SD/FAST User's Manual

Symbolic Dynamics, Inc.

Version B.2

SD/FAST User's Manual
Version B.2
September 1994 (This printing: 9/96)

AUTHORS
Michael G. Hollars, Ph.D., P.E.
Dan E. Rosenthal, Ph.D. (dr@symdyn.com)
Michael A. Sherman (sherm@symdyn.com)

IMPORTANT NOTICE

The information in this manual is subject to change without notice and should not be construed as a commitment by Symbolic Dynamics, Inc.

TRADEMARKS

SD/FAST is a trademark of Symbolic Dynamics, Inc. and Parametric Technology Corporation.

VAX, VMS, and DEC are trademarks of Digital Equipment Corporation.

UNIX is a trademark of Unix Systems Laboratories.

Sun and Sun Workstations are trademarks of Sun Microsystems, Inc.

ACSL is a trademark of Mitchell and Gauthier Associates.

IMSL is a trademark of IMSL, Inc.

Matrix-X and System Build are trademarks of Integrated Systems, Inc.

Control-C is a trademark of Systems Control, Inc.

Easy 5 is a trademark of Boeing Computer Services.

Pro-Matlab and Simulink are trademarks of The Mathworks, Inc.

LICENSING INFORMATION

For information on purchasing SD/FAST or additional copies of this manual, contact:

U.S.A., Canada and Worldwide:

Symbolic Dynamics, Inc.
561 Bush Street
Mountain View, CA, U.S.A. 94041
telephone: +1 (415) 960-1532
fax: +1 (415) 960-0338
email: info@symdyn.com
web: http://www.symdyn.com
anonymous ftp: ftp.symdyn.com

Europe:

Rapid Data LTD.
Amelia House, Crescent Road
Worthing, West Sussex
England, U.K. BN11 1RL
telephone: +44 (1903) 821266
fax: +44 (1903) 820762
email: ali@radata.demon.co.uk

TECHNICAL SUPPORT

U.S.A., Canada and Worldwide:

Symbolic Dynamics, Inc.

telephone *(U.S.A.)*:	+1 (415) 960-1532
fax:	+1 (415) 960-0338
email:	support@symdyn.com

Europe:

Rapid Data LTD.

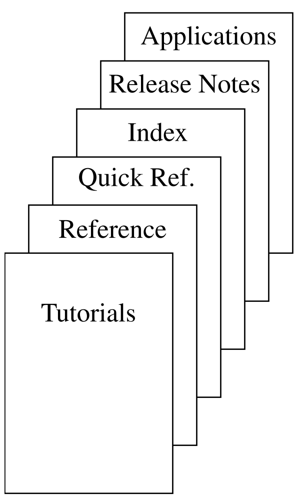
telephone *(U.K.)*:	+44 (1903) 821266
fax:	+44 (1903) 820762
email:	ali@radata.demon.co.uk

Table of Contents

Preface

This manual describes SD/FAST, a software product used in the development of high-performance analyses and design studies of mechanical systems. This preface includes sections on using this manual to your maximum benefit, and typographical conventions for understanding the examples in this manual.

P1 How To Use This Manual



The *intended user* of this manual is an analyst with a good understanding of mechanics, analysis and simulation techniques.

There are six sources of documentation for SD/FAST:

- A Set of Tutorials
- A Reference Section
- A Quick Reference Guide
- Extensive Index
- Release Notes
- Application Notes

The Tutorials, Reference Section, Quick Reference Guide, and Index are provided in this manual. Installation instructions and manual revisions are distributed as Release Notes on an as-needed basis. Application Notes on theory, performance, and specialized uses of SD/FAST are available upon request from your SD/FAST representative.

If you are a *prospective user* or *new user*, first read sequentially through the *Tutorials*. Each Tutorial explains new material and builds upon information presented in the previous Tutorials. Complete examples are presented to help you get started. All example code presented in the Tutorials is provided on the SD/FAST distribution media.

Experienced users will find complete documentation on all features of SD/FAST in the *Reference Section*. Specific Tutorials are also helpful when first setting up a problem of the type discussed in the Tutorial. An extensive *Index* with cross-references to both the Tutorials and the Reference Section is provided at the end of this manual.

The *Quick Reference Guide*, which follows the Reference Section, is a short summary of keywords, SD/FAST subroutine usage, and user guidelines. The Quick Reference Guide can be photocopied and kept near your computer for handy reference.

If the SD/FAST software does not appear to be functioning as described, first check the *Release Notes* for updated information. If no revisions apply, often just looking over the examples in the Tutorials or available *Application Notes* on problems related to your application may clear up your questions. Otherwise, help is available from Symbolic Dynamics, both in the form of brief telephone conversations and extensive, on-site consultation for more difficult problems. Telephone, FAX, and email numbers are given at the beginning of this manual.

P2 Conventions

Typographical conventions used in this manual are:

- A `typewriter` or `bitmap` font is used to represent literal text to appear in computer input or output.
- In conjunction with the `typewriter` font, *italics* in angle brackets is used to indicate a class of expected response, as in `body` = <*bodyname*>.
- *Italics* are also used for emphasizing crucial ideas and words with specific or precise definitions which may be different from common usage.

Each major division of the manual has its own prefix symbol for page numbers:

- T for Tutorials
- R for Reference
- Q for Quick Reference Guide
- I for Index
- Add-ons to the manual such as the Release Notes and Application Notes are numbered conventionally.

For symbols and equations:

- Boldface symbols such as **A** and **b** represent matrices and vectors. Italic symbols such as x and y represent scalars.
- Equations are numbered and referenced by the section they occur in such as:

$$\mathbf{A}\dot{\mathbf{u}} = \mathbf{b} \qquad \text{Eqn. T1.1}$$

- A “hat” on a vector, such as $\hat{\mathbf{n}}$, denotes a unit vector.

TUTORIAL 1 Introducing SD/FAST

Objectives

- Learn how to use the Tutorials effectively.
- Understand what can be done with SD/FAST.
- Run a very simple example from start to finish.
- Identify the prerequisite skills needed for successful use of SD/FAST.
- Examine the features of SD/FAST.
- Summarize the procedure for using SD/FAST.

T1.1 How To Use The Tutorials

First, if you have not already done so, please read the Preface to learn how the entire manual is organized and what conventions are used. Make sure that your copy of SD/FAST is installed and operational on your computer, and that all the *example files* distributed with SD/FAST are installed. Your system administrator can help you locate these files. The Tutorials make extensive use of the examples and hands-on practice is the quickest way to master SD/FAST.

Each Tutorial begins with a set of *objectives* which are reviewed in a *summary* at the end. Each objective is a numbered section within the Tutorial which makes it easy to find later by using the Table of Contents. Because each Tutorial relies on information presented in previous Tutorials, it is essential for first-time users to progress through each objective in each Tutorial *sequentially.* Once you have completed the Tutorials, use the Reference Section, specific examples in the Tutorials, the Quick Reference Guide, and the Index to find the information you need to solve a particular problem.

The remainder of this Tutorial consists of an introduction to the kinds of problems that can be solved with SD/FAST, a simple but complete example to familiarize you with the use of SD/FAST, information about the skills required for successful use of SD/FAST, a discussion of the available features, and a discussion of how SD/FAST fits into an overall analysis environment.

T1.2 Applications of SD/FAST

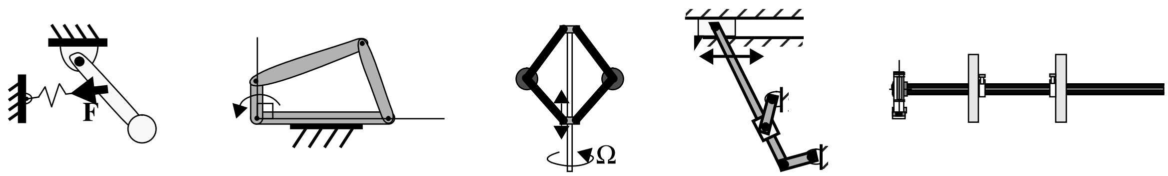
SD/FAST can be used to perform analysis and design studies on any mechanical system which can be modeled as a set of rigid bodies interconnected by *joints*, influenced by *forces*, driven by *prescribed motions*, and restricted by *constraints*. A modeled system can be *free-flying* or *grounded*. Within this broad framework, virtually any mechanical system can be modeled and analyzed.

In this section, we will take a brief look at some of the applications to which SD/FAST can be applied. At this point we will not attempt to explain what features of SD/FAST are used to model and analyze the systems shown. Most of the analyses depicted below can be performed with just SD/FAST and a small amount of user-written Fortran or C code. Some require more coding, or the use of an external analysis tool like ACSL, Matrix-X, Easy 5, Pro-Matlab, Simulink, or any other package designed to work with user-supplied subroutines (which, in this case are SD/FAST-supplied instead).

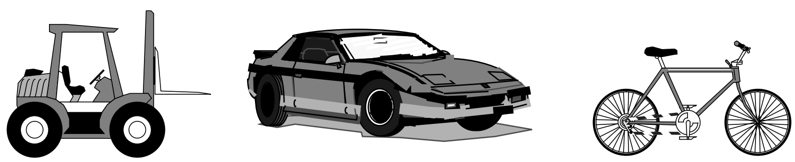
First we will look in general at the kinds of mechanical systems which can be modeled. Then we will look more closely at force, prescribed motion, and constraint models. Finally, we will list some of the types of analyses and design studies which can be performed on these models.

T1.2.1 Mechanical systems

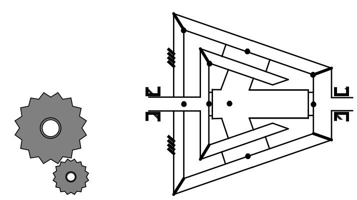
Here are some examples of mechanical systems which can be modeled using SD/FAST:



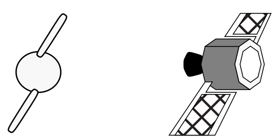
Mechanisms and Machines



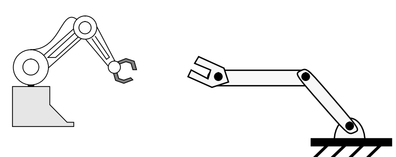
Ground vehicles



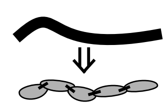
Gear trains



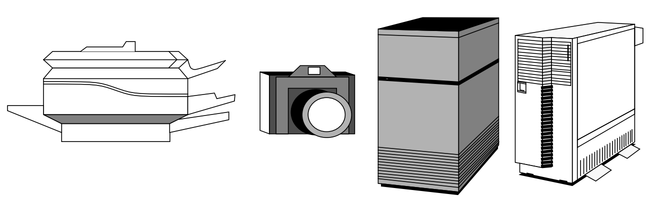
Articulated spacecraft



Manipulators



Cables

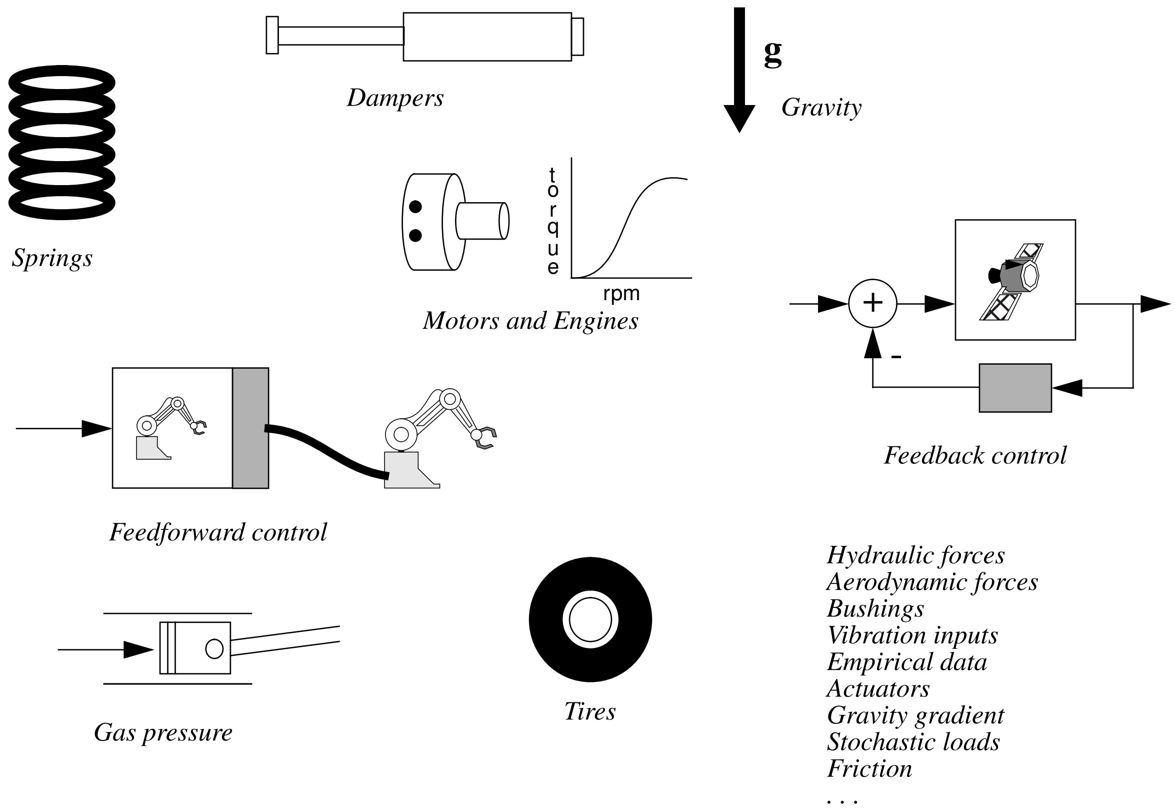


High speed electromechanical devices

Suspension components
Multiple-arm robots
Anti-lock brakes
Helicopters
Walking/hopping machines
Farm machinery
Construction equipment
Precision pointing mechanisms
Momentum wheels
Biomechanics
. . .

T1.2.2 Force Models

Forces acting on the system can come from any source, some examples of which are pictured below.



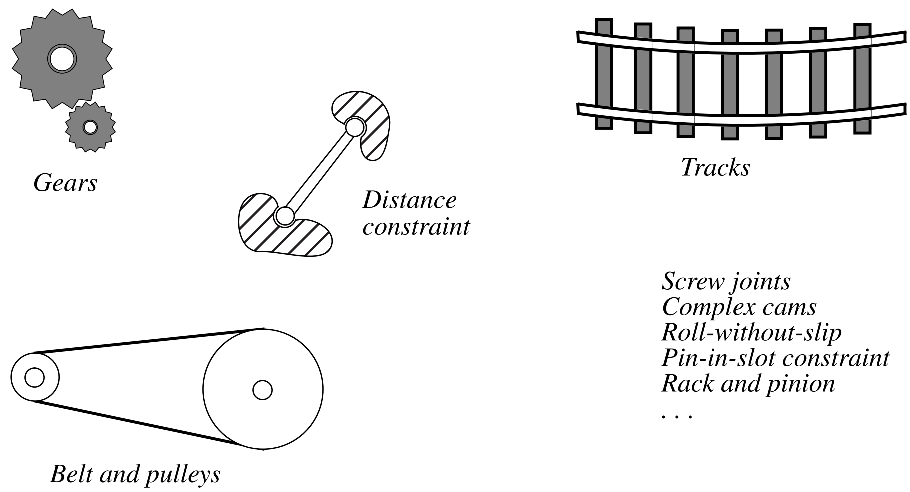
T1.2.3 Prescribed Motions

Prescribed motions used to drive joints in the system can be arbitrary functions of time and system state. The forces required to implement the desired motions are calculated and returned by SD/FAST. Some applications of prescribed motion are:

Locked joints
Constant-speed motors
Road profiles
Orbital motion
Sinusoidal motion
Cam follower
. . .

T1.2.4 Constraints

Constraints used to restrict allowable motion can be arbitrary functions of time and system state. The reaction loads required to implement the constraints are calculated and returned by SD/FAST. Some applications of constraints are:



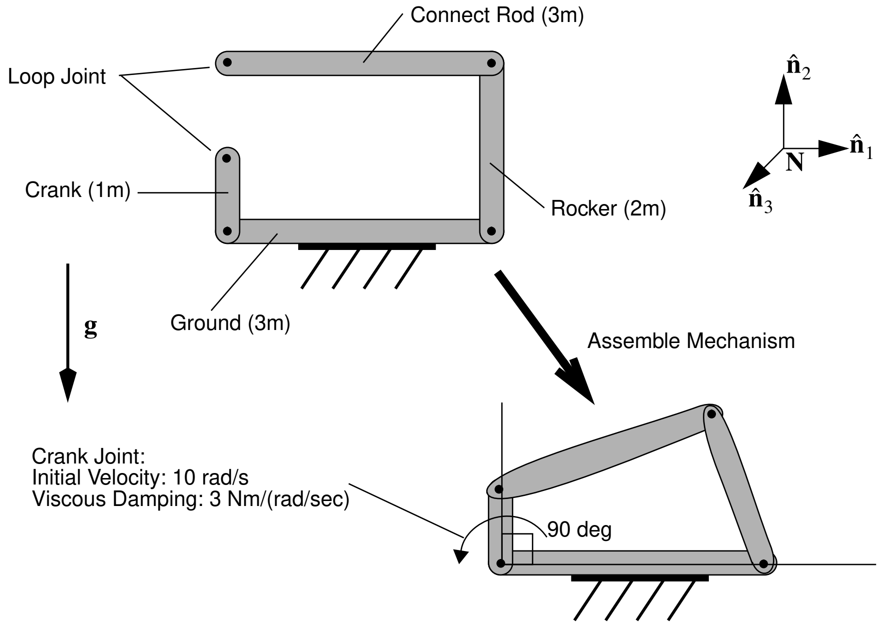
Screw joints
Complex cams
Roll-without-slip
Pin-in-slot constraint
Rack and pinion
. . .

T1.2.5 Types of Analyses and Design Studies

SD/FAST can be used to perform a wide variety of analyses and design studies. Some examples are:

Assembly analysis
Velocity (initial conditions) analysis
Forward and inverse dynamics
Statics and steady motion
Limit cycle identification
Mechanical advantage
Power flow
Design sensitivity
Design optimization (with external optimizer)
Real-time hardware-in-the-loop
. . .

Figure T1-1 Four-Bar Mechanism Model



T1.3 A Simple Example

This section will take you through a very simple example from start to finish. Do not worry about how the simulation was written, details will be covered in later Tutorials. The purpose of this example is to demonstrate the ease of using SD/FAST.

The example we will demonstrate is assembly and transient motion of a common four-bar mechanism shown in Figure T1-1, which consists of ground, a crank, a rocker, and a connect bar. The model parameters are shown in the figure and in the SD/FAST input System Description File. The system operates under gravity and has 3 Nm/(rad/sec) viscous damping in the joint connecting the crank to ground.

We first want to have SD/FAST assemble the mechanism since we don't want to compute the geometry of the assembled system. The system is assembled with the crank in its initial vertical position—only the rocker and connect bar are allowed to move during assembly. Next, compatible initial velocities are computed for all the bodies with the crank set at a desired initial angular rate of 10 rad/sec. Finally, we integrate the equations of motion, allowing the system to start with the crank velocity of 10 rad/sec, then slow down and drop due to the viscous damping and gravity. Reaction forces at the crank-ground pin are also computed.

The user-written System Description File used by SD/FAST to generate the equations of motion and all other computational routines is in `fbar.sd` on your distribution media and is reproduced in Figure T1-1. After installation of the SD/FAST software, you can run this file to generate the simulation code by simply typing "`sdfast fbar.sd`". Note that the file is very succinct, easy to read, and uses friendly user-selected names for the bodies.

Figure T1-1 Complete SD/FAST System Description File for Four-Bar Mechanism (`fbar.sd`)

Gravity

```
gravity = 0 -9.8 0
```

Description for each body

```
body = crank inb = $ground joint = pin
    mass = 1                        inertia = 1 0 1
    bodytojoint = 0 -.5 0           inbtojoint = 0 0 0
    pin = 0 0 1

body = rocker inb = $ground joint = pin
    mass = 2                        inertia = 2 0 2
    bodytojoint = 0 -1 0            inbtojoint = 3 0 0
    pin = 0 0 1

body = connect inb = rocker joint = pin
    mass = 3                        inertia = 0 3 3
    bodytojoint = 1.5 0 0           inbtojoint = 0 1 0
    pin = 0 0 1
```

A comment line

```
# This is the loop joint.
```

Loop joint description

```
body = connect inb = crank joint = pin
    bodytojoint = -1.5 0 0          inbtojoint = 0 .5 0
    pin = 0 0 1
```

All of the basic geometry and mass properties, and gravity, are passed to SD/FAST in the System Description file. Any other features of the model that depend upon the state of the system (such as viscous damping force being a function of the velocity state) are passed at run time to the SD/FAST routines via user-written routines linked with the simulation code.

Figure T1-2 Complete Driver Simulation for Four-Bar Mechanism (`fbar.f`)

SD/FAST provides dimensions

```
      real*8 state(6),dstate(6),t,dt,ctol,tol,frc(4,3),trq(4,3)
      integer lock(3),fcnt,err,i,flag
      data dt,ctol,tol/0.05d0,1d-5,1d-6/
      data lock,flag/1,0,0,1/

c Set initial conditions to a starting guess and initialize sd model.
      data t,state/0d0,0d0,0d0,0d0,10d0,1d0,-1d0/
```

Initialize SD/FAST model

```
      call sdinit

c Do assembly and velocity analysis.  Only q(1) and u(1) are locked.
```

Assemble mechanism

```
      call sdassemble(t,state,lock,1d-7,1000,fcnt,err)
```

Compute compatible velocities

```
      call sdinitvel(t,state,lock,1d-7,1000,fcnt,err)
      print 5, state
    5 format(' ',6f12.5)

c Do motion analysis. Constraints maintained to `ctol'; integration to `tol'.
      do 10 i=1,200
```

Integrate E.O.M.

```
         call sdmotion(t,state,dstate,dt,ctol,tol,flag,err)
```

Compute reaction loads

```
         call sdreac(frc,trq)
         print 5, t,state(1),state(4),frc(1,1),frc(1,2)
   10 continue
```

Error checking

```
      call sdprinterr(6)
      end

c Apply viscous damping torque to the crank joint.
      subroutine sduforce(t,q,u)
      double precision t,q(3),u(3)
```

Viscous damping model

```
      call sdhinget(1,1,-3d0*u(1))
      return
      end
```

The main driver program is usually written by the user along with routines that describe time- and state-dependent model features, such as applied loads (forces and/or torques) and prescribed motion. The entire program used to drive the crank model is in `fbar.f` and is reproduced in Figure T1-2. This includes a main program (in Fortran) and a routine that specifies the viscous damping for the crank joint.

After compiling, linking, and running the simulation, we get the following state vector just after the assembly and initial velocity analyses are done:

```
0.00000     0.08463     0.25268     10.00000     4.87298     -5.16398
```

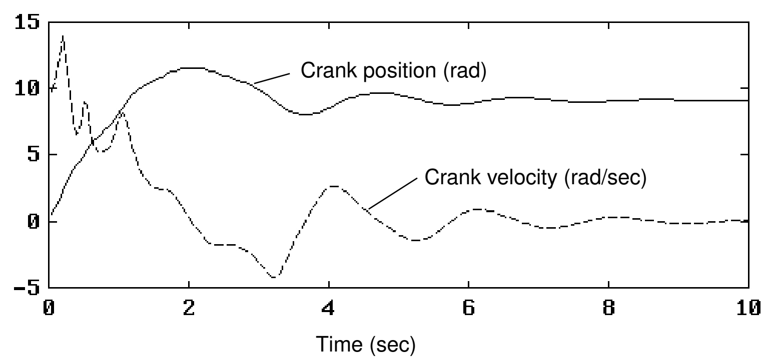
Zero (rad) initial crank angle (as desired)

Rocker and connect rod initial angles required to assemble mechanism

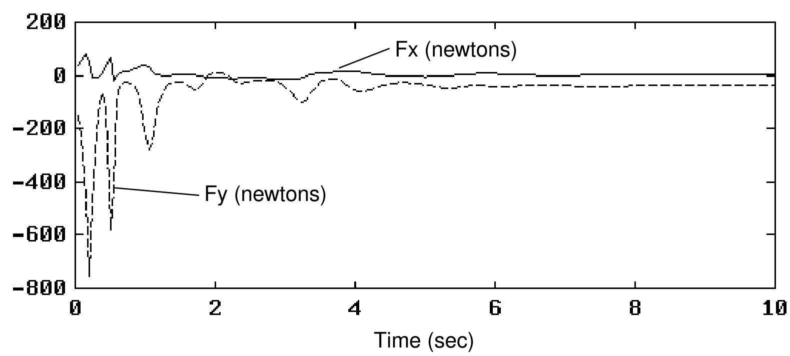
Initial crank velocity of 10 rad/sec (as desired)

Rocker and connect rod angular velocities required to prevent mechanism from “flying apart”

Rather than list the time history of the subsequent motion analysis, the crank position and velocity, and the reaction forces in the $\hat{\mathbf{n}}_1$ and $\hat{\mathbf{n}}_2$ directions (Fx and Fy) acting on the crank were plotted using one of many commercially available plotting packages:



Note that the crank had sufficient kinetic energy to swing around once through 2π radians before the viscous damping and gravity trapped the mechanism into oscillatory motion that rapidly damped out. The reaction forces were also high, particularly in the vertical direction, until the motion significantly damped.



T1.4 Skills Needed to Use SD/FAST

SD/FAST is a tool for the engineering analyst or sophisticated designer. It is designed to provide these specialists with maximal expressive power for analysis and design studies of even the most advanced and complex designs, while retaining a naturalness and familiarity that makes it useful for even the simplest of problems.

To achieve these goals, SD/FAST builds on certain knowledge and skills that its users are expected to possess. These requirements are listed below.

Required skills

These are the minimum set of skills necessary for successful use of SD/FAST.

1. Understanding of the basic concepts in mechanics.

   These include: rigid bodies and their mass properties; forces and torques; position, velocity, and acceleration; orientation, angular velocity, and angular acceleration.

2. Understanding of appropriate mathematical concepts.

   User should be comfortable with: simple vector arithmetic; the concepts of a derivative and integral; functions; linear vs. nonlinear; basic linear algebra.

3. Ability to write a modest program in a general purpose language like Fortran or C, or in a simulation language like ACSL, Easy 5, Simulink or Matrix-X.

   Necessary skills include: ability to use a text editor; declaring and indexing arrays; assigning values to variables; calling Fortran or C subroutines and functions; writing DO loops; writing simple subroutines; and generating numerical output.

4. Ability to produce plots from numerical data.

   The results of most analyses and design studies are more readily interpreted when presented in plots rather than as tables of numerical data. If you use an external analysis package like ACSL, Matrix-X, Easy 5, Control-C or Pro-Matlab you will need to know how to use the plotting facilities of these tools. If you work directly in Fortran or C, you will need access to some plotting package and the ability to use it to produce a plot from numerical output.

Useful skills

Users with the following skills will find they can put them to good use in conjunction with the advanced features of SD/FAST.

1. Advanced modeling of force elements and actuators.

   Users with large investments in pre-existing software models of complex nonlinear elements such as tires, control systems, rockets and thrusters, combustion forces, aerodynamic and hydraulic effects, gravity models, road profiles or empirical data can generally apply these models directly in analyses using SD/FAST with little or no re-coding.

2. Design optimization.

   SD/FAST has a complete and natural facility for incorporating design variables into models being analyzed. Figures of merit (objective functions) can be constructed for any aspect of interest. Design variables can be driven programmatically in sensitivity studies. For the user with access to a numerical optimizer such as those in OPT-DES, Pro-Matlab, or Matrix-X, the design variables can be driven automatically to improve the design.

3. Control system design and evaluation.

   In addition to the use of SD/FAST to perform nonlinear simulations to test linear control systems, an external linearization capability can be used with SD/FAST in the design of these control systems. Such capabilities are provided in packages like ACSL, Matrix-X, and Pro-Matlab.

T1.5 The Features of SD/FAST

This section outlines the various features of SD/FAST, including a more detailed discussion of the classes of physical systems which can be modeled, user-interface features, analysis, and design features.

T1.5.1 System Models Accepted by SD/FAST

Put succinctly, SD/FAST is a very powerful software tool that can be used to model any mechanical system which can be described as a set of *joint-connected rigid bodies plus constraint conditions*. The following sections describe acceptable system geometries, the types of motion constraints that can be imposed upon the system, the types of forces and torques that can be applied internally and externally to the system, and the types of analyses and design studies that can be performed with SD/FAST.

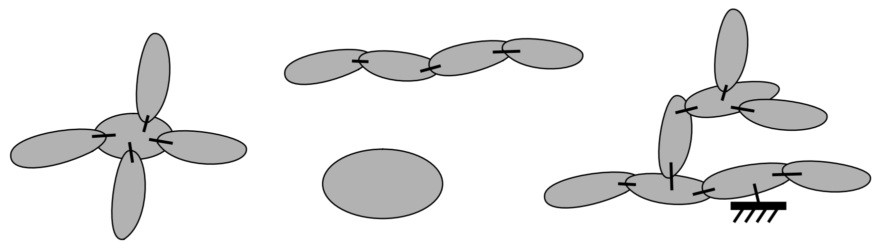
System Geometry

Any set of joint-connected rigid bodies can be considered topologically equivalent to a tree (chains of bodies with branches) which may have some branches connected to form loops. Various rotational and sliding joints connect the bodies.

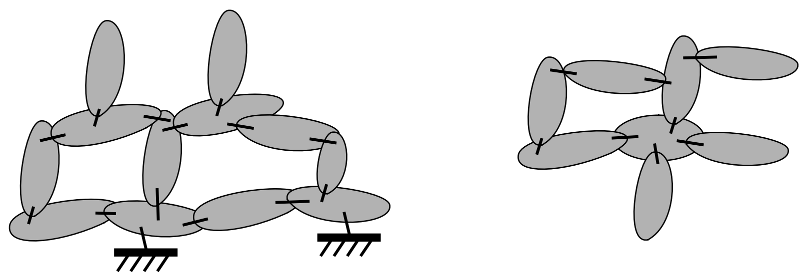
- **System Topologies:** The system topology can be open-loop (trees) or closed-loop. The systems can be *free-flying*, (e.g., a satellite) or *grounded* (e.g., a manipulator or vehicle). Multiple ground connections are allowed. *Multiple interacting systems* coupled by forces (such as gravitational interaction between two spacecraft) or by constraints (such as a multiple-arm manipulator) can be modeled. Figure T1-4[1] shows some of the system topologies accepted by SD/FAST.
- **Wide Selection of Joint Types:** SD/FAST allows specification of *joints* based upon assemblies of three joint primitives: one-dimensional rotational joints (*pin joint*), one-dimensional translational joints (*slider joint*), and three-dimensional *ball joints* (no hinge axes). You can assemble your own special joints using these primitives or use pre-defined joint types which include *Ujoints* (two-dimensional rotational joint using hinge axes), *gimbal joints* (three-dimensional rotational joint using

1. In this manual, we represent bodies pictorially as “blobs” (in three dimensions you might think of them as “potatoes”) rather than as regular figures such as lines or rectangles. We have found that this technique prevents one’s intuition from leaping to unwarranted generalizations based on the behavior of simple bodies. It also serves as a reminder that the bodies’ geometry is irrelevant in the model — only the mass properties and joint locations matter.

Figure T1-4 System Topologies Acceptable to SD/FAST



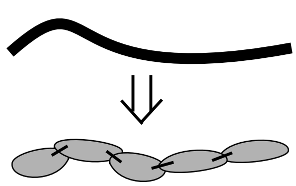
Open-Loop Tree Topologies



Closed-Loop Topologies

hinge axes), *bearing joints* (four-DOF joint with one slider and a gimbal, useful in non-redundant modeling of shaft bearings), two kinds of six degree-of-freedom “joints” (*free joint*, using three sliding joints and a ball joint, and *bushing joint*, using three sliding joints and a gimbal joint), *cylinder joints* (combined pin and slider joint), *planar joints* (two slider joints plus a pin joint), and *weld joints* (a zero DOF “joint” useful for attaching bodies together rigidly). Figure T1-5 shows the SD/FAST-defined joints.

- **Large Systems:** SD/FAST accepts systems with up to 300 rigid bodies and 1000 degrees of freedom. The advanced Kane’s formulation[2] and Order(N) formulation[3] combined with symbolic equation manipulation produces simulations that run much faster than conventional methods, thus allowing such large systems to be reasonably modeled.

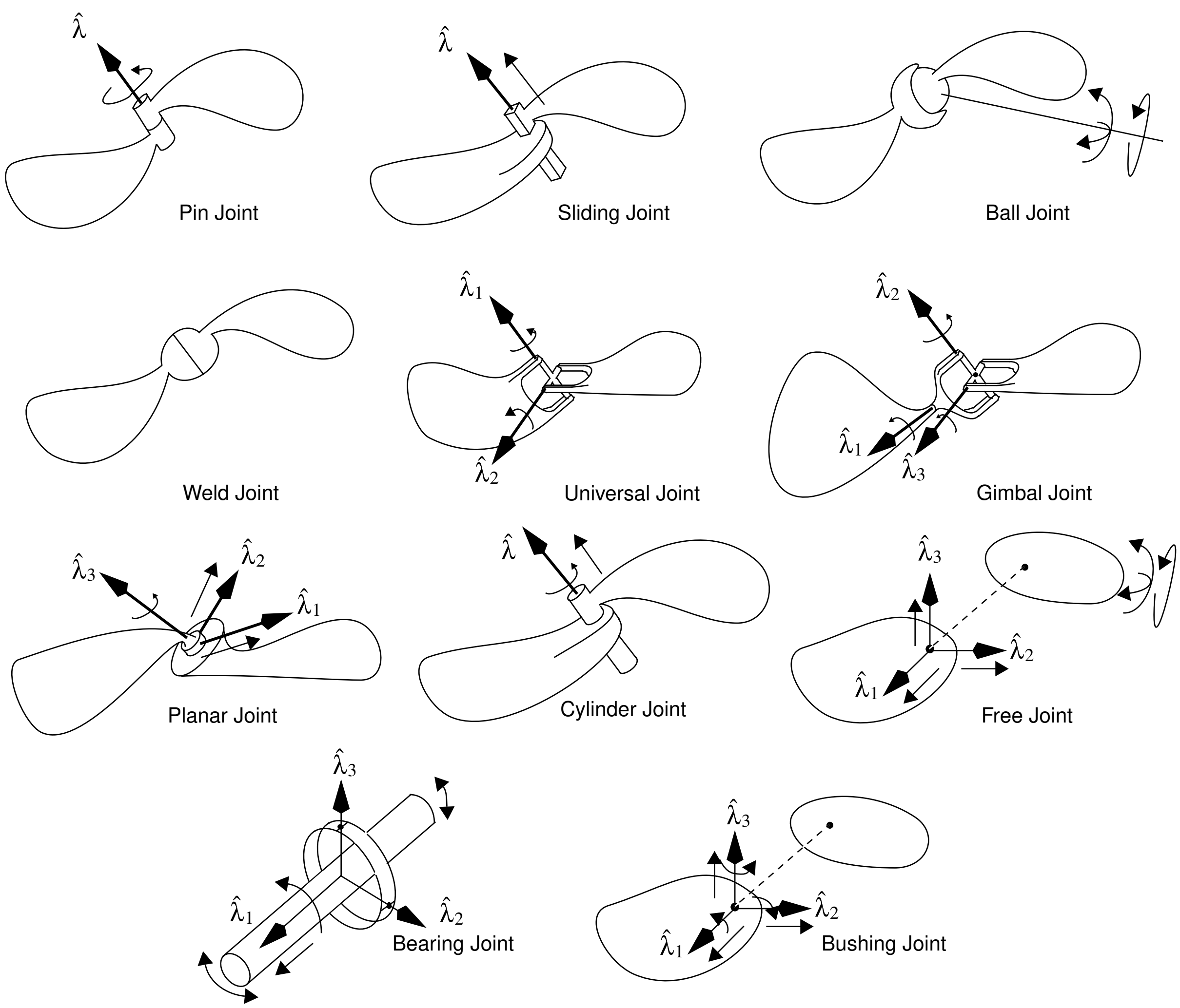


- **Modeling Flexible Bodies:** It is important to point out that SD/FAST *does not* provide explicit models of body flexibility. Instead, a flexible body is represented by a set of rigid bodies connected by joints and subject to internal forces appropriate to the material properties of the flexible body. Also, SD/FAST can be used as a high-performance rigid body “core” of a more comprehensive user-provided simulation

2. Kane, T.R. and Levinson, D.A., *Dynamics: Theory and Application*, McGraw-Hill, New York, 1985.

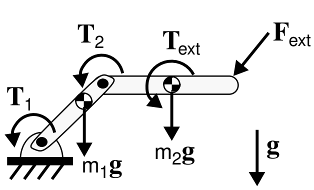
3. Rosenthal, D.E.,“An Order *n* Formulation for Robotic Systems,” *The Journal of the Astronautical Sciences*, Vol. 38, No. 4, OCt-Dec 1990, pp. 511-529.

Figure T1-5 SD/FAST Built-in Joint Types



environment which adds in linear flexibility effects on top of the nonlinear rigid-body results.[4]

Forces and Torques



In general, forces and torques (to which we often refer collectively as *loads*) can be applied to any body and about any joint axis. SD/FAST provides three ways of specifying these loads:

4. See for example, Liu, D., Yocum, J., and Kang, D., “Control and Dynamics of a Flexible Spacecraft During Station-keeping Maneuvers,” *Proceedings of the Fourth NASA Workshop on Computational Control of Flexible Aerospace Systems*, Williamsburg, Virginia, July 1990.

- **Externally Applied:** External forces can be applied to *any location* on *any* body. Likewise, external torques can applied to any body. Thus system components such as thrusters, tires, external friction, and fluid forces can easily be modeled. Interacting forces, such as a spring connecting two arbitrary bodies in the system, are modeled as equal and opposite external forces acting on the bodies.

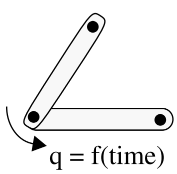
- **Internally Applied:** Internal forces and torques at any *joint axis* can be applied. Thus internal system motors, hydraulic actuators, hinge springs, damping, and other internal forces and torques are easily modeled.

- **Gravity:** A *uniform* gravitational field acting on a system can be specified in the input System Description File. The effects of this field are then included in the generated routines so that they do not need to be accounted for in user-supplied forces at simulation run time. Nonuniform gravitational forces are handled through externally applied forces.

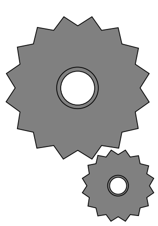
Motion Constraints

Motion constraints are handled in three ways by SD/FAST: as joint constraints, prescribed motion, and user-definable general constraints:

- **Joint Constraints:** A joint enforces certain allowable motions between two bodies, such as relative rotation about joint axes. For joints in the "tree" system, the equations of motion are derived to allow only the motion permitted by the joints, so no explicit constraint equations are necessary. For those joints which make the final connection of tree branches into topological loops (called *loop joints*), explicit constraint equations are generated to enforce the joints' motion restrictions. All constraint equations needed to enforce the joint connections and the loop connectivity are generated *automatically* by SD/FAST. *Redundant constraints* are detected and eliminated automatically.



- **Prescribed Motion:** In addition to enforcing motion between two bodies to occur about certain joint axes (joint constraints), the actual motion about those axes can be *prescribed* as a function of time (and sometimes system state). That is, instead of allowing motion to occur about the joint axis as would be caused by natural dynamic motion, the user can specify that motion (position, velocity and acceleration) such as would occur if an actuator of infinite bandwidth moved the axis in a pre-determined manner. This allows modeling of motors that (for practical purposes) drive parts of machines regardless of the dynamic loads imposed. This also allows *inverse dynamics* computations, in which motion is prescribed and the forces and torques needed to achieve that motion are computed. Prescribed motion can be *turned on and off* during a simulation to model changes such as a solar panel locking into position after being deployed. Again, the constraint equations needed to enforce prescribed motion are generated automatically by SD/FAST.



- **User-Supplied General Constraint Equations:** User-supplied general constraint equations (*user constraints*) can be provided explicitly to implement such motion restrictions as gears, tracks, and wheels rolling without slip. Like prescribed motion, user constraints can also be *turned on and off* during a simulation to model situations such as manipulators grasping objects on the ground or vehicle wheels hopping off the ground.

T1.5.2 Types of Analyses and Design Studies

Many types of analyses can be performed on systems modeled by SD/FAST. Some can be performed simply by calling the basic routines for calculating information about the system. This information includes:

Available Information

?

- the angular and translational locations, velocities, and accelerations of any body, point, or joint axis in the system, expressed in any reference frame. Various conversion tools are available for expressing body orientation several different ways.
- reaction forces at every joint
- joint forces needed to implement prescribed motions
- constraint multipliers (e.g., gear tooth contact forces) for every constraint
- the system angular and linear momentum and kinetic energy
- system center of mass and central inertia matrix

Much more powerful analyses and design studies can be performed by making repeated calls to these routines under changing conditions guided by a numerical integrator, linear or nonlinear root finder, numerical optimizer, or user-written code. There are several methods available for driving SD/FAST-generated routines:

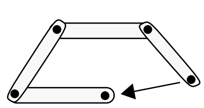
- Use problem specific, SD/FAST-generated *Simplified Analysis Routines*.
- Use generic, SD/FAST-generated *General Analysis Routines*.
- Use an external, commercially available analysis code such as ACSL, Matrix-X, Easy 5, Pro-Matlab, Simulink, or OPTDES.
- Use analysis and optimization tools available in numerical libraries such as IMSL, NAG, and EISPACK.
- Write your own analysis code in a language like Fortran or C.

Features accessible with all these methods are discussed below.

Simplified Analysis Routines

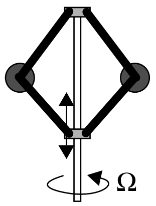
SD/FAST generates *Simplified Analysis Routines* for some of the more common analyses. These routines are generated specifically for the problem at hand. As a consequence, they are very simple to use but are not very general in their capability. They are, however, built on more general routines described below.

Here are the analyses that can be performed with Simplified Analysis Routines:



- **Assembly Analysis:** Systems with topological loops can be assembled *automatically* so that you do not have to worry about computing initial system geometries. The general unconnected tree topology is entered and SD/FAST generates a routine which assembles all the loop joints at run time.
- **Velocity Analysis:** Compatible initial velocities for the assembled system are *automatically* computed using the velocity analysis routine.
- **Kinematic and Dynamic Motion Analysis:** The time history of the system kinematics, dynamics and reaction forces are computed for given constraints, loads, and parameter values. Time is advanced using either a fixed-step or variable-step explic-

it integrator. Constraint errors are monitored, and constraint stabilization is provided using user-selectable feedback rates.



- **Static Analysis:** Computes the system equilibrium position given forces and torques.
- **Steady Motion Analysis:** Computes system steady motion solutions given forces, torques, and some desired non-zero joint velocities, e.g., flyball governors.

General Analysis Routines

For more general analysis needs, the routines underlying the Simplified Analysis Routines may be called directly. These are called *General Analysis Routines*. While these routines are useful in conjunction with other SD/FAST-generated routines, they are fully general and may be used in other contexts as well. There are three basic capabilities provided:

- **Numerical Integration:** Two methods are provided, a basic fixed-step Runge Kutta fourth-order integrator with error estimation, and a variable time-step integrator built on the basic method. Any number of state variables may be integrated. The step size adjustments can be based on error estimates relative to a user-specified tolerance, constraint violations, or any user-detected conditions. This is not a sophisticated integrator, however it is very robust and can handle a wide variety of problems, including those with discontinuities. The integrator can detect lockup and other intractable conditions, and returns with time set just before the disaster would occur. Because this is an explicit integrator, it does not perform well on stiff problems. However, it will produce correct answers even for these problems.
- **Constrained Linear Least-Squares Solver:** A routine is provided which can solve constrained linear least-squares problems. This corresponds to solving sets of linear equations in which some equations *must* be solved (the constraints) while others are to be solved as well as possible (the objectives). Any size problem can be accommodated, subject to computer memory limitations. For underdetermined problems, the solution vector of minimum 2-norm (least squares solution) is found. For overdetermined problems, the solution which minimizes the 2-norm of the residual error in the constraints is found (least squares error). Any remaining degrees of freedom are used to minimize the residual error in the objectives.
- **Constrained Nonlinear Root Finder:** A routine is provided which can solve constrained sets of nonlinear equations. This corresponds to solving a set of nonlinear equations in which some of the equations *must* be solved (the constraints) while others are to be solved as well as possible (the objectives). Problems of this nature arise naturally in many analyses and design studies. For example, the Assembly and Static Simplified Analysis Routines are based on this root finder. As another example, the root finder can be used to perform Inverse Static Analysis, i.e., find values for the parameters of force elements (e.g., spring constants) which will make the current configuration a static equilibrium.

  The user of this routine provides a function for it to call which, when passed the current solution vector, returns the residual error in each of a set of equations. The nonlinear root finder is based on the Newton-Raphson method, but it uses the above-described Constrained Linear Least Squares Solver for its internal linear steps. This makes it extremely robust — a singular Jacobian does not cause difficulties. It is often convenient to use the nonlinear solver to solve linear problems as well, to avoid the need to set up matrices.

External Analysis Codes and Libraries

Using external tools such as those described above in conjunction with SD/FAST-generated models, very powerful analyses are possible. For example:

- **Numerical integration of stiff problems:** many external codes and libraries contain implicit integrators, such as Gear's integrator, which perform well on stiff problems. Other more sophisticated integrators may yield better performance than SD/FAST's built-in integrators. ACSL and Matrix-X, for example, contain implicit integrators.
- **Extensive plotting and visualization:** plotting and animation facilities provided by external codes and libraries can be used to great advantage to display the results of SD/FAST-generated analyses. All the available analysis codes contain plotting facilities of various sophistication levels. In addition many commercial plotting packages and libraries are available on all platforms. Externally-drivable animation packages exist for some of the more powerful graphics workstations.
- **Linearization and control system design:** the linearization and eigenvalue analysis features of many external codes and libraries can make use of SD/FAST-generated models in control system design. ACSL, for example, has a linearization capability.
- **Frequency analysis:** further analysis of the linearized models can yield frequency response information such as Bode plots. Pro-Matlab and Matrix-X, for example, have this capability.
- **Automated Design Optimization:** SD/FAST provides the capability to build parameterized models and construct merit functions to evaluate the quality of candidate designs. External packages containing numerical optimizers (OPTDES, for example) can be used to drive the design parameters to seek optimal values for the merit (objective) functions.

User-written Analyses and Design Studies

Because SD/FAST-generated routines are designed to be driven from general purpose languages such as Fortran and C, all the capabilities of these languages can be brought to bear to provide a great deal of flexibility in analysis and design studies. A few simple examples:

- **Design sensitivity:** design parameters can be varied by small amounts and the derivatives of merit functions can be calculated with respect to any of these variables.
- **Design parameter sweep:** the looping constructs of Fortran or C can be used to iterate one or more design parameters through a predefined range to generate a coarse search of the entire design space.
- **Mechanical advantage, power flow, etc.:** by combining some of the above analyses (such as initial velocity analysis) with simple calculations, a great many interesting properties of mechanical systems can be measured.

A more extensive discussion of the types of analyses possible is given in Reference Section R1.

T1.5.3 System Description File

The SD/FAST software accepts a user-supplied System Description File that describes the system to be modeled. The information in that file is used to generate a problem-specific *source-code* numerical model (in either Fortran or C) to be incorporated into your simulation environment. This section presents the features available for modeling the system in the System Description File. The next section presents the features of the SD/FAST-generated source code.

These are some of the important features of the SD/FAST System Description File:

Natural System Description

The system to be simulated is described to SD/FAST in a simple, natural input language. The system is described in familiar engineering terms in a free format, keyword-oriented specification that is both easy to produce and easy to read.

Free Format Input

The input file can be produced with any convenient text editor and the information specified can be spaced and aligned in any manner which proves convenient and readable for the user.

Two Methods of Parameter Specification

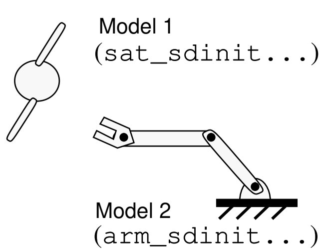
```
# Engine Model

body = crankshaft
  mass = 15
  inertia = 10 0 0
             0 ? 0
             0 0 5?
```

Any system *parameter* can be specified in two ways:

- **Constant Parameter:** System parameters which are known in advance can be provided as constants in the SD/FAST System Description File. These constants are then used to symbolically simplify the equations of motion, so that unneeded calculations are not done during simulation run time.
- **Variable Parameters:** Optionally, system parameters can be left unspecified in the input description (using a "?" instead of a number), with actual values to be provided later at run time. A default numerical value can be provided before the "?". This allows equations of motion to be produced with varying degrees of generality, allowing a trade-off of flexibility versus performance. With variable parameters, design studies can be performed without needing to regenerate the system model. This variable parameter feature also applies to prescribed motion constraints so that they may be turned on and off during simulation.

Comments

Commentary text (following "#") may be included in the system description, making it easier to share and archive system descriptions to be re-used at a later date or by another person.



User-Selectable Prefix on SD/FAST-Generated Simulation Code

The complete set of simulation routines generated by SD/FAST can be prefaced by user-selectable characters. This allows several multibody systems to be simulated *simultaneously* with interacting forces and torques. The prefixes also allow the user to assemble *libraries* of models for effective management of large simulation projects.

Error Diagnostics

SD/FAST reports any problems found in the System Description File with clear and detailed error messages. These include both syntactic or typographical errors and problems with the physical realizability of the system described.

Information File Produced

After successfully processing the System Description File, SD/FAST produces an information file with tables showing the body interconnections, joint types, coordinate definitions, state/joint index maps, system parameters, and the number, type and placement of constraints. This is useful for verifying that the system has been correctly specified, and for associating run time quantities with their corresponding bodies and joints. Also produced is a list of all the array dimensions needed to call SD/FAST-generated routines.

T1.5.4 Generated Code

Following are some of the more important features of the SD/FAST-generated subroutines:

Extremely Efficient

$$\begin{array}{c} a + b + c - b \\ \downarrow \\ a + c \end{array}$$

SD/FAST uses *advanced Kane's dynamics and our proprietary Order(N) formulation* for the equations of motion combined with *symbolic algebra* to precompute and simplify many of the terms. This produces equations that execute many times faster than simulations produced by conventional methods (i.e., with a general purpose numerical multibody simulation program).

Multiple Output Languages Available

The SD/FAST-generated simulation code is produced in either Fortran or C as requested. Generated Fortran routines can be compiled with any ANSI-standard Fortran 77 compiler that has an extension allowing names up to 16 characters in length[5]. Generated C routines can be compiled with most C compilers, including both K&R and ANSI C. The routines are designed so that *optimizations* commonly performed by compilers will be highly effective.

User-Selectable Precision

ACSL

Matrix-X System Build

Matlab Simulink

Easy 5

***Your* Special Analysis Tools**

The SD/FAST-generated simulation code may be generated in either *single* or *doubl*e precision, as specified by the user in the System Description File or on the command line. (We normally recommend double precision, however.)

Compatible with Simulation Languages

Because the output routines are in standard Fortran or C, they can be called from any of the commonly used simulation languages such as ACSL or Matrix-X System Build. They can be used with numerical libraries like IMSL and NAG as well. Also, experience has shown that the generated routines fit smoothly into almost *any* common simu-

5. The ANSI-standard Fortran 77 requires maximum 6 character names. We have used longer names for improved readability, and most Fortran compilers have this extension.

lation methodology, resulting in much higher performance without requiring major changes to the way users work.

Constraint Stabilization

When using Ordinary Differential Equation (ODE) integrators (such as those provided with SD/FAST) to integrate constrained systems, the position and velocity constraints can drift if they are not stabilized. SD/FAST prevents this drift by using the *Baumgarte Stabilization*[6] method. The Baumgarte constants are user-selectable, providing for the most efficient stabilization on a problem-specific basis. Several callable routines are provided for evaluating constraint drift so that cases in which stabilization is necessary are easily identified.

Compatible with Differential-Algebraic Integrators

A routine is available for computing residuals as needed by a Differential-Algebraic Equations (DAE) integrator. For those familiar with these methods, the problem is presented in Index 2 form. This eliminates the potential for drift in the velocity constraints. Position constraints can be stabilized (if necessary) using Baumgarte feedback.

Reusable Numerical Library

In addition to the problem-specific generated code, SD/FAST outputs a separate Library File containing a set of generic routines which have proven useful in mechanical system analysis. These routines included numerical integrators and linear and nonlinear root finders as described in General Analysis Routines on page T-15. Other routines for performing commonly needed calculations and transformations are available.

Error Diagnostics

Run time error-checking routines can be called to find errors such as not specifying variable parameters, exceeding array dimensions, and improper use of the generated routines.

Powerful Routines Available for Extracting Information

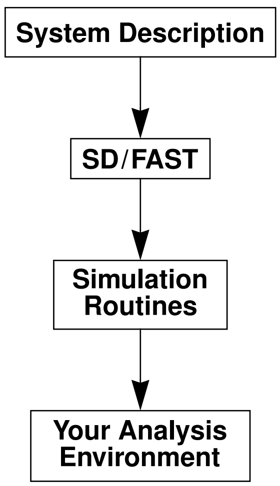
```
call sdreac(frc,trq)
call sdacc(bdy,pt,acc)
```

- **Directly Computes Bearing Reaction Forces and Torques:** Any component of bearing reaction force and torque at any joint can be computed. The constraint Lagrange Multipliers are also available.
- **Full Kinematic Information:** The orientation, angular velocity and angular acceleration of any body; the position, velocity and acceleration of any point on any body; and the position, velocity and acceleration of any joint axis are available in any coordinate frame.
- **Coordinate Frame Conversions:** Routines are available for transforming a vector defined in any coordinate frame to any other frame. Other utility routines are available for conversion of body (coordinate frame) orientation among three forms: direction cosines, Euler angles, and Quaternions (Euler parameters).
- **Total System Kinematic and Dynamic Information:** System angular momentum, system kinetic energy, system linear momentum, system mass center, and system in-

6. Baumgarte, J. “Stabilization of Constraints and Integrals of Motion in Dynamical Systems,” Comp. Meth. Appl. Mech. Engr., 1 (1972), 1-16.

ertia are available. These quantities can also be used to verify that conservation laws are being obeyed by the simulation.

T1.6 Using SD/FAST



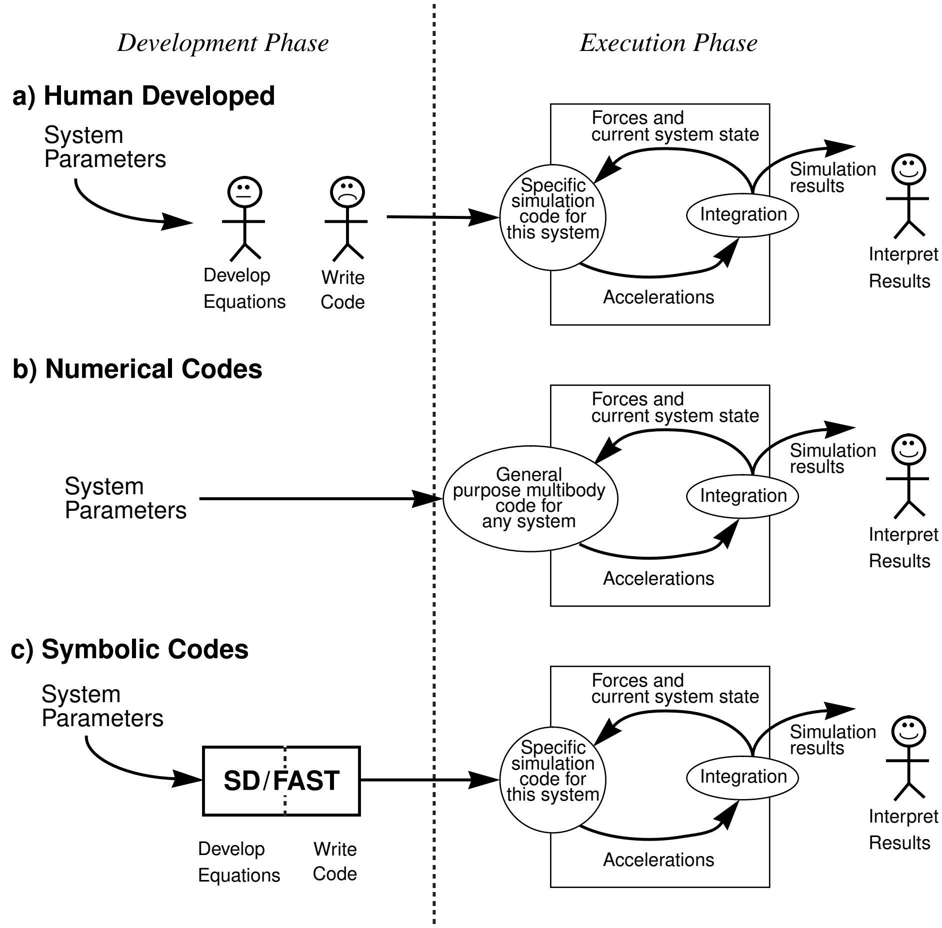
SD/FAST handles only *one part* of the simulation process. It produces the equations of motion describing the dynamic behavior of the system being simulated. For many systems of interest, the equations of motion account for the largest portion of the simulation code. They are also *extremely difficult to write by hand* and very difficult to program correctly on a computer. Furthermore, they are in some sense the least interesting part of the simulation—there is an unambiguous “right answer”. While the task of generating equations of motion for a large multibody system is extremely hard, and may take an experienced analyst many months, it leaves little room for engineering creativity. Some creativity is often required in the valiant attempts made to debug the equations after it is realized that they are not producing correct results. But the real talents of good engineers are largely wasted in “grinding out” equations of motion.

