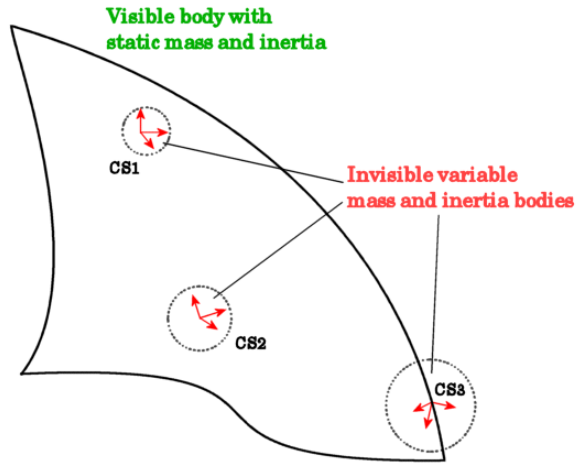
The Variable Mass & Inertia Actuator block gives you a way to vary a body's mass and/or inertia tensor as external functions of time. You specify these functions with incoming Simulink signals.

Caution The Variable Mass & Inertia Actuator block does not apply any thrust forces or torques to the Body so actuated. Mass loss or gain in a particular direction results in thrust forces and torques on the body. You must apply these forces/torques to the Body separately with Body Actuator blocks.

The variable mass/inertia actuator affects a body's motion only when you apply forces/torques on the body. When a body's motion is determined only by initial conditions, changing the mass or inertia tensor of a body does not affect its motion, because the variable mass/inertia actuator does not apply forces/torques to the body.

The Variable Mass & Inertia Actuator block changes the actuated Body's mass and rotational inertia by attaching an invisible body to the actuated body at a particular Body coordinate system (CS). This invisible body has a mass and an inertia tensor that vary in time as specified by the Actuator's external Simulink signal. The simulation treats the actuated body and the invisible body as a single composite body. The composite body has a new mass, new center of gravity (CG), and new inertia tensor compounded from its two constituent bodies.

You can add multiple Variable Mass & Inertia Actuator blocks to one Body. In that case, the simulation treats the actuated body and all attached invisible bodies as a single composite body. This composite body's mass, CG, and inertia tensor are compounded from its constituent bodies.



Attaching Variable Mass and Inertia Bodies to a Visible Body

To vary the mass and/or inertia tensor of a Body with this Actuator:

1. From the Sensors & Actuators library, drag a Variable Mass & Inertia Actuator block into your model.
2. Attach the Actuator's connector port to the Body CS on the Body where you want the invisible variable mass to be. If a suitable Body CS port does not exist on the Body, open its dialog and create one.
3. Create an external Simulink signal to model the time-varying mass and/or inertia tensor for this invisible body. Connect it to the Variable Mass & Inertia Actuator block's Simulink input port.

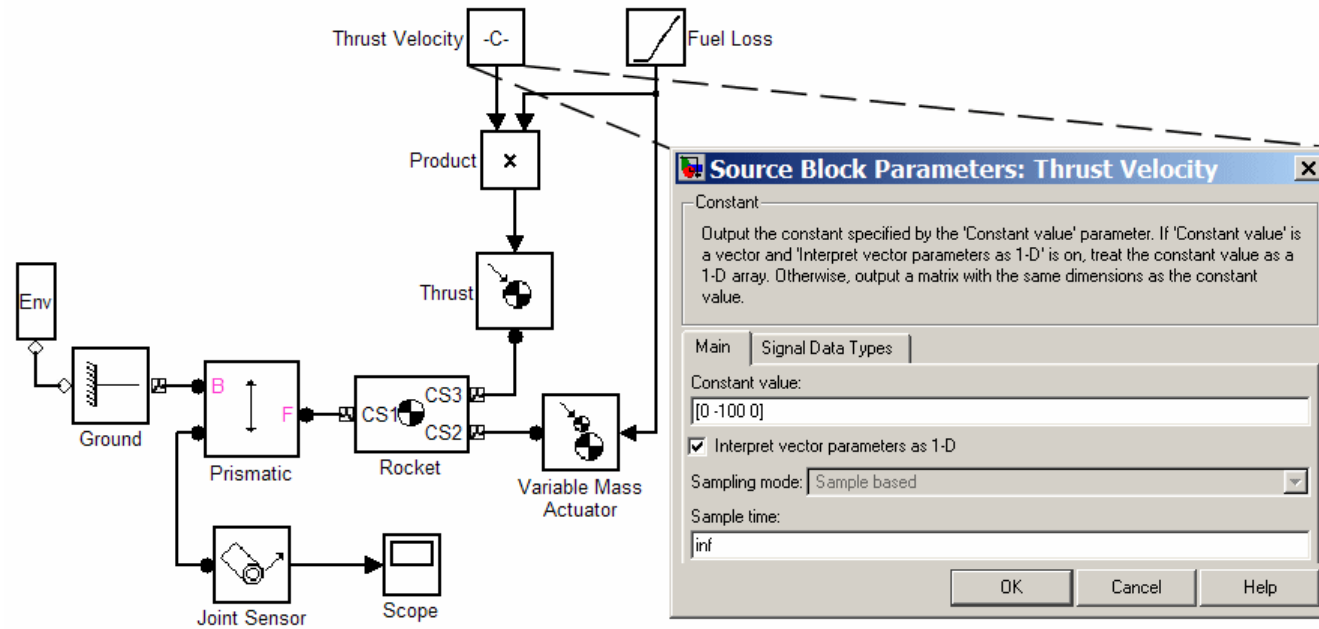
   This Simulink signal can have one, nine, or ten components, depending on whether you are varying the mass only, the inertia tensor only, or both.

Example: Simple Rocket

The following model simulates a simple rocket. It treats the rocket as a point mass moving upward (+*y* direction) with an exhaust pointing downward (-*y* direction). The rocket loses mass at a constant rate.

The Rocket block is the point mass. The Thrust Velocity block represents the downward exhaust and, multiplied by the mass loss represented by the Fuel Loss block, actuates the Rocket body with a thrust force pointing upward. The Thrust block (a body actuator) applies this force at the *local* Body CS, which, for a point rocket, is identical to the Rocket's CG CS.

The same mass loss from the Fuel Loss block that produces the thrust force also must vary the rocket's mass directly. The Variable Mass Actuator block accomplishes this by feeding the same mass loss signal to the Rocket block.



Actuating a Joint

You individually actuate each of the prismatic and revolute primitives of an assembled joint with a Joint Actuator. You can apply

- Forces or translational motions (but not both) to prismatic primitives
- Torques or rotational motions (but not both) to revolute primitives

---

Caution You cannot actuate spherical or weld primitives, disassembled joints, or massless connectors.

You can connect multiple Actuators to the same joint primitive. But it halts and displays an error message if you attempt to update or simulate a model containing such a connection.

Exception: You can apply a Joint Initial Condition Actuator and force or torque actuation (including stiction) to the same primitive. You cannot apply a Joint Initial Condition Actuator and motion actuation to the same primitive. See "Specifying Initial Positions and Velocities" on page 1-62.

To actuate a prismatic or revolute joint primitive of an assembled joint:

1. Create an Actuator port on the Joint block for the primitive (see "Creating Actuator and Sensor Ports on a Joint" on page 1-28).
2. Drag a Joint Actuator or Joint Stiction Actuator from the Sensors & Actuators library into your model and connect its output port to the Actuator port on the Joint.

   The remaining steps in this procedure apply to the creation of a standard Joint Actuator. For information on creating a stiction actuator, which applies classical Coulombic friction to a prismatic or revolute joint, see the Joint Stiction Actuator block reference page.
3. Open the Joint Actuator's dialog box.
4. Select the primitive you want to actuate from the **Connected to primitive** list on the dialog box.

5. Select the type of actuation you want to apply from the **Actuate with** pull-down menu, either `Generalized Forces` or `Motion`.

6. If you are actuating a prismatic primitive:

   - If you selected `Generalized Forces` as the actuation type, select the units of force from the **Applied force units** list.
   - If you selected `Motion` as the actuation type, select the units for each motion to be actuated (position, velocity, acceleration).

7. If you are actuating a revolute primitive:

   - If you selected `Generalized Forces` as the actuation type, select the units of torque from the **Applied torque units** list.
   - If you selected `Motion` as the actuation type, select the units for each motion to be actuated (angle, angular velocity, angular acceleration).

8. Click **OK** to apply your choices and dismiss the dialog box.

   Each joint primitive that you motion-actuate is lost as a true degree of freedom in your machine. That is because the DoF can no longer respond freely to externally applied forces or torques. See "Counting Model Degrees of Freedom" on page 1-89.

9. Create a signal that specifies the applied force, torque, or motions at each time step.

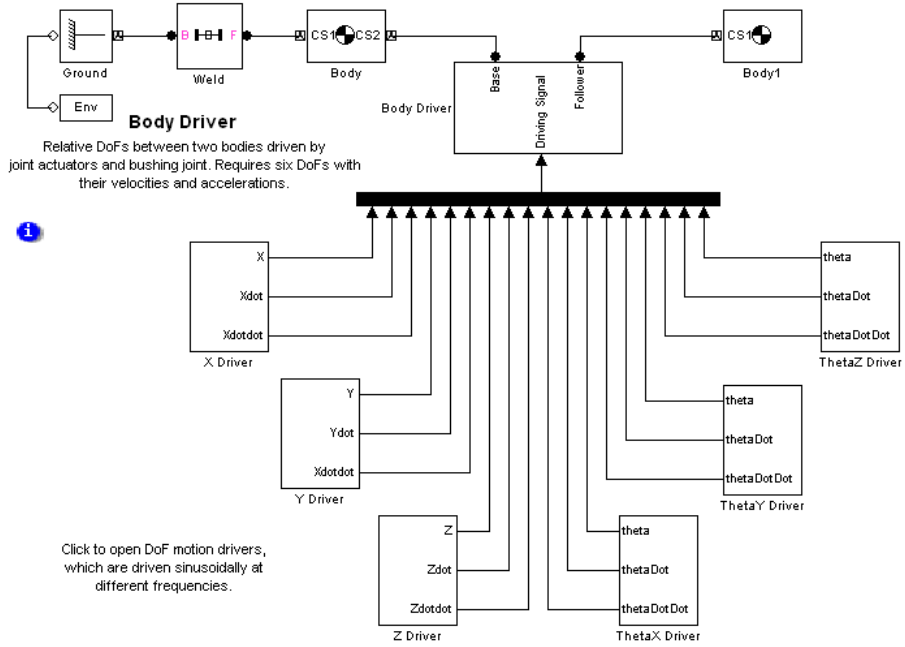
   You can use any Simulink source block or any combination of blocks to create the actuator signal. You can also connect the output of a Sensor block attached to the Joint to the Actuator input, thereby creating a feedback loop. You can use such loops to model springs and dampers attached to the joint.

   A force or torque signal must be a scalar signal. A motion signal must be a 1-D array signal comprising three components: position, velocity, and acceleration. The directionality of the joint determines the response of the follower to the sign of the actuator signal (see "Joint Directionality" on page 1-24).

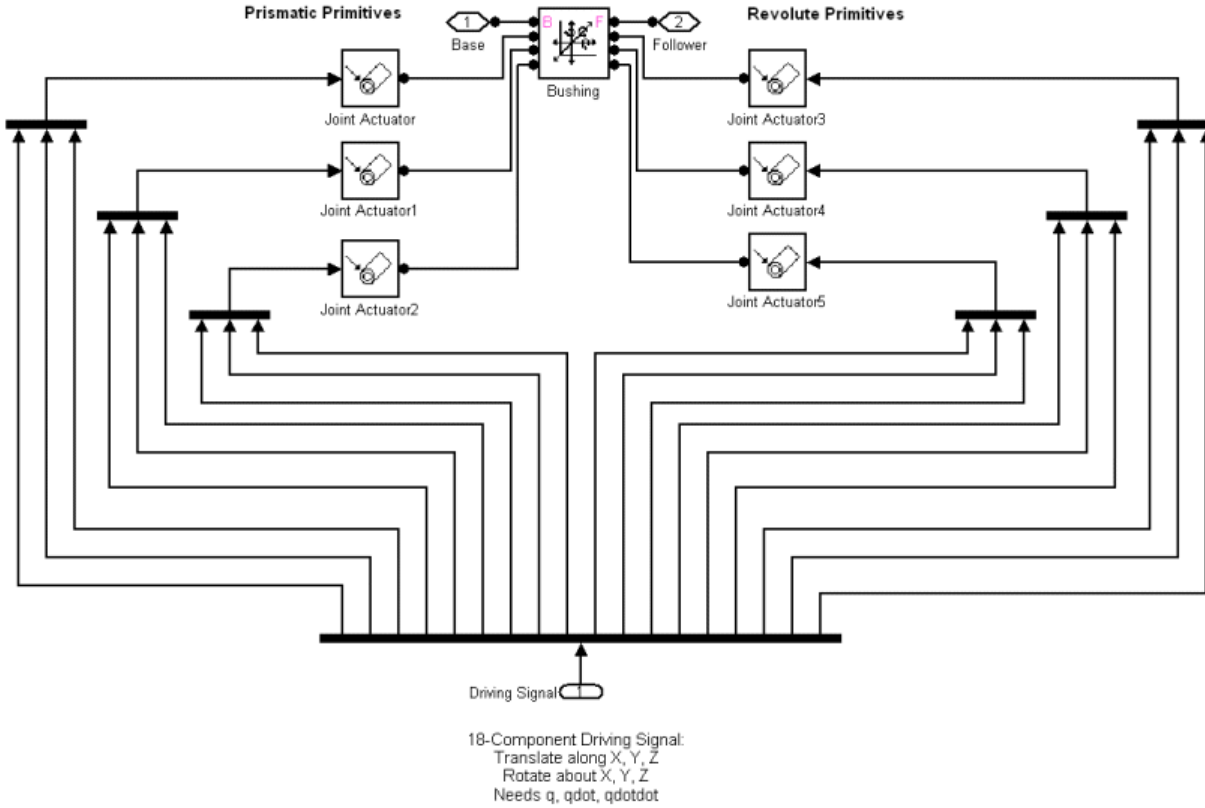
10. Connect the Actuator signal to the Actuator port on the Joint.

Joint Actuator Example: Body Driver

The `mech_body_driver` model illustrates the use of Joint Actuators to create a custom driver.



The Body Driver subsystem accepts an 18-component signal that feeds the coordinates, velocities, and accelerations for all six relative DoFs between Body and Body1. The subsystem uses a Bushing block that contains three translational and three rotational primitives to represent the relative DoFs:



You can modify the body driver to move only one of the bodies, thereby creating a motion actuator. To move Body1 relative to World, for example, remove the blocks Body and Weld and connect the subsystem Body Driver directly to Ground.

Joint Stiction Actuator Example: Mixed Static and Kinetic Friction

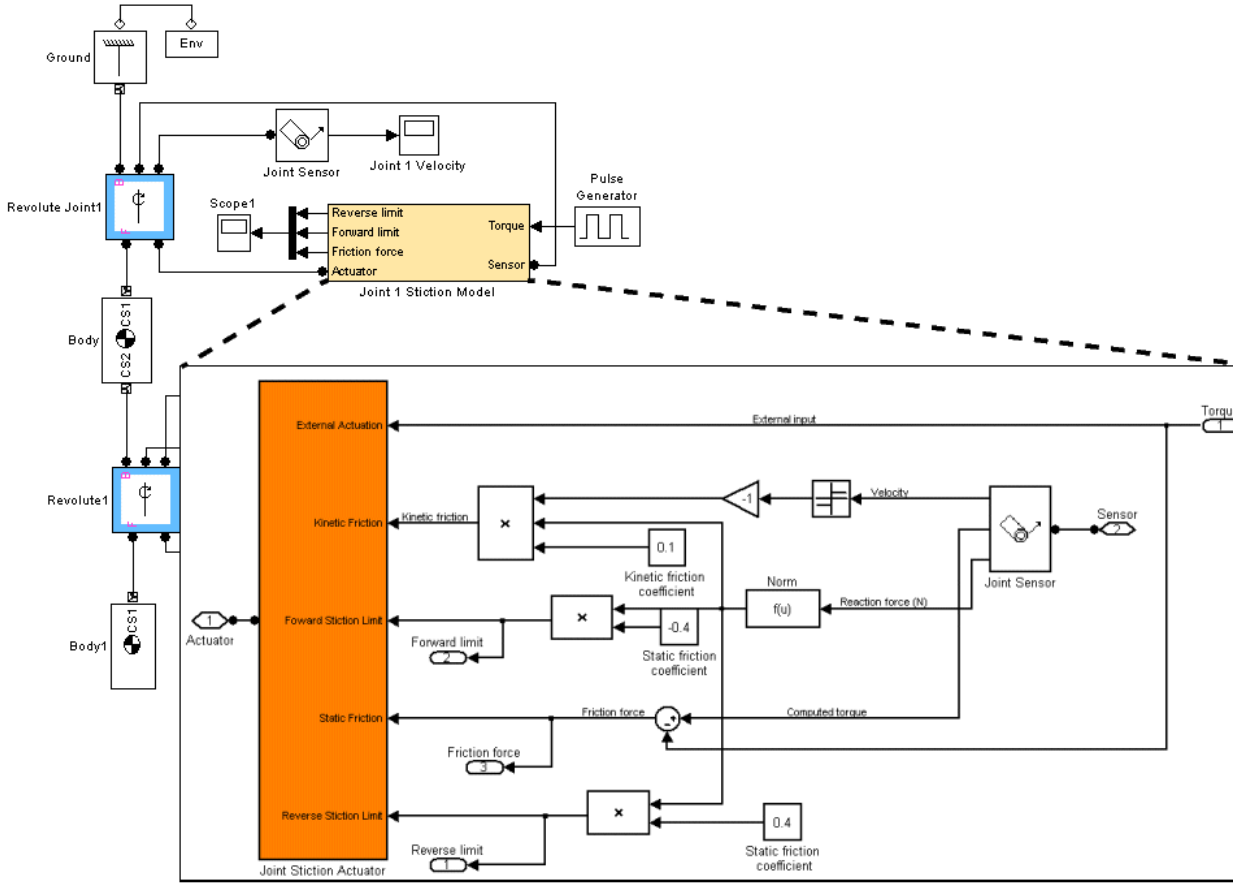
---

Tip You should use the Joint Stiction Actuator block only if you need static (locking) friction that removes one or more degrees of freedom from your machine.

You can model pure kinetic friction (damping) with other Actuator and Sensor blocks. See "Actuating a Body" on page 1-50 and "Adding Internal Forces" on page 1-74.

The `mech_dpen_sticky` model in the Demos library illustrates a driven double pendulum, with "sticky" friction or stiction applied to both revolute joints with the Joint Stiction Actuator block.

Open the unmasked Joint1 or Joint2 Stiction Model blocks (marked in yellow) to view the subsystems:



Each Stiction subsystem contains a Joint Stiction Actuator block (marked in orange) that requires static and kinetic friction coefficients via their respective blocks. For either revolute, an angular velocity threshold, specified through the block dialog, determines if a joint locks. Once locked, the joint cannot move until a combination of forces reaches a threshold specified by the `Forward Stiction Limit` or `Reverse Stiction Limit`.

Run the model with different kinetic and static friction coefficients and different velocity thresholds. View the results in the Scope blocks and through a visualization window. You can find more details on how SimMechanics stiction works by consulting the Joint Stiction Actuator block reference page.

Actuating a Driver

Actuating a Driver with a Driver Actuator allows you to specify the time dependence of the rheonomic constraint applied by the Driver.

To actuate a Driver:

1. Create an additional connector port on the Driver for the Actuator.

   Create the additional port in the same way you create an additional Sensor/Actuator port on a Joint (see "Creating Actuator and Sensor Ports on a Joint" on page 1-28).

2. Drag an instance of a Driver Actuator from the Sensors & Actuators library into your model.

3. Connect the Actuator's output port to the Actuator port on the Driver.

4. Create a signal that specifies the time dependence of the Driver constraint.

5. Connect the actuation signal to the input port of the Driver Actuator.

Specifying Initial Positions and Velocities

The Joint Initial Condition Actuator (JICA) block allows you to specify the initial positions and velocities of unactuated joints and hence the bodies attached to them. You can use JICA blocks to

- Specify nonzero initial joint velocities

  The default initial velocity of a joint primitive is zero. You must use a JICA block to specify a joint's initial velocity if the initial velocity is not zero.

- Override the initial position settings of a body pair

  The CG CS origin settings in the dialog boxes of Body blocks specify the bodies' initial positions. Using JICA blocks, you can override these initial

body positions by resetting their relative positions in the Joints connecting them.

Your model simulation starts with your machines at first in their home configurations, defined by the Body dialog data. It then transforms your machines to their initial configurations by applying JICA data.

Caution You cannot simultaneously actuate a joint primitive with a Joint Initial Condition Actuator and motion actuation from a Joint Actuator block.

Using JICA Blocks

Specifying initial conditions on a joint primitive is a special kind of actuation, one that occurs only once at the beginning of simulation. That is why the JICA block resides in the Sensors & Actuators library.

Note A JICA block, unlike other Actuators, does not have an input port. The JICA's dialog box specifies the Actuator input completely.

With a JICA block, you can specify the initial positions and velocities of any combination of prismatic and revolute primitives within a given Joint. (You cannot specify ICs for spherical and weld primitives.)

To specify the initial velocity and/or position of a joint primitive:

1. Drag a JICA block from the Sensors & Actuators library and drop it into your model window.
2. Create an additional connector port on the Joint block containing the primitive whose initial condition you want to specify.
3. Connect the connector port on the JICA block to the new connector port on the Joint block.

Caution Do not connect the JICA block to the Joint ports marked "B" or "F" (base or follower). These ports are intended for connecting to Bodies.

4. Open the JICA block's dialog box. From the primitive list for the Joint, choose the primitives you want to actuate by selecting their check boxes.

5. Enter the initial positions of the actuated primitives, relative to the Body CSs attached to the Joint, in the **Position** field.

   From the pull-down menu on the right, select **Units** for the initial positions.

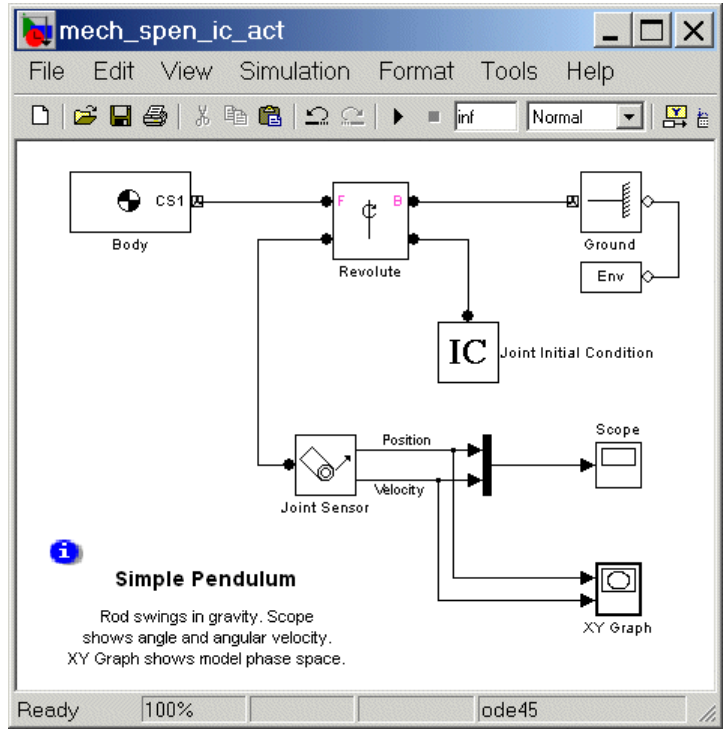
6. Enter the initial velocities of the actuated primitives, relative to the Body CSs attached to the Joint, in the **Velocity** field.

   From the pull-down menu on the right, select **Units** for the initial velocities.

7. Click **Apply** or **OK**.

JICA Example: A Simple Pendulum

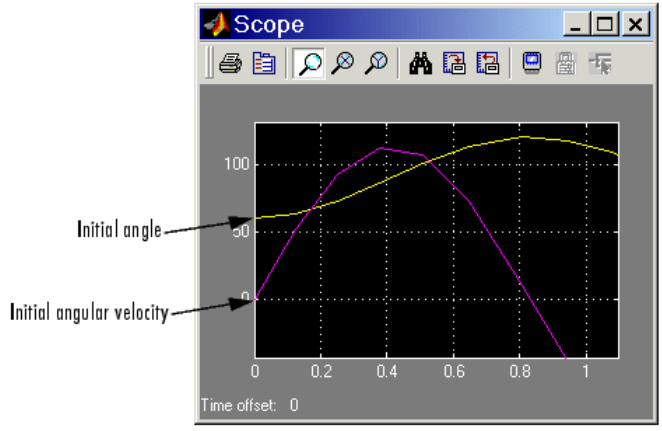
Open `mech_spen` from the Demos library, then open the Sensors & Actuators library. Follow the steps from the preceding section, "Using JICA Blocks" on page 1-63, to connect one Joint Initial Condition Actuator block to the Revolute block and configure it. This Joint contains only one primitive, `R1`, which is the primitive listed in the JICA dialog box.



Set the initial conditions in two ways and compare the resulting simulations in the scope:

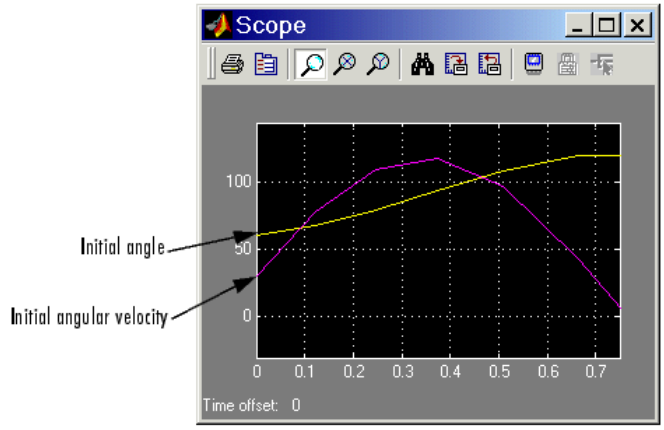
1. First set the initial **Position** (angle) to `60 deg`, which is 60° down from the left horizontal (30° clockwise from vertically down), and set the initial **Velocity** to `0 deg/s`.

2 Run the simulation for one second. Note in the scope that the initial angle (yellow curve) is displaced upward to 60°, while the initial velocity (purple curve) still starts at zero.



3 Now reset the initial **Velocity** to `30 deg/s`, leaving the initial **Position** (angle) at `60 deg`.

4 Rerun the simulation for one second. Note in Scope that the initial angle is still displaced upward to 60°, but the initial velocity is also displaced upward to 30°/sec.



The joint directionality is assigned in `mech_spen` so that the positive rotation axis is the +*z*-axis. Looking from the front, positive rotation swings down and right, counterclockwise.

Sensing Motions and Forces

In this section...

About Sensors

The SimMechanics Sensors & Actuators library provides a set of Sensor blocks that enable you to measure

- Body motions
- Joint motions and forces or torques on joints
- Constraint reaction forces and torques

All Sensor output is defined with respect to a fixed, conventional "zero." See "Home Configuration and Position-Orientation Measurements" on page 1-68.

Tip You can feed Sensor output back into Actuator blocks to model springs, dampers, and other mechanical devices that depend on force feedback. See "Actuating a Body" on page 1-50, "Actuating a Joint" on page 1-56, "Adding Internal Forces" on page 1-74, and "Validating Mechanical Models" on page 1-85.

Home Configuration and Position-Orientation Measurements

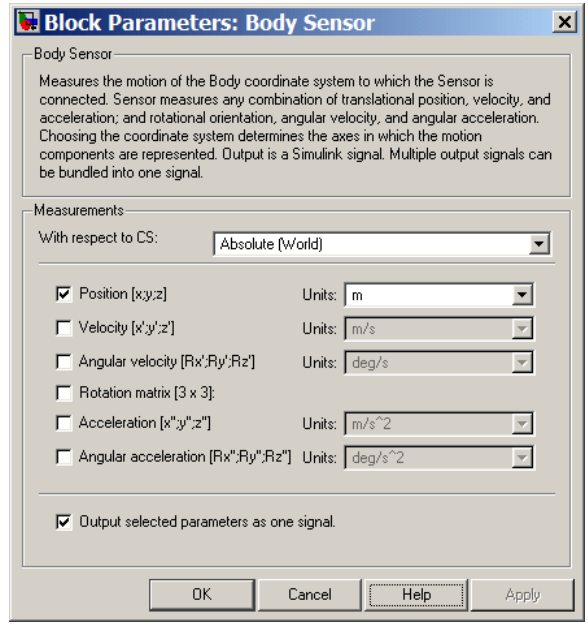
The Body and Joint Sensor blocks can measure the position and/or orientation of bodies and degrees of freedom. They make these measurements relative to the home configuration of the machine, the machine state *before* the application of initial condition actuators and assembly of disassembled joints. Thus motion sensors include the effect of the latter, which act before the simulation starts.

For further discussion, see “Modeling Disassembled Joints” on page 1-34 and “Specifying Initial Positions and Velocities” on page 1-62, and “Kinematics and the Machine’s State of Motion”.

Sensing Body Motions

To sense the position, velocity, or acceleration of a body represented by a Body block with a Body Sensor:

1. If the Body block does not have a spare local coordinate system with a Body CS port, create one (see “Managing Body Coordinate Systems” on page 1-17).
2. Drag a Body Sensor block from the Sensors & Actuators library into your model.
3. Connect its connector port to a spare Body CS port on the Body.
4. Open the Sensor’s dialog box.



5 Select the coordinate system relative to which the sensor measures its output from the **With respect to CS** list.

6 Select the check boxes next to the motions that you want to sense (see the Body Sensor block reference page).

7 If you have chosen to sense more than one type of motion and want the Sensor to multiplex the motions into a single output signal, select the **Output selected parameters as one signal** check box.

8 Click **OK** or **Apply**.

9 Connect the output of the Body Sensor block to a Simulink Scope or other signal sink or to a motion feedback loop, depending on your needs.

Sensing Joint Motions and Forces

The Joint Sensor block enables you to measure the motions of degrees of freedom. It can also measure the relative forces and torques between the bodies connected to the joint. These include the *computed force* or *torque* (the force or torque needed to reproduce the joint's motion) and the *reaction force* and *torque* on a joint primitive. (You cannot measure the computed force or torque on a spherical or weld primitive.) You must connect a separate Joint Sensor block to a Joint block for each joint primitive that you want to sense.

To sense the motions, forces, and torques of a joint primitive contained by a Joint block:

1 If the Joint block does not have a spare Sensor port, create one (see "Creating Actuator and Sensor Ports on a Joint" on page 1-28).

2 Drag a Joint Sensor block from the Sensors & Actuators library into your model.

3 Connect its connector port to the spare Sensor port on the joint.

4 Use the Sensor's dialog box to configure the Sensor to measure the motions, forces, and torques that you want to measure (see the Joint Sensor block reference page).

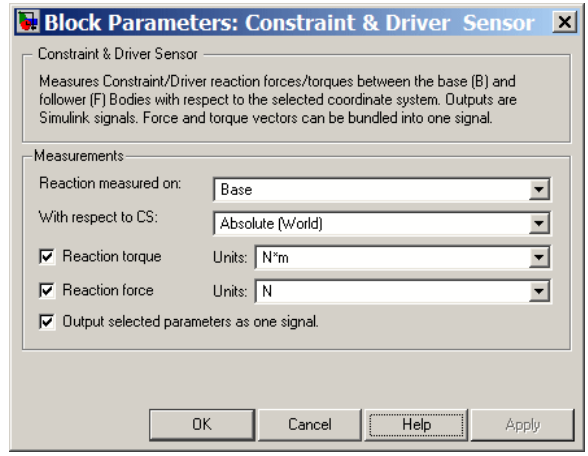
5 Connect the output of the Joint Sensor block to a Simulink Scope or other signal sink or to a motion feedback loop, depending on your needs.

Sensing Constraint Reaction Forces

The Constraint & Driver Sensor block enables you to measure the reaction forces and torques induced on the constraints modeled by SimMechanics Constraint and Driver blocks.

To sense the reaction force and/or torque induced by a constraint or driver,

1. If the Constraint or Driver does not have a spare Sensor port, create one.
2. Drag a Constraint & Driver Sensor block from the Sensors & Actuators library into your model.
3. Connect its connector port to a Sensor port on the Constraint or Driver block.
4. Open the Sensor block's dialog box.



5. Select the body (follower or base) on which to measure the reaction force from the **Reactions measured on** list.
6. Select the coordinate system relative to which the Sensor measures its output from the **With respect to coordinate system** list.
7. Select the **Reaction torque** check box if you want the Sensor to output the reaction torque on the base (or follower) body.

8 Select the **Reaction force** check box if you want the Sensor to output the reaction force on the base (or follower) body.

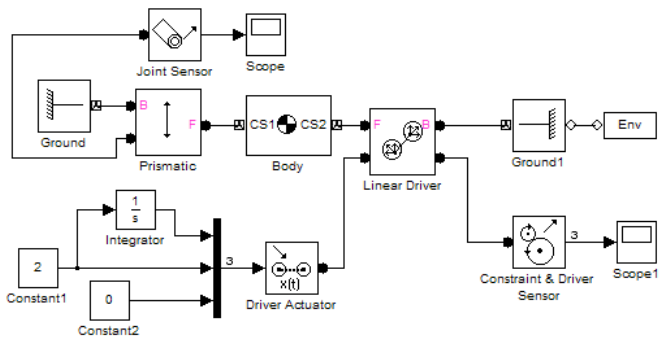
9 If you have chosen to output both reaction force and torque and want the Sensor to multiplex them into a single output signal, select the **Output selected parameters as one signal** check box.

10 Click **OK** or **Apply**. Connect the output of the Constraint & Driver Sensor block to a Simulink Scope or other signal sink or to a motion feedback loop, depending on your needs.

Not all the reaction force/torque components are significant. Only those components projected into the subspace of constrained or driven degrees of freedom (DoFs) are physical. Components orthogonal to the constrained or driven degrees of freedom are not physical.

Example: Linear Driver

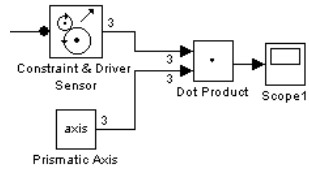
In this example, you drive a body along the *x*-axis, but only allow it a prismatic DoF tilted at an angle in the *x-y* plane. Construct the following model.



Configure the Constraint & Driver Sensor to measure only the reaction force, not the torque. Configure the Linear Driver to drive the Body along the World *x*-axis, but set up the Prismatic with a primitive axis along (1, 2, 0). The body can then move only along this axis, but is driven along the horizontal *x*-axis. Measure all motions and forces in World. Leave all other settings at default.

Open the Scopes and run the model. The measured reaction force lies along the *x*-axis, with a value of -19.62 N (newtons) = -2*mg*. Because the constrained DoF is not parallel to the *x*-axis, you need to project the reaction force along the unit vector $(1, 2, 0)/\sqrt{5}$ defining the direction of the prismatic primitive to obtain the physical part.

Add to the model the Simulink blocks that form a dot product between the reaction force signal (three components) and the prismatic unit vector (also three components). (You can define a workspace vector for this axis and use it in both the joint and the dot product.) Reconnect Scope1 to measure this physical component of the reaction force.



The physical component of the reaction force is $-(19.62\text{ N}) \cdot (1/\sqrt{5}) = -8.77$ N. The component of the reaction force orthogonal to (1, 2, 0) is not physical.

Adding Internal Forces

In this section...

About Force Elements

Internal forces are forces the machine applies to itself as a result of its own motion. Unlike actuation forces, you do not apply these forces from outside the machine with Simulink signals. The body motions instead generate the forces and torques directly.

The Force Elements library provides ready-made blocks to represent certain kinds of internal forces and torques acting between bodies. You can also create your own customized sensor-actuator feedback loops to model springs, dampers, and more complex internal forces.

Inserting a Linear Force Between Bodies

A generalized linear force between two bodies is a linear function of the two bodies' relative displacement vector $\boldsymbol{r}$ and relative velocity $\boldsymbol{v}$, with constant coefficients. The Body Spring & Damper block models a force acting between two bodies along the axis $\boldsymbol{r}$ connecting them:

$$F = -k(r - r_0) - bv_{||}$$

The block is connected on either side to Bodies at a Body coordinate system (CS). The displacement $\boldsymbol{r}$ is a vector from one Body CS on one Body to the other Body CS on the other Body. Newton's third law requires that the forces that the bodies exert on one another be equal and opposite.

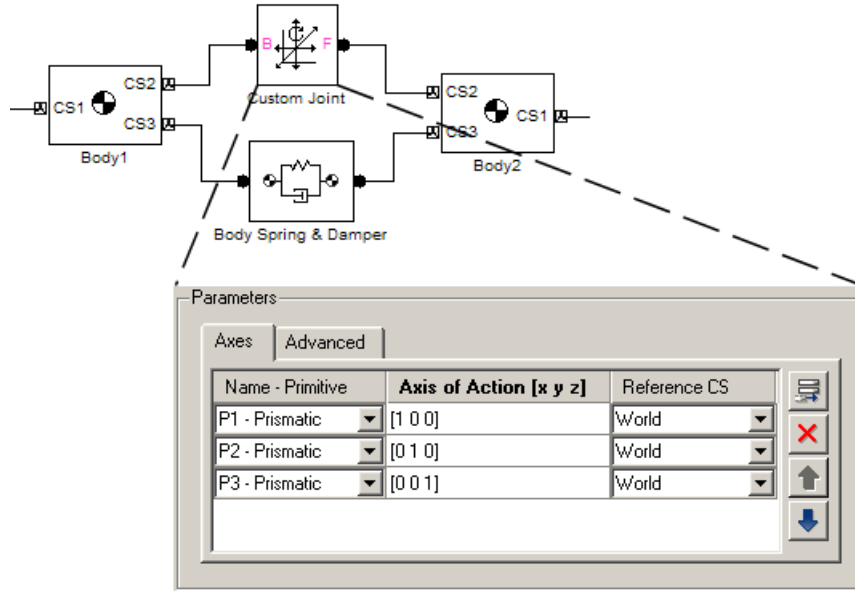
The common physical system this force model represents is a spring-damper combination, where the damper is a dashpot acting only along the spring axis. The damping is solely a function of the component $v_{||}$ of the velocity

vector projected along the displacement ***r***. (Thus the damping in this block cannot represent the damping due to a viscous medium, because there is no damping force perpendicular to the spring axis. See "Inserting a Linear Force or Torque Through a Joint" on page 1-76.)

You enter the constant parameters r_0, k, and b in the Body Spring & Damper dialog. r_0 is the spring's natural length, the length it has when no forces are acting on it. The spring constant k and damping constant b should be nonnegative.

To complete a linear force model between bodies, you need to model the translational degrees of freedom (DoFs) between them, as the Force Element block itself does not represent these DoFs. You can use any Joint block containing at least one prismatic primitive to represent translational motion. The two Bodies, the Joint, and the Body Spring & Damper must form a closed loop.

The following block diagram represents two Bodies with a damped spring between them. The Custom Joint represents the bodies' relative translational DoFs with three prismatic primitives. In this case, CS2 and CS3 on Body1 are the same, and CS2 and CS3 on Body2 are the same. Thus, the Joint is connected to the same Body CSs that define the ends of the spring-damper axis.



Inserting a Linear Force or Torque Through a Joint

Another way of inserting a linear force element between two bodies is to connect it to a joint that already connects the bodies. You have to apply the force element, like an actuator, to each primitive in the joint individually. This approach has several advantages over the Body Spring & Damper:

- You can create a different force law, with a different spring length, spring constant, and damping constant, for each of the joint's primitives.
- The spring and damper forces acting on each primitive act independently in their respective directions, instead of depending on just the interbody distance with a single spring length, spring constant, and damping constant.

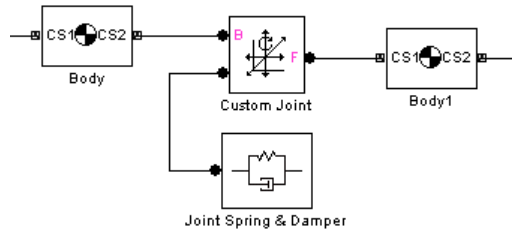
  This allows you to create spring and damping forces that act independently in two or three dimensions, unlike the Body Spring & Damper force, which acts only along a single axis. Damping forces acting on multiple primitives act as a two- and three-dimensional viscous medium, not as a dashpot.
- The joint representing the DoFs between the bodies is already present.