It is important to point out that SD/FAST is *not* a simulation language or environment. The philosophy behind SD/FAST is to produce the critical dynamic model *portion* of a multibody simulation in an open way which allows it to be used with a wide variety of other analysis tools. The complete simulation or other analysis will in general include models of sensors, actuators, materials, and control systems as well as an integrator and often a plotting or animation system for displaying the output.

SD/FAST provides some examples of these tools, such as integrators. However, it does not lock you into *our* idea of what is the best approach to the overall analysis. Conversely, SD/FAST is designed to integrate smoothly into *your* idea of the best simulation approach. SD/FAST will continue to work well in your environment even as you upgrade your analysis methodologies to take advantage of new technologies.

The rest of this section is intended to provide some perspective about how SD/FAST relates to other methods of analysis of mechanical systems. Figure T1-6 shows three ways in which a simulation of a mechanical system can be implemented. For each of these methods, we discuss the trade-offs which occur in development time versus execution time.

Figure T1-6 Three Ways to Model System Dynamics in a Simulation



Human Developed

Figure T1-6(a) represents the process of generating equations of motion by hand. Engineers work from a specification of the system parameters to develop the explicit equations of motion of that system. These equations are then coded in a programming language, such as a Fortran or C subroutine.

This routine is then incorporated into a larger (often pre-existing) simulation framework. The complete simulation maintains the system state vector, and uses it in combination with external commands to compute forces and torques. These forces and torques are passed to the hand-generated routine which computes the resultant accelerations. These accelerations are then integrated by a suitable numerical method and used to modify the simulation's representation of the system state, which is monitored to obtain information about the system's behavior. The process is repeated at each time step during the simulation.

Numerical Codes

As may be apparent from the above discussion, generating an executable model of a system's dynamic behavior is an *ideal task for a computer* to do. It is a shame to waste the creativity and engineering judgement of a valuable human being on this tedious task. In recognition of this, computer codes have been available since the early 1970's which contain generic *numerical codes* for the equations of motion of multibody systems. These codes can be used to model system dynamics in the same way as a hand-written routine containing specific equations of motion. Figure T1-6(b) shows how a general purpose code can be used instead of the hand-written code in Figure T1-6(a). The specific system parameters are provided to the multibody code at simulation run time. These parameters are used in the fully-general multibody model to produce the dynamic response of the desired system.

There are several drawbacks to the use of a *general purpose multibody code* of this type. By far the biggest problem is that the *run time performance is abysmal*. The general purpose multibody program is unable to exploit the simplifications available in most physical systems to reduce the complexity of the equations. The engineers of Figure T1-6(a) will notice the symmetric bodies, perpendicular or coplanar hinge axes, terms which cancel out or must evaluate to zero, identical expressions which need be computed only once, and many other opportunities for simplification which abound in the task of programming equations of motion. The general purpose multibody code, on the other hand, will be unable to take advantage of any of these since the equations were written long before the specific system parameters are known. From actual measurements, between 60 and 90 percent of the computations performed in executing the most general form of the equations of motion of a typical multibody system are unnecessary.[7]

Symbolic Codes

Figure T1-6(c) shows the SD/FAST approach. Instead of acting as a replacement for the human-generated equations of motion in the development phase of a simulation, SD/FAST acts as a replacement for the *humans*! Like the engineers in the left half of Figure T1-6(a), SD/FAST takes a description of a particular system and produces a Fortran or C subroutine containing the equations of motion derived *specifically* for that system. This routine can then be incorporated into the larger simulation framework. Since the generated routine is written with full knowledge of the system parameters, it is able to employ all the simplifications available in the physics of the system to produce the simplest possible equations of motion. Consequently, this technique combines the high run time performance of hand-generated equations of motion with the accuracy and shortened development time provided by the generic-equation codes.

With SD/FAST you can produce simulations whose performance is comparable or better to that provided by hand-written equations of motion, but with the reliability and productivity of a general purpose multibody code. Since SD/FAST generates explicit Fortran or C subroutines just as a human programmer would, its equations can be used

---

7. Rosenthal, D.E. and Sherman, M.A., "High performance multibody simulations via symbolic equation manipulation and Kane's method," *The Journal of the Astronautical Sciences*, Vol. 34, No. 3, July-September 1986, pp. 223-239, (also available from your SD/FAST distributor) discusses in detail this problem and others associated with the generic-equation approach, and gives specific performance measurements.

in any environment in which you would have been able to use the hand-generated code. And you can go back to interesting and intellectually challenging work such as designing appropriate models, designing control systems, developing simulation methodologies, and interpreting results.

Summary

- First read the Tutorials in sequential order to learn how to use SD/FAST. Then use the Reference section, Index, and Application Notes to develop your specific simulation. Hands-on practice with the sample problems in the Tutorials is the quickest way to develop proficiency in using SD/FAST.
- SD/FAST can model a wide variety of mechanical systems that can be represented as a set of joint-connected rigid bodies influenced by forces and restricted by constraint conditions.
- Your simulation environment interacts with SD/FAST via routine calls. A simple four bar linkage example demonstrates the powerful routines available for the user, such as one-line calls to initialize or assemble a model.
- An SD/FAST user must have a certain set of prerequisite skills to make effective use of the program. These include the ability to write modest programs in Fortran, C or a simulation language, the ability to produce plots of numerical results, and an understanding of the basic concepts of mechanics.
- A natural, easy-to-use system description format makes entering models straightforward. An extremely efficient Fortran or C source code model of the system is then generated by SD/FAST for incorporation into your favorite simulation environment, or for direct use from a user-written driver routine.
- SD/FAST provides an extremely powerful and flexible tool for generating multibody simulations by combining the best features of human-generated code with the automation, accuracy, and speed of a computer.

TUTORIAL 2

Simple Pendulum

Objectives

- Go through a complete example from start to finish.
- Develop the model.
- Write the system description input file.
- Run SD/FAST.
- Incorporate SD/FAST simulation code into a simple simulation.
- Run a suite of static, dynamic, and analytic analyses on the pendulum.

T2.1 A Complete Example

This tutorial will take you through a complete example from start to finish. Not all the features of SD/FAST will be covered, but the process described will be similar for all the simulations you will develop using SD/FAST.

The basic steps in using SD/FAST for any simulation project are:

1. Develop a model of the system.
2. Write the input System Description File.
3. Run SD/FAST.
4. Incorporate the SD/FAST-generated routines in your simulation environment.

The necessary computer files for this example can be found by looking in the SD/FAST installation directory for all the files called:

```
pend*
```

where "*" stands for a suffix indicating a System Description File, a file containing all the SD/FAST-generated routines, or the main program controlling the analysis and input/output. Please see the Release Notes for your system to determine the recommended character extensions and suffixes. This tutorial will use character extensions and suffixes for a simulation developed in Fortran on a UNIX-based Sun Workstation. The files for this case are:

`pend.sd`	User-generated input System Description file
`pend_dyn.f`	SD/FAST-generated DYNamic simulation code
`pend_sar.f`	SD/FAST-generated Simplified Analysis Routines
`pend_info`	SD/FAST-generated INFOrmation file
`pend.doc`	Machine-specific DOCumentation for running this example
`pend.f`	User-generated simulation Fortran code
`sdlib.f`	SD/FAST-generated LIBrary file (required for all analyses)

Note that SD/FAST used the file prefix "`pend`" to automatically generate the files `pend_dyn.f`, `pend_sar.f`, and `pend_info`. The automatic prefix ("`pend`" in this case) can be selected by the user as will be shown shortly. Avoid using "`_dyn`," "`_sar`," and "`_info`" in any file names that you generate.

The library file "`sdlib.f`" contains analysis code that is independent of a particular model, such as integration routines, and needs to be generated and compiled only once for each version of the software you receive. Refer to your Release Notes for the method of generating this file. Be sure to include this file in your link commands.

T2.2 Develop Model

There are six steps in developing the system model to be entered in the System Description File:

1. Draw picture and define topology.
2. Define coordinate frames and the Reference Configuration.
3. Specify geometry and joints.
4. Specify mass properties.
5. Specify constraints and prescribed motion.
6. Specify gravity if needed.

Not all of these steps are necessary for this example, but will be used for other systems. We shall use this pattern to present a rigorous approach to using SD/FAST. We suggest that you try our methodology before developing your own.

T2.2.1 Draw Picture and Define Topology

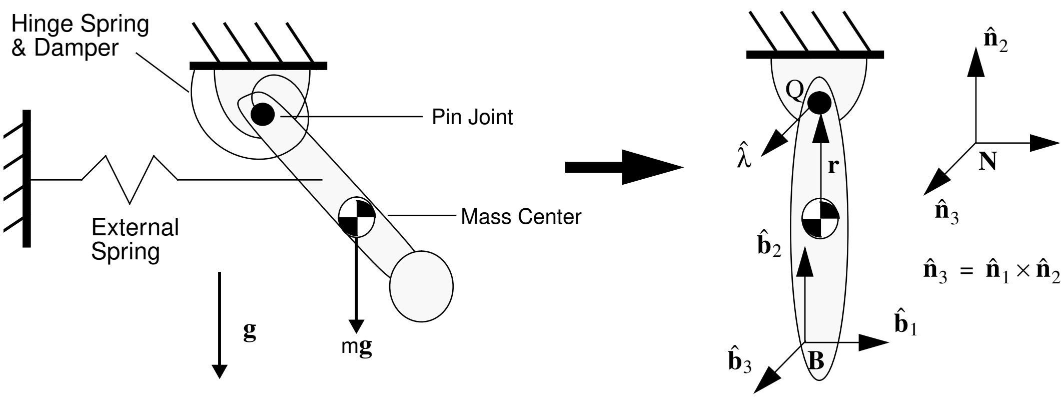
Figure T2-1shows a simple pendulum on the left with various internal and external forces and torques applied. On the right we have drawn a simple "blob" picture of the system showing the topology (one body in an open-tree structure, connected to the ground), and the joints (a single pin joint). The forces and torques in this problem are gravity acting downward, an external spring attached to the pendulum, and a torsional spring and damper at the hinge. Note that we have put the system in a so-called *refer-*

ence configuration and have added coordinate frames and vectors to be explained shortly.

Key Idea

The key idea in this step is to draw a simple picture devoid of extraneous information, such as the shapes of the bodies, so that just the information essential for the dynamic model is clearly shown.

Figure T2-1 Model of a Simple Pendulum



Original System *Reference Configuration*

T2.2.2 Define Coordinate Frames and the Reference Configuration

The so-called *reference configuration* shows the bodies in a different configuration than the original system sketch, and coordinate frames and vectors have been added. This is necessary to generate the proper information concerning the location and orientation of each body and joint for the input file and for interpreting results from the simulation. Correct understanding and usage of coordinate frames and vectors defined within them are critical to using SD/FAST properly. Before we rigorously define the reference configuration, we shall discuss coordinate frames.

There are only *two* types of coordinate frames: *inertial and all others*. Note that all coordinate frames are assumed to be right-handed and orthogonal. That is, the coordinate frame is defined by a set of sequentially-numbered unit vectors arranged orthogonal to each other and obeying the equation: $\hat{\mathbf{n}}_3 = \hat{\mathbf{n}}_1 \times \hat{\mathbf{n}}_2$, where boldface **n** is a vector, the hat means it is a unit vector, and the cross represents a vector cross-product. We use the convention that the frame defined using unit vectors $\hat{\mathbf{a}}_i$ is represented by the boldface capital letter **A**; similarly, **N** for $\hat{\mathbf{n}}_i$, etc.

Inertial Frame

Symbol for Inertial Frame

The *inertial* or *ground frame* is that frame about which all the equations of motion are generated. It is always present and must be used at least once in the definition of the problem. In the input file, it is referred to by the reserved keyword “`$ground`”. We shall reserve the frame **N** for inertial frames in all examples in this manual. The selection of the inertial frame must be carefully considered by the analyst. Since no truly inertial frame exists, one must be chosen. For mechanisms and ground vehicles, a frame attached to the Earth is usually sufficient. However for vehicles in orbit, a frame attached to the Earth, the Sun, or moving along the orbit may be used, depending upon relative accelerations generated.[1] In most cases, common sense can be used to select the inertial frame.

All Other Frames

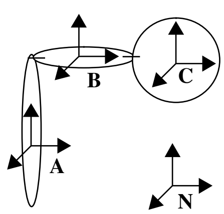
All other coordinate frames include one attached to *each* rigid body and any others which you may wish to define for your use. For the purposes of defining the position and orientation of a rigid body, a coordinate frame and rigid body are *identical* (as opposed to a flexible body which may undergo relative motion or deformation). For this example, the frame B is attached to the pendulum.

Reference Configuration

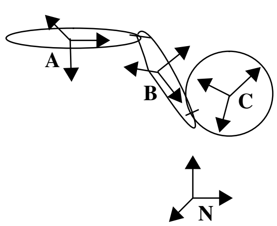
Key Idea

The key idea behind the reference configuration is that *when all of the generalized coordinates defining the position of each body or frame in the system are equal to zero, the axes of all the coordinate frames are exactly aligned.* More importantly, *all geometry, mass properties, joint locations and orientations are defined in this configuration.*

Thus, in the pendulum example we have chosen the reference configuration to be that position with the pendulum hanging straight down, which is the "natural" position usually chosen, and the angle defining the position of the pendulum is zero. Other positions could have been chosen; it is your choice. Reference configurations are chosen for a variety of reasons, including a "natural" equilibrium position, or one in which the geometries and mass properties are easy to compute, such as bodies aligned or at right angles and coordinate frames aligned with axes of symmetry of each body.



A Reference Configuration



Subsequent Motion

Some care should be given to selection of the reference configuration. You can take better advantage of SD/FAST's symbol manipulator by choosing a reference configuration in which many of the parameters, such as products of inertia, joint locations, and pin vector coordinates, are zero. Because explicit equations of motion are generated from the description of the system in its reference configuration, a choice which results in geometry being described by vectors with some components being "0" or "1" will generate simpler equations (which run faster) than vectors with more general orientations and lengths. A reference configuration in which all pins are aligned on a single axis is ideal. Initial conditions at runtime can be used to put the hinges into the starting configuration—*it is not necessary for the reference configuration to be the starting configuration.* The performance effects of various reference configuration choices are discussed further in Reference Section R16.2.

T2.2.3 Specify Geometry and Joints

Now that the general topology, coordinate frames and the reference configuration have been defined, numerical specifics for the geometry and the joint are required. The geometry and joint information are all given in terms of *vectors* written in the chosen reference configuration. Since all the coordinate frames are aligned in the reference configuration, *it doesn't matter* in which frame (**B** or **N**) you write the vector!

---

1. For a rigorous methodology in selecting inertial frames, see *Dynamics: Theory and Applications*, Kane, T.R., and Levinson, D.A., McGraw-Hill, New York, 1985, pp. 158-169.

For *any* system, the only information needed to specify the geometry are vectors from a body mass center to a joint. The specification of which body is connected to which joint naturally leads to the system topology, as we shall see. For the pendulum shown in Figure T2-1, the vector from the mass center of the pendulum to hinge point Q lying on the axis of the pin joint is given by:

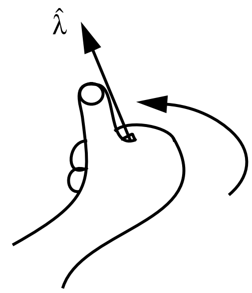
$$\mathbf{r} = 1.5\hat{\mathbf{n}}_2 \qquad \textbf{Eqn. T2.1}$$

which is equivalent to $1.5\hat{\mathbf{b}}_2$. It is important to note that this vector is *not* unique for a 1-D rotational joint (pin joint) or 1-D translational joint (slider joint). The point Q can lie anywhere along the hinge axis. See Reference Section R11.4 (Pin Joint) for more details.

The orientation of the hinge axis for the pendulum is given by the hinge vector:

$$\hat{\lambda} = \hat{\mathbf{n}}_3 \qquad \textbf{Eqn. T2.2}$$

Note that the hinge vector is always a unit vector[2] (in this case it happened to line up with $\hat{\mathbf{n}}_3$, but it can be in any direction). Also a relative rotation between the connected bodies that yields a "right-hand-rule" vector aligned with $\hat{\lambda}$ gives an *increasingly positive* value in the coordinate measuring that rotation. (See Reference Section R11 for rigorous definitions and descriptions.)



Right-Hand Rule
Gives a Positive Rotation

Thus for the pendulum, the coordinate describing the rotation of the pendulum with respect to the ceiling is zero when the pendulum hangs straight down (its location in the reference configuration) and increases positively as the pendulum swings to the right (the "right-hand-rule" applied to the rotation gives a vector aligned with $\hat{\lambda}$).

We have not mentioned units yet. Only one type of unit, angular displacement, is assumed and it is always in *radians*. All other units including mass, length, and time units are left *unspecified*. That is, the distance of 1.5 from the mass center of the pendulum to the pin joint could be in feet, meters, miles, etc. These units are irrelevant to SD/FAST. *It is the user's responsibility to choose consistent units!* For example, if time is in seconds, length is in meters, and mass is in kilograms; inertia must be in kg-m^2 and force must be in kg-m/s^2 (= 1 newton). See Reference Section R21 for more information.

For this example, we shall use the MKS system (meters, kilograms, seconds) and choose the distance from the pendulum mass center to the pin joint to be 1.5 meters.

T2.2.4 Specify Mass Properties

Each body in the system must have a mass and inertia matrix specified. Mass is a scalar quantity, and for this example we choose 10.0 kg as the mass of the pendulum. Note that the ground or inertial frame is considered to be of "infinite mass" and is not specified. Massless bodies created by entering zero for the mass and/or inertia, can also be

---

2. SD/FAST automatically normalizes any hinge vector you give it. Thus a hinge vector given by $\hat{\lambda} = 3\hat{\mathbf{n}}_2 + 4\hat{\mathbf{n}}_3$ will be normalized to $\hat{\lambda} = \frac{3}{5}\hat{\mathbf{n}}_2 + \frac{4}{5}\hat{\mathbf{n}}_3$.

used to connect pre-defined joints to create special joints. These should be used cautiously, however, and in accordance with the restrictions discussed in Reference Section R13.3. No massless bodies are used in this example.

The inertia matrix (alternatively dyadic or tensor of second order) of each body is specified about the *mass center* of that body, for that body in the *reference configuration*. The rigorous definition of the SD/FAST inertia matrix is given in Reference Section R13.2. It is important to note that different engineering disciplines use different conventions for the sign of the products of inertia (the off-diagonal terms of the inertia matrix). Most dynamicists, including us, use the “minus sign” convention, while some structural analysts use a “plus sign” convention. Make sure that you know which convention is employed by your source of mass properties so that you can reverse the sign of the products of inertia if necessary.

For the pendulum example we shall use the inertia matrix:

$$\mathbf{I} = \begin{bmatrix} 5 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 5 \end{bmatrix} \text{ kg-m}^2 \qquad \textbf{Eqn. T2.3}$$

Again, the ground or inertial frame is immovable and has no specific mass properties.

T2.2.5 Specify Constraints and Prescribed Motion

The motion constraint imposed by the pin joint has already been specified. No other constraints or prescribed motion are needed for this problem. Later Tutorials make use of these features.

T2.2.6 Specify Gravity

The pendulum operates in a uniform gravitation field in the $-\hat{\mathbf{n}}_2$ direction with a magnitude of 9.8 m/s^2. This constant known force will be specified in the input file so that the symbol manipulator can incorporate the known constant directly into the equations of motion. The springs and damper generate time varying forces and torques on the pendulum or at the joint. They are specified at runtime using appropriate routine calls as discussed in Section T2.5.

T2.3 Write System Description File

Now we are ready to write the SD/FAST input System Description File using the information we gathered as we developed the model. The example System Description File is named `pend.sd` in your SD/FAST installation directory. Since the System Description File is free-format, the exact format is completely up to the user. However, we suggest that you try using the format shown in this manual. See Reference Section R19 for exact rules of syntax. The contents of `pend.sd` is shown in Figure T2-2.

Figure T2-2

System Description File for Pendulum Example

Comments

```
#  File:  pend.sd
#
#  This is an SD/FAST input file describing the pendulum example in
#  Tutorial 2 of the SD/FAST User's Manual
#
#  The system is a one degree of freedom pendulum with gravity acting
#  in the -n2 direction; the pin joint along n3.
#
```

Preamble

```
gravity = 0 -9.8 0
```

Body Paragraph

```
body = pendulum  inb = $ground  joint = pin
  mass = 10?
  inertia = 5  1  5
  bodytojoint = 0  1.5 0
  pin = 0 0 1
```

That's the entire input file! We always start with a comment section describing the system which is modeled. Any line beginning with "#" is treated as a comment and is not processed. The rest of the file is a sequence of "paragraphs": a preamble of globally applied commands followed by a paragraph for each body.

T2.3.1 Write Preamble

The preamble contains any information that applies to the *entire* system. In this case only information on gravity is included in the preamble. Other keywords such as "`single`" for generating single precision code would be included here. See Table R-3 on page 108 in the Reference Section for a list of all SD/FAST keywords and their usage. Following the keyword "`gravity`" is an equal sign followed by three scalars. These scalars are the measure numbers (or components) of the vector representing gravity in the reference configuration:

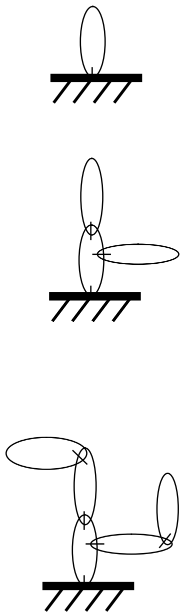
$$\mathbf{g} = 0\hat{\mathbf{n}}_1 - 9.8\hat{\mathbf{n}}_2 + 0\hat{\mathbf{n}}_3 \qquad \textbf{Eqn. T2.4}$$

Gravity always remains constant[3] with respect to the inertial frame **N** as subsequent motion occurs. Since we are using MKS units, the acceleration due to gravity is in units of m/s^2.

T2.3.2 Write Body Paragraph for Pendulum

The first body paragraph contains all the information concerning the one pendulum. The keyword "`body`" is followed by an equal sign and its alphanumeric identifier chosen by the user. This identifier is useful when describing connections to it by other bodies and when interpreting simulation results. All information that is listed after "`body=`", but *before* the next "`body=`" is assumed to apply to that s*pecific body only*, hence the natural tendency to use a separate paragraph for each body.

3. SD/FAST can also be used with more complex gravity models such as gravity gradients, using the general force application methods rather than the "`gravity`" keyword.



Add bodies by connecting to previously defined bodies.

The next phrase "`inb = $ground`" tells SD/FAST to which body this body is being connected. It is identified as "`$ground`". `$ground` is a built-in body name that indicates the inertial or ground frame. This *inboard* body (hence, `inb`) must be already defined (`$ground` or one you previously defined). Thus the order of the paragraphs is very important. The global keywords must appear first in a preamble, and *each body paragraph must reference an already defined body.*

The next phrase "`joint = pin`" tells SD/FAST that the connection to the *inboard* body is a simple pin joint. Both of these words are SD/FAST keywords. Joint connections to *outboard* bodies never need be specified since the system topology can be specified using only inboard connections. More complex system topologies will be demonstrated in later Tutorials. The vector $\mathbf{r} = 1.5\hat{\mathbf{n}}_2$ (Eqn. T2.1) from the center of mass of the pendulum to point Q on the pin joint is passed using the phrase "`bodytojoint = 0 1.5 0`", where the implied units are meters. (Remember that SD/FAST makes no assumptions regarding units except that angular units are radians. The user is responsible for using consistent units.) The axis of positive rotation of the pin joint $\hat{\lambda} = \hat{\mathbf{n}}_3$ (Eqn. T2.2) is passed using the phrase "`pin = 0 0 1`".

The mass of the pendulum is passed to SD/FAST using the phrase "`mass = 10?`" where the units are implied to be kilograms to be consistent with other quantities. Note that a question mark has been added so that we can change the mass to a different value at runtime, although 10 will be the default. Had we specified just the numerical constant, the SD/FAST symbol manipulator would fold the constant into the equations of motion for maximum efficiency, but regeneration would be required to change the value. If we had just supplied the "`?`", there would be no default value.

The inertia (Eqn. T2.3) is passed to SD/FAST using the phrase "`inertia = 5 1 5`". There are two ways of passing inertia properties: (1) via 3 numbers which assumes that the three numbers are the principle inertias and all the products of inertia are zero, or (2) via 9 numbers which are all the components of the inertia matrix, entered by rows. Any legitimate inertia matrix is symmetric about the main (top left to lower right) diagonal. You can avoid duplicate entry by providing only one of each pair of symmetric terms, and specifying just a "`?`" as a placeholder for the other term. In this case the "`?`" does not represent any variability in the model.

Again, the implied units for inertia in this problem are kg-m^2 and the components are measured with respect to the reference configuration about that body's center of mass. The exact syntax and usage of all the keywords in the body paragraphs can be found in the Reference Section.

T2.4 Run SD/FAST

At this point we are ready to run SD/FAST and generate the equations of motion for the pendulum system. It is assumed that you have typed (or copied from the installation directory) the SD/FAST System Description File "`pend.sd`" onto a computer that has a licensed copy of SD/FAST available. There are two ways of running SD/FAST: (1) a verbose interactive mode useful for debugging the model, and (2) a quiet batch mode for iterating small changes in the model.

T2.4.1 Interactive Mode

For this example, follow this script:

1. To start SD/FAST, type: `sdfast`
2. Copyright, licensing, and time-stamp information should be followed by:
3. `System description file:`

   To which you type: `pend.sd`
4. The System Description File is replayed and if the file is syntactically acceptable you see:

   `Input file successfully processed.` Followed by:
5. `Name for generated Dynamics File: (pend_dyn.f)`

   To which you type a carriage return to get the suggested default. Followed by:
6. `Name for generated Information File: (pend_info)`

   To which you type a carriage return to get the suggested default. Followed by:
7. `Name for generated Analysis File: (pend_sar.f)`

   To which you type a carriage return to get the suggested default. Followed by:
8. The SD/FAST "Roadmap" of the system is listed:

```
ROADMAP (pend.sd)

Bodies        Inb
No  Name      body Joint type  Coords q
--- --------- ---- ----------- ----------------
  0 $ground
  1 pendulum    0  Pin(1D)       1
```

   from which you can check if the system described is indeed the one you thought you described. This roadmap is duplicated in the "`_info`" file and will be described in detail in the next section.
9. Information on the progress of computing various parts of the dynamics is followed by a summary of equation complexity in terms of the number of adds, multiplies, divides and assignments. SD/FAST should terminate normally with no error messages and return the system prompt.

If SD/FAST terminates with no errors and the Roadmap is consistent with the system you wanted to model, then you are ready to incorporate the SD/FAST-generated simulation code in the file "`pend_dyn.f`" with your simulation environment[4]. If errors occur, information messages (such as "`Unrecognized joint type 'glorp' for body 'pendulum'`") should lead to correction of the error.

T2.4.2 Batch Mode

A way to run SD/FAST without generating all the screen output is to pass the system description input file directly to SD/FAST on the first line entered:

4. "`pend_dyn.f`" is created by appending "`_dyn.f`" to the *basename* of the input file name: "`pend`" from "`pend.sd`". Suffixes are different under DOS or NT, see Section R7.2.

```
sdfast pend.sd
```

After the work is all done and the output files (using the default file names) are generated, the prompt will return quietly unless errors are found. This should take only a few seconds.

Other options can also be entered on the command line that can select, for example, use of the Order(N) formulation, desired precision, language, and output file “basename” prefix used. See Reference Section R7.1 and the Release Notes for your machine for details.

T2.5 Create Simulation

At this point, we have all the SD/FAST-generated routines for simulating the dynamics of the simple pendulum. As shown in Figure T1-6 on page T-21, we now need to add routines for numerically integrating the equations of motion and passing the results to the output device of choice. The steps for creating a simulation are:

1. Choose method of using SD/FAST routines.
2. Extract reference information from information file.
3. Write simulation driver code.
4. Run simulation.

T2.5.1 Choose Method of Using SD/FAST Routines

Section T1.5.2 on page T-14 discussed the different methods which can be used with SD/FAST-generated routines to perform analyses and design studies. These methods include the use of Simplified Analysis Routines (Reference Section R18), General Analysis Routines (Reference Section R8), external analysis environments and libraries, and user-written analysis code.

In this example, we will use only Simplified Analysis Routines and a little user-written code. In general, unless you are using an external analysis environment, we recommend the use of Simplified Analysis Routines if they can do what you need.

Simplified Analysis Routines

These are a set of six simple “we-have-done-it-all-for-you” routines that allow you to quickly write and test most simulations or portions of simulations. The Simplified Analysis Routines are in the generated “`_sar`” file. These routines will probably allow you to perform most of your analysis tasks, but do *not* provide completely general analysis capability. For more general analyses without use of an external analysis environment, use the General Analysis Routines.

We will only use two of the six Simplified Analysis Routines in the pendulum example:

- `SDSTATIC()`: Finds static equilibrium configurations of the system. See Reference Section R18.3 and the Quick Reference Guide. More general capability, including inverse static analysis, can be performed with the underlying General Analysis Routine `SDROOT()` as described in Reference Section R8.2.

- `SDMOTION()`: This routine propagates the system equations of motion through time using a Runge-Kutta-Merson variable step integrator. See Reference Section R18.5 and the Quick Reference Guide. Note that `SDMOTION()` is limited in its usefulness because it can *not* be used to propagate user-defined differential equations such as controllers. The General Analysis Routine `SDVINTEG()` (Reference Section R8.1) provides a more sophisticated capability. Fixed step integration is also available, as required for real-time simulations, and is discussed in these same Reference Sections.

T2.5.2 Extract Reference Information

Before you can write the main simulation code, you should examine the Information File produced by SD/FAST to find the reference numbers and array dimensions for making calls to SD/FAST-generated routines. The default name for this file is created by adding "`_info`" to the root of the input file name; in this case, "`pend_info`" (Figure T2-3).

The Information File contains (1) a Roadmap, which describes the system topology as a cross-check of the input System Description File, (2) a State Index Map which maps the "position" states, the *q*'s, and the "velocity" states, the *u*'s, to their locations in the state vector; and (3) a list of system parameters and their values. The Information File is discussed briefly below, and in more detail in Reference Section R9.

Roadmap

The Roadmap is the table at the top of the Information File. The Roadmap for the pendulum system is very simple. Reading down the Roadmap, we see that the first body is "`$ground`," which is numbered "`0`". The next body is "`pendulum`," which is numbered "`1`" and is connected to the inboard body (`Inb body`) "`0`," the ground. The type of joint connecting the pendulum to the ground is a 1-D pin joint (`Pin(1D)`). This description matches the system we expected.

State Index Map

The second table in the information file is the state index which maps the "position" states, the *q*'s, and the "velocity" states, the *u*'s, to their locations in the state vector. There are two states for the pendulum. The first, and only, position state or coordinate of the pin joint, *q*, is the first component of the state vector. The velocity state, *u*, for the pin joint is the second component. More complex examples will follow in later tutorials.

Figure T2-3 SD/FAST Information File for Pendulum

Roadmap

State Index

System Parameters

```
SD/FAST Information File: pend.sd
Generated 26-May-1990 17:21:30 by SD/Fast, Kane's formulation
(sdfast BX.1.4 #10535) on machine ID 170065c5

ROADMAP (pend.sd)

Bodies        Inb
No  Name      body Joint type  Coords q
--- --------- ---- ----------- ----------------
  0 $ground
  1 pendulum    0  Pin(1D)       1

STATE INDEX TO JOINT/AXIS MAP (pend.sd)

 Index
  q|u   Joint  Axis   Joint type    Axis type
 -----  -----  ----   -----------   ----------
  1|2      1     1   Pin(1D)       rotate

SYSTEM PARAMETERS (pend.sd)

Parameter  Value  Description

nbod          1  no. bodies (also, no. of tree joints)
njnt          1  total number of joints (tree+loop)
ndof          1  no. degrees of freedom allowed by tree joints
nloop         0  no. loop joints
nldof         0  no. degrees of freedom allowed by loop joints

nq            1  no. position coordinates in state (tree joints)
nu            1  no. rate coordinates in state (tree joints)
nlq           0  no. position coordinates describing loop joints
nlu           0  no. rate coordinates describing loop joints

nc            0  total no. constraints defined
nlc           0  no. loop joint constraints
npresc        0  no. prescribed motion constraints
nuserc        0  no. user constraints
```

System Parameters

The third table of the information file lists the overall system parameters such as the total number of bodies, the number of degrees of freedom in the system, the number of loop joints, etc. For the pendulum, there is one body (`nbod`), which is also the number of tree joints and total joints (`njnt`) in the system. There is one degree of freedom (`ndof`) allowed by tree joints, one position coordinate (`nq`), and one rate state (`nu`). There are no loop joints, constraints, or prescribed motion in this problem.

Key Idea

It is important to note that you *must* use the body, joint, state, and axis numbers listed in the Roadmap and State Index Map when making calls to SD/FAST routines. Necessary arrays should be declared using the provided System Parameter values. These reference numbers and system parameters are also available by calling `SDINFO()`, `SDJNT()`, and `SDCONS()`, described in Reference Section R17.3. This feature is available for those developing general purpose simulation codes that automatically extract information about arbitrary models.

T2.5.3 Write Force and Torque Models

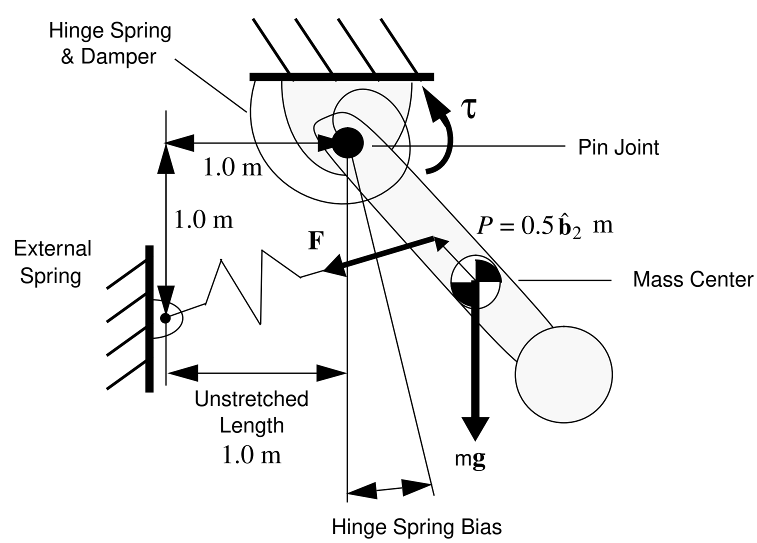
The main simulation program is called "`pend.f`" in your tutorial directory. Rather than reproduce that file in one figure, we discuss each segment as it is used and then show the computer output generated by that segment of code. This section covers the user-written force and torque models, and the next section covers the analyses performed. To get this example running on your machine, follow the directions in `pend.doc`. Various steps including compiling and linking the library file may be needed. While going through this example you may wish to have a printout copy of the entire code `pend.f`.

To use the Simplified Analysis Routines, you must write at least one, possibly two, routines that provide the rest of the system model required for simulation, namely, (1) the applied forces and torques through `sduforce()` [SD-User-FORCEs], and (2) prescribed motion functions through `sdumotion()` [SD-User-MOTIONs]. If you have user constraints defined, additional routines will be required; see Reference Section R24 (User Constraints).

We shall use only the first user-supplied routine, `sduforce()`, to compute the torsional spring and damper torques and the force from the externally attached linear spring. The other user-supplied routines are not needed for this problem and will be explained in later tutorials. Note that `sduforce()` *must* be written if you are linking with any Simplified Analysis Routines, even if it only returns without any forces or torques. This is because SD/FAST can not determine from the input file if the user wants to apply forces or torques, but it *can* determine if prescribed motions or user constraints are needed. Therefore the Simplified Analysis Routines always make a call to `sduforce()`.

A detailed drawing of the forces and torques applied to the pendulum are shown in Figure T2-4. The gravity force, m**g**, has already been accounted for in the input file.

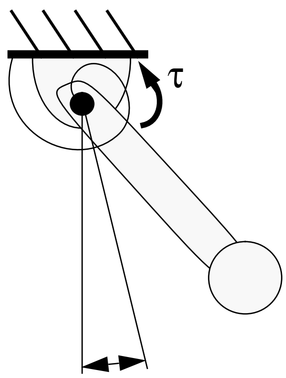
Figure T2-4 Forces and Torques Applied to the Pendulum



Write `sduforce()`: Hinge Spring and Damper

The torque applied at the hinge is given by:

$$\tau = \{-K_{\text{rotate}}(q - q_{\text{bias}}) - F_{\text{frict}} u\}\hat{\lambda} \qquad \textbf{Eqn. T2.5}$$



where q is the angle measured from vertical with a positive rotation counterclockwise about the hinge axis $\hat{\lambda}$, q_{bias} is the offset of the torsion spring zero point from $q = 0$, u is the angular rate of the pendulum, and K_{rotate} and F_{frict} are the spring and damping constants, initially 100 N-m/rad and 10 Nm/(rad/sec), respectively.

The initial set-up and hinge torque code segment of `sduforce()` is:

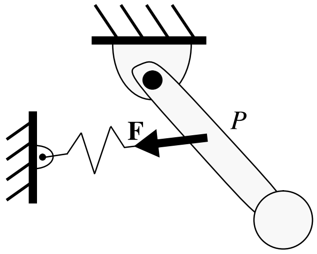
```
      SUBROUTINE SDUFORCE(T,Q,U)

c  Set up variables, common block to PEND, and all known parameters.

      REAL*8  T,Q(1),U(1),FFRICT,KROTATE,TAU,BIAS,FREELENGTH,KLINEAR
      REAL*8  PNTP(3),VVEC(3),FVEC(3),STRETCH,MAG,POSP(3),GNDPT(3)

      COMMON / USER1 / BIAS, FVEC

      DATA PNTP /0.0D0, 0.5D0, 0.0D0/
      DATA GNDPT /-1.0D0, -1.0D0, 0.0D0/
      FFRICT = 10.0D0
      KROTATE = 100.0D0
      KLINEAR = 10.0D0
      FREELENGTH = 1.0D0

c  Compute spring and damper hinge torque TAU and apply to axis 1 of joint 1.
      TAU = -FFRICT*U(1) - KROTATE*(Q(1)-BIAS)
      CALL SDHINGET(1,1,TAU)
```

The variables passed to `sduforce(T,Q,U)` are time, `T`, a vector of all the generalized coordinates of the system, `Q(NQ)`[5], and a vector of all the generalized speeds of the system, `U(NU)`. The variables will be explained when they are used. The common block is used to pass some variables to the main program for printing or changing during the simulation. We shall change `BIAS` during the simulation and use `FVEC` in some computations in the main program.

The hinge torque, `TAU`, is computed as in (Eqn. T2.5) with the variable names roughly equivalent to the mathematical symbols. `TAU` is passed to SD/FAST through the utility routine call `SDHINGET()` [SD-HINGE-Torque] where the first parameter is the joint number, the second parameter is the axis number of that joint (e.g., a U-joint has 2 numbered axes), and the third parameter is the hinge torque passed to SD/FAST.[6]

Write `sduforce()`: Attached Spring

The external force $\mathbf{F}$ applied to the pendulum by the linear spring is given by:



$$\mathbf{F} = K_{\text{linear}}(l - l_{\text{free}})\hat{\mathbf{v}} \qquad \textbf{Eqn. T2.6}$$

5. Symbols in parentheses such as "`NQ`" inside of `Q(NQ)` indicate a vector of size 1 x `NQ`.

6. The only supported way to pass information to and from the SD/FAST model is through variables in the routine list. Direct access to SD/FAST-generated common blocks or global variables is not supported and not recommended.

where K_{linear} is the linear spring constant, 10 N/m; l is the total length of the spring in meters; l_{free} is the free length of the spring in meters, i.e., the length of the spring for which the force it exerts is zero; and $\hat{\mathbf{v}}$ is a unit vector aligned with the linear spring in the direction of the force *applied* to the pendulum. Note that this vector equation must be transformed between various coordinate frames, as required by the various routines. The external spring force is rather complicated in that the direction of the applied force changes in a nonlinear fashion with changes in the pendulum angle.

The code written to implement this external spring force constitutes the remainder of the `sduforce()` routine:

```
c  Compute inertial location of linear spring attach point pntp, put into posp
       CALL SDPOS(1,PNTP,POSP)
c  Find a unit vector in the direction of the spring force
       VVEC(1) = GNDPT(1) - POSP(1)
       VVEC(2) = GNDPT(2) - POSP(2)
       VVEC(3) = GNDPT(3) - POSP(3)
       MAG = SQRT( VVEC(1)**2 + VVEC(2)**2 + VVEC(3)**2 )
       IF ( MAG .GT. 0.0D0 ) THEN
         VVEC(1) = VVEC(1) / MAG
         VVEC(2) = VVEC(2) / MAG
         VVEC(3) = VVEC(3) / MAG
       ELSE
         WRITE(6,*) ' Error in SDUFORCE: Zero length unit vector'
         STOP
       END IF
c  Find the stretch of the spring beyond its free length
       STRETCH = MAG - FREELENGTH
c  Find the resultant components of the linear spring force
       FVEC(1) = KLINEAR * STRETCH * VVEC(1)
       FVEC(2) = KLINEAR * STRETCH * VVEC(2)
       FVEC(3) = KLINEAR * STRETCH * VVEC(3)
c  Transform the spring force back to the pendulum frame
       CALL SDTRANS(0,FVEC,1,FVEC)
c  Apply the spring force as a point force acting at point P, PNTP, on pendulum.
       CALL SDPOINTF(1,PNTP,FVEC)

       RETURN
       END
```

We first compute the inertial position of the linear spring attach point P (`PNTP`) which is 0.5 meters along $\hat{\mathbf{b}}_2$ from the center of mass of the pendulum. This is accomplished by using the utility routine `SDPOS()` which computes the inertial position[7] of any point on any body. The first parameter passed to `SDPOS(1,PNTP,POSP)` is the number of the body, in this case 1, as we learned from the Roadmap (Figure T2-3). The second parameter passed is a vector *in the body frame* from the body mass center to the desired point, in this case, point P, $0.5\hat{\mathbf{b}}_2$ meters form the mass center. The third variable, `POSP`, is a vector of the returned position of the point *in the inertial or ground frame*, N.

The unit vector, `VVEC`, aligned with the linear spring is formed by subtracting the `POSP` vector from the linear spring ground attach point `GNDPT` ($-1\hat{\mathbf{n}}_1 - 1\hat{\mathbf{n}}_2 + 0\hat{\mathbf{n}}_3$), and then normalizing by its magnitude, `MAG`. The stretch of the spring is given by subtracting its

7. You may have noticed that we have not identified the location of the origin of the inertial frame N. An arbitrary location could have been specified by an `inbtojoint` (inboard-to-joint) vector in the pendulum body paragraph of the input file. Left unspecified, the default location is the *first* joint connection to ground.

free length from MAG. Finally the linear spring force is computed and resolved along the three components of VVEC.

Note that FVEC has been computed in the inertial frame, but must be transformed to the pendulum frame as required by SDPOINTF(). We do this by using the SDTRANS(0,FVEC,1,FVEC) utility routine which transforms from frame 0 (the ground or inertial frame) the vector FVEC to frame 1 (the pendulum) the vector FVEC. Note that we simply overwrote the new vector onto itself. Finally we call SDPOINTF(1,PNTP, FVEC) to apply to body 1 (the pendulum) at PNTP, the attach point of the spring, the external force FVEC, computed in the body frame.

T2.6 Run Analyses on Pendulum

Now that we have written the routines required to complete the force and torque models for the pendulum, we can write the main simulation or analysis program for performing the analyses. After setting up the main program, we shall perform a suite of static, dynamic, and analytic analyses.

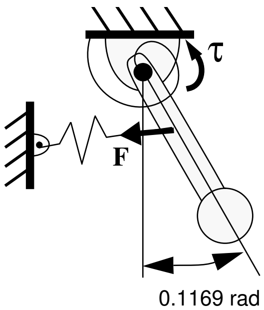
T2.6.1 Setup Main Program

The initial setup code of the main program pend.f is:

```
c  File:  pend.f
c
c  This is a simulation main program file that performs the static
c  and motion analyses for the pendulum example in
c  Tutorial 2 of the SD/FAST User's Manual, version B.1.0.
c
       PROGRAM PEND

c  Set up variables, common block to SDUFORCE, and all known parameters
       REAL*8  Y(2),DY(2),T,DT,CTOL,TOL,BIAS,FORCE(1,3)
       REAL*8  TORQUE(1,3),COMASS(3),ACC(3),M1,GRAV(3),FVEC(3)
       INTEGER NSTEP,ILOOP,LOCK(1),MAXEVALS,FCNT,FLAG,ERR,ISQUES,ISQUES3(3)

       COMMON / USER1 / BIAS, FVEC

       DATA COMASS /3*0.0D0/
       DATA Y / 0.0D0, 0.0D0 /
       T    = 0.0D0
       DT   = 0.05D0
       CTOL = 0.0D0
       TOL  = 1.0D-4
       LOCK(1) = 0
       MAXEVALS = 1000
```

The variables will be described as they are used. The common block USER1 is used to pass variables to and from sduforce() as explained before.

T2.6.2 Exercise 1: Perform Initial Static Analyses



Static Equilibrium for 10 kg Pendulum

The next segment of code performs a static analysis of the pendulum with the mass set to its initial value of 10 kg and the bias of the rotational spring set to 0.3 radians.

```
c  Initialize all SD subroutines before making any calls to analysis routines.
        CALL SDINIT

c  Perform static analysis for mass = 10 kg (default) and spring bias as shown.
        BIAS = 0.3D0
        CALL SDSTATIC(T,Y,LOCK,CTOL,TOL,MAXEVALS,FCNT,ERR)
        WRITE (6,*)
        WRITE (6,*) 'Static equilibrium for mass = 10 kg: '
        WRITE (6.*) ' Y = '.Y(1).'   FCNT = '.FCNT.'   ERR = '.ERR
```

`SDSTATIC()` attempts to find a set of hinge positions q which constitute a static configuration for the system, that is, one in which all the accelerations are zero when the velocities are as supplied in the state vector. `SDSTATIC()` uses a nonlinear root finder to drive all the hinge accelerations to zero. Note that `SDINIT()` must be called *after* all System Description File "?" parameters have been set but *before* any analysis is performed. This means that any future changes to a mass, inertia, length, i.e., any question mark parameter in the System Description File *must* be followed by a call to `SDINIT()` before another analysis is performed. (For details on ordering requirements of SD/FAST-generated routines, see Reference Section R3.1 or the diagram in Quick Reference Section Q3.)

The variables passed to `SDSTATIC()` are the current time `T`, the initial guess at a static state vector `Y(NQ+NU)`, a vector `LOCK(NU)` [8] whose nonzero elements indicate to `SDSTATIC()` *not* to vary that coordinate during the analysis (useful for limiting which coordinates can change from your initial setting and for specifying known values such as prescribed motion coordinates to speed up the analysis), the allowable tolerance on maintaining constraints `CTOL` which is set to 0 since this problem has no constraints, the desired tolerance of the solution for all accelerations `TOL` which is set to 1.0D-4, and the maximum number of function evaluations `MAXEVALS` allowed by the root finder, which is set to 1000. Passed back is the state vector `Y` in a static configuration (if one is found), the actual number of derivative evaluations used `FCNT`, and an error variable `ERR` which returns 0 if successful, 1 if search failed, 2 if exceeded `MAXEVALS`.

The result of running this segment of code is:

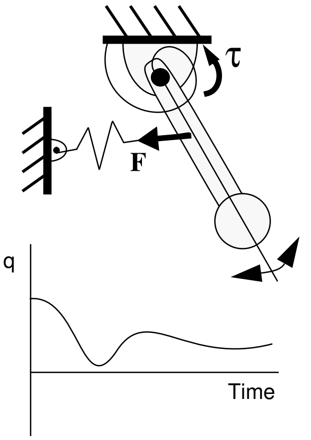
```
 Static equilibrium for mass = 10 kg:
  Y =    0.11692117535719   FCNT =   6   ERR =   0
```

Thus the static equilibrium point is computed to be 0.1169 radians, the number of function calls was only 6 out of a maximum allowable of 1000, and the error variable indicates a successful analysis.

---

8. `LOCK` is dimensioned `NU` rather than `NQ` because the four Euler parameters used to represent ball joint orientations are not all independent. See Reference Section R18.1 (Assembly Analysis) for a discussion.

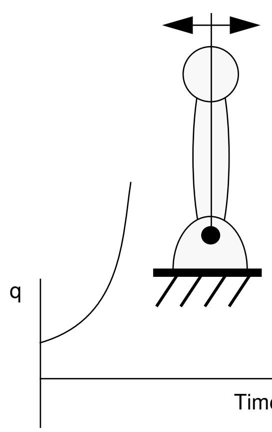
T2.6.3 Exercise 2: Perform Dynamic Analysis to Confirm Static Result



Perturb Stable Equilibrium

Since the static equilibrium analysis uses only a first-order root finding technique, stable versus unstable equilibria can not be determined (for example a pendulum just balanced to point straight up is in equilibrium, but at an unstable equilibrium point). To determine if indeed the equilibrium point is stable, a small perturbation can be introduced (to “knock” the pendulum over if it’s actually at an unstable equilibrium point) and motion analysis can be run for a while (say 10 steps at 0.05 seconds per step):

```
c  Perturb solution slightly and run a motion analysis at static configuration
c  to verify no unstable motion occurs.
c  Note that FLAG = 1 must be set each time SDMOTION is first called or
c  the state is changed discontinuously, such as after an impulsive load.
        FLAG = 1
        NSTEP = 10
        Y(1) = Y(1) + 0.0001D0
        WRITE (6,*)
        WRITE (6,*) 'Check subsequent motion to verify static condition:'
        WRITE (6,*) '      Time (sec)             Angle(rad)           Rate(rad/sec)'
        DO 10 ILOOP = 1, NSTEP
          CALL SDMOTION(T,Y,DY,DT,CTOL,TOL,FLAG,ERR)
          WRITE (6,1000) T,Y
1000      FORMAT (1X,3(D18.10,5X))
10      CONTINUE
```



Perturb Unstable Equilibrium

Note that a call to `SDINIT()` is not needed since no system parameters were changed since the last time `SDINIT()` was called. The variables passed to `SDMOTION()`, the equations of motion propagation routine are: the current time `T`, the state vector `Y(NQ+NU)`, the state derivative vector `DY(NQ+NU)` which currently has *incorrect* information since we perturbed the state, the time increment `DT` which was previously set to 0.05 sec, the allowable tolerance on maintaining constraints `CTOL` which is set to 0 since this problem has no constraints, the desired tolerance of the solution for all accelerations `TOL` which is set to 1.0d-4, the initial call `FLAG`[9] which *must* be set to `1` each time `SDMOTION()` is first called or `Y` has been changed by the user (such as after an impulsive load has been analytically computed and imposed), and `ERR` which reports 0 for success, 1 as a warning that a discontinuity in the derivatives was encountered, 2 the system has locked up and integration can’t proceed, or 3 constraints were violated and integration can’t proceed.

The results of this code segment show that, indeed, the pendulum returns to the 0.1169 radians solution (from the perturbed value of 0.1170) as the motion analysis proceeds:

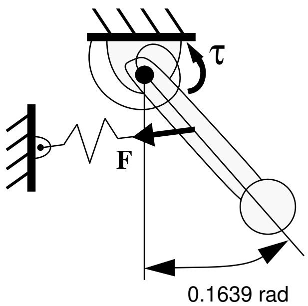
```
Check subsequent motion to verify static condition:
      Time (sec)             Angle(rad)           Rate(rad/sec)
   0.5000000000D-01       0.1170200221D+00      -0.4590198193D-04
   0.1000000000D+00       0.1170166166D+00      -0.8992137436D-04
   0.1500000000D+00       0.1170110777D+00      -0.1310798354D-03
   0.2000000000D+00       0.1170035713D+00      -0.1684824388D-03
   0.2500000000D+00       0.1169943053D+00      -0.2013367501D-03
   0.3000000000D+00       0.1169835247D+00      -0.2289692025D-03
   0.3500000000D+00       0.1169715046D+00      -0.2508384447D-03
   0.4000000000D+00       0.1169585437D+00      -0.2665454073D-03
   0.4500000000D+00       0.1169449570D+00      -0.2758399171D-03
   0.5000000000D+00       0.1169310683D+00      -0.2786237675D-03
```

9. The purpose of the flag `FLAG` is to indicated that initial derivatives need to be generated.

T2.6.4 Exercise 3: Change the Mass of the Pendulum and Redo Static Analysis

Next we run the static analysis again, this time for a pendulum mass of 5 kg:

```
c  Redo static analysis for mass set to 5 kg, bias unchanged.
c  Note that all velocities, Y(2), must be set to zero for SDSTATIC() to work
       CALL SDMASS(1,5.0D0)
       CALL SDINIT
       Y(2) = 0.0D0
       CALL SDSTATIC(T,Y,LOCK,CTOL,TOL,MAXEVALS,FCNT,ERR)
       WRITE (6,*)
       WRITE (6,*) 'Static equilibrium for mass = 5 kg: '
       WRITE (6,*) ' Y = ',Y(1),'   FCNT = ',FCNT,'   ERR = ',ERR
```



Static Equilibrium for 5 kg Pendulum

The mass was changed by calling `SDMASS(1,5.0D0)` where `1` indicates the body number of the pendulum and `5.0` is the new mass. The mass can only be changed if the input file initially had a "`?`" attached to the value of the mass. If we had attempted to change the mass or any other parameter that was not "changeable," the analyses would proceed (incorrectly), but the error handling routine `SDPRINTERR()` would report the error when called. Note that `SDINIT()` had to be called after the change in the mass of body 1 was made. Also note that the pendulum velocity had to be set exactly to zero since `SDSTATIC()` will try to solve using the given velocities. The result of running this code segment is:

```
Static equilibrium for mass = 5 kg:
 Y =    0.16392023725289   FCNT =    6   ERR =    0
```

where the static equilibrium point is computed to be 0.1639 radians, the number of function calls was only 6 out of a maximum allowable of 1000, and the error variable indicates a successful analysis. We shall proceed using a 5 kg pendulum mass for the remaining analyses.

T2.6.5 Exercise 4: Release Pendulum and Check That F=ma Holds

Next we shall instantaneously reset the rotational spring set-point bias from 0.3 radians to 0 radians and check Newton's Second Law $(\Sigma \mathbf{F} = m\mathbf{a})$ for the pendulum just at the start of its subsequent motion. Here we shall show more uses of the SD/FAST-generated routines to extract useful information from the simulation.

Since we instantaneously changed a torque acting on the system by changing the offset bias of the rotational spring, the system derivatives in `DY` are now invalid. However, we can re-evaluate the equations of motion just at the instant the torque was changed by setting the time increment `DT` to zero. (Normally we should set the "derivatives invalid" flag `FLAG`, but for `DT`=0 this is unnecessary.[10]):

```
c  Set spring bias to zero to allow the pendulum to fall.
c  At the instant of release, check that F = ma for all three axes.
c  Note that setting DT = 0 causes the derivatives to be evaluated, but not propagated.
       BIAS = 0.0D0
       DT = 0.0D0
       CALL SDMOTION(T,Y,DY,DT,CTOL,TOL,FLAG,ERR)
```

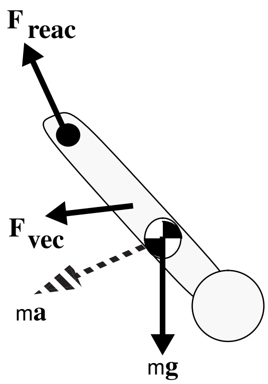
10. When in doubt, go ahead and set the `FLAG`. It won't ever hurt except for some additional computations.

Next we call `SDREAC(FORCE,TORQUE)` which passes back all the current reaction forces `FORCE(NJNT,3)` and torques `TORQUE(NJNT,3)` acting *on* all the bodies at their hinge connections:

```
c  Get reaction forces and torques acting on the pendulum
        CALL SDREAC(FORCE,TORQUE)
        WRITE (6,*)
        WRITE (6,*) 'Force = '
        WRITE (6,1000) FORCE
        WRITE (6,*) 'Torque= '
        WRITE (6,1000) TORQUE
```

with the results reported in the *body* frame:

```
   Force =
     -0.4236262476D+01      0.4809531435D+02      0.0000000000D+00
   Torque=
      0.0000000000D+00      0.0000000000D+00      0.0000000000D+00
```



Check
$\Sigma\mathbf{F} = m\mathbf{a}$

Then we transform the reaction force from the body frame to the inertial frame using `SDTRANS()`; ask for the acceleration `ACC` of the center of mass of the pendulum using `SDACC(1,COMASS,ACC)`, which is already given in the inertial frame; transform the external force `FVEC` from the linear spring to the inertial frame; ask SD/FAST for its current values of the mass of the pendulum and gravity; and compute both sides of the vector equation $(\Sigma\mathbf{F} = m\mathbf{a})$:[11]

```
 c  Transform the force vector from the pendulum frame to the inertial frame
        CALL SDTRANS(1,FORCE,0,FORCE)
 c  Find the acceleration of the center of mass of the pendulum
        CALL SDACC(1,COMASS,ACC)
 c  Transform the external force vector from SDUFORCE to inertial frame
        CALL SDTRANS(1,FVEC,0,FVEC)
 c  Get the current values of the pendulum mass and gravity for F = ma eqn.
        CALL SDGETMASS(1,M1)
        CALL SDGETGRAV(GRAV)
        WRITE (6,*)
        WRITE (6,*) 'Set torsion spring bias to zero'
        WRITE (6,*) 'Compute external forces, accelerations, and position'
        WRITE (6,*) 'at the time of release:'
        WRITE (6,*) 'Force = '
        WRITE (6,1000) FORCE
        WRITE (6,*) 'Acceleration = '
        WRITE (6,1000) ACC
        WRITE (6,*)
        WRITE (6,*) 'Check if sum of external forces equals mass*acceleration:'
        WRITE (6,*) '(along three inertial unit vectors, n1, n2, and n3)'
        WRITE (6,*) '                F       =     m      a      ?'
        WRITE (6,*) ' n1: ',FORCE(1,1)+M1*GRAV(1)+FVEC(1), ' = ',M1*ACC(1),' ?'
        WRITE (6,*) ' n2: ',FORCE(1,2)+M1*GRAV(2)+FVEC(2), ' = ',M1*ACC(2),' ?'
        WRITE (6,*) ' n3: ',FORCE(1,3)+M1*GRAV(3)+FVEC(3), ' = ',M1*ACC(3),' ?'
```

11. It is a good idea to get into the habit of asking SD/FAST directly for any model parameters, since you may have changed them in other routines or by terminal i/o. There's no point in entering the same numerical values twice and inviting future errors when changes have to be made.

with the expected results:

```
Set torsion spring bias to zero
Compute external forces, accelerations, and position
at the time of release:
Force =
  -0.1202801263D+02       0.4675929955D+02       0.0000000000D+00
Acceleration =
  -0.2732109596D+01      -0.4519028418D+00       0.0000000000D+00

Check if sum of external forces equals mass*acceleration:
(along three inertial unit vectors, n1, n2, and n3)
                F        =      m       a       ?
 n1:    -13.660547979872 =    -13.660547979872 ?
 n2:    -2.2595142092015 =    -2.2595142092015 ?
 n3:   0. =   0. ?
```

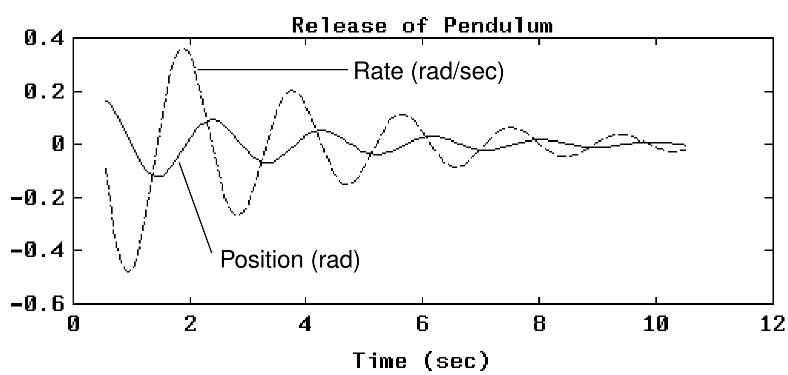
T2.6.6 Exercise 5: Motion Analysis of Pendulum After Release

To propagate subsequent motion of the pendulum, we simply set DT = 0.05 and allow to run from the current 0.50 seconds to 10.50 seconds (200 steps):

```
c  Set DT = 0.05 to allow motion to propagate for a while
        WRITE (6,*)
        WRITE (6,*) 'Motion After Release:'
        WRITE (6,*) '      Time (sec)             Angle(rad)           Rate(rad/sec)'
        DT = 0.05D0
        NSTEP = 200
        DO 20 ILOOP = 1, NSTEP
          CALL SDMOTION(T,Y,DY,DT,CTOL,TOL,FLAG,ERR)
          WRITE (6,1000) T,Y
20      CONTINUE
```

Rather than print all the output, we have plotted the results in Figure T2-5. (Note that the first point plotted is at `T`=.55.)

Figure T2-5 Time History of Position and Velocity After Release of the Pendulum



T2.6.7 Exercise 6: Static Analysis of Final Resting Position

Finally, we can check the final position of the pendulum after all motion has stopped by performing another static analysis. Note that the velocity, `Y(2)`, had to be set to zero or the static analysis would have attempted to find a solution with the last velocity computed by the motion analysis:

```
c  Redo static analysis for spring bias set to 0.
c  Long term motion of damped pendulum should lead to hanging vertically.
c  Note that all velocities, Y(2), must be set to zero.
      Y(2) = 0.0D0
      CALL SDSTATIC(T,Y,LOCK,CTOL,TOL,MAXEVALS,FCNT,ERR)
      WRITE (6,*)
      WRITE (6,*) 'Static equilibrium for bias = 0: '
      WRITE (6,*) ' Y = ',Y(1),'   FCNT = ',FCNT,'      ERR = ',ERR

c  ALWAYS include a call to SDPRINTERR() at the end of your program
c  to report any problems found by SD/FAST.
      CALL SDPRINTERR(6)
      STOP
      END
```

with the expected result of the pendulum hanging vertically:

```
Static equilibrium for bias = 0:
 Y =     4.5445817738361D-09   FCNT =   4      ERR =   0
```

Summary

- This tutorial covered a complete example of using SD/FAST from start to finish, including (1) develop the model, (2) write system description file, (3) run SD/FAST, and (4) incorporate SD/FAST-generated routines into a stand-alone simulation and execute.
- The key idea in developing the system model for use by SD/FAST is to extract the required information in the most advantageous manner. That is, select a reference configuration to minimize complex geometries and mass properties. Selection of a reference configuration with simple geometry and mass properties also has the side benefit of allowing the SD/FAST symbolic manipulator reduce the computational load.
- Once the model is developed, the input System Description File is easily written by simply picking off the model parameters from the sketches.
- Running SD/FAST on the input file generates up to four files: (1) a "`_dyn.f`" file containing most system-specific routines; (2) a "`_sar.f`" file containing the Simplified Analysis Routines; (3) an "`_info`" file containing the system roadmap, state index, and system parameters; and (4) a library file, "`sdlib.f`", if not previously generated, containing the General Analysis Routines and some utilities.
- You then write a main driver program or write the required "hooks" in your simulation environment that call the SD/FAST-generated routines.
- The analyses demonstrated in this tutorial included: (1) static analyses to determine system equilibrium configurations (`SDSTATIC()`); (2) dynamic analyses to compute time histories of system motion (`SDMOTION()`); and (3) extraction of system information before and after instantaneous changes in system state or model parameters (`SDMOTION(DT=0)`).
- Several generated routines were introduced including: (1) application of hinge and body loads (`SDHINGET()`, `SDPOINTF()`); (2) transforming vectors between coordinate frames (`SDTRANS()`); (3) changing model parameters (`SDMASS()`); (4) obtaining information about the system (`SDPOS()`, `SDACC()`, `SDGETMASS()`, `SDGETGRAV()`, `SDREAC()`); and (5) checking for errors (`SDPRINTERR()`).

TUTORIAL 3

Slewing Spacecraft

Objectives

- Develop a complex open-loop topological model.
- Use prescribed motion.
- Approximate flexible-body dynamics using lumped mass-springs.
- Use a control system in a simulation.
- Integrate user states using General Analysis Routines.
- Perform a design study by changing model parameters.

T3.1 A Complex Spacecraft Example

This tutorial will take you through the modeling and simulation of a realistic spacecraft with prescribed motion, simple flexible-body dynamics, a control system with sensors and actuators, and studies of proposed changes in the design of a camera on the spacecraft. The spacecraft model is based upon an early three-axis (non-spinning) interplanetary exploration spacecraft built by the Jet Propulsion Laboratory[1]. The necessary computer files for this example can be found by looking in the SD/FAST installation directory for all the files starting with `slew.`

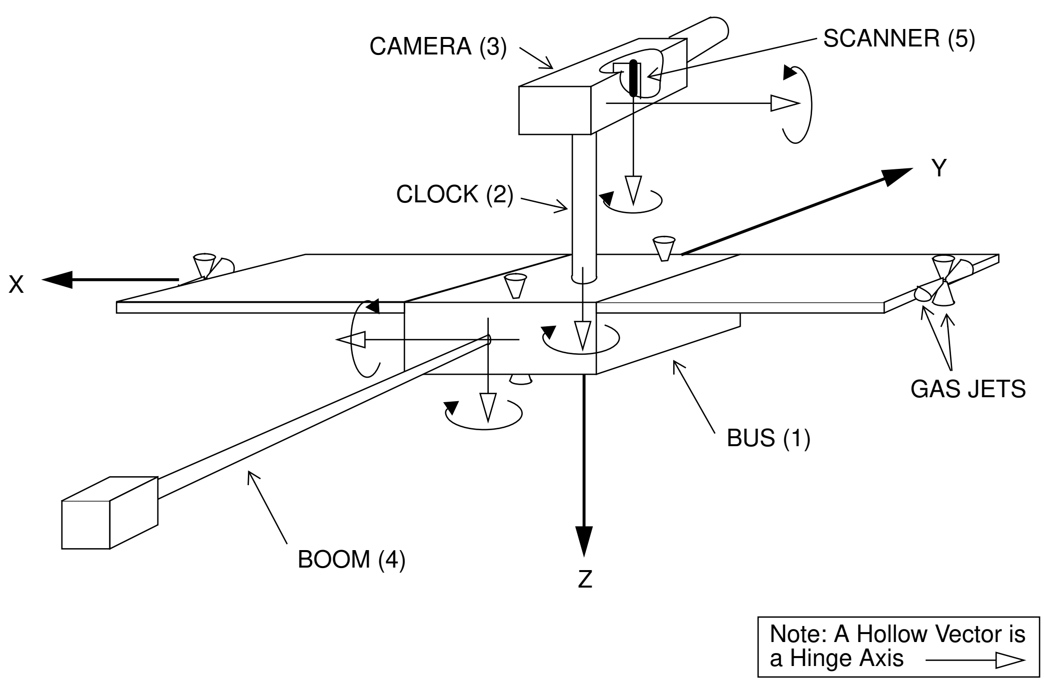
This section covers the rigid-body model of the spacecraft, the SD/FAST System Description File, and the generated roadmaps.

1. G. E. Fleischer and P. W. Likins, "Attitude Dynamics Simulation Subroutines for Systems of Hinge-Connected Rigid Bodies", NASA JPL Technical Report 32-1592, pp. 13-36.

T3.1.1 Spacecraft Model

The model to be used for the spacecraft is pictured in its reference configuration in Figure T3-1. The model chosen has five bodies and a total of eleven degrees of freedom (assuming that prescribed motion is not enforced). The bus and its attached solar panels are modeled as a single rigid body which is the base body for this system. The camera sits on a shaft called the “clock” which is attached to the bus by a pin joint. The camera is hinged to the clock by another pin joint orthogonal to the clock-bus pin joint. Inside the camera is a small high-speed scanner. The scanner oscillates under control of a very high band-width controller whose dynamics are unaffected by the relatively slow motions of the camera and the spacecraft attitude dynamics. Thus its motions can be fully prescribed as discussed in Section T3.3.

Figure T3-1 Reference Configuration of the Five-Body Spacecraft Model



There is a flexible boom attached to the bus which is modeled as a rigid body connected to the bus by a U-joint. The selection of appropriate hinge torques to model the fundamental frequency of the actual flexible boom is discussed in Section T3.4.

The spacecraft is equipped with a rate-integrating gyro that reports bus attitude and bus attitude rates about its three orthogonal body-fixed axes. There are gas jet thrusters on the bus which are pulsed (with a minimum on-time) to exert restoring torques if the bus attitude or attitude rates exceed acceptable tolerances as the slew maneuver of the camera progresses. Initially all bus attitude and attitude rates are zero and the attitude control system attempts to keep them close to zero. Motors at the clock and camera hinges are slewed by applying varying amounts of torque. A controller model is provided which drives these motors. All sensors, actuators, and controller models are described in Section T3.5.

T3.1.2 The SD/FAST System Description File

The SD/FAST System Description File for this spacecraft model using the mass properties of the original JPL model (except the scanner which is new in this model) is in `slew.sd` in the installation directory and is reproduced in Figure T3-2.

Figure T3-2 SD/FAST System Description File for the Spacecraft Model

```
body = bus
  mass = 410
  inertia = 115   -14    14
             ?    316   -34.6
             ?     ?    440

body = clock  inb = bus  joint = pin
  mass = 6.8
  inertia = 0.35  0.35  0
  bodyToJoint =  0  0  -0.75
  inbToJoint  =  0  0  -1.5
  pin = 0 0 1

body = camera inb = clock joint = pin
  mass = 57.5
  inertia = 4.85   0.41  -0.07
              ?    2.2    0.54
              ?     ?     5.5
  bodyToJoint = 0  -0.22  0.2
  inbToJoint  = 0  -0.1  -0.75
  pin = -1 0 0

body = boom inb = bus joint = ujoint
  mass = 10.7
  inertia = 27.2  0.2  27.2
  bodyToJoint = 0   3.3  0
  inbToJoint  = 0  -1.2  0
  pin =  0 0 1
  pin =  1 0 0

body = scanner inb = camera joint = pin   prescribed
  mass = 1.0
  inertia = 0.1 0.1 0.5
  bodyToJoint = 0 0.01 0
  inbToJoint  = 0 0? 0
  pin =  0 0 1
```

This System Description File should be completely understandable at this point with three exceptions: (1) The three question marks used in the lower off-diagonal terms of the inertia matrices are simply an easy way to enforce the symmetry of the matrix without possibility of incorrectly typing the symmetric entry, (2) the keyword `prescribed` on the fifth body (scanner) tells SD/FAST that the motion of the pin joint connecting the scanner to the camera will be prescribed as discussed in Section T3.3, and (3) the combined number 0 and the "?" entered as the second component of the `inbToJoint` vector for scanner indicates to SD/FAST that we will be changing this parameter during the simulation and thus leave it in symbolic form rather than reducing it numerically, and that 0 is the default value placed in the variable. Study the mass properties and geometry to convince yourself that this is a plausible model of the system pictured in Figure T3-1.

T3.1.3 Generated Roadmaps

Once the input file has been processed by SD/FAST (use the command: `sdfast -ge slew.sd` to get all the files), we can look at the `slew_info` file and examine the roadmap for a more complex system:

```
SD/FAST Information File: slew.sd
Generated  5-Jan-1991 19:04:31 by SD/Fast, Kane's formulation
(sdfast BX.1.2 #10643) on machine ID 1700e754

ROADMAP (slew.sd)

Bodies        Inb
No  Name      body Joint type  Coords q         Multipliers
--- --------- ---- ----------- ---------------- -----------------------
  0 $ground                                     |
  1 bus          0  6dof(Trans)   1   2   3     |
                    6dof(Rot)     4   5   6  12 |
  2 clock        1  Pin(1D)       7             |
  3 camera       2  Pin(1D)       8             |
  4 boom         1  U-joint(2D)   9  10         |
  5 scanner      3  Pin(1D)      11p            |  1p
```

There are five bodies plus ground. We shall use the body numbers assigned to integer variables (`SCANNER=5` being the scanner, etc.) to reference these bodies in the driver code (this is recommended to help keep the code more readable for complex systems). The inboard body reference numbers all indicate the topology desired. The joint types are all correct. The assigned coordinates indicate that there are indeed eleven degrees of freedom, with the scanner being prescribed as indicated by the `11p`. One multiplier is needed to adjoin the prescribed motion constraint to the equations of motion. We are not concerned with the multiplier in this problem, but if you want more information, see Constraint Multipliers on page R-22.

The State Index To Joint/Axis Map shows the location in the state vector of the position coordinate q and the rate coordinate u for each joint axis, the joint type and the axis type for each joint. Note that there is not a matched number of q's and u's for the 6dof joint used to track the `bus` attitude. There are four quaternions and only three angular velocity measure numbers. The use of four parameters in quaternions guarantees that no singularity will be encountered in the attitude equations (See Euler Parameters on page R-27 for details). You can either use this map to pick out what you need from the state vector, or use the `SDINDX()` routine to avoid explicit dependencies on the assignment of coordinates to place in the state array. We have taken the latter approach in this example, and recommend it highly.

```
STATE INDEX TO JOINT/AXIS MAP (slew.sd)

 Index
  q|u   Joint  Axis   Joint type    Axis type
 -----  -----  ----   -----------   ----------
  1|13      1     1   6dof          translate
  2|14      .     2   .             translate
  3|15      .     3   .             translate
  4|16      .     4   .             quaternion
  5|17      .     5   .             quaternion
  6|18      .     6   .             quaternion
  7|19      2     1   Pin(1D)       rotate
  8|20      3     1   Pin(1D)       rotate
  9|21      4     1   U-joint(2D)   rotate
 10|22      .     2   .             rotate
 11|23      5    1p   Pin(1D)       rotate
 12|        1     7   6dof          4th quat
```

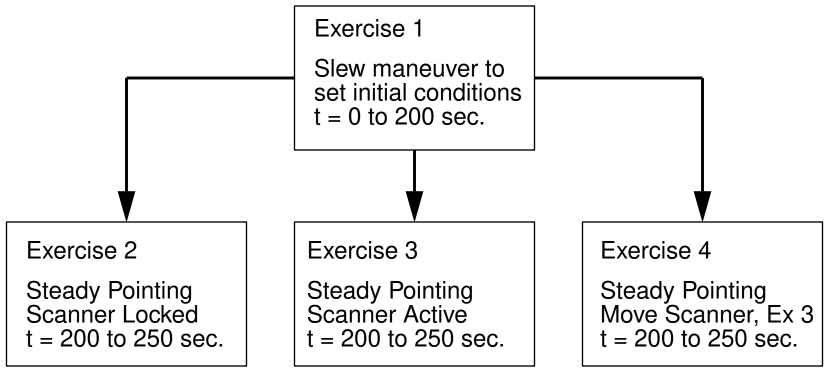
The System Parameters list summarizes the numbers of various components in the system. These numbers are useful for dimensioning certain arrays in your routines. For the spacecraft there are 5 bodies, 5 joints, 11 degrees of freedom including any prescribed axes, no loop joints and loop joint degrees of freedom, 12 position coordinates versus 11 rate coordinates (one extra quaternion), no loop position or rate coordinates, and only one constraint (which comes from the prescribed motion).

```
SYSTEM PARAMETERS (slew.sd)

Parameter  Value  Description

nbod           5  no. bodies (also, no. of tree joints)
njnt           5  total number of joints (tree+loop)
ndof          11  no. degrees of freedom allowed by tree joints
nloop          0  no. loop joints
nldof          0  no. degrees of freedom allowed by loop joints

nq            12  no. position coordinates in state (tree joints)
nu            11  no. rate coordinates in state (tree joints)
nlq            0  no. position coordinates describing loop joints
nlu            0  no. rate coordinates describing loop joints

nc             1  total no. constraints defined
nlc            0  no. loop joint constraints
npresc         1  no. prescribed motion constraints
nuserc         0  no. user constraints
```

T3.2 Analyses of Slew Maneuver

Rather than first discussing the details of modeling prescribed motion, controllers, etc., we shall show the results of the simulations to motivate the modeling. Four simulations are developed for the spacecraft: (1) Initial slew maneuver to set up initial conditions for the rest of the exercises, (2) steady pointing of the camera with its scanner turned off and locked, (3) steady pointing of the camera with its scanner turned on, and (4) a design study which repeats run #3, but with the scanner located at a different location in the camera body. In all cases we are examining the spacecraft pointing errors. Figure T3-3 shows the four simulations:

Figure T3-3 Spacecraft Simulations



First the main program set-up code is listed:

```
c  Spacecraft simulation example.  This is the example from
c  Tutorial 3 of the SD/FAST manual.  Time histories for each of
c  the four analyses go into fort.11, fort.12, fort.13, and
c  fort.14, resp.

c  NQ = # of q's (generalized coordinates)
c  NU = # of u's (generalized speeds)
c  NEQ = # of equations to integrate (q's, u's and command rates)
c  AZCMD, ELCMD = location of command values in system state array

      integer NQ,NU,NEQ,AZCMD,ELCMD
      parameter (NQ=12,NU=11,AZCMD=NQ+NU+1,ELCMD=AZCMD+1,NEQ=NQ+NU+2)

c  The body numbers.  Note that these are also the numbers for
c  these bodies' inboard joints.

      integer GROUND,BUS,CLOCK,BOOM,CAMERA,SCANNER
      parameter (GROUND=0,BUS=1,CLOCK=2,CAMERA=3,BOOM=4,SCANNER=5)

      real*8 y(NEQ),t,dt,tol,camscan(3)
      real*8 savey(NEQ),tscan,slwstrt,slwstp,azrate,elrate,scanrt
      integer nout,nbtw,i,scan,SDINDX
      external deriv

      common /scanner/tscan,scanrt,scan
      common /slewparm/slwstrt,slwstp,azrate,elrate

c  The slew is set up to begin at the camera current position, to
c  slew for 10s at the indicated rates, after waiting for 1s.
      slwstrt = 1d0
      slwstp  = 11d0
      azrate  = -.025d0
      elrate  = .01d0
```

Setup Slew Parameters

```
c  During integration, report any state or constraint
c  errors gt tol.
       tol = 1d-3

c  When the scanner is enabled, it scans at this rate (rad/s).
       scanrt = 1d0
```

Initialize SD Routines

```
c  We'll start out using the default parameters for the system, as
c  provided in the System Description File.
       call SDINIT
```

T3.2.1 Exercise 1: Initial Slew Maneuver with Scanner Off

First we perform the standard camera slew maneuver with the scanner locked:

```
c===================================================================
c  ANALYSIS #1: simulate the slew maneuver with the scanner locked.
c===================================================================

c  Set up initial conditions.  Start with everything at 0 (reference
c  configuration).  We'll fill in the base body orientation coordinates
c  as though they were 1-2-3 Euler angles, rather than quaternions.
c  (Then the reference configuration is 0,0,0.)  Then we can use
c  SDANG2ST to convert to quaternions.
```

Start Body at 0,0,0 Euler Angles

```
      t = 0d0
      do 10 i=1,NEQ
10       y(i) = 0d0
      call SDANG2ST(y,y)

c  Command the camera's initial azimuth and elevation, and set the
c  initial conditions for these angles to the commanded values.
```

Setup Camera Command

```
      y(AZCMD) = 4d0
      y(ELCMD) = -0.5d0
      y(SDINDX(CLOCK,1))  = y(AZCMD)
      y(SDINDX(CAMERA,1)) = y(ELCMD)
```

Lock Scanner

```
c  Lock the scanner.
      scan = 0

c  We run long enough to let the system settle down after the slew.
```

Run Initial Slew Simulation

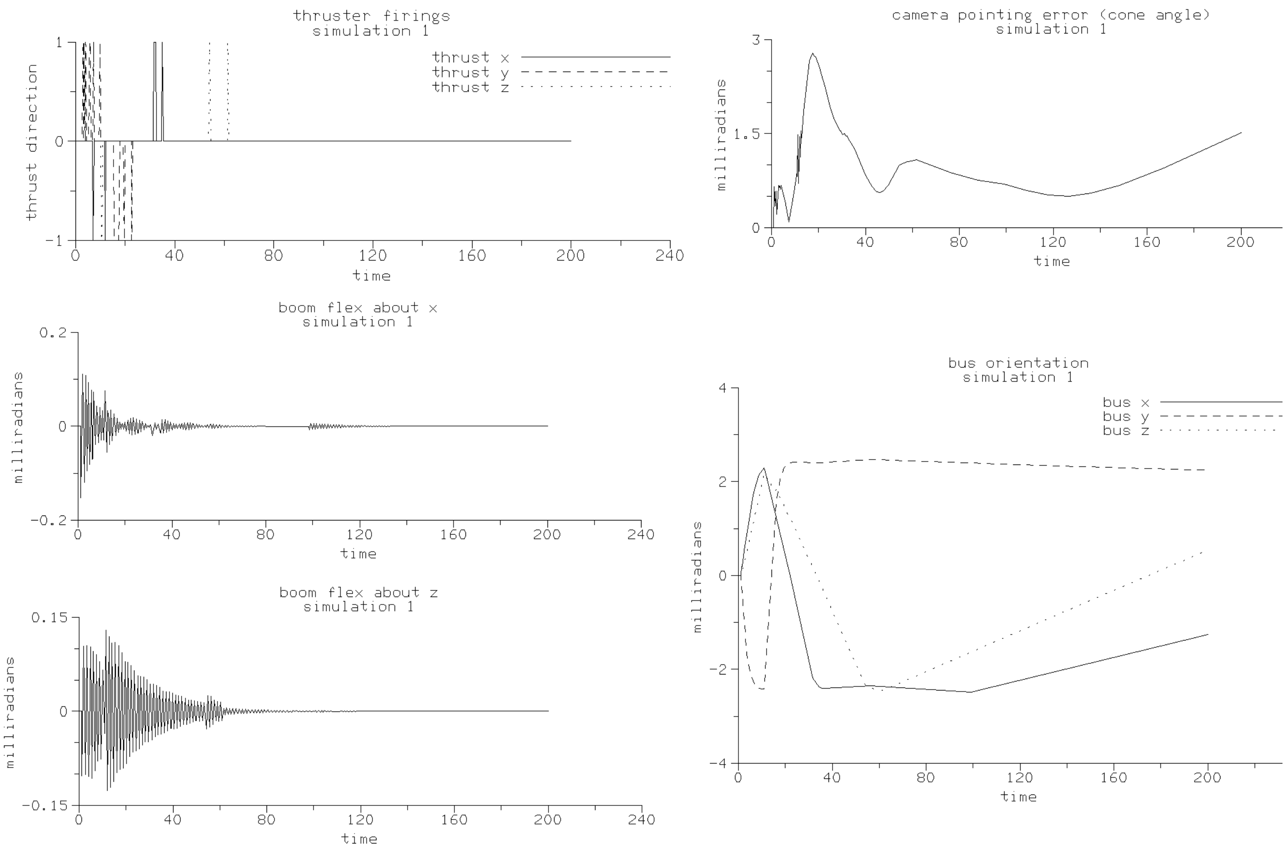
```
      dt = .05
      nout = 1000
      nbtw = 4
      call simulate(1,nout,nbtw,dt,tol,scan,t,y)

c  Save the state at the end of the above simulation, so we can run
c  several analyses each beginning at this state.
```

Save End State To Start Other Simulations

```
      do 40 i=1,NEQ
40       savey(i) = y(i)
      tscan = t
```

Rather than printing the numerical output, the following plots shows the time history of the simulation results. Note that the maximum pointing error is almost 3 milliradians during the slew with the scanner off. The thrusters fire early in the slew maneuver, and the flexible boom is excited. The spacecraft bus attitude is kept within the 2.5 milliradian deadband by the thruster controller.

Figure T3-4 Exercise 1 Results



T3.2.2 Exercise 2: Base Simulation: Steady Pointing with Scanner Off

Next we simply allow the steady pointing to continue with the scanner off.

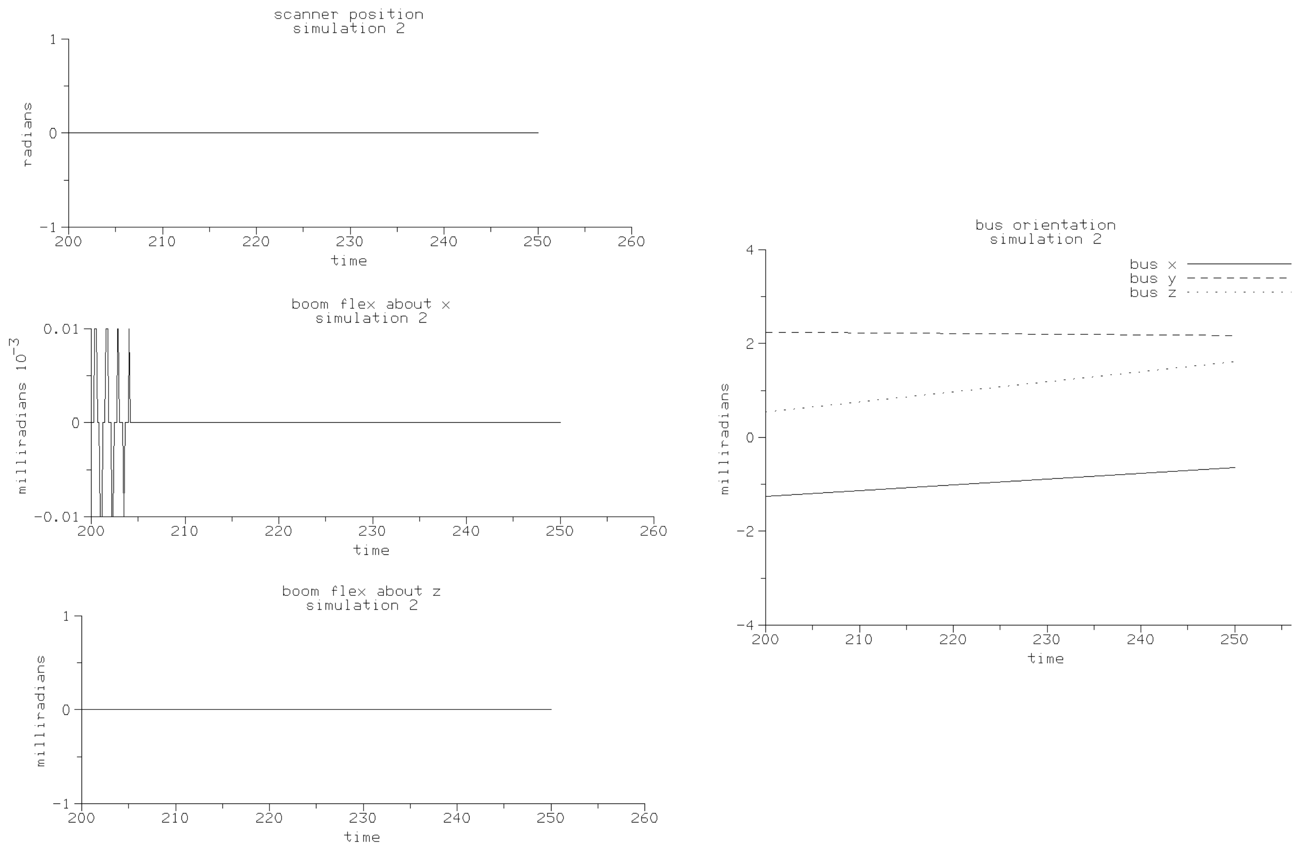
```
c=====================================================================
c  ANALYSIS #2: continue run with scanner locked as a control study.
c=====================================================================
c After finishing the slew, the camera is supposed to remain still in
c inertial space, with its bore sight at AZCMD and ELCMD in the
c ‘celestial sphere’.  Now as a control, we’ll observe the pointing
c accuracy with the scanner off, continuing the above simulation.
```

Continue Run as a Control Study

```
     nout = 1000
     nbtw = 1
     call simulate(2,nout,nbtw,dt,tol,scan,t,y)
```

Plots of the simulation show that the flex boom finally damps out, and the bus attitude is kept within the desired 2.5 milliradian deadband.

Figure T3-5 Exercise 2 Results



T3.2.3 Exercise 3: Steady Pointing with Scanner Turned On

Next we restart the steady pointing of Exercise 2, but this time with the scanner on.

Turn Scanner On

```
c=====================================================================
c  ANALYSIS #3: simulate effect of scanner on camera pointing error.
c=====================================================================
c We will enable the scanner and observe its effect on camera pointing
c accuracy.  (Scanner motion is defined in subroutine motions() as a
c sinusoid beginning at tscan.)
        scan = 1
```

Reset Simulation to End of Sim 1

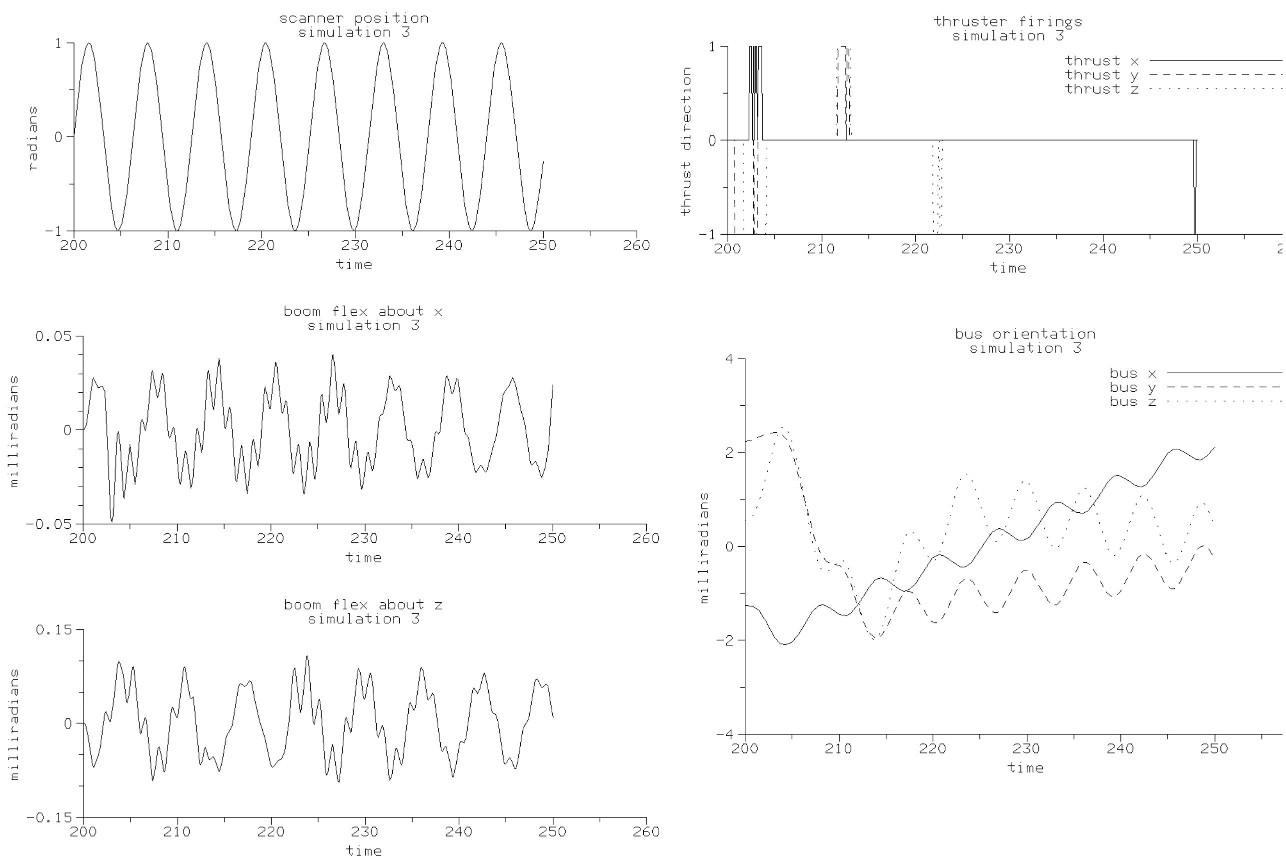
```
c  Put state back to the end of the first analysis.
        do 44 i=1, NEQ
 44        y(i) = savey(i)
        y(   SDINDX(SCANNER,1)) = 0d0
        y(NQ+SDINDX(SCANNER,1)) = scanrt
        t = tscan
```

Run with Scanner On

```
        call simulate(3,nout,nbtw,dt,tol,scan,t,y)
```

The simulation results show that the scanner causes a steady vibration in the flexible boom and some thruster firings.

Figure T3-6 Exercise 3 Results



T3.2.4 Exercise 4: Steady Pointing with Scanner In Different Location

A very powerful feature of SD/FAST is the ability to leave some of the model parameters as variables in the simulation so that the user can change the physical model without having to rerun SD/FAST, re-compile, and re-link. In the fourth exercise, we perform a design study by changing the location of the scanner hinge location in the camera by using the `SDITJ()` routine (to modify the Inboard To Joint vector), and rerun the third analysis. In this case, we want to move the scanner along the y-axis of the camera body.

```
c================================================================
c  ANALYSIS #4: repeat #3 but with different location for scanner.
c================================================================
```

Change Scanner Location. Note SDINIT was called again because we changed a system parameter

```
      write(6,*) 'Changing geometry.'
      camscan(2) = -.2d0
      call SDITJ(SCANNER,camscan)
      call SDINIT
```

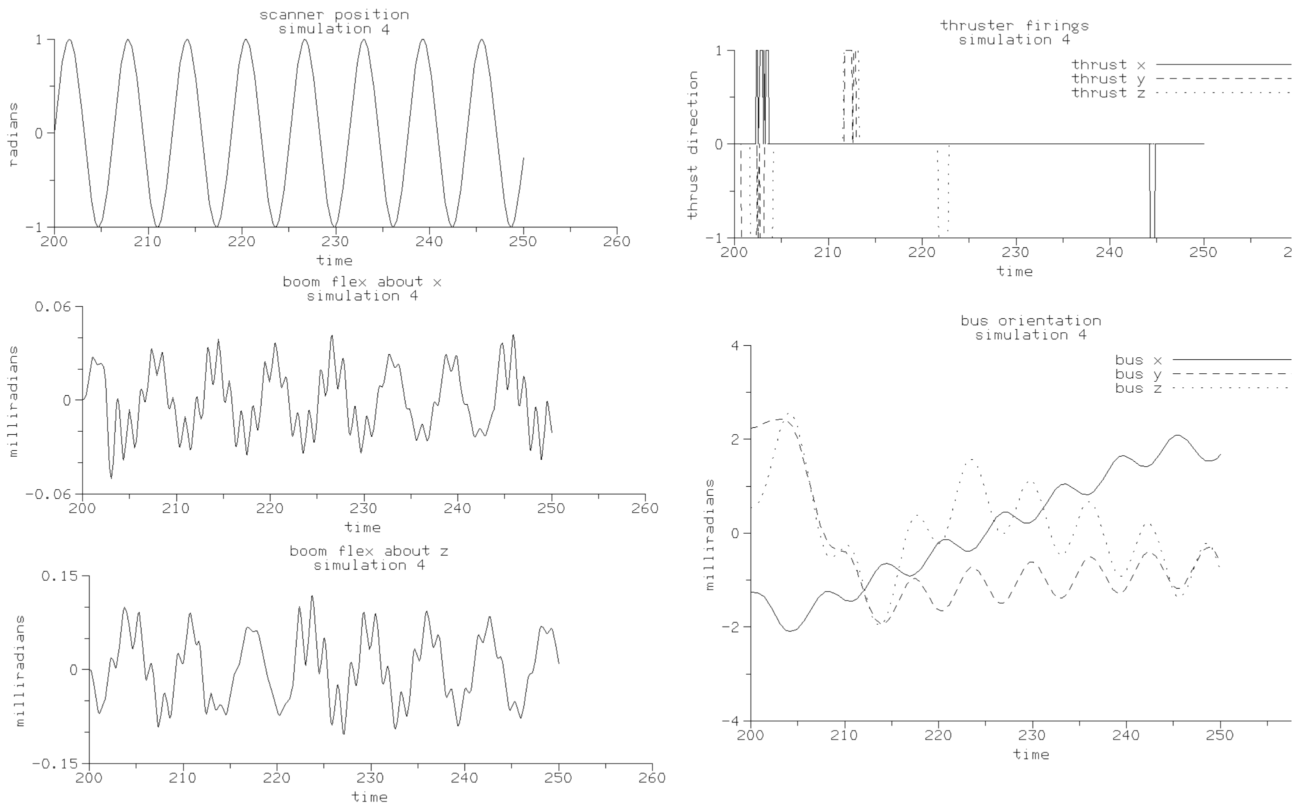
Reset Simulation to End of Sim 1

Run with Scanner in Different Location

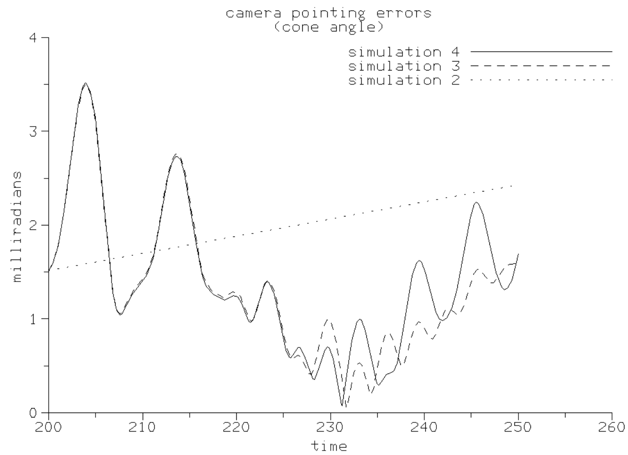
```
c  Put state back to the end of the first analysis.
         do 60 i=1, NEQ
 60         y(i) = savey(i)
         y(   SDINDX(SCANNER,1)) = 0d0
         y(NQ+SDINDX(SCANNER,1)) = scanrt
         t = tscan

         call simulate(4,nout,nbtw,dt,tol,scan,t,y)

         call SDPRINTERR(6)
         end
```

The simulation plots show qualitatively similar behavior to Exercise 3.

Figure T3-7 Exercise 4 Results



Finally, the camera pointing error is plotted for the last three simulations to show the relative performance of the scanner off, then on in two different locations.



In the rest of this Tutorial, we shall describe in detail how we modeled the prescribed motion of the scanner, the flexible boom, the spacecraft control system with sensors and actuators, and the use of Generalized Analysis Routines to integrate user states. Finally the `simulate()` routine that ties all these models together for easy use by the main program is discussed.

T3.3 Prescribed Motion

Prescribing the motion of a mechanism (or just part of a mechanism) simply means imposing a time history of position, velocity, and acceleration on one or more joint axes, regardless of the effects of any other forces or torques applied to the system. Prescribed motion is useful in primarily two cases: (1) driving a system with known motion to obtain the "inverse dynamics" solution, i.e., you want to determine the forces and torques required to cause a desired motion of the system, and (2) easily modeling a part of the system under high-bandwidth control compared to the rest of the system, e.g., a small scanning mirror driven at high frequency compared to the natural frequencies of the rest of the system[2]. We are using prescribed motion for the latter reason in this problem. Note that SD/FAST does not restrict mixing prescribed motion with natural motion, i.e., a simulation can contain both "forward dynamics" and "inverse dynamics." Other examples of using prescribed motion include modeling high-speed momentum wheels in a spacecraft, and driving input shafts to machines.

In summary, the motion of a joint axis can be either calculated by SD/FAST as a function of the current system state and the forces acting on the system, or the motion can be

2. Note, however, that using prescribed motion on a sub-assembly of the system makes the dynamic coupling one-way. For example, the motion of the high-speed scanner may affect the natural motion of the spacecraft to which it is attached, but no possible motion of the spacecraft can affect the scanner motion relative to the camera body.

prescribed, usually as a function of time but in general as a function of time and some system states. When the motion is prescribed, SD/FAST calculates the actuator force or torque which would be required at that axis to produce the desired motion. Prescribed motion is discussed in detail in the reference, Section R14, which also explains how to turn prescribed motion on and off at runtime.

T3.3.1 Analytic Model

Key Idea

For prescribed motion to work best, *position, velocity, and acceleration of the joint axis must all be specified and must be compatible*[3], i.e., the equation used to specify velocity must *exactly* be the derivative of the equation used to specify the position, and acceleration is exactly the derivative of velocity. Likewise, velocity must be the integral of acceleration and position must be the integral of velocity. The three prescribed motion routines used to specify the joint axis prescribed motion are `SDPRESPOS()` for position, `SDPRESVEL()` for velocity, and `SDPRESACC()` for acceleration. Generally, all specifications of prescribed motion should be grouped together in the user-written routine (commonly called `sdumotion()`), just as forces are grouped in `sduforce()`. See Section R15.3.2 for information on writing the `sdumotion()` routine.

For the spacecraft example, the scanner is either locked (prescribed to be zero) or scanning with a sinusoidal motion. The equations for prescribed sinusoidal position P, velocity V, and acceleration A are:

$$P = \sin(\omega(t - t_{scan})) \tag{T3.1}$$

$$V = \omega\cos(\omega(t - t_{scan})) \tag{T3.2}$$

$$A = -\omega^2\cos(\omega(t - t_{scan})) \tag{T3.3}$$

where t_{scan} is the time at which the scanner is turned on.

T3.3.2 `motions()` Routine

The user-written code to implement this prescribed motion for the spacecraft scanner is in the routine `motions()` in the file `slew.f`. Note that we would have named this routine `sdumotion()` if we wanted to use the Simplified Analysis Routines. However, in this case we are writing our own analysis driver so the name doesn't matter.

```
c  motions
c
c  Prescribe the scanner motion.  If 'scan' is zero, set the
c  motion to 0 (i.e., locked).  Otherwise, prescribe it to
c  follow a sinusoidal motion with frequency w=scanrt radians
c  per second.
```

3. Using some integration methods, specifying all prescribed motions is optional. However, we strongly recommend that all three are specified and are compatible to maintain stability of the integration process, and to detect errors. See Section R14.

```
subroutine motions(t,q,u)
integer NQ,NU,SCANNER
parameter (NQ=12, NU=11, SCANNER=5)
real*8 t,q(NQ),u(NU),w,tscan,scanrt
integer scan
common /scanner/tscan,scanrt,scan

w = scanrt
if (scan .eq. 0) then
  call SDPRESPOS(SCANNER,1,0d0)
  call SDPRESVEL(SCANNER,1,0d0)
  call SDPRESACC(SCANNER,1,0d0)
else
  call SDPRESPOS(SCANNER,1,      sin(w*(t-tscan)))
  call SDPRESVEL(SCANNER,1,    w*cos(w*(t-tscan)))
  call SDPRESACC(SCANNER,1, -w*w*sin(w*(t-tscan)))
end if

return
end
```

Prescribed Motion for Locked Scanner

Prescribed Motion for Moving Scanner

Notice that the information from `slew_info` was used to set integer variables `NQ`, `NU`, and `SCANNER` to the correct values for dimensioning the state vectors and joint numbers. Each call to `SDPRESPOS()`, `SDPRESVEL()`, and `SDPRESACC()` takes the joint number (`SCANNER=5`), the axis (1 in this case for a pin joint), and the prescribed values to be passed. `scan` is a flag defined elsewhere and used to switch the scanner on and off for different analyses. Real variable `tscan` is the time at which the scanner joint position is zero and `scanrt` is the user-defined scan rate of the scanner mirror.

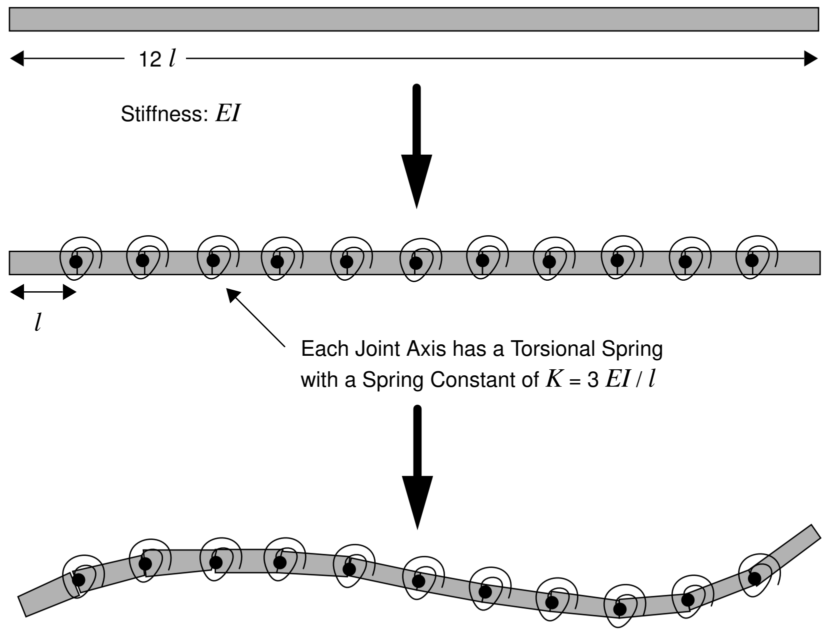
T3.4 Flexible Body Model

To model flexible bodies in SD/FAST, the body must be broken into rigid bodies connected by joints with spring (and possibly damper) torques at the joint axes. Correctly dividing a body and finding joint locations, and spring and damping constants can be very difficult, especially if you want to match several mode shapes and their frequencies. This area is usually reserved for the expert structural analyst.

T3.4.1 Simple Model for a Uniform Beam

In the case of bending in two orthogonal directions for a simple uniform beam, one possible model is to divide the beam into “n” equal length segments connected by U-joints. Using a simple static deflection model for the segment of length l with a stiffness EI (I may be different in the two orthogonal directions), the torsional spring constant is found to be $K = 3EI / l$. Note that even if you use 10 or more segments, only the first two or three mode shapes and frequencies will be even close to being correct. Damping is usually kept small to cover worst case vibrations. Figure T3-8 summarizes this model.

Figure T3-8 A Multi-Rigid-Body Model of a Uniform Flexible Beam



T3.4.2 Spacecraft Flexible Beam

For our case, only the first mode and its frequency are of interest. Thus a single beam element is sufficient. The spring constant and damping constant for a single U-joint between the beam and the bus were determined in a manner similar to above. The routine `boomflex` below implements this simple model and will be called by `forces()`.

Spring & Damping Model for Both Axes

```
c  boomflex
c
c  This routine models the first cantilever mode of the boom,
c  using empirically derived stiffness and damping constants.

      subroutine boomflex(q,u)
      real*8 q(*),u(*),k,b,torq
      integer BOOM,SDINDX
      parameter (BOOM=4)
      data k,b/2000d0,10d0/

      torq = - (k*q(SDINDX(BOOM,1)) + b*u(SDINDX(BOOM,1)))
      call SDHINGET(BOOM,1,torq)
      torq = - (k*q(SDINDX(BOOM,2)) + b*u(SDINDX(BOOM,2)))
      call SDHINGET(BOOM,2,torq)
      return
      end
```

Again, note that the integer variable `BOOM` is set to the body number 4 (same as the joint number for non-loop joints) to improve readability. The SD/FAST utility routine `SDINDX()` is used to find the correct q and u locations in the state vectors. You could have used the numbers directly from the State Index to Joint/Axis Map, but again, the use of `SDINDX()` is recommended to enhance maintainability and clarity. The routine `SDHINGET()` is used to apply the computed hinge torques.

T3.5 Control Systems

Control systems are a common component of many vehicles and machines, from motor velocity regulators to spacecraft attitude controllers. Control systems usually consist of three parts: (1) one or more sensors to measure some states of the system, (2) a control algorithm that may contain internal states, and (3) one or more actuators that apply loads to the system to cause it to behave in a desired manner. SD/FAST provides utilities to help the user model these sensors, control algorithms, and actuators.

In this section, we will discuss the gyro and camera sensor models used in the spacecraft, the thruster model, the camera actuator model, and the `force()` routine that ties them all together.

T3.5.1 Sensor Models

The `sensor()` routine models the camera azimuth and elevation angle sensors, the gyro sensing of the bus attitude and attitude rates, and the logic used to compute the deviation in these quantities from their desired values. The inputs are the `q` and `u` state vectors, and the desired (commanded) azimuth `azcmd` and elevation `elcmd` of the camera. The attitude and attitude rates are always desired to be zero in this study, so are not inputs. The outputs are the azimuth error `azerr`, the azimuth rate error `azerrt` (deviation from desired zero), the elevation error `elerr`, the elevation error rate `elerrt` (deviation from desired zero), and a combined attitude/attitude rate error signal `err` for the 1-2-3 Euler angles describing the bus attitude. After the subroutine setup:

```
c  sensor
c
c  The sensed quantities are camera az and el positions and
c  rates, base body attitude error and error rate
c  (123 Euler Angles).

      subroutine sensor(q,u,azcmd,elcmd,
     &                  azerr,azerrt,elerr,elerrt,err)
      real*8 q(*),u(*),azcmd,elcmd,azerr,azerrt,elerr,elerrt
      real*8 atterr(3),raterr(3),dc(3,3),err(3),gyro
      integer BUS,CLOCK,CAMERA,SDINDX,i
      parameter (BUS=1,CLOCK=2,CAMERA=3)
      data gyro/2d0/
```

the azimuth and elevation position and rate errors are computed using the current and commanded states of the camera angles:

Position and Rate Errors for Camera

```
      azerr  = azcmd - q(SDINDX(CLOCK,1))
      elerr  = elcmd - q(SDINDX(CAMERA,1))
      azerrt = -u(SDINDX(CLOCK,1))
      elerrt = -u(SDINDX(CAMERA,1))
```

The SD/FAST utility `SDORIENT()` is used to obtain the 3x3 direction cosine matrix `dc` for the bus attitude. The SD/FAST utility `SDDC2ANG()` is used to convert `dc` to 1-2-3 Euler angles, `atterr` (it's also the error since 0,0,0 are the desired Euler angles).

Get Spacecraft Attitude Error

```
c  Convert bus attitude to 1-2-3 Euler angles.
        call SDORIENT(BUS, dc)
        call SDDC2ANG(dc,atterr(1),atterr(2),atterr(3))
```

The bus inertial angular rate is obtained using the utility `SDANGVEL()` and the attitude position and rate errors are combined (`gyro` is a scale factor on the rate error) to generate an overall attitude error signal `err`.

Get Spacecraft Rate Error

```
c  Get bus inertial angular velocity (in the bus frame).
        call SDANGVEL(BUS,raterr)
```

Compute Gyro Error Signals

```
c  The deadband logic mixes pos and vel errors for each axis.
        do 10 i = 1,3
 10       err(i) = gyro*raterr(i) + atterr(i)

        return
        end
```

T3.5.2 Thruster Model

The thrusters apply body-fixed torques about any of the spacecraft's three axes. However, the thrusters must remain on for a minimum length of time `tmin`. The `thrust()` routine provides the logic to provide a delay of at least `tmin` seconds in turning the thrusters off when the combined attitude/attitude rate error signal `error` falls within the acceptable deadband `dband` for a particular axis. The actual application of the thrust is handled in the next routine, `actuator()`.

```
c  thrust
c
c   Thruster model. Each thruster can fire in the positive or
c   negative direction.  If the error is large enough to
c   trigger a thruster firing, the thruster must remain on for
c   a minimum amount of time.  This routine can cause problems
c   for variable step integrators because of the 'memory'
c   built into the routine.

        real*8 function thrust(t,axis,error)
        real*8 t,error,dband,tmin,fire(3),toff(3)
        integer axis
        data dband,tmin /0.0025d0, 0.02d0/
        data fire,toff /3*0d0,3*0d0/
```

Fire Thrusters
if Exceed Deadband

```
      if (error .lt. -dband) then
          fire(axis) = 1d0
          toff(axis) = t + tmin
      else if (error .gt. dband) then
          fire(axis) = -1d0
          toff(axis) = t + tmin
      else if (t .ge. toff(axis)) then
          fire(axis) = 0d0
      end if

      thrust = fire(axis)
      end
```

Note that the direction of the thrust is handled by the side of the deadband that the error lies on, and `toff` is used to enforce the minimum off-delay.

T3.5.3 Camera Pointing Controller

The routine `actuator()` handles the camera motor control and the actual application of the body-fixed thruster torques. After the routine setup code:

```
c  actuator
c
c  This routine applies the forces and torques acting on the
c  spacecraft, as a function of the passed-in errors.

      subroutine actuator(t,azerr,azerrt,elerr,elerrt,err)
      real*8 t,azerr,elerr,err(3),azerrt,elerrt
      real*8 torq(3),thrust,k1,k2,b1,b2,l(3)
      integer BUS,CLOCK,CAMERA,axis
      parameter (BUS=1,CLOCK=2,CAMERA=3)
      data k1,k2,b1,b2/2*3500d0,2*20d0/
      data l/0.23d0,0.21d0,0.31d0/
```

the camera motor control torques are applied to the camera using the SD/FAST `SDHINGET()` routine. Note that the controller consists of two independent Proportional-Derivative (PD) control laws:

Proportional-Derivative
Camera Controller

```
c  The camera controller just uses rate and position error
c  feedback.
      call SDHINGET(CLOCK,1,  k1*azerr + b1*azerrt)
      call SDHINGET(CAMERA,1, k2*elerr + b2*elerrt)
```

The body-fixed torques are applied using the SD/FAST `SDBODYT()` routine:

Apply Body Torques
From Thruster Pulses

```
c  Compute base body torques from model of thrusters and their
c  controller, expressed in base body frame.

      do 10 axis = 1,3
10        torq(axis) = l(axis)*thrust(t,axis,err(axis))
      call SDBODYT(BUS,torq)

      return
      end
```

The user-written `thrust()` routine conveniently enforces the minimum off-delay for each of the three spacecraft axes.