You use the Joint Spring & Damper block to implement such spring-damper forces/torques together with a Joint. With it, you can apply a linear spring and damper force to each prismatic primitive and a linear torsion and damper torque to each revolute primitive in a Joint block. (You cannot apply these torques to a spherical primitive.)

Pick a Joint already connected between two Bodies. You connect the Joint Spring & Damper block to a Joint block at a sensor/actuator port on the Joint. (The section "Actuating a Joint" on page 1-56 explains how to create such a port.) The Joint Spring & Damper dialog then lists each primitive in the Joint.

For each prismatic primitive you want to actuate with a spring-damper force, you specify a natural spring length (offset), spring constant, and damping constant. For each revolute primitive you want to actuate with a torsion-damper torque, you specify a natural torsion angle (offset, or angle in which the primitive points absent any torques), torsion constant, and damping constant. You make these specifications in the Joint Spring & Damper dialog.

Here are two bodies connected by a Custom Joint in turn connected to a Joint Spring & Damper block.



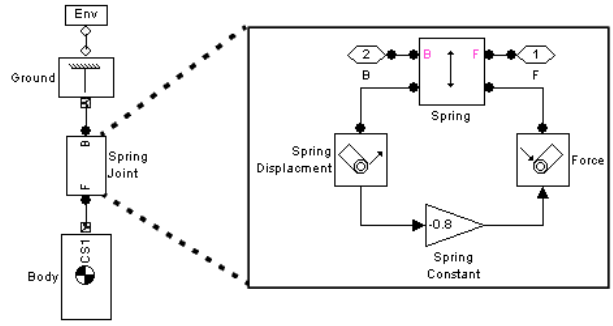
Unlike the example in the preceding section, "Inserting a Linear Force Between Bodies" on page 1-74, the Custom Joint can have up to three prismatics and three revolutes, each with a separate linear force or torque acting through it. Each force or torque acts equally and oppositely on each body, following Newton's third law.

Customizing Force Elements with Sensor-Actuator Feedback

You can create your own force elements acting through Joints or on Bodies by using Sensor-Actuator feedback loops. With this technique, you can not only model linear forces, but any force that depends on body or joint positions and velocities.

This simple example illustrates the method with a linear spring force law. Hooke's law states that the force exerted by an extended spring is proportional to its displacement from its unextended position: $F = -kx$.

The following SimMechanics model represents a spring that obeys Hooke's law.



The model uses the Gain block labeled Spring Constant to multiply the displacement of the prismatic joint labeled Spring along the World's *y*-axis by the spring constant `-0.8`. The output of the Gain block is the force exerted by the spring. The model feeds the force back into the prismatic joint via the Actuator labeled Force. The model encapsulates the spring block diagram in a subsystem to clarify the model and to allow a spring to be inserted elsewhere.

Combining One- and Three-Dimensional Mechanical Elements

In this section...

About Interface Elements

SimMechanics software is built on the Simscape environment, which supports one-dimensional domains of translational and rotational motion, along or about a single axis for one body at a time. The mechanical elements of the Simscape Foundation library include masses, inertias, and internal forces and torques, as well as sensors and actuators. The blocks of the Interface Elements library allow you to selectively couple a SimMechanics machine to a mechanical circuit.

Consult the Simscape documentation for more about Physical Networks and one-dimensional domains.

How Mechanical Interface Elements Couple Motion and Forces Between SimMechanics Machines and Simscape Circuits

Because Simscape models simulate motion along or about one axis, one Interface Element block can couple only one SimMechanics joint primitive at a time to a Simscape circuit, interfacing through a Sensor/Actuator port on a Joint block. An Interface Element neither adds nor subtracts degrees of freedom (DoFs) to or from the combined machine-mechanical circuit. Its coupling is like a force element between the two domains (see "Adding Internal Forces" on page 1-74):

- From the point of view of the SimMechanics machine, an Interface Element behaves like a force actuator acting on the selected joint primitive. The interface block injects force or torque from the mechanical circuit between the Bodies connected to either side of the Joint.

- From the point of view of the Simscape mechanical circuit, an Interface Element behaves like a motion actuator. The interface block injects translational or rotational motion from the machine into the circuit connection line.
- The directionality or sense of motion, established by the base (B)-follower (F) order in the SimMechanics Joint, is preserved in the Simscape mechanical circuit.
- An Interface Element preserves the force or torque flowing through the Interface Element into the machine and the motion acting across the Joint transmitted into the mechanical circuit. Interface Elements thus conserve mechanical power, transferring it without loss between the two domains.

Interface Elements can couple prismatic or revolute joint primitives to translational or rotational motion, through the Prismatic-Translational Interface or Revolute-Rotational Interface blocks, respectively.

Limitations on the Interfaced Simscape Mechanical Circuit

SimMechanics and Simscape mechanical simulations are separately valid. However, simulation of moving bodies modeled as Simscape mass and inertia elements coupled through Interface Elements to a SimMechanics machine is not complete and requires care to avoid unphysical results. These limitations arise from their different representations of motion and dynamics coming into conflict:

- One-dimensional motion in Simscape circuits versus three-dimensional motion in SimMechanics machines.

  A Simscape circuit does not model the motion of such bodies along or about axes orthogonal to the coupled primitive axis chosen in the interfaced Joint.

- Absolute motion of each Simscape mass and inertia represented by connection lines versus relative motion, represented by Joints, between SimMechanics bodies.

  All masses in Simscape models live in an implicit inertial reference frame. A Simscape mechanical circuit interfaced to a SimMechanics machine in general moves in an accelerated frame. A simulation with such a circuit does not include the pseudoforces acting on the Simscape mass and inertia elements as experienced in such a noninertial frame and thus violates Newton's second law of mechanics.

As the mass and/or inertia modeled in the interfaced mechanical circuit is increased, so is the violation of Newton's second law. As such mass and/or inertia is decreased, so is the violation.

Warning **Model all masses and inertias in your system as Bodies in the SimMechanics machine and avoid placing mass and inertia elements into any interfaced Simscape mechanical circuits.**

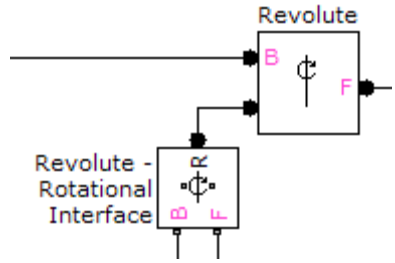
Models with mass and inertia elements in Simscape mechanical circuits interfaced to a SimMechanics machine are not physically valid. Simulating with such models does not yield valid results.

Working with Interface Elements

To interface a Joint with a Simscape mechanical circuit:

1. Select the appropriate Interface Element block, prismatic or revolute, from the Interface Elements library.
   - If you wish to couple a translational mechanical circuit to a prismatic primitive, select Prismatic-Translational Interface.
   - If you wish to couple a rotational mechanical circuit to a revolute primitive, select Revolute-Rotational Interface.

   You cannot mix translational and rotational motion with an Interface Element.
2. Copy the selected Interface Element block into your model.
3. Open the Joint dialog and add an extra Sensor/Actuator port. Close the dialog.
4. Connect the Interface Element mechanical connector port ○ to the new Sensor/Actuator port on the Joint.



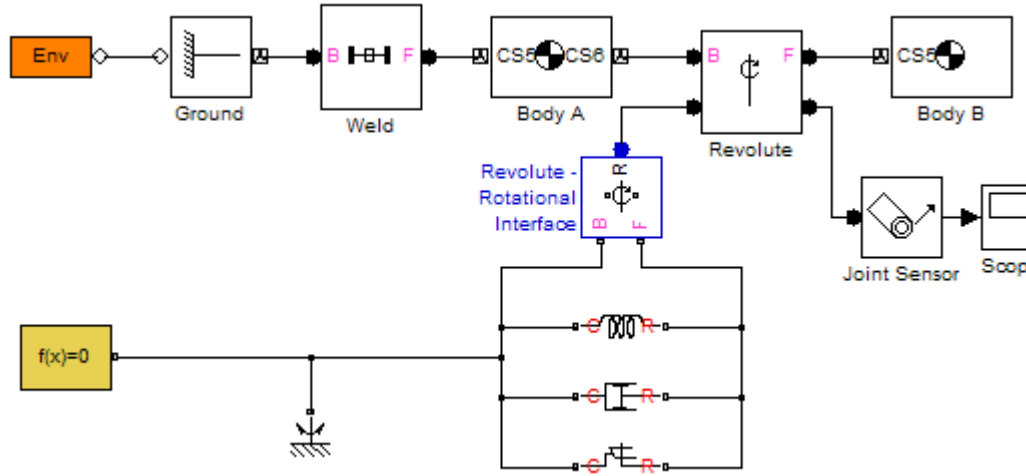
5 Open the Interface Element dialog. The **Connected to primitive** pull-down menu contains a list of all the primitives of appropriate type (prismatic or revolute) in the interfaced Joint.

   Select the primitive you want to interface. The Simscape circuit will move along or about that axis. Click **OK** or **Apply**.

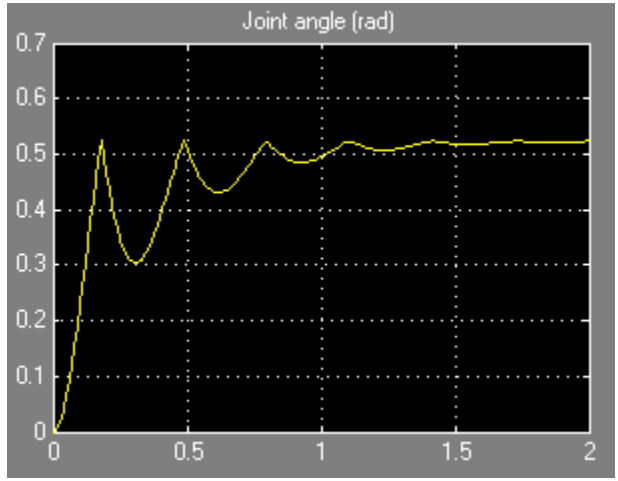
On the machine side, the Joint must follow the standard rules for Joints and in particular be connected to a Body on each side. (See "Modeling Degrees of Freedom" on page 1-19.) You should connect the Interface Element with the rest of the mechanical circuit.

Example: Rotational Spring-Damper with Hard Stop

The `mech_interface_rot_spr_damper` demo illustrates proper interface of Simscape mechanical elements with a three-dimensional SimMechanics machine.



The interfaced mechanical circuit has no inertia or mass elements, which prevents the problems discussed in “Limitations on the Interfaced Simscape Mechanical Circuit” on page 1-80. It contains only force elements: a rotational spring, a rotational damper, and a rotational hard stop. Together, these force elements create a hard stop for the Revolute block. This block contains only one primitive, `R1`, which you can view by opening its dialog. Through this primitive, the Simscape force elements act between Body A and Body B, limiting their relative angular motion about the revolute `R1` axis to ±π/6 radians.


Joint angle (rad)
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.5
1
1.5
2

Validating Mechanical Models

In this section...

Essential Tests for Model Validity

Simulink can simulate a SimMechanics model only if it is valid. A model is valid if it satisfies the following rules:

- Each machine in the model contains at least one Ground, and exactly one Ground in each machine is connected to a Machine Environment block. Each submachine connected to a full machine by a Shared Environment block must have at least one Ground.

  See "Representing Machines with Models" on page 1-2.

- Every machine in the model is topologically valid. See "Verifying Model Topology" on page 1-85.
- The model contains at least one degree of freedom. See "Counting Model Degrees of Freedom" on page 1-89.
- Any machine in the model interfaced to Simscape mechanical circuits satisfies both SimMechanics and Simscape modeling rules. See "Combining One- and Three-Dimensional Mechanical Elements" on page 1-79.

Verifying Model Topology

To avoid simulation failures, you must ensure that the topology of your block diagram is valid. A block diagram is topologically valid if each machine that it contains is valid. A machine is valid if its *spanning tree* is valid. Thus to determine if your model is valid, first determine the spanning tree of each machine that it contains and then the validity of each resulting tree.

Machine Topology and Subsystems

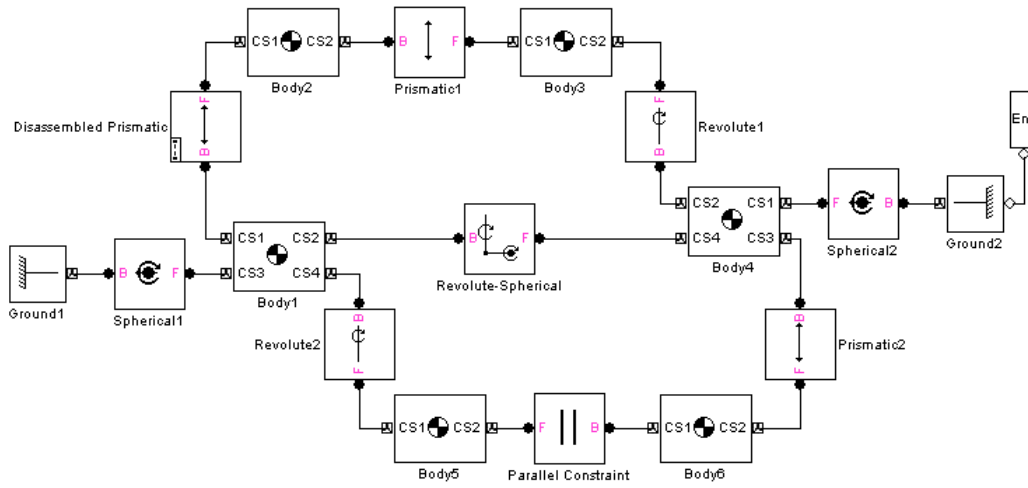
When examining your model's topology, be sure to inspect all its subsystems, including masked subsystems, down to the bottom of the model's subsystem hierarchy.

Determining a Machine's Spanning Tree

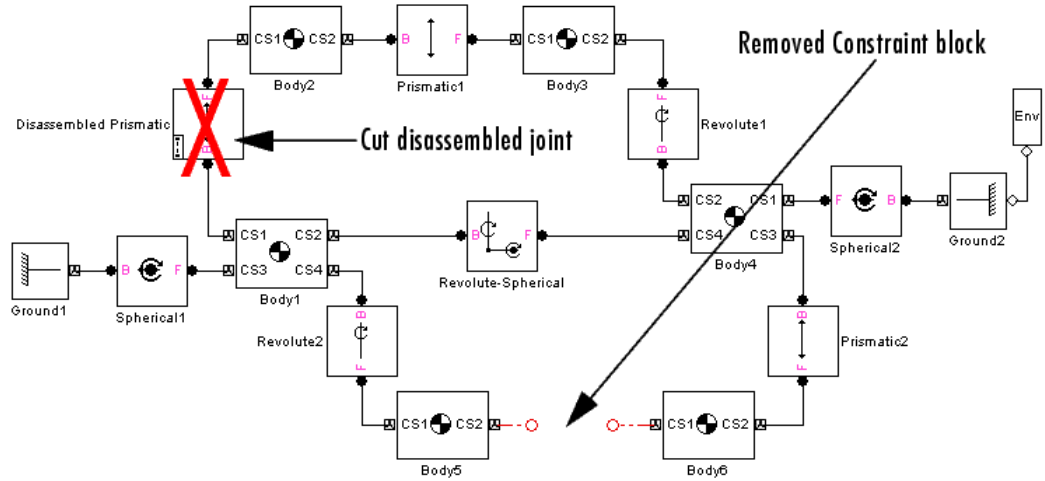
You can think of a machine as a graph with *elements* (bodies) and *connectors* (joints, constraints, and drivers). A *spanning tree* is a reduced graph with bodies connected only by joints and all closed loops cut once.

To determine the spanning tree of a machine, remove all blocks from the machine except Body and Joint blocks and open every closed loop in the resulting reduced machine. To open a closed loop, follow the loop-cutting rules in "Cutting Machine Diagram Loops" on page 1-46.

For example, here is a machine with two closed loops.



Cutting the top loop at the Disassembled Prismatic and removing the Parallel Constraint block (thus simultaneously cutting the bottom loop) yields the machine's spanning tree, as shown here.



Determining the Validity of a Spanning Tree

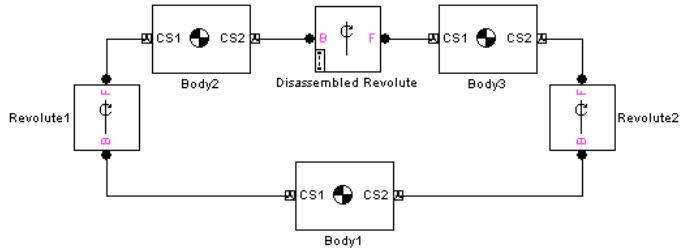
To be valid, a spanning tree must meet these requirements:

- The spanning tree must have at least one Ground block to serve as a reference to World.
- Every Joint block must be connected to exactly two Body blocks.
- Every non-Ground Body block must have a unique path to a Ground block. (This need not be true of the whole machine.) This ensures that, while each body moves via joints relative to other bodies, the simulation can resolve all bodies' motions relative to one another into absolute motions with respect to World.
- Every non-Ground Body block at an end of a chain of Bodies must have nonzero inertia (mass or inertial moment) associated with all joint primitives that can move. Each translational DoF must carry a nonzero mass, and each rotational DoF a nonzero inertial moment. This prevents infinite accelerations when forces and torques are applied.

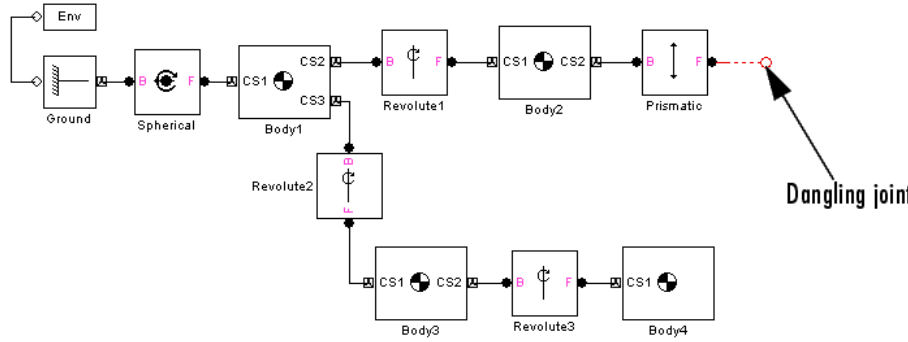
Examples of Invalid Machine Topologies

Here are some examples of invalid topologies:

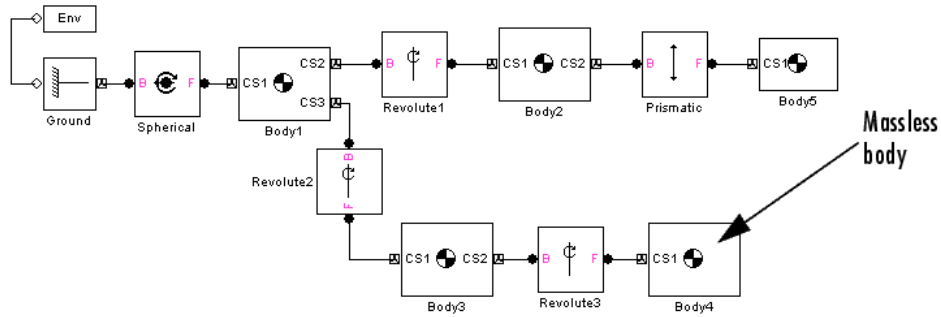
- This one-loop machine lacks a Ground block.



- This open machine has a dangling Joint block.



- Another open machine features a zero-mass body at one end of a chain of bodies.



The last two invalid examples are dynamically (but not topologically) equivalent, because a zero-mass body is dynamically no body at all.

Counting Model Degrees of Freedom

Identifying and counting the independent degrees of freedom (DoFs) of a machine are important for trimming and linearizing SimMechanics models (see “Trimming Mechanical Models” on page 3-18 and “Linearizing Mechanical Models” on page 3-32) and for correcting simulation errors (see “Troubleshooting Simulation Errors” on page 2-26).

Your SimMechanics model must have at least one DoF to be valid. A free physical body has six DoFs: three translational and three rotational. But in a machine, connections between bodies by joints, constraints, and drivers, and motion actuation by joint and body actuators reduce the machine’s independent DoFs to a smaller number. You also reduce a body’s DoFs if you confine the machine’s motion to one or two spatial dimensions.

A SimMechanics Body block has no DoFs. Connecting Joints to a Body adds DoFs to the machine. The joint primitives represent the Body’s DoFs relative to other connected Bodies or Grounds. Connecting Constraint and Driver blocks to Bodies or motion-actuating joint primitives in Joints removes DoFs from the machine. A locked Joint Stiction Actuator also removes a DoF.

Degrees of Freedom in Subsystems

When you examine your model to identify and count its DoFs, be sure to open and inspect all its subsystems, including masked subsystems, to the bottom of the model's subsystem hierarchy.

Finding Independent Degrees of Freedom

Here is the formula for determining the *number of independent DoFs* your model has:

```
# of independent DoFs = # of body DoFs + # of primitive DoFs -
                        # of motion restrictions
```

The following three steps define each term on the right side:

1. Calculate the *number of body DoFs* from the number of Body and Joint blocks in your model:

   ```
   # of body DoFs = 6 * (number of Bodies - number of Joints)
   ```

   If you have confined the machine to move in only two dimensions, replace the 6 by 3. If you have confined the machine to move in only one dimension, replace the 6 by 1.

2. Calculate the *number of primitive DoFs* by adding up the primitive DoFs from the Joint dialog boxes:

   - Count one for each prismatic (`P`) or revolute (`R`) primitive.
   - Count three for each spherical (`S`) primitive.
   - Count zero for each weld (`W`) primitive.

   Do not count a primitive DoF that is motion-actuated by a Joint Actuator.

3. Calculate the *number of motion restrictions* by adding up the motion restrictions of each Constraint and Driver block and from each locked Joint Stiction Actuator. Different blocks from the Constraints & Drivers library impose different numbers of motion restrictions. Stiction actuators apply to individual joint primitives.

Constraint Block	Restrictions	Driver Block	Restrictions
Gear	One	Angle	One
Parallel	Two	Distance	One
Point-Curve	Two	Linear	One
		Velocity	One

Be sure not to count redundant motion restrictions. These are restrictions that forbid the motion of joint primitives that could not move anyway even if the constraint were removed, because of how the joints are configured.

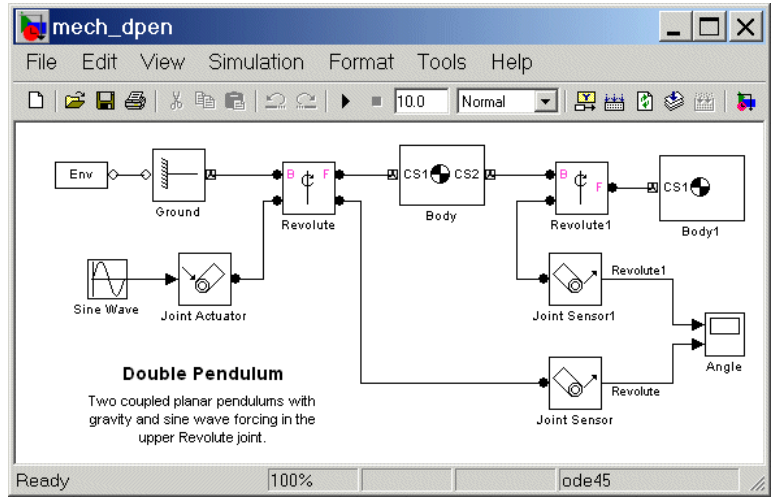
Example: A body is connected to a ground by a single prismatic. You place a constraint on the body that prevents it from moving perpendicularly to the prismatic axis. The body could not move in that direction even if you removed the constraint. So the constraint is redundant, and you would not count it as a motion restriction.

The Role of Joint Stiction Actuators

A Joint Stiction Actuator can remove or restore a DoF during a simulation. It is the only block that can change the number of independent DoFs after you start simulating. You must count an additional motion restriction during the period when a stiction-actuated primitive is locked. The primitive counts as another DoF if it is unlocked.

DoF Example: Double Pendulum

The `mech_dpen` model from the Demos library represents planar double pendulum motion actuated by a Joint Actuator.



The double pendulum has two rigid bodies, such as two rods, confined to move in two dimensions. Ignoring the Joint Actuator temporarily, there are two bodies, two joints, and two revolute primitives, and thus 3 * (2 - 2) + 2 = 2 independent DoFs. There are many ways to represent these two DoFs, but the two revolute primitives are the simplest way.

Including the Joint Actuator in the DoF count removes the revolute primitive in the Revolute block as an independent DoF. So this model actually only has one independent DoF, the revolute primitive in the Revolute1 block.

DoF Example: Four Bar Mechanism

The example in "Creating a Closed-Loop Mechanical Model" has four revolutes. You can establish that only 3 * (3 - 4) + 4 = 1 of these DoFs is actually independent and arrive at the same result obtained in the example.

2

Running Mechanical Models

SimMechanics software gives you multiple ways to simulate and analyze machine motion in the Simulink environment. Running a mechanical simulation is similar to running a simulation of any other type of Simulink model. It entails setting various simulation options, starting the simulation, interpreting results, and dealing with simulation errors. See the Simulink documentation for a general discussion of these topics. This chapter focuses on aspects of simulation specific to SimMechanics models.

Configuring SimMechanics Models in Simulink

In this section...

SimMechanics and Simulink Options

Simulink provides an extensive set of simulation options that apply to any type of model. Additional options apply specifically to simulating SimMechanics models. This section discusses those standard Simulink options for which mechanical models require special consideration and the additional SimMechanics options specific to mechanical systems .

Distinguishing Models and Machines

Respecting the distinction introduced in "Representing Machines with Models" on page 1-2, you need to make two categories of settings, one for each machine in a model and one for the entire SimMechanics model. To configure a mechanical model for simulation, you need to interact with two dialogs.

- "Machine Settings via the Machine Environment Block" on page 2-2 makes use of the Machine Environment block dialog.
- "Model-Wide Settings via Simulink and Simscape Software" on page 2-3 uses the Simulink Configuration Parameters dialog.

"Configuring Methods of Solution" on page 2-6 discusses the settings in detail.

Machine Settings via the Machine Environment Block

Every machine in your model requires exactly one Machine Environment block to be connected to one of its Ground blocks. The mechanical settings that you enter in that Machine Environment block determine the mechanical environment for that machine only. Other machines are controlled by their respective Machine Environment blocks.

This block controls the connected machine's mechanical environment, including simulation dynamics, machine dimensionality, gravity, tolerances, constraints, motion analysis modes, and visualization. See the Machine Environment reference page for a full description of the block dialog's four tabs.

The Machine Environment settings are also presented in the following sections:

- "Defining Gravity" on page 2-6
- "Choosing Your Machine's Dimensionality" on page 2-7
- "Choosing an Analysis Mode" on page 2-8
- "Controlling Machine Assembly" on page 2-12
- "Maintaining Constraints" on page 2-12
- "Handling Motion Singularities" on page 2-18
- "Setting Up Visualization" on page 2-22

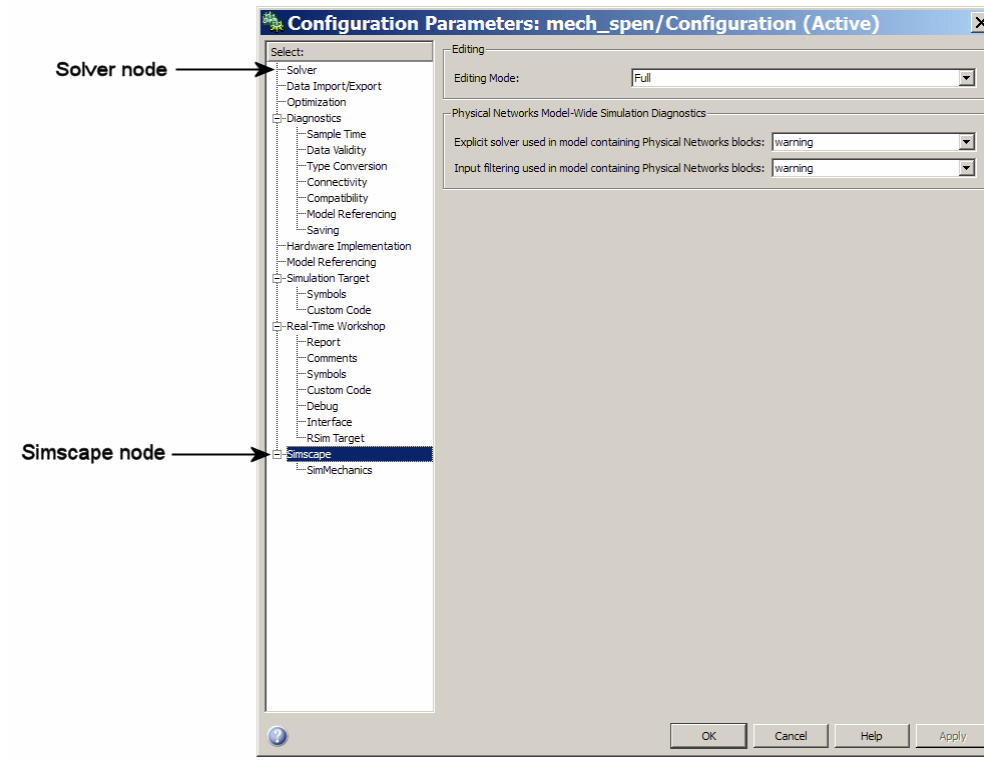
Model-Wide Settings via Simulink and Simscape Software

Mechanical and general settings for an entire model are located in the Simulink Configuration Parameters dialog, accessed through the Simulink **Simulation** menu. Every node in this dialog is relevant to controlling your model's simulation, including visualization. See the Simulink documentation for more details about this dialog.

At a minimum, you need to check and possibly adjust the settings in the **Solver** node and the **Simscape** node, with its **SimMechanics** subnode, before running a mechanical model:

- The active **Editing** area of the **Simscape** node allows you to choose the Simscape software editing mode. To change this setting, see "Using the Simscape Editing Mode" on page 2-20.
- The SimMechanics-specific controls appear on the **SimMechanics** subnode of the **Simscape** node. It has two active areas, **Diagnostics** and **Visualization**.

  - For more about configuring simulation diagnostics, see “Avoiding Simulation Failures” on page 2-17.
  - For more about configuring visualization, see “Setting Up Visualization” on page 2-22.
- The choice and configuration of the solver are Simulink settings, located on the **Solver** node. This node has two active areas, **Simulation time** and **Solver options**.
  - “Configuring a Simulink Solver” on page 2-16 contains the basic information to get you started.
  - To optimize solver settings for better simulation, see “Improving Performance” on page 2-32.
  - For general information about the Simulink solvers, see the Simulink documentation.



Simulink® Configuration Parameters Dialog (Simscape™ Node Shown)

SimMechanics Default Settings Not Changed If SimMechanics Blocks Are Absent

If you have the SimMechanics product installed, any model you build will display the **SimMechanics** subnode under the **Simscape** node. However, if you then build a model that does *not* include any SimMechanics blocks, any nondefault SimMechanics settings you make in the **SimMechanics** subnode will *not* be saved in that model. Upon saving, closing, and reopening the model, the SimMechanics settings will revert to their defaults.

Configuring Methods of Solution

In this section...

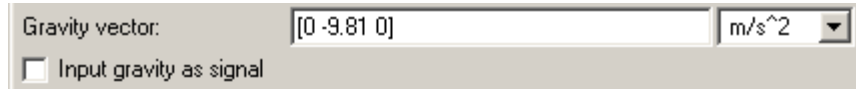
About Mechanical and Mathematical Settings

In this section, you choose and configure the settings necessary to simulate mechanical motion with a SimMechanics model.

To gain a better understanding of how SimMechanics software solves for mechanical motion, see "How SimMechanics Software Works" on page 2-24 and "Improving Performance" on page 2-32.

Defining Gravity

The most basic aspect of a machine's environment is the gravitational acceleration it experiences. You control a machine's gravity in the **Parameters** tab of its Machine Environment dialog.



Setting a Constant Gravitational Acceleration

A uniform gravity field is applied to the motion of every machine. The default is a constant vector of `[0 -9.81 0]` with units of meters/seconds^2 and x-, y-, and z-components, respectively.

You can change this value to a different constant vector by modifying the entry in the **Gravity vector** field of the **Parameters** tab. You can change the units by using the units pull-down menu.

Introducing Gravity as an External Simulink Signal

In addition to constant gravity, you can apply a time-varying, spatially uniform, gravity vector through a Simulink signal. You enable this option by selecting the **Input gravity as signal** check box in the **Parameters** tab.

Once you make this selection, the Machine Environment block acquires a Simulink inport to accept this Simulink signal. The signal must be a three-component vector. You can still change the units through the pull-down menu.

Choosing Your Machine's Dimensionality

In general, you simulate machine motion in all three spatial dimensions. If a machine can move in only two dimensions, however, ignoring the third dimension makes the simulation more efficient. By default, the simulation automatically determines whether your machine moves in all three or only two dimensions and optimizes the simulation accordingly.

You can override this default by requiring simulation in either three or two dimensions. You choose the simulation dimension of a machine in the **Machine dimensionality** pull-down menu of the **Parameters** tab of the Machine Environment dialog. If you attempt to simulate a three-dimensional machine in two dimensions, the simulation stops with an error.

Requirements for Two-Dimensional Simulation

Your machine must meet certain criteria before you can require simulation in two dimensions:

- The prismatic primitives must define a set of parallel planes.

- The revolute primitives must rotate about axes perpendicular to the prismatic planes.

The bodies of a two-dimensional machine do not all have to lie in a single plane, but they should slide and rotate only in parallel planes.

Code Generated from Two-Dimensional Models

Code generated from simulations restricted to two-dimensional motion is also restricted to two-dimensional motion. See "Restrictions on Two-Dimensional Simulation" on page 2-44.

Blocks That Require Three-Dimensional Simulation

The SimMechanics library contains certain blocks that, if you use them in a machine, require you to simulate in three dimensions.

- Any Joint block with more than two prismatic primitives, more than one revolute primitive, or any spherical primitives
- Disassembled Joints
- Massless connectors

Choosing an Analysis Mode

You can analyze motion in a SimMechanics model with these analysis types.

Analysis Result	Analysis Type
Motion that results from applying forces	Forward dynamics
Steady-state motion	Trimming
Effect of slightly perturbing the motion	Linearization
Forces required to produce a specified motion	Inverse dynamics

The **Parameters** tab of the Machine Environment dialog allows you to choose the analysis mode you want to simulate in. You make this choice via the **Analysis mode** pull-down menu. In the case of linearization, use the **Linearization** tab to set the size of the small perturbations. See Chapter 3, "Analyzing Motion" for detailed instructions and examples concerning the motion analysis modes.

By choosing one of these analysis modes, you implement the type of motion analysis you want.

Analysis Type	Analysis Mode	Description
Forward dynamics	Forward Dynamics	Computes the positions and velocities of a system's bodies at each time step, given the initial positions and velocities of its bodies and any forces applied to the system.
Linearization	Forward Dynamics	Computes the effect of small perturbations on system motion through the Simulink `linmod` command.
Trimming	Trimming	Enables the Simulink `trim` command to compute steady-state solutions of system motion.
Inverse dynamics (open-loop)	Inverse Dynamics	Computes the forces required to produce a specified velocity for each body of an open-loop system.
Inverse dynamics (closed-loop)	Kinematics	Computes the forces required to produce a specified velocity for each body of a closed-loop machine.

Forward Dynamics Mode

Use this mode to simulate a model that represents the initial positions and velocities of the system's bodies and the forces on those bodies.

Run these examples in the Forward Dynamics mode:

- "Running a Demo Model"
- "Creating a Simple Mechanical Model" and "Creating a Closed-Loop Mechanical Model"

Consider also the many examples of Chapter 1, "Modeling Mechanical Systems".

Trimming Mode

Use this mode to allow you to run the Simulink `trim` command on your model. The `trim` command allows you to find steady-state solutions for your model.

Trimming mode inserts a subsystem and an output port at the top level of your model. These blocks output signals corresponding to the constraints on the system represented by your model. Configure the `trim` command to find equilibrium points where the constraint signals are zero. This ensures that the equilibrium points found by the `trim` command satisfy the constraints on the modeled system.

See "Trimming Mechanical Models" on page 3-18 for examples of using this mode to find the equilibrium points of a mechanical system.

To Linearize a Machine's Motion

You can determine the effect of small perturbations on system motion by linearizing your machine. To linearize, set the analysis mode to Forward Dynamics and run the Simulink `linmod` command on your model.

You can fix the size of the perturbation or let the simulation find an optimal perturbation for you. Enter these settings in the **Linearization** tab of the Machine Environment dialog.

See "Linearizing Mechanical Models" on page 3-32 for examples of using this mode to find the effect of small perturbations on mechanical motion.

Inverse Dynamics Mode

Use this mode to simulate an open-loop system whose model specifies the velocity of every degree of freedom of every body at every time step.

See "Inverse Dynamics Mode with a Double Pendulum" on page 3-8 for an example of using this mode to find the forces on an open-loop system.

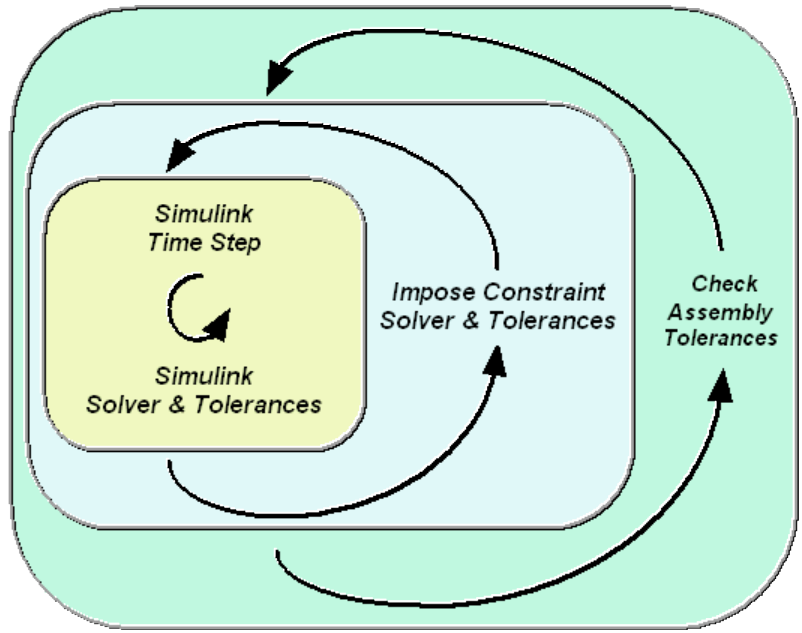
Kinematics Mode

Use this mode to simulate a closed-loop machine whose model specifies the velocity of every independent degree of freedom at every time step. The tolerancing constraint solver is recommended in this mode. (See "Maintaining Constraints" on page 2-12.)

See "Kinematics Mode with a Four Bar Machine" on page 3-13 for an example of using this mode to find the forces on a closed-loop machine.

Hierarchy of Solvers and Tolerances

Simulating your SimMechanics model is a cooperative effort between SimMechanics software and Simulink. A SimMechanics simulation interprets your machine's purely mechanical aspects through machine assembly and a constraint solver. Simulink controls the purely mathematical aspects of the simulation through your chosen Simulink solver. Together, they try to harmonize your choices of Simulink solver and solver tolerances, constraint solver and solver tolerances, and assembly tolerances in this dynamic hierarchy:



The next three sections discuss these choices in top-down order:

- "Controlling Machine Assembly" on page 2-12
- "Maintaining Constraints" on page 2-12
- "Configuring a Simulink Solver" on page 2-16

Controlling Machine Assembly

The linear and angular assembly tolerances specify the precision with which

- A model must specify the initial locations and angles of a machine's joints.
- A simulation must solve the initial positions and angles of a machine's unassembled joints.

The **Parameters** tab of a machine's Machine Environment dialog allows you to change the default assembly tolerances in the **Linear assembly tolerance** and **Angular assembly tolerance** fields. You can also adjust the linear and angular units in the respective pull-down menus.