T3.5.4 `forces()` Routine

The user-written `forces()` routine simply calls all the sensor and actuator routines in the proper order, along with the flexible boom model. Note that we would have named the routine `sduforce()` if we wanted to use the Simplified Analysis Routines. However, in this case, we are writing our own analysis driver routines so the name doesn't matter.

Read Sensors

Apply Camera & Thruster Torques

Flex Boom Forces

```
c  forces
c
c  This subroutine takes time, the system state, and the camera
c   azimuth and elevation commands as inputs. These are used to
c   generate sensor outputs which are then passed to the
c   actuator routine.  Bending of the flexible boom is handled
c   in a separate subroutine.

      subroutine forces(t,q,u,az,el)
      real*8 t,q(*),u(*),az,el
      real*8 azerr,azerrt,elerr,elerrt,buserr(3)

c  Sense errors in camera and base body orientations and rates.
      call sensor(q,u,az,el,azerr,azerrt,elerr,elerrt,buserr)

c  Apply control forces to reduce the sensed errors.
      call actuator(t,azerr,azerrt,elerr,elerrt,buserr)

c  Apply forces to model boom flexibility.
      call boomflex(q,u)

      return
      end
```

T3.6 Using General Analysis Routines

To integrate the equations of motion supplied by SD/FAST, or to perform other types of analyses, the user has three options:

1. Use the Simplified Analysis Routines (See Simplified Analysis Routines on page R-98) if your problem can be solved using these easy-to-use routines provided by SD/FAST. This is the easiest approach.
2. Use the General Analysis Routines (See General Analysis Routines on page R-34) if your problem can't be solved using the Simplified Analysis Routines, such as if you have additional states which must be integrated (like the command rates in this problem), or need the services of a nonlinear root finder.
3. Use your own simulation environment, such as ACSL, Easy 5, System Build with Matrix-X, Simulink, or a proprietary tool.

The Simplified Analysis Routines are discussed in Tutorials 1, 2, and 4. We shall use the General Analysis Routines in this Tutorial. The General Analysis Routines consist of two integrators: one a variable-step, the other a fixed-step; and a nonlinear root finder. For this problem, we shall use the fixed-step integrator, `SDFINTEG()`, a 4th order Runge-Kutta integrator with a Merson error estimator. We used the fixed-step integrator because the thruster model we wrote used off-delay time "memory" that a variable-step integrator would cause to malfunction.

Rather than simply passing the numerical derivatives to be integrated, like we did when using `SDMOTION()` (See page R-42), we must write the derivative routine ourselves. Here we show the `deriv()` routine for the slewing spacecraft problem. `deriv()` must be declared `external` in the main program. This first section of code is the set-up. Note the two additional states for the command rates, `az` and `el` are indexed by integers `AZCMD` and `ELCMD` to be tacked to the end of the `state()` and `dstate()` arrays:

```
c  deriv
c
c  Compute state derivatives.  We'll return with non-zero
c  status if a constraint error is larger than a particular
c  tolerance.  The tolerance is passed in param(1).  (In this
c  problem, the only constraint is the prescribed motion on
c  the scanner.)
```

Deriv() Routine for Integrating SD and User States

```
      subroutine deriv(time,state,dstate,param,status)
      integer NQ,NU,NC,AZCMD,ELCMD
      parameter (NQ=12,NU=11,NC=1,AZCMD=NQ+NU+1,ELCMD=AZCMD+1)
      real*8 time,state(*),dstate(*),param(1),errs(NC)
      real*8 slwstrt,slwstp,azrate,elrate
      common /slewparm/slwstrt,slwstp,azrate,elrate
      integer status,i
```

Next we assign the command rate states to their positions in the state vector, call `SDSTATE()` to supply the current states to SD/FAST (`SDSTATE()` will not affect the user states), call `forces()` and `motions()` to compute and apply the current forces and prescribed motions, and finally call `SDDERIV()` to compute the SD/FAST derivatives. Note that the `state(1)` through `state(NQ)` are the SD/FAST *q*'s, `state(NQ+1)` through `state(NQ+NU)` are the SD/FAST *u*'s, and `state(NQ+NU+1)` and `state(NQ+NU+2)` (that is, `state(AZCMD)` and `state(ELCMD)`) are the two user states.

```
    call SDSTATE(time,state,state(NQ+1))
    call forces(time,state,state(NQ+1),state(AZCMD),state(ELCMD))
    call motions(time,state,state(NQ+1))
    call SDDERIV(dstate,dstate(NQ+1))
```

Next we compute the state derivatives for the two user states:

```
c  Determine the currently commanded rate for the clock and
c  camera motors.  This rate is a constant during the slewing
c  maneuver, and zero otherwise.
```

Compute User
State Derivatives

```
        if ((time .ge. slwstrt) .and. (time .lt. slwstp)) then
            dstate(AZCMD) = azrate
            dstate(ELCMD) = elrate
        else
            dstate(AZCMD) = 0.0D0
            dstate(ELCMD) = 0.0D0
        end if
```

We also should check for velocity and position constraint errors. The tolerance is passed through the available `PARAM` array. If the constraints are violated, `status=1` is returned.

```
c  Check that constraint errors are below tol.
```

Constraint Error
Checking

(Should be done by
User when using
Generalized Analysis
Routines.)

```
        status = 1

        call SDVERR(errs)
        do 10 i=1,NC
10          if (abs(errs(i)) .gt. param(1)) return
        call SDPERR(errs)
        do 20 i=1,NC
20          if (abs(errs(i)) .gt. param(1)) return

        status = 0

        return
        end
```

The routine `simulate()` organizes one simulation and is called for each desired run. `simulate()` makes calls to `deriv()` using the fixed-step integrator `SDFINTEG()` where each element in the call is explained on page R-37.

Initialize

```
c simulate
c
c Run a simulation of the spacecraft to produce nout data points
c beginning at time t and separated by (nbtw*dt) seconds.  Complain if
c the integration error or a constraint error exceeds tol.  Output all
c the interesting data.  Set scan nonzero to enable the scanner.
c Output goes to file 10+<simulation number>, i.e., fort.11, fort.12,
c fort.13, or fort.14.
c
        subroutine simulate(n,nout,nbtw,dt,tol,scan,t,y)
        integer NQ,NU,NEQ,AZCMD,ELCMD,BUS,BOOM,SCANNER
        parameter (NQ=12,NU=11,AZCMD=NQ+NU+1,ELCMD=AZCMD+1,NEQ=NQ+NU+2)
        parameter (BUS=1,BOOM=4,SCANNER=5)
        integer i,j,n,nout,nbtw,scan,err,SDINDX
        real*8 dt,tol,t,y(NEQ),param(1),dy(NEQ),cone,ori(3),dc(3,3)
        real*8 est,work(4*NEQ),fire(3)
        common /firing/ fire
```

Generate Headings

```
      write(6,*)'Simulation #',n,' for ',dt*nout*nbtw,
     &           ' seconds from t=',t
      if (scan .ne. 0) write(6,*)'Scanner is on.'
      if (scan .eq. 0) write(6,*)'Scanner is off.'

c  Supply the tolerance for the prescribed motion constraint.
      param(1) = tol
```

Compute Initial Derivatives

```
c  Integrator requires correct derivatives passed in dy --
c  evaluate now.
      call deriv(t,y,dy,param,err)

c  Integrate and write out data.  Note that bus orientation is 1-2-3
c  Euler angles (millirad) rather than Euler parameters.

      do 20 i = 1,nout+1
        call SDORIENT(BUS,dc)
        call SDDC2ANG(dc, ori(1), ori(2), ori(3))
        call pointerr(y(AZCMD),y(ELCMD),cone)
```

Extract Output

```
        write(10+n,30) t, ori(1)*1000., ori(2)*1000., ori(3)*1000.,
     &      y(SDINDX(SCANNER,1)), y(SDINDX(BOOM,1))*1000.,
     &      y(SDINDX(BOOM,2))*1000., fire(1), fire(2), fire(3), cone
```

Integrate Between Print Intervals Using Fixed-Step Kutta-Merson

```
        if (i .le. nout) then
            do 10 j = 1,nbtw
              call SDFINTEG(deriv,t,y,dy,param,dt,NEQ,work,est,err)
              if ((err .ne. 0) .or. (est .gt. tol)) then
                  print *,'at time=',t,' err=',err,' errest=',est
              end if
10          continue
        end if
20    continue
30    format(10f10.5)
      end
```

The `pointerr()` routine called above computes the camera pointing error:

```
c To compute pointing accuracy, just reading the gimbal angle
c errors is insufficient since it does not take into account base
c body pointing error.  If the system is in its reference
c configuration, the correct camera pointing vector v is given by
c first rotating the camera about the clock z-axis, followed by a
c rotation about the camera -x axis. Expressed in the inertial
c frame this is:
c
c      v = [ -sin(az)*cos(el)   cos(az)*cos(el)   -sin(el) ]
c
c The pointing error can be found by transforming the actual camera
c bore sight into the inertial frame, dot multiplying with v to
c obtain the cosine of the cone angle, and then using acos.
c The cone angle is reported in milliradians.

        subroutine pointerr(az,el,cone)
        real*8 az,el,cone,v(3),bore(3),boreg(3),c
        integer CAMERA,GROUND
        parameter (GROUND=0,CAMERA=3)
```

```
c  This is the orientation of the bore sight in the camera's
c  local frame.
        data bore/0d0,1d0,0d0/

        v(1) = -sin(az)*cos(el)
        v(2) =  cos(az)*cos(el)
        v(3) = -sin(el)

        call SDTRANS(CAMERA,bore,GROUND,boreg)
        c = boreg(1)*v(1) + boreg(2)*v(2) + boreg(3)*v(3)
        cone = acos(c)*1000d0
        end
```

Compute Bore sight Pointing Error

Summary

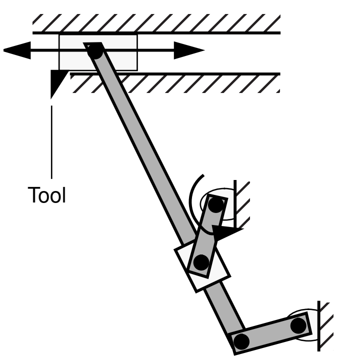
- This tutorial covered an example of using SD/FAST for a complex open-loop topological system with prescribed motion, flexible body dynamics, a control system, integration of user-defined states using General Analysis Routines, and design study capability.
- The roadmaps in the "`_info`" file are essential to use the SD/FAST routines correctly.
- Prescribed motion can be enforced on any hinge axis, and turned on and off if the position and velocity states are continuous. *Compatible* position, velocity, and acceleration time histories must be supplied.
- Flexible-body dynamics can be approximated in SD/FAST by using lumped mass-spring models, but the correct geometry, masses, and spring constants must be supplied by the user.
- Control systems and other models requiring the integration of user-defined states require the usage of General Analysis Routines or a user-supplied integrator.
- Physical model parameters, such as hinge locations and body masses can be left as variables in the SD/FAST routines to allow studies of system performance versus design changes.
- Several generated routines were introduced including: (1) user-written prescribed motion routines `SDPRESPOS()`, `SDPRESVEL()`, `SDPRESACC()` specifying prescribed position, velocity and acceleration; (2) specifying the location of state variables in the state vector using `SDINDX()`; (3) obtaining information about a free-flying body (`SDORIENT()`, `SDDC2ANG()`, `SDANGVEL()`); (4) applying body torques (`SDBODYT()`); and (5) General Analysis Routine for fixed-step integration (`SDFINTEG()`).

TUTORIAL 4 Quick-Return Mechanism

Objectives

- Discuss special considerations for modeling and analyzing closed-loop systems.
- Develop and assemble the model of a quick-return mechanism.
- Run a suite of typical mechanism analyses on the quick-return mechanism.

T4.1 Introduction to Closed-Loop Systems



This tutorial will take you through an example of a simple, but useful closed-loop system: a quick-return mechanism. This particular quick-return mechanism is used to cut a workpiece by applying a shaping tool rigidly mounted on the horizontal sliding body. The tool slides along a horizontal axis as the workpiece is engaged. At the end of the working stroke, the tool quickly retracts back to the starting position, ready to repeat the cycle. A motor turning the crank at the joint in the center drives the mechanism.

Systems with closed-loop topology are one instance of constrained mechanical systems. SD/FAST can handle closed loops automatically and provides the tools needed for the user to specify more general constraints, including non-holonomic constraints. We will only consider closed-loop systems in this tutorial. More general constraints are covered in Reference Section R24 of this manual and in Application Notes.

In Section T4.1, we discuss closed-loop mechanical systems in general and specifically how SD/FAST models closed-loop systems. In Section T4.2 and Section T4.3, we will run through the model development and several analyses of a quick-return mechanism.

The necessary computer files for this example can be found by looking in the “tutorials” subdirectory of the SD/FAST distribution directory for all the files called:

```
quickret*
```

where "*" stands for a suffix indicating the various SD/FAST-generated or user-generated files. Please see the Release Notes for your system to determine the recommended character extensions and suffixes. This tutorial will use character extensions and prefixes for a simulation developed in Fortran on a UNIX-based Sun Workstation.

T4.1.1 Comparison to Open-Loop Systems

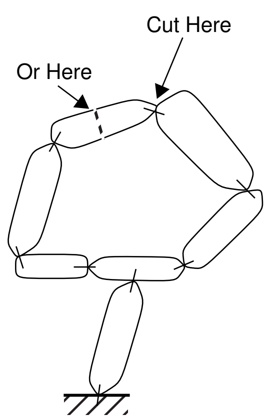
Closed-loop systems such as mechanisms, that is, chains of bodies connected into one or more ringlike structures, usually are quite different from open-loop systems such as spacecraft or robotic manipulators. Open-loop systems are typically characterized by a large number of degrees of freedom for providing general position and orientation, while closed-loop systems characterizing mechanisms have only one or a few degrees of freedom, so the motion is highly constrained. SD/FAST is designed to favorably handle these highly-constrained closed-loop systems as well as the open-loop systems.

Open-loop systems generally move in all three dimensions, whereas many closed-loop mechanisms are characterized by all motion constrained to lie in a plane—perhaps because of limitations of available synthesis and analysis tools for mechanism designers. However, SD/FAST is capable of easily handling analyses in full three dimensions as well as two.

The objectives of mechanism designers can be different from those for designers of open-loop systems such as spacecraft. Whereas spacecraft or robot designers may be interested in achievable pointing accuracy and stability, the mechanisms designer is concerned with function and path generation for the output body of the mechanism, power transmission and conversion, mechanical advantage, static and dynamic balancing, bearing reactions, etc. All these analysis capabilities, and more, are available in SD/FAST.

T4.1.2 How SD/FAST Models Closed-Loop Systems

The equations of motion underlying SD/FAST directly represent open-loop chains. The capability for analyzing closed-loop systems is achieved by adding loop joints that connect open-loop chains together. Additionally, tools for assembling these "broken loops" into the final closed-loop system are provided that eliminate any need for the user to compute complex geometrical relationships or compatible velocities of the bodies.



Thus the user can enter the system as a set of "parts" laid out for easy entry of geometry and mass properties, and SD/FAST will assemble these parts into the final system with compatible velocities. The additional tasks in using SD/FAST to model closed-loop systems are to (1) define loop joints, (2) perform an assembly analysis, if needed, and (3) perform a velocity analysis, if needed.

Loop Joints

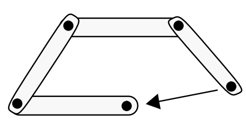
When modeling a closed-loop system with SD/FAST, the model must be divided into one or more open-loop topological "tree" systems and a set of "loop joints" which produce closed loops in the topology. This is done by identifying loops in the system topology and making one cut" in each loop. Although any joint in the loop may be cut, it is most efficient to cut at the joint with the *most* degrees of freedom since that produces the least number of constraints. It is also good to minimize the maximum length of any chain of bodies in the system, so cutting a loop in the middle is generally better than cutting it at either end.

You can also cut *inside a body* and re-form the loop by placing a weld joint between the two pieces. This has advantages in terms of uniform treatment of all joints. However, no DOFs are removed and weld joint adds 6 constraints, so this method is computationally more expensive. Note that the easiest way to "cut" a body is to replace it with two identical bodies, each with one-half the density of the original body and occupying the same space when assembled. The weld joint is simply placed at the center of mass.

If the reference configuration is input such that all the loop joints are already assembled, then the loop joint specifications are exactly like those for joints in the tree system. Otherwise, some additional vectors can be specified to control the allowable assembled configurations and to define the zero positions for the loop joint coordinates (called "pseudo coordinates"). If these additional vectors are not specified, SD/FAST will choose defaults. The body pin will be aligned with the final inboard pin, and the reference lines will be parallel.

There are a few minor restrictions that apply to loop joints which do not apply to tree joints. Please refer to Reference Section R11.1 for details.

Assembly Analysis

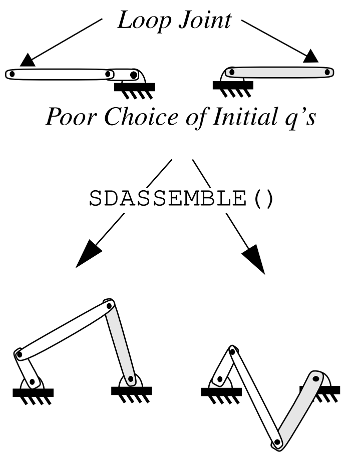


SD/FAST *automatically* supplies a routine (`SDASSEMBLE()`) to assemble loop joint constraints, enforce prescribed motion constraints, and enforce user-defined constraints (Section R24). All of these tasks are solved under the general approach called "assembly analysis," perhaps more appropriately thought of as "initial constraint enforcement." The most obvious use is assembling the various parts laid out into the final assembled system.

The user can also write functions to enforce many other types of initial constraints such as: positioning the ends of open loop chains (as in positioning the end of a robot), giving initial conditions otherwise difficult to find geometrically, orienting bodies with respect to the ground or other bodies, enforcing a specific distance between two points, etc. In fact, any number of functions of the coordinates can be imposed as "required" to be absolutely enforced or "desired" to be enforced in a "least-squares" minimization form after the "requireds" are satisfied. Also these functions can be imposed just to generate an initial condition of the system, to impose constraints throughout an analysis, or to be activated at any time (turned on and off at will) *during* an analysis.

The SD/FAST assembly analysis routine attempts to find a set of hinge position q's which meet all the position constraints so that the largest error in any one constraint is at or below a desired tolerance. The velocities (u's) passed are ignored.

The analysis proceeds incrementally from the passed-in *initial* state to obtain the assembled system configuration. If there are any prescribed motion or user-defined constraints in the problem, the SD/FAST assembly analysis routine will enforce those constraints as well. There is no guarantee that SD/FAST will find a solution even if there is one. If it fails, and you feel that there actually is a solution, you should provide SD/FAST with a better initial guess in the passed-in q's.



Two Possible Branches in the Assembly of a Crank-Rocker Mechanism

Note that the initial guess influences the configuration of the final assembled system. In particular, a different "branch" of the system than that desired may be found (multiple "branch" solutions may exist for some systems of connected bodies). Thus, you should make an initial rough guess to the final desired configuration.

Any particular joint can be "locked" at a desired q if a specific relative joint angle is desired. If the problem contains prescribed motion, it is a good idea to set the q to its correct prescribed value for time t, and then lock that q to remove it from consideration during the analysis. If you start a q a long distance from its correct prescribed value, it may slow convergence or even prevent a solution from being found.

Velocity Analysis

Once the system is assembled to force the loop joints to coincide in space, velocities of the bodies must be computed to allow the system to remain assembled. Otherwise the system will "fly apart." An SD/FAST routine (`SDINITVEL()`) is provided to find initial u's to satisfy loop closure velocities, prescribed velocity conditions, and user-defined velocity constraints. Every user-defined position constraint *must* have a user-defined velocity constraint.

The velocity analysis assumes that the position analysis is already done. That is, the q's are not changed. The velocity analysis assures that the loop joint connect points have the same velocities (derivatives of the position constraints are zero). It also ensures that the relative angular velocity of the bodies connected by a loop joint lies along the joint axis, in the case of a pin joint for instance.

In a manner similar to locking hinge axes in the assembly analysis, locks are also used to set required hinge velocities (such as a motor or crank) which will then remain unchanged by the velocity analysis. Compatible velocities of other bodies are then computed. Unlike the assembly analysis, there is no concept of entering a "wrong branch" in the velocity analysis.

T4.1.3 Special Considerations for Constrained Mechanisms

This section discusses the special considerations that pertain only to constrained mechanisms such as those with closed loops. These considerations can cause problems for the unsuspecting mechanisms designer or analyst. However, most of the problems arising from these considerations can be avoided or mitigated by proper mechanism design in the first place.

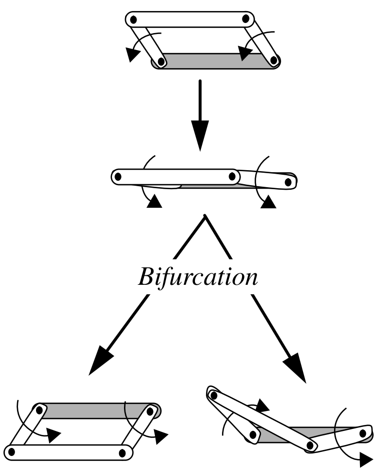
Stabilization of Loop Joints

The equations of a constrained multibody system are a set of coupled differential and algebraic equations. If these are converted into a set of differential equations only (as is commonly done) the new equations will be subject to "drift" during a motion simulation. This drift is caused by the imperfect nature of numerical integration and will cause the original set of algebraic constraints to be violated as the integration proceeds. In that case, the numerical drift must be *stabilized* to prevent the constraint violations from becoming arbitrarily large (i.e., the mechanism literally "flies apart").

SD/FAST automatically includes a stabilization method called Baumgarte Stabilization[1] in the equations of motion and monitors constraint errors in `SDMOTION()`.

Baumgarte Stabilization works roughly like a proportional-derivative (PD) control feedback loop acting at the level of the accelerations. The user may choose two constants: the two feedback "gains" on the proportional (position) and derivative (velocity) loops. These constants must generally be chosen for the specific problem, with larger gains for higher-frequency problems. By default, the gains are zero so there is no stabilization. We suggest that you start with no stabilization and monitor constraint errors. If you must change these constants to effect stabilization or improve response to constraint errors, Baumgarte suggests that the position to velocity ratio be changed as $a^2:2a$ to maintain "critical damping." For many mechanism problems, selecting a=1 is a good place to start if constraints are being violated during a simulation. See Section R4.2 for more detail on choosing stabilization constants.

In general, once the Baumgarte constants are large enough to stabilize the system, further increases reduce the position and velocity errors to just within the integration error tolerance, and no further. However, the speed of recovery from initial condition errors and impulsive changes to the system is increased with larger Baumgarte constants (similar to the way increasing the gains in a PD controller increases the frequency of the second-order error dynamics). However, larger Baumgarte constants introduce higher frequencies in the system which may slow down a variable-step integrator or destabilize a fixed-step integrator. Effect on CPU usage is variable with changes in Baumgarte constants. Some experimentation may be required to find good values to use.



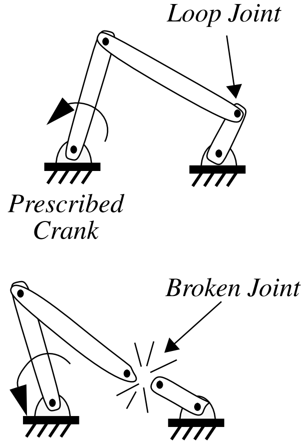
Bifurcation

A "bifurcation" in motion can occur when the system has two (or more) possible *distinct* paths to take when moving through a configuration that momentarily increases the number of degrees of freedom in the system (i.e., some constraint becomes redundant, allowing a momentary increase in the degrees of freedom). A simple example is a four-bar mechanism with exactly equal length crank and rocker (a double-crank mechanism). If the system is allowed to move through the position where all links are aligned, the system can continue around as a parallelogram or bifurcate into an "X" shaped mechanism, with the chosen path bearing no relation to the physical parameters of the model.

In physical systems, bifurcations may never happen if there is joint slop or flexibility. However, since infinitely hard materials are used to model rigid bodies in SD/FAST, any sudden switch in the direction of motion is modeled as a perfectly elastic "collision" and entire chains of bodies will happily "bounce" back the other direction and proceed. In other words, bifurcations are perfectly correct behavior for these very rigid systems modeled by SD/FAST, but are not usually desirable behavior in physical systems one wants to model.

Bifurcation should never occur in a properly designed mechanism, but can occur if, say, the crank and rocker become equal in length during variation of rocker length in a design study. The analyst should then be aware of the problem and correct it if it occurs.

1. Baumgarte, J. "Stabilization of constraints and integrals of motion in dynamical systems," Comp. Meth. Appl. Mech. Engrg., 1 (1972), 1-16.



"Lockup"

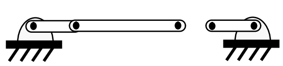
Lockup

A lockup occurs if a mechanism driven by prescribed motion (see Section T3.3) is not able to physically complete its motion without breaking a loop joint or violating a constraint. An example would be a four-bar mechanism with the rocker shorter in length than the crank, which is driven in a circle by prescribed motion. As the crank tries to complete a circle, the links must stretch or a joint must break. If you use the SD/FAST variable-step integrators (`SDMOTION()` or `SDVINTEG()`), an error flag will be set if an apparent lockup occurs. (There may be situations other than lockup which can prevent the integrator from advancing. These may be numerically similar to lockup, so the analyst should apply some common sense to see if the system is really locked up. See Reference Section R8.1 for more information.)

It is important to note that if the motion that causes the mechanism to stop moving is driven by *loads* rather than *prescribed motion*, lockup is *not* considered to have occurred. That is, if the crank in the above example was driven by a constant torque, a joint does not have to break. The mechanism will bounce and, if damping is present, settle to a perfectly acceptable static configuration; similar to the static configurations we studied in Tutorial #2. The key idea is that *only* prescribed motion can drive a system into a true lockup.

Improper Assembly

"Improper" (i.e., undesired) assembly occurs if the system assembles into the wrong branch or configuration. This was discussed in the assembly analysis section above (page T-73). For example, a crank-rocker mechanism could assemble with the rocker roughly 180° turned around. This is easily fixed by changing initial q's to start the system closer to its desired assembled configuration.

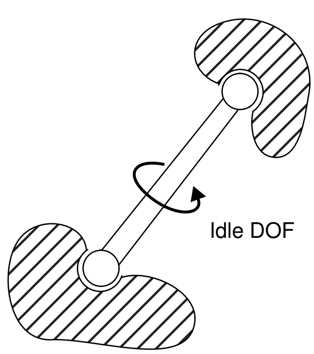


Incompatible Constraint

Incompatible Constraints

Incompatible constraints cause the mechanism to assemble into an immovable system or the system may not even assemble. If the system cannot assemble, an error is returned by `SDASSEMBLE()`. Using the crank-rocker example we've been discussing, the assembly could fail if the links are not long enough to reach between the ground points, or two loop-joint pins (one on each body to be joined) entered in the input file are not parallel to each other. No amount of adjusting the system could cause the pins to align. Thus the system cannot be assembled.

The system would be immovable if any one pin (either regular tree-joint pin or the final assembled loop-joint pin) was not parallel with the others. Binding would occur and the mechanism would not move. These are simple examples, but these problems can easily occur in more complex mechanisms if one is not careful. Note that the user can set a desired tolerance for assembling systems and checking for binding. Prescribed motion and user constraints can also generate incompatible constraints.



Idle or Passive Degrees of Freedom

An idle or passive degree of freedom (DOF) is a motion of the system not associated with the "principal" or primary motion of the system. The definition of what constitutes "principal" motion is, of course, up to the user. The problem usually arises when an unwanted idle or passive DOF appears solely by virtue of the model used. For example, a link with ball joints at both ends can spin freely about the line connecting the two ball joints. Unless that spinning motion is important, you would be better off replacing one of the ball joints with a Ujoint to remove the idle DOF. Of course, if no torques are applied about the spin axis, no acceleration occurs. A problem may also appear if the idle DOF was not discovered and the inertia of the link was modeled as zero about its spin axis to save compute time. Any axial torque would cause infinite acceleration! Note that idle degrees of freedom can also occur in unconstrained systems, but show up more frequently in closed-loop systems.

Another example would be to have cylindrical joints instead of pin joints in the four-bar planar mechanism we have been discussing. This would allow links to “lift off” out of the plane of the mechanism. Again, this is not usually a problem unless forces are applied which will accelerate the body along the passive DOF. But it is usually a better idea (and more efficient computationally) to rearrange joints or add constraints to eliminate passive DOFs.

Redundant Constraints and Non-Uniqueness of Bearing Loads

Most mechanisms are overconstrained, i.e., have redundant constraints. For example, the four-bar mechanism we’ve been discussing would still move in a plane if one of the pin joints were replaced with a ball joint. Thus the two additional constraints provided by the pin joint over the ball joint were not needed; they were redundant. (In fact to completely remove all redundant constraints, you would replace one pin joint with a ball joint and another pin joint with a cylindrical joint). Another example is a door hung by three hinges. Only one rigid hinge is needed to hold the door in place and thus the other two are redundant (for real doors with real-world flexibility and joint slop, at least two separate hinges are advisable!).

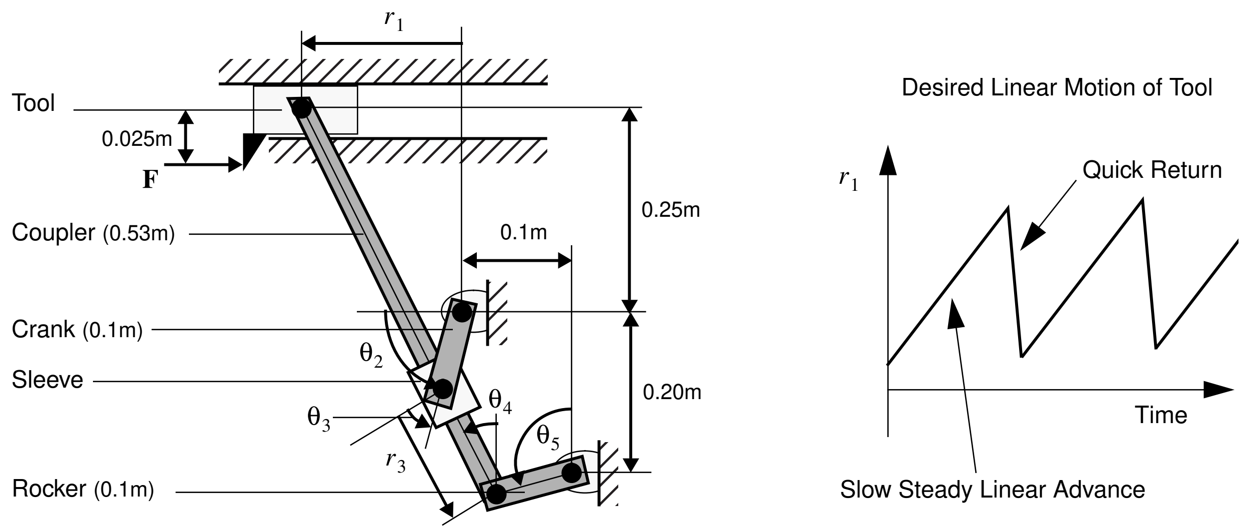
Some mechanism analysis codes require the user to eliminate all redundant constraints. This might be easy for some systems, but can be very difficult for complex systems. However, redundant constraints are not a problem with SD/FAST. Those constraints which are always redundant are often deleted by the SD/FAST symbol manipulator and eliminated from the equations. A subset of the remaining constraints which yields an active nonredundant set is continuously computed at each time step. This subset of constraints is computed to yield the most numerically “robust” solution, and may change during a simulation.

Key Idea

However, *bearing loads will not be unique* for a redundant system. The load path and resultant bearing forces will be correct for the set of active constraints chosen by SD/FAST. But in systems with redundant constraints, the actual bearing loads depend upon details of the joints and elastic models of the bodies not considered by SD/FAST. However, in some special situations, *some* bearing loads reported for redundant mechanisms are correct. For example, the out-of-plane bearing forces in the joints of a planar four-bar linkage constructed with four pin joints are not unique, whereas the *in-plane* bearing forces *are* uniquely defined. In general, the in-plane loads for all planar mechanisms are unique, provided that there are no in-plane redundancies.

If the user wants to eliminate redundant constraints to find all bearing loads uniquely, more of the local effects must be taken into account, such as joint slop. Thus a theoretical pin joint may be better modeled as a ball joint to account for small tipping motion allowed by the joint. In fact, high-speed or high-precision mechanism are usually *designed* to avoid redundant constraints (such as the joint between the connecting rod and its piston in an internal combustion engine which is actually designed to *allow* small tipping motions that are better modeled by a ball joint than a pin joint.)

Figure T4-1 Model of the Quick-Return Mechanism



T4.2 Model and Assemble the Quick-Return Mechanism

In this section we model and assemble the quick-return mechanism. First we give an overview of the mechanism. Next the unassembled model is developed. Then the method for writing system description files for closed-loop mechanisms is described. Finally, the actual SD/FAST assembly and velocity analyses are performed.

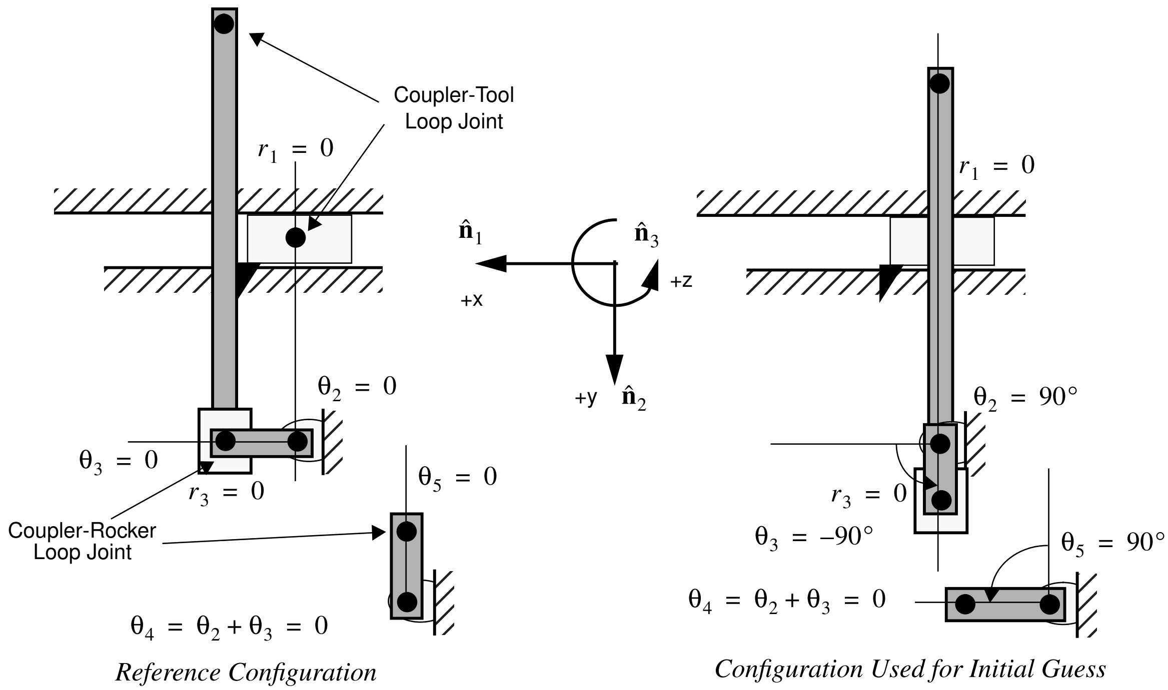
T4.2.1 Overview of the Quick-Return Mechanism

Figure T4-1 shows the assembled quick-return mechanism. This particular quick-return mechanism is used to cut a workpiece by applying a shaping tool rigidly mounted on the body named "tool." The tool slides along a horizontal axis as the workpiece is engaged. At the end of the working stroke, the tool quickly retracts back to the starting position, ready to repeat the cycle. The force **F** applied to the shaping tool is zero during the retraction of the tool.

The mechanism consists of five moving links(1:tool, 2:crank, 3:sleeve, 4:coupler, 5:rocker) plus the ground. The five desired position coordinates for the bodies are tool displacement r_1, crank angle θ_2, sleeve displacement r_3, coupler angle θ_4, and rocker angle θ_5. Also, the angle between the sleeve and the crank, θ_3, will be used. The links are connected by five pin joints and two sliding joints. There are two closed-loops in this mechanism. The length of each link is given in the figure, if needed by the model. Mass properties and other model parameters will be introduced as needed.

Depending upon the analysis under consideration, the crank in this problem is driven by either prescribed motion or a motor with a linear torque-speed curve. When driven by a motor, the mechanism has one dynamic degree of freedom. When driven by prescribed motion, the mechanism has *zero* degrees of freedom (that is, no loads applied anywhere on the mechanism can affect the motion). The crank is actually a flywheel (for smoothing the stroke under varying shaper loads), but has been drawn as a link for clarity.

Figure T4-2 Quick-Return Mechanism Configurations



T4.2.2 Choosing the Unassembled Model

There are several methods of presenting the geometrical model of the mechanism to SD/FAST. One method would be to enter the mechanism already assembled. While this may be easy for simple symmetrical mechanisms such as equal-length-link four-bar mechanisms, complex mechanisms such as this quick-return would require a lot of tedious trigonometric calculations. Thus, we should allow SD/FAST to automatically assemble the mechanism.

The next question then, is to decide how to cut the loops and lay out the resulting trees to be joined. We can either cut the mechanism at a joint, thus making it a loop joint, or we could cut a body itself to be later joined by the zero DOF weld joint. Cutting at a joint is preferable since a weld joint always adds six more constraints, whereas a loop joint adds only zero (for a sixdof joint) to five constraints (for a pin joint), and removes the equations associated with the tree joint it replaces. However, if your mechanism is simple and you would prefer to avoid pseudo loop joint coordinates and their additional considerations, use weld joints. We shall cut at the joints so that the more efficient case can be studied.

We have two loops to break, and should break them to create short-chained trees (more efficient computationally). We choose to break the pin joint between tool and the coupler, and the pin joint between the rocker and the coupler. We then lay the parts out in their reference configuration as shown in Figure T4-2. Note that the parts are positioned either vertically or horizontally for easy entry and generation of lots of zeros in the mass and geometrical parameters. The parts have been arranged in orientations we desire to represent zero for our chosen position coordinates.

All desired position variables, except θ_4, are the relative joint angles between bodies in unassembled trees, which are automatically reported by SD/FAST. θ_4 can be easily found by simply adding up all the relative joint angles along the tree from the ground: $\theta_4 = \theta_2 + \theta_3$. We could also compute θ_4 by asking SD/FAST for the orientation of the coupler in the ground frame—the third Euler angle is exactly θ_4.

The reference configuration, however, does *not* have to be the initial configuration from which the assembly analysis is begun (*any* initial set of tree joint coordinates can be used). In fact, the reference configuration shown is not the best initial configuration. The rocker could move either to the left as desired or to the right and produce the wrong branch. Thus for the initial guess for the assembly, we chose an initial configuration with the loop joints closer together and clearly in the correct branch. We rotated the rocker to the left $90°$, then rotated the crank-sleeve-coupler tree down $90°$, followed by rotating just the sleeve-coupler back up $90°$. We will check the assembly to make sure we entered the correct branch.

T4.2.3 Write System Description File

Now that we have defined our model for the quick-return mechanism, we can proceed with writing the SD/FAST system description file. The details of describing open-loop tree geometry and mass properties are the same as previously described and will not be discussed extensively. The example system description file is named `quickret.sd` on your distribution media and is shown in Figure T4-3, without the comments.

There is no preamble, since we ignored gravity for this system. The five body paragraphs describe the geometry and mass properties of the five bodies in the mechanism as laid out in the unassembled *reference configuration* of Figure T4-2. The bodies are assumed to be uniform bars with mass centers midway along their lengths (except for the crank which is really a flywheel with its mass center at its attach point to ground). The masses and inertias are typical for the actual mechanism. Note that the crank has been set with optional prescribed motion, since some of the analyses will require it.

The last two paragraphs describe the loop joints. SD/FAST recognizes these paragraphs as describing loop joints because the body "coupler" has been previously defined. Only the `bodytojoint`, `inbtojoint`, and `pin` vectors are defined since the mass properties have already been specified (indeed, you'll get an error if you try to multiply define mass properties of a body) and no special vectors or reference lines for the loop joints are needed since the pins on the inboard and outboard bodies are already aligned in the reference configuration, and we will not need to use loop joint pseudo-coordinates.

Figure T4-3 System Description File for Quick-Return Mechanism

Body Paragraphs

Loop Joint Paragraphs

```
#
body = tool inb = $ground joint = slider
  mass = 15.87              inertia = 0 0 0.01366
  bodytojoint = 0 0 0       inbtojoint = 0 0 0         pin = 1 0 0

body = crank inb = $ground joint = pin   prescribed = ?
  mass = 0                  inertia = 0 0 0.68
  bodytojoint = 0 0 0       inbtojoint = 0 0.25 0      pin = 0 0 1

body = sleeve inb = crank joint = pin
  mass = 1.36               inertia = 0 0 0.00066
  bodytojoint = 0 0 0       inbtojoint = 0.1 0 0       pin = 0 0 1

body = coupler inb = sleeve joint = slider
  mass = 4.54               inertia = 0 0 0.10756
  bodytojoint = 0 0.265 0   inbtojoint = 0 0 0         pin = 0 1 0

body = rocker inb = $ground joint = pin
  mass = 1.81               inertia = 0 0 0.00156
  bodytojoint = 0 0.05 0    inbtojoint = -0.1 0.45 0   pin = 0 0 1

# The next two paragraphs describe the loop joints

body = coupler inb = tool joint = pin
  bodytojoint = 0 -0.265 0  inbtojoint = 0 0 0         pin = 0 0 1

body = coupler inb = rocker joint = pin
  bodytojoint = 0 0.265 0   inbtojoint = 0 -0.05 0     pin = 0 0 1
```

T4.2.4 Run SD/FAST and Examine Information File

Next run the quick-return mechanism system description file through SD/FAST:

```
sdfast quickret.sd
```

and examine the resulting `quickret_info` file in Figure T4-4. Notice that there are new sections containing information about the loop joints.

In the `ROADMAP`, an entire section named `Loop Joints` is added for closed-loop systems. The pseudo coordinates `lq` are available to the user exactly the same way as for tree joints with the exception of no wraparound (integrate the `lu`'s to accumulate total number of revolutions). We have set up this problem to avoid the use of pseudo coordinates.

The `STATE INDEX` section lists the mapping from the state vector to the specific joint and axis, with an additional section exclusively for pseudo coordinates and rates.

The `SYSTEM PARAMETERS` section lists overall degrees of freedom, numbers of joints, etc., as discussed previously. Note that the total number of constraints `nc` is 11. Assuming all prescribed motion and user constraints are active and the rank of the constraint matrix reported by `SDMULT()` falls below 11, the system *has redundant constraints* and thus any bearing loads reported should be suspected to be *non-unique*.

Examination of the generated code in `quickret_dyn.f` shows that 6 of the 11 constraints were eliminated symbolically and do not even appear in the code. The tree system has 5 DOF. So the 5 constraints yield 0 DOF when prescribed motion is enabled, 1 DOF otherwise. Since the mechanism is planar and has no in-plane redundancies, the in-plane loads are unique.

All the user-written code for this example is in the file `quickret.f`.

Figure T4-4 Information File for the Quick-Return Mechanism

```
ROADMAP (quickret.sd)

Bodies        Inb
No  Name      body Joint type  Coords q         Multipliers
--- --------- ---- ----------- ---------------- -----------------------
  0 $ground                                     |
  1 tool        0  Sliding       1              |
  2 crank       0  Pin(1D)       2?             |  1p
  3 sleeve      2  Pin(1D)       3              |
  4 coupler     3  Sliding       4              |
  5 rocker      0  Pin(1D)       5              |

Loop Joints                    Pseudo Coords lq

  6 coupler     1  Pin(1D)       1              |  2   3   4   5   6
  7 coupler     5  Pin(1D)       2              |  7   8   9  10  11


STATE INDEX TO JOINT/AXIS MAP (quickret.sd)

 Index
  q|u   Joint  Axis   Joint type    Axis type
 -----  -----  ----   -----------   ----------
  1|6     1      1    Sliding       translate
  2|7     2      1?   Pin(1D)       rotate
  3|8     3      1    Pin(1D)       rotate
  4|9     4      1    Sliding       translate
  5|10    5      1    Pin(1D)       rotate

 lq|lu

  1|3     1      1    Pin(1D)       rotate
  2|4     2      1    Pin(1D)       rotate


SYSTEM PARAMETERS (quickret.sd)

Parameter  Value  Description

nbod           5  no. bodies (also, no. of tree joints)
njnt           7  total number of joints (tree+loop)
ndof           5  no. degrees of freedom allowed by tree joints
nloop          2  no. loop joints
nldof          2  no. degrees of freedom allowed by loop joints

nq             5  no. position coordinates in state (tree joints)
nu             5  no. rate coordinates in state (tree joints)
nlq            2  no. position coordinates describing loop joints
nlu            2  no. rate coordinates describing loop joints

nc            11  total no. constraints defined
nlc           10  no. loop joint constraints
npresc         1  no. prescribed motion constraints
nuserc         0  no. user constraints
```

Possible prescribed motion with multiplier to enforce

Section for loop joints

Pseudo coordinates for loop pin joints

Multipliers to enforce ten loop joint constraints

Section for loop joints

Tree DOF

Constraint Information

T4.2.5 Applied Loads

Since we have ignored gravity in this system (it's not a major contributor to the dynamics of the system), only two sources of applied loads remain: (1) the force applied to the shaper during its cutting stroke, and (2) the motor driving the crank. The applied loads will be computed in user-written `sduforce()` since we shall use the Simplified Analysis Routines.

Cutter Force

The cutter force is a constant force of 450 Newtons applied only during the forward stroke of the tool:

$$\mathbf{F} = \begin{cases} -450\hat{\mathbf{n}}_1,\ \dot{r}_1 \geq 0 \\ 0\hat{\mathbf{n}}_1,\ \dot{r}_1 < 0 \end{cases} \qquad \textbf{Eqn. T2.4}$$

The initial set-up code and cutter force code in `SDUFORCE()` to implement this applied force is:

```
c sduforce
c
c This routine computes and applies the forces acting in the system.

      subroutine sduforce(t,q,u)
      real*8 t,q(*),u(*),cutter(3),fk(3),w,torq
      integer presval,isques,TOOL,CRANK,SDINDX
      parameter(TOOL=1,CRANK=2)
      data cutter/0d0,0.025d0,0d0/

c Compute and apply the cutter force. If the tool is moving forward,
c the force is applied.  Otherwise no force is applied.  `cutter'
c is the location of the point on body TOOL at which the cutter force is
c to be applied.

      fk(1) = 0d0
      if (u(SDINDX(TOOL,1)) .ge. 0d0) fk(1) = -450d0
      call SDPOINTF(TOOL,cutter,fk)
```

The first element of the force vector `fk()` is set to zero, unless the velocity of the tool is greater than zero. Note that the velocity of the tool (which is one of the generalized speeds passed in u) is obtained by using `SDINDX()` with the parameter `TOOL` set to the body number "`1`" as reported by the roadmap of the system. If the velocity is positive, the force is computed and applied to `TOOL` at point of application "`cutter`" using `SDPOINTF()`.

Crank Motor

Unless the crank motor is prescribed by `sdumotion()`, the motor applies a torque as a function of the speed of the crank according to the following linear torque-speed relationship:

$$\tau = \begin{cases} 90,\ u < 0 \\ 90 - 3u,\ 0 < u < 30 \\ 0,\ u > 30 \end{cases} \qquad \textbf{Eqn. T2.5}$$

where the stall torque is 90 Newton-meters and the no-load speed is 30 rad/sec.

The crank motor torque code in `sduforce()` is:

```
c If the crank motion is not prescribed, we'll use motor torque-rpm
c line and apply a torque to crank.

      call SDGETPRES(CRANK,1,presval)
      if (presval .eq. 0) then
          w = u(SDINDX(CRANK,1))
          torq = 90d0 - 3d0 * w
          if ( w .gt. 30d0) torq = 0d0
          if ( w .lt. 0d0 ) torq = 90d0
          call SDHINGET(CRANK,1,torq)
      end if

      return
      end
```

where `presval` (returned from `SDGETPRES()`) determines if prescribed motion is turned on (remember to always use SD/FAST-supplied routines to determine the values of parameters or the state of flags, etc.). The velocity of the crank to use in the torque-speed relation is also extracted using the `SDINDX()` routine. `torq` is applied through `SDHINGET()`.

T4.2.6 Prescribed Motion

The only prescribed motion is a constant spin rate for the crank during some of the analyses. It is generally a good idea to provide position, velocity, and acceleration information if you can, but only acceleration is required. If all are provided, make sure that the expression you give for velocity is the time derivative of the one you provide for position, and that likewise the acceleration is the derivative of velocity. See Reference Section R14 for more information.

The code for the prescribed motion of the crank is contained in `sdumotion()`:

```
c sdumotion
c
c If enabled, prescribe crank motion to a constant velocity rps plus an
c initial crank rotation theta2, both passed in common.

      subroutine sdumotion(t,q,u)
      real*8 t,q(*),u(*),rps,theta2,pos,vel,acc,pi,dtr
      integer presval,isques,CRANK
      parameter(CRANK=2)
      common /radps/rps,theta2
      common /pidtr/pi,dtr

      acc = 0d0
      vel = rps
      pos = theta2*dtr + t*vel

      call SDGETPRES(CRANK,1,presval)
      if (presval .ne. 0) then
          call SDPRESPOS (CRANK,1,pos)
          call SDPRESVEL (CRANK,1,vel)
          call SDPRESACC (CRANK,1,acc)
      end if

      return
      end
```

Once again, the user wisely checks to see if prescribed motion is desired by using the SD/FAST-supplied routine `SDGETPRES()`. Note that the prescribed motion is computed exactly correct analytically. Integration of acceleration to give velocity and another integration to give position may drift somewhat from the analytical results (although not when the acceleration is a constant as it is in this case). Providing position and velocity information allows tracking of any errors, and can be used directly to prevent drift if constraint stabilization is enabled (see Reference Section R4.2). This information is also used during assembly and initial velocity analysis to bring the initial conditions into line with the desired prescribed positions and velocities.

T4.2.7 Exercise 1: Assembly and Initial Velocity Analysis

At this point we are ready to show how to perform an assembly and initial velocity analysis. Since we shall be using the assembly and initial velocity analyses several times, we have written these analyses into the routine `initconds()`:

```
c  initconds
c
c  Set up initial conditions.  Initializes crank angle to passed-in
c  theta2 (degrees), and crank velocity to passed-in rps (rad/s).  Pass
c  in y with an initial guess to control assembled branch and improve
c  convergence speed.  On return, y is a fully compatible state vector
c  unless an error message has been printed out.

       subroutine initconds(t,theta2,rps,y)
       integer NQ,NU,NEQ,CRANK
       parameter(NQ=5,NU=5,NEQ=NQ+NU,CRANK=2)
       real*8 t,theta2,rps,y(NEQ),ctol,pi,dtr
       common /pidtr/pi,dtr
       integer lock(NU),fcnt,err,SDINDX

c lock the crank angle and rate to the passed-in values
       data lock/0,1,0,0,0/
       ctol = 1e-7

       y(SDINDX(CRANK,1)) =  theta2*dtr
       call SDASSEMBLE (t,y,lock,ctol,500,fcnt,err)
       if( err .ne. 0) write(6,*) 'assembly failed'

       y(NQ+SDINDX(CRANK,1)) = rps
       call SDINITVEL(t,y,lock,ctol,500,fcnt,err)
       if( err .ne. 0) write(6,*) 'velocity analysis failed'

       return
       end
```

This routine allows the system to be assembled for any desired crank angle, `theta2`, which is locked at the passed-in value. Note that the initial configuration we show in Figure T4-2 is good for assembling the mechanism only when θ_2 is near 90°. For other values of θ_2, we will pass in configurations close to the desired assembled configuration (see Exercise 2).

The assembly analysis is performed using a call to `SDASSEMBLE()` where the parameters passed are time `t`, the initial guess state vector `y`, the `lock` vector indicating that the crank is locked at its current position, the constraint tolerance `ctol` indicating that the allowable error in assembling the joints is 1×10^{-7} meters and radians, a limit of 500 function calls, and an error variable that returns 0 if the assembly is successful. Details on the routine `SDASSEMBLE()` can be found in Reference Section R18.1.

The initial velocity analysis is performed *after* the assembly analysis by using a call to `SDINITVEL()` where the parameters passed are identical to those passed for `SDASSEMBLE()`, except that the velocities *u* in the state vector `y` are used as the initial guess for the velocities and the `lock` vector applies to locking the velocities. The *q*'s are assumed all locked. Details on the routine `SDINITVEL()` can be found in Reference Section R18.2. Next we set up the main program and perform the first assembly and velocity analyses.

The main program setup code used by all the later analyses is:

```
c  quickret
c
c  This is a set of analyses performed on a quick-return mechanism
c  used as a cutter in a machine tool discussed in Tutorial 4 of the
c  SD/FAST User's Manual, version B.1.0.  See quickret.sd for a description
c  of the mechanism.
c
      program quickret

c Parameters of the quick return model, from the quickret_info file.
      integer NQ,NU,NEQ,NJNT,NC
      parameter(NQ=5,NU=5,NEQ=NQ+NU,NJNT=7,NC=11)

c Parameters for use with the root finder. NF=number of functions,
c NV=number of variables, the rest are sizes of work arrays.
      integer NF,NV,JWSZ,DWSZ,RWSZ,IWSZ
      parameter(NF=1,NV=1,JWSZ=NF*NV, DWSZ=2*(NF+NV)*(NF+NV),
     1          RWSZ=9*(NF+NV), IWSZ=4*(NF+NV))

c Body numbers (also the joint number of the associated inboard joint).
      integer GROUND,TOOL,CRANK,SLEEVE,COUPLER,ROCKER
      parameter(GROUND=0,TOOL=1,CRANK=2,SLEEVE=3,COUPLER=4,ROCKER=5)

      real*8 t,pi,dtr,y(NEQ),dy(NEQ),yinit(NEQ)
      real*8 rps,ctol,tol,qdot(NQ),udot(NU),vec(3)
      integer fcnt,err,i,i1,flag,SDINDX
      real*8 r1,r3,theta2,theta4,theta5
      real*8 w(3),r1d,r3d,theta4d,theta5d
      real*8 alpha(3),r1dd,r3dd,theta4dd,theta5dd
      real*8 force(NJNT,3),torque(NJNT,3)
      real*8 fx,fy,torq,crx,cry
      real*8 theta2d,theta2dd,dt,delta,parm(3)
      real*8 com(3),pos(3),vel(3)
      real*8 jw(JWSZ),dw(DWSZ),rw(RWSZ),fret(NF)
      integer ilock,iw(IWSZ)
      integer rank,multmap(NC)
      real*8 mults(NC)
      common /radps/rps,theta2
      common /pidtr/pi,dtr
      common /icond/ yinit
      data com/3*0d0/
      external resid

      pi = acos(-1d0)
      dtr = pi/180d0
```

A nice way to use system parameters such as the number of *q*'s, `NQ`, or body numbers such as `CRANK=2` is to write them as parameters (instead of integer numbers) to improve the readability of the code. The variables used are self-explanatory or will be explained as needed. The common blocks are used to pass some variables between routines. The external function `resid` is used in Exercises 5 and 6.

Next `SDINIT()` is called to initialize the SD/FAST-generated code. Then the configuration for the initial guess shown in Figure T4-2 is generated and placed into `yinit`.

```
        call SDINIT

c This is an initial guess at the assembled configuration, to make sure
c we get into the right branch.

        do 10 i=1,NEQ
 10         yinit(i) = 0d0
        yinit(SDINDX(CRANK,1))  =  90d0*dtr
        yinit(SDINDX(SLEEVE,1)) = -60d0*dtr
        yinit(SDINDX(ROCKER,1)) = 135d0*dtr
```

The first exercise is to assemble the system with prescribed crank motion with the initial position $\theta_2 = 90°$ (`theta2`) and $u_2 = 20$ rad/sec (`rps`). Prescribed motion is turned on using a call to `SDPRES()`. The initial guess is passed from `yinit` to `y`. The assembly and velocity analyses are performed by calling `initconds()`.

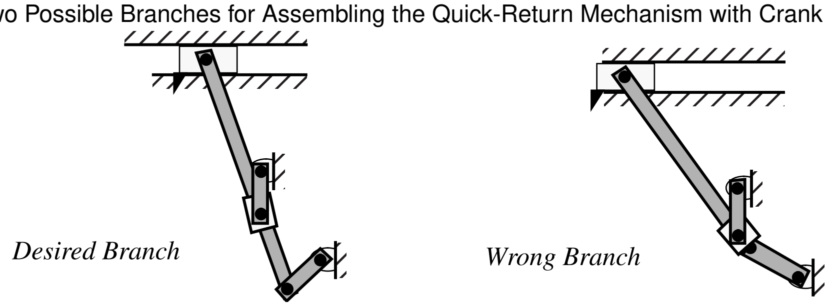
```
c EXERCISE #1: ASSEMBLY and INITIAL VELOCITY ANALYSIS

c Start the system out with crank angle at ninety degrees, crank
c angular velocity of 20 rad/sec.

c Start with prescribed motion ON at crank.
      call SDPRES(CRANK,1,1)

      do 15 i = 1,NEQ
 15       y(i) = yinit(i)

      write(6,*) ''
      write(6,*) 'EXERCISE 1: ASSEMBLY and INITIAL VELOCITY ANALYSIS'
      write(6,*) ''

      t = 0d0
      rps = 20d0
      theta2 = 90d0


c do assembly and velocity analysis
          call initconds(t,theta2,rps,y)

c check if position and velocity errors are below ctol
      call sdperr(perr)
      write(6,*) 'Position errors for the 11 constraints:'
      write(6,*) perr
      call sdverr(perr)
      write(6,*) 'Velocity errors for the 11 constraints:'
      write(6,*) verr
      write(6,*) ''
```

The reported position and velocity errors are well below the desired `ctol` of 1×10^{-7}:

```
EXERCISE 1: ASSEMBLY and INITIAL VELOCITY ANALYSIS

Position errors for the 11 constraints:
  0.  0.  0.    1.2007747332126D-11   -6.4223334224518D-12  0.  0.  0.
   -2.6610368910555D-12   -4.8272844055397D-12  0.
Velocity errors for the 11 constraints:
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
```

Since no errors were reported by `initconds()`, the user is guaranteed that the position and velocity constraint errors are less than `ctol`. So checking here is just for added assurance. Next we can check the configuration to see if the correct branch was entered.

Figure T4-5 Two Possible Branches for Assembling the Quick-Return Mechanism with Crank at 90°



The following code segment prints out the information on the positions and orientations of the bodies to check the validity of the assembly analysis:

```
c verify that the mechanism assembled into the correct branch.

      write(6,*) 'Mechanism generalized coodinates (relative q''s)'
      write(6,*) ''
      write(6,100) 'TOOL DISPLACEMENT:',y(SDINDX(TOOL,1))
      write(6,100) 'CRANK ANGLE      :',y(SDINDX(CRANK,1))/dtr
      write(6,100) 'SLEEVE ANGLE     :',y(SDINDX(SLEEVE,1))/dtr
      write(6,100) 'COUPLER EXTENSION:',y(SDINDX(COUPLER,1))
      write(6,100) 'ROCKER ANGLE     :',y(SDINDX(ROCKER,1))/dtr
      write(6,*) ''

      call SDPOS(TOOL,com,pos)
      write(6,100) 'TOOL  LOCATION:',pos
      call SDPOS(CRANK,com,pos)
      write(6,100) 'CRANK LOCATION:',pos
      call SDPOS(SLEEVE,com,pos)
      write(6,100) 'SLEEVE LOCATION:',pos
      call SDPOS(COUPLER,com,pos)
      write(6,100) 'COUPLER LOCATION:',pos
      call SDPOS(ROCKER ,com,pos)
      write(6,100) 'ROCKER LOCATION:',pos
```

Checking a few key results in the printout show that the correct branch was entered:

```
Mechanism generalized coodinates (relative q's)

  TOOL DISPLACEMENT:       0.0620
  CRANK ANGLE      :      90.0000
  SLEEVE ANGLE     :     -79.9487
  COUPLER EXTENSION:       0.1745
  ROCKER ANGLE     :     135.9435

     TOOL  LOCATION:       0.0620        0.0000        0.0000
     CRANK LOCATION:       0.0000        0.2500        0.0000
    SLEEVE LOCATION:       0.0000        0.3500        0.0000
   COUPLER LOCATION:       0.0158        0.2609        0.0000
    ROCKER LOCATION:      -0.0652        0.4859        0.0000
```

The crank angle is at the locked angle of 90° as desired. The tool displacement, sleeve angle, coupler extension, and rocker angle are all close to the desired branch shown in Figure T4-5. The center of mass locations of each body also match those of the desired branch. Thus, a rough sketch of the assembled mechanism, or better, a graphical display of the system, can be used to ensure that the correct branch was entered. Usually, an incorrect branch is immediately apparent.

A check of the velocities can also give insight into the assembled system:

```
        write(6,*) ''
        call SDVEL(TOOL,com,vel)
        write(6,100) 'TOOL  VELOCITY:',vel
        call SDVEL(CRANK,com,vel)
        write(6,100) 'CRANK VELOCITY:',vel
        call SDVEL(SLEEVE,com,vel)
        write(6,100) 'SLEEVE VELOCITY:',vel
        call SDVEL(COUPLER,com,vel)
        write(6,100) 'COUPLER VELOCITY:',vel
        call SDVEL(ROCKER ,com,vel)
        write(6,100) 'ROCKER VELOCITY:',vel

100     format(a20,3(f14.4))
        write(6,*) ''
        pause 'DONE WITH EXERCISE #1'
```

The resulting velocities of the mass centers are:

```
        TOOL  VELOCITY:       -4.8151        0.0000        0.0000
        CRANK VELOCITY:        0.0000        0.0000        0.0000
       SLEEVE VELOCITY:       -2.0000        0.0000        0.0000
      COUPLER VELOCITY:       -2.7803        0.3607        0.0000
       ROCKER VELOCITY:       -0.3728        0.3607        0.0000

 PAUSE: DONE WITH EXERCISE #1
```

The tool is clearly retracting. Since the crank is really a flywheel, its mass center velocity will always be zero. The sleeve is attached to the tip of the crank with a velocity of:

$$\mathbf{v} = \omega \times \mathbf{r} = 20\hat{\mathbf{n}}_3 \times 0.1\hat{\mathbf{n}}_2 = -2.0\hat{\mathbf{n}}_1 \qquad \textbf{Eqn. T2.6}$$

which matches the computed result. Other velocities, such as at the ends of the coupler, could also have been checked for correctness.

Key Idea

The main point of this exercise was to not only set up the assembly and velocity analysis, but to point out that the final assembled system should be started with an initial guess “close” to the desired configuration and then carefully checked for assembly into the desired branch. Fortunately, it is usually obvious when an incorrect branch has been entered—but check it!

T4.3 Suite of Typical Mechanism Analyses

This section consists of five more exercises that take you through some typical mechanism analyses. These five exercises are:

Ex #2: Inverse Dynamics: Generate a specified motion and applied load at the tool (all prescribed so that the system has *zero* degrees of freedom) and find the torque applied at the crank required to generate that motion. This would be useful in sizing a motor to drive this mechanism.

Ex #3: Mechanical Advantage: Compute the ratio of cutter force to crank torque. This is a standard mechanism kinematic analysis to find such things as regions of possible binding, approximate motor sizing, etc.

Ex #4: Dynamic Analysis: Run dynamic analysis with actual motor model. This is used to check the actual timing of the cutting stroke for the selected motor.

Ex #5: Design Study A: The chosen motor may not be able to drive the mechanism back to the same crank velocity as it started with. We could change the motor size to match the desired initial crank velocity, or, since motors come in discrete sizes, simply run a suite of dynamic analyses with a range of starting velocities to find the actual starting velocity for that motor driving the mechanism (after initial transients have died out). Then we check if the stroke cycle time meets our requirements.

Ex #6: Design Study B: Design Study A is repeated using just *one* pass through the nonlinear root finder supplied by SD/FAST. This advanced technique shows how to perform these types of studies much more quickly.

Exercises 1 through 4 are essential for learning to set up mechanism analyses using SD/FAST. Exercises 5 and 6 discuss more advanced features.

T4.3.1 Exercise 2: Inverse Dynamics

An inverse dynamics analysis computes the loads given a desired motion, as opposed to a (forward) dynamic analysis which computes the motion given the loads. Inverse dynamics is commonly used to compute feed-forward commands to obtain a desired motion, get a rough idea of actuator sizing, determine bandwidth requirements of a feedback controller to follow a desired trajectory, etc.

For example, a robot may have its tip commanded to follow a specified path in a high-speed, repetitive manufacturing process. An inverse dynamics analysis would give you the feed-forward command history for each motor in the robot. The command history could be computed off-line and then stored for use by the robot in real-time. For the quick-return mechanism, we desire to compute the motor torque applied to the crank to give the desired tool motion.

There are two ways (at least) to use SD/FAST to compute inverse dynamics. The method that first comes to mind is to run an actual forward dynamic analysis with all motion prescribed (zero DOF for system), and then read back the motor torque required to create the motion. The second method would be to re-assemble the mechanism for a range of crank angles that cover the entire 360° crank cycle, say, every 10°; and compute dynamic equilibrium at each angle of the crank. The second method may appear to be inefficient, but an assembly and velocity analysis is actually quite fast and can be more computationally efficient than integrating the equations of motion.

For this example, we shall use the second method of re-assembling every 10° and computing dynamic equilibrium (Integration of the equations of motion is shown in Exercise 4, for forward dynamics. The identical code will work for inverse dynamics if all the motion is prescribed.). Each new re-assembly uses the previous configuration to ensure that the initial guess for the assembly is very “close.” This makes the new assembly analysis very fast using only a few derivative evaluations, as opposed to the numerous derivative evaluations required to integrate the equations of motion.

There is one caveat in using this method. If the mechanism is being assembled near a bifurcation configuration (or what *would* be a bifurcation configuration if link lengths were slightly adjusted, such as making a crank-rocker close to a double-crank), then the system may re-assemble into an undesired branch. If you suspect this may be a problem, then we suggest you use the first method of running a dynamic analysis with all the motion prescribed.

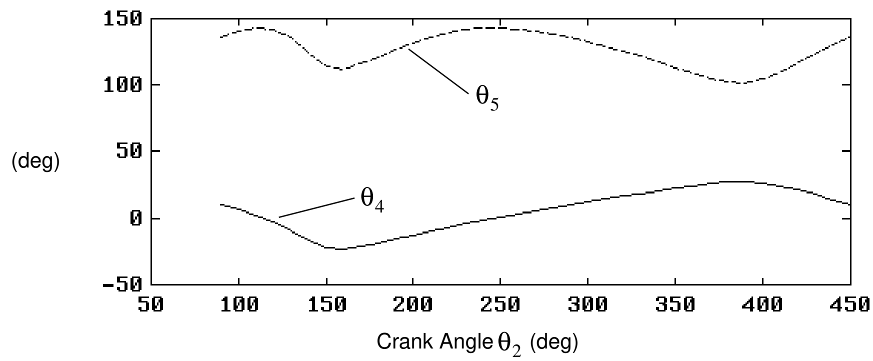
The following code segment implements the inverse dynamics for each $10°$ by first calling `initconds()` to perform a new assembly and velocity analysis, then calling `sduforce()` to apply system loads . Finally, `SDDERIV()` is called to obtain the system derivatives and corresponding loads Note that `sdumotion()`, used to enforce prescribed motion, is already called in `SDINITVEL()`.

```
c EXERCISE #2: INVERSE DYNAMICS

c Rotate crank at 20 rad per sec and compute crank torque required
c to maintain this speed while load is applied to cutter.

c Start with prescribed motion ON at crank.
       call SDPRES(CRANK,1,1)

       do 20 i = 1,NEQ
 20        y(i) = yinit(i)

       write(6,*) ''
       write(6,*) 'EXERCISE 2: INVERSE DYNAMICS ANALYSIS'
       write(6,*) ''

       t = 0d0
       rps = 20d0
       theta2 = 90d0

c Move theta2 (crank angle) 360 degrees, 10 degrees per step.
       do 30 i1 = 1,37

c  do assembly and velocity analysis
           call initconds(t,theta2,rps,y)

c  apply forces and motions and compute resulting derivatives
           call sduforce(t,y,y(NQ+1))

           call SDDERIV(qdot,udot)
```

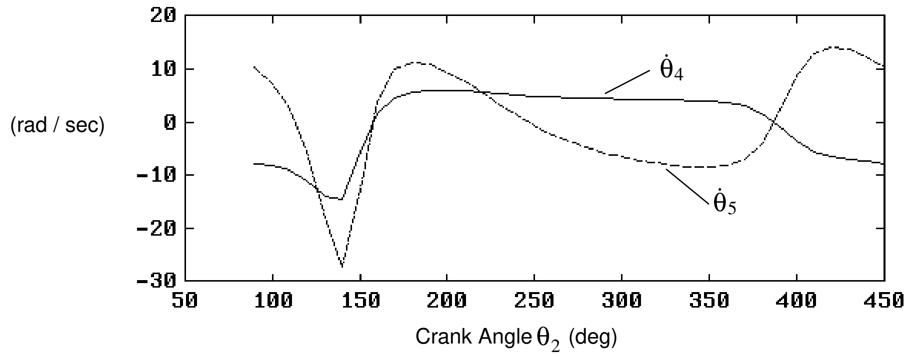
The next code segment computes the position, velocity, and acceleration for each of the five coordinates of interest: tool displacement r_1, crank angle θ_2 (already prescribed), sleeve displacement r_3, coupler angle θ_4, and rocker angle θ_5. Note the exclusive use of parameters to identify bodies, instead of numbers, and direct calls to SD/FAST-generated utility and indexing routines such as `SDANGVEL()` and `SDINDX()` to reduce the chance of programming errors.

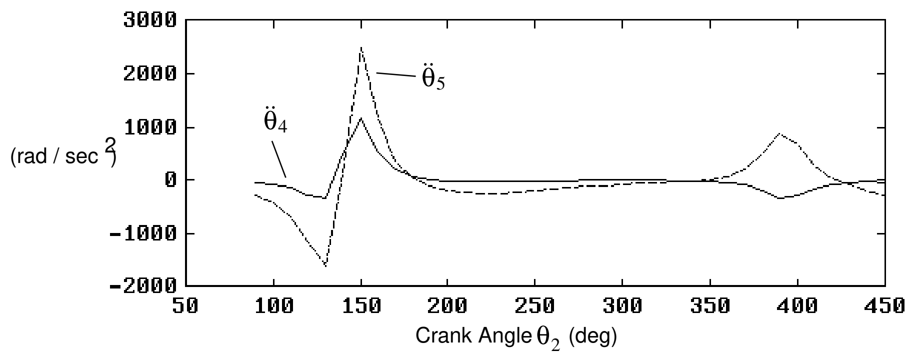
```
c calculate outputs for inverse dynamics analysis
c theta2  - crank angle (input) (degrees)
c theta4  - angle between COUPLER and global Y (degrees)
c theta5  - angle between ROCKER and global Y (degrees)
c theta2d,theta4d,theta5d - angular rates in radians/s
c theta2dd,theta4dd,theta5dd - angular accelerations in rad/s**2
c r1      - TOOL displacement from O
c r3      - distance between pts A and B
c torq    - driving torque required on crank
c crx,cry - x,y components of crank-ground reaction (ground frame)
c fx,fy   - x,y components of rocker-ground reaction (gnd frame)
c rank    - rank of the constraint matrix. if rank < NC, then redundancies

          theta2d  = y(NQ+SDINDX(CRANK,1))
          theta2dd = udot(SDINDX(CRANK,1))

          theta4 = theta2 + y(SDINDX(SLEEVE,1))/dtr
          call SDANGVEL(COUPLER,w)
          theta4d = w(3)
          call SDANGACC(COUPLER,alpha)
          theta4dd = alpha(3)

          theta5 = y(SDINDX(ROCKER,1))/dtr
          call SDANGVEL(ROCKER,w)
          theta5d = w(3)
          call SDANGACC(ROCKER,alpha)
          theta5dd = alpha(3)

          r1   = y(   SDINDX(TOOL,1))
          r1d  = y(NQ+SDINDX(TOOL,1))
          r1dd = udot(SDINDX(TOOL,1))

          r3   = y(   SDINDX(COUPLER,1))
          r3d  = y(NQ+SDINDX(COUPLER,1))
          r3dd = udot(SDINDX(COUPLER,1))
```

Plots for each of the positions, velocities, and accelerations were obtained by processing the output of this example program through a commonly available plotting package.

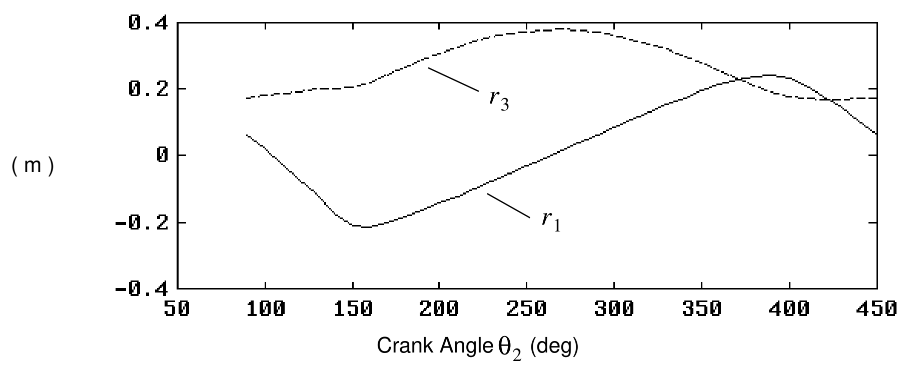


Note that the coupler angle θ_4 and rocker angle θ_5 rock a few tens of degrees as expected. The rates and accelerations change slowly during the cutting stroke, and are very large at the turnaround points.

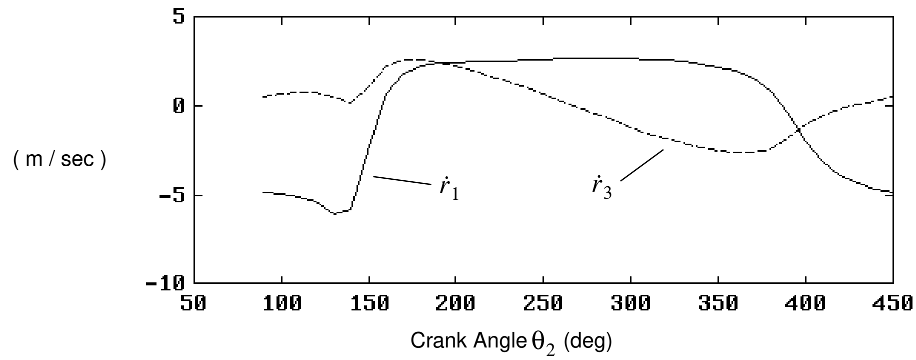


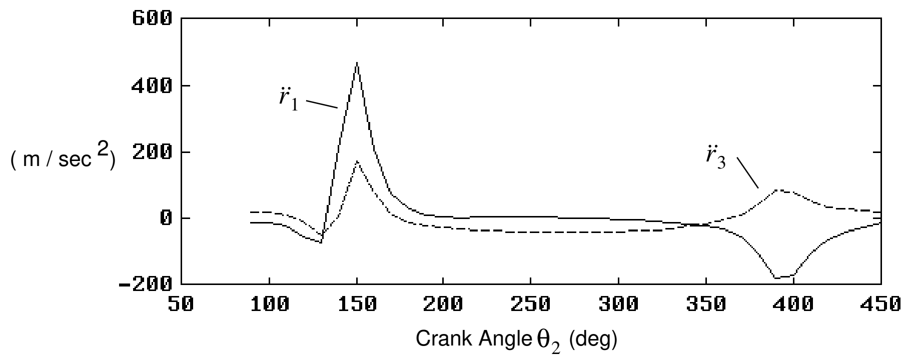


The cutter stroke r_1 is a smooth ramp during the cut as expected:



Again, the rates and accelerations are low during the forward and backward strokes, whereas they are large near the turnaround points:





The next code segment calls `SDGETHT()` to find the hinge torque applied at the crank to produce the prescribed motion, and `SDREAC()` to find the reaction loads at the joints. The reaction loads are transferred from their local body frames to the ground frame using `SDTRANS()`.

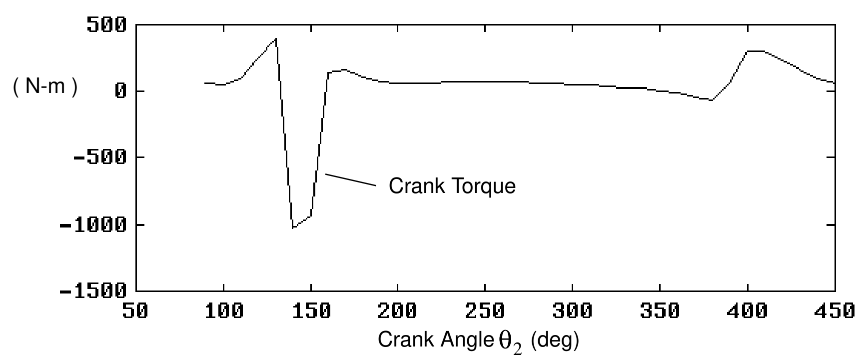
```
call SDGETHT(CRANK,1,torq)
call SDREAC(force,torque)

vec(1) = force(CRANK,1)
vec(2) = force(CRANK,2)
vec(3) = force(CRANK,3)
call SDTRANS(CRANK,vec,GROUND,vec)
crx = vec(1)
cry = vec(2)

vec(1) = force(ROCKER,1)
vec(2) = force(ROCKER,2)
vec(3) = force(ROCKER,3)
call SDTRANS(ROCKER,vec,GROUND,vec)
fx = vec(1)
fy = vec(2)
```

Finally, the rank of the constraint matrix is computed using a call to `SDMULT()`, and all the desired variables are both printed and stored for plotting. You will find that the constraint matrix always has a rank of 5, meaning that 6 of the 11 constraints are *redundant*.

However, the in-plane constraints are not redundant and thus the in-plane reaction loads are unique. Ten degrees is added to the crank angle and the loop rerun until a full revolution has occurred.

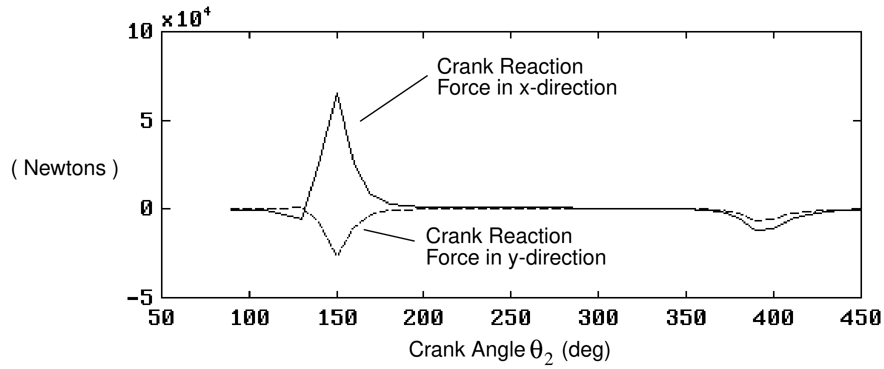
```
      call SDMULT(mults,rank,multmap)

      write(6,105)  theta2,theta4,theta4d,theta4dd
      write(61,105) theta2,theta4,theta4d,theta4dd
      write(6,105)  theta2,theta5,theta5d,theta5dd
      write(62,105) theta2,theta5,theta5d,theta5dd
      write(6,105)  theta2,r1,r1d,r1dd
      write(63,105) theta2,r1,r1d,r1dd
      write(6,105)  theta2,r3,r3d,r3dd
      write(64,105) theta2,r3,r3d,r3dd
      write(6,105)  theta2,torq,crx,cry
      write(65,105) theta2,torq,crx,cry
      write(6,105)  theta2,fx,fy
      write(66,105) theta2,fx,fy
      write(6,106)  rank
      write(99,106) rank
      write(6,*) ''
105   format(5(f14.3))
106   format(i10)

      theta2 =  theta2 + 10d0
 30   continue
      pause 'DONE WITH EXERCISE #2'
```

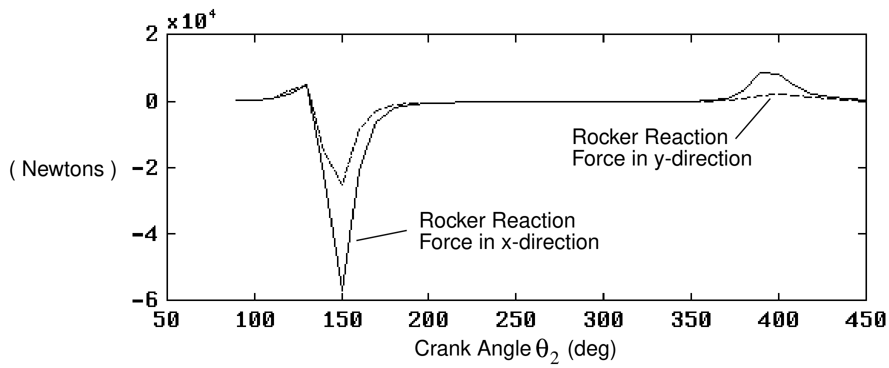
The hinge torque `torq` applied to the crank as a function of the crank angle to make the mechanism follow the prescribed motion is plotted next:



Note that the crank torque is very large near the turnaround points. If this level of torque is not available from the motor, two solutions can be tried. First we can run a forward dynamics solution with the actual motor model implemented and check if the deviations from desired motion are acceptable. Second, we could increase the flywheel inertia to smooth out the torque requirements. In this case, we will accept the performance as is (see Exercise 4).

The in-plane reaction forces at the crank (`crx` and `cry`) and the in-plane reaction forces at the rocker ground attach point (`fx` and `fy`) are plotted below. The reactions are small throughout most of the forward and backward strokes, and large during turn-around as expected.





T4.3.2 Exercise 3: Mechanical Advantage

Mechanical advantage is usually computed as a *kinematic* function of the ratio of the differential change in the *input position* of the mechanism (the crank in this case) over the differential change in *output position* (the tool). In terms of the loads required to hold the mechanism in equilibrium, the mechanical advantage is the ratio of the o*utput load* over the *input load*. (Note the input/output ratio is *reversed*).

Using SD/FAST, we can go one step further, and find the mechanical advantage for the mechanism in *dynamic* motion, rather than just in static equilibrium. That is, take the ratio of output over the input loads while the system is in motion. This is perhaps closer conceptually to the “real” meaning of mechanical advantage for a mechanism that operates near a steady state velocity (crank angular rate). We shall compute “static” mechanical advantage in this exercise.

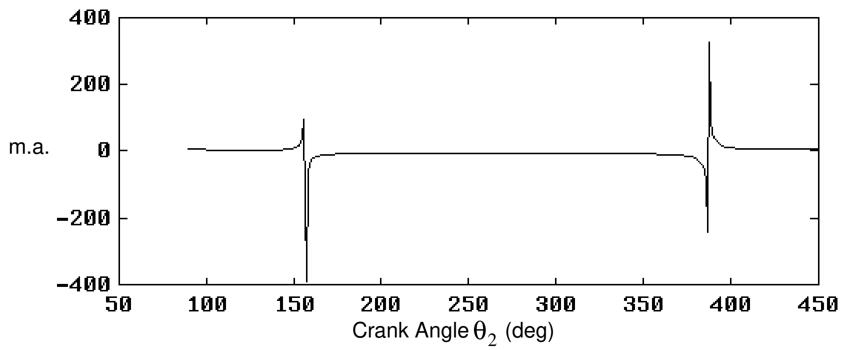
The code for implementing the mechanical advantage analysis is almost identical to the code for the inverse dynamics analysis. We shall use the re-assembly method instead of forward dynamic analysis. This time, the analysis is performed with the crank rotation rate prescribed at 0 rad/sec to produce a “static” analysis at each configuration. For this analysis, we shall loop every degree instead of every 10 degrees.

```
c EXERCISE #3: MECHANICAL ADVANTAGE

c Step crank from 90 deg through 450 deg and compute crank torque required
c to resist load applied to cutter.

c start with prescribed motion ON at crank

      call SDPRES(CRANK,1,1)

      do 40 i = 1,NEQ
 40       y(i) = yinit(i)

      write(6,*) ''
      write(6,*) 'EXERCISE 3: MECHANICAL ADVANTAGE'
      write(6,*) ''

      rps = 0d0
      theta2 = 90d0
      write(6,108)'theta2','torque','mech advtg.'
      do 50 i1 = 1,361

c  do assembly and velocity analysis
          t = 0d0
          call initconds(t,theta2,rps,y)

c  now compute derivatives
          call sduforce(t,y,y(NQ+1))
          call sdumotion(t,y,y(NQ+1))
          call SDDERIV(qdot,udot)
```

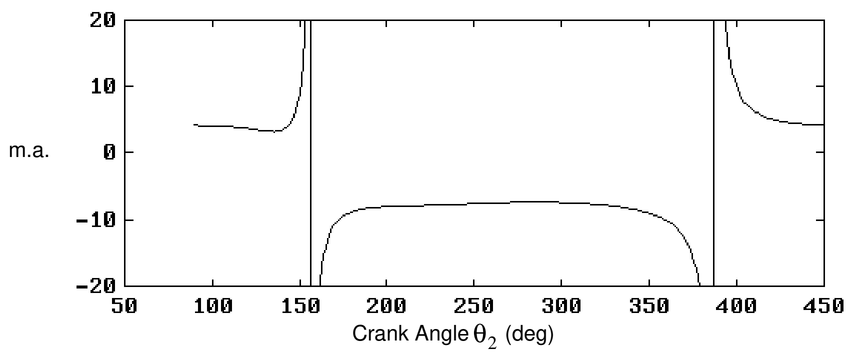
Mechanical advantage is computed using the known constant applied force at the tool. (Note that for this analysis the force is applied throughout the forward *and* backward stroke since `sduforce()` is coded to apply the force whenever the velocity is less than or *equal* to zero. Note also that any force could have been used to compute mechanical advantage—most typically a unit force. However, we already had a 450N force applied for use in other analyses, so we’ll use that.)

```
c calculate outputs for mechanical advantage analysis
c torq = driving torque required on crank
c m.a. = (cutter force)/(crank torque)

          call SDGETHT(CRANK,1,torq)

          write(6,105)  theta2,torq,-450d0/torq
          write(67,105) theta2,torq,-450d0/torq

          theta2 =  theta2 + 1d0
 50   continue
      pause 'DONE WITH EXERCISE #3'
```

The resulting plot of mechanical advantage is:



The mechanical advantage approaches infinity near the two turnaround points. A close-up of the constant region shows a mechanical advantage of about 10 for the forward cutting stroke. A constant mechanical advantage over the forward stroke indicates a well designed mechanism that should supply the cutter with a fairly constant force.



T4.3.3 Exercise 4: Dynamic Analysis

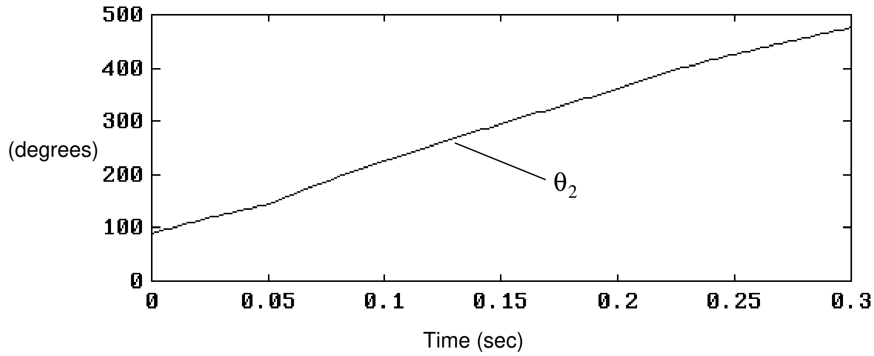
In the dynamic analysis we shall use the motor model in `sduforce()` instead of prescribing the motion. Before we begin the dynamic analysis, we first turn off prescribed motion and perform an assembly and velocity analysis to place the mechanism back in our initial configuration with a crank velocity of 20 rad/sec.

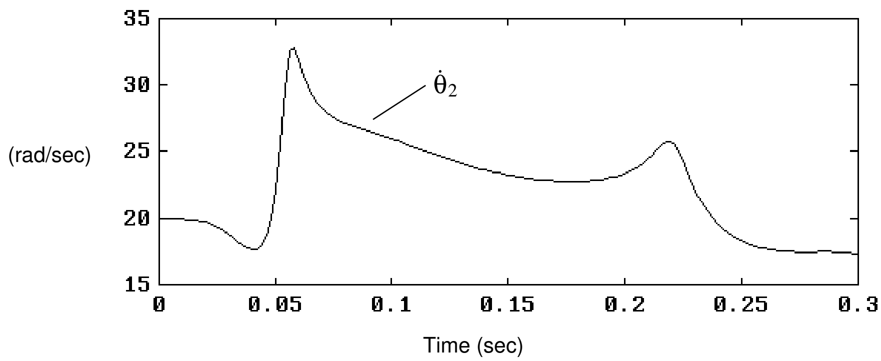
```
c EXERCISE #4: DYNAMIC ANALYSIS

c Start crank at 20 rad per sec and compute time history, using
c motor torque vs. rpm curve.

       write(6,*) ''
       write(6,*) 'EXERCISE 4: DYNAMIC ANALYSIS'
       write(6,*) ''
       write(6,*) 'initial crank angular velocity: 20 rad/sec'
       write(6,108) 't', 'theta2', 'theta2d','theta2dd'
108    format(5(a14))

c turn prescribed motion  OFF  at crank
       call SDPRES(CRANK,1,0)

c  do assembly and velocity analysis
       do 60 i = 1,NEQ
 60        y(i) = yinit(i)
       t = 0d0
       rps = 20d0
       theta2 = 90d0
       call initconds(t,theta2,rps,y)
```

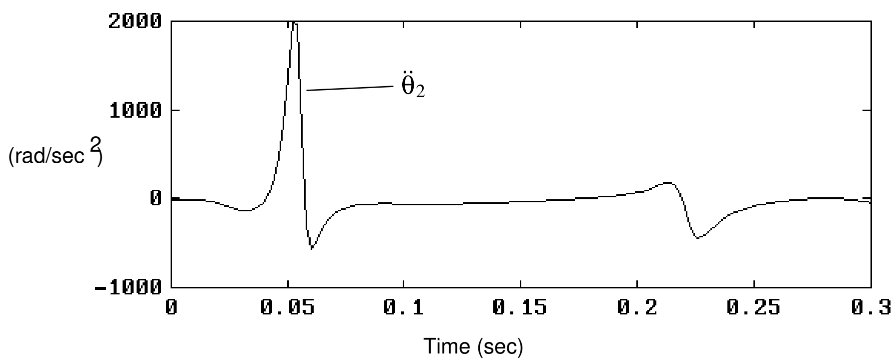
Next we set up the desired step size, tolerances, etc. Note that we call `SDMOTION()` once at the beginning with `dt=0` just so that we have acceleration information for plotting at the start time.

```
c  now perform a motion analysis

       flag = 1
       dt   = 0.002d0
       ctol = 1d-4
       tol  = 1d-5

c  call once with dt=0 just to evaluate derivatives at t=0

       call SDMOTION(t,y,dy,0d0,ctol,tol,flag,err)
       if (err .ne. 0) write(6,*) 'at t = ',t,'error=',err

c  simulate for long enough to cover a full cycle
       do 70 i1 = 1,151

c calculate outputs for dynamic analysis
c theta2 - crank angle
c theta2d - crank angular velocity
c theta2dd - crank angular acceleration

           theta2   = y(   SDINDX(CRANK,1))/dtr
           theta2d  = y(NQ+SDINDX(CRANK,1))
           theta2dd = dy(NQ+SDINDX(CRANK,1))
           write(6,105)  t,theta2,theta2d,theta2dd
           write(70,105) t,theta2,theta2d,theta2dd

           call SDMOTION(t,y,dy,dt,ctol,tol,flag,err)
           if (err .ne. 0) write(6,*) 'at t = ',t,'error=',err
 70    continue
       pause 'DONE WITH EXERCISE #4'
```

For this analysis we shall only plot the position, velocity, and acceleration for the crank angle.







Note that the crank velocity at the end of one cycle is well below the velocity at the start. This means that the motor is not powerful enough to sustain the desired rate. If we allow the system to integrate long enough, the natural damping generated by the back EMF in the motor (as represented by the torque-rpm curve) will dampen out the transients, and a steady state cycle at a lower average crank velocity will emerge. The final two exercises show other methods of finding the steady state cycle.

T4.3.4 Exercise 5: Design Study A

In this design study we shall run a suite of dynamic analyses that sweep through initial crank velocities to find one that just matches the crank velocity after one cycle. This will give us the steady cycle motion for the given motor.

Since we shall be performing several dynamic analyses, we wrote routine `resid()` to perform the analysis and return the difference in initial and final crank velocity. The reason we named the routine "resid" is that we shall also use it in the next exercise in which it is called by a nonlinear root finder to find the "residual" error to reduce. The variables passed to `resid()` are `rps`, the initial crank velocity, and `parm`, a vector of integration control parameters. The variable returned is the difference in the initial and final crank velocities, `delta`.

```
c resid
c
c Perform a single cycle of the mechanism, and then return the amount
c by which the initial angular velocity of the crank (rps) differs
c from the final angular velocity.  Accuracy is controlled by parm.

      subroutine resid(rps,parm,delta)
      integer NQ,NU,NEQ,CRANK,err,i,flag,SDINDX
      parameter(NQ=5,NU=5,NEQ=NQ+NU,CRANK=2)
      real*8 rps,parm(3),delta,y(NEQ),yinit(NEQ),dy(NEQ)
      real*8 t,dt,pi,dtr,th,thd,w,theta2,theta2d,ctol,tol
      common /pidtr/pi,dtr
      common /icond/yinit

c Get constraint and integration tols, and maximum step size dt.
      ctol = parm(1)
      tol  = parm(2)
      dt   = parm(3)
```

An assembly and velocity analysis is performed before each dynamic analysis by calling `initconds()` as usual:

```
c  do assembly and velocity analysis
      do 10 i = 1,NEQ
 10       y(i) = yinit(i)
      t = 0d0
      theta2 = 90d0
      theta2d = rps
      call initconds(t,theta2,theta2d,y)
      write(6,*) 'initial conditions:',t,theta2d
```

Next the dynamic analysis is started and proceeds until the crank angle just passes its initial position (plus 360 degrees). Stall conditions are also checked just in case the initial velocity is not large enough to allow the mechanism to complete a cycle.

```
c now integrate until we complete one revolution
       flag = 1
       th = theta2
100    continue
           call SDMOTION(t,y,dy,dt,ctol,tol,flag,err)
           if (err .ne. 0) write(6,*) 'at t = ',t,' error=',err
           theta2  = y(   SDINDX(CRANK,1))/dtr
           theta2d = y(NQ+SDINDX(CRANK,1))

c          detect stalled condition
           if (th .ge. theta2) then
               write(6,*) 'not going around'
               stop
           end if

           if (theta2 .lt. 450d0) then
               th = theta2
               thd = theta2d
               goto 100
           end if
c      end integration loop
```

An interpolation is performed to provide a much better estimate of the final crank velocity just as the crank angle passes the start angle:

```
c the crank made a full revolution. Interpolate to compute
c crank angular velocity corresponding to one rev.

       w = (theta2d - thd)*(450d0 - th)/(theta2 - th) + thd
       delta = w - rps
       write(6,*) 'terminal velocity:', w, '   delta:', delta
       write(6,*) ''
       write(71,*) rps,delta

       return
       end
```

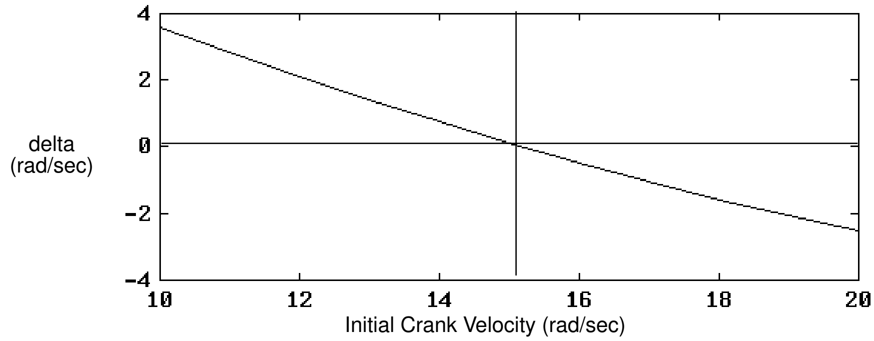
The design study in the main program starts with variable assignments, turning prescribed motion off so that the motor model is enabled, and performing an assembly and velocity analysis:

```
c EXERCISE #5: DESIGN STUDY

c We want the mechanism to operate in a steady cycle. This
c can only happen for a particular initial crank velocity.
c We will compute the crank terminal angular velocity (after one
c revolution) for various initial crank angular velocities.  We're
c looking for the initial crank velocity which produces an identical
c terminal crank velocity.

      write(6,*) ''
      write(6,*) 'EXERCISE 5: DESIGN STUDY'
      write(6,*) ''

c turn prescribed motion  OFF  at crank
      call SDPRES(CRANK,1,0)

c start with velocity at 10 and increase by 1 each trial
      rps = 10d0
      do 90 i1 = 1,11

c  do assembly and velocity analysis

          do 80 i = 1,NEQ
 80           y(i) = yinit(i)
          t = 0d0
          theta2 = 90d0
          call initconds(t,theta2,rps,y)
```

Next the parameters required by the `resid()` routine are loaded and `resid()` is called. The crank velocity is increased by one rad/sec and the analysis repeated.

```
c Now evaluate the amount by which the terminal angular velocity differs
c from the initial angular velocity.  This requires perfoming a simulation
c of one cycle.  `resid' calculates and prints out the values as we go.

          ctol = 1d-2
          tol  = 1d-3
          dt   = 0.01d0
          parm(1) = ctol
          parm(2) = tol
          parm(3) = dt
          call resid(rps,parm,delta)

          rps = rps + 1d0
 90   continue
      pause 'DONE WITH EXERCISE #5'
```

The plot of delta in crank velocity versus initial crank velocity shows that the initial crank velocity that yields an equivalent final crank velocity is approximately 15.2.



T4.3.5 Exercise 6: Design Study B

In this design study we repeat the analysis just performed, but using the nonlinear root finder supplied by SD/FAST to find the correct crank velocity quickly and more accurately. We start by turning prescribed motion off:

```
c EXERCISE #6: FIND LIMIT CYCLE INITIAL CONDITIONS

c We will repeat previous study, but this time we will use the
c SD/FAST root-finder to find the initial crank angular velocity which
c will initiate a steady period.

      write(6,*) ''
      write(6,*) 'EXERCISE 6: FIND LIMIT CYCLE INITIAL CONDITIONS'
      write(6,*) ''

c turn prescribed motion  OFF  at crank
      call SDPRES(CRANK,1,0)
```

Then we set up all the variables required to be passed to `SD2ROOT()`. Note that we used `SD2ROOT()` instead of `SDROOT()`. They are exactly the same routines, but an additional copy had to be created (by passing the prefix "`SD2`" to SD/FAST) because Fortran does not allow reentrant routines (`SD2ROOT()` calls `resid()` which calls `initconds()` which calls `SDASSEMBLE()` which calls `SDROOT()`). Check Reference Section R8.2 for complete details on how to use `SDROOT()`.

```
      rps = 10d0
      ctol = 1d-2
      tol  = 1d-3
      dt   = 0.01d0
      parm(1) = ctol
      parm(2) = tol
      parm(3) = dt
      ilock = 0
      call SD2ROOT(resid,rps,parm,NF,NV,0,ilock,tol,0d0,50,
     1             jw,dw,rw,iw,fret,fcnt,err)
      if (err .ne. 0) then
          write(6,*) 'failed to find solution, err=', err
      else
          write(6,*) 'correct initial speed:',rps
      endif
      pause 'DONE WITH EXERCISE #6'
```

The printed results of the last few calls to `resid()` show that a closer answer for the initial crank velocity that yields the same final velocity is 15.193:

```
initial conditions:  0.    15.198269041016
terminal velocity:    15.195555029652   delta:   -2.7140113635884D-03

initial conditions:  0.    15.198270041016
terminal velocity:    15.195555463802   delta:   -2.7145772143093D-03

initial conditions:  0.    15.193472702930
terminal velocity:    15.193455668639   delta:   -1.7034291374785D-05

correct initial speed:    15.193472702930
PAUSE: DONE WITH EXERCISE #6
```

Finally, a call to `SDPRINTERR()` is made to check for usage errors.

```
c END OF EXERCISES

c check for SD/FAST routine usage error

       call SDPRINTERR(6)

       end
```

Summary

- This tutorial covered the first example of using SD/FAST for a closed-loop system, namely, a common quick-return mechanism used for shaping parts.
- Closed-loop systems are created by joining branches of open-loop tree systems by using loop joints. The open-loop tree systems are created by “cutting” the loops at joints or inside bodies. The system is re-assembled by (1) laying out the parts into an easy-to-model *reference configuration* of your choice, (2) adjusting the parts by changing the hinge coordinates to get the system into its *initial configuration* which is “close” to the assembled configuration you desire, (3) perform an assembly analysis (routine supplied by SD/FAST) to bring the cut loops together, and (4) perform an initial velocity analysis (routine supplied by SD/FAST) to compute compatible velocities for the system (so it won’t fly apart).
- Special considerations for constrained systems such as closed-loop systems were discussed, including: (1) numerical stabilization of loop joints using Baumgarte’s method; how to recognize and avoid problems such as (2) mechanism bifurcations, (3) mechanism lockup, (4) assembly into an undesired branch, (5) incompatible constraints, (6) passive degrees of freedom, and (7) redundant constraints with non-unique reaction loads.
- A quick-return mechanism was modeled and assembled using routines supplied by SD/FAST. A reference configuration is chosen for which it is easy to write the system description and provides computational advantages. The reference configuration is adjusted to place the system “close” to the desired assembled configuration to avoid assembly into the wrong branch.
- A suite of mechanism analyses was performed on the quick-return mechanism, including (1) inverse dynamic analysis, (2) mechanical advantage, (3) motion analysis, and (4) a design study (performed two different ways) to find steady state cycle of the mechanism. The motion was studied both as prescribed crank motion and using an actual motor model.

SD/FAST Reference

This chapter contains the detailed reference material for the SD/FAST software package. Entries are arranged alphabetically. Each entry is intended to be technically rigorous, yet concise. The Table of Contents contains a compressed listing of all the entries and the Index contains extensive cross references between the Tutorials and this reference. The Tutorials and this Reference are also cross-referenced to help you quickly find needed information. Please read the following section on the conventions used to describe SD/FAST-generated code and its usage.

Reference Section Conventions

Many of the subsections in the Reference portion of this manual document subroutines and functions generated by SD/FAST. We use the general term "routines" to include both subroutines and value-returning functions.

Most routines are generated specifically for the system at hand, and are placed by SD/FAST into the Dynamics or Simplified Analysis files. Some routines are the same every time they are generated and thus are placed in the Library File, which need be generated only once. The routines are grouped in the reference section by function, rather than by the file in which they are placed.

Routines are documented formally here, by showing their declarations in both Fortran and K&R C (of course ANSI C compilers can also be used). When referring to an SD/FAST-generated routine by name, we write the name in upper case typewriter font and follow it by parentheses, for example `SDSTATE()`. When referring to a user-written routine, we use lower case typewriter font instead, for example `sduforce()`. Examples of user-written code are also shown in lower case, except that values declared in `parameter` statements and calls to SD/FAST-generated routines are shown in upper case.

Table R-1 Names for System Parameter Constants

Constant Name	Meaning
NBOD	no. bodies (also, no. of tree joints)
NJNT	total number of joints (tree+loop)
NQ	no. position coordinates in state (tree joints)
NU	no. rate coordinates in state (tree joints)
NLQ	no. position coordinates describing loop joints
NLU	no. rate coordinates describing loop joints
NC	total no. constraints defined
NUSERC	no. user constraints

When discussing array dimensions, we use the names shown in Table R-1 for the various system parameters. The values for these names can be obtained from the generated Information File. We strongly recommend that you use these names, rather than the numbers they represent, in your analysis code. In Fortran, this can be done with `PARAMETER` or `DATA` statements. In C, use the `#define` feature to define names. This technique eliminates many common errors which occur when a code has to be modified from its original specification, which is almost inevitable. It also facilitates development of a common framework which can be reused for different analysis tasks.

In Fortran, where an array of known length is passed as a parameter, that length is shown in the parameter declaration. This is not really legal Fortran, but serves to get the point across. For example, in the declaration

```
SUBROUTINE SDMULT(MULTS,RANK,MULTMAP)
   DOUBLE PRECISION MULTS(NC)
   INTEGER RANK,MULTMAP(NC)
```

the `MULTS` and `MULTMAP` parameter are both known to be of length `NC` (number of constraints), but that number is not passed in since the `SDMULT()` routine was generated with full knowledge of the array lengths. In the actual generated code, the arrays are simply declared to be the appropriate length. For example, if `NC`=7 the actual declaration for `SDMULT()` is

```
SUBROUTINE SDMULT(MULTS,RANK,MULTMAP)
   DOUBLE PRECISION MULTS(7)
   INTEGER RANK,MULTMAP(7)
```

C allows constants to be given names which are known throughout the entire file, unlike Fortran `PARAMETER` statements which are visible only in the current subroutine. (Many Fortran compilers also support a `#define` pre-processor which works the same way as in C. We recommend this over `PARAMETER` statements, but it is non-standard).

We show the array dimensions in C declarations assuming that the appropriate constants have been defined. For example, `SDMULT()` is declared in C as

```
sdmult(mults,rank,multmap)
     double mults[NC];
     int    *rank, multmap[NC];
```

Again, the actual declaration in the generated code simply has the constant name replaced by its value.

In Fortran, we use the common convention that array indices begin at 1 and range up to *n* for an array of length *n*. This applies to all enumerated items, such as body, joint and axis numbers. In C, all indices follow the C convention of beginning at 0, with maximum index *n*-1.

In both languages the ground “body” (named `$ground`) is given the body number one less than the first user-defined body. That is, in Fortran `$ground` is body number 0 and in C it is body number -1.

Floating point parameters are always shown declared in double precision since this is the most common (and recommended) usage. If SD/FAST has been run with the single precision option, the declarations should be considered to have `REAL` substituted for `DOUBLE PRECISION` in Fortran, and `float` substituted for `double` in C. We do not recommend use of the single precision option on computers whose single precision real numbers are less than 56 bits wide, especially for systems with constraints.

Some of the SD/FAST-generated routines must be called before other calls are valid. These are common-sense requirements, for example you cannot ask for the acceleration of a point until after the system derivatives have been calculated. This is discussed more fully in Section R3, and the complete list of routines and their ordering requirements is given in the SD/FAST Quick Reference Guide, Section Q4 and Section Q3.

R1 Analysis Types

SD/FAST can be used to perform a wide range of engineering analyses for multibody systems. Spacecraft dynamics are typically studied by simulation, which we generally refer to as *dynamic analysis* or *motion analysis*. Mechanism and machine dynamics may require additional types of analysis, such as *static analysis* which is used to find an equilibrium configuration. Quite often, a mechanism may be analyzed only at certain configurations of interest. Any mechanical system can benefit from *design studies* in which the effects of changing design parameters can be analyzed or optimized. SD/FAST offers the versatility to support almost every type of analysis that may be desired.

Below, we discuss several of the most common analyses done with SD/FAST. See the Tutorials for examples showing most of these analyses. This is not an exhaustive list — the analyses possible with SD/FAST are limited primarily by the imagination of the analyst.

R1.1 Dynamic (Motion) Analysis

Dynamic analysis is concerned with obtaining the response over time of a multibody system when driven by loads (forces and torques). The motion is a highly nonlinear function of the applied loads. The basic characteristic of dynamic analysis is that the system is subjected to loads, and some or all of the motion is unknown. Dynamic analysis is usually used to predict the behavior of a multibody system during transient conditions.

The motion of a multibody system is governed by equations called dynamical equations of motion. These equations can be formulated by any number of classical methods. Modern treatments of multibody dynamics give rise to formalisms that are well-suited for computer solution, because of the need to obtain numerical results. This is because the governing equations are too complicated to be solved by hand. The equations of motion comprise a set of differential equations, with algebraic constraint equations. The differential equations are an expression of physical laws (Newton's Laws of Motion), while the constraint equations take into account desired restrictions on the geometry of the system, or its motion. Closed loops of joint-connected bodies and prescribed motion are common types of constraints.

The computer model of the multibody system is organized to allow the system motion to be propagated in time. The motion of the system is represented by the system state vector. This consists of a set of coordinates and a set of velocity variables (see Section R20 for details). Knowledge of the state vector is sufficient to compute the position and velocity of every material point of the multibody system. The number of velocity variables in an unconstrained system is said to be the number of *degrees of freedom* (DOFs). Each independent constraint added to the system reduces the number of usable degrees of freedom by one. The difference between the number of degrees of freedom in an unconstrained system and the number of independent constraints on that system is called the system's *mobility.* This number is not necessarily constant as the system moves.

The motion of the multibody system is driven by three possible sources: initial conditions, applied loads, and prescribed motion. The applied loads can be functions of the system state or time. There are three possible types of loads: forces applied to points, torques applied to bodies, and loads applied at joint axes (these may be forces or torques depending on the joint type). Prescribed motion can be used when the time history of a joint is known as a function of time. This means that the joint angle (or position), its rate, and acceleration are specified. Given the complete description of the multibody system, the loads acting on it, and any prescribed motion functions, the equations of motion are numerically integrated in time, starting from the given initial conditions. The state vector for any subsequent instant of time can then be used as a new initial condition for further analysis. To perform the numerical integration, the derivatives of the state vector are computed from the equations of motion. This step is done in such a way that no constraint equations are violated. The integrator then updates the solution in time, attempting to meet the user's solution accuracy requirements.

Some problems may require execution of two additional analyses before dynamic analysis can begin. This is the case when the system must satisfy constraint equations. The coordinates must satisfy position constraint equations, while the velocity variables must satisfy velocity constraints. Special routines generated by SD/FAST can be used to perform these prerequisite analyses, called *assembly analysis* and *velocity analysis*. These are described below.

R1.2 Inverse Dynamics

Inverse dynamics is simpler than dynamic analysis. In this case, all the motion of the system is specified. This means that the constraint equations and prescribed motion functions completely determine the solution time history. Additional applied loads do not change the motion, but do alter bearing reaction loads. The applied joint loads needed to make the motion occur are computed. This type of analysis is useful for sizing actuators and assessing joint loads, and for computing the loads needed to drive a system (such as a robot) through a known motion sequence.

Since all the motion is specified, the position, velocity and acceleration of any point can be found just through kinematic analysis. This gives the user a choice of solution procedure if inverse dynamic analysis is to be performed at many different solution points. If only one solution point is desired, the system state vector for a constrained system must be initialized by performing an assembly analysis and a velocity analysis. Then the applied loads and prescribed motion functions must be executed. The state derivatives are computed next. All information regarding accelerations is then available. This includes the acceleration of points, the angular acceleration of bodies, all bearing loads in the system, and driving actuator loads at prescribed joints. If many solution points are required, inverse dynamics can be simply treated as a dynamic analysis case. The solution can be obtained by numerical integration. In this way, solution points are obtained which are spaced in time by the integration time step.

In some cases, the user is interested in performing inverse dynamic analysis at just a few widely spaced points. Then, it can be more efficient to analyze each point separately, by performing a new assembly and velocity analysis before each derivative computation. This is because the numerical integrator makes multiple evaluations of the system derivative to advance each time step, and these evaluations usually require more arithmetic operations than performing an assembly and velocity analysis. However, whenever an assembly analysis is performed, one must be careful to verify that the mechanism has assembled into the correct branch.

R1.3 Static Analysis

Here the problem is to determine the equilibrium configuration of a multibody system, given the loads acting on it. This is a configuration in which all the bodies are at rest or moving at a known, steady velocity, and all accelerations are zero. The typical situation is to allow loads acting upon the system to be functions of the system state (such as springs). The value of state-dependent loads will then also be found during the static analysis.

Static analysis requires the user's starting guess at an equilibrium configuration. Sometimes a static configuration can be found that is unstable or otherwise undesired. In this case a different initial guess closer to desired configuration must be input. The stability of an equilibrium point can be tested by perturbing the system state and performing a dynamic analysis. (See Tutorial Section T2.6.3 for an example of this procedure.) If the system state stays within an acceptable neighborhood of the equilibrium the system is probably stable in an engineering sense. Static configurations can also be found by adding damping to the joints of the system and then performing a dynamic analysis. This method can be used to find an estimate of a static configuration, by letting the system settle for some length of time. A subsequent static analysis can then determine the coordinates to greater accuracy, and with far less expenditure of computer time, compared to letting the system settle solely under the influence of added damping.

R1.4 Inverse Static Analysis

When the equilibrium configuration is given, the statics problem becomes that of finding applied loads compatible with the equilibrium state. In some cases, this type of static analysis can be treated as a special case of inverse dynamics. This is so when the desired load type is an actuator force or torque at a joint. Then the motion can be prescribed to zero, and inverse dynamics analysis used to obtain the compatible applied hinge loads.