For more about machine assembly and assembly tolerances, see "Modeling Degrees of Freedom" on page 1-19 and specifically, "Modeling Disassembled Joints" on page 1-34.

How Assembly Tolerances Work

A SimMechanics simulation checks the locations and angles of a machine's assembled joints when it initializes the model and later in the simulation. If any of the joint locations or angles fails to meet the corresponding assembly tolerances, Simulink halts the simulation and displays an error message. If this happens, you should check your machine to ensure that it specifies the locations and angles of its assembled joints to the precision specified in the **Parameters** tab. If not, either change the locations and angles that fail to meet the assembly tolerances or increase the tolerances themselves.

Assembly tolerances can also be violated during the course of a simulation by insufficiently accurate constraint and motion solvers. See "Maintaining Constraints" on page 2-12 and "Configuring a Simulink Solver" on page 2-16.

Maintaining Constraints

If your model contains implicit or explicit constraints on a machine's motion, the SimMechanics simulation uses a constraint solver to find a solution for the motion that satisfies those constraints.

This section describes how the constraint solvers work and what you need to decide to make proper use of them. These constraint choices and settings for a machine are found on the **Constraints** tab of its Machine Environment dialog.

The simulation imposes constraints when it initializes the model, then later checks if the constraints remain satisfied during the simulation. If any of the degrees of freedom (DoFs) fail to satisfy the constraint tolerances, Simulink halts the simulation and displays an error message. If this happens, you should either switch to a looser constraint solver or increase the constraint tolerances (if you have manual control of the constraint tolerance).

Constraints can also be violated during the course of a simulation by an insufficiently accurate Simulink solver. See "Configuring a Simulink Solver" on page 2-16.

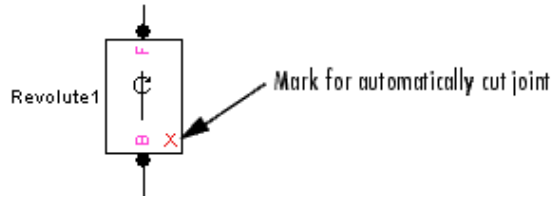
Origins of Mechanical Constraints

The need to impose constraints on a machine's motion arises in two ways, explicit or implicit. In either case, motion is restricted to a subspace of DoFs.

- Imposing a time-independent or time-dependent mechanical constraint on a system's DoFs. This requires you to insert a Constraint or Driver block that restricts the motion represented by a Joint. See "Constraining and Driving Degrees of Freedom" on page 1-38.
- Cutting closed loops in a SimMechanics block diagram. Each closed loop is cut at one Joint, Constraint, or Driver block. The simulation internally replaces the cut block with an implicit constraint equivalent to the original closed loop. See "Cutting Machine Diagram Loops" on page 1-46.

Marking Automatically Cut Joints. Selecting the **Mark automatically cut joints** check box in the **SimMechanics** node of your model's Configuration Parameters dialog causes Simulink to mark the icons of any Joint blocks in closed loops that it cuts during simulation of the model.

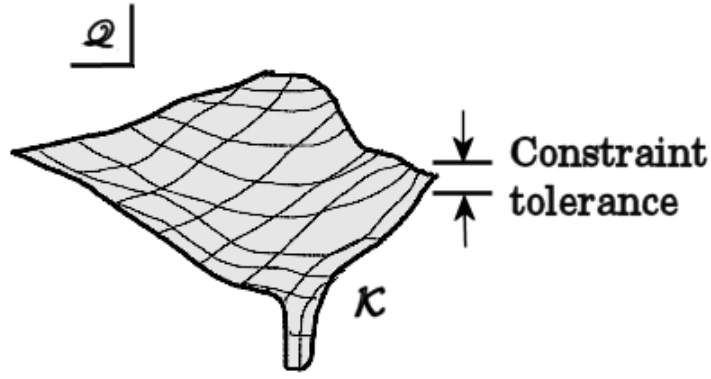
By default, the check box is not selected.



Projecting the Motion on to the Constraint Manifold

The space of motion allowed to the DoFs by the constraints, or *constraint manifold* $\mathcal{K}$, is a subspace of the full space $\mathcal{Q}$ of DoFs. ($\mathcal{Q}$ includes both coordinates and their velocities.) A SimMechanics simulation initializes your model by projecting the initial state of its machines on to $\mathcal{K}$. During simulation, the constraints ensure that the motion remains on $\mathcal{K}$, and Simulink solves for the motion only in the constrained subspace.

The projection cannot be done with infinite precision, but only within the constraint tolerance. Your constraint settings determine the method and precision of projection.



Constraint Manifold as a Subspace of DoFs

Identifying and Eliminating Redundant Constraints

It is possible for you or the simulation to overspecify constraints. Simulation proceeds if the extra or redundant constraints are consistent with the others,

but having redundant constraints always runs the risk of inconsistency, which leads to the simulation halting with errors.

Several checks identify and eliminate redundant constraints, both at the start of and during the simulation.

- You can enable a warning to indicate if a small perturbation to the model initial state changes the number of constraints.
- You can enable a warning to indicate if your model is subject to redundant constraints, whether they conflict or not.
- You can specify how similar constraints have to be before they are treated as redundant, or you can let the simulation decide for you.

See "Configuring SimMechanics Simulation Diagnostics" on page 2-18 and the **Constraints** tab of the Machine Environment block.

Comparing and Choosing Constraint Solvers

Each constraint solver has advantages and disadvantages relative to the others, subject to the fundamental tradeoff of accuracy and speed.

Constraint Solver	Tolerance	Computational Cost	Accuracy	Simulation Speed
Stabilizing	Dynamic attractor	Lowest	Lowest	Fastest
Tolerancing	User-controlled	Intermediate	Intermediate	Intermediate
Machine Precision	Tolerance ~ `eps`	Highest	Highest	Slowest

Stabilizing Constraint Solver

This solver adds a self-correcting term to the equations of motion that stabilizes the solution by causing it to evolve toward, rather than drift away from, the constraint manifold $\mathcal{K}$ It is the least accurate of the constraint solvers.

SimMechanics simulations use this solver by default. It is typically faster than the other solvers, but it can settle into a solution that exceeds the machine's assembly tolerances. If assembly tolerance errors occur during the simulation, use one of the other constraint solvers instead.

Tolerancing Constraint Solver

This solver finds the system's motion while imposing the constraints to the tolerance that you specify. Specifically, the solver stops refining the solution when the difference between two successive solutions satisfies the condition

$$|error| < \max(|rtol * x|, atol)$$

where *error* is the difference between successive solutions, *rtol* is the relative constraint tolerance, *x* is the motion to be solved, and *atol* is the absolute constraint tolerance.

Use this solver you plan to run the simulation in Kinematics mode. It is more accurate than the stabilizing solver, but less accurate than the machine precision solver, with a computational cost between the two.

Setting Constraint Tolerances. If you use the tolerancing solver, the constraint tolerances maintained during simulation are under your control. You can view and change the constraint tolerances in the **Constraints** tab of the Machine Environment dialog.

Machine-Precision Constraint Solver

This solver imposes the constraints to the numerical precision of the computer on which the simulation is running. Select this solver if you want to obtain the most accurate simulation permitted by the computer, regardless of simulation time or computational cost. It is the most accurate of the solvers and typically the slowest.

Configuring a Simulink Solver

A SimMechanics model uses one of the ordinary differential equation (ODE) solvers of Simulink to solve a system's equations of motion, typically in tandem with a constraint solver (see "Maintaining Constraints" on page 2-12).

Simulink provides an extensive suite of ODE solvers that represent the most advanced numerical techniques available for solving differential equations in general and equations of motion in particular. The **Solver** node of your model's Configuration Parameters dialog allows you to select any of these solvers for use by Simulink in solving the model's dynamics. See the Simulink documentation for more details about choosing a Simulink solver.

- By default, Simulink uses a variable-step solver, whose accuracy is controlled by setting its absolute and relative tolerances.
- You can also use a fixed-step solver, whose accuracy is controlled by setting the time step.

See "Improving Performance" on page 2-32 for further details on variable- versus fixed-step solvers.

Setting Simulink Solver Tolerances

By default, Simulink automatically determines the absolute tolerance used by ODE solvers. The resulting tolerance might not be small enough for a mechanical system, particularly a nonlinear or chaotic system. Try running a simulation with the relative tolerance set to `1e-3` (the default) and the absolute tolerance set to `1e-4`. Then increase the tolerances if the simulation takes too long or decrease them if the solution is not sufficiently accurate.

Solver Tolerances and Stiction

If your model contains one or more Joint Stiction Actuator blocks, you must also take into account the velocity thresholds of these blocks when setting the absolute tolerance of the ODE solver. If the absolute tolerance of the solver is greater than a joint's velocity threshold, the simulation might never detect the locking or unlocking of a joint. To prevent this from happening, set the absolute tolerance to be no more than 10% of the size of the smallest stiction velocity threshold in your model.

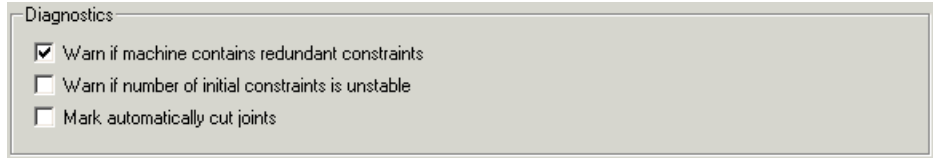
Avoiding Simulation Failures

You can anticipate and avoid many types of simulation failure by the following adjustments. You make them in the **SimMechanics** subnode of your model's Configuration Parameters dialog and the **Constraints** tab of a machine's Machine Environment dialog.

See "How SimMechanics Software Works" on page 2-24 and "Troubleshooting Simulation Errors" on page 2-26 for further information about identifying and recovering from simulation errors. See "Maintaining Constraints" on page 2-12 for more about constraints.

Configuring SimMechanics Simulation Diagnostics

Certain SimMechanics diagnostics help you understand and, if necessary, troubleshoot simulation problems. You can adjust these diagnostics in the **Diagnostics** area of the **SimMechanics** subnode.



Warning on Redundant Constraints. Selecting the **Warn if machine contains redundant constraints** check box triggers a warning if there are more constraints than necessary in your model. This situation by itself does not cause simulation errors. But in certain configurations, too many constraints might lead to conflicts and thus to errors during the course of the simulation.

By default, the check box is selected.

Warning on Unstable Constraints in Initial State. Selecting the **Warn if number of initial constraints is unstable** check box triggers a warning if small changes to your model's initial state leads to changes in the number of constraints. In certain configurations, this instability can lead to too few or too many (conflicting) constraints on your system and prevent the simulation from finding a solution for the motion.

By default, the check box is not selected.

Handling Motion Singularities

At certain simulation times, one or more degrees of freedom in a mechanical system might change quickly compared to the others. If these sudden, quick motions are too fast compared to the slower motions, the Simulink solver has difficulty finding an accurate solution in a reasonable simulation time. Imposing constraints on the motion often exacerbates this problem. In extreme cases, the simulation can stop with an error.

You can alleviate these motion singularities by selecting the **Use robust singularity handling** on the **Constraints** tab of the Machine Environment

dialog. This option requires extra computation whether or not singularities exist. Select it only if you cannot find a Simulink solver that solves your model in a reasonable amount of time without it.

See “Maintaining Constraints” on page 2-12 and “Configuring a Simulink Solver” on page 2-16 for more discussion of motion singularities and their relationship to the Simulink solvers.

Starting Visualization and Simulation

In this section...

About Simscape and Visualization Settings

After you have considered and adjusted SimMechanics mechanical and mathematical settings, discussed in "Configuring Methods of Solution" on page 2-6, you should review Simscape and visualization settings before proceeding to simulation. Open your model's Configuration Parameters dialog from its **Simulation** menu.

Using the Simscape Editing Mode

The Simscape node of the Configuration Parameters dialog contains the **Editing** area and **Editing Mode** pull-down menu. Select the editing mode here, either `Full` or `Restricted`. The default is `Full`.

- The Full mode allows you to open, simulate, change, and save models that contain SimMechanics blocks, without restriction. It requires the SimMechanics product to be installed and a SimMechanics license.
- The Restricted mode allows you to open, simulate, and save models that contain SimMechanics blocks, without requiring a SimMechanics license, as long as the SimMechanics product is installed. In this mode, you can also change a limited set of SimMechanics block dialog parameters.

For more information about Simscape editing modes, see the Simscape documentation.

Editing Block Parameters in Restricted Mode

When you open a SimMechanics model in Restricted editing mode, you cannot change certain block parameters in the block dialogs. The general editing rules for Restricted mode are:

- You can edit dialog fields that contain numerical values or variables.
- You cannot change pull-down menu settings.
- You cannot change check box selections.

Exceptions to the Restricted Mode Editing Rules

There are exceptions to the general block parameters editing rules in Restricted mode.

Machine Environment Block. The Machine Environment dialog is unrestricted in Restricted mode (including pull-down menus), except for:

- **Analysis mode** pull-down menu
- **Input gravity as signal** check box

Editing Parameter Tables in Dialogs. Certain block dialogs use tables to organize parameter fields. You cannot edit such parameters in Restricted mode. The block dialog components affected are:

- Body coordinate systems tabs (**Position** and **Orientation**) in the Body dialog
- **Actuation** area in the Joint Initial Condition Actuator dialog
- **Primitives** tab in the Joint Spring & Damper dialog
- **Axes** tab in any Joint dialog

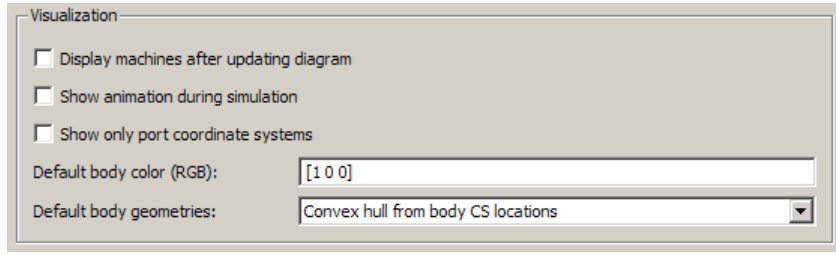
To work around these restrictions, place a workspace variable name (instead of a numerical value) in these parameter fields while editing in Full mode. Then in Restricted mode, you can change the value of the workspace variable, although you cannot change the dialog field entry itself.

Setting Up Visualization

Above the level of individual Body blocks, configuring visualization requires entering settings at the machine and model level. See the *SimMechanics Visualization and Import Guide* for complete information about visualization.

Visualization Settings for an Entire Model

You enter the visualization settings for an entire model in the **Visualization** area of the **SimMechanics** subnode of the Configuration Parameters dialog. Model-wide visualization is disabled by default.



To start visualization, you must select at least one of the first two check boxes:

- **Display machines after updating diagram** for static visualization
- **Show animation during simulation** for dynamic animation

For the others settings, consult “Introducing Visualization and Animation” in the visualization guide.

Every machine within your model inherits the model-wide body color and geometry settings. However, you can override these defaults on a per-machine and per-body basis.

Visualization Settings for Each Machine in a Model

You can choose whether and how to visualize a specific machine in your model through the **Visualization** tab of its Machine Environment dialog. A single window displays all selected machines in a model.

Machine-specific visualization is enabled by default. You can override model-wide default body geometry and color settings on each machine individually.

Visualization Settings for Each Body in a Machine

You can choose how to visualize a specific body in a specific machine through the **Visualization** tab of its Body dialog. You can override machine-wide default body geometry and color settings on each Body individually.

Visualization Settings in the SimMechanics Visualization Window

The SimMechanics visualization window itself contains all other visualization controls.

Starting the Simulation

Once you configure the Simulink and SimMechanics settings to simulate a mechanical system, you can run your model.

As the simulation proceeds, you might encounter warnings, errors, and unexpected or unsatisfactory results. Consult these sections to learn how to identify errors and improve your simulation.

- "How SimMechanics Software Works" on page 2-24
- "Troubleshooting Simulation Errors" on page 2-26
- "Improving Performance" on page 2-32
- "Generating Code" on page 2-38
- "Limitations" on page 2-42

How SimMechanics Software Works

In this section...

About Machine Simulation

This brief overview of how SimMechanics simulation works should help you to construct models and understand errors. "Troubleshooting Simulation Errors" on page 2-26 discusses fixing errors.

The machine simulation sequence has four major phases, described below. The first two occur before machine motion actually starts. The premotion machine configurations (home, initial, and assembled) are discussed in "Kinematics and the Machine's State of Motion" and in their respective "Glossary" entries.

Model Validation

The simulation first checks your data entries from the dialogs and the local connections among neighboring blocks. It then validates the Body coordinate systems; the joint, constraint, and driver geometries; and the model topology. Body positions and orientations defined purely by Body dialog entries constitute the *home configuration*.

Machine Initialization

The simulation next checks the assembly tolerances of Joints that you manually assembled.

The simulation then cuts each closed loop once. An equivalent implicit, or invisible, constraint replaces each cut Joint, Constraint, or Driver block. The simulation checks all constraints and drivers for mutual consistency and eliminates redundant constraints. It also checks whether a small perturbation

to the initial state changes the number of constraints. Such a singularity might lead, during machine motion, to violation of the constraints.

Any Joint Initial Condition Actuators now impose initial positions and velocities, changing body geometries from their dialog box configurations as necessary and transforming the machines from their home configurations to their *initial configurations*. The simulation then finds an assembly solution for disassembled joints and initializes them in position and velocity, defining the *assembled configuration*. Assembly tolerances are checked again.

A "sticky" joint primitive, actuated by a Joint Stiction Actuator, can be in one of three stiction modes: locked, waiting, or unlocked. Iterating through non-time-increment simulation steps (algebraic loop), The simulation finds a mutually consistent set of stiction modes for all sticky joints.

Force Analysis and Motion Integration

In Forward Dynamics or Trimming analysis mode, the simulation begins the solution of machine motion by applying and integrating external forces and torques, stepping in simulation time. It maintains assembly, constraint, and solver tolerances and checks constraint and driver consistency. It also detects whether, within each Joint block, distinct joint primitive axes align and destroy one or more independent DoFs. Such an event is a joint axis singularity.

In Inverse Dynamics and Kinematics modes, the simulation now applies motion constraints, drivers, and actuators to find the machine motion and derive forces and torques. It also checks tolerances and consistency and detects singular alignment of joint primitives.

Stiction Mode Iteration

If stiction is present, the simulation checks at each time step whether the sticky joints transition from one stiction mode to another, then checks for mutual consistency of locked and unlocked sticky joint primitives across the whole model. Non-time-increment simulation steps (algebraic loops) are necessary here.

Troubleshooting Simulation Errors

In this section...

About Simulation Errors

SimMechanics simulations can stop before completion with one or more error messages. This section discusses generic error types, and most errors and error-fixing strategies fall into broad categories. These groupings are reflected in the keywords occurring in the error messages. These sections summarize these groupings.

The previous sections, "Configuring Methods of Solution" on page 2-6 and "How SimMechanics Software Works" on page 2-24, should be useful for identifying and tracing errors. Many common errors also appear in "Representing Machines with Models" on page 1-2 and "Validating Mechanical Models" on page 1-85.

"Improving Performance" on page 2-32 discusses strategies that can prevent errors.

Data Validation Errors

Every numerical entry you make in a SimMechanics model must be a real numerical expression or MATLAB® equivalent. Spatial vectors are 3-vectors, such as `[3 4 5]`. Spatial tensors are 3-by-3 matrices, such as rotation matrices and the inertia tensor.

Tip You can specify a two-dimensional curve in the Point-Curve Constraint block with 2-vectors.

Ground and Body Geometry Errors

Every machine must have a least one Ground block. Every Body block must have at least one Body CS, defined at the body's center of gravity (CG). You must directly or indirectly define the Body coordinate systems (CSs) of a machine relative to a Ground or to World. You cannot enter cyclic (circular) Body CS definitions. The Body CS definitions must separately satisfy these criteria in the **Position** and **Orientation** tabs of the Body dialog.

For example, defining CS3 relative to CS2, defining CS2 relative to CS1, then defining CS1 relative to CS3, results in a definition that is both cyclic and missing any reference to a Ground or World. You can break the cycle by referencing CS1 to a Ground or to World.

To be displayed in visualization, a Body must be connected to at least one Joint that is connected to the rest of the machine. You cannot visualize with equivalent ellipsoids a body whose principal inertial moments do not satisfy the *triangle inequalities*. (See "About Body Color and Geometry: Default, Standard, and Custom".)

Joint Geometry Errors

The geometric configuration of joints, constraints, and drivers can conflict with assembly requirements and restrictions on certain blocks.

Assembly Tolerances Violated

Assembled joints must satisfy assembly tolerances on their connected Body CSs at all times. Disassembled joints assembled at model initialization must also satisfy assembly tolerances during the simulation. (See "Controlling Machine Assembly" on page 2-12.)

Zero Massless Connector Distance

The initial distance between two Body CS origins connected by a massless connector must be nonzero. The massless connector holds the distance between two Body CS origins constant during motion.

Composite Joints: Restrictions Among Primitives

Certain composite Joint blocks place restrictions on their primitive joint axes. For example, Bearing must have its prismatic axis `P1` aligned to its third revolute axis `R3`.

Block Connection and Topology Errors

General rules on how to connect SimMechanics blocks are discussed in Chapter 1, "Modeling Mechanical Systems". In particular, consult these sections of that chapter:

- "Representing Machines with Models" on page 1-2
- "Validating Mechanical Models" on page 1-85

Some restrictions are properties of individual blocks, as explained in their reference pages. See the SimMechanics block reference.

Motion Inconsistency and Singularity Errors

Inconsistencies in motion arise from misapplication of constraints, drivers, and actuators, from conflicting stiction requirements, and incorrect simulation dimensionality.

Motion simulation errors often occur because of *singularities* or dividing by very small numbers. Simulink solvers can integrate certain singularities, at a cost. Others, like loss of a degree of freedom (DoF), can be fatal. See "Maintaining Constraints" on page 2-12, "Configuring a Simulink Solver" on page 2-16, and "Smoothing Motion Singularities" on page 2-34 and the Machine Environment block reference.

Zero Masses and Moments of Inertia

A body moving on a prismatic axis must have nonzero mass if you actuate it with forces. A body rotating about a revolute axis or pivoting about a

spherical must have nonzero inertial moments about the axis or pivot if you actuate it with torques. If you want a massless rigid body, consider using a Massless Connector from the Joints/Massless Connectors sublibrary.

Note You can use point bodies (nonzero mass but zero moments) in SimMechanics models, if the connected revolute axes and spherical pivots are dislocated from the body. Although the moments are zero about a point body's CG, the displacement of the body from the axis or pivot shifts the moments from zero to nonzero values.

Alignment of Primitives — Coincidence of Identical Bodies

Within a single Joint block, two distinct prismatic axes or two distinct revolute axes should never align during the simulation. If either occurs, a translational or rotational DoF is lost, and the simulation cannot determine the subsequent motion. An example of such singularities is "gimbal lock." Two of the three revolute primitive axes in the Gimbal block become parallel, reducing the number of independent DoFs in the Joint from three to two.

Two or more physically identical bodies (having the same masses and inertia tensors) should never coincide in space.

No Degrees of Freedom

Your machine cannot move if it has no degrees of freedom. Each Constraint, Driver, and motion-actuating Actuator block you add to a machine reduces the number of independent DoFs. (See "Counting Model Degrees of Freedom" on page 1-89.) Cure such errors by removing one or more of these blocks from your machine, until you have at least one independent DoF.

Incorrect Machine Dimensionality

You cannot run a three-dimensional machine with a simulation restricted to two dimensions. See "Choosing Your Machine's Dimensionality" on page 2-7.

Redundant Constraints

Some constraints can restrict what another constraint is already restricting. If redundant constraints are present and in conflict, fix these errors by

identifying and removing the redundancies. If the simulation misidentifies one or more redundant constraints, adjust the redundant constraint tolerance. See “Maintaining Constraints” on page 2-12.

Violated Constraints

Some machine motions or simulations might not be able to maintain assembly tolerances at a particular simulation step while simultaneously satisfying the constraints. One or more joints might become disassembled. Any one of these conditions leads to errors.

You can correct this situation in several ways. First, identify the joint, constraint, or driver causing the error and examine its physical configuration when the error occurs to isolate the conflict. Then try any combination of these steps:

- Decrease the Simulink solver tolerances or the step size.
- Switch to a more robust Simulink solver.
- Decrease the constraint solver tolerances.
- Increase the redundant constraint tolerance.
- Switch to the machine precision constraint solver.
- Increase the assembly tolerances.

See “Maintaining Constraints” on page 2-12 and “Configuring a Simulink Solver” on page 2-16.

Conflicting Actuators

You cannot put more than one actuator on a joint primitive.

Exceptions You can simultaneously place an initial condition actuator and a force/torque actuator on a joint primitive.

The Joint Stiction Actuator block does accept an input signal for nonfrictional forces/torques, which the block adds to the stiction.

Sticky Joints in Conflict

If your machine has two or more stiction-actuated ("sticky") joints, a conflict among them can put the simulation into an infinite loop and prevent determination of the machine motion. Or one locked joint can prevent the other joints, sticky or not, from moving. The machine stops moving.

For example, one sticky joint becomes unlocked and requires the other to lock, which then requires the first to lock.

Remove these conflicts by removing one or more stiction actuators or by changing the Joint Stiction Actuator locking thresholds.

Analysis Mode Errors

Certain restrictions apply to the analysis modes presented in "Choosing an Analysis Mode" on page 2-8. Consult individual analysis modes for more:

- "Finding Forces from Motions" on page 3-7
- "Trimming Mechanical Models" on page 3-18
- "Linearizing Mechanical Models" on page 3-32

Improving Performance

In this section...

Optimizing Mechanical and Mathematical Settings

SimMechanics software is a general-purpose mechanical simulator. With it, you can model and simulate many types of machines with very different behaviors. In some cases, the settings you use for "well-behaved" machines are not optimal for more-difficult-to-simulate systems. Simulink and SimMechanics software give you great freedom to change the mechanical and mathematical settings used in your simulations. Use this flexibility to avoid simulation errors and optimize performance, subject to the fundamental tradeoff between speed and accuracy. This section explains techniques for achieving these goals.

Also consult

- "Configuring Methods of Solution" on page 2-6 and "Troubleshooting Simulation Errors" on page 2-26 to learn about simulation settings and correcting and avoiding simulation failures
- "Generating Code" on page 2-38 to learn about speeding up simulations by generating and compiling code from your models

Simplifying the Degrees of Freedom

In general, the more degrees of freedom (DoFs) you add to your model, the slower the simulation.

Eliminating Unnecessary Degrees of Freedom

Under certain circumstances, a model can contain DoFs not practically necessary to predict system behavior. For example, a subsystem might contain very light masses whose motion is almost completely determined by the heavier masses in the system and that have almost no inverse influence on the larger system.

Consider freezing or eliminating such degrees of freedom from your model in order to speed up the simulation.

Freezing "Fast" and "Slow" Degrees of Freedom

A related distinction can be made between DoFs that change rapidly and those that change slowly. Such systems are "stiff" (literally, in the case of a stiff spring that oscillates at a very high frequency) and often hard to simulate accurately in a reasonable time.

One approach to improving the speed is to selectively freeze certain DoFs.

1. Freeze or eliminate the "fast" DoFs and simulate only the "slow" DoFs.
2. Freeze the "slow" DoFs in some representative configuration and simulate the motion of only the "fast" DoFs.

Such a split simulation between "fast" and "slow" DoFs can isolate important features of the system behavior, while ignoring unimportant features.

Caution Splitting DoFs between "fast" and "slow" sets and simulating the two sets separately neglects coupling between the two sets of DoFs. Only a full simulation can capture such coupling.

See "Solving Stiff Systems" on page 2-35 for a different approach to handling speed mismatches among DoFs.

Removing Stiction Actuators

Stiction requires computationally expensive algebraic loops. If possible, remove Joint Stiction Actuator blocks from your model to speed it up.

Simulating in Two Dimensions

If your machine moves in only two dimensions, not three, it qualifies for the SimMechanics two-dimensional simulation option. By reducing the linear and rotational directions from three to two and three to one, respectively, this option can noticeably improve simulation performance.

See "Choosing Your Machine's Dimensionality" on page 2-7.

Adjusting Constraint Tolerances

Maintaining constraints on a system's DoFs is a major and computationally expensive part of a simulation. If your simulation seems to run slowly or stops with constraint errors, especially when the mechanism passes through certain configurations, consider relaxing the constraint tolerances and/or solver. This step generally speeds up the simulation, although it also makes the simulation less accurate. Decreasing the tolerances increases the accuracy of the simulation but can increase the time required to simulate the model.

To view and change these settings in your machine, see "Maintaining Constraints" on page 2-12 and the Machine Environment block reference.

Smoothing Motion Singularities

Singularities in a system's equations of motion can dramatically slow down a standard Simulink solver or even prevent it from finding a solution to a system's equations of motion. Because mechanical motion can become singular, you have the option of *robust singularity handling,* which works together with your selected solver to solve singular equations of motions efficiently. This feature allows Simulink in many cases to simulate models that otherwise cannot run or cannot be solved in a reasonable time. To enable robust singularity handling, see "Avoiding Simulation Failures" on page 2-17.

Exact singularities are recoverable if they form isolated configurations that can be avoided by perturbing the initial state or "stepping over" them during simulation. In that case, the neighborhood of the exact singularity is quasi-singular and appropriate for robust singularity handling. If the machine has a whole neighborhood of continuously related singular configurations, motion in that neighborhood cannot be simulated. For examples of typical singularities, see "Motion Inconsistency and Singularity Errors" on page 2-28.

Avoiding Singular Initial Configurations

Avoid starting a machine in a singular configuration. Its subsequent motion violates assembly tolerances, as the simulation incorrectly removes one or more necessary constraints. Common singular configurations include these:

- The machine can move in two or three dimensions, but starts in exactly one or two dimensions, respectively.
- Two or more identical bodies spatially coincide in position and orientation.

Work around an initial singularity by slightly misaligning the singular joint axes or slightly displacing the coincident bodies, within assembly tolerances, before starting the simulation.

The **SimMechanics** node of the Configuration Parameters dialog allows you to enable simulation warnings for possible singular initial configurations. See "Avoiding Simulation Failures" on page 2-17.

Changing the Simulink Solver and Tolerances

The Dormand-Prince solver (`ode45`) that Simulink uses by default works well for many mechanical systems. But if your simulation seems to be slow and/or inaccurate you should consider changing the solver and/or adjusting the solver's relative and absolute tolerances. Chaotic and highly nonlinear systems especially require experimentation with different solvers and tolerances to obtain optimal results.

Consult the Simulink documentation for more about choosing Simulink solvers and tolerances.

Solving Stiff Systems

The default Simulink solver typically requires too much time to solve systems that are *stiff*, that is, have bodies moving at widely differing speeds or have many discontinuities in their motions. An example of a stiff system is a pair of coupled oscillators in which one body is much lighter than the other and hence oscillates much more rapidly. Any of the Simulink stiff solvers might require significantly less time to solve a stiff system.

See the Simulink documentation on solvers and simulation for more about stiff solvers.

Real-Time Simulation and Ignoring Motion Details with Fixed-Step Solvers

For most mechanical systems, variable time-step solvers are preferable. Fixed time-step solvers, depending on the size of the time step, often fail to resolve certain motion details.

Using a fixed-step solver can be advantageous in some cases, however:

- If you want to ignore unimportant motion details. Ignoring them can speed up your simulation, especially for a larger time step.
- If you are simulating in real time with generated code. Fixed-step solvers are typically, but not exclusively, the norm for real-time simulation.

For such cases, choose one of Simulink's fixed-step solvers and select the largest time step that produces reasonable simulation results.

Most of Simulink's fixed-step solvers are explicit. For stiff systems and larger time steps, an implicit solver such as the `ode14x` fixed-step solver can be superior to an explicit solver in speed and accuracy.

Adjusting the Time Step in Real-Time Simulation

A real-time simulation using code generated and compiled from your model must keep up with the actual mechanical motion. To this end, you must ensure that the solver time step is greater than the computation time needed by your compiled model.

To meet this condition, you might have to increase the time step or decrease the computation time. Increasing the time step often requires removing the model's "fast" DoFs. Decreasing the computation time requires simplifying your model. You can do this most easily by removing DoFs and/or constraints. See "Simplifying the Degrees of Freedom" on page 2-32.

Reference

[1] Moler, C. B., *Numerical Computing with MATLAB*, Philadelphia, Society for Industrial and Applied Mathematics, 2004, Chapter 7.

Generating Code

In this section...

About Code Generation from SimMechanics Models

You can use SimMechanics software with Real-Time Workshop® to generate stand-alone C or C++ code from your mechanical models and enhance simulation speed and portability. Certain features of Simulink also make use of generated or external code. This section explains code-related tasks you can perform with your SimMechanics models.

Code versions of SimMechanics models typically require fixed-step Simulink solvers, which are discussed in "Improving Performance" on page 2-32. Some SimMechanics features are restricted when you translate a model into code. See "Limitations" on page 2-42.

Note Code generated from SimMechanics models is intended for rapid prototyping and hardware-in-the-loop applications. It is not intended for use as production code in embedded controller applications.

SimMechanics software shares most of the same code generation features as Simscape software. This section describes code generation features specific to SimMechanics software. Consult the Simscape documentation for general information on code generation and Physical Modeling.

Using Code-Related Products and Features

With Simulink, Real-Time Workshop, and xPC Target™ software, using several code-related technologies, you can link existing code to your models and generate code versions of your models.

Code-Related Task	Component or Feature
Link existing code written in C or other supported languages to Simulink models	Simulink S-functions to generate customized blocks
Speed up Simulink simulations	Accelerator mode Rapid Accelerator mode
Generate stand-alone fixed-step code from Simulink models	Real-Time Workshop software
Generate stand-alone variable-step code from Simulink models	Real-Time Workshop Rapid Simulation Target (RSIM)
Convert Simulink models to code and run them on a target PC	Real-Time Workshop and xPC Target software
Generate blocks representing a Simulink models or subsystems	S-function Target*
Generate code for designated models or subsystems	Model Reference Accelerator Mode

* S-function Target is supported with SimMechanics models or subsystems, but not with Simscape software. Converting a SimMechanics subsystem to an S-function block allows you to run a model with Simulink alone.

How SimMechanics Code Generation Differs from Simulink

In general, using the code generated from SimMechanics models is similar to using code generated from Simscape and normal Simulink models. The Simscape documentation discusses the differences between code generation in Simulink and in Simscape.

Limited Set of SimMechanics Tunable Parameters

The major difference between Simscape and SimMechanics code generation is that a few SimMechanics blocks do support a limited set of tunable parameters. Consult “Using Run-Time Parameters in Generated Code” on

page 2-40 and "Most Tunable Parameters Not Supported by SimMechanics Software" on page 2-43 following, as well as the SimMechanics block reference.

Using Run-Time Parameters in Generated Code

When SimMechanics software generates code for a model, it creates a set of code source and header files. This set includes *modelname*.c and *modelname*_data.c, containing all the model's run-time parameters. (For C++, these are .cpp files.) In addition, SimMechanics software generates two files that contain data structures and function prototypes for the SimMechanics blocks alone.

The *modelname*.c file contains all the run-time parameters used in the compiled simulation. *modelname*_data.c and the two special SimMechanics files are auxiliaries to aid in locating and changing the run-time data.

Changing Run-Time Parameters

As with code generated from any Simulink model without parameter inlining, you can change any run-time parameters by modifying their values in the block parameters data structure implemented in *modelname*_data.c. In this data structure, however, SimMechanics block parameters are not associated with their original blocks. Rather, SimMechanics block parameters are grouped together into a single vector associated with the first SimMechanics S-function for each machine in the model.

The data structures and functions found in the special SimMechanics files, rt_mechanism_data.h and rt_mechanism_data.c, allow you to modify SimMechanics block parameters in generated code. The special header file contains a data structure, MachineParameters_*modelname_uniqueid*, for each machine in the model, that includes a field for each block run-time parameter. To modify mechanical run-time parameters,

1. Use the function rt_vector_to_machine_parameters_*modelname_uniqueid* in the special code source file to create an instance of the machine parameters data structure from the vectorized parameters associated with the SimMechanics S-function.

2. Make the necessary modifications to the values in the data structure instance.

3. Use `rt_machine_parameters_to_vector_`*`modelname_uniqueid`* to reconstruct the vectorized parameters from the data structure instance.

4. Recompile your generated code.

Example: Changing a Block Parameter

This code listing is an example of a simple function that updates the mass of the first body in the demo `mech_dpen`. The argument `p` should be a pointer to the parameter vector associated with the SimMechanics S-function. The argument `mass` is the new mass for the first body. You should call this function before model initialization.

```
void update_mech_dpen_parameters(real_T *p, real_T mass)
{
    MachineParameters_mech_dpen_752c07b6 ds;
    /*
     * convert parameter vector into data structure
     */
    rt_vector_to_machine_parameters_mech_dpen_752c07b6(p, &ds);
    /*
     * change the mass of the first body in the double pendulum
     */
    ds.Body.Mass = mass;

    /*
     * convert the data structure back to the parameter vector
     */
    rt_machine_parameters_to_vector_mech_dpen_752c07b6(&ds, p);
}
```

Limitations

In this section...

About SimMechanics and Simulink Limitations

Some Simulink features and tools either do not work with models containing SimMechanics blocks or work only with restrictions. Others work with SimMechanics models but only on the normal Simulink blocks in those models.

Continuous Sample Times Required

The sample times of all SimMechanics blocks are always continuous, and you cannot use them with discrete solvers. You also cannot override the sample time of a nonvirtual subsystem containing SimMechanics blocks.

Restricted Simulink Tools

Certain Simulink tools are restricted in use with SimMechanics software.

- A SimMechanics model with closed loops cannot be linearized with the Simulink `linmod2` command.
- Enabled subsystems can contain SimMechanics blocks. But you should always set the **States when enabling** parameter in the **Enable** dialog to `held` for the subsystem's Enable port.

  Setting **States when enabling** to `reset` is not supported and can lead to simulation errors.

- Simulink configurable subsystems work with SimMechanics blocks only if all of the block choices have consistent port signatures.
- For Iterator, Function-Call, Triggered, and While Iterator nonvirtual subsystems cannot contain SimMechanics blocks.
- An atomic subsystem with a user-specified (noninherited) sample time cannot contain SimMechanics blocks.
- SimMechanics software supports external mode, but without visualization.
- SimMechanics software supports Simulink model referencing, with these restrictions:
  - A SimMechanics model can be referenced only once by another model.
  - SimMechanics software does not support reparameterization in a referencing block.
  - A SimMechanics machine cannot be visualized if it is referenced.

Unsupported Simulink Tool

The Simulink Profiler does not work with SimMechanics models.

Simulink Tools Not Compatible with SimMechanics Blocks

Some Simulink tools and features do not work with SimMechanics blocks:

- Execution order tags do not appear on SimMechanics blocks.
- SimMechanics blocks do not invoke user-defined callbacks.
- You cannot set breakpoints on SimMechanics blocks.
- Reusable subsystems cannot contain SimMechanics blocks.
- You cannot use the Simulink Fixed-Point Tool with SimMechanics blocks.
- The Report Generator reports SimMechanics block properties incompletely.

Most Tunable Parameters Not Supported by SimMechanics Software

You cannot tune most SimMechanics block parameters during simulation.

The exceptions that you can tune are:

- The **Gravity vector** field of the Machine Environment block.
- All three fields under **Parameters** in the Body Spring & Damper block.

Restrictions on Two-Dimensional Simulation

Certain blocks are not supported in two-dimensional simulation mode. These include disassembled joints, massless connectors, and joints that can move in three dimensions. See "Choosing Your Machine's Dimensionality" on page 2-7.

Restrictions with Generated Code

Code generated from models containing SimMechanics blocks has certain limitations.

Stiction-Related Algebraic Loops Disabled

Stiction implemented with Joint Stiction Actuator blocks requires algebraic loops iterated at a single time step to detect discrete events. In generated code versions of models with stiction, the mode iteration to determine joint locking and unlocking instead occurs over multiple time steps, possibly reducing simulation accuracy.

Closed-Loop Limitations

Closed-loop models in certain analysis mode configurations use nonlinear solvers with no upper limit on iterations. Code generated from such models is valid but, in general, not truly "real time." These configurations include:

- Forward Dynamics mode when **Constraint solver type** in the Machine Environment block is set to `Machine Precision` or `Tolerancing`
- Kinematics mode

Restrictions on Code Generated from Two-Dimensional Machines

If you generate code from a model containing one or more machines simulated in two dimensions, the generated code is also restricted to two-dimensional motion. Thus, if you change run-time parameters in the generated code, you

must ensure that the new values do not violate the two-dimensional motion restriction.

The choice of machine dimensionality is either automatic or manual, but this restriction on generated code applies in either case. See “Choosing Your Machine’s Dimensionality” on page 2-7.

Restriction on S-Functions Generated from SimMechanics

You cannot generate code from a SimMechanics model that itself contains one or more S-functions generated from other SimMechanics models.

3

Analyzing Motion

SimMechanics analysis modes allow you to study machine motion beyond the simple forward dynamics integration of forces. This chapter explains how to specify machine motion, then deduce the necessary forces and torques, with the Inverse Dynamics and Kinematic analysis modes. You can also specify a machine steady state and analyze perturbations about any machine trajectory by trimming and linearizing your model, respectively.

Chapter 4, "Motion, Control, and Real-Time Simulation" covers more sophisticated motion analysis and control design techniques applied to more complex systems.

Dynamics of Mechanical Systems

In this section...

About Machine Dynamics

As explained in "Representing Motion", kinematics describes the motion of bodies, while *dynamics* explains the motion in terms of forces and torques. By Newton's laws of motion, the accelerations of the bodies' positions are directly related to the forces and torques applied to the bodies.

You can predict accelerations if you are given the applied forces/torques, or relate known accelerations to the forces/torques that cause them, as this section explains. The section concludes by presenting Newton's laws of dynamics for translational and rotational motion.

The books of Goldstein [1] and José and Saletan [5] present rigid body mechanics in great detail.

About the SimMechanics™ Machine State To perform inverse dynamics, trimming, and linearization tasks, you might need to look at and manipulate the mechanical state of your SimMechanics machine or model.

- The machine state components arise from individual joint primitives in your machine's Joint blocks.
- These components represent relative degrees of freedom between one body and another or between a body and ground.

See the `mech_stateVectorMgr` command reference for more information about constructing and interpreting the machine state.

Forward and Inverse Dynamics

Dynamical equations such as Newton's laws of motion relate cause and effect. In mechanics, the cause is a set of forces and torques applied to the bodies of a mechanical system; the effect is the set of resulting motions. Dynamical equations allow you to analyze motion in either direction:

- In *forward dynamics*, you apply a given set of forces/torques to the bodies to produce accelerations. SimMechanics simulation integrates the accelerations twice to yield the velocities and positions as functions of time.

  A set of *initial conditions* is needed to specify the initial positions and velocities and produce a complete solution for the motion. Initial conditions must be checked for consistency with constraints.

- *Inverse dynamics* starts with given motions as functions of time and differentiates them twice to yield the forces and torques needed to produce the given motions. The given motion functions of time must be checked for consistency with constraints.

You can use SimMechanics analysis modes to analyze mechanical motion in both cases. The mode you choose can depend on the topology of your system.

Analysis Mode	Type of Analysis
Forward Dynamics	Forward dynamics (any topology)
Trimming	Forward dynamics (steady-state motion)
Inverse Dynamics	Inverse dynamics (open topology)
Kinematics	Inverse dynamics (closed topology)

Applying the Motion Modes

For more about motion modes, see these other sections.

- "Simulating and Analyzing Mechanical Motion" is an overview of the SimMechanics analysis modes.
- "Choosing an Analysis Mode" on page 2-8 contains detailed steps to implement these modes in your model.

- The case study "Finding Forces from Motions" on page 3-7 applies inverse dynamics to SimMechanics models.

Forces, Torques, and Accelerations

Newton's second law of motion relates the force on a body, its mass, and the acceleration it experiences as a result of that force. The equivalents for rotational motion are the Euler equations.

Newton's Equations for Translational Dynamics

Let $\boldsymbol{F}_A$ be the net force acting on a body A that has a constant mass m_A and a center of gravity (CG) position $\boldsymbol{x}_A$. Newton's second law, valid for an inertial observer, relates the force on A to the translational acceleration of its CG.

$$\boldsymbol{F}_A = m_A \frac{d^2 \boldsymbol{x}_A}{dt^2}$$

Equivalently, the *linear momentum* $\boldsymbol{p}_A = m_A \boldsymbol{v}_A$ relates to force as $\boldsymbol{F}_A = d\boldsymbol{p}_A / dt$.

In forward dynamics, the force $\boldsymbol{F}_A$ is given and the motion $\boldsymbol{x}_A(t)$ is found by integration, supplemented by initial position and velocity. In inverse dynamics, the motion $\boldsymbol{x}_A(t)$ is given and the force on the body is found. In both cases, the mass must be known.

Euler's Equations for Rotational Dynamics

Rotational motion requires a *pivot*, the fixed center of rotation, and the *angular velocity* vector $\boldsymbol{\omega}$ with respect to that pivot. If $\boldsymbol{r}$ is the position, with respect to the pivot, of any point in a body, the velocity $\boldsymbol{v}$ of that point is $\boldsymbol{v} = \boldsymbol{\omega} \times \boldsymbol{r}$.

The equivalent of the mass of a body in rotational dynamics is the inertia tensor I, a 3-by-3 matrix.

$$I_{ij} = \int_V dV \left[\delta_{ij} |\boldsymbol{r}|^2 - r_i r_j \right] \rho(\boldsymbol{r})$$

The body's mass density $\rho(\boldsymbol{r})$ is a function of $\boldsymbol{r}$ within the body's volume V. The indices i, j range over 1, 2, 3, or x, y, z. Thus

$$I_{xx} = \int_V dV\left[y^2 + z^2\right]\rho(\boldsymbol{r}),\ I_{xy} = \int_V dV\left[-xy\right]\rho(\boldsymbol{r}),\ \text{etc.}$$

The *angular momentum* of a body is $\boldsymbol{L} = \boldsymbol{I}\cdot\boldsymbol{\omega}$. The equivalent of the force on a body in rotational dynamics is the torque τ, which is produced by a force $\boldsymbol{F}$ acting on the body at a point $\boldsymbol{r}$ as $\boldsymbol{\tau} = \boldsymbol{r} \times \boldsymbol{F}$.

The analog to Newton's second law for rotational motion, as measured by an inertial observer, just equates the torque $\boldsymbol{\tau}_A$ applied to a body A, defined with respect to a given pivot, to the time rate of change of $\boldsymbol{L}_A$. That is, $\boldsymbol{\tau}_A = d\boldsymbol{L}_A/dt$. It is easiest to take the pivot as the origin of an inertial coordinate system such as World. Unlike the case of translational motion, however, where the mass m_A remains constant as the body moves, the inertia tensor $\boldsymbol{I}_A$ changes as the body rotates, if it is measured in an inertial frame. There is no simple way to relate $d\boldsymbol{L}_A/dt$ to the angular acceleration $d\boldsymbol{\omega}/dt$.

The common solution to this difficulty is to work in the body's own rotating frame, where the inertia tensor is constant, and take the body's CG as the pivot. Diagonalize the inertia tensor. Since I is real and symmetric, its eigenvalues (I_1, I_2, I_3) (the *principal moments of inertia*) are real. Its eigenvectors form a new orthogonal triad, the *principal axes* of the body. But this frame fixed in the body is not inertial, and the torque-angular acceleration relationship is modified from its inertial form into the *Euler equations:*

$$\begin{aligned}
I_1\dot{\omega}_1 - \omega_2\omega_3\left(I_2 - I_3\right) &= \tau_1 \\
I_2\dot{\omega}_2 - \omega_3\omega_1\left(I_3 - I_1\right) &= \tau_2 \\
I_3\dot{\omega}_3 - \omega_1\omega_2\left(I_1 - I_2\right) &= \tau_3
\end{aligned}$$

The components of the rotational vectors here are projected along the principal axes that move with the body's rotation.

Linearizing the Dynamical Equations

To study a system's response to and stability against external changes, you can apply small perturbations in the motion or the forces/torques to a

known trajectory and force/torque set. SimMechanics software and Simulink provide analysis modes and functions for analyzing the results of perturbing mechanical motion. The later sections of this chapter, demonstrate their use:

- "Trimming Mechanical Models" on page 3-18
- "Linearizing Mechanical Models" on page 3-32

You can perturb Newton's and Euler's laws with a small additional force $\Delta \boldsymbol{F}$ and torque $\Delta \boldsymbol{\tau}$ and determine the associated perturbations in motion, $\Delta \boldsymbol{x}$ and $\Delta \boldsymbol{\omega}$. You can also perturb the system inversely, making small changes to the motion and determining the forces and torques necessary to create those changes.

The perturbed Newton's and Euler's equations are

$$\boldsymbol{F} = m \cdot d^2 \left(\Delta \boldsymbol{x} \right) / dt^2$$

and

$$\begin{aligned} I_1 \Delta \dot{\omega}_1 - \left(\Delta \omega_2 \cdot \omega_3 + \omega_2 \cdot \Delta \omega_3 \right) \left(I_2 - I_3 \right) &= \Delta \tau_1 \\ I_2 \Delta \dot{\omega}_2 - \left(\Delta \omega_3 \cdot \omega_1 + \omega_3 \cdot \Delta \omega_1 \right) \left(I_3 - I_1 \right) &= \Delta \tau_2 \\ I_3 \Delta \dot{\omega}_3 - \left(\Delta \omega_1 \cdot \omega_2 + \omega_1 \cdot \Delta \omega_2 \right) \left(I_1 - I_2 \right) &= \Delta \tau_3 \end{aligned}$$

The vector components of the Euler's equations are projected along the body's moving principal axes.

Linearizing the Constraints

If your model has constraints, you must perturb them as well:

$$g \left(\boldsymbol{x}, \dot{\boldsymbol{x}}, t \right) = 0 \quad , \quad \frac{\partial g}{\partial \boldsymbol{x}} \cdot \Delta \boldsymbol{x} + \frac{\partial g}{\partial \dot{\boldsymbol{x}}} \cdot \Delta \dot{\boldsymbol{x}} = 0$$

Finding Forces from Motions

In this section...

About Inverse Dynamics in SimMechanics Software

The SimMechanics Kinematics and Inverse Dynamics modes enable you to find all the forces on a closed-loop machine or an open machine, respectively, given a model that completely specifies the system's motions. (See "Choosing an Analysis Mode" on page 2-8.) You can use the Forward Dynamics mode to analyze inverse dynamics, but these two alternative modes are more efficient: unlike Forward Dynamics mode, they do not need to compute the positions, velocities, and accelerations of the model's components, because the model specifies them. Consequently, Kinematics and Inverse Dynamics modes take less time than Forward Dynamics to compute the forces on a system. The time saving depends on the size and complexity of the system being simulated.

To use these modes, you must first build a *kinematic model* of the system, one that specifies completely the positions, velocities, and accelerations of the system's bodies. You create a kinematic model by interconnecting blocks representing the bodies and joints of the system and then connecting actuators to the joints to specify the motions of the bodies.

A model does not have to actuate every joint to specify completely the motions of a system. In fact, the model need actuate only as many joints as there are independent degrees of freedom in the system. (See "Counting Model Degrees of Freedom" on page 1-89.) For example, a model of a four bar mechanism need actuate only one of the mechanism's joints, because a four bar mechanism has only one degree of freedom. To avoid overconstraining the model's solution, the number of actuated joints should not exceed the number of independent degrees of freedom.

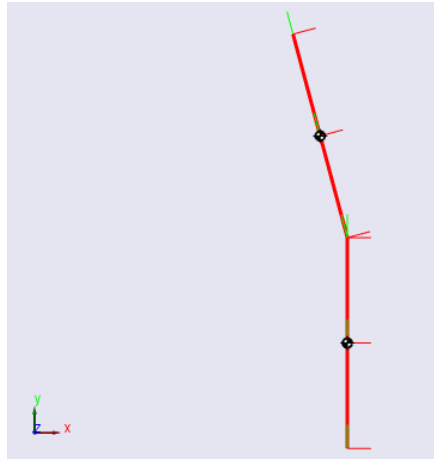
The following sections show how to use the Inverse Dynamics and Kinematics modes to find the forces on the joints of an open- and closed-topology system, respectively.

Warning **Attempting to simulate an overconstrained model causes Simulink to stop the simulation with an error.**

Inverse Dynamics Mode with a Double Pendulum

Caution The Inverse Dynamics mode works only on open topologies and requires motion-actuating every independent DoF (see "Counting Model Degrees of Freedom" on page 1-89).

Consider a double pendulum consisting of two thin rods each 1 meter long and weighing 1 kilogram. The upper rod is initially rotated 15 degrees from the perpendicular.



Suppose that you want the pendulum to follow a certain trajectory. How much torque is required to make the pendulum follow this prescribed motion? Solving this problem entails building a kinematic model of the moving pendulum.

- The model must represent the geometry of the double pendulum and specify its motion trajectory throughout the simulation.
- The model must also measure the *computed torque* on each joint, the torque necessary to reproduce the specified motion.

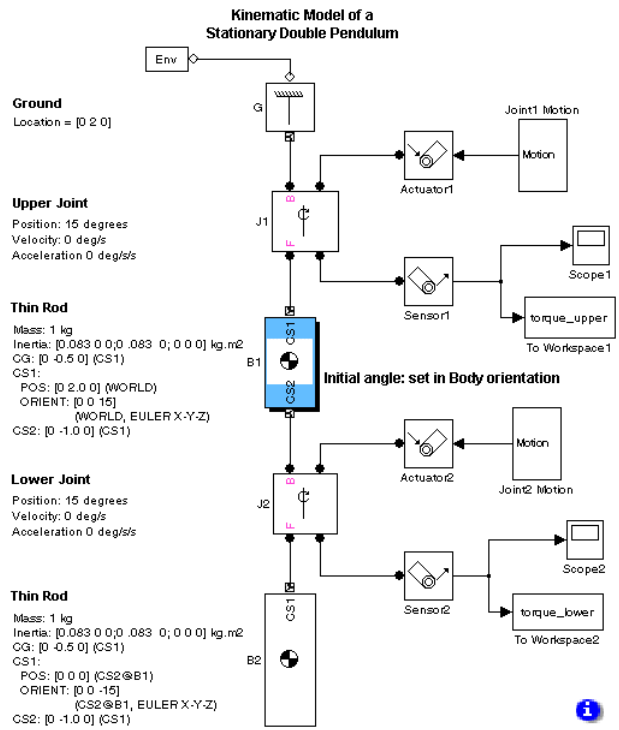
Except in simple cases, you can find these computed torques only as approximate functions of time.

The kinematic model can take different approaches to specifying the initial state of the pendulum.

- One approach uses Body block parameters to specify the initial states.
- Another approach uses Actuator block signals.

Using Body Blocks to Specify Initial Conditions

Open the model `mech_dpend_invdyn1`. It illustrates the Body block approach to modeling initial states.

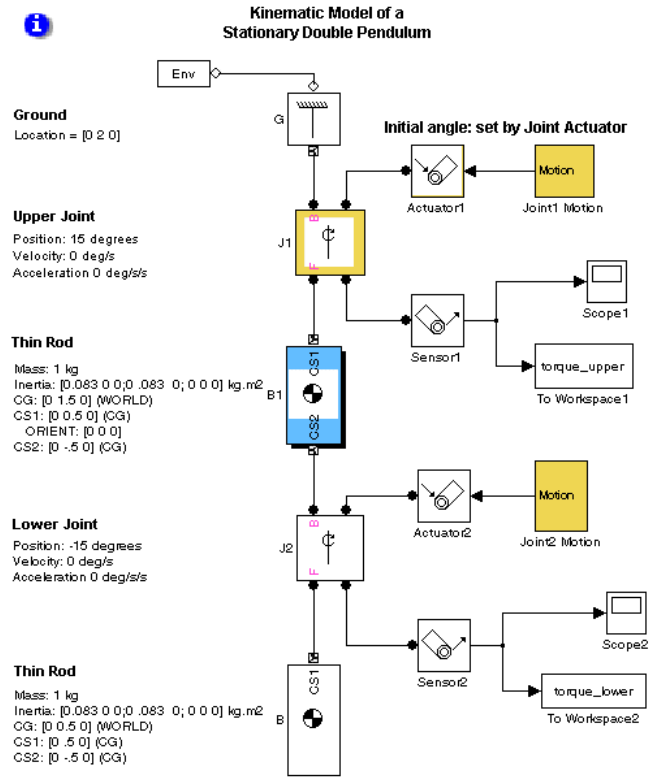


This model represents the pendulum by two Body blocks and two Revolute Joint blocks.

- The CS1 axis of the upper body (B1) of the pendulum is rotated 15 degrees from the perpendicular (see annotation for block B1).
- The coordinate systems for the lower block (B2) are aligned with CS1 of the upper block. The CS1 of B2 is rotated -15 degrees relative to CS1 of B1, i.e., it is perpendicular to the World coordinate system.

Using Actuator Blocks to Specify the Initial States

Open the model `mech_dpend_invdyn2`. It shows the use of Joint Actuator blocks to specify the initial kinematic state. Using actuators to specify the displacement slightly simplifies the configuration of the Body blocks.



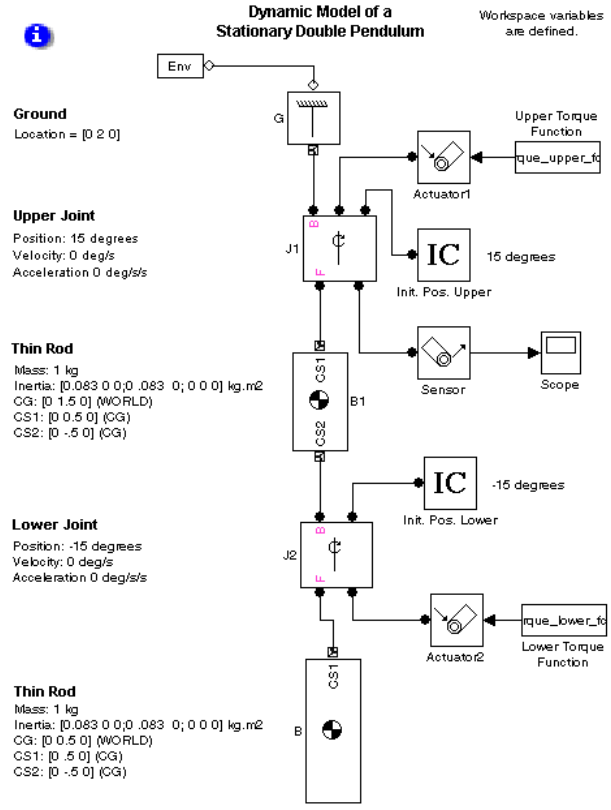
Specifying the Motion and Measuring the Computed Torques

In either model, the Joint Actuator blocks connected to the Joint blocks specify that the upper and lower joints accelerate at two distinct rates, π/2 and -π/4 radians/second^2, respectively. Sensor blocks connected to To Workspace blocks measure the computed torques on the upper and lower joints as MATLAB workspace variables `torque_upper` and `torque_lower`, respectively. These vectors capture the upper and lower computed torques at each major time step. You must simulate either model in Inverse Dynamics mode to compute the joint torques required to maintain the pendulum in its motion.

Using the Computed Torques in Forward Dynamics

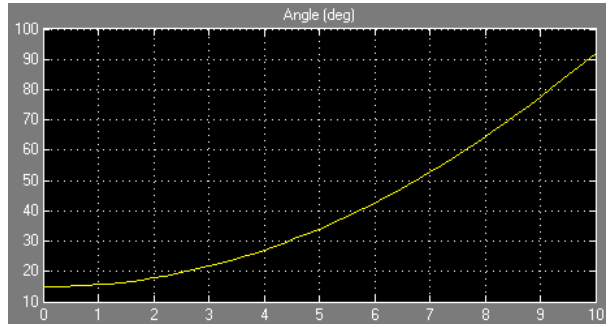
Once you know the computed torques as functions of time, you can verify that these are the correct answers by creating a version of the model that applies the computed torques to the joints and simulating that model in Forward Dynamics mode.

Open the model `mech_dpend_act`. It illustrates a forward dynamics version of the kinematic model that uses the joint actuators to specify the initial angular displacement of the pendulum bodies.



This model uses Initial Condition blocks to specify the initial 15 degree displacement of the upper body from the vertical in the world coordinate system and the corresponding initial -15 degree displacement of the lower body from the vertical in the coordinate system of the upper body. The negative displacement of the lower body is equivalent to positioning it as vertical in the world coordinate system.

From a MAT-file, the model loads the upper and lower torques, `torque_lower_fcn` and `torque_upper_fcn`, as two matrices representing discrete functions of time. Simulating this model in Forward Dynamics mode results in the following display on the upper joint scope.



If the computed torques were known exactly as continuous functions of time in the two inverse dynamics models, this plot would exactly match the upper joint motion in the original models. But the torques are measured only in a discrete approximation, and `mech_dpend_act` does not exactly reproduce the original motion.

Making More Accurate Torque Measurements

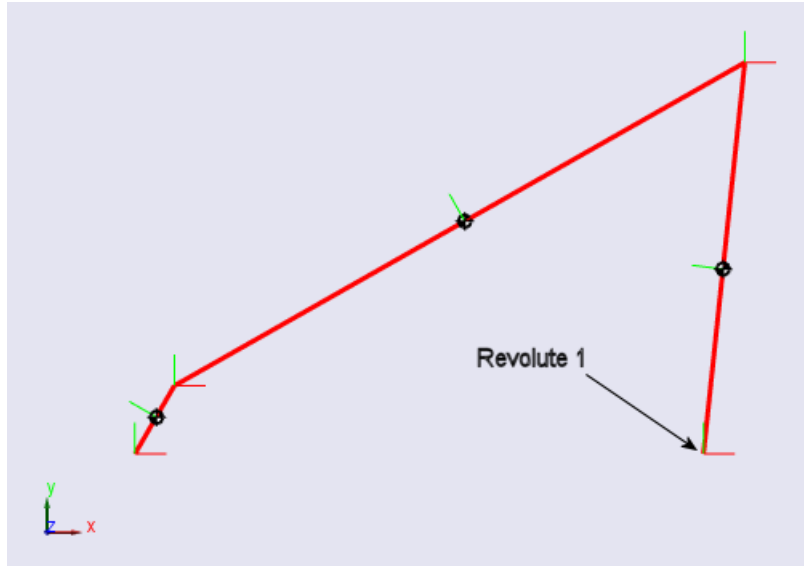
You can achieve better approximations by adjusting Simulink to report sensor outputs in the original models with finer time steps. Refer to the Simulink documentation for more about exporting simulation data and refining output.

Kinematics Mode with a Four Bar Machine

Caution The Kinematics mode works only on closed topologies and requires motion-actuating every independent DoF (see “Counting Model Degrees of Freedom” on page 1-89).

Also, there must be no Joint Stiction Actuators and no nonholonomic constraints.

Consider the four bar system illustrated by the tutorial titled “Creating a Closed-Loop Mechanical Model”. The model is `mech_four_bar`.



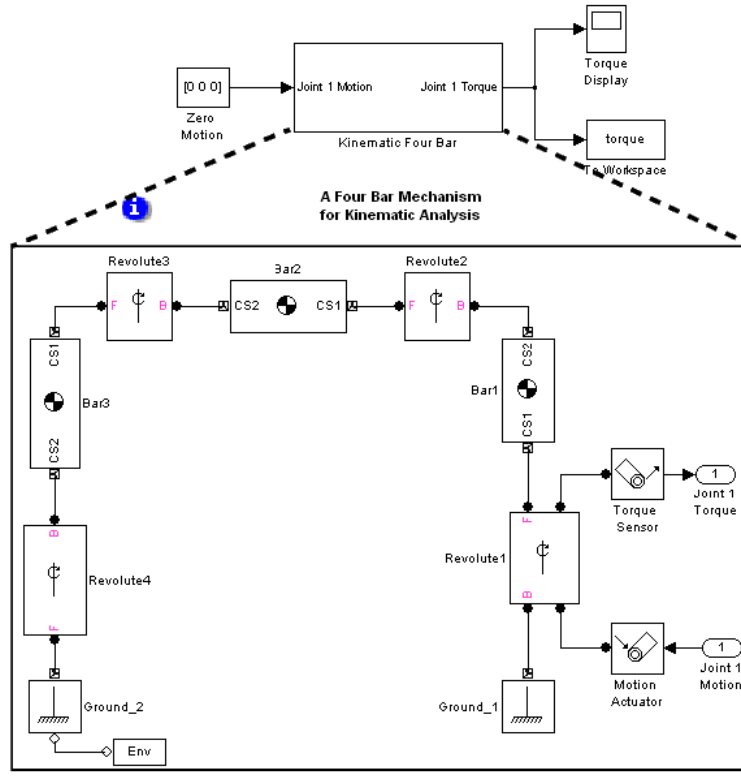
Suppose that you want to keep this system from collapsing under its own weight. Because the four bar has only one degree of freedom, applying a counterclockwise torque to the joint labeled Revolute1 would accomplish this objective. But how much torque is sufficient?

To answer this question, you must build a kinematic model of the stationary four bar system, starting with the tutorial model. The kinematic model must specify how the system moves over time. In this case, the four bar remains stationary. You can use a Joint Actuator to implement this requirement.

Transforming Forward into Inverse Dynamics

Open the model `mech_four_bar_kin`, derived from `mech_four_bar`.

- The model uses a Joint Actuator block driven by a Constant block to specify the motion on the Revolute1 joint. The Constant block outputs a three-element vector that specifies the angular position, velocity, and acceleration, respectively, of the joint as 0.
- The model uses a Joint Sensor block connected to a Scope block to display the resulting torque on the joint and a To Workspace block to save the torque signal to the MATLAB workspace.

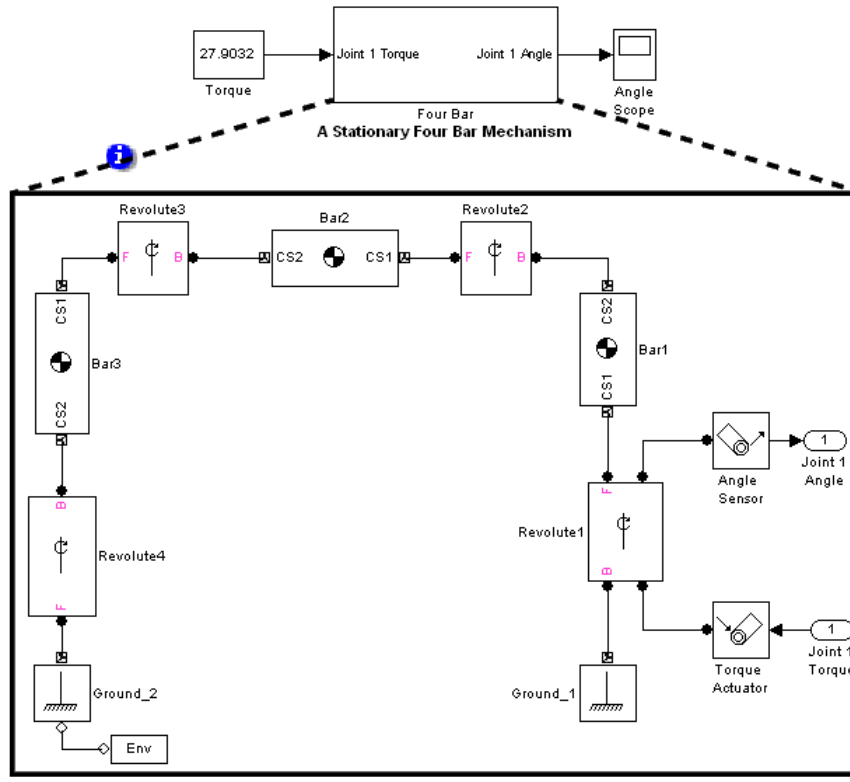


Finding and Checking the Needed Torque

Now obtain and verify the inverse dynamics solution to the question.

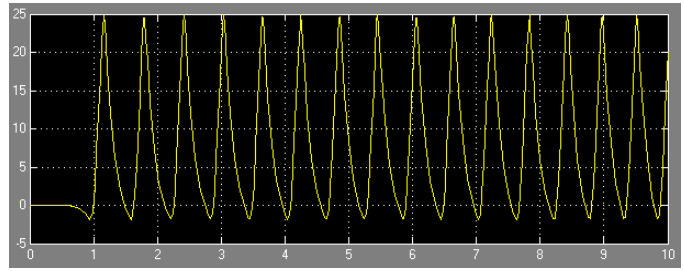
1. Run this model in Kinematics mode. The output reveals that the torque on the Revolute1 joint is 27.9 newton-meters, to the precision of the assembly tolerances specified in the Machine Environment block.

2 To verify that the computed torque is, indeed, the torque required to keep the system stationary, create a forward-dynamics model that applies the computed torque to the Revolute1 joint. Open such a model contained in `mech_four_bar_stat`.



3 Run the model in Forward Dynamics mode, with the Revolute1 Angle Scope open.

The Scope display reveals that the machine does, indeed, remain stationary, although only for about 0.4 seconds. The derived computed force is not exact, and the model is begins nonlinear oscillations after this period.



Tip You can reduce the inaccuracy of the derived computed force by rerunning the mech_four_bar_kin model with more restricted solver, assembly, and constraint tolerances. For the highest accuracy (at greater computational cost), consider shifting to the machine precision constraint solver. See "Configuring Methods of Solution" on page 2-6.

Trimming Mechanical Models

In this section...

About Trimming in SimMechanics Software

Trimming a mechanical system refers to the finding of solutions for inputs, outputs, states, and state derivatives satisfying conditions that you specify beforehand. For example, you can seek steady-state solutions where some or all of the derivatives of a system's states are zero. To use the Simulink `trim` command on a system represented by a SimMechanics model, you must select the SimMechanics Trimming mode (see "Choosing an Analysis Mode" on page 2-8). You must also specify the conditions that the solution must satisfy. The examples following then show you how to trim mechanical models.

Consult the Simulink documentation for more on trimming models. You can also enter `help trim` at the MATLAB command line.

Restrictions on Trimming Mechanical Models

You should avoid using certain SimMechanics or Simulink features when trimming a model.

- A trimmed SimMechanics mechanism must be assembled. Do not use disassembled joints while trimming.

  For more information, see "Modeling Disassembled Joints" on page 1-34.

- You cannot use Driver blocks while trimming a model.
- Joint Initial Condition Actuator blocks in a trimmed SimMechanics model are ignored.
- Do not incorporate events or motion discontinuities in your trimmed model. In particular, do not use SimMechanics Joint Stiction Actuator blocks. Trimming mechanical models with stiction causes an error.

Trimming in the Presence of Motion Actuation

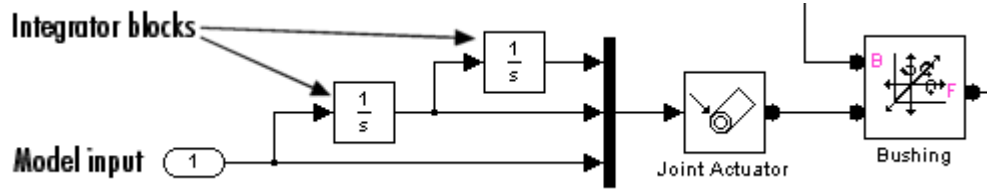
If you want to trim a SimMechanics model containing motion actuators, you must

1. Make the velocity and position/angle parts of the motion actuation signal dependent only on the acceleration signal
2. Make the velocity and position/angle consistent with the acceleration part by use of Integrator blocks. A motion actuation signal is a vector with components ordered as position/angle, velocity, and acceleration, respectively.

   This technique is recommended in "Stabilizing Numerical Derivatives in Actuator Signals" on page 1-49. It is required here.

SimMechanics Trimming mode uses only the acceleration as an independent motion actuation input because it is equivalent to a force or torque. As a consequence, only the acceleration signal can be used as an independent motion actuation input.

A similar restriction holds for model linearization; see "Linearizing in the Presence of Motion Actuation" on page 3-33.

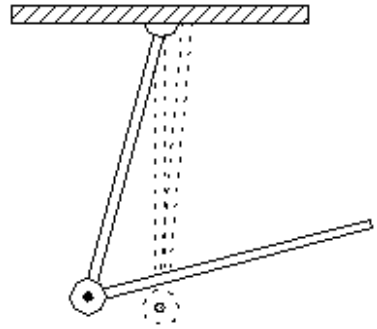


Motion Actuation as a Model Input for Trimming

Motion Actuation as an Indirect Input. You can put your model input port in another part of your model, then feed that input as an acceleration into a motion actuator with a Simulink signal line. You must still derive the velocity and position/angle motion actuation signals in the same way: by integrating whatever signal you use for acceleration once and twice, respectively.

Unconstrained Trimming of a Spring-Loaded Double Pendulum

Consider the following spring-loaded double pendulum.

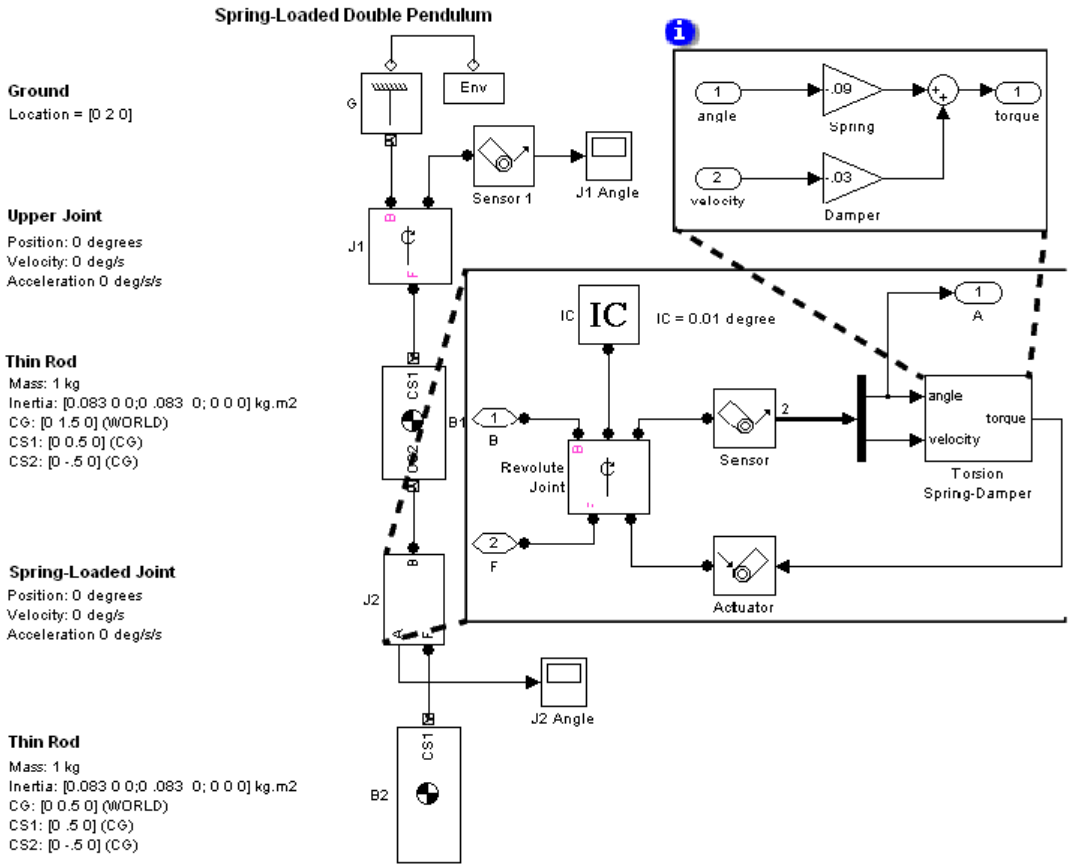


The joint connecting the upper and lower arms of this pendulum contains a torsional spring and damper system that exerts a counterclockwise torque linearly dependent on the angular displacement and velocity of the joint. Suppose that the lower arm is folded upward almost vertically and then allowed to fall under the force of gravity. At what point does the spring-damper system reach equilibrium. That is, at what point does it cease to unfold?

Making an Initial Equilibrium Guess

To find an equilibrium point for the spring-loaded double pendulum,

1 Build a SimMechanics model of the system. This diagram shows an example of such a model, `mech_dpend_trim`.



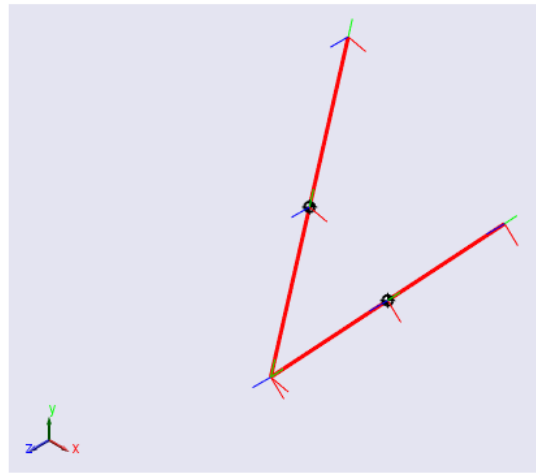
- This model uses Body blocks to model the upper and lower arms of the pendulum and a Revolute Joint block (J1) to model the connection between the pendulum and ground.
- The model uses a Subsystem block (J2) to model the spring-loaded revolute joint between the arms. This subsystem uses a negative feedback loop to model a joint subject to a damped torsional spring by multiplying the angular displacement and velocity of the joint, respectively, by spring and damper constants. The loop sums the

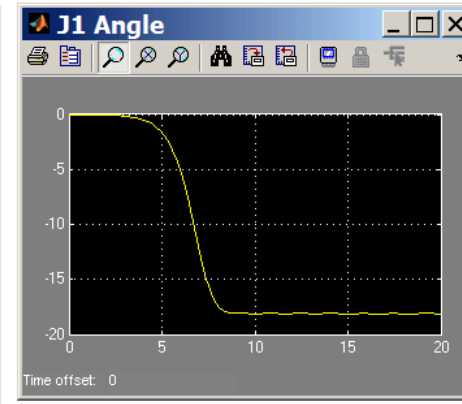
resulting torques and feeds them back into the joint with a Joint Actuator block.

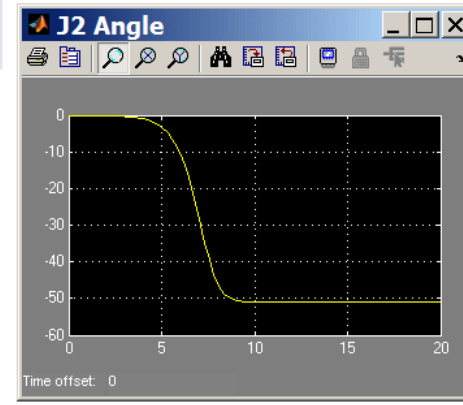
The result is that the joint experiences a torque opposing its motion and proportional to its angular displacement and velocity. You could also model this damped torsional spring with a Joint Spring & Damper block.

The spring and damper constants used here were chosen by running the model with various candidate values and choosing the ones that resulted in a moderate deflection of the pendulum.

2 Run the model in Forward Dynamics mode to estimate an initial guess for the nontrivial equilibrium point of the pendulum.







The simulation reveals that the spring stops unfolding after about 9 seconds; that is, it reaches a steady-state point. At this point the angles of the upper and lower joints are about -18 and -51 degrees, respectively, and the velocities are zero. The `trim` command can find the values of these states precisely.

Analyzing and Initializing the State Vector

Examine the model's state vector and prepare it for use in trimming.