In the more general case, the applied loads will have unknown application points, force magnitudes, or directions expressed as functions of design variables. If the user can parameterize the set of loads acting upon the system, then a static configuration can be found using nonlinear root-finding methods. This problem is set up as a design problem with an underlying single-step dynamic analysis. The root finder provided with SD/FAST (see Section R8.2) can be used to find values for the design variables (and hence the forces) which produce zero system accelerations. Once solved, static analysis can be used to verify the result.

R1.5 Steady Motion Analysis

Steady motion analysis is sometimes called "dynamic equilibrium". This is a special case of motion in which the system moves in a simplified way. In particular, all joint accelerations are zero. Examples of steady motion are a spinning coin and an airplane in level flight (if the airplane is "trimmed" properly). The difference between a steady motion problem and a static problem is that, in addition to the coordinates, the velocities are unknown in the steady motion problem.

Usually, a steady motion is used as a precursor to dynamic analysis. An aerospace vehicle could be intended to perform a steady flight. The steady motion analysis would show the relationship between "stick" setting and steady state climb, for instance. Dynamic analysis would show how the system deviated from the desired profile in response to gusts, or slight initial errors. During steady motion, the velocity variables remain constant. Their derivatives are zero. The hinge coordinates might also be constant, in which case the whole system would be moving as a single rigid body. In other cases, parts of the system may have relative motion, as when the system includes spinning wheels or shafts. This type of steady motion occurs in many spacecraft problems.

Note that if all velocities are known in advance, it is more efficient to use static analysis than steady motion since there are fewer unknowns.

R1.6 Assembly and Velocity Analyses

The above analysis types apply to all systems, constrained or unconstrained. For constrained systems (commonly called *mechanisms*) additional analyses may be required. These include assembly and velocity analysis. Most mechanisms possess loops formed by the bodies. Assembly analysis describes the computation of the joint angles or positions (and hence body positions) required to close the loops. Velocity analysis refers to the computation of velocities that are compatible with the loop conditions in the mechanism. When some of the joint angle rates or body rotation rates are specified, the rest of the velocity variables in the system must be chosen so that the loops are not "flying apart."

A basic assembly analysis solves for a set of coordinates which satisfy all loop-joint constraint equations, prescribed motion position functions, and all user-written holonomic (position) constraints. Loop-joint constraint equations include requiring hinge points on designated bodies to become coincident (for rotational joints) and causing axes to have the correct relative orientation. Arbitrary additional constraints on joint or body configurations can be accommodated with the SD/FAST-generated nonlinear root finder.

Assembly analysis can succeed or fail, depending upon the user's initial guess at an assembled configuration. The user can freeze any subset of the system position variables, and not allow the assembly analysis to alter the assumed values for these variables. This can speed up the assembly process, but if too many variables are frozen it may become impossible to find a solution. After assembly of a mechanism, the user can obtain the position of any point of the system, or the orientation of any rigid body.

Velocity analysis is used to initialize the system velocity variables so that all loop velocity constraints, prescribed motion velocity functions, and user velocity constraints are satisfied. Velocity analysis is initiated after a successful assembly analysis, so that all coordinates are known. Velocity analysis requires solution of a set of linear equations, so it usually requires very little computer time. Again, arbitrary velocity constraints on hinges or bodies can be accommodated with the SD/FAST-generated root finder.

R1.7 Mechanical Advantage and Transmission Angle Analyses

Mechanism analyses also traditionally include such types as mechanical advantage or transmission angle calculation. These are related to the force transmission characteristics of the mechanism. Other mechanism analyses are concerned with path or function generation, and rigid body guidance. To perform mechanical advantage analysis, a mechanism is assembled in the desired configuration first. Then a unit load is applied at the mechanism input. The equilibrating output force can be found by performing static analysis as described above. Since mechanical advantage is related to the inverse of the velocity ratio between the input point and the output point, it can also be computed by assigning a arbitrary value to a joint rate, performing a velocity analysis, and computing the velocity ratio between the input point and the output point. Transmission angle determination requires computing the angle between the static force in a link and the direction of motion of the coupler point. These quantities can be found by performing a static analysis from which the bearing loads can be obtained, followed by a velocity analysis to determine the direction of the hinge point velocity vector.

R1.8 Design Studies

Engineering design typically involves studying trade-offs between different parameters under the engineer's control. This is primarily what engineers "do". SD/FAST can be used effectively to study the performance of a system while parameters such as mass, inertia, length, pivot location, and spring stiffness are varied. This lets the engineer find the parameters which are important to the success of the design, and learn how to modify a design to improve the performance or meet new requirements.

In a typical design study, the multibody system is viewed as operating as a function of design variables. The design variables alter some, possibly many, dimensions or properties of the mechanism. For instance, the mass of a part could be a design variable. The length of a link could be a design variable also. In this case, the location of the part mass center, the part mass, the part inertia matrix, and the location of the part's pivots could all be regarded as functions of the part length. The designer specifies a series of analyses, and extracts from each analysis a figure or figures of merit. The variation of these figures of merit as a function of small changes in the design variables yields sensitivity and tolerance information. Plots of the figures of merit versus larger changes in the design variables (called a *parameter sweep*) provide valuable design guidance. Finally, using the figures-of-merit calculations to guide an externally-supplied nonlinear optimizer yields an automated design capability.

SD/FAST provides a very convenient analysis capability, to support design studies that would otherwise be extremely difficult to perform. The format in which SD/FAST produces the multibody model (that is, explicit subroutines) allows the user to exercise all model features programmatically. This means that the SD/FAST model can be used directly from Fortran or C, or in conjunction with any other program which can access subroutines written in these languages. All the looping, testing, and computational facilities of these languages can thus be brought to bear on the design problem, for example to drive the design variable changes for a parameter sweep. SD/FAST models can be embedded in simulation programs written in special computer languages, or used as analysis modules within optimization routines.

R2 Applied Loads

Most analyses require that loads (forces and torques) be applied to the system being analyzed. SD/FAST allows loads to be applied in four ways:

1. Specify a uniform gravitational field in the System Description File.
2. Apply a force (vector) to a point on a body.
3. Apply a torque (vector) to a body.
4. Apply a hinge force or torque (scalar) at a hinge axis.

Uniform gravitational loads are specified as acting on all bodies in the system, and thus are entered in the System Description File. Gravitational loads are discussed below. All other loads (including non-uniform gravitational loads) are usually state dependent and are generated by user-written code. These loads are discussed in the following section, beginning on page R-10.

R2.1 Gravity

Gravity is handled as a special case for convenience. You can specify a gravity vector in the System Description File using the `gravity` keyword with constants and/or question marks. If there are question marks in the gravity specification, then the actual gravity vector can be specified (or turned off) at run time using the `SDGRAV()` routine. The gravity vector **g**, if present, produces a force $m\mathbf{g}$ acting at the center of mass of each body in the system. Note that this produces the effect of a uniform gravitational field. If more sophisticated modeling is required, such as gravity gradient or gravitational interactions among the bodies, these forces should be modeled using the general load application routines described below rather than with the System Description File gravity feature.

```
SUBROUTINE SDGRAV(GRAV)
DOUBLE PRECISION GRAV(3)
```

`SDGRAV()` is called at the beginning of an analysis. It is most commonly used to turn gravity on and off as needed for different analyses. Only those elements of `GRAV` which were specified with question marks in the System Description File will be changed by `SDGRAV()`. The others are ignored and need not be set. If there were no question marks in the System Description File gravity specification, `SDGRAV()` will post an error which can be retrieved with `SDPRINTERR()`.

After calling `SDGRAV()`, `SDINIT()` must be called before any further analysis can be done.

```
SUBROUTINE SDGETGRAV(GRAV)
DOUBLE PRECISION GRAV(3)
```

The current value of the gravity vector can be retrieved with `SDGETGRAV()`.

C Language

```
sdgrav(grav)
double grav[3];

sdgetgrav(grav)
double grav[3];
```

R2.2 General Loads

There are three generated routines (`SDPOINTF()`, `SDBODYT()`, and `SDHINGET()`) corresponding to the last three methods described above for applying loads. Whenever a new state is specified (by a call to `SDSTATE()`) all of these loads are set to zero. Then, the three load-applying routines can be called as many times as necessary to specify all the loads. The effect of these routines is cumulative so that, for example, if two torques are applied to the same body the effective torque applied is their vector sum.

Applying a force or torque to ground is allowed but has no effect. If non-existent bodies or joint axes are specified, the call is ignored and an error is posted for retrieval with `SDPRINTERR()`.

All loads should be applied before evaluating state derivatives either with `SDDERIV()` or `SDRESID()`. Any attempt to call these routines either before `SDSTATE()` or after `SDDERIV()` will raise an error for `SDPRINTERR()`.

The calls to the load-application routines are usually grouped together in a user-written subroutine conventionally named `sduforce()`. See Section R15.3.1 on page R-83 for more information and examples. In addition to their use in providing external loads on the system, these routines are used in the user constraint force routine `sduconsfrc()` to apply reaction loads which result from user constraints. See the discussion in Section R24 for details.

```
SUBROUTINE SDPOINTF(BODY,POINT,FORCE)
INTEGER BODY
DOUBLE PRECISION POINT(3),FORCE(3)
```

`SDPOINTF()` applies a force vector to a particular point on a given body. The point and the force vector are expressed in the body local frame.

```
SUBROUTINE SDBODYT(BODY,TORQUE)
INTEGER BODY
DOUBLE PRECISION TORQUE(3)
```

`SDBODYT()` applies a pure torque to a body. The torque is expressed in the body local frame.

```
SUBROUTINE SDHINGET(JOINT,AXIS,TORQUE)
INTEGER JOINT,AXIS
DOUBLE PRECISION TORQUE
```

SDHINGET() applies a torque (force in the case of a slider) at a particular hinge of a joint. The joint can be either a tree joint or a loop joint. Only the magnitude of the force is given since the direction is implicitly the same as the hinge axis. The given load is applied to the outboard body, while an equal and opposite load is applied to the inboard. In the case of a ball joint (or the rotational portion of a six degree of freedom joint), three “hinge torques” can be applied. These represent the measure numbers of the active torque vector applied by the inboard body to the outboard body, expressed in the outboard body’s frame. This can be expressed more succinctly using SDBODYT(), but in that case you must apply equal and opposite torques to the two bodies yourself.

C Language

```
sdpointf(body,point,force)
int body;
double point[3],force[3];

sdbodyt(body,torque)
int body;
double torque[3];

sdhinget(joint,axis,torque)
int joint,axis;
double torque;
```

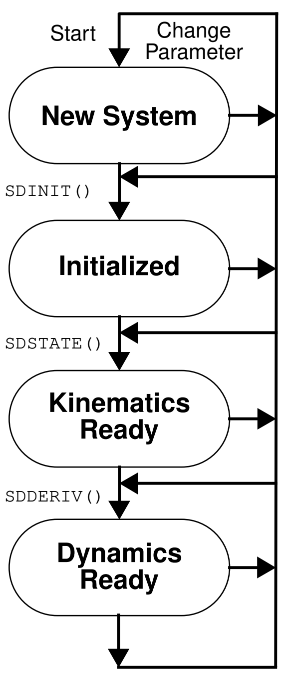
R3 Computations

This section describes the SD/FAST-generated routines which perform the main computations involved in a multibody analysis. These computations can be loosely divided into three categories: initialization, kinematics, and dynamics. Other SD/FAST-generated routines depend on the computations performed by these routines. Below, we first discuss the stages through which the computations proceed. Then we discuss the computational routines themselves.

R3.1 Computational Stages

During execution of the generated routines, your program will be in one of four computational *stages*, depending on what routines have been called so far. These stages are called (1) *New System*, (2) *Initialized*, (3) *Kinematics Ready*, and (4) *Dynamics Ready.* Typically a program proceeds through these stages in order, and then toggles between Kinematics Ready and Dynamics Ready during analyses. Some generated routines can be called only when the program is in a particular stage or stages. (These are common-sense requirements — for example, you cannot ask for point positions until kinematic computations have been performed.) In addition, some routines will cause the computation to advance to a different stage. Routines are classified into several classes according to the stages they can be called in and the resulting stage.

Each of the four computational stages is described below. For a detailed list showing the routines in each classification, the stage(s) they are allowed in, and the resulting stage, see the SD/FAST Quick Reference Guide, Section Q4 and Section Q3.



New System Stage

This is the stage that the SD/FAST generated routines are in initially, that is, before any of the routines have been called. In addition, a program returns to this stage if any system parameter (specified with "?" in the input) is changed.

While in the New System stage, the only generated routines which can be called are: Utilities, Change Parameter Routines, and `SDINIT()`. The latter results in a change to the Initialized stage.

Initialized Stage

This stage is entered from any other stage by a call to `SDINIT()`. Most commonly, `SDINIT()` is called only once in a program, at the beginning or after all the "?" parameters have been set with Change Parameter Routines.

While in the Initialized stage, the only generated routines which can be called are: Utilities, Simplified Analysis Routines, or `SDSTATE()`. The Simplified Analysis Routines `SDASSEMBLE()` and `SDINITVEL()` put the program in the Kinematics Ready stage, while the remainder of these routines put the program in the Dynamics Ready stage. `SDSTATE()` always puts the program in the Kinematics Ready stage.

Kinematics Ready Stage

Being in the Kinematics Ready stage means that all computations involving positions and velocities have been performed. This stage is entered only by `SDSTATE()`, or indirectly by the Simplified Analysis Routines `SDASSEMBLE()` and `SDINITVEL()` each of which calls `SDSTATE()` as its final act.

While in this stage, a program may call: Utilities, Position and Velocity Information Routines, Force and Motion Application Routines, and Derivative Computation Routines. If `SDDERIV()` or `SDRESID()` is called, the routines will be left in the Dynamics Ready stage. Alternatively, you may call `SDSTATE()` again with a new state, or a Change Parameter Routine or `SDINIT()` to begin the analysis anew.

Dynamics Ready Stage

Being in the Dynamics Ready stage means that all computations involving accelerations and forces have been performed. This stage is entered only by `SDRESID()` and `SDDERIV()`, or indirectly by the Simplified Analysis Routines `SDSTATIC()`, `SDSTEADY()`, `SDMOTION()`, and `SDFMOTION()` each of which calls `SDDERIV()` as its final act.

While in this stage, a program may call: Utilities, Position and Velocity Information Routines, and Acceleration and Force Information Routines. `SDSTATE()` or any of the Simplified Analysis Routines can be called to continue analysis at a new state. A Change Parameter Routine or `SDINIT()` may be called to begin the analysis anew.

R3.2 Computational Routines

This section documents the three main computational routines `SDINIT()`, `SDSTATE()`, and `SDDERIV()` and the routine `SDRESID()` which performs derivative computations for Differential-Algebraic Equation (DAE) integrators. Each of these moves the computation to the next stage.

Initialization

```
SUBROUTINE SDINIT
```

This parameterless routine *must* be called once before beginning a simulation. `SDINIT()` normalizes pin vectors and computes fixed quantities like total system mass. It also verifies that all “`?`” parameters have been given a value. An error indication will be posted if an error (such as not setting a defaultless “`?`” parameter) is detected.

In addition to the call at the beginning, `SDINIT()` must be called whenever any system “`?`” parameters have been changed via the Change Parameter Routines `SDGRAV()`, `SDMASS()`, etc. described in Section R17.1.

Kinematics Computations

```
SUBROUTINE SDSTATE(T,Q,U)
DOUBLE PRECISION T,Q(NQ),U(NU)
```

SDSTATE() is called whenever a new state has been calculated, to register the new state and to compute state-dependent quantities, such as body positions and velocities. This information is placed in globals (or common) for access by subsequently-called routines. The current state is saved for later use. Applied forces and prescribed accelerations are set to zero. Prescribed positions and velocities are set to the corresponding state variables (from Q or U) so that there is no current error.

SDSTATE() should also be called to perform a second calculation (with different forces) at the same state. In this case it performs almost no computation, it just resets the applied forces to zero and the prescribed motions as above. If only the U's have changed, only velocity-related computations are performed. When modeling Coulomb friction, for example, SDSTATE() (and SDDERIV()) may be called repeatedly during fixed-point iteration to converge the forces. Since the state doesn't change, the iteration will execute very quickly.

The state variables Q and U are organized as described in Section R20. SDSTATE() checks that the state variables are valid and posts an error if not. There are only three conditions which are considered invalid: (1) gimbal lock, (2) badly unnormalized Euler parameters, and (3) singular mass matrix. The first two are detected immediately by SDSTATE() while singular mass matrix detection is deferred until the mass matrix (or similar computation in Order(N)) is actually required.

Gimbal lock can only occur if there are gimbal-containing joints (either tree or loop joints) in the system. Gimbal-containing joints are: gimbal, bearing, and bushing. Gimbal lock is the condition in which the first and third axes of a gimbal joint are aligned or very nearly aligned. Physically, gimbal lock is a configuration in which the normally three-degree-of-freedom gimbal joint is reduced to a two-degree-of-freedom joint. In SD/FAST, where the internal gimbals of the joint are modeled as massless, gimbal lock corresponds to an untenable situation in which any torque applied to the first pin would result in infinite acceleration of one of the intermediate gimbals. This can produce numerically invalid results, and probably does not represent a configuration within the operating region of the system, so SDSTATE() treats this as an error case. Even if gimbal lock is detected, however, SDSTATE() will continue to execute after raising the error condition. You should be skeptical of the results, however, if a gimbal lock condition has been reported.

SDSTATE() expects any Euler parameters (ball joint and sixdof coordinates) in Q to be normalized, or nearly normalized. (See Section R6 for more information about Euler parameters.) These parameters are always normalized before use internally, however the passed-in Q's are never modified by SDSTATE(). If any set of Euler parameters is excessively far from being normalized (defined as norm less than 0.9 or greater than 1.1), an error is posted which can be retrieved with SDPRINTERR(). (This is considered a usage error since Euler parameters should remain normalized to at least integration tolerance during simulations.) Even in this case, however, SDSTATE() will normalize and proceed. Euler parameters of (0,0,0,0) are normalized to (0,0,0,1).

If any illegal massless or inertialess bodies have been specified, the error "Singular mass matrix" will be reported by SDPRINTERR() after the condition has been detected. Although this is only a function of the Q's, the actual computation is deferrered and is normally not done until `SDDERIV()` is called (see below). See Section R13.3 on page R-75 for more information on massless and inertialess bodies.

Dynamic Computations

```
SUBROUTINE SDDERIV(QDOT,UDOT)
DOUBLE PRECISION QDOT(NQ),UDOT(NU)
```

`SDDERIV()` computes state derivatives and constraint multipliers for the system using the state `(T,Q,U)` as last passed to `SDSTATE()`. It also makes use of any applied loads or prescribed motions as described in Section R2 and Section R14. State derivatives $\dot{q}$ and $\dot{u}$ are returned in `QDOT` and `UDOT`. The `QDOT` and `UDOT` arrays are numbered the same way as `Q` and `U` (see Section R20). `SDINDX()` can be used to index them in the (*joint,axis*) format.

Note that the error condition “Singular mass matrix” may be detected by `SDDERIV()`, although it is really just a problem with the state. See `SDSTATE()` above for more information.

Constraint stabilization is done using the current value of the Baumgarte constants `STABVEL` and `STABPOS`, which are set using `SDSTAB()` as described in Section R4.2. `STABVEL` times velocity constraint errors and `STABPOS` times position constraint errors are added to acceleration constraint errors to stabilize the numerical integration. The default values for these are zero, so no stabilization is performed. If stabilization is desired, set these to non-zero values with `SDSTAB()`. Suggestions for choosing these values appropriately are given in Section R4.2. Prescribed motion constraints can be stabilized if you provide velocity or position constraints via `SDPRESVEL()` and `SDPRESPOS()` as described in Section R14.

If there are user constraints, `SDDERIV()` will issue calls to the four user-written routines `sduaerr()`, `sduverr()`, `sduperr()` and `sduconsfrc()`.

Dynamic computations for DAE integration

The following routine is analogous to `SDDERIV()` but is for use with a differential-algebraic equation (DAE) integrator such as DASSL[1]. In integrators of this sort, the accelerations and multipliers are estimated by the integrator rather than calculated directly from the state and forces as they are with `SDDERIV()`. These estimates are passed to a user-written routine which returns a residual array which, when driven to zero, indicates the correct solution. In general, it is possible to compute a residual array in much less time than it takes to compute accelerations and multipliers with `SDDERIV()`. The following SD/FAST-generated routine can be used as the basis for the user residual-calculating routine.

```
SUBROUTINE SDRESID(QDOT,UDOT,MULT,RESID)
DOUBLE PRECISION QDOT(NQ),UDOT(NU),MULT(NC)
DOUBLE PRECISION RESID(NQ+NU+NC)
```

1. See Brenan, Campbell, and Petzold, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, North-Holland, New York, 1989, chapter 5.

SDRESID() computes residual errors in passed-in state derivatives and multipliers using the state (T,Q,U) as last passed to SDSTATE(). It takes as inputs estimated $\dot{q}$, $\dot{u}$, and Lagrange multipliers in QDOT, UDOT and MULT. It also makes use of any applied loads or prescribed motions as described in Section R2 and Section R14. Errors in state equations and velocity constraints are returned in RESID, suitable for use by a DAE solver like DASSL, operating on an unstabilized Index 2 form of the problem. In addition, if STABPOS (the Baumgarte position stabilizing constant set with SDSTAB()) is non-zero, STABPOS times the position errors will be added to the velocity errors returned in RESID. This adds some position stabilization to the otherwise uncontrolled position errors. This is often unnecessary, however, and should be used cautiously since it may adversely affect the execution time of the integrator for some problems.

MULT must always be present but is not referenced if there are no constraints. See the SDMULT() routine (Section R4.3) for information on how to initialize the multipliers when doing a DAE simulation. SDDERIV() is used to initialize QDOT and UDOT.

Routines requiring acceleration and multiplier information (e.g., SDACC(), SDREAC()) can be called and will return valid answers immediately after SDRESID() has been called. However, when using SDRESID() as the underlying routine for use by a DAE integrator, you may want to make a call to SDDERIV() after the integrator returns at the end of a communication interval. SDDERIV() will give somewhat more accurate accelerations and multipliers since it calculates them directly from the state and applied forces and motions, while the DAE integrator estimates them.

If there are user constraints, SDRESID() will issue calls to the appropriate user-written routines, namely sduverr(), sduperr() and sduconsfrc(). Note that although SDRESID() does not call sduaerr(), that routine must be written anyway since it is necessary for initialization which requires a call to SDDERIV().

C language

```
sdinit()

sdstate(t,q,u)
double t,q[NQ],u[NU];

sdderiv(qdot,udot)
double qdot[NQ],udot[NU];

sdresid(qdot,udot,mult,resid)
double qdot[NQ],udot[NU],mult[NC],resid[NQ+NU+NC];
```

R4 Constraints

Multibody systems are idealized in SD/FAST as collections of bodies, connections between bodies, and loads acting on the system. Additional conditions, called constraints, may be imposed upon the multibody system. Constraints are added to the multibody system to reflect modeling assumptions about the system, and cause the system motion and internal forces to differ from the unconstrained case. Examples of constrained systems are: a point forced to move on the surface of a sphere, a coin rolling without slipping on the surface of a plane, a robot whose end-effector contacts the ground, a multibody system in which bodies form loops. The last example is by far the most common situation which causes constraints to appear. Almost all mechanisms and machines have a multi-loop structure. In both tree- and loop-structured systems, prescribed joint motion is another very common example of a constraint.

A constraint condition can be expressed as a function of the system generalized coordinates (the configuration variables), and the generalized speeds (the motion variables). If the constraint expresses an algebraic relationship involving only the configuration coordinates and time, it is called a *position constraint*. If the constraint involves generalized speeds as well, it is called a *velocity constraint*. Physically realizable constraints can always be expressed as either nonlinear position constraints or velocity constraints which are linear in the generalized speeds. Velocity constraints are normally obtained by differentiating position constraints with respect to time. Constraints of this type, consisting of position constraint equations and their derivatives, are called *holonomic* constraints. Examples are joints and distance constraints. Some cases exist in which a velocity constraint is present, but it cannot be obtained by differentiating a position constraint. These are called *nonholonomic* constraints. The case mentioned above of a rolling coin is a classical example of a nonholonomic constraint. Another is gear motion. In both cases, the condition of rolling without slipping restricts the possible motion, but it does not restrict the configurations which the coin or gears can attain.

The most common example of a constraint between two bodies is a joint. Joints in SD/FAST are classified as either tree joints or loop joints. Both types of joints restrict the relative motion allowed between the bodies. All joints are equivalent to a set of position and velocity constraints. (We can even introduce the time derivative of the velocity constraint, the acceleration constraint.) For instance, a ball and socket joint imposes the condition that a material point of one body remain coincident with a material point of a different body. This results in three position constraints and three velocity constraints. Every joint can be defined by the constraints it imposes.

In the method used by SD/FAST to model multibody systems, a distinction is drawn between tree joints and loop joints, as discussed in Section R11.1. All the constraints associated with tree joints are eliminated by the multibody formalism used. That is, for the tree system, equations appear only for the degrees of freedom *granted* by the joint — none appears for the constraints *imposed* by the joint. Consequently, tree joints do not produce explicit constraints. Loop joints, on the other hand, are implemented by equations representing the constraints imposed by the joints, while none are needed for the degrees of freedom granted by the joint.

The constraint condition does not just "happen" in the multibody system. Each body of the multibody system is moving in accordance with Newton's Laws of motion. A constraint condition is enforced by the addition of constraint forces to the multibody system. For each constraint SD/FAST introduces a constraint force (called a Lagrange Multiplier), which appears in the governing equations as an additional unknown, along with the system accelerations.

Constraints in SD/FAST can be invoked in two different ways: by using "built-in" constraints, namely loop joints and prescribed motion, or by using user-defined constraints. The user-defined feature allows the user to extend the capabilities of SD/FAST beyond its built-in capability.

The routines in this section are used in managing the sometimes intransigent numerical behavior of constrained systems. These numerical issues fall into four categories: (1) constraint violation, (2) constraint stabilization, (3) redundant constraints, and (4) inconsistent constraints. These are discussed below.

R4.1 Constraint Violation

An arbitrary system state (that is, a set of values for the hinge positions and rates of the tree joints) will, in general, violate system constraints to some degree. An analysis which alters the system state must restrict its solutions to states in which the constraint violations are limited to within a predefined tolerance. By this we mean that the constraint error of largest absolute value is still smaller than the tolerance. For example, an assembly analysis searches for a set of generalized coordinates (tree hinge positions) which meet all the position constraints in the system. A motion analysis should produce a series of system states each of which limits violation of position and velocity constraints to tolerance. Acceleration constraints are met automatically when using `SDDERIV()` (if they can be met at all) since the accelerations are directly computed so as to meet the acceleration constraints.

The routines in this section can be used to monitor constraint errors so that violation can be detected during a motion analysis. These routines are also used during assembly and velocity analysis to provide the information needed by the root finder as it searches for states which produce small constraint errors.

```
SUBROUTINE SDPERR(PERRS)
DOUBLE PRECISION PERRS(NC)

SUBROUTINE SDVERR(VERRS)
DOUBLE PRECISION VERRS(NC)

SUBROUTINE SDAERR(AERRS)
DOUBLE PRECISION AERRS(NC)
```

These routines return the constraint errors for all constraints (prescribed motion, loop joints, and user constraints). `SDPERR()` returns position constraint errors, `SDVERR()` returns velocity constraint errors, and `SDAERR()` returns acceleration constraint errors. The errors are ordered in the `PERRS`, `VERRS`, and `AERRS` arrays as shown under `Multipliers` in the Roadmap section of the Information File generated by SD/FAST (see Section R9).

SDPERR() and SDVERR() may be called anytime after SDSTATE() has been called. If these routines are to report prescribed motion constraint errors, the appropriate SDPRESPOS() and SDPRESVEL() routines must have been called first. SDAERR() may be called only after SDDERIV() has been called. Note that SDAERR() will always return zeroes unless some of the constraints are inconsistent (meaning that there is no possible solution). This can be used to detect lockup conditions.

C Language

```
sdperr(perrs)
double perrs[NC];

sdverr(verrs)
double verrs[NC];

sdaerr(aerrs)
double aerrs[NC];
```

R4.2 Constraint Stabilization

The equations of a constrained multibody system are a set of coupled differential equations of motion and algebraic constraint equations. If these are converted (by differentiation of the constraint equations) into a set of differential equations only (as is commonly done) the new equations will be subject to “drift” during motion simulations caused by the imperfect nature of numerical integration. This drift will cause the original set of algebraic constraints to be violated as the integration proceeds. In that case, the numerical drift must be stabilized to prevent the constraint violations from becoming arbitrarily large.

The SD/FAST-generated routine SDDERIV() uses the above-described method and simulations using it are subject to numerical drift. To stabilize the solution, Baumgarte’s method is employed.[2] This involves feeding back a multiple of the velocity and position constraint violations into the calculations for the accelerations, so that subsequent integration tends to reduce these errors. Two feedback constants are required, one for the position error feedback and one for the velocity error feedback. We refer to these as STABPOS and STABVEL. Baumgarte suggests that these be chosen in the ratio a^2:$2a$, to achieve a kind of “critical damping.” SD/FAST chooses a=0 by default, so no stabilization is performed.

These constants may be set to any desired values by the user. Values too large will slow down or destabilize the integration (depending on the chosen integration time step and method). Values too small will either slow down the integrator by forcing a small time step, or be unable to stabilize the constraint errors. In practice, however, it is usually fairly easy to find a reasonably good value of a. We suggest the following procedure for choosing the Baumgarte constant a for use with SDDERIV():

2. Baumgarte, J. “Stabilization of Constraints and Integrals of Motion in Dynamical Systems,” Comp. Meth. Appl. Mech. Engr., 1 (1972), 1-16.

> Run the system unstabilized, monitoring the constraint violation using the routines discussed in the previous section (`SDMOTION()` and `SDFMOTION()` do that automatically). Add Baumgarte stabilization if either (1) the constraints become violated or (2) the step size seems unreasonably small and the constraints are *close* to being violated. The latter condition indicates that the step size is being held down to keep the constraints from being violated. Choose a single value *a* and call `SDSTAB(`2.**a*, *a***a*`)` (see below). Start with a choice for *a* which is 1/100 of the highest frequency in the system (e.g., 2000HZ => *a*=20). Then raise *a* as needed to keep the constraint errors under control. You may want to continue raising *a* as long as the higher values are producing better performance (this happens because the integrator can avoid step size reductions which may be required to keep constraints from being violated if the Baumgarte constants are not sufficiently high).

The SD/FAST-generated routine `SDRESID()` supports the more recent Differential Algebraic Equation (DAE) Integration methods as represented in the DASSL integration code.[3] This method of integration allows the constrained system to be integrated in its original form — that is, as a set of coupled differential and algebraic equations. When handled this way, the constraint errors cannot drift, so stabilization is not required. In practice, however, we support the so-called “Index 2” form of the differential-algebraic equations, in which only the velocity constraint equations are included in the DAE system. For holonomic (position) constraints, the position errors can still drift. These are much easier to stabilize than velocity constraints, however. `SDRESID()` thus uses the `STABPOS` Baumgarte constant to stabilize the position errors. As mentioned above, the default value for `STABPOS` is 0 so position constraints will not be stabilized by default. We suggest the following procedure for choosing the Baumgarte position feedback constant `STABPOS` for use with `SDRESID()`:

> Run the system unstabilized, monitoring the position constraint violation with the `SDPERR()` routine described above. If the observed drift is unacceptable, set `STABPOS` to 1 by calling `SDSTAB(0d0,1d0)`. This value will generally produce equal position and velocity constraint errors. However, you should then monitor both position and velocity constraint errors, and reduce the integration tolerance if they are still unacceptable.

When using either the ODE or DAE methods, Euler parameters used to represent ball and sixdof joint orientations are stabilized using the `STABVEL` value. The need for stabilization is indicated for Euler parameters if they become substantially unnormalized during a simulation. See Section R6 for more information.

3. Brenan, Campbell, and Petzold, *Numerical Solutions of Initial-Value Problems in Differential-Algebraic Equations*, North-Holland, 1989.

The following routines are available for setting and examining the Baumgarte constants.

```
SUBROUTINE SDSTAB(VEL,POS)
DOUBLE PRECISION VEL,POS

SUBROUTINE SDGETSTAB(VEL,POS)
DOUBLE PRECISION VEL,POS
```

`SDSTAB()` is used to change the Baumgarte stabilization constants. `VEL` is used to stabilize velocity constraint errors and `POS` is for position constraint errors. To control the shape of the error decay, we commonly use a single stabilization value a, and set `VEL`=$2a$ and `POS`=a^2. By default, both of the Baumgarte constants are set to zero, corresponding to a=0. Advice on choosing an appropriate value for a is given above.

`SDGETSTAB()` returns the current values of both the Baumgarte stabilization constants in `VEL` and `POS`.

C Language

```
sdstab(vel,pos)
double vel,pos;


sdgetstab(vel,pos)
double *vel, *pos;
```

R4.3 Redundant Constraints

Another numerical issue is the fact that constraints are not always independent. As an example, a planar four bar linkage (say a crank and rocker) modeled with four pin joints has five constraints but only two which are independent. The other three constraints are redundant and contribute nothing to the motion or forces in the system. SD/FAST will eliminate and ignore these so-called *redundant constraints*. As a system moves, the number of independent constraints may change, or the particular set of constraints which are chosen as the independent ones may change. These redundancies or changes may cause difficulties for some analysis methods. A routine, described below, is provided for monitoring the set of independent constraints.

Numerical Considerations of Redundancy

Redundancy of the constraints refers to the numerical rank of the coefficient matrix of the acceleration constraint equations. Nonredundant mechanisms possess completely independent constraint equations and a non-singular constraint coefficient matrix. Redundant mechanisms possess constraints which are not independent. This means that some set of the acceleration constraints can be expressed as linear combinations of the remaining constraints. Redundancy can exist for all configurations of a mechanism, or at isolated points. The degree of redundancy can change during the motion as well.

The presence of redundancy in a mechanism has implications for computing the motion of the mechanism, and solving for the bearing reaction loads. When a redundant mechanism is to be modeled, numerically reliable procedures must be used. The solution for the motion is physically unique, but the governing equations may become numerically poorly conditioned. SD/FAST uses numerically sound procedures to assess the independence of the system constraints. Thus, the motion can be computed accurately.

Reaction loads in a system with redundant constraints are not unique. The reaction loads can be determined only up to a minimal set, plus a set of reactions which produce no accelerations, but only contribute to “squeezing forces” in the mechanism. SD/FAST will always determine a set of loads which correspond to deleting enough of the constraints to make the reduced system have full rank. This means that in general, no statement can be made about the physical meaning of the calculated bearing loads in a redundant mechanism. In the real mechanism, the actual bearing loads will be determined by imprecision or flexible effects in the joints which comprise the system, that is, effects which are not modeled in SD/FAST. SD/FAST will determine a possible load path, corresponding to zero load at the constraints which were deleted. Which constraints were deleted can be determined by reference to the generated Information File which orders and labels the constraints, together with the map returned by `SDMULT()`, described below. This reports the number of active constraints and indicates which equations were active.

Constraint redundancy can be removed by changing the model of a mechanism, often in rather subtle ways. Use of bushings in place of rigid joints is one common way of eliminating redundancy, although there are performance implications. The vast literature on mechanism theory sets forth ways to assess the freedom of motion in a redundant mechanism[4]. Further discussion of this topic is beyond the scope of this manual. Suffice it to say, redundancy in the computer model can sometimes be an indication of potential problems in the design. Consequently, if SD/FAST detects redundancy, careful thought should be given to the system design, or to the model of the system, to determine whether the redundancy is indicative of a design or modeling problem.

A notable exception of frequent interest is the analysis of planar mechanisms. These will normally exhibit redundancy in the out-of-plane constraints. For example, in a four-bar linkage implemented with four pin joints three of the five constraints provided by the loop joint are redundant. These are the out-of-plane translational constraint and both rotational constraints (which prevent out-of-plane rotation). Only the two in-plane translational constraints are active. (For most planar mechanisms, SD/FAST will detect and eliminate the redundant constraints symbolically so that they do not even appear in the generated equations.) The in-plane bearing loads (reaction forces and torques) in a planar mechanism are unique and will be calculated correctly by SD/FAST, unless the mechanism also has redundant degrees of freedom in the plane of motion. Only the out-of-plane loads, which are not normally needed, are indeterminate.

Constraint Multipliers

In most cases, direct access to the constraint multipliers is unneeded. However, there are times when the multipliers are needed. The routine described below provides the

4. As an example, see: Phillips, J., *Freedom in Machinery; Volume 1: Introducing Screw Theory*, Cambridge University Press, 1984.

multipliers, as well as information about the independent subset of the multipliers which actually influenced the system accelerations in the last call to `SDDERIV()`.

```
SUBROUTINE SDMULT(MULTS,RANK,MULTMAP)
DOUBLE PRECISION MULTS(NC)
INTEGER RANK, MULTMAP(NC)
```

`SDMULT()` returns the constraint multipliers last calculated by `SDDERIV()`, and their rank, that is, the number of independent (nonredundant) constraints. The first `RANK` elements of `MULTMAP` contain indices, in ascending order, of the multipliers in `MULTS` which were chosen as the independent set. All other multipliers in `MULTS` are set to 0. The rank can be of interest in spotting singularities in the mechanism motion — a rank reduction indicates a configuration in which some constraints become redundant or inconsistent. A change in `MULTMAP` (even if there is no rank change) indicates a change in problem structure which, for some integration methods, may necessitate a restart of the integration on the restructured problem.

The multipliers are not normally needed in an ODE simulation (such as a simulation performed using `SDMOTION()` or `SDFMOTION()`), but they are required for initialization of a DAE simulation. The values returned by `SDMULT()` are not affected by calls to `SDRESID()`.

C Language

```
sdmult(mults,rank,multmap)
double mults[NC];
int    *rank, multmap[NC];
```

R4.4 Inconsistent Constraints

When constraints are redundant, they may also be *inconsistent*. That means that the redundant acceleration constraint equations are not all solved by the same set of multipliers. This most commonly occurs with a prescribed motion constraint being inconsistent at some point with the integrity of a loop joint. That is, to continue to follow the prescribed motion would cause a loop joint to break. The point at which this inconsistency first occurs is called a *lockup* configuration of the mechanism.[5]

Lockup is most easily detected by examining the values returned by the `SDAERR()` routine after a call to `SDDERIV()`. One or more of the acceleration errors will be substantially non-zero if the constraints are inconsistent in the current state. Also, at a lockup point the numerical integrator will be unable to proceed without breaking a velocity or position constraint.

5. Note that driving a mechanism with a force (as opposed to a prescribed motion) cannot produce an inconsistency of this type. Instead of locking up, the mechanism will respond by bouncing or, if damping is present, by settling into a static configuration.

R5 Equations of Motion

The routines in this section provide direct access to the system equations of motion. These routines are for very specialized applications and most users will never need to call them. If you think you may need a routine from this section, you might want first to make a call to Symbolic Dynamics customer support to see if there might be a simpler way to accomplish your task.

The full equations of motion are $\mathbf{M}\dot{u} + \mathbf{A}^{\mathbf{T}}\lambda = f$, where $\dot{u}$ represents the tree-system hinge accelerations, λ is the constraint multiplier array, and f is an array of forces and torques, one for each tree hinge in the system. **M** is the system mass matrix, and **A** is the constraint coefficient matrix. Note that SD/FAST does not necessarily form these matrices explicitly, especially when using the Order(N) formulation. Under normal use, SD/FAST is used to efficiently calculate $\dot{u}$ and λ using `SDDERIV()`. Routines in this section are used when explicit access to **M**, **A**, or f is needed.

The routines covered here are: `SDEQUIVHT()`, `SDCOMPTRQ()`, `SDFULLTRQ()`, `SDMASSMAT()`, `SDFRCMAT()`, and `SDREL2CART()`.

R5.1 Gravity Compensation

```
SUBROUTINE SDEQUIVHT(TAU)
DOUBLE PRECISION TAU(NU)
```

`SDEQUIVHT()` returns the hinge torques τ which would produce the same motion as all the applied forces and torques, ignoring inertial forces (see `SDFRCMAT()` below if you want inertial forces included). You can use this for gravity compensation like this:

```
call SDSTATE(t,q,u)
call SDEQUIVHT(tau)
do i=1,NU
  call SDHINGET(i,1,-tau(i))
end do
call SDDERIV(qdot,udot)
```

The example above assumes that gravity is on and that all the joints have one degree of freedom (that way the joint number is the same as the index into the `tau` array). In general, this would have to be done joint by joint if some joints have multiple degrees of freedom, using `SDINDX()` to index into the `tau` array.

This routine illustrates the equivalence between two cases: 1) the mechanical system at rest, but acted upon by arbitrary forces including gravity, and 2) the same system at rest in the same configuration, but acted upon only by hinge torques (and hinge forces). Both systems have the same hinge accelerations when the hinge torques are computed by this routine.

R5.2 Computed Torque Routines

```
SUBROUTINE SDCOMPTRQ(UDOT,TAU)
DOUBLE PRECISION UDOT(NU),TAU(NU)

SUBROUTINE SDFULLTRQ(UDOT,MULT,TAU)
DOUBLE PRECISION UDOT(NU),MULT(NC),TAU(NU)
```

The `SDCOMPTRQ()` routine returns the hinge torques (and forces) τ which would produce the indicated $\dot{u}$'s, ignoring constraints. Here's an example:

```
call SDSTATE(t,q,u)
... apply forces if you want ...
... set udot to desired u̇'s ...
call SDCOMPTRQ(udot,tau)
do i=1,NU
  call SDHINGET(i,1,tau(i))
end do
call SDDERIV(qdot,udot)
... now the u̇'s will be as you desired ...
```

Again the `tau` indexing here assumes one degree of freedom per joint; see above.

The `SDFULLTRQ()` routine is the same as `SDCOMPTRQ()` except that it takes constraints into account and therefore expects both $\dot{u}$'s and multipliers λ to be passed in.

Note that `SDFULLTRQ()` is an "open loop" routine in that you must pass in multipliers, but that those multipliers might not be the correct ones for the system with the constraints enforced. The routine maps the passed-in multipliers into physical forces acting on the mechanism and then computes the additional hinge torques which would be required to produce the passed-in $\dot{u}$'s.

R5.3 System Matrices

```
SUBROUTINE SDMASSMAT(M)
DOUBLE PRECISION M(NU,NU)

SUBROUTINE SDFRCMAT(F)
DOUBLE PRECISION F(NU)
```

These routines return the mass matrix **M** and the "right hand side" forces *f*. Note that `SDFRCMAT()` is actually the same as `SDEQUIVHT()` except that `SDFRCMAT()` includes inertial forces while `SDEQUIVHT()` ignores them.

Already-computed matrices are returned for **M** and *f* if possible. Otherwise, these routines will do the minimum calculations necessary to return the matrices. In particular, no decomposition of the mass matrix will have been performed.

Note: if you are using the Order(N) formulation, which does not produce a mass matrix, `SDMASSMAT()` will be very expensive since the mass matrix will have to be constructed on the spot. (A mass matrix has N^2 elements and thus cannot even be written down in Order(N)!) Much of the performance benefit of the Order(N) message will be lost if you call SDMASSMAT(). However, there is normally no need for an explicit mass matrix while using the Order(N) method. If you believe that you need an explicit mass matrix for your application, please consider calling Symbolic Dynamics customer support to discuss ways in which you may avoid it.

R5.4 System Jacobian

```
SUBROUTINE SDREL2CART(COORD,BODY,PT,LIN,ROT)
INTEGER COORD,BODY
DOUBLE PRECISION PT(3),LIN(3),ROT(3)
```

This routine returns vectors indicating how the orientation of a body and the location of a point on that body change with motion about a particular joint hinge axis (coordinate). The coordinate must be from 1 to NU (number of tree degrees of freedom). No attempt is made to give derivatives with respect to Euler parameters for ball joints.

You can easily build up a complete Jacobian by repeated calls to this routine; very little computation is involved in each call. This is very fast if you are using Kane's formulation, because it can use previously calculated information. It is a little more expensive with Order(N).

C Language

```
sdequivht(tau)
double tau[NU];

sdcomptrq(udot,tau)
double udot[NU],tau[NU];

sdfulltrq(udot,mult,tau)
double udot[NU],mult[NC],tau[NU];

sdmassmat(m)
double m[NU][NU];

sdfrcmat(f)
double f[NU];

sdrel2cart(coord,body,pt,lin,rot)
int coord,body;
double pt[3],lin[3],rot[3];
```

R6 Euler Parameters

The orientation of one body (reference frame) with respect to another can be represented in several different ways.[6] The most common of these are called *orientation angles* or *Euler angles*, which represent the orientation by a series of three rotations around specified axes. The common pitch, roll and yaw coordinates are an example of Euler angles. In SD/FAST, the relative orientation of two bodies connected by a rotational joint (e.g., pin, U-joint, gimbal) is implicitly represented as a sequence of Euler angles, one for each joint axis.

For bodies connected by a ball joint or a six degree-of-freedom joint, however, Euler angles are not a convenient way to represent the relative orientation. One reason is that there is no obvious set of axes about which to perform the rotations. More importantly, any three-coordinate representation of orientation must necessarily have a singular configuration, that is, a configuration in which the derivatives of the orientation go to infinity as the configuration is approached during a simulation. For gimbal joints this configuration is called *gimbal lock*, and corresponds to the alignment of the first and third axes. Gimbal lock is a physical characteristic of gimbal joints, so it is not a problem to have them represented by Euler angles. For ball or sixdof joints there is no inherent singular configuration, so the use of Euler angles would lead to limitations in the motion of the model which do not exist in the physical system. This is resolved by using *quaternions*, a representation of orientation which uses four coordinates rather than three.

$$\varepsilon_i \equiv \lambda_i \sin\frac{\theta}{2} \qquad (i = 1, 2, 3)$$

$$\varepsilon_4 \equiv \cos\frac{\theta}{2}$$

Different sets of quaternions can be used for this purpose. We use a set called *Euler parameters*, defined as follows. The orientation of a frame B relative to a frame A can always be described as a simple rotation θ about some unit vector $\lambda = (\lambda_1\ \lambda_2\ \lambda_3)$ which is fixed in both A and B during the rotation. (This is sometimes known as Chasle's Theorem.) The four Euler parameters $\varepsilon_1\ \varepsilon_2\ \varepsilon_3\ \varepsilon_4$ are then given by the definitions shown at the left. Note that these parameters cannot vary freely — their definition requires that they obey the constraint $\varepsilon_1^2+\varepsilon_2^2+\varepsilon_3^2+\varepsilon_4^2 = 1$. Thus the four Euler parameters collectively have three degrees of freedom, as required for representing orientation.

A drawback of Euler parameters is that they cannot capture "tumbling" motion. That is, adding multiples of 4π to θ in the above expressions does not change the Euler parameters. Euler angles, on the other hand, include this additional rotational information. If you need to track tumbling motion, you should use gimbal joints rather than ball joints.

R6.1 Conversion to and from Euler Angles

While the use of Euler parameters resolves the undesirable numerical properties of Euler angles, they are generally inconvenient to work with. It is usually preferable to think of orientations in Euler angles or other convenient representation, and convert to Euler parameters only when necessary. For example, you can initialize your state vector as though ball and sixdof joint orientations were measured with Euler angles, then call an

6. For an extensive discussion, see Kane, Likins, and Levinson *Spacecraft Dynamics*, McGraw-Hill, New York, 1983, pp. 1-38.

SD/FAST-generated routine which will automatically alter the state to contain the corresponding Euler parameters instead. Also, if you are using a root finder to perform an analysis, you will find that the nonlinear methods generally work better on the unrestricted Euler angles than the constrained Euler parameters.

SD/FAST provides the routines described below to convert a complete set of configuration coordinates between Euler angles and Euler parameters. Other routines (described in Section R12.3) are available which will convert individual sets of Euler parameters to and from the equivalent direction cosine matrix.

```
SUBROUTINE SDST2ANG(Q, QANG)
DOUBLE PRECISION Q(NQ),QANG(NU)

SUBROUTINE SDANG2ST(QANG, Q)
DOUBLE PRECISION QANG(NU),Q(NQ)
```

`SDST2ANG()` takes as input a configuration state array `Q` (that is, just the *q*'s, not the *u*'s) containing all the hinge positions with ball and sixdof joint orientations represented with four Euler parameters each. `Q` is copied to the output array `QANG` except that the ball and sixdof orientations are replaced with equivalent body-fixed 1-2-3 Euler angles. (These are sometimes called "Bryant angles.") The Euler angles are measured around the local coordinate frame axes of the outboard body (the "body-fixed 1-2-3" indicates that the rotation sequence is around the first axis, then around the new position of the second axis, then around the final position of the third axis). Note that the `NQ–NU` elements at the end of `Q` (in which the fourth Euler parameters reside, see Section R20) are not present in the output `QANG`. It is allowed for `Q` and `QANG` to be the same array, in which case the conversion is done in place.

`SDANG2ST()` takes as input a configuration array `QANG` containing all the hinge positions with ball and sixdof joint orientations represented by body-fixed 1-2-3 Euler angles as described in the previous paragraph. `QANG` is copied to the output array `Q` except that the ball and sixdof orientations are replaced with equivalent Euler parameters. The first three Euler parameters replace the corresponding Euler angles directly; the associated fourth Euler parameters are placed in the last `NQ-NU` elements of `Q`, an organization which can be used directly with the SD/FAST-generated routines. The Euler parameters returned will always be normalized (see below). It is allowed for `QANG` and `Q` to be the same array, in which case the conversion is done in place.

Note that for both of the above routines it is acceptable to pass in a full state array of length `NQ+NU` for either `Q` or `QANG` since the *u* portion of the array will be neither examined nor changed.

Warning: a call to `SDST2ANG()` followed by a call to `SDANG2ST()`, or vice versa, will not necessarily return the original coordinates.

R6.2 Normalizing Euler Parameters

As mentioned above, a legitimate set of Euler parameters always satisfies the equation $\varepsilon_1^2+\varepsilon_2^2+\varepsilon_3^2+\varepsilon_4^2 = 1$. During numerical integration of the Euler parameter derivatives, the computed Euler parameters will deviate from this ideal somewhat, due to normal integration error. Euler parameter derivatives computed by SD/FAST can be stabilized, so the constraint violation should not grow without bound; however, there will always be some error. Euler parameter stabilization is performed by feeding normalization errors back into the derivative calculations using the stabilization feedback constant STABVEL. See Section R4.2 for information on setting STABVEL. Warning: by default, Euler parameters (and all other constraints) are unstabilized, so they should be monitored.

Before beginning an analysis, you may want to “clean up” the state array by normalizing the Euler parameters. A routine is provided which will normalize any Euler parameters present in a state array.

```
SUBROUTINE SDNORMST(Q, NORMQ)
DOUBLE PRECISION Q(NQ),QANG(NU)
```

SDNORMST() normalizes all Euler parameters in the passed-in configuration state array Q. Euler parameters of (0,0,0,0) are normalized to (0,0,0,1). Any other set of Euler parameters is normalized by dividing each element by $\varepsilon_1^2+\varepsilon_2^2+\varepsilon_3^2+\varepsilon_4^2$. It is allowed for Q and NORMQ to be the same array, in which case the normalization is done in place. Note that only the *q* portion of the state is affected — if the actual arrays are longer than NQ, the elements after NQ (the *u*’s) will neither be examined nor changed.

R7 Executing SD/FAST

This section discusses the command line options available when executing the SD/FAST program on your computer, and documents the files which SD/FAST produces as a result.

R7.1 Command Line Options

This is the Unix and DOS command line description. The feature descriptions here are correct for other systems as well, but see the Release Notes for your system for the specific invocation instructions.

```
usage: sdfast [-nbvs] [-l language] [-p prefix] [-g dilse]
               [infile [basename]]
       -n  Use the Order(N) formulation
       -b  Break up Dynamics file into smaller files
       -v  Verbose: output roadmap, stats, etc.; -vv echos
           input file also
       -s  Use single precision
       -l  Specify output language: fortran, c
       -p  Specify prefix to be used for all generated
           external symbol names
       -g  1-4 letters to generate dyn,sar,info, or lib
           files (def: dsi); e means generate everything
       infile    System Description File name
       basename  Base to use in forming output file names
```

Parameters can appear in any order, with options appearing before the `infile` name and `basename`. Options which do not take arguments can appear together, e.g. `-vn` means “verbose, use Order(N)”. For options which do take arguments, the argument can follow the key letter immediately or be separated by white space.

The Order(N) formulation “`-n`” is an optional high-performance formulation for use of SD/FAST on large systems (typically over twenty degrees of freedom). This requires a license and an option key from Symbolic Dynamics.

The breakup “`-b`” option causes the Dynamics file to be broken into several smaller files. This can be helpful to avoid compiler limits which may prevent compiliation of very large Dynamics files. Instead, you compile several smaller files and link them together.

The single precision “`-s`” option should be used very cautiously. We do not recommend using this option on computers where single precision numbers occupy fewer than 56 bits.

Legal languages are `c`, `f`, and `fortran` (`f` is short for `fortran`), with Fortran the default output language. When generating in C, K&R C is generated rather than ANSI C, so that either K&R or ANSI compilers can compile the generated code. (ANSI compilers accept K&R syntax, but not vice versa.)

The -g option takes a string of up to four of the letters d, s, i, and l, or the letter e. These say exactly which files will be generated. The default is -gdsi. Output file names are constructed by adding suffixes to the basename. If there is no explicit basename, one is built by removing the final suffix (if any) from infile.

If there is no infile name on the command line, and the Dynamics, Simplified Analysis or Information File is to be generated, SD/FAST prompts for file names of the System Description File and any required output files and enables verbose mode. A "return" gives the default file name. If only the library file is being generated, no System Description File is requested and verbose mode is not enabled.

Command line options supercede options specified in the System Description File.

Examples

`sdfast`

Prompt for all needed files. Use defaults for all options unless they are set inside the System Description File.

`sdfast crank.sd`

Run SD/FAST on the System Description File crank.sd and produce output files crank_info, crank_dyn.f and crank_sar.f (crank_i, crank_d.f and crank_s.f under DOS).

`sdfast -lc crank.sd`

Same as previous example except output is generated in C rather than Fortran. The output files are crank_info, crank_dyn.c and crank_sar.c (crank_i, crank_d.c and crank_s.c under DOS).

`sdfast -gl`

Generate only the library sdlib.f.

`sdfast -gl -lc`

Generate a C version of the Library File (its name will be sdlib.c).

`sdfast -nb -ge -p xyz crank.sd test`

Run SD/FAST on crank.sd, generating all the output routines prefixed with xyz instead of the default sd (the space between -p and xyz is optional). In addition, we generate the library (because -ge means "generate everything") whose name will be xyzlib.f rather than the usual sdlib.f. The Order(N) formulation will be used to derive the equations. The Dynamics file will be broken into several smaller pieces, with the output file names controlled by the explicit basename test, so they will be test_info, test_dyn.f, test_dyn00.f, test_dyn01.f, etc. and test_sar.f (or the DOS equivalents).

Table R-2 Suffixes for Generated Files

	Unix	**VMS**	**DOS, NT**
Dynamics with “`-b`”	_dyn.<lang> _dyn*nn*.<lang>	_dyn.<lang> _dyn*nn*.<lang>	_d.<lang> _d*nn*.<lang>
Analysis	_sar.<lang>	_sar.<lang> _s.<lang>	
Info	_info	_info	_i
Library	*sd*lib.<lang>	*sd*lib.<lang>	*sd*lib.<lang>
	<lang>=f,c	<lang>=for,c	<lang>=f,c

R7.2 Generated Files

SD/FAST normally generates three files:

Dynamics File	Contains all system equations and system-specific generated code. This file may be broken into several smaller files using the “`-b`” option to SD/FAST.
Simplified Analysis File	Contains system-specific “canned” analyses for common operations.
Information File	Contains conveniently organized information about the specific system.

In addition, upon request it will generate:

Library File	Contains purely numerical, problem independent library routines used by routines in the Dynamics and Simplified Analysis Files, or useful in working with those routines.

The Dynamics, Simplified Analysis and Library files are source code in Fortran or C. The Information file is just a text file. Whether to generate each file can be controlled through command line options. By default, the Dynamics, Simplified Analysis and Information files will be given the same name as the System Description File, minus its suffix if any, and then followed by the suffix shown in Table R-2.

Note for DOS or “FAT” filesystem NT users: since file names are limited to eight characters, and a suffix like “`_d`” uses two of them, you must limit your basename to six characters. When using the breakup (`-b`) option, the suffix will be like “`_d02`” and will consume four characters. In that case, keep your basename to four.

All Library Files produced by a given version of SD/FAST are identical and need be generated and compiled only once (actually once for each language to be used). Once generated and compiled, the same Library File can be linked with any Dynamics and

Simplified Analysis Files produced by the same version of SD/FAST. The Library File's name will be as shown above. If you have specified a prefix other than "`sd`" for the generated routines, the corresponding Library File name will begin with that prefix rather than "`sd`".

For convenience, your system administrator may have placed an already-compiled version of the SD/FAST library in a system directory. If so, you can simply link with the library when you need it. Otherwise you will have to generate and compile one yourself. Check with your System Administrator to find the location of the precompiled library and to get instructions for linking with it.