1. Determine the layout of the model's state vector, in order to tell the `trim` command where in the model's state space to start its search for the pendulum's equilibrium point (the point where it stops unfolding). Use the SimMechanics `mech_stateVectorMgr` command to perform this task. Refer to the Ground block, G.

```
StateManager = mech_stateVectorMgr('mech_dpend_trim/G');
StateManager.StateNames

ans =
    'mech_dpend_trim/J2/RevoluteJoint:R1:Position'
    'mech_dpend_trim/J1:R1:Position'
    'mech_dpend_trim/J2/RevoluteJoint:R1:Velocity'
    'mech_dpend_trim/J1:R1:Velocity'
```

   The `StateNames` field of the state vector object returned by `mech_stateVectorMgr` lists the names of the model's states in the order in which they appear in the model's state vector. Thus the field reveals that the model's state vector has the following structure:

```
x(1) = position of lower joint (J2)
x(2) = position of upper joint (J1)
x(3) = velocity of lower joint (J2)
x(4) = velocity of upper joint (J1)
```

2. Determine an initial state vector.

   The initial state vector specifies the point in a system's state space where the `trim` command starts its search for an equilibrium point. The `trim` command searches the state space outward from the starting point, returning the first equilibrium point that it encounters. Thus, the starting

point should not be at or near any of a system's trivial equilibrium points. For the double pendulum, the point [0; 0; 0; 0] (i.e., the pendulum initially folded up and stationary) is a trivial equilibrium point and therefore should be avoided. The initial state vector must be a column vector and must specify angular states in radians.

Often, the choice of a good starting point can be found only by experiment, that is, by running the `trim` command repeatedly from different starting points to find a nontrivial equilibrium point. This is true of the double pendulum of this example. Experiment reveals that this starting point,

```
ix(1) = J2 (lower joint) angle = -35 degrees = -0.6109 radians
ix(2) = J1 (upper joint) angle = -10 degrees = -0.1745 radians
ix(3) = J2 angular velocity = 0 radians/second
ix(4) = J1 angular velocity = 0 radians/second
```

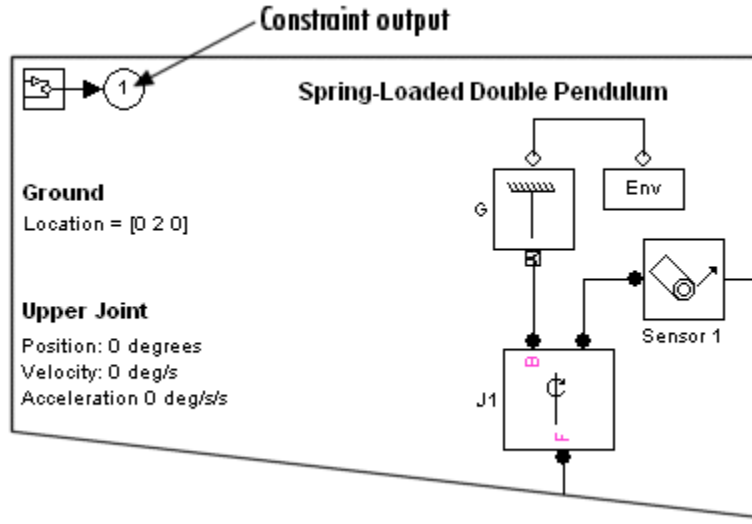
yields a nontrivial equilibrium point.

Caution The `trim` command ignores initial states specified by Joint Initial Condition Actuator blocks. Thus, you cannot use these blocks to specify the starting point for trimming a model. If your model contains IC blocks, create the initial state vector as if the IC blocks did not exist.

Trimming the System to Equilibrium

1 Reset the analysis type to `Trimming` on the **Parameters** tab of the Machine Environment dialog.

This option inserts a constraint subsystem and associated output at the top level of the model. Trimming inserts the constraint output to make the constraints available to the `trim` command. The spring-loaded double pendulum has no constraints. Hence the constraint outport does not output nontrivial constraint data and is not needed to trim the pendulum.



2 Enter the following commands to find the equilibrium point nearest to the starting point.

```
ix = [-35*pi/180; -10*pi/180; 0; 0];
iu = [];
[x,u,y,dx] = trim('mech_dpend_trim',ix,iu);
```

The array `ix` specifies the starting point determined in "Analyzing and Initializing the State Vector" on page 3-23. The array `iu` specifies the initial inputs of the system. Its value is null because the system has no inputs. (Thus the `u` and `y` outputs are null.) In this form, the `trim` command finds a system's steady-state (equilibrium) points, i.e., the points where the system's state derivatives are zero. The array `x` contains the state vector corresponding to the first equilibrium point located by `trim`:

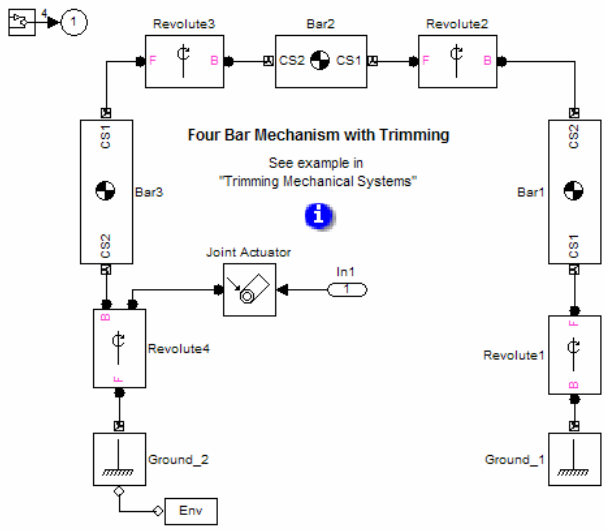
```
x =
    -0.8882
    -0.3165
    -0.0000
     0.0000
```

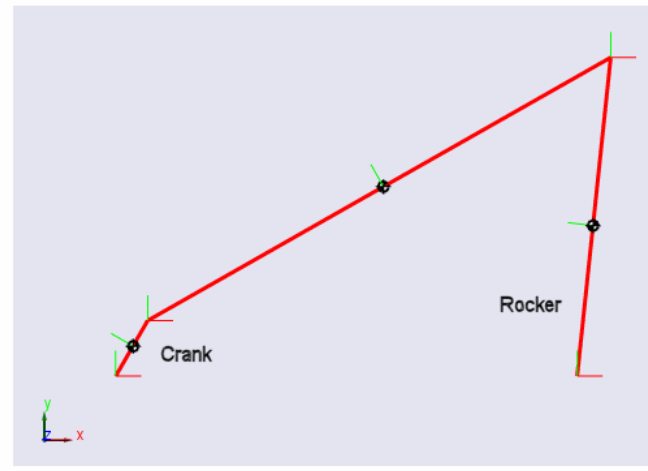
The resulting states are angular positions and velocities expressed in radians. Based on the layout of the model's state vector (determined

previously in “Analyzing and Using the State Vector” on page 3-27) the pendulum reaches equilibrium when its upper joint has deflected to an angle of -18.1341 degrees and its lower joint to an angle of -50.8901 degrees. The system state derivatives `dx` are zero, within tolerances.

Constrained Trimming of a Four Bar Machine

Consider a planar four bar system consisting of a crank, a coupler, and a rocker. The following figure shows a block diagram and a convex hull display of the four bar system. The model is `mech_four_bar_trim`.





This system is constrained by virtue of being a closed loop. Not all the degrees of freedom are independent. (In fact, only one is.) Suppose you want to find the torque required to turn the crank at an angular velocity of 1 radian/second over a range of crank angles. This section outlines the procedure with the `trim` command and the SimMechanics Trimming mode to determine the torque.

Setting Up the Four Bar for Trimming

Reconfigure the model before performing the trim.

1 Cut the closed loop that represents the four bar system at the joint (Revolute1) connecting the rocker to ground (see “Modeling Grounds and Bodies” on page 1-9).

   Manually cutting the rocker joint ensures that the simulation does not cut the four bar loop at the crank joint and thereby eliminate the crank's position and velocity from the system's state vector.

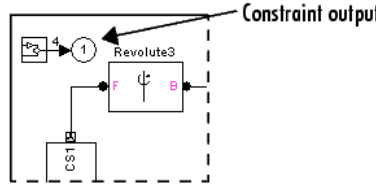
   For instructions and additional information on cutting joints, see “Cutting Machine Diagram Loops” on page 1-46 and “Maintaining Constraints” on page 2-12.

2 Select **Signal Dimensions** from the **Format > Port/Signal Displays** menu.

   Simulink then displays the width of signals on the model diagram and hence enables you to read the number of constraints on the four bar system from the diagram in the next step.

3 Set the analysis mode to `Trimming` in the Machine Environment block.

   Trimming mode then inserts a subsystem and an output block that outputs a signal representing the mechanical constraints on the four bar system. These constraints arise from the closure of the loop.



   The width of the constraint signal (4) reflects the fact that the four bar system is constrained to move in a plane and thus has only four constraints: two position constraints and two velocity constraints.

Analyzing and Using the State Vector

Examine the state vector and prepare it for use in trimming.

1 Reveal the layout of the system's state vector with `mech_stateVectorMgr`:

```
Handle = get_param('mech_four_bar_trim/Revolute2','handle');
StateManager = mech_stateVectorMgr(Handle);
StateManager.StateNames

ans =
    'mech_four_bar_trim/Revolute2:R1:Position'
    'mech_four_bar_trim/Revolute3:R1:Position'
    'mech_four_bar_trim/Revolute4:R1:Position'
    'mech_four_bar_trim/Revolute2:R1:Velocity'
    'mech_four_bar_trim/Revolute3:R1:Velocity'
    'mech_four_bar_trim/Revolute4:R1:Velocity'
```

2 Specify the initial state vector `x0` and the index array `ix`:

```
x0  = [0;0;0;0;0;1];
ix  = [3;6];
```

The array `x0` specifies that the `trim` command should start its search for a solution with the four bar system in its initial position and with the crank moving at an angular velocity (state 6) of 1 radian/second. The array `ix` specifies that the angular position (state 3) and velocity (state 6) of the crank must equal their initial values, 0 radians and 1 radian/second, respectively, at the equilibrium point. It is not necessary to constrain the other states because the four bar system has only one independent position DoF and only one independent velocity DoF.

3 Specify zero as the initial estimate for the crank torque:

```
u0  = 0;
```

4 Require the constraint outputs to be 0:

```
y0  = [0;0;0;0];
iy  = [1;2;3;4];
```

The `y0` array specifies the initial values of the constraint outputs as zero. The `iy` array specifies that the constraint outputs at the solution point must equal their initial values (0). This ensures that the solution satisfies the mechanical constraints on the system.

5 Specify the state derivatives to be trimmed:

```
dx0 = [0;0;1;0;0;0];
idx = [6];
```

The `dx0` array specifies the initial derivatives of the four bar system's states. In particular, it specifies that the initial derivative of the crank angle (i.e., the crank angle velocity) is 1 radian/second and all the other derivatives (i.e., velocities and accelerations) are 0. The `idx` array specifies that the acceleration of the crank at the solution point must be 0; i.e., the crank must be moving at a constant velocity. It is not necessary to constrain the accelerations of the other states because the system has only one velocity DoF.

Note The four bar system has only constraint outputs. If you were trimming a system with nonconstraint outputs, you would have to include the nonconstraint outputs in the initial output vector.

The four bar system also has only mechanical states. If you were trimming a system with nonmechanical Simulink states, you would have to also include those nonmechanical states in the initial state vector.

Trimming the Four Bar

Carry out the trimming and study the output.

1. Trim the system at the initial crank angle to verify that you have correctly set up the trim operation:

   ```
   [x,u,y,dx] = ...
      trim('mech_four_bar_trim',x0,u0,y0,ix,[],iy,dx0,idx);
   ```

   Trim the system over a range of angles.

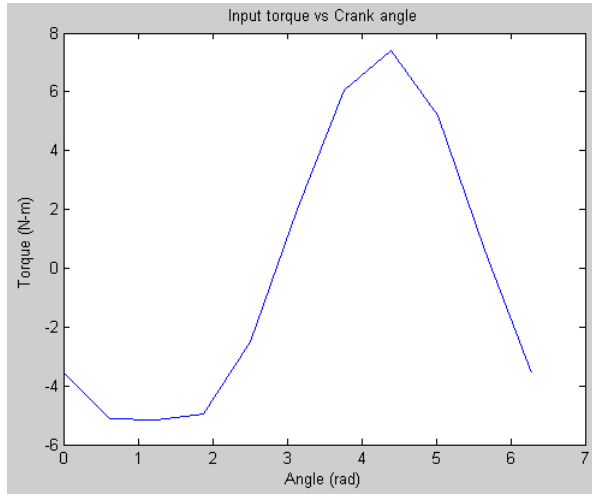
   ```
   Angle = [];
   Input = [];
   State = [];
   dAngle = 2*pi/10;
   Constraint = [];

   for i=1:11;
   ```

```
    x0(3) = (i-1)*dAngle;
    x0(6) = 1;
    [x,u,y,dx] = ...
trim('mech_four_bar_trim',x0,u0,y0,ix,[],iy,dx0,idx);
   disp(['Iteration: ', num2str(i), ' completed.']);
   Angle(i) = x0(3);
    Input(:,i) = u;
    State(:,i) = x;
    Constraint(:,i) = y;
     if (i>3),
         u0 = spline(Angle,Input,Angle(end) + dAngle);
         x0 = spline(Angle,State,Angle(end) + dAngle);
     else
         x0 = x;
         u0 = u;
      end; end;
```

2 Plot the results.

```
figure(1);
plot(Angle,Input);
grid;
xlabel('Angle (rad)');
ylabel('Torque (N-m)');
title('Input torque vs crank angle');
```

The following figure shows the resulting plot.



For More Information About Trimming Closed-Loop Machines

The following section, “Linearizing Mechanical Models” on page 3-32 contains an example, “Closed-Loop Linearization: Four Bar Machine” on page 3-40, of trimming the system in a different way, searching for the stable natural equilibrium of the four bar mechanism.

Linearizing Mechanical Models

In this section...

About Linearization and SimMechanics Software

The Simulink `linmod` command creates linear time-invariant (LTI) state-space models from Simulink models. It linearizes each block separately. You can use this command to generate an LTI state-space model from a SimMechanics model, for example, to serve as input to Control System Toolbox™ commands that generate controller models. The `linmod` command allows you to specify the point in state space about which it linearizes the model (the *operating point*). You should choose a point where your model is in equilibrium, i.e., where the net force on the model is zero. You can use the Simulink `trim` command to find a suitable operating point (see "Trimming Mechanical Models" on page 3-18). By default, `linmod` uses an adaptive perturbation method to linearize model. The Machine Environment dialog allows you to require that `linmod` use a fixed perturbation method instead (see "Choosing an Analysis Mode" on page 2-8). The examples then following illustrate the use of `linmod` to linearize SimMechanics models.

Consult the Simulink documentation for more on "Linearizing Models". You can also enter `help linmod` at the MATLAB command line.

Restrictions on Linearizing Mechanical Models

There are restrictions on how you linearize mechanical models.

- If you specify any joint primitive initial conditions with Joint Initial Condition Actuator blocks, these initial condition values always override any state vector initial values specified via the `linmod` command.

  Joint primitives with JICA blocks are preferentially chosen for the set of independent states in linearization.

- Avoid incorporating discrete events or motion discontinuities in a linearized model. If you include event- or discontinuity-triggering blocks, ensure that the machine does not induce discontinuities as it moves through the linearized regime you are modeling.

  Use of Joint Stiction Actuator blocks in a linearized model causes an error.

- Because closed loops impose constraints on states, you cannot linearize a closed-loop SimMechanics model with the `linmod2` command.

Linearizing in the Presence of Motion Actuation

SimMechanics linearization uses only the acceleration as an independent motion actuation input because it is equivalent to a force or torque. A similar restriction holds for model trimming; see "Trimming in the Presence of Motion Actuation" on page 3-19. As a consequence, the only motion actuation signal that can be set as a model input is the acceleration signal.

If you want to linearize a SimMechanics model containing motion actuators, you must

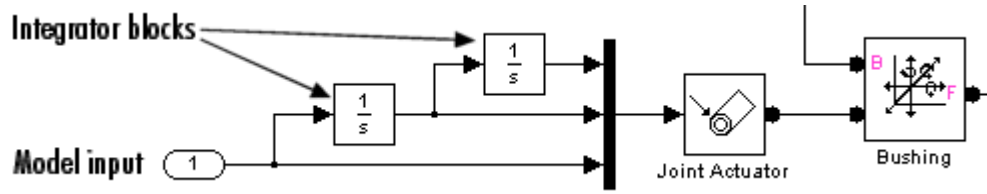
1. Make the velocity and position/angle parts of the motion actuation signal dependent only on the acceleration signal

2. Make the velocity and position/angle consistent with the acceleration part by use of Integrator blocks. A motion actuation signal is a vector with components ordered as position/angle, velocity, and acceleration, respectively.

   This technique is recommended in "Stabilizing Numerical Derivatives in Actuator Signals" on page 1-49. It is required here.

SimMechanics linearization uses only the acceleration as an independent motion actuation input because it is equivalent to a force or torque. As a consequence, only the acceleration signal can be used as an independent motion actuation input.

A similar restriction holds for model trimming; see "Trimming in the Presence of Motion Actuation" on page 3-19.

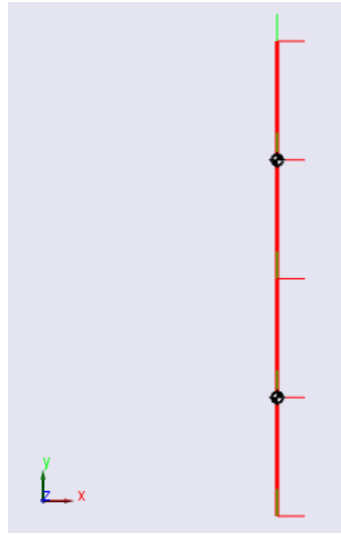


Motion Actuation as a Model Input for Linearization

Linearizing in the Presence of Motion Actuation. You can put your model input port in another part of your model, then feed that input as an acceleration into a motion actuator with a Simulink signal line. You must still derive the velocity and position/angle motion actuation signals in the same way: by integrating whatever signal you use for acceleration once and twice, respectively.

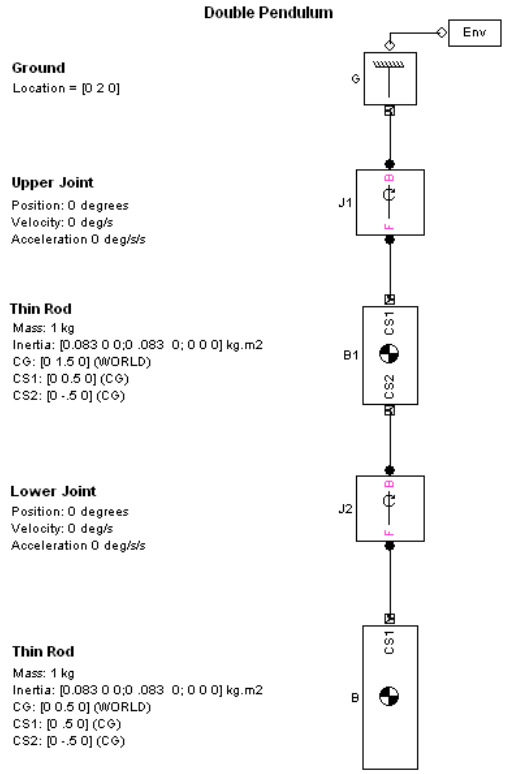
Open-Topology Linearization: Double Pendulum

Consider a double pendulum initially hanging straight up and down.



The net force on the pendulum is zero in this configuration. The pendulum is thus in equilibrium.

Open the `mech_dpend_forw` model.



Linearizing the Model

To linearize this model, enter

```
[A B C D] = linmod('mech_dpend_forw');
```

at the MATLAB command line. This form of the `linmod` command linearizes the model about the model's initial state.

Note Joint initial conditions specified with IC blocks always override any state vector initial values passed to the `linmod` command.

The double pendulum model in this example contains no IC blocks. The initial conditions specified with the `linmod` command are therefore implemented without modification.

Deriving the Linearized State Space Model

The matrices *A, B, C, D* returned by the `linmod` command correspond to the standard mathematical representation of an LTI state-space model:

$$d\boldsymbol{x}/dt = A \cdot \boldsymbol{x} + B \cdot \boldsymbol{u}$$
$$\boldsymbol{y} = C \cdot \boldsymbol{x} + D \cdot \boldsymbol{u}$$

where x is the model's state vector, y is its outputs, and u is its inputs. The double pendulum model has no inputs or outputs. Consequently, only A is not null. This reduces the state-space model for the double pendulum to

$$d\boldsymbol{x}/dt = A \cdot \boldsymbol{x}$$

where

```
A =
         0         0    1.0000         0
         0         0         0    1.0000
 -137.3400   39.2400         0         0
   39.2400  -19.6200         0         0
```

This model specifies the relationship between the state derivatives and the states of the double pendulum. The state vector of the LTI model has the same format as the state vector of the SimMechanics model. The SimMechanics `mech_stateVectorMgr` command gives the format of the state vector as follows:

```
StateManager = mech_stateVectorMgr('mech_dpend_forw/G');
StateManager.StateNames
```

```
ans =
    'mech_dpend_forw/J2:R1:Position'
    'mech_dpend_forw/J1:R1:Position'
    'mech_dpend_forw/J2:R1:Velocity'
    'mech_dpend_forw/J1:R1:Velocity'
```

Right-multiplying A by the state vector $\boldsymbol{x}$ yields the differential state equations corresponding to the LTI model of the double pendulum,

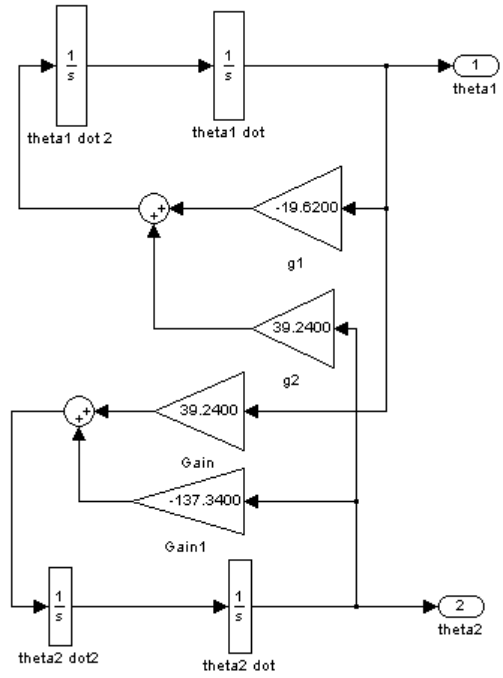
$$\ddot{\theta}_1 = -19.62 \cdot \theta_1 + 39.24 \cdot \theta_2$$
$$\ddot{\theta}_2 = +39.24 \cdot \theta_1 - 137.34 \cdot \theta_2$$

where

θ_1 = position of top joint (J1)
θ_2 = position of bottom joint (J2)

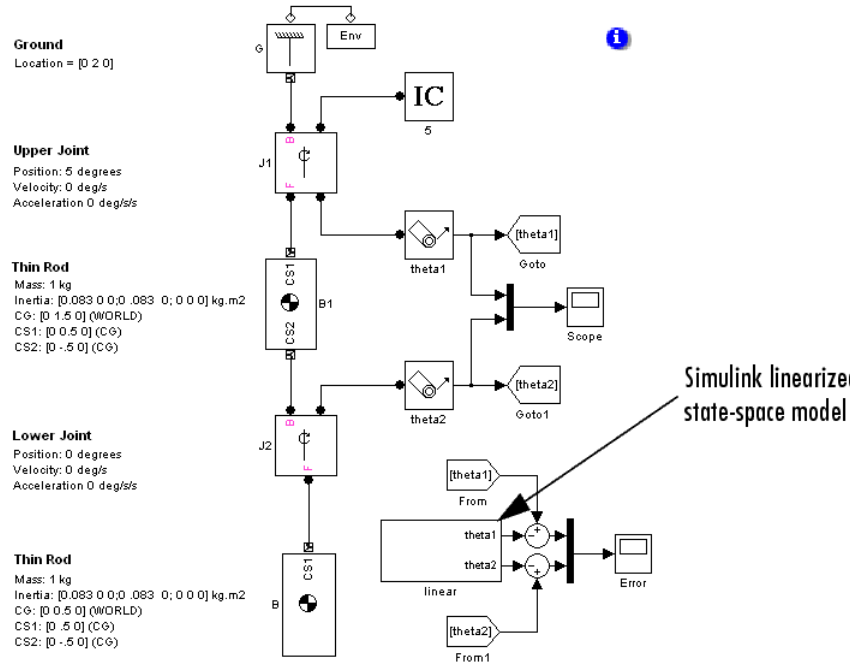
The array of coefficients on the right-hand side of the differential equations represents a matrix of squared frequencies. The eigenvalues of this matrix are the squared frequencies of the system's response modes. These modes characterize how the double pendulum responds to small perturbations in the vicinity of the operating point, which here is the force-free equilibrium.

The following Simulink model implements the state-space model represented by these equations.

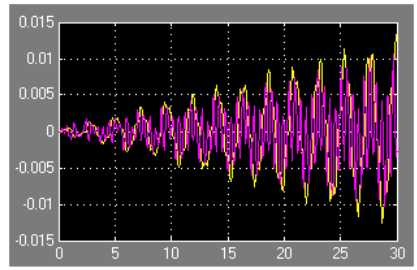


Modeling the Linearization Error

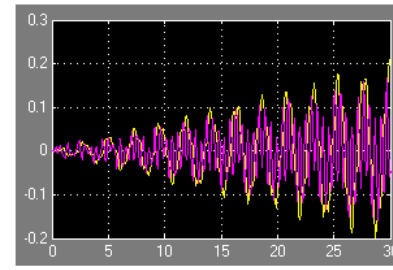
This model in turn allows creation of a model located in `mech_dpend_lin` that computes the LTI approximation error.



Running the model twice with the upper joint deflected 2 degrees and 5 degrees, respectively, shows an increase in error as the initial state of the system strays from the pendulum's equilibrium position and as time elapses. This is the expected behavior of a linear state-space approximation.



2 degrees



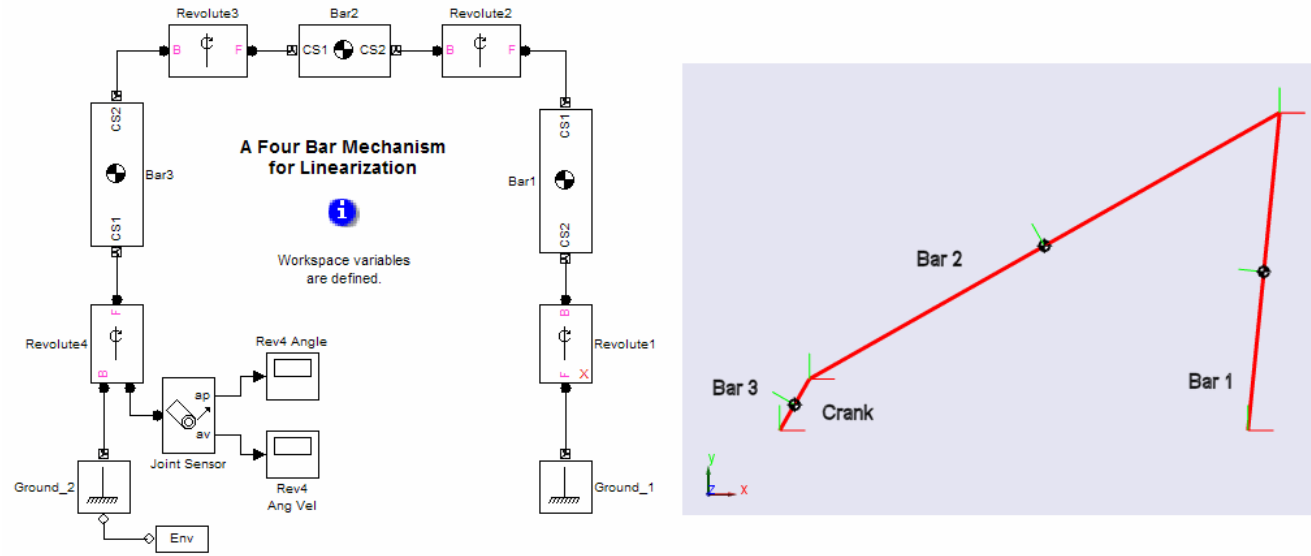
5 degrees

Closed-Loop Linearization: Four Bar Machine

Control System Toolbox™ Function This section uses the Control System Toolbox function `minreal`. Refer to the Control System Toolbox user's guide for more about this function and state-space analysis.

Linearizing a closed-loop machine is more complex than open-topology analysis. Each closed loop in the machine imposes implicit constraints that render some of the degrees of freedom (DoFs) dependent. Linearization of such a system must recognize that not all the DoFs are independent. A straightforward implementation of the `linmod` command results in redundant system states. You can eliminate these with the `minreal` function, which finds the minimal state space needed to represent your linearized model. To ensure that `minreal` produces a nonnull state space, you must linearize a closed-loop machine with at least one input *u* and one output *y*.

`mech_four_bar_lin` illustrates this reduction of independent DoFs: of the four revolute joints, only one is an independent DoF, which can be taken as any one of the revolutes. This model defines workspace variables in order to configure the initial geometry of lengths and angles (expressed in the model in meters and radians, respectively). Run the model in Forward Dynamics mode.

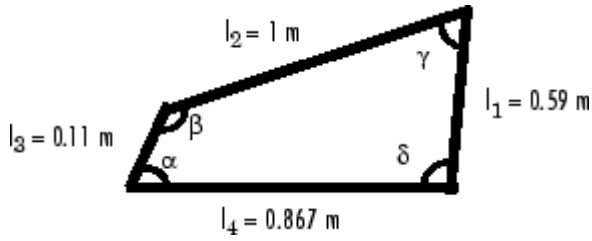


Consider a strategy to linearize the model about the four bar's (stable) natural equilibrium. You first find the natural equilibrium configuration, which is best accomplished by analyzing the loop constraints, making a guess, and then using the `trim` command to determine the equilibrium exactly. After choosing a system input and output, you then linearize the system.

"Modeling, Simulating, and Visualizing Simple Machines" presents this system in detail, in the section "Creating a Closed-Loop Mechanical Model". The preceding sections of this chapter, "Inverse Dynamics Mode with a Double Pendulum" on page 3-8 and "Constrained Trimming of a Four Bar Machine" on page 3-26, discuss the inverse dynamics and trimming of the four bar system.

Analyzing the Four Bar Geometry and Closed-Loop Constraint

You can determine the constraints and independent DoFs of the four bar with geometric and trigonometric identities applied to its quadrilateral shape. The lengths of the bars are l_1, l_2, and l_3, with the fixed base having length l_4.



The four joint angles satisfy $\alpha + \beta + \gamma + \delta = 2\pi$. Imagine cutting the quadrilateral diagonally from the α to the γ vertices, then from the β to the δ vertices. The law of cosines applied to these diagonals and the triangles so formed results in two constraints:

$$l_1^2 + l_2^2 - 2l_1l_2\cos\gamma = l_3^2 + l_4^2 - 2l_3l_4\cos\alpha$$

$$l_2^2 + l_3^2 - 2l_2l_3\cos\beta = l_1^2 + l_4^2 - 2l_1l_4\cos\delta$$

The four angles are thus subject to three constraints. Choose α (the crank angle) as the independent DoF. You can determine β, γ, and δ from α by inverting the constraints.

Making an Equilibrium Guess

First guess the natural equilibrium. An obvious guess for the natural equilibrium is for the crank (Bar 3) to point straight down, $\alpha = -90^o$.

1. Use the quadrilateral constraints to find

   $\beta = 313.2^o$, $\gamma = 60.3^o$, and $\delta = 76.5^o$

2. Redefine the workspace angles to these values (converted to radians).

   ```
   alpha = -90*pi/180; beta = 313.2*pi/180; gamma = 60.3*pi/180;
   delta = 76.5*pi/180;
   beta2 = pi - gamma - delta; delta2 = pi - delta;
   ```

3. Update the diagram and run the model again. This configuration is not the natural equilibrium, but it is close.

Determining the Natural Equilibrium with `trim`

Now find the natural equilibrium exactly by trimming the four bar in a manner similar to "Constrained Trimming of a Four Bar Machine" on page 3-26, but without external torque actuation. Revolute1 is already manually configured to be the cut joint in the closed loop, ensuring the DoF represented by Revolute4 is not eliminated from state space when the loop is cut.

1. Set the analysis mode to `Trimming`. Trimming mode inserts a subsystem and an output block that outputs a four-component signal representing the mechanical constraints resulting from the closed loop.

2. Use `mech_stateVectorMgr` to obtain the system's state vector:

```
StateManager = ...
   mech_stateVectorMgr('mech_four_bar_lin/Ground_2');
StateManager.StateNames
ans =
    'mech_four_bar_lin/Revolute2:R1:Position'
    'mech_four_bar_lin/Revolute3:R1:Position'
    'mech_four_bar_lin/Revolute4:R1:Position'
    'mech_four_bar_lin/Revolute2:R1:Velocity'
    'mech_four_bar_lin/Revolute3:R1:Velocity'
    'mech_four_bar_lin/Revolute4:R1:Velocity'
```

   Revolute1 is the cut joint and is missing from the list. States 1, 2, and 3 are the revolute 2, 3, and 4 angles, respectively; while states 4, 5, and 6 are the revolute 2, 3, and 4 angular velocities, respectively.

3. Set up the necessary trimming vectors:

```
x0 = [0;0;0;0;0;0]; ix = [];
u0 = []; iu = [];

y0 = [0;0;0;0];
iy = [1;2;3;4];
dx0 = [0;0;0;0;0;0];
idx = [3;6];
```

   The `x0` vector tells the `trim` command to start its search for the equilibrium with the four bar in its initial configuration (the equilibrium guess you entered into the workspace previously) and with zero angular velocities.

The index vector `ix` sets the states that, in the actual equilibrium, should keep the values specified in `x0`. Here there are none.

The `u0` and `iu` vectors specify system inputs, but there are none.

The `y0` vector sets the initial values of the constraint outputs to zero. The index vector `iy` requires that the constraint outputs at equilibrium be equal to their initial values (0). This ensures that the solution satisfies the mechanical constraints.

The `dx0` vector specifies the initial state derivatives. The initial derivatives of the angles (i.e., the angular velocities) and of the angular velocities (i.e., the angular accelerations) are set to zero. The index vector `idx` specifies that the velocity and acceleration of Revolute4 in the natural equilibrium must vanish. It is not necessary to constrain the derivatives of the other states because the system has only one independent DoF.

4 Now trim the system:

```
[x,u,y,dx] = ...
   trim('mech_four_bar_lin',x0,u0,y0,ix,iu,iy,dx0,idx);
```

The `u` vector is empty. The components of `y` and `dx` vanish, within tolerances, indicating that in equilibrium, respectively, the mechanical constraints are satisfied and the state derivatives vanish. The last three components of `x` vanish, indicating zero angular velocities at equilibrium. The first three components of `x` represent the natural equilibrium angles (in radians), measured as deviations from the initial configuration. The Revolute4 angle is -0.2395 rad = -13.7° from the starting point.

From `x`, you can calculate all the angle values. The natural equilibrium is α_{eq} = -90° - 13.7° = -103.7°, β_{eq} = 310.1° + 13.0° = 323.1°, γ_{eq} = 60.3° + 2.5° = 62.8°, and δ_{eq} = 360° - α_{eq} - β_{eq} - γ_{eq} = 74.7°.

Linearizing the Model at the Natural Equilibrium

You can now linearize the system at this trim point.

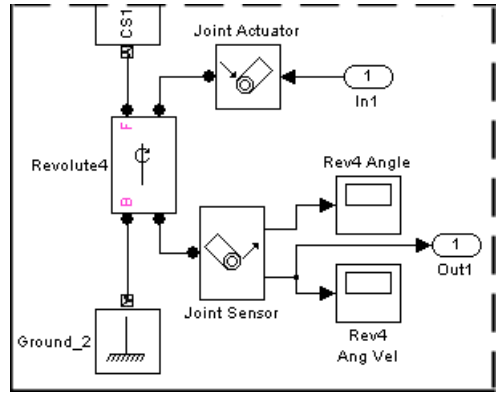
1 Reset the angles in your workspace to the natural equilibrium point:

```
alpha = alpha + x(3); beta = beta + x(2); gamma = gamma + x(1);
delta = 2*pi - alpha - beta - gamma; beta2 = pi - gamma - delta;
```

```
delta2 = pi - delta;
```

2 Change the analysis mode back to `Forward Dynamics` and update the diagram. Run the model to check that the mechanism indeed does not move.

3 To obtain a nontrivial linearized model, you need at least one input and one output. Connect a Joint Actuator to Revolute4 to actuate it with a torque. Then insert Simulink Inport and Outport blocks to input the torque and measure the angular velocity.



4 Set the input torque to zero and the initial state to the model's initial configuration, the natural equilibrium:

```
u = 0; x = [0;0;0;0;0;0];
```

5 Linearize the model and use `minreal` to eliminate the redundant states:

```
[A,B,C,D] = linmod('mech_four_bar_lin',x,u);
[a,b,c,d] = minreal(A,B,C,D);
```

leaving two states, α and dα/dt. The component $a(2,1) = -80.1 < 0$, indicating that this natural equilibrium is stable. The linearized motion is governed by $d^2\alpha/dt^2 = a(2,1)*\alpha$.

For More Information About State Space and Linearization

See "Open-Topology Linearization: Double Pendulum" on page 3-34 for more about the linearized state space representation.

4

Motion, Control, and Real-Time Simulation

SimMechanics software and Simulink form a powerful basis for advanced controls applications: trimming and linearizing motion, analyzing and designing controllers, converting plant and controller models to code, and simulating controller and plant on dedicated hardware. This chapter is a connected set of case studies illustrating these methods. As its example system, the studies use the Stewart platform, a moderately complex, six degree-of-freedom positioning system.

Before attempting these intricate case studies, you should understand the simpler motion analysis concepts, methods, and results of Chapter 3, "Analyzing Motion".

"Translating a CAD Stewart Platform" presents a related example, converting a Stewart platform computer-aided design assembly into a SimMechanics model.

Guide to This Chapter

In this section...

About the Stewart Platform and How It Is Modeled

The chapter starts with a summary of the Stewart platform and the models.

About the Case Studies

The studies use Stewart platform models and a suite of products to help you carry out advanced mechanical design and simulation tasks. The tasks are grouped into the following case studies. In them, you make use of such techniques as M-file scripts, linked libraries, and configurable subsystems to simplify the task of defining a complex Simulink and SimMechanics simulation.

Structure and Dependencies

The studies begin with motion analysis and control design. You learn about the Stewart platform's motion, then use this understanding to implement

controllers for it. The studies end with code generation and hardware implementation. You learn how to convert controller and platform models to code, compile and run the code, and how to put that code on a hardware target.

The first study is important for a deeper understanding of trimming and might be useful before attempting the control design studies that follow. The last two studies are connected, and you should work through them in order.

Caution SimMechanics code generation is intended for rapid prototyping and hardware-in-the-loop applications. It is not intended for use as production code in embedded controller applications.

Case Study Files

Each study has an associated set of demo files and is based on an appropriate variant model of the Stewart platform.

Saving Intermediate Stages of Work

It is recommended that you complete each case study in one session. If you cannot, for lack of time, you should periodically save your intermediate results from your workspace to a MAT-file.

Products Needed for the Case Studies

The case studies of this chapter require MATLAB, Simulink, and the SimMechanics product throughout. You should have a good working knowledge of all three.

In addition, you use several specialized products for specific tasks in each study. You should have at least a beginner's level experience with each.

Product	Required for Case Study
Control System Toolbox	"Trimming and Linearizing Through Inverse Dynamics" on page 4-24 (one part) "Analyzing Controllers" on page 4-39 "Designing and Improving Controllers" on page 4-50

Product	Required for Case Study
Robust Control Toolbox™	"Designing and Improving Controllers" on page 4-50 (last part)
Simulink® Control Design™	"Designing and Improving Controllers" on page 4-50
Real-Time Workshop	"Generating and Simulating with Code" on page 4-71 and "Simulating with Hardware in the Loop" on page 4-81
xPC Target	"Simulating with Hardware in the Loop" on page 4-81

References

[1] Stewart, D., "A platform with six degrees of freedom," *Proc. Inst. Mech. Eng.,* Vol. 180, part I(15), 1965-1966, pp. 371-386.

[2] Wilkie, J., M. Johnson, and R. Katebi. *Control Engineering: An Introductory Course.* Hampshire, United Kingdom: Palgrave/St. Martin's Press, 2002.

[3] Maxwell, J. C., "On Governors," *Proc. R. Soc.* (London), Vol. 16(100) (March 1868), pp. 270-283.

[4] Smith, N., and J. Wendlandt, "Creating a Stewart Platform Model Using SimMechanics," *MATLAB Digest* 10(5) (September 2002), http://www.mathworks.com/company/newsletters/digest/sept02/stewart.html. This case study includes only the actual leg trajectory in the derivative term, not the reference trajectory. The derivative term acts in that case like damping, not as error correction.

[5] Parsons, L., and J. Glass, "Recommendations for Creating Accurate Linearized Models in Simulink," *MATLAB Digest* 12(4) (July 2004), http://www.mathworks.com/company/newsletters/digest/july04/linmodels.html.

[6] Glover, K., and D. C. McFarlane. "Robust stabilisation of normalised coprime factor plant descriptions with H-infinity bounded uncertainty." *IEEE Trans. on Automatic Control,* Vol. 34, 1989, pp. 821-830.

[7] Georgiou, T. T., and M. C. Smith. "Optimal robustness in the gap metric." *IEEE Trans. on Automatic Control,* Vol. 35(6), 1990, pp. 673-687.

[8] Janka, R. S., *Specification and Design Methodology for Real-Time Embedded Systems* (New York/Berlin: Springer-Verlag, 2002).

[9] Li, Q., and C. Yao, *Real-Time Concepts for Embedded Systems* (Gilroy, California: CMP Books, 2003).

[10] Ledin, J., M. Dickens, and J. Sharp, "AIAA 2003: Single Modeling Environment for Constructing High-Fidelity Plant and Controller Models," American Institute of Aeronautics and Astronautics, 2003, http://www.mathworks.com/products/xpctarget/technicalliterature.html.

About the Stewart Platform

In this section...

Origin and Uses of the Stewart Platform

The Stewart platform is a classic design for position and motion control, originally proposed in 1965 as a flight simulator, and still commonly used for that purpose [1]. Since then, a wide range of applications have benefited from the Stewart platform. A few of the industries using this design include aerospace, automotive, nautical, and machine tool technology. The platform has been used to simulate flight, model a lunar rover, build bridges, aid in vehicle maintenance, design crane hoist mechanisms, and position satellite communication dishes and telescopes, among other tasks.

Characteristics of the Stewart Platform

The Stewart platform has an exceptional range of motion and can be accurately and easily positioned and oriented. The platform provides a large amount of rigidity, or stiffness, for a given structural mass, and thus provides significant positional certainty. The platform model is moderately complex, with a large number of mechanical constraints that require a robust simulation.

Most Stewart platform variants have six linearly actuated legs with varying combinations of leg-platform connections. The full assembly is a parallel mechanism consisting of a rigid body top or mobile plate connected to an immobile base plate and defined by at least three stationary points on the grounded base connected to the legs.

The Stewart platform used here is connected to the base plate at six points by universal joints. Each leg has two parts, an upper and a lower, connected by a cylindrical joint. Each upper leg is connected to the top plate by another universal joint. Thus the platform has 6*2 + 1 = 13 mobile parts and 6*3 = 18 joints connecting the parts.

Counting Degrees of Freedom in the Stewart Platform

The standard Stewart platform design has six independent degrees of freedom (DoFs). The mobile plate, if disconnected from the legs and thus unconstrained, also has six DoFs. The Stewart platform therefore exactly reproduces the possible motion of a free plate, but with the added benefit of stable and precise positioning control.

Here are two ways to count the Stewart platform DoFs.

- "Counting Degrees of Freedom on Bodies in Space" on page 4-8 starts with the disassembled platform parts as physical bodies in space.
- "Counting Degrees of Freedom as Joint Primitives" on page 4-9 starts with the platform represented as connected Body and Joint blocks.

Counting Degrees of Freedom on Bodies in Space

Start with the disassembled Stewart platform parts as unconstrained moving bodies. As you assemble the platform, you constrain the bodies as you connect them with joints. The base plate is immobile.

This approach is *not* the way that a SimMechanics simulation counts DoFs. See "Counting Degrees of Freedom as Joint Primitives" on page 4-9.

Bodies with DoFs. Each free body in space has six DoFs. Only after you attach them to one another with joints are they no longer able to move freely.

Joints as Constraints. Connecting bodies with joints constrains the two bodies so they can no longer move freely relative to one another.

For example, a universal joint connection allows two rotational DoFs, but imposes four constraints, three translational (positional) and one rotational.

Assembling the Stewart Platform Parts. Start assembling the Stewart platform. Each joint attachment simultaneously connects and constrains the bodies. In all, each leg imposes 12 constraints on itself and the top plate.

- The universals connecting the lower legs to the base plate impose four constraints:
  - Three positional, requiring two points to be collocated

  - One rotational, preventing the lower leg from rotating about its long axis (with respect to the immobile base)
- The cylindricals connecting the upper to the lower legs impose four constraints:
  - Two positional, allowing the two legs to slide along the long axis but not translate in the other two directions
  - Two rotational, allowing the upper leg to rotate about the long axis (with respect to the lower leg) but not rotate about the other two directions
- The universals connecting the top plate to the upper legs impose four constraints:
  - Three positional, requiring two points to be collocated
  - One rotational, preventing the upper leg from rotating about its long axis (with respect to the top plate — *not* with respect to the lower leg)

Obtaining the Independent DoFs. The Stewart platform has 13 moving bodies. With no constraints, the disassembled Stewart platform has 13*6 = 78 DoFs.

Assembling the parts imposes 12*6 = 72 constraints. Therefore, the Stewart platform has 13*6 - 12*6 = 6 independent DoFs.

Counting Degrees of Freedom as Joint Primitives

Start with the Stewart platform as an assembled SimMechanics model.

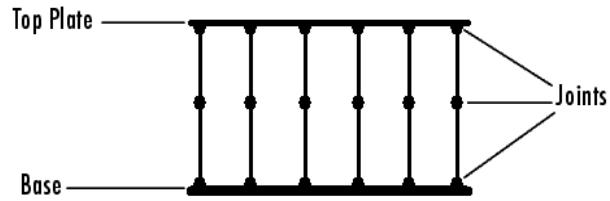
Bodies Without DoFs. A SimMechanics Body carries no DoFs. Instead, pairs of Bodies are connected by Joints, which express the motions of one Body relative to another.

Six Grounds represent the base plate. Thirteen Bodies represent the moving parts.

Joints Primitives as DoFs. Each Joint contains primitives. Translational and rotational primitives each express one DoF. (These are the only primitive types used here.) The Stewart platform model contains 18 Joints containing 6*6 = 36 primitives, of which 30 are rotational and 6 are translational.

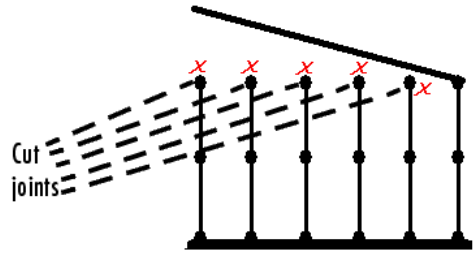
- Six Universal joints connecting the lower legs to the base. Each contains two rotational primitives.
- Six Cylindrical joints connecting the lower to the upper legs. Each contains a rotational and a translational primitive.
- Six Universal joints connecting the upper legs to the top plate. Each contains two rotational primitives.

Counting Loops. The Stewart platform legs form six loops, but only five are independent. You can obtain a topologically equivalent platform by flattening the top plate and base into lines and counting five loops that have the six legs as sides:



Cutting the Stewart Platform Joints and Deriving the Tree. To simulate a machine with closed loops (like the Stewart platform), a SimMechanics simulation replaces it internally with an equivalent machine (the *spanning tree*) obtained by cutting all the independent loops once and replacing the cuts with (invisible) equivalent constraints.

Obtain the spanning tree by cutting five of the six upper Universals. This cutting is just enough to open all loops but not disconnect the machine into disjoint parts. The tree contains 13 (uncut) Joints constituting 6*(2+2) + 2 = 26 DoFs.



Imposing the Cutting Constraints and Deriving the Independent DoFs. To complete the conversion of the closed-loop machine into an equivalent tree, impose constraints to replace the cut Joints. There are 20 such constraints. Each constraint is equivalent to reattaching a cut Joint and analyzes into five sets of

- Three positional constraints, requiring two points to be collocated
- One rotational constraint, preventing the upper leg from rotating about its long axis relative to the top plate

Thus reattaching the cut Joints to reassemble the platform leaves 26 - 5*4 = 6 independent DoFs.

Representing the Independent Degrees of Freedom

These six independent DoFs are usually taken to be the six leg lengths. Every other DoF identified here is now dependent on these six lengths. Each time you change a length, the universals connecting the legs to the base and top plate rotate, the top plate shifts and rotates, and the upper legs rotate about their long axes.

Alternatively and equivalently, you can take the six independent DoFs to be the six DoFs of the top, mobile plate. By connecting the top plate, you replace the six independent DoFs of an unconstrained plate with six DoFs under the precise and stable control of the six-leg positioning system.

The six DoFs of the connected top plate are not in addition to the leg-length DoFs. They are just an equivalent, replacement description of the same six independent DoFs. The whole platform system, once fully connected, always has exactly six independent degrees of freedom.

For More About Bodies, Joints, Degrees of Freedom, and Topology

Chapter 1, “Modeling Mechanical Systems” shows how SimMechanics Bodies and Joints represent bodies and DoFs. See especially these sections:

- “Modeling Degrees of Freedom” on page 1-19
- “Validating Mechanical Models” on page 1-85

Chapter 2, “Running Mechanical Models” explains the steps executed to analyze and simulate a machine. See especially these sections:

- “How SimMechanics Software Works” on page 2-24
- “Troubleshooting Simulation Errors” on page 2-26

Modeling the Stewart Platform

In this section...

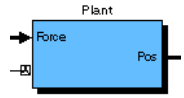
How the Stewart Platform Is Modeled

This section explains the essential details of modeling the Stewart platform in the SimMechanics environment. To understand the section better, use any top-level model from the case studies of this chapter, except the models of "Trimming and Linearizing Through Inverse Dynamics" on page 4-24. These are different because they lack a controller subsystem and consist of a plant model alone.

The control design model, `mech_stewart_control`, is this section's example.

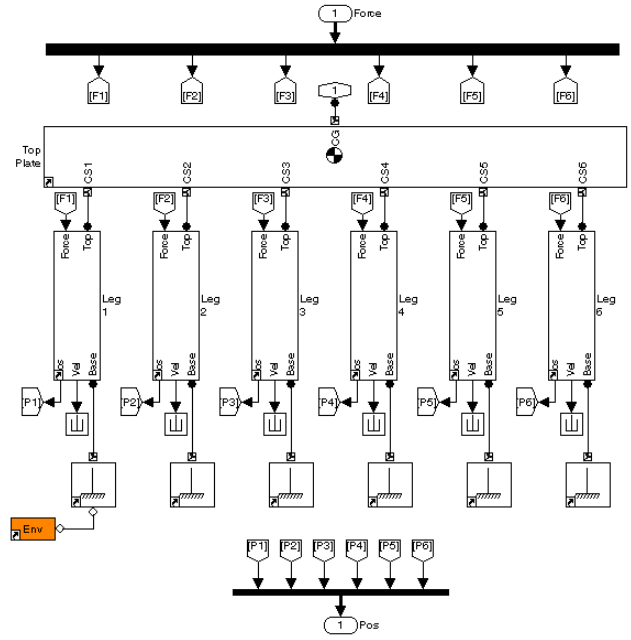
Modeling the Physical Plant

In three of the case studies, a larger control system model contains a Plant subsystem that incorporates the platform.



Viewing the Platform Model

The Plant subsystem models the Stewart platform's moving parts, the legs and top plate. Open this subsystem.



Stewart Platform Model (Control Design Version)

Each of the legs is an instance of a library block located in another library model, `mech_stewartplatform_leg` or `mech_stewart_control_equil_leg`.

1. Select one of the leg subsystems and right-click. Select **Link Options**, then **Go To Library Block**, to open this library.

2. Open the masked library block, Leg Subsystem, and the individual Body and Joint blocks that make up a whole leg.

3. Now close the blocks, subsystems, and linked libraries and return to the top-level model.

Modeling Controllers

Except in the "Trimming and Linearizing Through Inverse Dynamics" on page 4-24 study, the Stewart platform models contain controllers imposing actuating forces that guide the platform's motion to follow as closely as possible a *nominal* or *reference* trajectory. Implementing a controller requires computing the motion errors, the difference of the reference and actual motions of the platform. All the case study models use proportional-integral-derivative (PID) control.

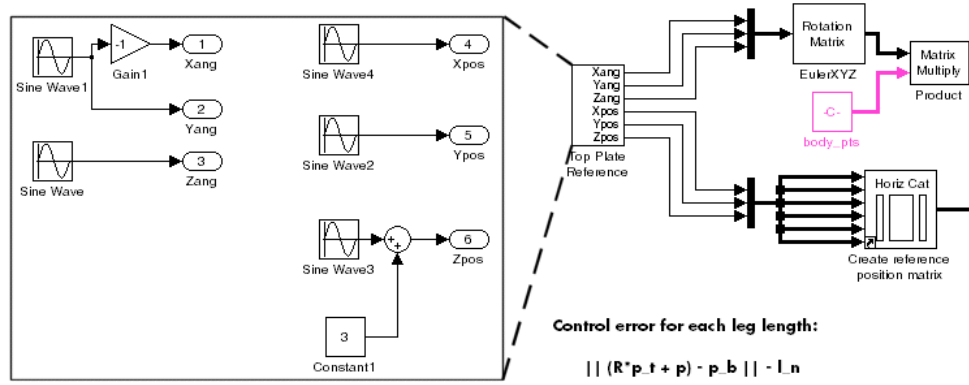
Generating the Reference Trajectory

Each model controller requires a reference trajectory.

1. Open the Leg Reference Trajectory subsystem.

   This set of blocks generates the set of six leg lengths, as functions of time, corresponding to a desired trajectory for the top plate.

2. Open the subsystem called Top Plate Reference. This set of blocks generates a reference trajectory in terms of linear position and three orientation angles, as a function of time. The workspace variable `freq` sets the frequency of the reference motion.



Stewart Platform Reference Trajectory Subsystem (Control Design Version)

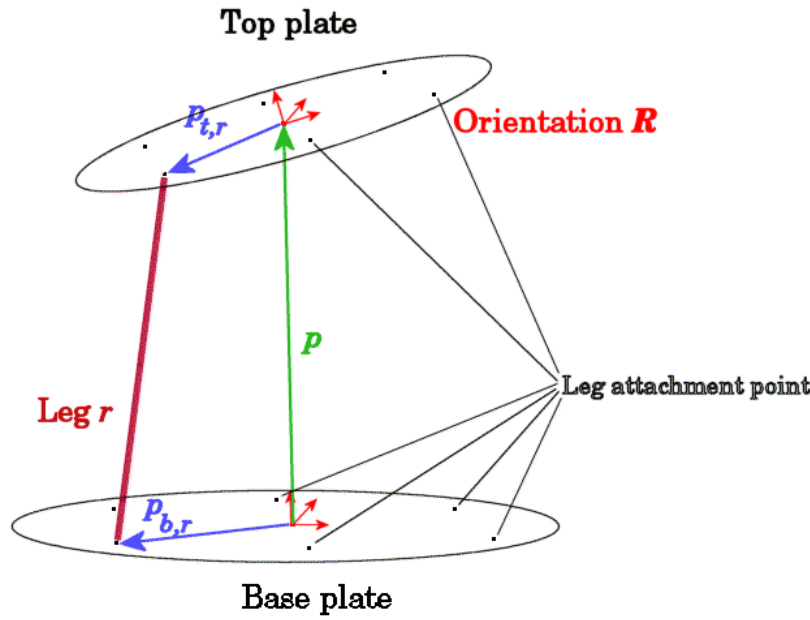
- The reference trajectory provided uses sinusoidal functions of time to define the rotational and translational degrees of freedom.

- If you want, you can design and implement another reference trajectory of your choosing and replace this sub-subsystem.

Whatever comes out of Top Plate Reference, the subsystem Leg Reference Trajectory assumes the translational position/three-angle form for the top plate. The rest of the Leg Reference Trajectory subsystem transforms these six degrees of freedom (DoFs) into the equivalent set of six DoFs expressed as the lengths of the six platform legs. The reference trajectory output of the subsystem is a six-vector of these leg lengths.

Finding the Motion Error

The actuating force on leg *r* is a function of the motion error. The error requires finding the instantaneous length of each leg from the positions of that leg's top and bottom connection points.



Defining the Length of a Stewart Platform Leg

The motion error is the difference of the desired or reference length of the leg and its instantaneous or actual length:

$$\begin{aligned}\text{Error} = E_r &= \text{reference length of leg} - \text{actual length of leg} \\ &= L_{\text{traj},r}(t) - |\,(R \cdot \boldsymbol{p}_{\text{t},r}) - \boldsymbol{p}_{\text{b},r}\,|\end{aligned}$$

The reference length $L_{\text{traj}}(t)$ is given as a function of time by the output of the Leg Reference Trajectory subsystem. The vectors $\boldsymbol{p}$, $\boldsymbol{p}_{\text{t,r}}$, and $\boldsymbol{p}_{\text{b,r}}$ are defined in the preceding figure. The orthogonal rotation matrix R specifies the orientation of the top plate with respect to the bottom.

The Standard PID Controller and Its Control Law

All the Stewart platform models use a simple PID controller and Joint Sensor blocks to measure motion. The simplest implementation of trajectory control is to apply forces to the plant proportional to the motion error. PID feedback is a common form of linear control.

A PID control law is a linear combination of a variable detected by a sensor, its time integral, and its first derivative. This Stewart platform's PID controller uses the leg position errors E_{r} and their integrals and velocities. The control law for each leg r has the form:

$$\text{F}_{\text{act},r} \;=\; K_p \text{E}_r \;+\; K_i \int_0^{\text{t}} \text{E}_r \; dt + \; K_d (d\text{E}_r / dt)$$

The controller applies the actuating force $F_{\text{act,r}}$ along the leg.

- If E_{r} is positive, the leg is too short, and $F_{\text{act,r}}$ is positive (expansive).
- If E_{r} is negative, the leg is too long, and $F_{\text{act,r}}$ is negative (compressive).
- If E_{r} is zero, the leg has exactly the desired length, and $F_{\text{act,r}}$ is zero.

The real, nonnegative K_{p}, K_{i}, and K_{d} are, respectively, the proportional, integral, and derivative gains that modulate the feedback sensor signals in the control law:

- The first term is proportional to the instantaneous leg position error, or deviation from reference.

- The second term is proportional to the integral of the leg position error.
- The third term is proportional to the derivative of the leg position error.

The result is $F_{act,r}$, the actuator force (input) applied by the controller to the legs. The proportional, integral, and derivative terms tend to make the legs' top attachment points $\boldsymbol{p}_{t,r}$ follow the reference trajectories by suppressing the motion error.

For More About Controllers

The case study, "About Controllers and Plants" on page 4-35, discusses controlling platform motion in greater detail. In that study, you also use an H-infinity controller, as well as use transfer functions to take motion derivatives.

In addition, consult "References" on page 4-5.

Initializing the Stewart Platform

When representing the physical components of the Stewart platform model with SimMechanics blocks and the control components with Simulink blocks, you must define the geometry of its initial state and the mass parameters of the Stewart platform bodies. Although each case study in this chapter uses a variant model, all initialize the platform and controller configuration in a common way.

Geometric, mass, dynamical, and controller information is specified in the block dialogs by referencing variables in your MATLAB workspace. An M-file script accompanies the Stewart platform models and sets these values.

Running this script configures the blocks in their starting geometric state, with the correct mass properties for the bodies. When you open it, each model uses the same initialization M-file as a pre-load function. To see this setting,

1. Go to the **File** menu and select **Model Properties**.
2. Then in the dialog, select the **Callbacks** tab and find the **Model pre-load function** field.

Stewart Platform Initialization M-File

File	Purpose
`mech_stewart_studies_setup`	M-file script to fill the workspace with geometric, dynamical, and controller data.
`inertiaCylinder`	M-file function called by `mech_stewart_studies_setup`. Computes the principal inertias of a solid cylinder.

Body and Joint Geometric Configuration

The script first defines basic angular unit conversions and axes. The World coordinate system (CS) is located at the center of the immobile base plate. The connection points on the base and top plate are defined with respect to World. These definitions include the offset angle of 60 degrees between the base and top plates, the radii of both the base and top plates, the initial position height of the top plate, and the vectors pointing along the legs. The array of top points is permuted so that the same index represents the top and bottom connection points for the same leg.

The script calculates the revolute and cylindrical axes used in the joint blocks of the leg subsystems. There are two revolute axes for each Universal joint that connects an upper leg to the top plate, one cylindrical and one revolute axis for the linear motion of the Cylindrical joint connecting upper and lower legs, and two revolute axes for each Universal block that connects a lower leg to the base plate. The script then configures all 13 moving bodies by defining coordinate systems at the center of gravity (CG) of each.

The top plate's home configuration is symmetric equilibrium: flat, with equal leg lengths specified by the workspace vector `leg_length`.

Body Mass Properties

The script defines the mass properties of all bodies. These comprise the inertia tensors and masses for the top plate, the bottom plate, and the legs. The mass properties calculation assumes that the platform is made with steel. The script calls the function `inertiaCylinder` to calculate the inertia tensors and masses of the legs and the top and base plates, given the material density, the length and inner and outer radii of the leg cylinders, and the thicknesses and radii of the top and base plates.

Motion Constants, Controller Parameters, and Initial Condition

In its final steps, the script defines motion and control constants as workspace variables: motion frequency, derivative filtering cutoff, leg actuator force saturation, and controller gains. Each case study model uses some or all of these constants, which you can change as desired.

Real force actuators are saturate at a specific force level. The Force Saturation block limits the actuating force to the value of the workspace variable `force_act_max`.

The integral (I) part of the PID controller exhibits an extended response time whose overall effect is controlled by the ratio of K_i to K_p. The Integrator for the I part has a nonzero **Initial condition** field, specified by the workspace variable `initCondI`, adjustable to compensate for initial transient behavior. The script initializes its value to

```
(upper_leg_mass+lower_leg_mass+(top_mass*1.3/6))*9.81/Ki
```

corresponding to the leg forces in symmetric equilibrium.

Motion and Filtering Constants

Dynamical Feature	Workspace Variable	Associated Natural Frequency	Associated Time Scale
Top plate motion	`freq` = π rad/s	`freq`/2π = 0.5 Hz	2π/`freq` = 2 s
Filtered derivative cutoff	`A` = 100*`freq` = 100π rad/s	`A`/2π = 50 Hz	2π/`A` = 0.02 s

PID Controller Constants

Dynamical Constant	Workspace Variable
Force saturation	`force_act_max` = 3e5 newtons (N)
Integral (I) gain	`Ki` = 1e4 (newtons/meter/second) (N/m-s)
Proportional (P) gain	`Kp` = 2e6 newtons/meter (N/m)
Derivative (D) gain	`Kd` = 4.5e4 newtons-seconds/meter (N-s/m)

Identifying the Simulink and Mechanical States of the Stewart Platform

For the purposes of SimMechanics motion analysis, you need to know the model's Simulink and mechanical states. These are distinct from the system's degrees of freedom (DoFs) and depend on the analysis mode you choose.

Pure Simulink States

If you use a controller or other subsystem made up of pure Simulink blocks with your Stewart platform, your model might contain Simulink states. For example, Integrator and Transfer Fcn blocks each have an associated state, and State-Space blocks can have many.

The default Stewart platform controller is a PID subsystem, which integrates six feedback signals and thus has six Simulink states. In the "Analyzing Controllers" on page 4-39 and "Designing and Improving Controllers" on page 4-50 studies, you can also choose to use the filtered derivative, which has 12 transfer functions and thus adds 12 Simulink states.

Mechanical States in Forward Dynamics Mode

A mechanical system modeled with Joint blocks contains mechanical states distinct from Simulink states that include both joint position and velocity. In Forward Dynamics mode, the Stewart platform contains 52 tree states, of which 12 are independent.

The joints and their related DoFs are discussed in "Counting Degrees of Freedom in the Stewart Platform" on page 4-8.

Joint Primitives and States. Each Joint consists of one or more primitives. The position and velocity of a joint primitive each have a state. The Stewart platform has 36 joint primitives and thus potentially 72 states.

Cutting Joints and Obtaining the Tree States. Because the Stewart platform has closed topology, SimMechanics model will cut five of the Joints to arrive at an equivalent open-topology or tree machine. These Joints are replaced internally by equivalent cutting constraints.

Five Universals and 5*2*2 = 20 joint primitives are eliminated this way. The equivalent open machine thus has 72 - 20 = 52 tree states.

Counting the Cutting Constraints. Not all these states are independent. There are 40 equivalent constraints that replace the cut Joints.

- Each cut Universal imposes one rotational and three position constraints.
- Each constraint also constrains the corresponding velocity.
- There are five cut Joints.

Thus there are 5*2*4 = 40 invisible constraints generated by the cutting.

Finding the Independent States. Thus the Stewart platform model has 52 - 40 = 12 independent mechanical states, corresponding to the six independent DoFs and their velocities.

Mechanical States in Trimming and Kinematics Modes

You can also analyze the Stewart platform's motion in inverse dynamics and locate steady-state operating points.

- Because the Stewart platform is a closed-loop machine, you must simulate its inverse dynamics in the Kinematics mode.
- You can find operating points in the Trimming mode with the Simulink `trim` command.

In both the inverse dynamics and trimming cases, the Simulink states associated with the SimMechanics joint primitives are *not* the DoFs, but the *(invisible) joint-cutting constraints* that reduce the tree states to independent states. The state values measure how well the constraints are satisfied. A zero value means a constraint is satisfied perfectly.

In the mechanical part of the Stewart platform model, there are 52 tree states and 12 independent states. Thus the SimMechanics model counts 52 - 12 = 40 cutting constraints. In the Kinematics and Trimming modes, these 40 constraints are the mechanical states.

Open Topology and Inverse Dynamics Mode. If the Stewart platform had an open topology, you would simulate its inverse dynamics in Inverse Dynamics mode instead. However, there would be no closed loops, and the simulation would not cut any Joints. With no cutting constraints, an open topology machine has no states in Inverse Dynamics or Trimming mode.

For More About Mechanical States, Cutting Loops, and Analysis Modes

Learn more about SimMechanics states and loops in Chapter 1, "Modeling Mechanical Systems":

- "Cutting Machine Diagram Loops" on page 1-46
- "Validating Mechanical Models" on page 1-85

Consult the `mech_stateVectorMgr` command reference as well.

For more about analysis modes, see

- "Simulating and Analyzing Mechanical Motion".
- Chapter 3, "Analyzing Motion".

Visualizing the Stewart Platform Motion

To view mechanical animation, consult the *SimMechanics Visualization and Import Guide*.

With the SimMechanics visualization window open, you can view the platform motion from different perspectives. View the platform in the *xy*-plane, from above. Then switch the view to the *xz*- or *yz*-plane.

The initial state of motion specified by the reference trajectory is slightly different from the home configuration and generates an initial transient.

Trimming and Linearizing Through Inverse Dynamics

In this section...

About Trimming and Inverse Dynamics

Note This study requires Control System Toolbox at an optional step, "Finding the Minimal Realization of the Linearized Model" on page 4-32.

This case study finds a Stewart platform steady state with the SimMechanics Kinematics mode . You specify motions and determine the forces and torques to produce those motions (the *inverse dynamics* problem). If you are not familiar with implementing inverse dynamics in the SimMechanics environment, work through the "Finding Forces from Motions" on page 3-7 before attempting this case study.

Use the Inverse Dynamics and Kinematics modes for inverse-dynamic analysis of open- and closed-topology systems, respectively. The Stewart platform has a closed topology and thus requires the Kinematics mode. Once you have an operating point, you can linearize the motion.

What Is Trimming?

Trimming a system means locating a configuration of its states with certain prior conditions imposed on the states and possibly their derivatives. In a mechanical context, it means imposing conditions on certain positions and velocities, then determining the remaining positions and velocities such that the entire state of the machine is consistent. A by-product of mechanical

trimming is determination of the forces/torques necessary to produce the specified motion. These motion states constitute a *trim* or *operating point*. Trimming problems can have one solution, more than one, or none.

Pure inverse dynamics imposes prior motions on all degrees of freedom. Then all the states are determined. (The consistency of the motions is not guaranteed, but must be checked.) Only the forces/torques remain to be found.

Ways to Find an Operating Point

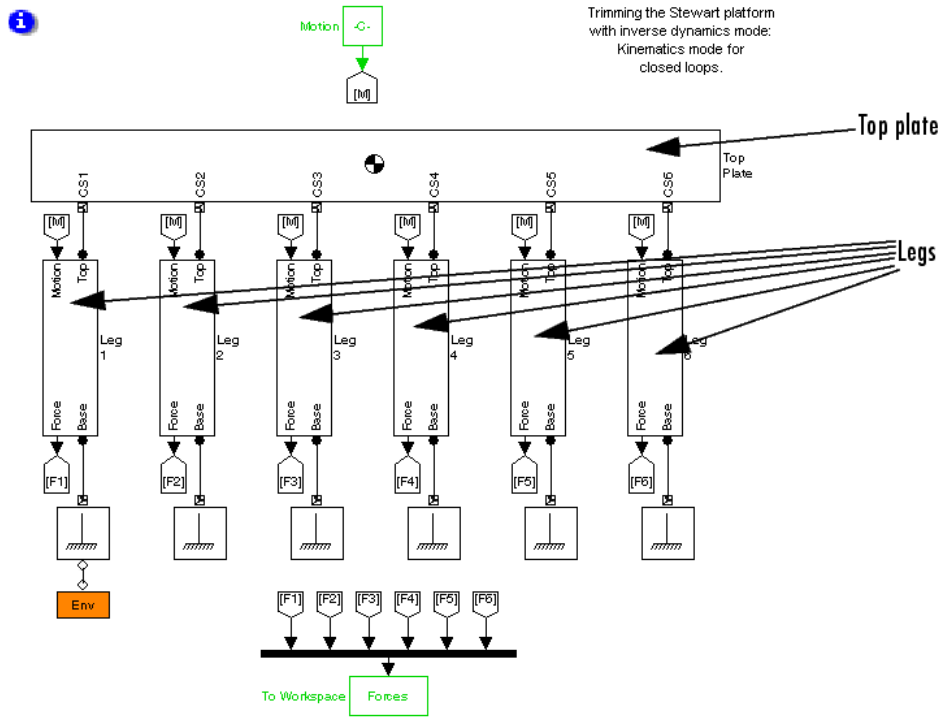
To find an operating point or steady state for a SimMechanics model,

- Use the `trim` command in Simulink. See "Trimming Mechanical Models" on page 3-18.
- Use the more powerful techniques provided by Control System Toolbox and Simulink Control Design. See "About Controllers and Plants" on page 4-35.
- Use the SimMechanics inverse dynamics modes. You can manipulate the mechanical states of your model directly with motion actuation rather than manipulate them through Simulink.

Trimming in the Kinematics Mode

Here are the files needed for this case study. The models also call the initialization M-files. Open the first model.

File	Purpose
`mech_stewart_control_equil`	Kinematics model for determining Stewart platform force equilibrium
`mech_stewart_control_equil_leg`	Library model of Stewart platform leg for kinematic analysis
`mech_stewart_control_plant`	Forward dynamics model for linearizing the Stewart platform
`mech_stewartplatform_leg`	Library model of Stewart platform leg for forward dynamic analysis



Simulation Settings for Inverse Dynamics

The mech_stewart_control_equil model has some preset nondefault settings.

Configuration Parameters

Setting	Value
Solver > Simulation time > Stop time	`0.005` seconds
Data Import/Export > Save to workspace	**Time** and **States** selected > `tout` and `xout`

Configuration Parameters (Continued)

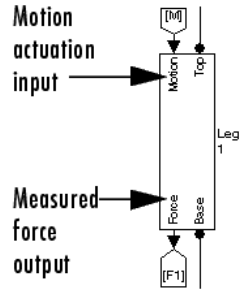
Setting	Value
SimMechanics > Diagnostics	**Mark automatically cut joints** selected
SimMechanics > Visualization	**Display machines after updating diagram** and **Show animation during simulation** selected

Machine Environment

Setting	Value
Parameters > Analysis mode	`Kinematics`
Parameters > Machine Dimensionality	`3D Only`
Constraints > Constraint solver type	`Machine Precision`
Constraints > Use robust singularity handling	Selected

Specifying the Motion

The six Stewart platform legs are instances of a basic leg saved in the `mech_stewart_control_equil_leg` library. It takes as inputs the motion actuation signals that specify position and velocity as a function of time. The position signals specify the platform's motion relative to the initial geometric configuration.



In mech_stewart_control_equil, the Motion subsystem specifies motion as trivial: zeroes for all six leg positions and velocities. That is, the model holds the platform still in its initial state.

Measuring the Steady-State Forces

Each Stewart platform leg outputs the computed leg force needed to maintain the motion specified by the motion actuation. These six measured forces are directed to your MATLAB workspace by the To Workspace block.

1. Open the To Workspace dialog.

   The output forces are stored in the vector variable `Forces`. The block retains the force vector only from the last time step.

2. Close the To Workspace dialog.

Running the Model and Obtaining the Outputs

Now run mech_stewart_control_equil.

1. Click **Start** and wait for the simulation to finish.

2. In your workspace, locate `tout` and `xout`. These are the time steps and the corresponding state values, respectively.

   In the Inverse Dynamics mode, there are 40 mechanical states counted by Simulink, associated with the mechanical constraints. Consult “Identifying the Simulink and Mechanical States of the Stewart Platform” on page 4-21.

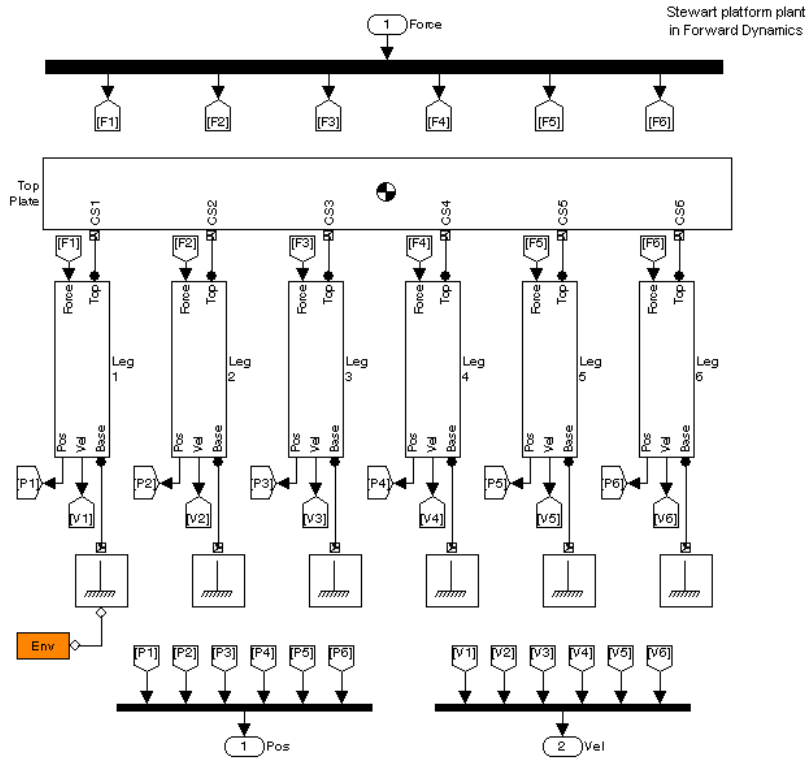
3 Locate `Forces` in the workspace. These are the six force values along each leg to hold the platform still against falling by gravity. The values are positive (expansive) along the legs.

Linearizing the Stewart Platform at an Operating Point

Knowing the steady-state forces needed to keep the platform still, you now linearize another version of the model, mech_stewart_control_plant. It has settings similar to mech_stewart_control_equil, except that:

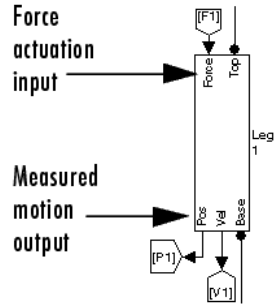
- The **Analysis mode** is set to `Forward Dynamics`.
- The simulation time is 10 seconds.
- **Time** and **Output**, `tout` and `yout`, respectively, are saved to workspace.

Open the `mech_stewart_control_plant` model.



- The six legs are instances of the mech_stewartplatform_leg library. This leg takes force as an input and outputs position and velocity, as appropriate for forward dynamics.
- The standard model input variable is `u`. The force vector signal is a model input.
- The position and velocity vector signals are model outputs. The **Data Import/Export** output variable is `yout` and will appear in your workspace assigned with data after you simulate.

Close the model.



Linearizing the Forward Dynamics Model

You can simulate the mech_stewart_control_plant model without opening it.

1 At the command line, enter

```
nomForces = Forces'; % Transpose the force vector
```

2 Linearize the model by entering

```
[A,B,C,D] = ...
   linmod('mech_stewart_control_plant',[],nomForces);
```

The arguments are, in order,

- Model name
- Model state vector (not used)
- Model input vector `u = nomForces`

These (unreduced) output matrices are the standard state-space representation of a linearized model. The space is defined by x, u, and y, the state, input, and output vectors, respectively.

$$\mathrm{d}\boldsymbol{x}/\mathrm{dt} = A \cdot \boldsymbol{x} + B \cdot \boldsymbol{u}$$
$$\boldsymbol{y} = C \cdot \boldsymbol{x} + D \cdot \boldsymbol{u}$$

There are 52 states, 6 inputs, and 12 outputs. Thus A, B, C, D have dimensions 52-by-52, 52-by-6, 12-by-52, and 12-by-6, respectively. Not all these matrix entries are independent.

Finding the Minimal Realization of the Linearized Model

Note This step requires Control System Toolbox.

Of the 52 mechanical states, the Stewart platform has only 12 independent states, corresponding to six degrees of freedom (DoFs). Each DoF corresponds to one position and one velocity.

To eliminate the redundant states, enter

```
[a,b,c,d] = ...
   minreal(A,B,C,D);
40 states removed.
```

at the command line. The a, b, c, d matrices are reduced in size to 12-by-12, 12-by-6, 12-by-12, 12-by-6, respectively.

For More About Linearization and State Space

See “Open-Topology Linearization: Double Pendulum” on page 3-34 and the Simulink documentation.

Further Suggestions for Inverse Dynamics Trimming

“Trimming in the Kinematics Mode” on page 4-25 and “Linearizing the Stewart Platform at an Operating Point” on page 4-29 present the simplest possible trimming scenario:

- All six degrees of freedom (DoFs) are determined by prior specification of positions and velocities. These are the inputs to the problem. The outputs are the forces necessary to maintain the specified motion. The simulation solves a pure inverse dynamics problem.
- The actual motion actuation signals require the platform to hold still relative to its initial geometric configuration.

General Trimming Conditions: Mixed Dynamics

In a more typical trimming problem, you specify some of the DoFs by motion actuation and leave the others free to respond to forces/torques. Such a scenario is a *mixed dynamics* problem. In the SimMechanics environment, you can solve such problems in

- Forward Dynamics mode, where the tree states (DoFs corresponding to uncut Joints) are the mechanical states
- Kinematics mode (closed topology), where the cutting constraints that replace the cut Joints constitute the mechanical states
- Inverse Dynamics (open topology), where there are no mechanical states

Complementarity of Inverse and Forward Dynamics

Actuate DoF with...	Sense on DoF...
Forces/torques	Motions
Motions	Forces/torques

If you want to solve such a problem for the Stewart platform, you need to

- Use a library leg with
  - Force input
  - Motion output

  for each leg simulated in forward dynamics. You actuate it with a force and measure its motion. Use the mech_stewartplatform_leg block library.
- Use a library leg with
  - Motion input
  - Force output

  for each leg simulated in inverse dynamics. You actuate it with a motion and measure the corresponding force. Use the mech_stewart_control_equil_leg block library.

Using the Operating Point to Linearize a Model

The steady-state outputs are in turn the inputs for linearization.

Complementarity of Trimming and Linearization

Trimming Output Becomes...	...Linearization Input
Measured motions become...	...Motion actuation signals
Measured forces/torques become...	...Force/torque actuation signals

To carry out a linearization of your system,

1. Create a variant model in Forward Dynamics mode that takes
   - The steady-state forces as linearization input force actuation
   - The steady-state motions as linearization input motion actuation
2. Linearize with `linmod`.