R8 General Analysis Routines

These routines are tools which can be useful for constructing analyses in a programming language (Fortran or C). These are the more general routines which underlie the Simplified Analysis Routines. There are two basic analysis methods used: *numerical integration* and *nonlinear equation solving*. Numerical integration is used for motion analyses, that is, for simulating the motion of the multibody system through time. Nonlinear equation solving is used to obtain configurations of the system which satisfy certain desired properties, such as having zero constraint errors or zero accelerations. Both of these methods have a wide variety of other applications as well, some of which are discussed under Analysis Types, Section R1.

Underlying the nonlinear equation solver is a linear equation solving method which is documented below in case the use of nonlinear methods is too inefficient and the problem being solved is actually linear. Both the linear and nonlinear solver are capable of solving under- and overdetermined constrained sets of equations.

It is important to stress that these routines are provided simply for your convenience. SD/FAST-generated routines are designed to work smoothly with any numerical methods you may have at your disposal. There are many integration methods superior to the ones provided here for many problems. Simulation environments like ACSL, Easy 5, Simulink and Matrix-X provide a good selection, as do numerical libraries like IMSL and NAG. Other linear and nonlinear equation solvers can be used, although the ones supplied by SD/FAST are as good or better than most and have some unusual capabilities. A numerical optimizer such as OPTDES (as opposed to a root finder) can be used with SD/FAST to perform automated design studies. SD/FAST does not provide an optimizer.

R8.1 Numerical Integration

SD/FAST provides two integrators in the Library File. These are both based on the venerable Runge-Kutta 4th order explicit integration formula with Merson error estimation. For many problems, other integration methods (such as Gear's stiff integration or DAE methods) will yield better performance, sometimes dramatically so. However, the variable time step RK4 integrator provided here is among the most robust methods available. It will almost always find its way over even the roughest terrain and yield good answers in a reasonable amount of time.

Because these are explicit integration methods, they will perform poorly on "stiff" problems. The fixed-step integrator will become unstable on these problems, while the variable-step will choose unreasonably small step sizes. Answers will still be correct, however, with the variable-step method on a stiff problem. Integration of stiff systems[7] can be done much more efficiently with an implicit integrator.

7. A simple test for stiffness is to run a motion analysis with the variable step integrator at two widely separated tolerances, while monitoring the step sizes it chooses. If the step sizes appear to be almost independent of the tolerance, the problem is likely to be stiff and you should consider an implicit integrator.

Please note that these integrators are provided for your convenience and so that we can provide the Simplified Motion Analysis routines SDMOTION() and SDFMOTION(). We are by no means recommending these as the integrators of choice for the solution of multibody equations. If you have access to more sophisticated routines, say through ACSL or Matrix-X, by all means use them. However, these routines may prove handy if you are doing your analysis in Fortran or C or if you need a fallback during debugging.

R8.1.1 Variable Step Integrator

```
      SUBROUTINE SDVINTEG(F,T,Y,DY,PARAM,DT,STEP,
     1                    NEQ,TOL,WORK,ERR,WHICH)
      EXTERNAL F
      DOUBLE PRECISION T,Y(NEQ),DY(NEQ),PARAM(*),
     1                 DT,STEP,TOL
      INTEGER NEQ,ERR,WHICH
      DOUBLE PRECISION WORK(6*NEQ)
```

F is defined:

```
      SUBROUTINE F(T,Y,DY,PARAM,STATUS)
      DOUBLE PRECISION T,Y(*),DY(*),PARAM(*)
      INTEGER STATUS
```

```
      ERR=0 => success
          1 => went over a step
          2 => can't continue
          3 => non-zero status return from user function F
```

SDVINTEG() integrates a set of functions from time T to T+DT, using a variable time step method built on a 4th order Runge-Kutta-Merson step. The function F is called to compute the derivatives of the state variables Y via a call like F(T,Y,DY,PARAM,STATUS). These derivatives are integrated between T and T+DT, with relative error normally kept to below TOL. On input, the parameter DY should already have been set to the derivative vector via a prior call to F(T,Y,...). The array PARAM is passed through unchanged to F().

STEP is used as an initial guess at the step size, and is returned as a suggested initial step size for the next call to SDVINTEG(). This reflects a “locality” assumption that the function’s behavior in the next interval is likely to be similar to its behavior in the current interval. The idea is that you set STEP=DT before the first call to SDVINTEG() and then leave it alone for subsequent calls. STEP will never be returned larger than DT.

On return Y, DY, and T will have been updated. DY will contain the result of F(T+DT,Y,...), i.e., the derivative of the updated Y at T+DT. This routine is normally called in a loop. In that case the returned DY from one iteration can simply be passed in unchanged to the next.

ERR is returned 0 if we're successful, 1 if the step size got too small, suggesting that we integrated over a discontinuity. In that case, WHICH is the index of the function exhibiting the first alleged discontinuity during the interval. We continue integrating over the whole interval, since ERR=1 means that the function returned to normal after a single minimum-width step which appeared to contain the discontinuity. Provided the discontinuity is simply a step function and not an impulse at the resolution of the minimum step, the local error will be no worse than any other step.

ERR=2 is much more serious in that the integrator could not continue because the function did not return to normal after a single small step. T, Y, and DY are returned just after the last successful step. WHICH says which function exhibited the non-recoverable behavior. This function might, for example, be leaping quickly towards infinity. (This happens, for example, when a mechanism being driven by prescribed motion reaches a lockup configuration.) This can also happen if a function is changing extremely fast but for very short periods of time ("chattering"). This usually indicates a modeling error.

ERR=3 says that the user-supplied function F() returned with a non-zero status (STATUS is set to 0 before calling F()). In that case T, Y, and DY are returned just after the last (very small) step in which F() returned 0 status. WHICH is returned set to the actual status returned by F() when the next step was attempted. This can be used, for example, to detect the occurrence of discontinuous events, measured quantities reaching predefined limits, or constraints exceeding some tolerance. The returned time and state are those one minimum step width or less before the event occurs.

Important Notes:

(1) When a function is very near 0, the relative error can become large even though the absolute error is still small. If the absolute error is below 0.1*TOL, we do not attempt to achieve further accuracy even if the relative error is high.

(2) Tolerance applies only within a single integration interval. In the worst case, *every* interval could introduce an error the size of TOL. If these were all in the same direction, the global error could be much larger than TOL. It is bounded by TOL*n where n is the number of time steps made, but the number of time steps is variable here and not explicitly tracked. Normally the errors are not systematic so the global error will be about the same as the local error. If you are concerned about global error, the best way to check it is to rerun your simulation at a higher tolerance and note how many leading digits of the answer do not change between the two simulations — those ones will have been integrated correctly.

(3) The functions being integrated must not contain any impulses of duration much less than DT. Otherwise, the variable step algorithm might take a step starting before the impulse and ending after, in which case the impulse will not be noticed at all. In that case, the potential error is unbounded.

C Language

```
sdvinteg(f,t,y,dy,param,dt,step,neq,tol,work,err,which)
int (*f)();
double *t,y[/*neq*/],dy[/*neq*/],param[],
       dt,*step,tol;
double work[/* 6*neq */];
int  neq,*err,*which;
```

`f` is defined:

```
f(t,y,dy,param,status)
double t,y[],dy[],param[];
int *status;
```

R8.1.2 Fixed Step Integrator

```
 SUBROUTINE SDFINTEG(F,T,Y,DY,PARAM,DT,NEQ,
1                    WORK,ERREST,FSTATUS)
 EXTERNAL F
 DOUBLE PRECISION T,Y(NEQ),DY(NEQ),PARAM(*),DT,ERREST
 INTEGER NEQ,FSTATUS
 DOUBLE PRECISION WORK(4*NEQ)
```

`F` is defined:

```
 SUBROUTINE F(T,Y,DY,PARAM,STATUS)
 DOUBLE PRECISION T,Y(*),DY(*),PARAM(*)
 INTEGER STATUS
```

This is a 4th order Runge-Kutta fixed step integrator with a Merson error estimator. The function `F(T,Y,DY,PARAM,STATUS)` sets `DY` to the derivatives of the `NEQ` functions evaluated at time `T` and state `Y`. Initially, `DY` must be passed in as the correct derivative at the initial point. The function is then integrated from `T` to `T+DT`. `Y` is returned as the state at `T+DT`, and `DY` is returned as `F(T+DT,Y,DY,...)`, where `Y` is the updated state. Normally, this routine is called repeatedly in a loop. In that case the `DY` returned by the previous iteration is passed in unchanged to the next iteration.

`ERREST` is returned with an estimate of the integration error introduced in the step. This value has the same interpretation as the `TOL` parameter of `SDVINTEG()`. That is, every integrated function had a relative error ≤ `ERREST` *or* an absolute error ≤ 0.1*`ERREST`.

`FSTATUS` is the status value returned by `F()` when it was called at the end of the interval. (`STATUS` is set to zero before `F()` is called.)

It is normally not a good idea to use a fixed step integrator for multibody simulations, especially if they contain constraints. In almost all circumstances, a variable step integrator is to be preferred. Even if the functions being integrated are known to be very smooth, the variable-step integrator can often be made to run faster since it can take a larger step size over the smoother portions of the function. There are several occasions which may call for a fixed-step integrator, however:

- Debugging
- Software models which cannot tolerate time moving backwards
- Real-time simulations

Debugging: The fixed-step integrator plows blindly ahead over problems while the variable-step may get stuck. In that case, you may get some hint as to what is wrong by examining the results that come from the fixed-step. However, you will not get correct answers from the fixed-step integrator if the variable-step (at a comparable initial step size) fails to solve the problem.

Time going backwards: One circumstance that requires a fixed-step integrator occurs when your simulation incorporates sensors or actuators which have "memory", say a rocket motor which, once turned on, cannot be turned off for a period of time (see Tutorial 3). Unless you code these very carefully, the penchant of variable step integrators to go backwards in time when they adjust the step can cause incorrect behavior.

Real-time: In a real-time simulation, *predictable* performance is more important than *absolute* performance. A fixed-step integrator always performs the same number of function evaluations per step, so its runtime is very predictable.

In any case, if you do use a fixed-step integrator, be sure to monitor the error estimate and use a smaller fixed step if the errors appear to be intolerable.

C Language

```
sdfinteg(f,t,y,dy,param,dt,neq,work,errest,fstatus)
int (*f)();
double *t,y[/*neq*/],dy[/*neq*/],param[],dt,*errest;
double work[/* 4*neq */];
int  neq,*fstatus;
```

`f` is defined:

```
f(t,y,dy,param,status)
double t,y[],dy[],param[];
int *status;
```

R8.2 Nonlinear Root Finding

```
 SUBROUTINE SDROOT(F,VARS,PARAM,NFUNC,NVAR,NDES,
1                  LOCK,RTOL,DTOL,MAXEVAL,
2                  JW,DW,RW,IW,FRET,FCNT,ERR)
 EXTERNAL F
 DOUBLE PRECISION VARS(NVAR),PARAM(*),RTOL,DTOL,
1                 FRET(NFUNC)
 INTEGER NFUNC,NVAR,NDES,LOCK(NVAR),MAXEVAL,FCNT,ERR
 DOUBLE PRECISION JW(NFUNC*NVAR),DW(2*(NFUNC+NVAR)**2)
 DOUBLE PRECISION RW(9*(NFUNC+NVAR))
 INTEGER IW(4*(NFUNC+NVAR))
```

F is defined:

```
 SUBROUTINE F(VARS,PARAM,RESID)
 DOUBLE PRECISION VARS(NVAR),PARAM(*),RESID(NFUNC)
```

This routine underlies the simplified analysis routines SDASSEMBLE(), SDINITVEL(), SDSTATIC(), and SDSTEADY(). It can be used to perform any analysis which requires the solution of nonlinear equations. For example, much more sophisticated assembly analysis than SDASSEMBLE() can be done with SDROOT() — assemble such that body A is at a certain orientation, or body A's x axis is parallel to body B's y axis, or some point on body A is within some distance of a point on body B, etc.

SDROOT() provides solutions to constrained systems of NFUNC nonlinear equations (of course, linear equations can be handled as well, but this is not the most efficient method, see SDLSSLV() on page R-40). Unlike linear equation solving, nonlinear equation solving is an iterative process fraught with perils. There is, in general, no guarantee that roots to nonlinear equations can be found even if they exist. And since there can be many solutions, a returned solution may not be the desired one. Furthermore, it may take arbitrarily long to find a solution even if the routine is heading in the right direction. Fortunately, in practice many interesting problems can be solved and the chances of finding a solution can be greatly increased if a fairly good initial guess can be provided as a starting point.

F() is a passed-in routine which can be called to simultaneously calculate the residual error in each of the NFUNC equations at a trial solution point. (PARAM is simply passed through to F() uninterpreted.) The first NREQ=NFUNC–NDES equations are *required* (constraints) while the last NDES equations are *desired* (objectives). An initial guess at the solution is passed in the array VARS. On return, VARS will be set to the best solution SDROOT() was able to find. In the best case, the returned solution is such that all the residuals are below tolerance (requireds must go below RTOL while desireds are pushed below DTOL if possible).

The LOCK array is used to prevent some of the VARS from being changed during the search for a solution. If LOCK(*i*) is nonzero, VARS(*i*) will not be changed. MAXEVAL limits the number of calls that SDROOT() can make to the passed-in F(). SDROOT() will return early if it exceeds MAXEVAL calls before finding a solution.

The actual number of calls made is returned in FCNT and may be somewhat larger than MAXEVAL.

On return, ERR=0 if the residuals for all required equations are below RTOL and the residuals for the desireds are all below DTOL. ERR=1 if the required residuals are below RTOL but the desireds exceed DTOL. ERR=2 if the required's residuals are not all below RTOL. If MAXEVAL >= FCNT on return, SDROOT() was still making progress (possibly very slow progress!) when it ran out of time. In that case another call may improve the solution.

C Language

```
sdroot(f,vars,param,nfunc,nvar,ndes,lock,rtol,dtol,
       maxeval,jw,dw,rw,iw,fret,fcnt,err)
int    (*f)();
double vars[/*nvar*/], param[], rtol, dtol,
       fret[/*nfunc*/];
int    nfunc, nvar, ndes, lock[/*nvar*/], maxeval,
       *fcnt, *err;
double jw[/* nfunc*nvar */],
       dw[/* 2*(nfunc+nvar)**2 */],
       rw[/* 9*(nfunc+nvar) */];
int    iw[/* 4*(nfunc+nvar) */];
```

f is defined:

```
f(vars,param,resid)
double vars[/*nvar*/], param[], resid[/*nfunc*/];
```

R8.3 Least Squares Solution to Constrained Linear Equations

```
   SUBROUTINE SDLSSLV(NR,NC,NRA,NCA,NDES,MAPR,MAPC,
1                    TOL,DW,RW,IW,W,B,X)
  INTEGER NR,NC,NRA,NCA,NDES,MAPR(NRA),MAPC(NCA)
  DOUBLE PRECISION TOL,W(NR,NC),B(NR),X(NC)
  DOUBLE PRECISION DW(2*(NRA+NCA)**2)
  DOUBLE PRECISION RW(4*(NRA+NCA))
  INTEGER IW(3*(NRA+NCA))
```

This routine is used by the SDROOT() nonlinear equation solver to solve intermediate linear systems which arise during the attempt to solve the nonlinear system. For most linear problems which occur in mechanism analysis (such as finding a compatible set of initial velocities) the nonlinear solver can be used with adequate performance. It is usually easier to set up the problem for SDROOT() than for SDLSSLV(). However, conservative assumptions in SDROOT() limit how far it can adjust the solution in one step, so many steps may be required to solve a linear problem which SDLSSLV() would solve in a single step. In those cases it may be desirable to use SDLSSLV() directly.

SDLSSLV() provides least squares solutions to constrained systems of linear equations WX=B, with W an NRA $\times$ NCA matrix in an NR $\times$ NC storage area (NRA $\leq$ NR, NCA $\leq$ NC).

X is NCA elements in an NC-element storage area, B is NRA elements in an NR-element storage area. MAPR contains NRA integers which provide the locations of the NRA rows of W out of the NR rows of the storage area. MAPC is applied similarly to the columns. That is, if A is the storage area, W=A(MAPR(I),MAPC(J)), I=1..NRA, J=1..NCA. X is similarly indexed by MAPC, and B by MAPR.

The first NREQ=NRA-NDES rows of W are treated as constraints which are *required* to be met to the greatest degree possible, while the last NDES rows are objectives which are *desired* to be met provided that can be done while keeping the requireds met. W can be an arbitrary shape, poorly conditioned, over- or under-determined. For overdetermined problems (no solution) X is returned as the solution which minimizes the 2-norm (sum of squares) of the residual. The solution returned for an underdetermined set of equations will be the one whose 2-norm is least.

This routine is both slower and numerically less accurate than a straight QR decomposition, especially when desired equations are involved. If NA=NRA=NCA, performance is about 3*NA**3 floating point operations if there are no desireds, and NDES*(NA**2)+3*(NA+NREQ)**3 otherwise. If NDES=NREQ=NA/2, the operation count is about 10*NA**3.

C Language

```
sdlsslv(nr,nc,nra,nca,ndes,mapr,mapc,tol,dw,rw,iw,w,b,x)
int    nr, nc, nra, nca, ndes, mapr[/*nra*/],
       mapc[/*nca*/];
double tol, w[/*nr,nc*/], b[/*nr*/], x[/*nc*/],
       dw[/* 2*(nra+nca)**2 */],
       rw[/* 4*(nra+nca) */];
int    iw[/* 3*(nra+nca) */];
```

R9 Information File

SD/FAST generates an Information File containing important information about the system for which it generated equations. This information is organized into three separate tables called the *Roadmap*, *State Index Map*, and *System Parameters*. Each of these is described below.

R9.1 Roadmap

The Roadmap is a tabular representation of the system topology. First the tree system is described, followed by the loop joints. Each joint (and body) is numbered, the type of joint is shown, and the inboard and outboard bodies involved are indicated. The hinge coordinate numbers and multipliers (if any) assigned to each joint are listed. Hinge coordinates and multipliers associated with prescribed motion are indicated with a "p" or a "?" following the index number. "p" indicates that the axis is prescribed, "?" means that the axis *may* be prescribed. In either case, a multiplier is always allocated for the prescribed motion constraint.

The Roadmap below is for a four-bar linkage consisting of ground and three links, with the tree system consisting of two pins and a cylinder joint. Thus there are four hinge coordinates (q's) and four rate variables (u's) in the tree. The first joint may have prescribed motion enabled at run time, so there is a multiplier (index 1 in the multiplier array) allocated. The loop joint which connects the rocker back to ground is a ball joint. That means it is represented by four Euler parameters, or quaternions. Note that the index for these pseudo-coordinates begins at 1 again. Three multipliers are allocated (indices 2, 3, and 4) to implement the three constraints provided by the ball joint.

```
ROADMAP (fourbar.sd)
Bodies        Inb
No  Name      body Joint type  Coords q         Multipliers
--- --------- ---- ----------- ---------------- -----------------------
  0 $ground                                     |
  1 crank       0  Pin(1D)       1?             |  1p
  2 coupler     1  Pin(1D)       2              |
  3 rocker      2  Cylinder      3   4          |

Loop Joints                    Pseudo Coords lq

  4 rocker      0  Ball          1   2   3   4 |  2   3   4
```

If there are user constraints, they are also numbered and listed in the Roadmap. Each user constraint is assigned a multiplier, beginning at the next available multiplier after the last loop joint multiplier.

When SD/FAST generates code in C instead of Fortran, the indices in the Roadmap (and elsewhere) are consistent with C's convention of numbering from 0 rather than 1 as in Fortran.

R9.2 State Index Map

The configuration and velocity of a system are represented by a state vector, composed of position coordinates (q's) followed by velocity variables (u's). Only the positions and velocities of the tree joint axes need be known to fully determine the system motion. However, SD/FAST also maintains a similar state vector of “pseudo-coordinates” which give the corresponding information for loop joint axes. The State Index Map documents the correspondence between the entries in the state or pseudo-state vectors and the associated joint number/axis number pair. The structure of the state array and pseudo-state variables is discussed in Section R20.

Normally, there is one *q* and one *u* for each joint axis. The state index for the *q* and *u* are thus shown on the same line in the table. However, for ball joints there are four *q*’s (quaternions) and only three *u*’s. To keep the indexing of the *q*’s and *u*’s in correspondence, the fourth quaternion is placed at the end of the *q*’s in the state, rather than adjacent to the other three *q*’s for that joint. (This is not clear from the example below since the ball was the only loop joint.)

SD/FAST provides a generated routine `SDINDX()` which performs this mapping, so it is not normally necessary to use the table. This routine is discussed in Section R20.5. However, if you want to know exactly which state variable corresponds to each system coordinate, the information is here.

```
STATE INDEX TO JOINT/AXIS MAP (fourbar.sd)

Index
 q|u   Joint  Axis   Joint type    Axis type
-----  -----  ----   -----------   ----------
 1|5       1     1?  Pin(1D)       rotate
 2|6       2     1   Pin(1D)       rotate
 3|7       3     1   Cylinder      translate
 4|8       .     2   .             rotate

lq|lu
 1|5       1     1   Ball          quaternion
 2|6       .     2   .             quaternion
 3|7       .     3   .             quaternion
 4|        1     4   Ball          4th quat
```

R9.3 System Parameters

The final table in the Information File simply lists the important system parameters, such as number of bodies and joints, number of constraints, etc. These parameters are important to know when using the SD/FAST-generated routines, since arrays are typically dimensioned to these values.

We recommend use of these names in `PARAMETER` statements in your Fortran program to avoid the potential confusion caused by the direct use of the numerical values. In C, use `#define` statements to the same end.

```
SYSTEM PARAMETERS (fourbar.sd)
Parameter  Value  Description

nbod           3  no. bodies (also, no. of tree joints)
njnt           4  total number of joints (tree+loop)
ndof           4  no. degrees of freedom allowed by tree joints
nloop          1  no. loop joints
nldof          3  no. degrees of freedom allowed by loop joints

nq             4  no. position coordinates in state (tree joints)
nu             4  no. rate coordinates in state (tree joints)
nlq            4  no. position coordinates describing loop joints
nlu            3  no. rate coordinates describing loop joints

nc             4  total no. constraints defined
nlc            3  no. loop joint constraints
npresc         1  no. prescribed motion constraints
nuserc         0  no. user constraints
```

R10 Joint Load and Reaction Information

This section explains how to obtain information about the loads occurring at joints. Two kinds of loads can be obtained: *reaction loads*, which are always applicable, and *computed hinge loads*, which are most valuable when a joint axis is being made to follow prescribed motion. Two SD/FAST-generated routines are provided for extracting this information. These routines are described below after some introductory material.

R10.1 Reaction Loads

During the motion of a multibody system, the individual bodies exert loads upon each other which arise because of the joints in the system. These loads, called reaction loads, are of interest in many mechanical design studies. Knowledge of the reaction loads allows the designer to make the parts and joints strong enough to withstand the stresses that will be experienced during the motion.

Reaction loads are available from SD/FAST after accelerations have been computed. A reaction load is computed for each joint in the system. A simple way to think of the reaction load is to think of cutting the joint, then visualizing the loads required to make the system move in the same manner as when the joint is present. This is what the reaction loads accomplish. Each joint enforces kinematic constraints upon the relative motion (rotation and translation) permitted by the joint. The reaction loads are the set of forces and torques needed to enforce the joint kinematic relationships, plus the forces and torques contributed by active elements at the joint, such as motors, springs, and dampers.

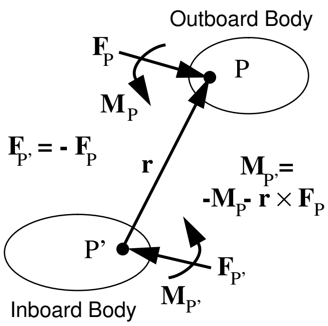
Reaction loads in SD/FAST include joint loads produced by actuators, springs, dampers, etc.[8] Actuator loads such as those produced by motors, springs, and dampers contribute to active forces, and are included in the reaction loads. For example, in a pin joint the reaction torque may have a component in the direction of the pin axis. This component is equal to the motor torque. In a U-joint, the reaction torque comes from contributions from the hinge torques and a joint constraint torque. The projection of the reported reaction torque vector in the direction of the joint axes gives the corresponding joint torque, even when the joint axis vectors are not perpendicular (as allowed in tree joints). In a gimbal joint, there is no constraint torque, so any reaction torque is due to hinge torques. Again, the projection of the reported reaction torque on the hinge axes gives the hinge torque.

The following routine is used to obtain the reaction loads.

```
SUBROUTINE SDREAC(FORCES,TORQUES)
DOUBLE PRECISION FORCES(NJNT,3), TORQUES(NJNT,3)
```

8. Some conventions do not include the hinge loads in the definition of a reaction load. We chose to include hinge loads since this preserves Newton's Third Law of equal and opposite reactions between bodies. So be careful!

SDREAC() returns all the reaction (bearing) loads for all the joints in the system. The forces are those applied by the inboard body to the outboard body, at the hinge point on the outboard body, and expressed in the outboard body frame. The torques are those applied by the inboard body to the outboard body, expressed in the outboard body frame. These are the total reaction loads felt by the outboard body, including those produced by active hinge loads. To find the reaction loads exclusive of the hinge load, subtract the hinge load applied at the outboard hinge from the reported reaction loads. You can use SDGETHT() to find the magnitude of this hinge load; the direction, of course, is the direction of the hinge axis. For a ball joint (which has no axes), there are no reaction torques exclusive of the active "hinge" torques.



The reaction loads obey Newton's Third Law. This means that an equal and opposite set of reaction loads are exerted upon the point of the joint's inboard body that is coincident with the hinge point of the outboard body. For joints which permit relative rotation only, the coincident point is the inboard hinge point of the joint. *For joints which permit relative translation, the reaction loads at the inboard hinge point can be found by using the transfer rules for shifting a set of forces from one point to another.* This means that the moment of the forces about the new point must be added to the torque vector. The moment arm of the force is the extension of the joint. Thus for translational joints, the absolute values of the reaction torques **M** at the inboard and outboard hinge points will differ by an amount $- \mathbf{r} \times \mathbf{F}$, where **F** is the reaction force on the outboard body and **r** is the vector from the hinge point P' to the hinge point P.

SDREAC() is a computational routine, that is, it is relatively expensive to call so should be called only once per integration communication interval. You cannot call SDREAC() until SDDERIV() or SDRESID() has been called, either directly or by a Simplified Analysis Routine.

R10.2 Compute Hinge Loads

When some joint axes are being made to follow prescribed motion, it is often desirable to know what loads would have to be applied at those axes to produce that motion. Knowledge of these loads can be valuable in sizing actuators and also in determining an appropriate set of commands to give the actuators to produce a desired motion.

The computed hinge loads are commonly referred to as *computed torques*, although the loads will actually be forces in the case of translational axes. The following routine is available for obtaining computed torques.

```
SUBROUTINE SDGETHT(JOINT,AXIS,HINGET)
INTEGER JOINT,AXIS
DOUBLE PRECISION HINGET
```

SDGETHT() returns the hinge torque being applied at a particular hinge axis of any joint. For sliders, this is a force rather than a torque. Only the magnitude is returned; the direction is the direction of the hinge axis. For ball joints, SDGETHT() should be called three times with successive "axes". The three returned values are the three components of the active torque vector.

If the hinge axis in question is not prescribed, then the returned torque will just be the user-applied hinge torque. When prescribed, the returned torque is that which would be required to obtain the desired prescribed motion. Note that if there are explicit loads applied with SDHINGET() to a prescribed hinge, these loads are included in the values returned by SDGETHT(). That is, the computed torque due just to the prescribed motion is the difference between the value reported by SDGETHT() and the sum of all the loads applied with SDHINGET() to the same hinge.

SDGETHT() is a data access routine, that is, it is inexpensive to call. The hinge torques are actually computed by SDDERIV() or SDRESID(). You cannot call SDGETHT() until SDDERIV() or SDRESID() has been called, either directly or by a Simplified Analysis Routine.

C Language

```
sdreac(forces,torques)
double forces[NJNT][3],torques[NJNT][3];

sdgetht(joint,axis,hinget)
int    joint,axis;
double *hinget;
```

R11 Joints

Joints are the devices that SD/FAST uses to make “connections” between the bodies, or between a body and the ground. These connections typically impose constraints on the relative motion between the bodies. In addition, internal loads (e.g., motors or springs at joints) can be applied between bodies using joints, and joint coordinates are used to measure the relative motion between bodies.

These connections range from models of classic devices such as a pin joint (door hinge) to a completely free six degrees-of-freedom (DOF) joint for tracking arbitrary motion between two bodies. SD/FAST has eleven defined joint types (Figure T1-5 on page T-12 contains illustrations of all eleven):

1. **Pin Joint**: A 1-DOF rotational joint.
2. **Slider Joint**: A 1-DOF translational joint.
3. **Cylinder Joint**: A 2-DOF joint with one translational and one rotational DOF aligned on the same axis.
4. **Universal (Hooke’s) Joint**: A 2-DOF rotational joint.
5. **Planar Joint**: A 3-DOF joint with two translational plus one rotational DOF (like a hockey puck on ice).
6. **Gimbal Joint**: A 3-DOF rotational joint.
7. **Ball Joint**: A 3-DOF rotational joint without axes.
8. **Bearing Joint**: A 4-DOF joint with one translational plus three rotational DOFs (like a slider followed by a gimbal). This joint is useful in non-redundant modeling of a shaft bearing, like a race of ball bearings through which the shaft can translate as well as rotate and tip.
9. **Bushing Joint**: A six DOF joint with three translational plus three rotational DOFs, with rotations occurring about defined axes, like a gimbal joint. The name comes from the fact that this joint is useful in modeling flexible bushings.
10. **Free (6dof) Joint**: A completely free, 6-DOF joint with three translational plus three rotational DOF, but without rotational axes, like a ball joint. Unlike a bushing joint, there is no singular configuration (gimbal lock) for a free joint.
11. **Weld Joint**: A zero DOF joint, i.e., no relative motion allowed.

Several of these joints come in “reverse” flavors for convenience in modeling the tree portion of a system which includes loop joints. Reverse just means that the DOFs are in the opposite order with respect to the joint’s inboard and outboard bodies. For example, a bearing joint normally has 1 translational DOF followed by 3 rotational DOFs. The reverse-bearing joint has 3 rotational DOFs followed by 1 translational DOF. The available reverse joints are: reverse planar, reverse bearing, reverse bushing and reverse free joints. Reverse joints are only used as tree joints, never loop joints, since reversing a loop joint is accomplished simply by interchanging its “inboard” and “outboard” bodies.

Any of these joints can also be combined to create other special joints. For example, a ball joint and a slider joint can be connected with a massless body[9] to form a telescoping ball joint. Below, we’ll discuss each of the eleven joint types. All joints can be used as a tree or a loop joint. Restrictions or extensions which apply only for use in the tree or as a loop joint are noted where applicable. Before we discuss each specific joint type, the two classes of joints (tree and loop), and general joint construction will be discussed.

R11.1 Tree Joints Versus Loop Joints

When modeling a system with SD/FAST, the model must be divided into an open loop topology "tree" system and a set of "loop joints" which produce closed loops in the topology. This is done by identifying loops in the system topology and making one "cut" in each loop. Although any body or joint in the loop may be cut, it is most efficient to cut at the joint with the most degrees of freedom since that produces the least number of constraints and removes the most degrees of freedom from the tree system. It is also good to minimize the maximum length of any chain of bodies in the system, so cutting a loop in the middle is generally better than cutting it at either end. If there are massless or inertialess bodies in the system, you must avoid making a cut which will leave the tree system ill-conditioned. This is discussed in detail in Section R13.3.

Tree joints are always assembled. If the reference configuration is input such that all the loop joints are also already assembled, then the loop joint specifications are exactly like those for joints in the tree system. Otherwise, some additional vectors can be specified to control the allowable assembled configurations and to define the zero positions for the loop joint coordinates (called *pseudo-coordinates*).

There are several restrictions applying to loop joints which do not apply to tree joints. These are:

1. Loop U-joint: the two axes must be perpendicular.
2. Loop gimbal and bearing joints: the first two axes must be perpendicular, and the third axis must be perpendicular to the second.
3. Loop planar joint: the three axes must be mutually perpendicular, and must form a right handed set. That is, the third pin must be in the direction of the vector formed by the cross product of the first pin with the second.
4. Loop free and bushing joints: as for the loop planar joint, the three axes must be mutually perpendicular and form a right handed set.
5. There are no "reverse" loop joints. Instead, simply reverse the order of the two bodies connected by the loop joint.

The coordinates for tree joints are position states of the system, as you would expect. However, loop joints can only add constraints to the system and do *not* represent any additional states. That is why we call loop-joint coordinates *pseudo*-coordinates. Special routines (and additional computations) are needed to access these coordinates. Also, while tree-joint coordinates are continuously integrated and can build up to represent

9. A "massless" body is simply an SD/FAST body specified with zero mass and/or inertia. Massless bodies are useful for creating complex joints from primitive joints, enforcing distance constraints (two ball joints with an intervening massless rod), and other constructs deemed useful by the analyst. However, these must be used cautiously. See Section R13.3 for details.

multiple revolutions of a joint axis, loop-joint pseudo-coordinates are computed kinematically and cycle through zero for each revolution. You can, however, integrate the loop-joint pseudo-velocities to capture multiple revolutions. Because of this additional complexity, we recommend that, whenever possible, loop joints be created only at joints for which large-rotation coordinate information is not needed.

Another way to avoid pseudo-coordinates is to break a loop in a *body* rather than at a *joint*. Then the two "half" bodies are connected with a loop weld joint. There is a computational penalty associated with this method, but it can be useful if the additional compute time can be tolerated. See Section R16.1 for more details.

In summary, all the necessary routines exist to use loop joints exactly as you would tree joints, but we recommend you weigh the following (sometimes conflicting) guidelines, in this order, in selecting the joint to cut:

1. Never cut a joint or body in such a way that the resulting tree system is ill-conditioned due to the placement of massless or inertialess bodies.
2. If you don't mind the additional computational burden or want to avoid pseudo-coordinates, cut loops only in bodies and use only `weld` loop joints.
3. If you do cut a joint, cut the highest-DOF joint in the loop. Each DOF in the cut joint reduces the number of constraint equations by one, *and* reduces the number of DOFs in the tree system by one.
4. Cut the joint or body that results in the shortest tree branches. Shorter tree branches reduce the computational load when using Kane's formulation, sometimes substantially. The effect on the Order(N) formulation is much less noticeable.
5. Cut a joint for which the coordinates are not of interest. In that case the pseudo-coordinates can be ignored and your driving program will be simpler.

R11.2 General Joint Construction

The best way to understand SD/FAST joints and how to use them is simply to proceed through a typical modeling process and generate the input file information. We shall proceed through a general example with specific examples shown in each joint description. The additional information required for specifying unassembled loop joints will also be discussed.

In describing a joint in the SD/FAST System Description File, the first information specified is the body (also called the *outboard* body), the *inboard* body, and the type of joint between these bodies:

`body =` *bodyname*   `inb =` *bodyname*   `joint =` *jointtype*

where *jointtype* is one of the defined SD/FAST joint types. In a tree joint, the named `body` is one which has not previously appeared in the System Description File. The inboard body is one which has already appeared as a `body` earlier in the file. Thus for a tree joint the inboard body always has the lower body number. For a loop joint, the choice of inboard and outboard is arbitrary, although it does determine sign conventions and pseudo-coordinate ordering. In this case, both the inboard and outboard body must already have appeared in earlier `body` statements in the tree section.

Figure R-1

Generic Joint Parameters

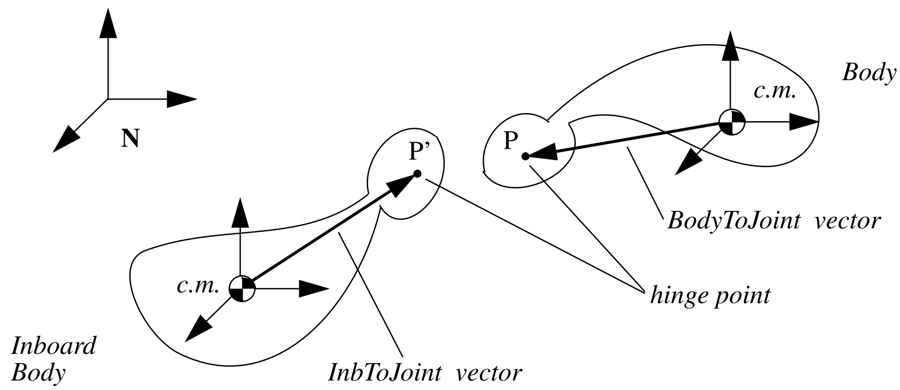


Figure R-1 diagrams the parameters common to all joint types (we draw the bodies as "blobs" to represent general rigid bodies). The (outboard) body has a hinge point P located by the vector `bodytojoint` from the center of mass of the body to point P. Similarly, the inboard body has a hinge point P' located by the vector `inbtojoint` from the center of mass of the inboard body to point P'. The hinge points and vectors are *fixed* in their respective bodies.

The point P' on the inboard body is selected from the set of all possible points such that *all* the joint axes of the joint pass through that point. For a pin joint, slider joint, or cylinder joint, P' can lie anywhere along a line parallel to and passing through the joint. For a U-joint, planar joint, gimbal joint, bearing joint, or ball joint, P' can only lie at the one point where all joint axes intersect. For a weld joint, free joint or bushing joint, P' can lie anywhere in space.

Assuming the joint is assembled, the outboard hinge point P must lie at exactly the same point as P' for welds and all joints that only involve rotation, namely: pin joint, U-joint, gimbal joint, and ball joint. For joints with translational degrees-of-freedom, P coincides with P' only when the coordinates of the translational joint axes are all zero (i.e., in the reference configuration).

The two hinge point vectors are defined in the system reference configuration and are entered as three scalar measures of the vectors:

`inbtojoint =` *x x x*   `bodytojoint =` *x x x*

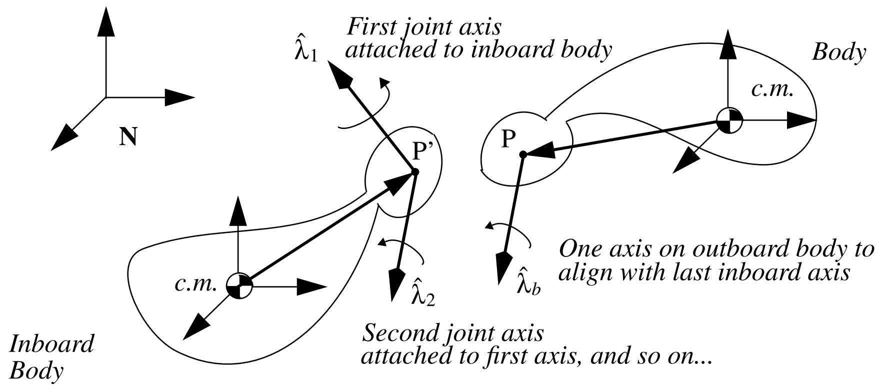
"*x*" can be either a constant real number or a variable parameter "`?`", possibly with a default value "*x*`?`". As we suggest for all system parameters, it is wise to choose a reference configuration in which as many as possible of the parameters are zero or constant to minimize the simulation computational load.

Next, the joint axes are defined using the "`pin`" keyword. There should be as many "`pin`" statements as there are joint axes in the joint. The pin, slider, and cylinder joints only have one joint axis; the U-joint has two; the planar, gimbal, bearing, free and bushing joints have three; and the ball and weld joint have none. The ball joint and the ball joint portion of the free joint have no joint axes since relative orientation is tracked using quaternions, *not* successive rotations about joint axes (see Section R6).

Figure R-2 suggests how axes are specified for joints in general. *The order in which the joint axes are entered is very important*. The joints are built by successively adding the pins, *always starting on the inboard body.*

Figure R-2

Generic Joint Axes



For example, in setting up a gimbal joint, a roll-pitch-yaw sequence is definitely not equivalent to a pitch-yaw-roll sequence! This is especially important when tracking the individual joint axes is required. Also, for joints which provide both rotational and translational motion along different axes, the ordering defines which axes are rotational and which are translational. The pin axes for, say a gimbal joint would be entered in order from the inboard body to the outboard body as follows:

```
pin = x x x
pin = x x x
pin = x x x
```

where the same options of real numbers and variable parameters "?" apply. Although joint axes are unit vectors, the specified pins do not have to be. They will be normalized as required for SD/FAST internal use. Also note that pin axes of a multi-axis joint on a tree joint do not have to be successively perpendicular. However, pin joints on *loop* joints must be successively perpendicular. No pin vector for any joint can be specified as zero. See the individual joint descriptions below for more specific restrictions on the allowable joint axes.

The last hinge axis is considered attached to the outboard body. For joints entered already assembled (including all tree joints), the last joint axis is fixed in the outboard body in the orientation it has in the reference configuration. Also, all joint coordinates for already-assembled joints have a value of zero in the reference configuration. However, when the joint is an unassembled loop joint, additional information is needed; namely, the orientation of the last joint axis in the outboard body (i.e., a "socket" into which to "plug" the last joint axis) and some reference lines to define zero.

When entering an unassembled loop joint, it is best to view the "break" in the joint as occurring between the last joint axis and the outboard body. That is, all the axes and intermediate frames of the joint should be viewed as though they were attached to the inboard body. Then an additional axis can be provided on the outboard body. When assembled, this axis will be aligned with the last of the axes “attached” to the inboard body.

In addition to the extra axis on the outboard body (specified with the `bodypin` keyword), you can specify a reference line in the outboard body, using the `bodyref` keyword. The angle between this line and a specified line on the inboard body is defined as the pseudo-coordinate of the joint’s final axis (the one attached to the outboard body). For joints with multiple axes, the reference line on the inboard body is one of the joint axes (see each joint for a specification). For single axis joints (pin, slider, cylinder, and weld) you may specify an additional axis in the inboard body (called the *inboard reference line* and specified with the `inbref` keyword) to use for measuring the axis rotation. The reference lines will be given default values by SD/FAST if they are unspecified, so unless you are interested in controlling loop joint pseudo-coordinate definitions you can safely ignore reference lines. Reference lines, if specified, must be perpendicular to the axis specified on the same body.

R11.3 Joint Numbering

A unique number is assigned by SD/FAST to each joint appearing in the System Desription file. This number is used to designate the joint in generated routines. For tree joints, the joint number is identical to the body number of the joint’s *outboard* body. (Bodies are numbered in the order their definitions appear in the System Description file.) Loop joints are numbered beginning at one greater than the highest-numbered tree joint, that is, at the highest body number plus 1.

The joint and body numbers assigned by SD/FAST may be seen in the Information File, as described in Section R9.

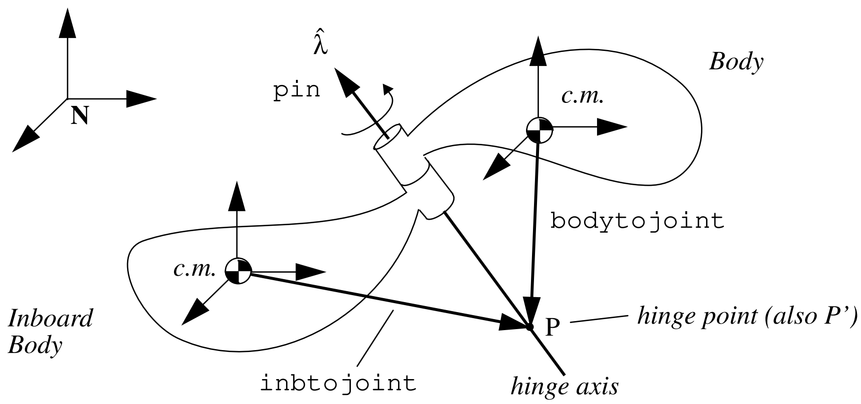
R11.4 Pin Joint

A pin joint is a one degree-of-freedom rotational joint. A diagram of an assembled pin joint is shown in Figure R-3. If the joint is entered into the system description file as assembled (or if you don’t care what defaults SD/FAST chooses for an unassembled joint), you only need to specify three vectors:

`inbtojoint = ` *x x x*	Vector from inboard body c.o.m. to hinge point P’.
`bodytojoint = ` *x x x*	Vector from body c.o.m. to hinge point P.
`pin = ` *x x x*	Vector defining joint axis on inboard body and body.

Figure R-3

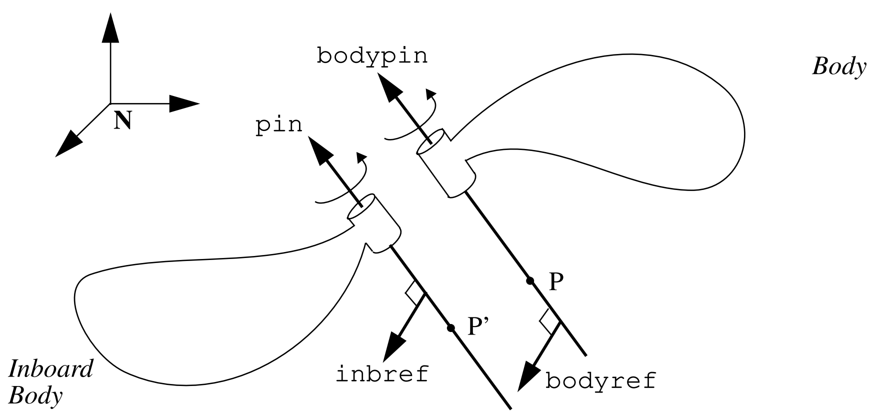
Assembled Pin Joint (1-D Rotational Joint)



There are no restrictions on the values of these vectors, except that `pin` can not have a length of zero. All vectors are specified when the bodies are in the reference configuration in which all coordinate frames are aligned. `Inbtojoint` and `bodytojoint` are vectors fixed in their respective bodies with their components using the assumed units of length. `Pin` is a vector fixed in both bodies defining the joint axis orientation. A positive increase in the coordinate defining the angle between the two bodies follows the right-hand-rule as defined by `pin`.

Figure R-4

Unassembled Pin Joint



For an unassembled loop pin joint shown in Figure R-4, you *may* also specify an inboard reference line `inbref`, an outboard pin `bodypin`, and an outboard reference line `bodyref`. `Inbref` is required to be perpendicular to `pin`, and `bodyref` is required to be perpendicular to `bodypin`. After assembly, the hinge points are collocated and the inboard and outboard pins are aligned. The angle between the reference lines is the value reported for the joint pseudo coordinate.

`inbref = ` *x x x*	Vector on inboard body defining the zero reference line.
`bodyref = ` *x x x*	Vector on body defining the zero reference line.
`bodypin = ` *x x x*	Vector defining joint axis on body.

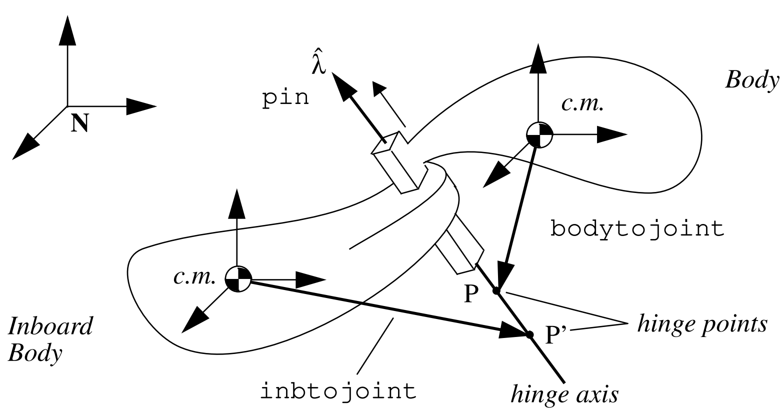
R11.5 Slider Joint

A slider joint is a one degree-of-freedom translational joint. A diagram of an assembled slider joint is shown in Figure R-5. If the joint is entered into the System Description File as assembled (or if you don't care what defaults SD/FAST chooses for an unassembled joint), you only need to specify three vectors:

`inbtojoint = ` *x x x*	Vector from inboard body c.o.m. to hinge point P'.
`bodytojoint = ` *x x x*	Vector from body c.o.m. to hinge point P.
`pin = ` *x x x*	Vector defining joint axis on inboard body and body.

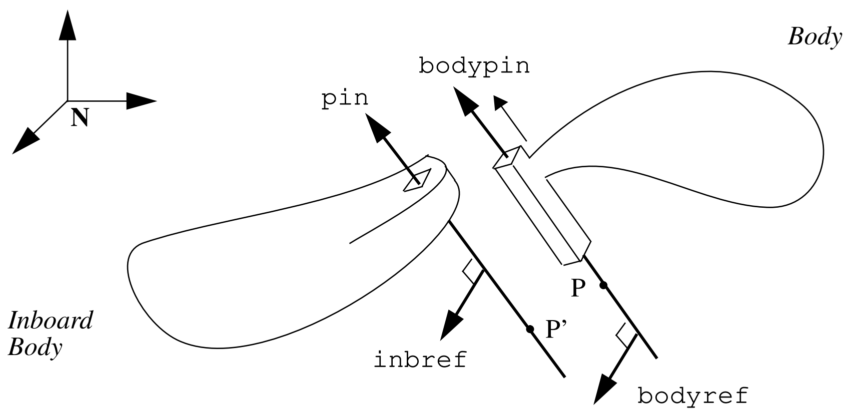
There are no restrictions on the values of these vectors, except that `pin` can not have a length of zero. All vectors are specified when the bodies are in the reference configuration in which all coordinate frames are aligned. `Inbtojoint` and `bodytojoint` are vectors fixed in their respective bodies with their components using the assumed units of length. `Pin` is a unit vector fixed in both bodies defining the joint axis orientation. A positive increase in the coordinate defining the distance between the two bodies occurs as hinge point P moves in the direction of `pin` relative to the inboard body.

Figure R-5 Assembled Slider Joint (1-D Translational Joint)



For an unassembled loop slider joint shown in Figure R-5, you *may* also specify an inboard reference line `inbref`, an outboard pin `bodypin`, and an outboard reference line `bodyref`. `Inbref` is required to be perpendicular to `pin`, and `bodyref` is required to be perpendicular to `bodypin`. After assembly, the inboard and outboard pins are aligned and the inboard and body reference lines are aligned (to set the relative orientation of the bodies, *not* to be a zeroing line for the pseudo-coordinate).

Figure R-6 Unassembled Slider Joint



Also, after assembly the displacement of P' from P has no component perpendicular to the axis, that is, P' lies on the sliding axis passing through P. The displacement (signed) between the hinge points is the value reported for the joint pseudo-coordinate. Note that assembly does not necessarily bring this displacement to zero as it does for the hinge points of purely rotational joints.

`inbref = ` *x x x*	Vector on inboard body defining the relative orientation.
`bodyref = ` *x x x*	Vector on body defining the relative orientation.
`bodypin = ` *x x x*	Vector defining joint axis on body.

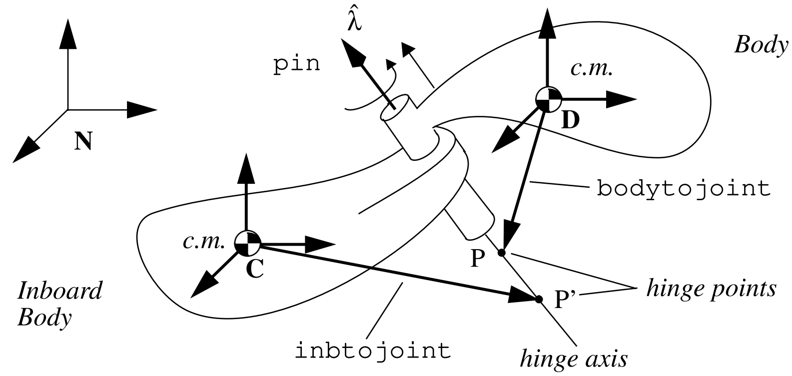
R11.6 Cylinder Joint

A cylinder joint is a two degree-of-freedom joint composed of a slider followed by a pin joint. Both the slider and pin are on the same axis. A diagram of an assembled cylinder joint is shown in Figure R-7. If the joint is entered into the system description file as assembled (or if you don't care what defaults SD/FAST chooses for an unassembled joint), you only need to specify three vectors:

`inbtojoint = ` *x x x*	Vector from inboard body c.o.m. to hinge point P'.
`bodytojoint = ` *x x x*	Vector from body c.o.m. to hinge point P.
`pin = ` *x x x*	Vector defining joint axis on inboard body and body.

Figure R-7

Assembled Cylinder Joint (1-D Translational Plus Coaxial 1-D Rotational Joint)

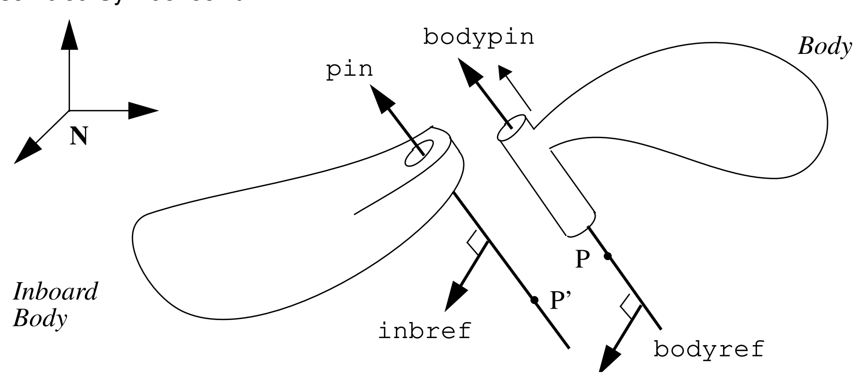


There are no restrictions on the values of these vectors, except that `pin` can not have a length of zero. All vectors are specified when the bodies are in the reference configuration in which all coordinate frames are aligned. `Inbtojoint` and `bodytojoint` are vectors fixed in their respective bodies with their components using the assumed units of length. `Pin` is a unit vector fixed in both bodies defining the joint axis orientation. A positive increase in the coordinate defining the distance between the two bodies occurs as hinge point P moves in the direction of `pin` relative to the inboard body. A positive increase in the coordinate defining the angle between the two bodies follows the right-hand-rule as defined by `pin`.

The first (lower-numbered) coordinate is the translational one, and the second is rotational. There is no explicit "reverse" cylinder joint. If your physical joint has the translational and rotational components reversed, you can still use the joint as defined here. Because the rotational and translational axes are identical, a rotation does not re-orient the translational axis. That means that the two coordinates are independent and could have been modeled in either order. Just change the order and sign of *your interpretation* of the coordinates to obtain the effect of a reverse cylinder joint.

Figure R-8

Unassembled Cylinder Joint



For an unassembled loop cylinder joint shown in Figure R-7, you *may* also specify an inboard reference line `inbref`, an outboard pin `bodypin`, and an outboard reference line `bodyref`. `Inbref` is required to be perpendicular to `pin`, and `bodyref` is required to be perpendicular to `bodypin`. After assembly, the inboard and outboard pins are aligned. P' lies on the sliding axis passing through P. The displacement (signed) between the hinge points is the value reported for the joint translational pseudo coordinate. The angle between the reference lines is the value reported for the second joint pseudo coordinate.

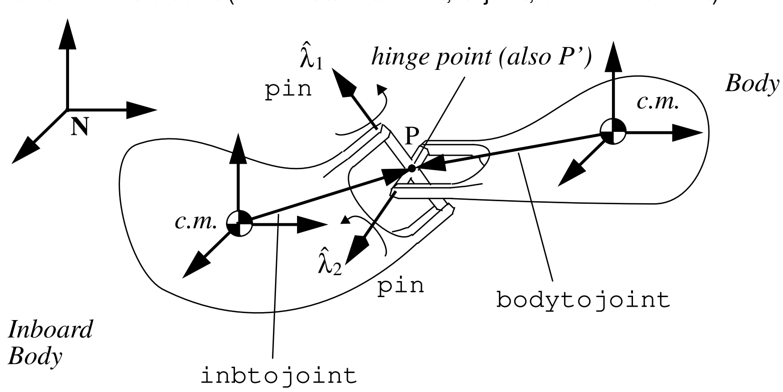
`inbref = ` *x x x*	Vector on inboard body defining the zero reference line.
`bodyref = ` *x x x*	Vector on body defining the zero reference line.
`bodypin = ` *x x x*	Vector defining joint axis on body.

R11.7 Universal (Hooke's) Joint

A Universal joint (U-joint) is a two degree-of-freedom joint composed of two pin joints. Both the pins intersect at hinge point P (equivalent to P'). A diagram of an assembled universal joint is shown in Figure R-9. If the joint is entered into the system description file as assembled (or if you don't care what defaults SD/FAST chooses for an unassembled joint), you only need to specify four vectors:

`inbtojoint = ` *x x x*	Vector from inboard body c.o.m. to hinge point P'.
`bodytojoint = ` *x x x*	Vector from body c.o.m. to hinge point P.
`pin = ` *x x x*	Vector defining joint axis fixed on inboard body.
`pin = ` *x x x*	Vector defining joint axis fixed on body and w.r.t the first pin.