   ```
   linmod('forward_dynamics_model_to_linearize', state, input)
   ```

   This command can feed model inputs into the linearized simulation as a command argument. See the command reference for more details.

About Controllers and Plants

In this section...

Modeling Controllers in Simulink and Plants in SimMechanics Software

Note The next two studies assume some knowledge of control systems. In addition to Simulink and the SimMechanics product, the studies use these products:

- Control System Toolbox
- Simulink Control Design
- Robust Control Toolbox

You should have some experience with these tools before proceeding.

To understand trimming better, work through "Trimming and Linearizing Through Inverse Dynamics" on page 4-24.

A classic engineering problem is the design of *controllers* for a physical system, the *plant* [2]. A SimMechanics model can represent a complex mechanical system and helps you design and implement a control system for the plant, in conjunction with Simulink and related control design products.

In the next two case studies, you use SimMechanics software to model the plant and Simulink to analyze and synthesize controllers. You explore a basic

challenge of control design, the tradeoff between responsiveness and stability, by implementing first a simple controller, then a more complex and robust one [4]. This section is preliminary to those studies.

Nature of the Control Problem

The motion of an uncontrolled physical system is represented by its position and velocity variables arranged into a *state vector* $\boldsymbol{X}$. The dynamics of the system is described by a force law:

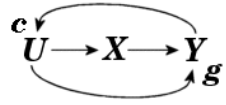
$$d\boldsymbol{X}/dt = \boldsymbol{f}(\boldsymbol{X})$$

Introducing control means introducing sensors and actuators that modify the system's otherwise natural motion. The actuators impose artificial forces — collectively, the *inputs* $\boldsymbol{U}$ — on the system, while the sensors detect motions and report *outputs* $\boldsymbol{Y}$. The dynamics of the controlled system are modified:

$$d\boldsymbol{X}/dt = \boldsymbol{f}(\boldsymbol{X},\boldsymbol{U})$$
$$\boldsymbol{Y} = g(\boldsymbol{X},\boldsymbol{U})$$

The $\boldsymbol{U}$ and $\boldsymbol{Y}$ are the *control variables* of the system.

By selecting the proper set of $\boldsymbol{U}$ and $\boldsymbol{Y}$ and a *feedback control* or *compensator law* $\boldsymbol{U} = \boldsymbol{c(Y)}$ that modifies the system's motion $\boldsymbol{X}$ in a desired way, you impose control actuator forces for the relevant range of $\boldsymbol{X}$, $\boldsymbol{U}$, and $\boldsymbol{Y}$.



Selecting $\boldsymbol{c}$ is the fundamental problem of control design. The desired trajectory of $\boldsymbol{X}$ is the reference or nominal trajectory. The difference of the actual and reference trajectories is the motion error. Finding the actuator forces needed to produce a desired motion is closely related to the problem of inverse dynamics. See the case study, "Trimming and Linearizing Through Inverse Dynamics" on page 4-24.

Control Transfer Function Forms and Units

The controller and plant transfer functions are often called *C* and *G*, respectively. The combined controller-plant transfer function forms are the *open-loop CG* and the *closed-loop CG/(1+CG)*.

Controller and plant response magnitudes are measured in decibels (dB).

Controller-Plant Case Study Files

The next two case studies use these files, in addition to the initialization M-files.

File	Purpose
`mech_stewart_control`	Main model
`mech_stewart_control_deriv`	Configurable subsystem: Derivative block or transfer function (filtered)
`mech_stewart_controllers`	Configurable subsystem: Null, PID, or H-infinity controller
`mech_stewartplatform_leg`	Library model of Stewart platform leg; used six times in the Plant subsystem of the main model

For More About Designing Controllers

The problems and techniques of the next two case studies only touch the basics of control design. In practice, you need to consider additional issues and goals. Also consult "References" on page 4-5.

Finding Other Operating Points

To fully understand the plant, you need to find plant operating points other than the simple ones used here and optimize the controller in other representative states.

See the preceding case study, "Trimming and Linearizing Through Inverse Dynamics" on page 4-24.

Compensating for Noise and Uncertainty

To design more robust controllers, you should consider the effect of parameter uncertainty and signal noise. This step involves comparing typical plant motion frequencies, noise frequencies, and filtered derivative cutoffs.

The following toolboxes can help with such tasks:

- Robust Control Toolbox
- Simulink® Design Optimization™

Designing for Hardware Implementation

To move toward hardware implementation, you must consider discretizing the controller [8]. Among other requirements, this necessitates using a fixed-step solver, optimizing the solver step size and sample rate, and adjusting the filtered derivative cutoff.

See the final two case studies:

- "Generating and Simulating with Code" on page 4-71
- "Simulating with Hardware in the Loop" on page 4-81

Analyzing Controllers

In this section...

Implementing a Simple Controller for the Stewart Platform

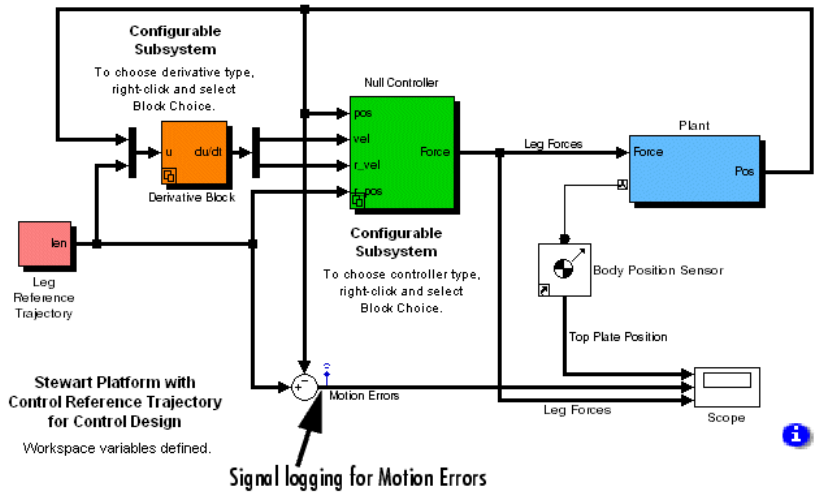
Note Before working through this study, consult the control design preliminary, "About Controllers and Plants" on page 4-35. The second control design study, "Designing and Improving Controllers" on page 4-50, builds on the results and concepts of this study.

In addition to Simulink and the SimMechanics product, this study uses the Control System Toolbox product.

This first control design case study implements the Stewart platform control system with the standard preoptimized proportional-integral-derivative (PID) controller. It introduces you to the overall model and the uncontrolled Stewart platform motion. It then shows how the PID controller works, how to make it more realistic with a filtered derivative, and how to exploit classical control techniques to analyze the PID response.

A First Look at the Stewart Platform Control Model

Open the mech_stewart_control model.



Stewart Platform Control Design Model

The green controller subsystem is linked to an enabled subsystem in a related library model, mech_stewart_controllers. The initial configuration is to the Null Controller, which imposes no forces at all on the platform, the blue subsystem labeled Plant. Open the Null Controller subsystem. This controller accepts trajectory information, but outputs zero for the imposed force.

Signal logging captures the motion errors. You use this feature later to analyze controller performance.

Viewing the Controller

To see the controller subsystem library:

1. Right-click the Null Controller block and select **Link Options**, then **Go To Library Block**. The mech_stewart_controllers library opens with the Template block highlighted.

   You can set this enabled subsystem in three ways, and you use all three in this case study: **Null Controller**, **PID Controller**, and **H_inf Controller**.

2. Open the controller subsystem in each setting to examine its block diagram.

3 Double-click the Template block to see the controller subsystem design. The three possible subsystem settings are listed in the Template dialog.

   Close the library model.

4 You select the subsystem configuration actually used for simulation back in the original model, mech_stewart_control.

   Right-click the Null Controller block and select **Block Choice**. The three possible subsystems from the mech_stewart_controllers library are listed, with **Null Controller** selected.

Configuring the Dynamics

To see the dynamical settings for the control design model:

1 Open the Plant subsystem and the orange Machine Environment block. In the block dialog, locate the **Parameters** tab. The gravity vector points in the negative *z* direction.

   Then locate the **Constraints** tab. The **Constraint solver type** is `Machine precision`, and the **Use robust singularity handling** check box is selected. For this model, such a combination is the most robust.

   Close the dialog and subsystem.

2 From the **Simulation** menu, open Configuration Parameters. Locate the **SimMechanics** node, **Diagnostics** area. In this simulation, automatically cut joints are marked.

   Because the Stewart platform is a closed-loop machine, the simulation cuts one joint in each closed loop formed by the two plates and a pair of legs during the simulation and marked with a red `X`. See “Counting Degrees of Freedom in the Stewart Platform” on page 4-8 and “Identifying the Simulink and Mechanical States of the Stewart Platform” on page 4-21.

   Close the dialog.

Simulating the Stewart Platform Without Controls

First simulate the Stewart platform without any control forces. The platform moves under the influence of gravity and initial conditions only. The reference trajectory is irrelevant because it is not used to generate any control forces.

To watch the natural or uncontrolled motion of the Stewart platform:

1. Open the Scope block. The Scope window displays three measurements:
   - Position of the top plate CG
   - Control errors
   - Control forces applied to move the legs
2. Start the model. Track the falling platform by watching the Top Plate Position graph in the Scope window. Because the controller does nothing in this version of the model, the control errors and forces are not important.

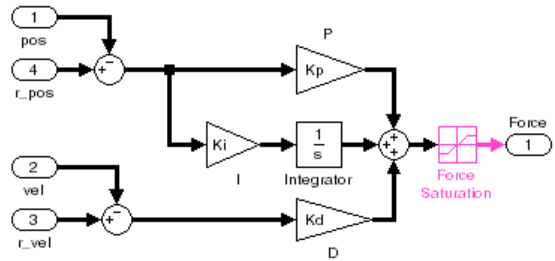
Improper and Biproper PID Controllers

Now change the model to control the Stewart platform's motion with the linear proportional-integral-derivative (PID) feedback system.

The initial controller settings are discussed in "Modeling Controllers" on page 4-15 and "Initializing the Stewart Platform" on page 4-18. Here you implement two versions of this controller, improper and biproper. See "Analyzing the PID Controller Response" on page 4-46 for more.

Switching to the PID Controller Subsystem

Switch the model's controller subsystem by right-clicking on the (green) control subsystem block, selecting **Block Choice**, then **PID Controller**. The block name changes from Null Controller to PID Controller. Open it.



Stewart Platform PID Controller Subsystem

This is the PID linear feedback control system, a copy of the original subsystem contained in the mech_stewart_controllers model library. The control transfer function has the form $K_i/s + K_d s + K_p$. The control gains K_i, K_p, and K_d in their respective blocks reference the variables `Ki`, `Kp`, `Kd` defined in your workspace. Check their initialized values:

```
Ki, Kp, Kd
     Ki = 10000
     Kp = 2000000
     Kd = 45000
```

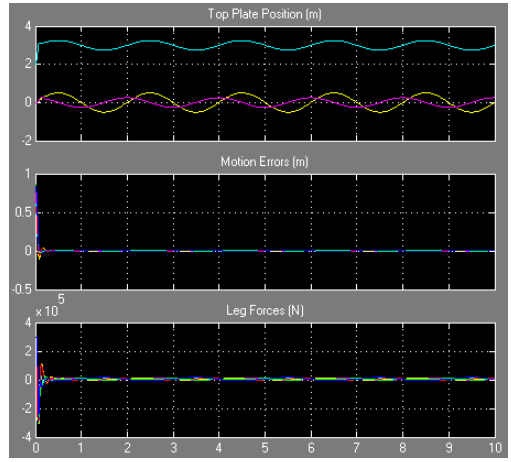
Simulating the Controlled Motion

Simulate the Stewart platform with the PID controller.

1. Open the Scope and start the simulation.

2. Observe the controlled Stewart platform motion. The Scope shows how the platform initially does not follow the reference trajectory, which starts in a different position from the platform's home configuration. The motion errors and forces on the legs are significant. Observe also that the leg forces saturate during the initial transient.

The platform moves quickly to synchronize with the reference trajectory, and the leg forces and motion errors become much smaller.



Stewart Platform Motion and Forces with the PID Controller

Finding the Numerical Derivative of the True and Reference Trajectories

The PID control law requires the time derivative of both actual and reference motion. For greater realism, the Stewart platform plant uses a Body Sensor block to detect only the actual position of the platform, leaving the velocity to be computed by the controller. Finding the reference and actual velocities requires taking numerical derivatives of the reference and actual trajectories, which each consists of the six leg lengths as functions of time.

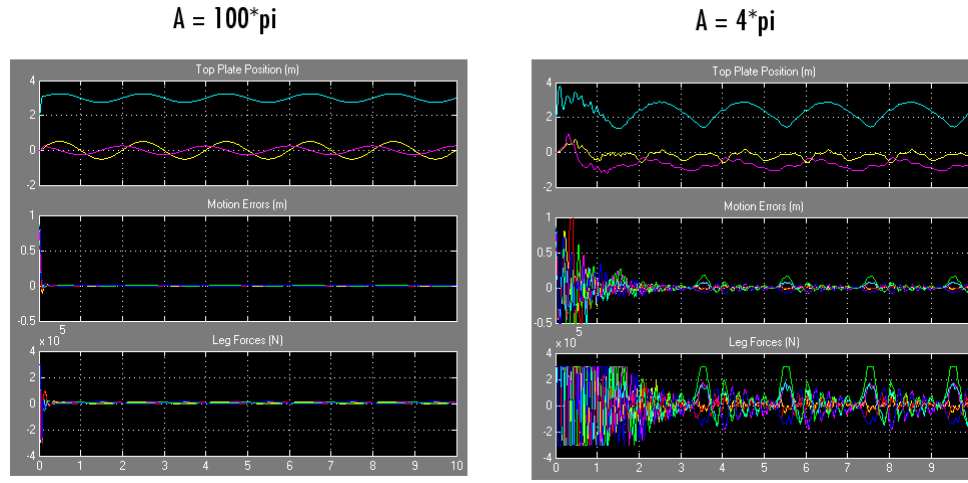
The model gives you two ways to do this. You can switch the numerical derivative configurable subsystem to implement either. This block is linked to the library mech_stewart_control_deriv, which contains the two subsystem implementations. Right-click the numerical derivative (orange) block and select **Block Choice**, then **Derivative Block** or **Filtered Derivative**.

- The first choice (improper) uses the Derivative block of Simulink. This block gives accurate but idealized results. This choice is the default.

- The second choice (biproper) applies a filter of Transfer Fcn blocks in the Laplace domain before transforming the signals back to the time domain. This choice is closer to a realistic implementation.

The transfer function has canonical form $As/(s+A)$. The transfer function acts as a low-frequency bandpass filter to damp out details of the derivative on time scales shorter than $2\pi/A$. The Transfer Fcn blocks use the workspace variable `A` representing A. Its value should be set to about 50 to 100 times the motion frequency variable `freq`. Keep the Transfer Fcn numerators and denominators in their canonical form in terms of `A`. The initialized value is $A = 100*\pi$.

The transfer function filtered derivative is more realistic, at the cost of some inaccuracy due to transients. Vary the filtered derivative behavior by adjusting `A` in your workspace. The unwanted transient behavior is worse for smaller A.



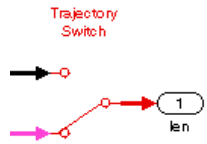
Stewart Platform Motion and Forces with PID Controller (Filtered Derivative)

Simulating at Symmetric Equilibrium

The Stewart platform's home configuration is the symmetric equilibrium of the top plate. Later in this study, you need to simulate the platform at rest. If you start the model in this state, the control forces are zero and the top plate does not move.

Keep the Filtered Derivative option and simulate this static trajectory.

1. Open the Leg Reference Trajectory subsystem. Locate the Trajectory Switch to the right. Double-click the Switch to the down position.



   The reference trajectory now specifies a static reference trajectory: a platform remaining still with all legs at the same constant length.

2. Close the subsystem and start the simulation. Observe the static platform in either the Scope, the visualization window, or both.

3. After rerunning the model, reset the Trajectory Switch back to up.

Analyzing the PID Controller Response

Note This section requires Control System Toolbox.

You can learn more about the effect of the PID controller on the Stewart platform's motion with two control theory techniques, the *s*-plane and the frequency response, both based on the Laplace transform. See "References" on page 4-5 and the documentation for Control System Toolbox for more information.

Improper PID Controller: Theory

The PID control law is an output-input relation whose transfer function is

$$C(s) = K_p + K_i/s + K_d s = (K_p s + K_i + K_d s^2)/s = K(s - s_+)(s - s_-)/s$$

where the gains K are real and nonnegative. The third version is the zero-pole-gain form.

C(s) is *improper*, rising without limit for large *s* and having more zeros (two) than poles (one, at $s = 0$). The poles determine controller response for longer times. The zeros modify how fast the controller approaches the steady state, especially if a zero approaches and nearly cancels a pole. Obtain the steady-state by multiplying the transfer function by *s*, then letting *s* vanish.

In the PID control law, the K_i gain is the steady-state response. The transient behavior is most strongly influenced by the highest power of *s* (the K_d term), then by the next power of *s* (the K_p term), and so on. As you vary the gains, different behaviors emerge.

- If K_i vanishes, the response is all transient, with a null steady state. One zero coincides with and cancels the pole. The other zero is $-K_p/K_d$.
- If K_d vanishes, only one zero remains, at $s = -K_i/K_p$.
- If $4K_dK_i > K_p^2$, the zeros become complex and move off the real *s*-axis.
- If the gain is more in higher powers of *s*, the transient response is stronger.
- If the gain is more in the lower powers of *s*, the transient response is suppressed and the steady-state response emerges more quickly.

Filtered Derivative and Proper PID Controller: Theory

The simple PID control law, with an ordinary derivative, gives rise to an improper transfer function *C(s)*. Changing the ordinary derivative to a filtered derivative softens the behavior of the modified controller *c(s)* at large *s*.

$$c(s) = K_p + K_i/s + K_d As/(s + A) = [K_p s(s + A) + K_i(s + A) + K_d As^2]/s(s + A)$$
$$= [(K_p + K_d A)s^2 + (K_i + K_p A)s + K_i A]/(s^2 + As)$$

This function is *biproper*, having two zeros and two poles, respectively, at

$$s_0 = 0,\ -A$$
$$s_{\pm} = -\left(\frac{K_p + K_i/A}{2\left(K_d + K_p/A\right)}\right)\left[1 \pm \sqrt{1 - 4\left(\frac{K_d + K_p/A}{K_p + K_i/A}\right)\left(\frac{K_i}{K_p + K_i/A}\right)}\right]$$

Recover the improper control law *C(s)* by letting $A \to \infty$.

PID Controller: Alternative Forms

Because *C(s)* is improper, Control System Toolbox cannot fully analyze the simple PID controller response. However, the filtered derivative alternative *c(s)* yields results similar to the ordinary derivative. A complete analysis of *c(s)* is possible.

With the `tf` command, define linear, time-invariant (LTI) transfer function objects for *C(s)* and *c(s)*, then analyze them with the LTI Viewer.

```
numC = [Kd Kp Ki]; % Improper numerator
denomC = [1 0]; % Improper denominator
cImproper = tf(numC,denomC) % Improper transfer function

numc = [Kd*A+Kp Kp*A+Ki Ki*A]; % Biproper numerator
denomc = [1 A 0]; % Biproper denominator
cBiproper = tf(numc,denomc) % Biproper transfer function
```

You can also convert *C(s)* and *c(s)* to state-space and zero-pole-gain (ZPK) forms. The latter is especially useful. Enter `help zpk` for more details.

```
zpk(cImproper) % Convert cImproper to zero-pole-gain
zpk(cBiproper) % Convert cBiproper to zero-pole-gain
```

The helpful `zpkdata` function extracts the zeros, poles, and gain from a ZPK-form controller.

PID Controller: LTI Analysis

Now open the LTI Viewer interface by entering `ltiview`.

1. Select the **File** menu, then **Import**. The **Import System Data** dialog opens.

   In the **Import from area**, select the **Workspace** option and, under **Systems in Workspace**, both entries, `cImproper` and `cBiproper`. Click **OK**.

2. Right-click within the LTI Viewer plot window to view the analysis options under **Plot Types** and **Characteristics**.

With *c(s)*, you can use all the LTI Viewer features. Your valid options for analyzing *C(s)* are limited.

- The Bode and Bode Magnitude plots show the frequency response *C(s)* for imaginary $s = j\omega$.
- The Pole/Zero plot shows the location of the poles and zeros of *C(s)*.

3 Display both the *C* and *c* systems simultaneously and compare the Bode and Pole/Zero plots.

The Bode plots are similar for small *s* (long times). For large *s* (short times), *C(s)* rises without limit, while *c(s)* levels off and results in better controller behavior.

The Pole/Zero plots show that *C(s)* has one pole and *c(s)* two poles, the common one being 0. Both transfer functions have two zeros. You can locate all of these with the `pole` and `zero` functions. Note that one zero is almost identical between *C(s)* and *c(s)*, while the other is shifted dramatically. This shift changes and softens the transient behavior of *c(s)* compared to *C(s)* for larger *s* (short times).

4 Examine the Step and Impulse plots for *c(s)* as well. These plots indicate the time behavior of the *c(s)* controller for stepped and impulsive inputs.

Designing and Improving Controllers

In this section...

Creating Improved Controllers for the Stewart Platform

Note Before working through this study, consult the control design preliminary, "About Controllers and Plants" on page 4-35, and work through the first control design study, "Analyzing Controllers" on page 4-39. This study builds on the results and concepts of the latter.

In addition to Simulink and the SimMechanics product, this study use these products:

- Control System Toolbox
- Simulink Control Design
- Robust Control Toolbox

This second control design case study begins by showing you how to create and optimize a new PID controller. It starts with the creation of a new PID controller *ab initio*, locates a steady state and linearizes the platform's motion about this equilibrium, and adjusts the linearized platform dynamics to optimize the new PID controller. The study ends by introducing multivariable synthesis as a step beyond PID control, implementing a more complex and realistic multivariable controller and comparing its performance with the new PID controller.

Designing a New PID Controller

Note This section requires Control System Toolbox. Saving intermediate model versions and workspace values is recommended.

The PID controller gains set by the initialization M-file are preoptimized. The preceding case study, "Analyzing Controllers" on page 4-39, uses these gain values as examples.

In the rest of this study, you follow a more realistic scenario where the gains are not initially known and you use control design tools in the MATLAB environment to create and optimize a filtered PID controller.

Making a First Guess for the Controller Gain

Make an initial guess for the integrator (I) gain K_i with dimensional analysis. K_i has dimensions force/length/time.

- An initial guess for the force is one-sixth the weight of the platform and legs.
- An initial guess for the length is range of vertical motion in the reference trajectory.
- An initial guess for 1/time is the natural frequency, $\pi/2\pi = 0.5$ Hz.

Thus an initial guess for the integrator gain is

```
Ki = 0.5*9.8*(top_mass/6+(upper_leg_mass+lower_leg_mass))/0.3

Ki = 7.1680e+003
```

Making a First Guess for the Controller Force

The initialization M-script sets the workspace variable `initCondI` to the value needed to put the platform in a symmetric equilibrium in the initial state. With a new K_i value, you need to recalibrate this initial condition.

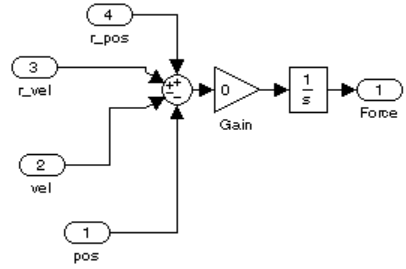
```
initCondI = ...
   (upper_leg_mass+lower_leg_mass+(top_mass*1.3/6))*9.81/Ki

initCondI = 0.6839
```

Modifying the Null Controller with a Constant Force

Start by turning off the PID controller and applying a constant force to the platform.

1. Right-click the controller subsystem. Select **Block Choice > Null Controller**.
2. Right-click **Null Controller** again. Select **Link Options > Go To Library Block**.

   The configurable subsystem library mech_stewart_controller opens.
3. Under **Edit**, select **Unlock Library**. Open the Null Controller template subsystem.
4. In the subsystem, between the Gain and Force (Output) blocks, insert an Integrator block.



5. Open the Integrator dialog. For **Initial condition**, enter `Ki*initCondI`. Click **OK**.
6. Close Null Controller. Save and close the mech_stewart_controller library.
7. Back in mech_stewart_control, update the diagram (**Ctrl+D**).
8. At the command line, enter `Ki*initCondI`.

   This is your first guess for the controller force in one leg: the product of your PID integrator (I) gain guess and your controller initial state guess.

Simulating the Platform with the Constant Force

Now observe the effect of this constant force on the platform.

1. In the Leg Reference Trajectory subsystem, set the Trajectory Switch position to down.
2. Open the Scope and start the simulation. The control force is less than the platform weight. The platform accelerates downward.

Trimming and Linearizing the Platform Motion

Note This section requires Control System Toolbox and Simulink Control Design. Saving intermediate model versions and workspace values is recommended.

A critical step in control design is to understand the response of a plant being controlled to small disturbances in its motion [5]. This step requires

- Trimming the platform, or finding an *operating point*. This is a time trajectory satisfying certain prior conditions that you specify.

  Here you search for the simple, useful operating point of symmetric equilibrium, where the platform does not move.

- Linearizing the platform motion about the operating point.

  You save the results of the linearization to use in the next section, "Improving the New PID Controller" on page 4-59.

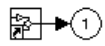
For More About Trimming

As described in "Trimming and Linearizing Through Inverse Dynamics" on page 4-24, you can trim SimMechanics models in many ways. Control System Toolbox and Simulink Control Design provide linear analysis tools richer and more powerful than what Simulink and SimMechanics software alone offer.

Setting Up the Model for Trimming

Now set up the model for trimming. In Trimming mode, the model's mechanical states are the 40 constraints that reduce the 52 free (forward dynamics) states to the 12 independent states.

1. Make sure the model observes these settings.
   a. Keep the controller subsystem **Block Choice** set to **Null Controller** and the derivative type to **Filtered Derivative**.
   b. Keep the Trajectory Switch down (static trajectory) in the Leg Reference Trajectory subsystem.
2. Reset the SimMechanics analysis mode to trimming.
   a. Open the Plant subsystem. Double-click the orange Machine Environment block. Locate the **Parameters** tab.
   b. For **Analysis mode**, change the pull-down menu to `Trimming`. Click **OK** and close the subsystem.
3. Observe the trimming output blocks that have appeared in the upper left of the main model.



Locating an Operating Point by Trimming

Next, locate an operating point for the Stewart platform plant.

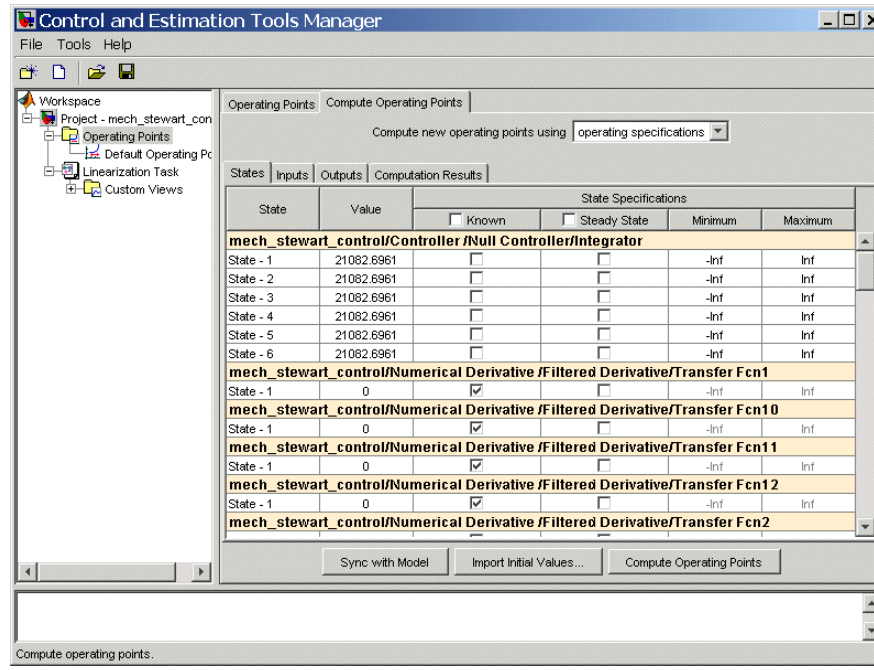
1. Select linearization points in your model as follows. Right-click, in turn, on each of the Simulink signal lines defining the input and output of the Plant subsystem:
   - Leg Forces (input)
   - Pos (output)

   On each signal line's right-click menu, under **Linearization Points**, select
   - Both **Input Point** and **Open Loop** for the input line
   - Both **Output Point** and **Open Loop** for the output line

Choosing the open-loop property for these signals breaks the feedback loop from controller to plant back to controller. The plant instead takes a given set of externally imposed controller forces.

2 Then, from the model menu bar, select **Tools > Control Design > Linear Analysis**. The **Control and Estimation Tools Manager** window opens.

3 To the left of the Manager window, select the **Operating Points** node. Then, to the right, select the **Compute Operating Points** tab. Click the **Sync with Model** button at the bottom of the tab.



The default subtab is **States**. The **Steady State** check boxes are selected by default. This choice searches for a plant operating point where the platform is at rest relative to its initial configuration.

4 Examine the states by scrolling down in the **States** window.

- There are six states associated with the null controller Integrator block.

Clear the **Steady State** check boxes for these states. The trimming will not hold the controller signal as fixed.

- Below these six are twelve states associated with the Transfer Fcn blocks in the Filtered Derivative subsystem.

  Free them from being fixed by clearing their **Steady State** check boxes. Make their values (0) known by selecting their **Known** check boxes.

The rest of the states are associated with the positions and velocities of the Stewart platform leg joints. Only six of these states are independent. The others are constrained. Leave their settings as the defaults.

5 Move to the **Outputs** subtab. Under **Output Specifications**, select the **Known** check box (the topmost check box in that column). This action specifies all outputs, the state deviations from the desired operating point. There are 40 states (constraints) in Trimming mode.

The output values are specified in the **Value** column. The values are all zero, indicating that all constraints on states (the specifications of the operating point) must be satisfied within tolerance.

States | Inputs | Outputs | Computation Results

Output	Value	Output Specifications		
		☑ Known	Minimum	Maximum
mech_stewart_control/MSB Trimming Out				
Channel - 1	0	☑	-Inf	Inf
Channel - 2	0	☑	-Inf	Inf
Channel - 3	0	☑	-Inf	Inf
Channel - 4	0	☑	-Inf	Inf
Channel - 5	0	☑	-Inf	Inf
Channel - 6	0	☑	-Inf	Inf
Channel - 7	0	☑	-Inf	Inf
Channel - 8	0	☑	-Inf	Inf
Channel - 9	0	☑	-Inf	Inf
Channel - 10	0	☑	-Inf	Inf
Channel - 11	0	☑	-Inf	Inf
Channel - 12	0	☑	-Inf	Inf
Channel - 13	0	☑	-Inf	Inf
Channel - 14	0	☑	-Inf	Inf
Channel - 15	0	☑	-Inf	Inf

6 From the Manager window menu bar, select **Tools > Options**. The **Options** window opens. Select the **Operating Point Search** tab.

In the **Optimization Method** area, select `Nonlinear least squares` in the **Optimization Method** menu.

Leave the other defaults. Click **OK**. The **Options** window closes.

7 Back in the **Control and Estimation Tools Manager**, click the **Compute Operating Points** button at the bottom of the **Compute Operating Points** tab.

The **Computation Results** subtab indicates the progress of the trimming. When finished, it should indicate that the operating point specifications were successfully met.

In the **Operating Points** node to the left, a new **Operating Point** subnode appears, `Operating Point`, containing the results of this trimming.

Interpreting and Saving the Operating Point

Examine and save the operating point results.

1 Click **Operating Point**. Look at the **States** and **Outputs** tabs.

Under **Outputs**, the **Desired dx** values (if not marked `N/A`) are zero. For the mechanical states (constraints), the **Actual dx** values (deviations from the requested operating point) are zero within tolerance.

This is not true for the Controller states, which you did not require to vanish. The Filtered Derivative states are all zero.

2 Save this operating point by right-clicking `Operating Point` and selecting **Export**. Except for the name, leave the defaults.

For **Variable Name**, enter `oppoint_PLANT`. Click **OK**.

You now have a workspace object (`opcond.OperatingPoint` class) called `oppoint_PLANT` representing the plant holding still at the start of simulation (t=0). Retain this object for later use.

3 Examine its states by entering

```
oppoint_PLANT % List plant states at t=0
```

4 Reset the controller initial condition to the new operating point.

```
initCondI = oppoint_PLANT.States(1).x(1);
```

Linearizing the Platform Motion at the Operating Point

Now switch the model back to Forward Dynamics mode. The mechanical states are now the 52 tree states corresponding to the uncut joint primitives.

1. Open the Plant subsystem, then its orange Machine Environment block. Locate the **Parameters** tab.

2. In the **Analysis mode** pull-down menu, select `Forward Dynamics`. Click **OK** and close the subsystem.

Then linearize the plant motion about the operating point you specified in earlier. Return to the Control and Estimation Tools Manager.

1. Select **Tools > Options**. In the **Options** dialog, select the **Linearization State Ordering** tab.

   Click the **Sync with Model** button at the bottom, then click **OK**.

2. Now select the **Linearization Task** node to left, then the **Operating Points** tab. Select the **Operating Point** called `Operating Point`.

3. At the bottom of the tab, make sure the **Plot linear analysis result in a** check box is selected. Then choose a plot type in the pull-down menu. For example, pick `Bode response plot`.

4. Then click the **Linearize Model** button. The LTI Viewer opens with a large family of Bode response plots.

   For later reference, you can choose other response plot types by right-clicking on one of the plots and, under **Plot Type**, selecting a different plot, such as **Bode**, **Step**, or **Impulse**. (You do not need to go back to **Linearization** and relinearize the model.)

Interpreting and Saving the Linearization Results

This plant linearization started with six inputs (the leg forces) and 12 outputs (six leg positions and six leg velocities). The LTI Viewer displays 6 x 12 = 72 response plots. To view one plot individually,

1. Right-click any one of the 72 plots and select **I/O Selector**. The **I/O Selector** dialog opens.

2 This dialog lets you to choose any response of one output relative to one input. To see that plot in the LTI Viewer, click the corresponding black dot.

Each plot shows how one of the outputs (a position or velocity) responds to the application of a small force in one of the input channels. Different plot types (impulse, step, Bode, etc.) yield different aspects of the response.

Export the results of your linearization.

1 Select **File > Export** in the LTI Viewer.

2 Choose your model and give it a unique name (call it `sys`) under **Export As**.

3 Click **Export to Workspace**. The model is saved as an LTI object. The variable class is `ss`, the canonical state space form used by Simulink.

Retain this LTI object for the next section, where you use it to improve the PID controller.

Further Suggestions

You can apply these results to other controllers (see "Synthesizing a Robust, Multichannel Controller" on page 4-66), as well as choose other operating points.

Improving the New PID Controller

Note This section requires Control System Toolbox and Simulink Control Design. Saving intermediate model versions and workspace values is recommended.

To proceed with this section, you need to have completed the preceding section, "Trimming and Linearizing the Platform Motion" on page 4-53.

In this section, you use the linearization results to create a controller to better match the plant. This information allows you to convert open-loop information about the controller and plant into closed-loop behavior of the coupled system.

A PID controller acts as the same controller on each of the platform legs. You can improve the controller's response to each leg's motion by working with the diagonal components of the plant response. These components represent a leg's motion response to the force acting on that leg. This control design paradigm is single-in, single-out (SISO). By symmetry, designing the PID settings with one of the leg's control behavior optimizes them for the other five.

The SISO approach ignores coupling between the legs. The last section of this study, "Synthesizing a Robust, Multichannel Controller" on page 4-66, tackles multichannel coupling to achieve a more accurate controller design.

What You Need from Previous Sections

From the preceding section, "Trimming and Linearizing the Platform Motion" on page 4-53, you should have these saved in your workspace:

- Linearized plant model as an LTI object (`ss` class) called `sys`
- Controller initial condition `initCondI` reset to the operating point
- Useful intermediate model versions and workspace variable MAT-files

Throughout this section, keep the derivative block as **Filtered Derivative** and the PID controller as biproper.

Reducing the State Space with Minimal Realization

Many of the mechanical states in `sys` are constrained. Remove them with the `sminreal` command. This reduction works with the structure of the `sys`, rather than (like `minreal`) with the numerical properties of `sys`.

```
G = sminreal(sys); % Structural reduction of linearized sys
```

`G` now represents the reduced linearized plant.

Exploring PID Gains, Filtered Derivative, and Force Saturation

One way to get a feel for the effect of PID feedback control on the Stewart platform's motion is to vary the gains, frequency cutoff, and force saturation systematically, while holding fixed the reference trajectory and the platform initial conditions.

The larger K_d is relative to K_p and K_i, the more sensitive the controller is to immediate changes in the reference signal. (The same is true of K_p relative to K_i.) The derivative term emphasizes rapid change. On the other hand, if K_d is small, the controller is more sluggish in response. The K_i term emphasizes memory of motion errors past. A fundamental tradeoff of control design is

- A more responsive PID controller is also less stable against high-frequency (short time-scale) disturbances such as noise.
- A more stable controller is less responsive to feedback.

For large filtering constant *A*, the biproper transfer function *c(s)* behaves at small *s* almost exactly like the improper *C(s)*. But as you reduce *A*, *c(s)* behaves less like *C(s)*. In the time domain, for smaller *A*, the controller *c(s)* shows more transient deviation from the pure derivative behavior of *C(s)*.

The PID controller also depends on the force saturation limit, set in the workspace by `force_act_max`. Making the force saturation limit too small means that the controller cannot actuate the legs sufficiently to make them keep up with the reference trajectory signal. The platform motion moves toward instability with a lower force saturation limit. Too low a limit eventually yields motion that is unacceptably extreme or completely unstable. Up to a point, you can compensate for a lower force saturation limit by making the controller more responsive.

Analyzing the Plant Response with the SISO Design Tool

A better way to optimize the PID controller is to analyze the open- and closed-loop machine response with the SISO design tool.

Open the **SISO Design Tool** by entering

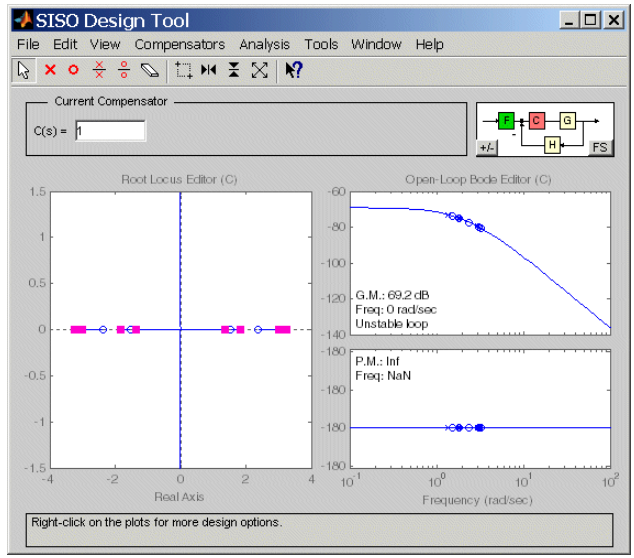
```
sisotool(G(1,1)); % SISO design tool for first leg-leg pair
```

The design tool opens with a unity controller (compensator), `C(s) = 1`. Use the **Help** menu for more information about the design tool, including how to interpret the plot symbols.

The **Root Locus Editor** to the left shows the closed-loop *CG/(1+CG)* response, the *s*-plane poles, zeros, and root-loci. The **Open-Loop Bode**

Editor to the right shows the open-loop *CG* plant response, including poles and zeros.

The closed-loop response has eight poles, four on the left-half and four on the right-half of the *s*-plane, the latter indicating instability. The open-loop Bode plot displays the gain and phase margins.



SISO Design Tool with Stewart Platform Plant at Rest and Unity Controller

Designing a New Biproper PID Controller with the Plant Response

To design a biproper PID controller, add two zeros and two poles and adjust the overall gain. Observe these general rules for the poles and zeros:

- The numerator coefficients, including the overall gain, must be positive. The easiest way to ensure this is for both zeros to have negative real parts.
- One pole must occur at zero. This corresponds to the integrator (I) part.
- The other pole must have negative real part.

To implement,

1. Select **Compensators > Edit > C**. The **Edit Compensator C** dialog opens. Add poles and zeros. Click **OK**. The dialog closes.

2. In the root-locus plot, you can move controller and closed-loop poles and zeros around by dragging them with your mouse. As you move closed-loop poles, you also change the overall controller gain. Be sure to leave the initially stable closed-loop poles in the left half-plane.

   In the Bode editor, you can move open-loop (controller) poles and zeroes by dragging them. You can also change the gain and phase margins.

3. The SISO design tool controller form is $\kappa(1+\alpha s)(1+\beta s)/s(1+\gamma]s)$. The overall control gain κ is K_i in this form.

   For K_i, use the value of your first guess found previously in "Designing a New PID Controller" on page 4-51.

Optimizing the New Biproper PID Controller with the Plant Response

To optimize your controller, change its response to suppress undesirable and enhance desirable feedback. The objectives, typical in control problems, are a high-gain response at low frequencies to achieve tracking performance and a diminishing response at high frequencies to limit the controller's sensitivity to plant variations and noise.

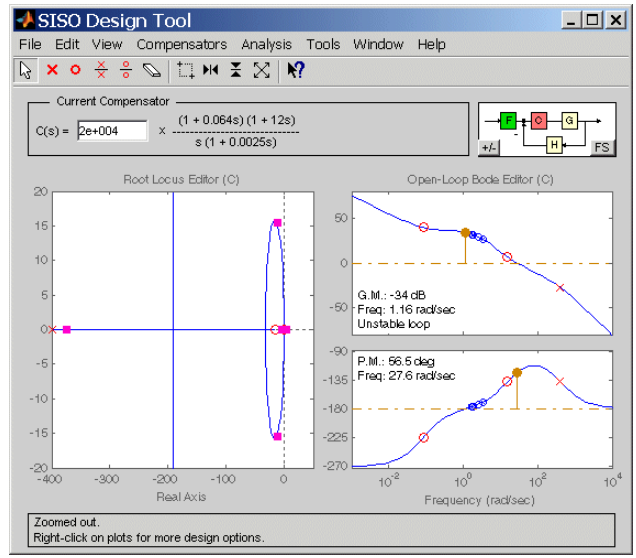
The platform motions have low bandwidth, typically only a few Hertz (Hz). The system should have strong response up to a few Hz (ω = about 10 rad/s), then falling response for higher frequencies.

One controller pole must always remain at zero. Five system poles have positive (unstable) real parts, a result of the first leg coupling to the other five. You cannot eliminate these in a SISO analysis.

Improve the controller by

- Making the nonzero controller pole more negative. This increases A and increases the phase margin while decreasing the gain margin.
- Improving transient response by adjusting the controller zeros.

- Lowering the gain margin by raising the overall Bode response. This increases the overall controller gain $\kappa = K_i$.



Saving the Optimized New Biproper Control Law

Once you have a satisfactory controller, you can export the new optimized biproper control law to the workspace and analyze it there to redefine the filtered PID controller parameters K_i, K_p, K_d, and A.

Export the modified compensator from the SISO design tool.

1. Go to **File > Export**. Select `Compensator`. Rename it `cBiproperOpt` under **Export as**.
2. Then click **Export to Workspace**.

`cBiproperOpt` is a zero-pole-gain form (LTI object of class `zpk`). For example,

```
cBiproperOpt

Zero/pole/gain:
6171074.4994 (s+15.51) (s+0.08378)
```

```
---------------------------------
            s (s+400)
```

Resetting the PID Gains and Derivative Cutoff

Extract the biproper PID controller parameters by inverting the zeros $s_{\pm}$, poles, and gain K. The standard zero-pole-gain form is

$$c(s) = K(s - s_+)(s - s_-)/s(s+A) = [(K_p + AK_d)*s^2 + (K_i + AK_p)*s + AK_i]/s(s + A)$$

- A = the negative of the biproper nonzero pole
- The gains are:

$$K_i = Ks_+s_- \text{ , } K_p = -[K(s_+ + s_-) + K_i]/A \text{ , } K_d = (K - K_p)/A$$

Reset your workspace variables accordingly.

```
[z,p,k] = zpkdata(cBiproperOpt) % Extract ZPK data from cBiproper
A = -p{1,1}(2) % Extract nonzero pole
Ki = k*z{1,1}(1)*z{1,1}(2)/A % Extract  Ki gain
Kp = -(k*(z{1,1}(1) + z{1,1}(2)) + Ki)/A % Extract Kp gain
Kd = (k - Kp)/A % Extract Kd gain
```

Checking the Symmetric Equilibrium

Check that the symmetric equilibrium is stable with your new controller.

1. Make sure the Trajectory Switch is set to down.
2. Update the diagram (**Ctrl+D**) and rerun the model.

   A trim point is rarely exact. There is typically a small but nonzero motion error as the platform relaxes toward equilibrium.

Simulating the Moving Platform and Capturing the Motion Errors

Now test the platform motion with the moving trajectory and your new retuned biproper control law.

1. Set the Trajectory Switch back to up.

2 Restart the model. You should see reasonable motion errors and leg forces, except perhaps for an initial transient.

3 Capture the Motion Errors from the logged signals structure `sigsOut`.

```
pid_opt_TS = sigsOut.('Motion Errors'); % Record motion errors
```

Synthesizing a Robust, Multichannel Controller

Note This part of the study requires Control System Toolbox and Robust Control Toolbox.

To complete this section, you need to have completed the preceding section, "Improving the New PID Controller" on page 4-59.

The controllers you have designed so far in this and the preceding control design studies are based on classical PID techniques, where each channel is subject to the same control law and the control law is tuned one channel at a time. This approach misses the *cross-coupling*, the effect that the force on one platform leg has on the motion of the other legs.

In this section, you redesign the Stewart platform controller by using modern techniques that take multichannel coupling into account and implementing a robust *H-infinity* controller [6], [7].

What You Need from Previous Sections

From preceding sections, you should have these saved in your workspace:

- Reduced state space representation `G` of the plant
- Time series structure `pid_opt_TS`

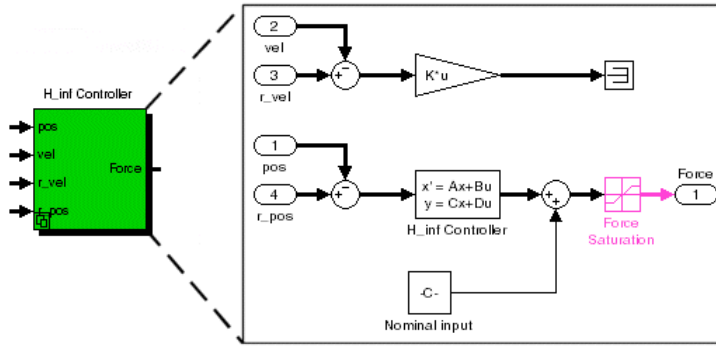
Viewing the H-Infinity Controller

Before starting,

1 From its right-click menu, under **Block choice**, switch the controller subsystem to **H_inf Controller**.

2 Make sure that the derivative subsystem remains set to **Filtered Derivative** and the Trajectory Switch in the Leg Reference Trajectory subsystem is set to up.

Examine the controller subsystem, which is implemented via state space.



Stewart Platform H-Infinity Controller Subsystem

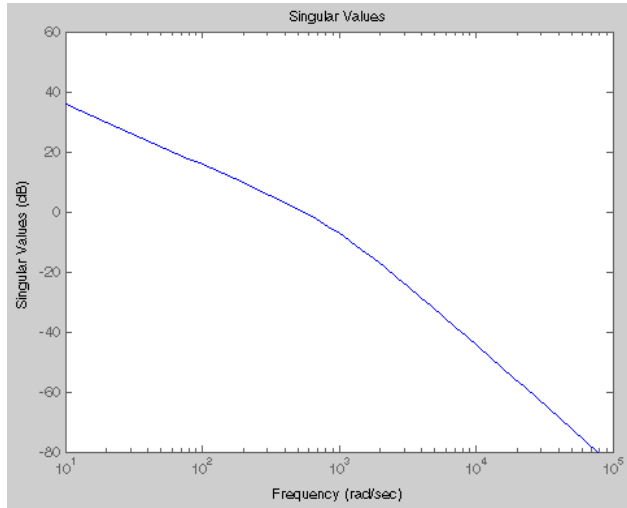
Defining a Desired Loop Shape Response

Start by specifying a desired open-loop response $|C*G(1,1)|$ and plot its singular values. For example,

```
Lsd = zpk([],[-1000 0],612770) % Define desired loop shape

Zero/pole/gain:
  612770
----------
 s (s+1000)

 sigma(Lsd) % Plot singular values
```

View the closed-loop response generated by this loop shape by entering:

```
step(feedback(Lsd,1)) % Feedback step response
```



Desired Loop Shape: Singular Values

Synthesize and Reduce a Controller with the Desired Loop Shape

Now create a controller using the desired loop shape and plant response:

```
[K_ls,CL,GAM,INFO] = loopsyn(G,Lsd); % Synthesize controller
```

Check the size of the controller by entering

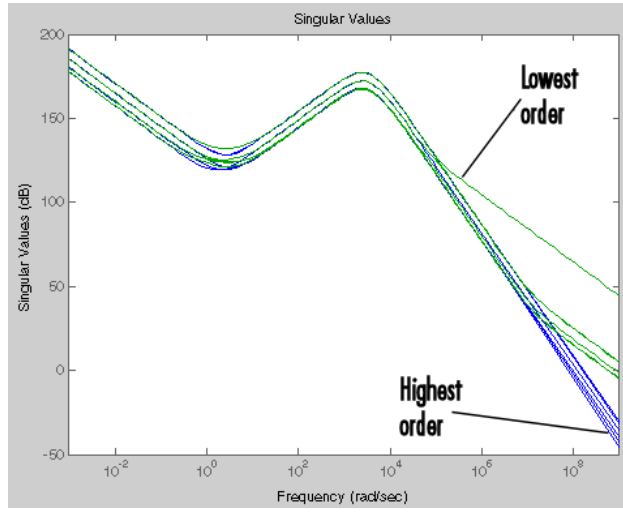
```
size(K_ls) % Check size of loopsyn controller
```

The example controller has 48 states. It is usually impractical to implement a controller of such high order and computational intensity. So try reducing the controller to 24th order:

```
Kr_ls = reduce(K_ls,24); % Reduce controller order
```

To estimate how many states you can ignore (truncate), plot both the full and reduced singular values

```
sigma(K_ls,Kr_ls) % Plot singular values
```



Full and Reduced Loop-Synthesized Controllers: Singular Values

Simulating the Robust Controller and Capturing Its Motion Errors

From the synthesized loop shape, extract the matrices needed to define the state space model used in the H_inf Controller subsystem.

```
[Ak,Bk,Ck,Dk] = ssdata(Kr_ls); % Extract state space model
```

Run the loop-synthesized controller model. Then capture the motion errors.

```
loopsyn_TS = sigsOut.('Motion Errors'); % Record motion errors
```

Plotting and Comparing the Results

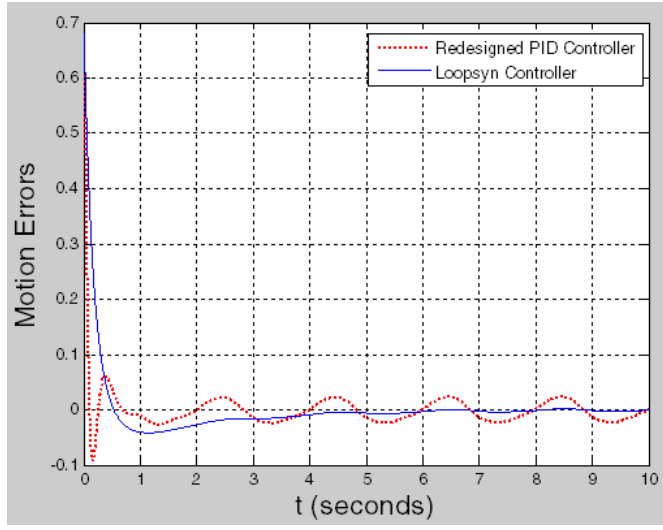
Finally, compare the motion error data from the two controllers:

- Redesigned PID
- Robust loop-synthesized

At the command line, enter:

```
figure
plot(pid_opt_TS.Time,pid_opt_TS.Data(1,:),'r', ...
   loopsyn_TS.Time,loopsyn_TS.Data(1,:),'b')
ylabel('Motion Errors','FontSize',16)
xlabel('t (seconds)','FontSize',16)
legend('Redesigned PID Controller','Loopsyn Controller')
```

Apart from the initial transient, the loop-synthesized controller performs better than the redesigned PID controller. In this example, the late-time robust controller motion errors are more than an order of magnitude smaller and exhibit no oscillatory "ringing."



Redesigned PID and Loop-Synthesized Control System Motion Errors

Generating and Simulating with Code

In this section...

About the Stewart Platform Code Generation Examples

Note This study requires some experience with the code generation features of Simulink. To complete it, you need to have Real-Time Workshop installed, in addition to the SimMechanics product.

This case study leads you through a representative set of tasks related to turning a Stewart platform model into generated code. After you read the introductory sections, proceed with the case study tasks. All code generation-related files and subdirectories are created in your current MATLAB folder.

1. Generating an S-Function block for the plant
2. Model referencing the plant
3. Generating stand-alone code for the controller and plant together

For More Information About Code Generation

To learn more about generating code from Simulink models, consult the documentation for Simulink and Real-Time Workshop.

To learn more about SimMechanics code generation, see “Generating Code” on page 2-38.

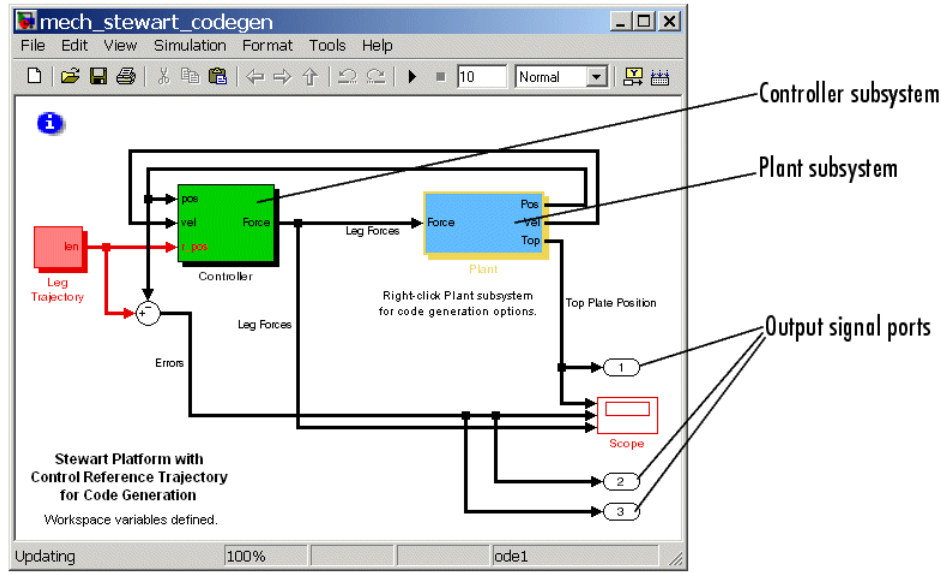
Learning About the Model

This study is based on these demo files, in addition to the initialization M-files. Copy them into an empty folder before starting each case study task.

File	Purpose
`mech_stewart_codegen`	Basic model
`mech_stewart_codegen_plant`	Plant subsystem as separate model

You use the second model file later for model reference in “Model Referencing the Plant” on page 4-77.

Open the first model. Then update the model by pressing **Ctrl+D** at the keyboard.



Solver and Sample Time Step Sizes

The model defines two time steps, `dt1` and `dt2`. The model initializes both to the same value, but you can make them different (see the table, Configuration Parameters for Stewart Platform Code Generation on page 4-74):

- `dt1` defines the fixed-step solver time step.
- `dt2` defines the sample rate for the generation of the trajectory signals. It must be an integer multiple of `dt1`.

Structure of the Model

The two major parts of the model are the PID controller and the plant. You can generate code from the entire model or from only part of it. In this study, you convert the plant subsystem to code in two ways, by an S-function block and by model reference. You then convert the whole model to stand-alone code.

Run the model before continuing with code generation. You can view the Stewart platform motion by opening the Scope. You can also enable visualization. (See the *SimMechanics Visualization and Import Guide*.)

Caution To convert a subsystem alone to code requires placing all of the SimMechanics blocks (the blocks with the distinctive Physical Modeling connector ports ● and Body Coordinate System ports ⚿) into the subsystem. mech_stewart_codegen encapsulates all SimMechanics blocks in the Plant subsystem.

Simulation Settings for Code Generation

Some of the Simulink and SimMechanics settings in mech_stewart_codegen are different from the defaults.

From the model's **Format** menu, check that these entries are selected:

- **Port/Signal Displays > Wide Nonscalar Lines**
- **Sample Time Display > Colors**

Other settings are optimized for code generation.

1 View the Plant subsystem parameters by right-clicking the subsystem and selecting **Subsystem Parameters**, then close the dialog.

   - **Treat as atomic unit** is selected.
   - **Minimize algebraic loop occurrences** is selected.

2 Now view the Configuration Parameters dialog by selecting it from the model's **Simulation** menu. View the different nodes, then close the dialog.

   - **Solver** node. The model uses a fixed-step solver. While S-function Target does not require fixed-step solvers, most Real-Time Workshop targets require fixed-step solvers.
   - **Data Import/Export** node. Time, states, and output are selected for data export.
   - Outputs correspond to the ports connected to the outport signals.

     Top Plate Position: translation and rotation (Port 1)

     Errors: difference of reference and actual top plate positions (Port 2)

     Leg Forces: control forces parallel to each Stewart platform leg (Port 3)
   - The states represent the states of the controller and the plant. The controller has Simulink states, and the plant has SimMechanics mechanical states.

     The model states are not identical to the system's independent degrees of freedom (DoFs). See "Counting Degrees of Freedom in the Stewart Platform" on page 4-8 and "Identifying the Simulink and Mechanical States of the Stewart Platform" on page 4-21.

Configuration Parameters for Stewart Platform Code Generation

Node	Settings
Solver	**Solver options**: **Type**: `Fixed-step` **Solver options**: **Solver**: `ode1 (Euler)` **Fixed-step size**: `dt1` (`5e-3` seconds)
Data Import/Export	**Time**: `tout` **States**: `xout` **Output**: `yout`

Configuration Parameters for Stewart Platform Code Generation (Continued)

Node	Settings
Optimization	**Simulation and code generation**: **Inline parameters** selected (needed for Model Reference)
Model Referencing	**Minimize algebraic loop occurrences** selected
Real-Time Workshop	**Target selection**: **System target file**:`ert.tlc (no auto configuration)` (Embedded Real-Time Target) **Interface**: **Software environment**: **continuous time** selected **Interface**: **Verification**: **MAT-file logging** selected
SimMechanics	**Diagnostics**: all cleared **Visualization**: all cleared

3 Now open the Plant subsystem and its orange Machine Environment block. Check the following settings, then close the dialog.

The constraint solver is set to stabilizing, a robust choice appropriate for a fixed-step simulation of moderate computational cost. Robust singularity handling is selected.

Machine Environment Settings for Stewart Platform Code Generation

Tab	Settings
Parameters	**Linear assembly tolerance**: `1e-3 m` **Angular assembly tolerance**: `1e-2 rad`
Constraints	**Constraint solver type**: `Stabilizing` **Use robust singularity handling** selected
Visualization	**Visualize machine** selected

Generating an S-Function Block for the Plant

The S-function Target feature of Real-Time Workshop lets you generate an S-function block for a subsystem. This block points to a (non-stand-alone) auxiliary binary file that hides the original subsystem. You can then use the S-function block in multiple instances in any Simulink model, including your original one, without SimMechanics software.

1. Right-click on the Plant subsystem. Select **Real-Time Workshop**, then click **Generate S-Function** in the submenu.

   A new window opens, **Generate S-function for Subsystem: Plant**, listing the workspace variables used in the subsystem. At this point, you can make ordinary Simulink parameters tunable, but you cannot tune SimMechanics parameters. See “Generating Code” on page 2-38.

2. Proceed with generating the code files by clicking **Build** in the tunable parameter window. Follow the generation in the command window.

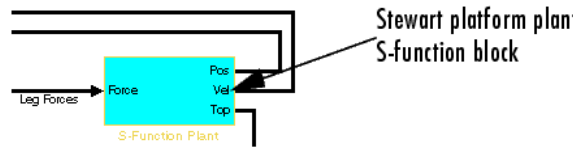
   Two auxiliary subdirectories are created, as well as C source and header files and a (non-stand-alone) linked binary. Each of these files has a name, `Plant_sf`, derived from the subsystem name.

   A new Simulink model window also appears, containing the new, reusable S-function block named Plant that points to the linked binary. Rename this block to S-Function Plant.

3. From the original mech_stewart_codegen model window, cut the Plant subsystem. Paste it into the new, untitled window.

   Save this new model, containing the S-function and subsystem blocks for future use, as mech_stewart_codegen_plant_sfunc.

4. Copy the S-Function Plant block from the new untitled window into the original model window. Connect the signal lines to the S-function block.



5 Now start the model with the new S-function block. This modified model no longer requires SimMechanics software. The performance is about the same as the original model with the subsystem.

   Save the modified model for future use as mech_stewart_codegen_sfunc.

Model Referencing the Plant

Real-Time Workshop gives you another way to generate code for a subsystem. Using the Model Reference feature, you can put the subsystem in a separate model, then replace the subsystem block in the original model with a model reference block that points to the new model holding the subsystem. For mech_stewart_codegen, the plant subsystem is contained in the model file mech_stewart_codegen_plant.

One advantage of model referencing is that it allows you to incrementally compile parts of your model, one at a time. This feature saves significant time when you generate code from large models.

Simulation Settings for Model Reference

Some settings in mech_stewart_codegen_plant differ from the defaults. Many differ in the same way that mech_stewart_codegen does. Here are additional settings in the Configuration Parameters of this model that differ from the defaults.

Node	Settings
Model Referencing	**Rebuild options for all referenced models**: **Rebuild options**: `Never` **Rebuild options for all referenced models**: **Never rebuild targets diagnostic**: `None`
Real-Time Workshop	**Interface**: **Code interface**: **Single output/update function** cleared

Setting Up and Running the Main Model for Model Reference

To reconstruct your model for model referencing,

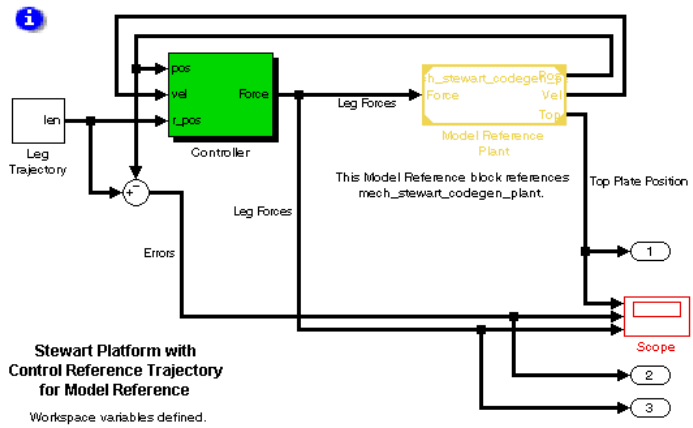
1 In mech_stewart_codegen, cut the Plant subsystem.

Keep this subsystem for future use. From the preceding part of the study, "Generating an S-Function Block for the Plant" on page 4-76, you have a model, mech_stewart_codegen_plant_sfunc, with the subsystem. If you have not already saved the subsystem here, do so now by pasting it in.

2 From the Simulink Ports & Subsystems library, drag and drop a Model block into mech_stewart_codegen. Rename the block Model Reference Plant.

3 Open the Model Reference Plant block dialog. In the **Model name** field, enter `mech_stewart_codegen_plant`. (Note the default **Simulation mode** is `Accelerator`.) Click **OK**.

Save the new, modified model as mech_stewart_codegen_modelref.



Stewart Platform with Control Reference Trajectory for Model Reference

4 In the toolbar of mech_stewart_codegen_modelref, click the **Start** button.

In the command window, watch the code generation for model referencing. When it finishes, the simulation starts. Watch the simulation results by opening the Scope block.

Running or updating the main model generates a code folder and a non-stand-alone (linked) binary file called mech_stewart_codegen_plant_msf from the referenced model. The model reference block in the referencing model points to this binary. You can view the model reference code generation

in the command window each time you update the diagram. Once it references the external plant model, double-clicking the Model Reference Plant block opens that model.

Note In the model reference block (Model Reference Plant), this model uses the default simulation mode, Model Reference Accelerator.

Generating Stand-Alone Code for the Whole Model

In this section, you generate a stand-alone executable from the original Stewart platform model, mech_stewart_codegen, using Real-Time Workshop and the Embedded Real-Time target. This executable is portable and independent of MATLAB.

1. From the **Tools** menu in the model menu bar, select **Real-Time Workshop**, then **Build Model**. The build process begins in the command window.

   Real-Time Workshop generates two auxiliary subdirectories, as well as a stand-alone executable named mech_stewart_codegen.

2. Start the executable by entering

   ```
   !mech_stewart_codegen
   ```

   A MAT-file called `mech_stewart_codegen.mat` is created whenever you run the executable. This file contains the output, state, and time data exported from the model.

3. You can load this MAT-file into your workspace and examine its variables, all distinguished by the `rt_` prefix.

   From the MATLAB Desktop **Current Folder** window, right-click `mech_stewart_codegen.mat` and select **Import Data**. The **Import Wizard** appears, listing the variables that were generated, at each time step, by running the executable. These include:

   - `rt_tout`: Simulation times
   - `rt_xout`: States

- `rt_yout`: Outputs

4 Click **Finish** on the **Import Wizard** dialog. The variables are loaded into your workspace.

   Examine the variables in workspace and double-click each of the three. The editor displays the variable values as arrays (in the two cases of outputs and states, arrays as parts of data structures).

These variables include:

- The variable `rt_tout` containing the simulation times.
- The variable `rt_xout` containing the state signals. These states include a six-column array representing the Controller subsystem states and a 52-column array representing the mechanical states of the Plant.
  - The six controller states are the six leg positions integrated by the Controller/Integrator block for PID control.
  - The 52 mechanical states are discussed in "Identifying the Simulink and Mechanical States of the Stewart Platform" on page 4-21.
- The variable `rt_yout` containing the output signals. These outputs are the three output signals designated by the model's output signal ports (Top Plate Position, Errors, and Leg Forces).

Simulating with Hardware in the Loop

In this section...

About Dedicated Hardware Targets for Stewart Platform Simulation

---

Note This study requires experience with code generation and dedicated hardware deployment. To complete it, you need to have installed the following products, besides MATLAB, Simulink, and the SimMechanics product:

- Real-Time Workshop
- xPC Target

Working first through "Generating and Simulating with Code" on page 4-71, is strongly recommended.

A common step after generating and compiling code from a model is to download the compiled executable to a computer dedicated to running just that application. For a model with a control system, you can download the complete model as a unit or separate the controller and plant into different executables on different computers. You can also execute the controller part as embedded code on a dedicated computer that controls an actual plant. Such application deployments are known as *hardware in the loop* or *rapid prototyping* [9].

xPC Target software and Real-Time Workshop allow you to generate and compile code from a SimMechanics model and download it to a computer with an IBM PC-type processor. xPC Target software acts as another target within Real-Time Workshop and requires a fixed-step solver. You can use xPC Target software to implement controller-plant models in many configurations [10].

This case study outlines some model conversion-downloading applications based on the Stewart platform modeled with SimMechanics blocks.

For More Information About xPC Target Software

Consult the xPC Target documentation for full instructions on downloading and running executable code in different configurations.

Files Needed for This Study

This study requires mech_stewart_xpc, as well as the initialization M-files.

Adjusting Hardware for Computational Demands

Simulation with a fixed simulation time is subject to the basic tradeoff between accuracy and speed. (See “Improving Performance” on page 2-32.) You can make a simulation more accurate by reducing its step size, but at the expense of creating more time steps and slowing down the real clock time. You can speed up the simulation by increasing the time step size, but you risk losing enough accuracy that the simulation fails to converge.

Real-Time Simulation Tradeoff

A typical requirement for code running on dedicated processors is that the simulation run in real time. That is, the compiled code should run with

- A finite number of steps (requiring fixed-step solvers)
- Execution time no longer than the physical time being simulated

These requirements are particularly critical for controller code.

With SimMechanics models, the accuracy-speed tradeoff is acute. SimMechanics simulation is computationally intensive and become even more so the more closed loops and constraints you add.

- With dedicated processor execution, reducing the step size ultimately leads to processor overload. The processor needs more clock time to execute a step than the solver time step allows.
- In SimMechanics simulations, convergence failure resulting from too large a time step typically appears as a failure of your simulation to respect constraint tolerances, assembly tolerances, or both.

Simple SimMechanics models require central processor speeds in the mid-hundreds of megahertz (MHz) range. More complex models such as the Stewart platform (with 36 degrees of freedom, as well as 5 independent closed loops and 40 constraints arising from cutting those loops) demand more processor speed, starting in the low gigahertz (GHz) range.

Mitigating the Real-Time Simulation Tradeoff

You have two ways to alleviate the conflict between accuracy and speed in real-time simulation.

- Increase the processor speed. This allows you to reduce the solver step size while keeping the clock time unchanged.
- Break up a complete model into parts, each simulated by its own model downloaded to and executed on a different processor.

Both approaches are complicated by additional factors, such as memory caching and bus speed. Real-time simulation distinguishes between the sample time in signal buses and the solver step size.

Caution Sample time must be a positive integral multiple of solver step size. For SimMechanics models, avoid making sample time larger than step size to prevent simulation convergence failures.

Downloading a Complete Model to the Target

As a trial of running the Stewart platform simulation on dedicated hardware, here you convert a model to code, then download it and run it on an external PC-type computer. The model requires a processor of speed approximately 2 GHz or faster, and a separate target computer monitor.

Consult the xPC Target documentation for details on preparing the target computer, establishing the host-target connection, and interacting with the target from the host.

Setting Up the Target Computer and Host-Target Connection

The results here were obtained with host and target PC-type computers, each with a 3 GHz Pentium 4 processor and 1 gigabyte of RAM, communicating with each other by an RS-232 connection.

To set up the connection and start the target, you need an RS-232 cable and a blank, formatted floppy disk. The target requires a floppy disk drive. You can observe target simulation on a target monitor, your host monitor, or both.

1. Connect the host and target computer to one another with their respective RS-232 ports and a cable.
2. From MATLAB, prepare an xPC Target boot floppy disk.
3. Insert the prepared xPC Target boot disk into the target PC floppy drive. Start the target computer.
4. After the target has finished booting, confirm the host-target connection.

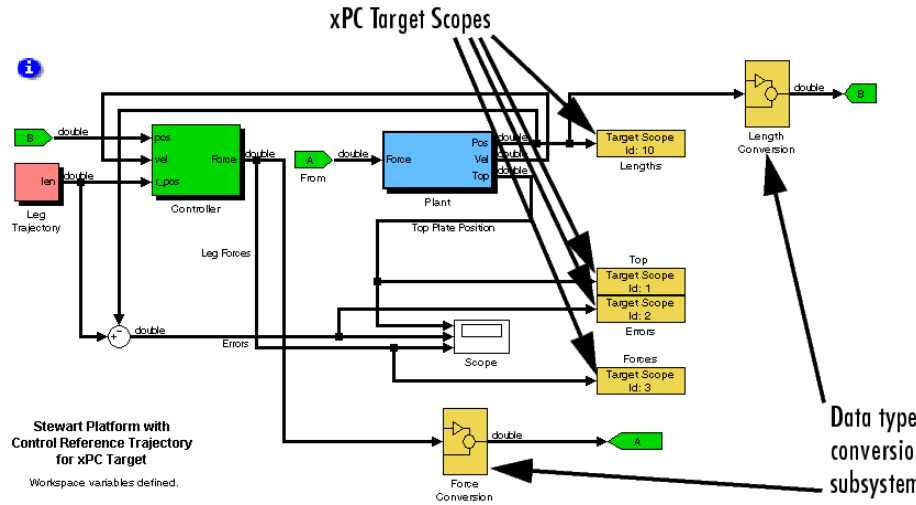
Examining and Running the xPC Model — Data Type Conversion

For this example, you use a variant of the code generation model presented in the preceding study, “Generating and Simulating with Code” on page 4-71.

- The model contains xPC Scope blocks for observing the simulation results later. The **Scope type** for each is `Target`. Thus they will appear on the target PC after you download the compiled code.
- The controller and plant work with the default Simulink 64-bit floating double data type. To test the effect of the type conversion needed for passing signals on a hardware bus, the model also contains subsystems that convert these floating doubles to fixed-point integers, then back to doubles.

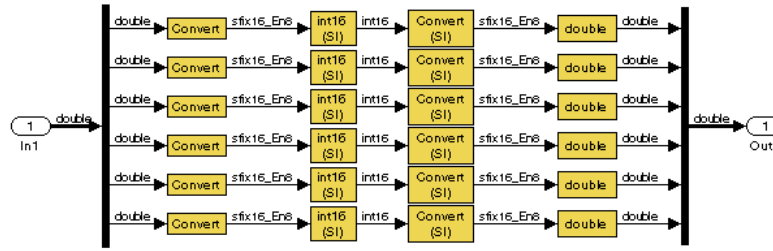
  The data conversion truncates the controller-plant data and changes the simulation behavior somewhat. It is critical to test the impact of such changes before deploying the code to hardware.

1 Open this model, mech_stewart_xpc. Update the diagram (**Ctrl+D**). The vector signals now appear as wide lines and display their data types.



Stewart Platform with Control Reference Trajectory for xPC Target™ Simulation

2 Open the Force Conversion and Length Conversion subsystems. Each subsystem converts a vector signal from floating doubles to 16-bit integers (typical of hardware buses) and back to doubles. These subsystems mimic the effect of hardware buses communicating between controller and plant.



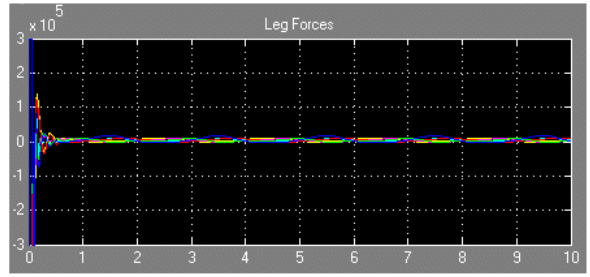
Before the data are converted to integer format, they must be converted from floating to fixed point, truncating the floating double signals. The Data Type Conversion blocks that change doubles to fixed points apply

scaling to ensure that information lost to truncation is “small,” as defined by the force and leg length numbers typical of this simulation. These scalings are set in the Data Type Conversion block dialogs.

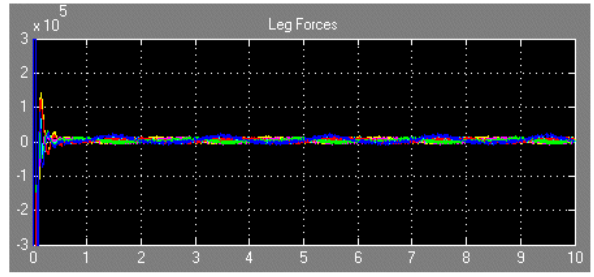
3 Close the Conversion subsystems. Open the Scope.

4 Run the model and observe the motion. Afterward, close the Scope.

The difference between this Stewart platform simulation and earlier ones is clear in the Leg Forces scope trace, which exhibits a small level of “noise” after the initial transient has passed. This “noise” is due to data truncation when the floating doubles are converted to fixed point.



Leg forces without fixed-point truncation



Leg forces with fixed-point truncation

Generating and Downloading Code from the xPC Model

In the next steps, you convert the model to code and download it to the target.

1 Confirm the solver step size (dt1) and sample time (dt2) by entering

```
dt1, dt2
```

at the command line. The values are 5.0 milliseconds (ms).

2 Check the code generation target selection in Configuration Parameters, under the **Real-Time Workshop** node, **Target selection > System target file**. The target selection is `xpctarget.tlc`.

   Under the **Real-Time Workshop** node, check the **xPC Target options** entry. Leave these default settings.

3 On the **Real-Time Workshop** tab, click **Build** to start code generation.

   Follow the progress on the command window, as Real-Time Workshop generates and compiles the model, then downloads it to your target computer. When the download is complete, you see the four empty xPC Target scope windows on the target monitor.

Running the xPC Stewart Platform Model on the Target

The xPC Target interface creates an object called `tg` that allows you to control the application on the target machine.

1 Using the xPC Target interface, start the target application.

   The target computer monitor displays the execution. In the command window, the xPC Target interface summarizes the execution results.

2 Stop the target application. The command window displays the execution summary. The target scopes display the simulation results.

Viewing the Target Simulation with xPC Scopes

xPC Target software allows you to observe simulation in various ways. The xPC Target documentation explains the details.

- In the first run, you observed target-type xPC scopes on the target monitor.
- You can change the **Scope type** of one or more xPC scopes to `Host` and observe them on your host computer instead.
- The xPC Target interface also allows you to connect and display such scopes while the simulation is running. You can make connecting and displaying scopes during simulation easier by changing the stop time to infinity (`inf`).

Adjusting the Step and Sample Times — Testing for CPU Overload

You can make your simulation more accurate by reducing the solver step size. But by requiring more steps, you also make the simulation more intensive. If the solver step size drops below the task execution time (TET), the target processor cannot keep up with the simulation and suffers CPU overload.

The xPC Target summary in the command window indicates if CPU overload has occurred when you start or stop target object (`tg`) execution.

Test for CPU overload by reducing `dt1` and `dt2`.

1. Enter

   ```
   dt1 = 0.0025; dt2 = 0.0025;
   ```

2. Build and download the generated code again.

3. Start the target application.

You can understand how close to, or how far into, CPU overload your model is by comparing the TET with the solver/sample time.

- If the TET value is smaller than the sample/solver time, the target processor is able to keep up with the solver.
- If the TET value is larger than the sample/solver time, the target processor cannot keep up with the solver. CPU overload halts target execution.

You can keep reducing the solver/sample time until you cause CPU overload. This point is the limit of your target processor with this model. You can work around CPU overload by

- Using a faster processor. The ratio of TET to sample time indicates roughly how much faster the processor needs to be.
- Increasing the solver/sample time. Be sure not to increase it too much, to avoid simulation convergence failures.

See “Adjusting Hardware for Computational Demands” on page 4-82.

Configuring for Realistic Hardware

Typical goals of downloading compiled code to a dedicated computer are

- Simulating controller and plant in real time
- Embedding a discretized version of the controller code on a dedicated computer that controls an actual plant

Separating Controller and Plant — Bus Communication — Discretization

Controller and plant communicate through a hardware bus configured with a specific data protocol. The xPC Target block library contains communication blocks based on a variety of data protocols matching common hardware buses. In realistic applications, the controller is often already discretized (simulated with discrete states) and requires no conversion from floating point.

The plant simulation remains continuous (not discrete) to better imitate the actual physical system.

Caution You cannot use discrete states with SimMechanics blocks in your model. Discretizing a controller requires separating controller and plant into different models.

Hardware Configuration Possibilities

Choose a model and hardware configuration depending on your needs.

- Separate controller and plant into different subsystems that communicate through a physical bus interfaced with xPC Target bus blocks, rather than Simulink signal lines. To run such a model on a target requires the target to have the corresponding hardware card and bus cable.
- Separate controller and plant into two different models that also communicate through a physical bus interfaced with xPC Target bus blocks. You then download the two models to two separate targets that communicate through a bus cable connected to the corresponding hardware cards.

Once you separate controller and plant into different models, you can discretize the controller.

- Embed the controller on a dedicated target that controls an actual Stewart platform. The target and platform communicate through a bus or other I/O hardware corresponding to the blocks used in the controller model.

Mitigating Real-Time Trade-offs

Real-time simulations are restricted by the tradeoff between accuracy and speed and limited by target execution time and maintaining convergence. You need to ensure that your memory caching and bus, not just your processor(s), are fast enough to cope with SimMechanics computational demands. See "Adjusting Hardware for Computational Demands" on page 4-82.

Index

T

V

W