Figure R-9 Assembled Universal Joint (2-D Rotational Joint, U-joint, or Hooke's Joint)



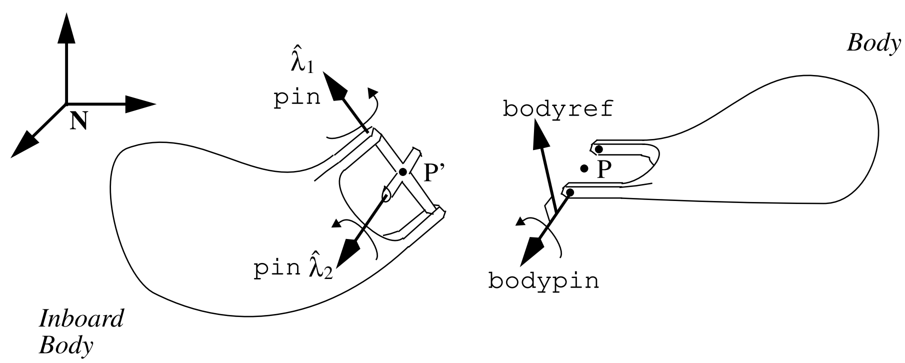
The restrictions on the values of these vectors are that `pin` cannot have a length of zero and the two pins cannot be collinear. Furthermore, if the joint is a loop joint, the two pins must be perpendicular. All vectors are specified when the bodies are in the reference configuration in which all coordinate frames are aligned. `Inbtojoint` and `bodytojoint` are vectors fixed in their respective bodies with their components using the assumed units of length.

Note that the order that the pins are entered is important. The first `pin` is a vector fixed in the inboard body defining the joint axis orientation. The second `pin` is a vector fixed with respect to the first pin and w.r.t. the outboard body.[10] A positive increase in the coordinate defining the angle between the two bodies follows the right-hand-rule as defined by each `pin`.

For an unassembled loop universal joint shown in Figure R-9, you *may* also specify an outboard pin `bodypin`, and an outboard reference line `bodyref` (no `inbref` is needed since the first inboard `pin` is used instead). `Bodyref` is required to be perpendicular to `bodypin`. After assembly, the hinge points are collocated and the second `pin` and `bodypin` are aligned. The first pseudo coordinate is the angle between the original orientation of the second inboard `pin` and the current orientation of the outboard `bodypin`. The second pseudo coordinate is the angle between the first inboard `pin` and `bodyref`.

`bodyref` = *x x x*	Vector on body defining the zero reference line.
`bodypin` = *x x x*	Vector defining joint axis on body.

Figure R-10 Unassembled Universal Joint



R11.8 Planar Joint

A planar joint is a three degree-of-freedom joint composed of two sliders followed by a pin joint. (A "reverse" planar joint is also available, in which the first axis is rotational and the second and third are translational.) All the pins intersect at hinge point P' on the inboard body. Normally the two sliders are perpendicular to one another and the pin axis is perpendicular to both of the sliders (the perpendicularity is required for a loop planar joint, but not a tree planar joint). As an example, a planar joint describes the relationship between a hockey puck and the ice (assuming the hockey puck remains in contact with the ice). A diagram of an assembled planar joint is shown in Figure R-11. If the joint is entered into the system description file as assembled (or if you don't care what defaults SD/FAST chooses for an unassembled joint), you only need to specify five vectors:

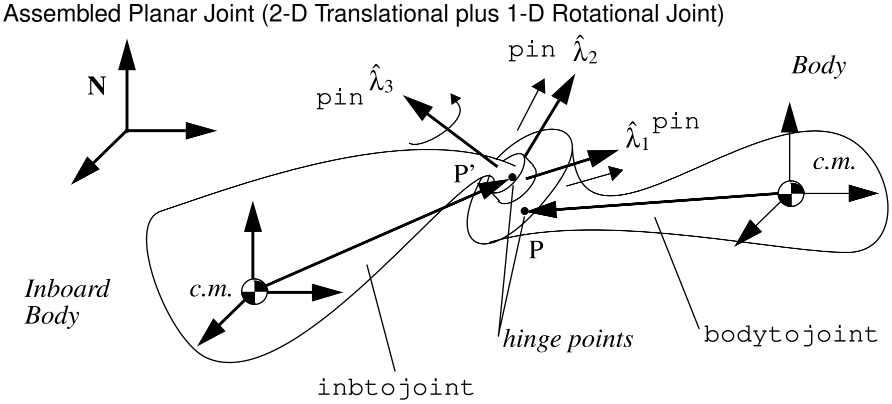
10. Since both pins are fixed with respect to each other, another way of looking at the joint would have both pins fixed to an inner massless gimbal body, as suggested by the figures.

`inbtojoint   =` *x x x*	Vector from inboard body c.o.m. to hinge point P’.
`bodytojoint =` *x x x*	Vector from body c.o.m. to hinge point P.
`pin =` *x x x*	Vector defining slider joint axis fixed on inboard body.
`pin =` *x x x*	Vector defining second sliding axis fixed on inboard body.
`pin =` *x x x*	Vector defining pin joint axis fixed on inboard body and body.

The restrictions on the values of these vectors are that no `pin` can have a length of zero and the two sliding pins can not be collinear. Furthermore, if the joint is a loop joint, all pins must be perpendicular with $\hat{\lambda}_3 = \hat{\lambda}_1 \times \hat{\lambda}_2$ (a right-handed set). All vectors are specified when the bodies are in the reference configuration in which all coordinate frames are aligned. `Inbtojoint` and `bodytojoint` are vectors fixed in their respective bodies with their components using the assumed units of length.

Figure R-11

Assembled Planar Joint (2-D Translational plus 1-D Rotational Joint)

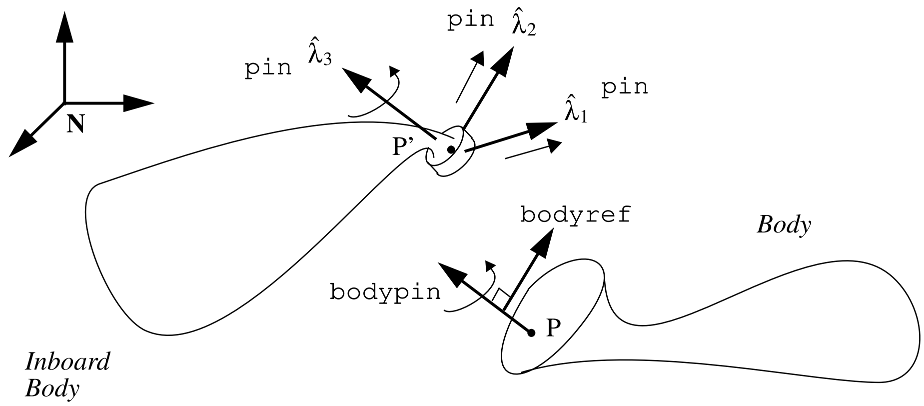


Note that the order that the pins are entered is important. The first `pin` is a vector fixed in the inboard body defining the orientation of the first sliding axis. The second `pin` is a vector fixed with respect to the first pin which also defines the plane of sliding. The third `pin` is a vector fixed w.r.t. to the sliding plane and is a rotational axis. A positive increase in each coordinate defining the distance between the two bodies occurs as hinge point P moves in the direction of each of the first two `pins` relative to the inboard body. A positive increase in the coordinate defining the angle between the two bodies follows the right-hand-rule as defined by the third `pin`.

For an unassembled loop planar joint shown in Figure R-11, you *may* also specify an outboard pin `bodypin`, and an outboard reference line `bodyref` (no `inbref` is needed since an inboard `pin` is used instead). `Bodyref` is required to be perpendicular to `bodypin`. After assembly, the hinge points are both in a plane formed by the first two `pins`, and `bodypin` is aligned with the third inboard `pin`. The first two pseudo-coordinates are the vector from the inboard hinge point to the outboard hinge point, projected along the first two axes. The third pseudo coordinate is the angle between the first `pin` and `bodyref`.

`bodyref =` *x x x*	Vector on body defining the zero reference line.
`bodypin =` *x x x*	Vector defining joint axis on body.

Figure R-12 Unassembled Planar Joint

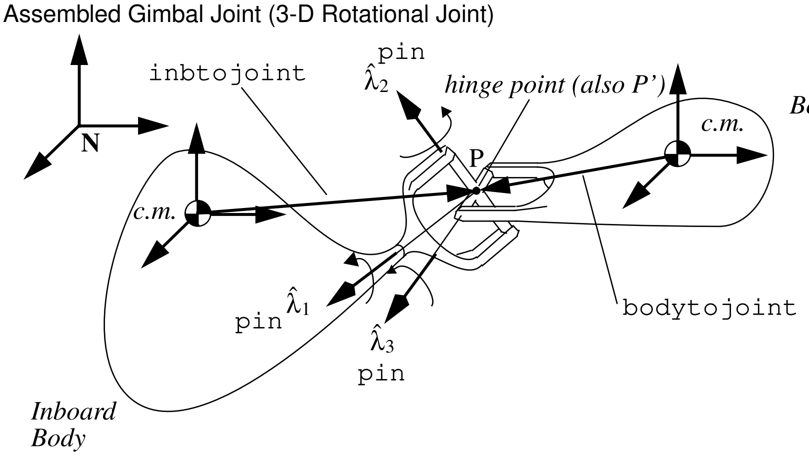


R11.9 Gimbal Joint

A gimbal joint is a three degree-of-freedom joint composed of three pin joints. All the pins intersect at hinge point P (equivalent to P'). A diagram of an assembled gimbal joint is shown in Figure R-13. If the joint is entered into the System Description File as assembled (or if you don't care what defaults SD/FAST chooses for an unassembled joint), you only need to specify five vectors:

`inbtojoint = ` *x x x*	Vector from inboard body c.o.m. to hinge point P'.
`bodytojoint = ` *x x x*	Vector from body c.o.m. to hinge point P.
`pin = ` *x x x*	Vector defining joint axis fixed on inboard body.
`pin = ` *x x x*	Vector defining joint axis fixed w.r.t the first pin.
`pin = ` *x x x*	Vector defining joint axis fixed on body and w.r.t the 2nd pin.

Figure R-13 Assembled Gimbal Joint (3-D Rotational Joint)



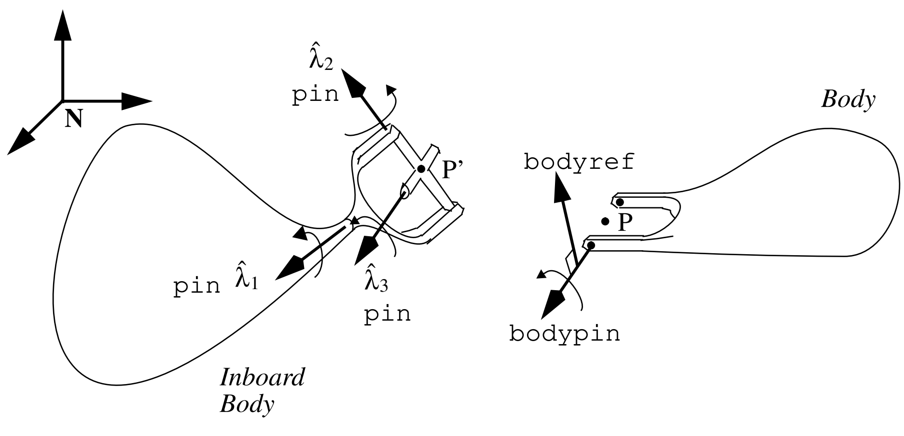
The restrictions on the values of these vectors are that no `pin` can have a length of zero and no two of the three pins can be successively collinear, i.e., neither $\hat{\lambda}_1 \parallel \hat{\lambda}_2$ nor $\hat{\lambda}_2 \parallel \hat{\lambda}_3$. Furthermore, if the joint is a loop joint, the pins must be successively perpendicular, i.e., $\hat{\lambda}_1 \perp \hat{\lambda}_2$ and $\hat{\lambda}_2 \perp \hat{\lambda}_3$. All vectors are specified when the bodies are in the reference configuration in which all coordinate frames are aligned. `Inbtojoint` and `bodytojoint` are vectors fixed in their respective bodies with their components using the assumed units of length.

Note that the order that the pins are entered is important. The first `pin` is a vector fixed in the inboard body defining the orientation of the first axis. The second `pin` is a vector fixed with respect to the first `pin`, and the third `pin` is a vector fixed w.r.t. the second `pin` and w.r.t. the outboard body. If the three pins form a right-handed triad such that $\hat{\lambda}_3 = \hat{\lambda}_1 \times \hat{\lambda}_2$, then the gimbal is equivalent to body-fixed 1-2-3 Euler angles. If $\hat{\lambda}_1 \parallel \hat{\lambda}_3$, then we have body-fixed 1-2-1 Euler angles, and so on.

Gimbal joints suffer from a singularity called "gimbal lock." This condition occurs when the first and third gimbal axes are aligned. In a physical gimbal joint, this is a configuration in which the normally three DOF joint allows only two DOFs. In SD/FAST, where the internal gimbals are modeled as massless, this is a singular configuration in which any torque applied to the first pin would result in infinite acceleration of one of the intermediate gimbals. Consequently, it is considered an error for a gimbal joint to be in this configuration. You should choose your axes carefully to make sure that the gimbal will not have to operate at or near gimbal lock during any movement of the system. If unrestricted motion is required, a ball joint should be used rather than a gimbal.

The inner gimbal bodies shown in Figure R-13 are modeled as massless. If you want to model a system with gimbals having mass, you must build it up with pin joints and bodies.

Figure R-14 Unassembled Gimbal Joint



For an unassembled loop gimbal joint shown in Figure R-14, you *may* also specify an outboard pin `bodypin`, and an outboard reference line `bodyref` (no `inbref` is needed since an inboard `pin` is used instead). `Bodyref` is required to be perpendicular to `bodypin`. After assembly, the hinge points are collocated. There are three pseudo-coordinates described as follows.

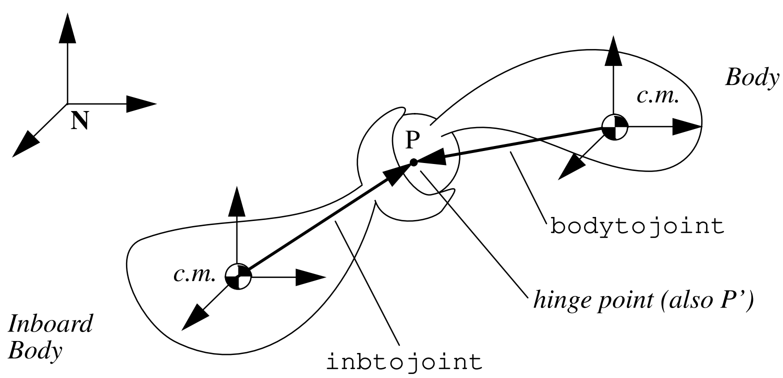
In the reference configuration, the pins are given as expressed in the inboard body's frame, which is itself aligned with the global frame. Let $\mathbf{p}_2$ and $\mathbf{p}_3$ be vectors expressed in the gimbal's first intermediate frame which initially were aligned with the second and third pins $\hat{\lambda}_2$ and $\hat{\lambda}_3$, respectively. The first pseudo coordinate is the angle between $\hat{\lambda}_2$ and $\mathbf{p}_2$. The second pseudo-coordinate is the angle between $\mathbf{p}_3$ and the outboard `bodypin`. The third pseudo-coordinate is the angle between $\mathbf{p}_2$ and the outboard `bodyref` line.[11]

`bodyref` = *x x x*	Vector on body defining the zero reference line.
`bodypin` = *x x x*	Vector defining joint axis on body.

R11.10 Ball Joint

A ball joint shown in Figure R-15 allows no pins and no reference axes. Its motion is always modeled as a relationship between the local coordinate frames of the two connected bodies. Where axes are desired, a gimbal joint should be used in place of a ball. Ball joint orientations (whether used alone or in a sixdof joint) are represented by quaternions (Euler parameters) for numerical stability (see Section R6). Some utility routines are provided for conversion among Euler parameters, direction cosines and Euler angles. A drawback of quaternions is that they cannot capture "tumbling" motion, that is, they represent only the relative orientations between the connected bodies not how many times they may have rotated to end up in their current orientations. If tumbling is to be modeled, it must either be tracked by user-written code, or a gimbal joint should be used in place of the ball.

Figure R-15 Assembled Ball Joint (3-D Rotational Joint without axes)



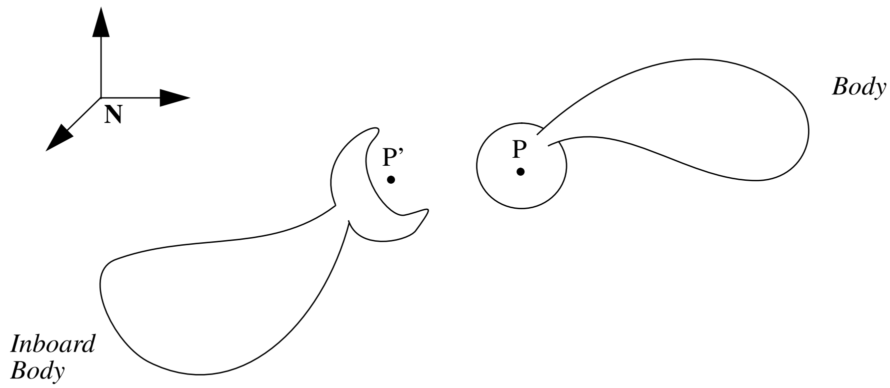
No pins or reference lines are allowed for this joint. After assembly, the hinge points are collocated. The four pseudo coordinates are the Euler parameters expressing the relative orientation of the outboard body's local frame with respect to the inboard body's local frame.

`inbtojoint` = *x x x*	Vector from inboard body c.o.m. to hinge point P'.
`bodytojoint` = *x x x*	Vector from body c.o.m. to hinge point P.

11. While this definition of the pseudo-coordinates is somewhat awkward, it is actually a very natural one in practice. This is best seen by experimenting with a loop gimbal joint to compare the reported pseudo-coordinates with your expectations.

An unassembled ball joint is shown in Figure R-16.

Figure R-16 Unassembled Ball Joint



R11.11 Bearing Joint

A bearing joint is a four degree-of-freedom joint composed of a slider followed by a gimbal joint. (A "reverse" bearing joint is available which is equivalent to a gimbal joint followed by a sliding joint.) One of the most common uses for this joint is for redundancy-free modeling of shaft bearings, hence the name. This joint is like a race of ball bearings through which a shaft can translate as well as rotate and tip (slightly). Since this joint contains a gimbal, it has the same singularity as a gimbal joint and is not intended for use when the outboard body can attain arbitrary orientations with respect to the inboard body. It is not, however, restricted to small angles; it just cannot be used near the "gimbal lock" configuration.

All the pins intersect at hinge point P'. A diagram of an assembled bearing joint is shown in Figure R-17. If the joint is entered into the System Description File as assembled (or if you don't care what defaults SD/FAST chooses for an unassembled joint), you only need to specify five vectors:

`inbtojoint` = *x x x*	Vector from inboard body c.o.m. to hinge point P'.
`bodytojoint` = *x x x*	Vector from body c.o.m. to hinge point P.
`pin` = *x x x*	Vector defining joint axis fixed on inb. body (rot & trans).
`pin` = *x x x*	Vector defining joint axis fixed w.r.t the first pin.
`pin` = *x x x*	Vector defining joint axis fixed on body and w.r.t the 2nd pin.

The restrictions on the values of these vectors are the same as for a gimbal joint. All vectors are specified when the bodies are in the reference configuration in which all coordinate frames are aligned. `Inbtojoint` and `bodytojoint` are vectors fixed in their respective bodies with their components using the assumed units of length.

Note that the order that the pins are entered is important. The first `pin` is a vector fixed in the inboard body defining the orientation of the first axis. This pin is used as the bearing's translational axis as well as its first rotational axis. The second `pin` is a vector fixed with respect to the first `pin`, and the third `pin` is a vector fixed w.r.t. the second `pin` and w.r.t. the outboard body.

Figure R-17 Assembled Bearing Joint (1-D Translational plus 3-D Rotational Joint)

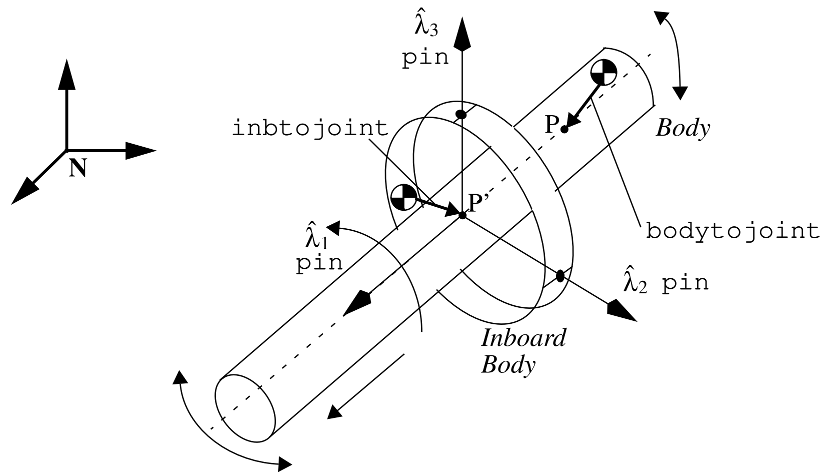
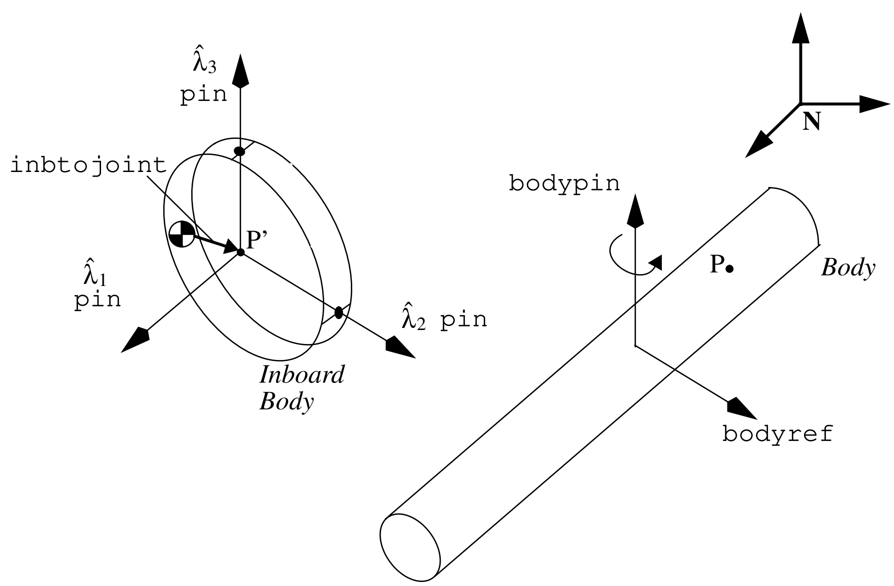


Figure R-18 Unassembled Bearing Joint



For an unassembled loop bearing joint shown in Figure R-18, you *may* also specify an outboard pin `bodypin`, and an outboard reference line `bodyref` (no `inbref` is needed since an inboard `pin` is used instead). `Bodyref` is required to be perpendicular to `bodypin`. After assembly, the hinge points will both lie along the translational (first) axis, but will not necessarily be collocated. There are four pseudo-coordinates. The first is the (signed) displacement giving the displacement of P from P' along the translational axis. The remaining three have the same definition as the three pseudo-coordinates of a loop gimbal joint (see above).

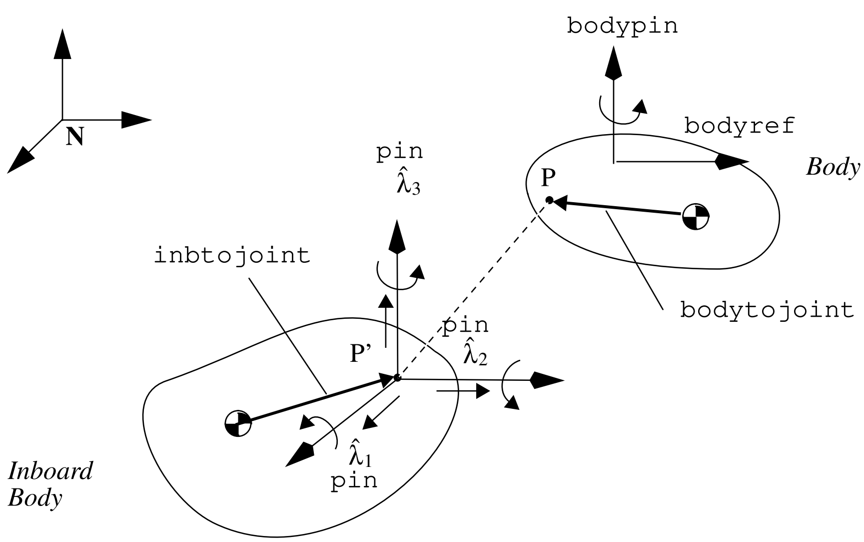
`bodyref = x x x`	Vector on body defining the zero reference line.
`bodypin = x x x`	Vector defining joint axis on body.

R11.12 Bushing Joint

The bushing joint shown in Figure R-19 represents unrestricted motion between two bodies, and is actually implemented as three sliding joints followed by a gimbal joint. (A "reverse" bushing joint is available which is equivalent to a gimbal joint followed by three sliding joints.) One of the most common uses for this joint is as the underlying joint connecting two bodies which are physically connected by a bushing. Bushing forces are easily computed with respect to the bushing joint axes, which form the coordinate frame for the bushing.

Since this joint contains a gimbal, it has the same singularity as a gimbal joint and is not intended for use when the outboard body can attain arbitrary orientations with respect to the inboard body. It is not, however, restricted to small angles; it just cannot be used near the "gimbal lock" configuration. If unrestricted orientations are required, you should use a free joint instead of a bushing (see below).

Figure R-19 Bushing Joint (3-D Translational plus 3-D Gimbal Joint)



If the bushing joint is a loop joint, then three mutually perpendicular axes are required for this joint, forming a right-handed set, i.e., $\hat{\lambda}_3 = \hat{\lambda}_1 \times \hat{\lambda}_2$. These axes are the translational axes and are also used as the reference configuration orientations for the rotational axes. If the bushing joint is a tree joint, the perpendicularity requirement is dropped, but no two joint axes may be collinear.

Note that even if this is a loop bushing, *it is always assembled*. The first three pseudo-coordinates are the vector from the inboard hinge point to the outboard hinge point, projected along each of the translational axes. The last three pseudo coordinates are defined exactly as for a loop gimbal joint (see above).

The vectors are required for any bushing joint, tree or loop:

`inbtojoint  = ` *x x x*	Vector from inboard body c.o.m. to hinge point P'.
`bodytojoint = ` *x x x*	Vector from body c.o.m. to hinge point P.
`pin = ` *x x x*	Vectors defining sliding axes. All are fixed on the inboard body.
`pin = ` *x x x*	
`pin = ` *x x x*	

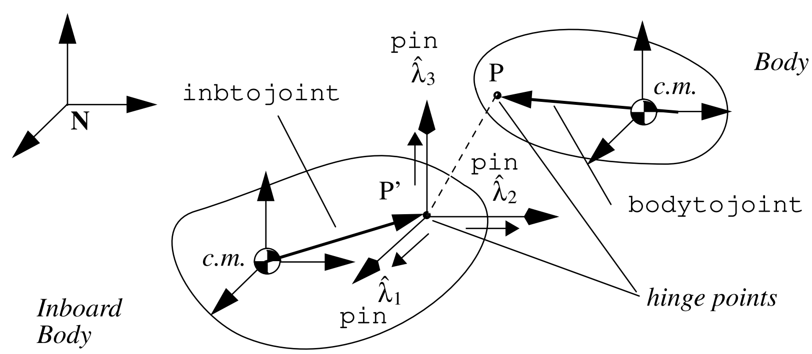
These vectors are optional for loop bushings:

`bodyref = ` *x x x*	Vector on body defining the zero reference line.
`bodypin = ` *x x x*	Vector defining joint axis on body.

R11.13 Free (6dof) Joint

The free joint (a.k.a. 6dof joint) shown in Figure R-20 represents unrestricted motion between two bodies, implemented as three sliding joints followed by a ball joint. (A "reverse" free joint is available which is equivalent to a ball joint followed by three sliding joints.) A free joint is used by default between the base body of a free-flying system and the ground. Note that this joint provides the identical motion as a "bushing" joint (see above) but a free joint has no singularities while the bushing joint can suffer from gimbal lock.

Figure R-20 Free Joint (3-D Translational plus 3-D Ball Joint)



If the free joint is a loop joint, then three mutually perpendicular axes are required for this joint, forming a right-handed set, i.e., $\hat{\lambda}_3 = \hat{\lambda}_1 \times \hat{\lambda}_2$. These axes are all translational. If the free joint is a tree joint, the perpendicularity requirement is dropped, but no two joint axes may be collinear.

When used as a loop joint, note that the free joint *is always assembled*. The first three pseudo coordinates are the vector from the inboard hinge point to the outboard hinge point, projected along each of the axes. The last four pseudo coordinates are the Euler parameters expressing the relative orientation of the outboard body's local frame with respect to the inboard body's local frame.

`inbtojoint = ` *x x x*	Vector from inboard body c.o.m. to hinge point P'.
`bodytojoint = ` *x x x*	Vector from body c.o.m. to hinge point P.
`pin = ` *x x x*	Vectors defining sliding axes. All are fixed on the
`pin = ` *x x x*	inboard body.
`pin = ` *x x x*	

No `bodypin` or `bodyref` vectors are allowed for a free joint.

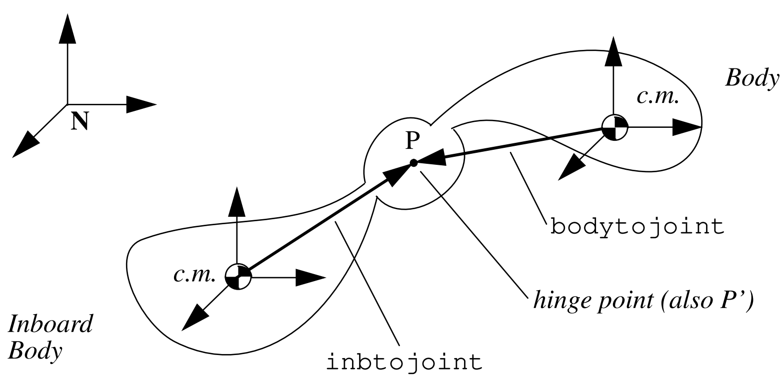
R11.14 Weld Joint

A weld is a zero DOF joint which simply joins two bodies together as shown in Figure R-21.

No pins are allowed for the tree form of this joint. However, for loop weld joints you may specify an inboard and outboard pin and an inboard and outboard reference line. After assembly, the hinge points are collocated, the inboard and outboard pins are aligned, and the inboard and outboard reference lines are aligned. There are no pseudo coordinates for this joint.

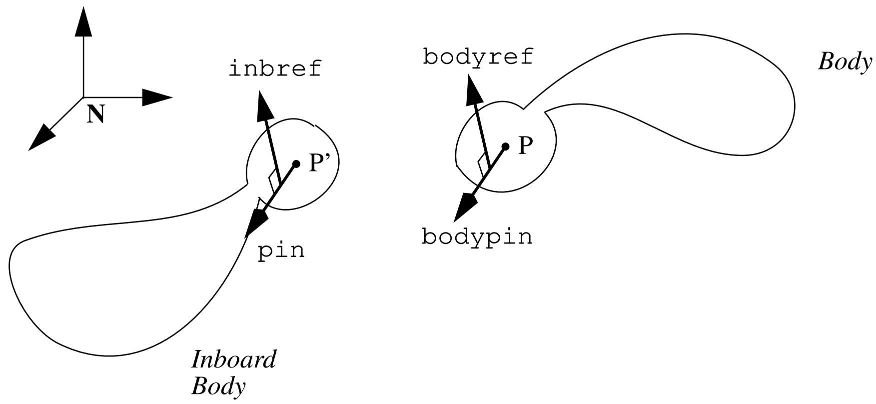
`inbtojoint = ` *x x x*	Vector from inboard body c.o.m. to hinge point P'.
`bodytojoint = ` *x x x*	Vector from body c.o.m. to hinge point P.
`pin = ` *x x x*	Vector defining joint "axis" on inboard body and body.
`bodypin = ` *x x x*	Vector defining joint "axis" on body.
`inbref = ` *x x x*	Vector on inboard body defining the reference line.
`bodyref = ` *x x x*	Vector on body defining the reference line.

Figure R-21 Assembled Weld Joint (Zero DOF Joint)



An unassembled weld joint is shown in Figure R-21.

Figure R-22 Unassembled Weld Joint



R12 Kinematic Information

Each body in the system has linear and angular position, velocity and acceleration. Every joint axis also has a position, velocity, and acceleration which may be linear or angular depending on the joint type. In addition, the system as a whole has angular and linear momentum, and kinetic energy. There are aggregate mass properties for the system as well (total system mass, center of mass location, and system inertia matrix). SD/FAST generates a collection of routines for use in retrieving these values, which we collectively call *kinematic information*. In addition, there are routines for manipulating this information. These routines are described below.

R12.1 Body-Specific Information

The routines in this section return body kinematic information as last computed by `SDSTATE()` (for positions and velocities) or `SDDERIV()` or `SDRESID()` (for accelerations). These are data access routines, that is, they do very little computation; rather, they report information that has already been calculated. Thus they are relatively inexpensive to call.

It is allowed for the body number to be the ground body (0 for Fortran, -1 for C), in which case the routines simply return the appropriate response, e.g. the angular velocity is zero.

```
SUBROUTINE SDPOS(BODY,POINT,LOC)
INTEGER BODY
DOUBLE PRECISION POINT(3),LOC(3)

SUBROUTINE SDVEL(BODY,POINT,VEL)
INTEGER BODY
DOUBLE PRECISION POINT(3),VEL(3)

SUBROUTINE SDACC(BODY,POINT,ACC)
INTEGER BODY
DOUBLE PRECISION POINT(3),ACC(3)

SUBROUTINE SDORIENT(BODY,DIRCOS)
INTEGER BODY
DOUBLE PRECISION DIRCOS(3,3)

SUBROUTINE SDANGVEL(BODY,ANGVEL)
INTEGER BODY
DOUBLE PRECISION ANGVEL(3)

SUBROUTINE SDANGACC(BODY,ANGACC)
INTEGER BODY
DOUBLE PRECISION ANGACC(3)
```

`SDPOS()` returns the location with respect to the global origin of a point on a body. The point is expressed in the body-local frame so point (0,0,0) yields the location of the center of mass. The returned location is expressed in the ground frame.

`SDVEL()` returns the linear velocity with respect to ground of a point on a body. The point is expressed in the body-local frame so point (0,0,0) yields the velocity of the center of mass. The returned velocity is expressed in the ground frame.

`SDACC()` returns the linear acceleration with respect to ground of a point on a body. The point is expressed in the body-local frame so point (0,0,0) yields the acceleration of the center of mass. The returned acceleration is expressed in the ground frame.

`SDORIENT()` returns the orientation of a body with respect to the ground frame, expressed as a direction cosine mapping a ground frame vector into a body frame vector. That is, if ${}^{N}C^{B}$ is the returned direction cosine and $\mathbf{v}^N$ a vector expressed in the ground frame, then $\mathbf{v}^B = \mathbf{v}^N \bullet {}^{N}C^{B}$ is that same vector expressed in the body frame.

`SDANGVEL()` returns a body's angular velocity with respect to the ground frame, but expressed in the body-local frame.

`SDANGACC()` returns the angular acceleration of the body frame with respect to the ground frame, but expressed in the body frame.

R12.2 System-Wide Information

These routines return system-global information, using information computed by the previous call to `SDSTATE()`. They are computational routines; that is, they are relatively expensive to call so they should be called only once per integration communication interval.

```
SUBROUTINE SDMOM(LM,AM,KE)
DOUBLE PRECISION LM(3),AM(3),KE


SUBROUTINE SDSYS(MTOT,CM,ICM)
DOUBLE PRECISION MTOT,CM(3),ICM(3,3)
```

`SDMOM()` returns absolute linear and angular momentum for the system with respect to ground, in the ground frame, in `LM` and `AM`. Angular momentum is referred to the system center of mass. `KE` is total system kinetic energy from rotation and translation.

`SDSYS()` returns the total system mass in `MTOT`, the system center of mass location with respect to ground in `CM` and the system inertia matrix for the system center of mass, measured in the ground frame, in `ICM`.

R12.3 Data Manipulation

These routines are used to manipulate information obtained from the routines above or from other sources.

```
SUBROUTINE SDTRANS(FROMBODY,VEC,TOBODY,OVEC)
INTEGER FROMBODY,TOBODY
DOUBLE PRECISION VEC(3),OVEC(3)

SUBROUTINE SDQUAT2DC(E1,E2,E3,E4,DIRCOS)
DOUBLE PRECISION E1,E2,E3,E4,DIRCOS(3,3)

SUBROUTINE SDDC2QUAT(DIRCOS,E1,E2,E3,E4)
DOUBLE PRECISION DIRCOS(3,3),E1,E2,E3,E4

SUBROUTINE SDANG2DC(A1,A2,A3,DIRCOS)
DOUBLE PRECISION A1,A2,A3,DIRCOS(3,3)

SUBROUTINE SDDC2ANG(DIRCOS,A1,A2,A3)
DOUBLE PRECISION DIRCOS(3,3),A1,A2,A3
```

`SDTRANS()` transforms a vector from one frame into another. Despite its name, this is a pure rotational transformation, not a translation. The starting frame is specified by a body number in `FROMBODY`. The input vector `VEC` is assumed to be expressed in that frame. The desired frame is given by `TOBODY`, and the resulting vector is returned in `OVEC`. It is fine for `OVEC` and `VEC` to be the same Fortran or C array. The ground frame is specified as body `0` (in Fortran) or `-1` (in C).

`SDQUAT2DC()` and `SDDC2QUAT()` convert quaternions (Euler parameters) to/from direction cosines. `SDQUAT2DC()` normalizes the quaternions (internally) before using them. (0,0,0,0) is treated as (0,0,0,1).

`SDANG2DC()` and `SDDC2ANG()` perform the same service for body-fixed 1-2-3 Euler angles. Note that although “tumbling” motion can be captured by Euler angles, it is lost in the conversion either to quaternions or direction cosines. Euler angles with tumbling information can be obtained directly only through the use of gimbal-containing joints.

C Language

```
sdpos(body,point,loc)
int body;
double point[3],loc[3];

sdvel(body,point,vel)
int body;
double point[3],vel[3];
```

```
sdacc(body,point,acc)
int body;
double point[3],acc[3];

sdorient(body,dircos)
int body;
double dircos[3][3];

sdangvel(body,angvel)
int body;
double angvel[3];

sdangacc(body,angacc)
int body;
double angacc[3];

sdmom(lm,am,ke)
double lm[3],am[3],*ke;

sdsys(mtot,cm,icm)
double *mtot,cm[3],icm[3,3];

sdtrans(frombody,vec, tobody,ovec)
int frombody,tobody;
double vec[3],ovec[3];

sdquat2dc(e1,e2,e3,e4,dircos)
double e1,e2,e3,e4,dircos[3][3];

sddc2quat(dircos,e1,e2,e3,e4)
double dircos[3][3],*e1,*e2,*e3,*e4;

sdang2dc(a1,a2,a3,dircos)
double a1,a2,a3,dircos[3][3];

sddc2ang(dircos,a1,a2,a3)
double dircos[3][3],*a1,*a2,*a3;
```

R13 Mass Properties

Each body in the system must have a mass and inertia matrix specified. In this section we discuss the definitions of these parameters as used by SD/FAST, and the restrictions on their allowable values. The special cases of zero mass (massless or inertialess bodies) and infinite mass (ground) are also discussed here. Section R19 shows how mass, inertia, and ground are entered into the System Description File.

R13.1 Mass

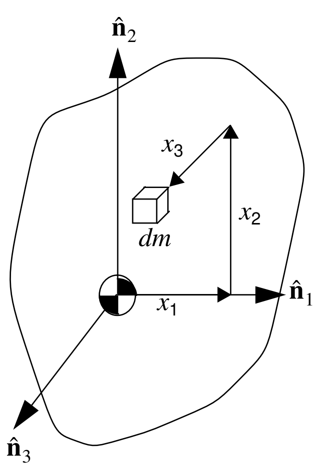
Mass is a scalar quantity. Generally speaking, there are no restrictions on the values allowable for a body's mass. However, a problem can become numerically poorly conditioned if bodies of widely varying mass appear in the same problem.

If your model contains bodies whose masses differ by a factor of about 10^6 or more, you should consider whether a different model might be more appropriate. You could model some of the bodies as massless. Or, you might break the problem into separate analyses with the more massive bodies treated as ground in an analysis of a model containing only the smaller bodies. See Section R13.3 and Section R13.4 below for more information.

R13.2 Inertia

The inertia matrix (alternatively dyadic or tensor of second order) of each body is specified about the *mass center* of that body, for that body in the *reference configuration*. For any coordinate frame with unit vector subscripts 1,2,3 (remember all the coordinate frames are aligned in the reference configuration) the inertia matrix **I** is given by:

$$\mathbf{I} = \begin{bmatrix} I_{11} & I_{12} & I_{13} \\ I_{21} & I_{22} & I_{23} \\ I_{31} & I_{32} & I_{33} \end{bmatrix}$$



where:

$$\begin{array}{ccc} I_{11} \equiv \int (x_2^2 + x_3^2) dm & I_{12} \equiv -\int x_1 x_2 dm & I_{13} \equiv -\int x_1 x_3 dm \\ I_{21} \equiv I_{12} & I_{22} \equiv \int (x_1^2 + x_3^2) dm & I_{23} \equiv -\int x_2 x_3 dm \\ I_{31} \equiv I_{13} & I_{32} \equiv I_{23} & I_{33} \equiv \int (x_1^2 + x_2^2) dm \end{array}$$

and x_i is the distance from the mass center to mass element dm along the axis in the direction of $\hat{\mathbf{n}}_i$. Note that the minus sign in the definitions of the products of inertia may be a different convention from your sources of mass property information.[13] To improve your simulation efficiency, try to pick reference configurations in which products of inertia are zero for as many of the bodies as possible.

13. Dynamicists usually use the minus sign while some structural analysts use a positive sign convention. Be careful!

R13.3 Massless and Inertialess Bodies

Massless and inertialess bodies can be used as fictitious bodies to aid construction of custom joints. In addition, when some bodies in a system have masses or inertias a great deal smaller than others, it often makes sense to treat them as though they had no mass or inertia at all. When a body is entered with a zero mass or principal inertia, the SD/FAST symbol manipulator can greatly simplify the equations of motion, and can avoid most of the numerical conditioning problems which can occur in problems containing bodies of widely varying mass and inertia. There is, however, an important restriction on the use of this feature.

Recall that SD/FAST models are structured as a set of bodies and joints which form a topological "tree" plus an additional set of joints (loop joints) which connect branches of the tree to form topological loops. Looking only at the tree system with the loop joints removed, a massless or inertialess body is *not allowed* if there exists any joint axis (or joint degree of freedom) on which an applied load would accelerate only the massless body or a set of massless bodies, or would produce a rotational acceleration only on a body or set of bodies with no inertia about the axis of rotation. Since this would produce infinite accelerations, the model would become numerically ill-conditioned and any results obtained from analysis using this model would be invalid.

Another way of expressing the above restriction is to say that a massless or inertialess body is only allowed if any force applied to the massless body or torque applied in the inertialess direction "drags along" a body with mass or inertia as well, when considering only the tree joints in the system. A body with neither mass nor inertia is *never* allowed as the terminal body of a tree branch, since the lack of an outboard body in that case means that a load applied at the body's inboard joint would produce infinite acceleration. A non-terminal massless or inertialess body can also be illegal if a load on either its inboard or outboard joints could cause infinite acceleration.

The above restriction applies without regard to the presence or absence of prescribed motion. That is, even if a massless body's motion is prescribed so that it can never undergo any large accelerations, the model will be numerically intractable if the massless body *could have* undergone these accelerations if the prescribed motion were removed.

It is important to note that the presence of massless or inertialess bodies should be taken into account when choosing where to break loops when modeling the system. You must avoid making choices which can produce a terminal massless or inertialess body in the resulting tree system. This is discussed more in Section R11.1.

R13.4 Ground

At the opposite extreme from massless bodies is ground, or the inertial frame. Ground is considered to be of "infinite mass". Again, the SD/FAST symbol manipulator can produce much better equations if extremely massive bodies (with respect to other bodies in the system) can be modeled as ground.

Generally, engineering judgement is a sufficient guide to the use of ground to represent massive bodies in a system. For example, when modeling an automobile, the Earth can safely be treated as ground. In some cases, the decision can be more difficult. For a rigorous treatment, see *Dynamics: Theory and Applications*, Kane, T.R., and Levinson, D.A., McGraw-Hill, New York, 1985, pp. 158-169.

R14 Prescribed Motion

Normally the motion of a joint axis is calculated by SD/FAST as a function of the current system state and the forces acting on the system. Instead, the motion can be *prescribed*, usually as a function of time but in general as a function of time and some system states. When the motion is prescribed, SD/FAST calculates the actuator force or torque which would be required at that axis to produce the desired motion. (Use `SDGETHT()` to obtain the computed value; see Section R10.2.)

The System Description File must request prescribed motion if it is to be used. If a question mark is used in the specification(see Table R-3 on page R-108), then prescribed or free motion can be controlled at run time and in fact can be turned on and off during a simulation. When turning prescribed motion on, you must be careful to see that all constraint conditions are met so that no discontinuous change in position or velocity is required. A routine is provided for enabling and disabling prescribed motion.

Prescribed motion is implemented as a constraint, requiring one constraint equation (and consequently one multiplier) for each joint axis whose motion is to be prescribed. Routines are available for providing desired accelerations, velocities and positions to SD/FAST for prescribed axes. These routines must be called whenever system derivatives are to be computed with `SDDERIV()` or `SDRESID()`. Generally, all specifications of prescribed motion should be grouped together in the user-written `sdumotion()` routine, just as forces are grouped in `sduforce()`. See Section R15.3.2 for information on writing the `sdumotion()` routine.

When using the ODE formulation (as implemented in `SDDERIV()`), prescribed motion is supplied as an acceleration constraint. Velocity and position constraints are optionally provided to control initial conditions and to provide numerical stability. When using the DAE formulation (as implemented in `SDRESID()`), prescribed motion is supplied as a velocity constraint. A position constraint is optionally provided to control initial conditions and to provide numerical stability. In that case the acceleration constraint is used only for initialization of constraint multipliers.

The routines available for dealing with prescribed motion are described below.

```
SUBROUTINE SDPRESACC(JOINT,AXIS,ACC)
INTEGER JOINT,AXIS
DOUBLE PRECISION ACC

SUBROUTINE SDPRESVEL(JOINT,AXIS,VEL)
INTEGER JOINT,AXIS
DOUBLE PRECISION VEL

SUBROUTINE SDPRESPOS(JOINT,AXIS,POS)
INTEGER JOINT,AXIS
DOUBLE PRECISION POS
```

```
SUBROUTINE SDPRES(JOINT,AXIS,PRES)
INTEGER JOINT,AXIS,PRES

SUBROUTINE SDGETPRES(JOINT,AXIS,PRES)
INTEGER JOINT,AXIS,PRES
```

`SDPRESACC()` is used to set the desired acceleration for a particular joint axis. Prescribed acceleration may be a function of time and any state variables. For a ball joint, three calls to `SDPRESACC()` are used to provide the prescribed angular acceleration vector. The prescribed acceleration is set to zero by `SDSTATE()`, so `SDPRESACC()` must be called between `SDSTATE()` and `SDDERIV()` to have any effect. (The Simplified Analysis Routines issue a call to the user-written `sdumotion()` routine in the appropriate place.) `SDDERIV()` uses this value to compute accelerations and constraint multipliers. Note that if `SDPRESACC()` is not called for a prescribed joint axis, the effect is to prescribe the acceleration of that axis to zero.

`SDPRESVEL()` and `SDPRESPOS()` are used to provide required values for the current velocities and positions, respectively. During a motion analysis (integration) these routines will have no effect unless stabilization has been enabled with `SDSTAB()`. Prescribed velocity may be a function of time and any state position variables. Prescribed position may be a function of time only. `SDPRESVEL()` is optional unless you are using the `SDRESID()` routine (generally because you are using a DAE integrator). For a ball joint, three calls to `SDPRESVEL()` are used to provide the prescribed angular velocity vector. `SDPRESPOS()` is always optional, and is not allowed for use with ball joints. However, if supplied these routines will affect Initial Velocity and Assembly analyses, respectively. Also, they can add numerical stability during integration if stabilization feedback constants have been set non-zero (see Section R4.2). Generally, it is best to provide the prescribed velocities and positions if you have them available, but don't turn on stabilization unless monitoring of constraint errors shows that it is necessary. If you do provide these, you must compute the velocity from a function which is the integral of the prescribed acceleration, and the position must come from a function which is the integral of the velocity. You cannot set them to arbitrary values as you can with the acceleration provided to `SDPRESACC()`. `SDSTATE()` sets the prescribed velocity and position to the current state values, so these routines must be called after `SDSTATE()` and before `SDDERIV()` or `SDRESID()` in order to have an effect. Note that if these routines are not called for a prescribed joint axis, the effect is to leave the velocity and position for that axis unconstrained.

If you are going to attempt to provide loop-joint prescribed position, using `SDPRESPOS()` can be tricky since rotational loop q's are always measured from $-\pi$ to π regardless of the number of turns. If your desired position is outside that range, you should integrate loop $\dot{q}$'s (from `SDPSQDOT()`) yourself and report them via an `SDPSSTATE()` call following your `SDSTATE()` call. (See Section R20.3 for details.) Normally, stabilizing just by specifying the velocity will be sufficient so we recommend that you not use `SDPRESPOS()` for rotational loop joint axes unless you are having a measurable stabilization problem. (This can be checked by reducing your desired q into the $-\pi$ to π range and seeing if it is acceptably close to the loop q returned by `SDPSEUDO()`.) A simpler alternative is to choose different "cuts" for your loops so that you can prescribe motion at a tree joint rather than a loop joint. This can always be done (at some cost in execution time) by splitting a loop inside a body rather than at a joint (see Section R16.1).

For the DAE method, using `SDRESID()` instead of `SDDERIV()`, it is the velocities that are prescribed rather than the accelerations. This is done using `SDPRESVEL()`, giving the desired velocity. This value is then used by `SDRESID()`. The function used to generate values for `SDPRESVEL()` must not have any discontinuities, although it does not have to be smooth.

`SDPRES()` is used to turn prescribed motion on or off at a particular joint axis, assuming that axis was designated as having “runtime” prescribed motion (i.e., a question mark appeared in the appropriate place in the System Description File). The `PRES` parameter is an integer, with `0` indicating that the axis motion is free and `1` indicating that it is prescribed.

`SDGETPRES()` returns `PRES` as an integer `0` or `1` to allow a test (most commonly in `sdumotion()` and `sduforce()`) to see whether axis motion is currently free or prescribed.

C Language

```
sdpresacc(joint,axis,acc)
int joint,axis;
double acc[3];

sdpresvel(joint,axis,vel)
int joint,axis;
double vel[3];

sdprespos(joint,axis,pos)
int joint,axis;
double pos[3];

sdpres(joint,axis,pres)
int joint,axis,pres;

sdgetpres(joint,axis,pres)
int joint,axis,*pres;
```

R15 Program Structure

SD/FAST generates subroutines for use in constructing various analyses of the input system. To use these routines, the user needs to write a simulation program in Fortran, C, or a simulation language like ACSL, Easy 5, Simulink, or Matrix-X which can call Fortran or C routines.

Normally, only a small fraction of the generated routines need be used for any particular analysis. There are several routines, called Simplified Analysis Routines, which contain commonly used "canned" analyses. When these are used, direct calls to most of the generated routines are unnecessary.

The following summarizes the recommended program structure, both for Simplified Analysis Routines and the more basic tools.

R15.1 Usage with Simplified Analysis Routines

The Simplified Analysis Routines are present in the Simplified Analysis File (suffix like "`_sar.f`" or "`_s.f`") generated by SD/FAST. This file need not be compiled and linked with your analysis program if you are not going to use these routines. However, even if you don't use these routines you may find the contents of the "`_sar`" file helpful as examples of how to set up an analysis using an integrator or root finder.

There is always one subroutine which must be written by the user when using the Simplified Analysis Routines. That routine, called `sduforce` (or more generally, <*prefix*>`uforce` where <*prefix*> is the routine prefix specified when SD/FAST was run), applies forces to the model as a function of time and state. Three routines are provided for applying a force at a point, a torque on a body, and a hinge torque or force (see Section R2.2). Even if there are no forces applied (or if the only force is gravity, which is provided automatically if requested) an empty `sduforce()` routine must be provided if any of the Simplified Analysis Routines are to be used.

If there is any prescribed motion defined for the system (even if it is currently disabled) an additional routine (called `sdumotion`) must be provided to apply the prescribed motions as a function of time and state.

Following are the basic steps for organizing an analysis program built around the Simplified Analysis Routines.

1.	write `sdumotion` and `sduforce` routines	see below
2.	set variable system parameters (?'s)	`SDMASS`,`SDPIN`,...
3.	initialize system	`SDINIT`
4.	close loops by adjusting q's	`SDASSEMBLE`
5.	compute compatible velocities (u's)	`SDINITVEL`
6.	perform static or steady analyses if needed	`SDSTATIC`, `SDSTEADY`

7. perform motion (dynamic) analysis if needed:

for each communication interval

- integrate equations across interval — `SDMOTION,SDFMOTION`
- compute system momentum, energy, mass — `SDMOM, SDSYS`
- compute reaction forces — `SDREAC`
- output quantities of interest

8. check for usage errors — `SDPRINTERR`

Example

The following example shows an analysis of the crank mechanism whose System Description File is shown in Section R19.3 on page R-112. The Simplified Analysis Routines are used to find compatible initial conditions for the mechanism and to perform a dynamic (motion) analysis.

```
      program crank
      integer NQ,NU,NEQ
      parameter (NQ=3, NU=3, NEQ=NQ+NU)
      integer lock(NU),fcnt,err,i,flag
      double precision state(NEQ),dstate(NEQ),t,dt,ctol,tol
      data lock/1,0,0/, dt,ctol,tol/.01d0,1d-5,1d-6/
c Set init conds   q1  q2  q3  u1  u2   u3  to a starting guess.
      data state/0d0,0d0,0d0,1d0,1d0,-1d0/, t/0d0/
      call SDINIT

c Perform assembly and velocity analysis. q1 and u1
c are locked, while the others are varied until compatible.
      call SDASSEMBLE(t,state,lock,1d-7,1000,fcnt,err)
      print 10,'sdassemble',err,fcnt
      call SDINITVEL(t,state,lock,1d-7,1000,fcnt,err)
      print 10,'sdinitvel',err,fcnt
10    format(' ', a10,' returned err=',i5,' fcnt=',i5)
      print 20,'qic=',(state(i),i=1,NQ)
      print 20,'uic=',(state(nq+i),i=1,NU)
20    format(' ', a5, 3f12.5)

c Do a motion analysis, printing out q1 and u1 at 100
c timesteps, each 'dt' apart.  Constraints must be maintained to
c 'ctol'.  The integration will be performed to a tolerance 'tol'.
      flag = 1
      do 40 i=1,100
          call SDMOTION(t,state,dstate,dt,ctol,tol,flag,err)
          print 30,t,state(1),state(nq+1),err
30        format(' ',3f12.5,i5)
40    continue

c Check for usage errors.
      call SDPRINTERR(6)
      end

c Apply the damping torque to the crank joint.
      subroutine sduforce(t,q,u)
      double precision t,q(*),u(*),damping
      data damping/3d0/
      call SDHINGET(1,1,-damping*u(1))
      return
      end
```

R15.2 Usage with General Analysis Routines

For analyses which do not use the Simplified Analysis Routines, we still recommend organizing your program with `sduforce()` and `sdumotion()` routines even though they are not required. In addition you may need to write several other routines which are described in this section.

To use an integrator, you will need to write a subroutine for the integrator to call when it needs state derivatives calculated. We will call this subroutine `sduderiv()`, although you may use any convenient name.

To use the root finder, you will need to write a routine for it to call when it needs to evaluate its current solution estimate. We refer to this routine as `sdueval()`, although you may use any convenient name.

The routines `sduaerr()`, `sduverr()`, `sduperr()`, and `sduconsfrc()` are required if there are user constraints in the system. Most users will not need user constraints and hence will not have to write these routines. If present, these routines must have the indicated names. See Section R24 for further discussion of user constraints.

Following are the basic steps for organizing an analysis program using more of the capabilities of the generated routines.

1. write `sdumotion` and `sduforce` routines	see below
2. if integrating, write `sduderiv` routine	see below
3. if using root finder, write `sdueval` routine	see below
4. write user constraint routines, if needed	see Section R24
for each system variant	
5. set variable system parameters	`SDMASS`, `SDPIN`,...
for each analysis of this variant	
6. initialize system	`SDINIT`
7. compute compatible initial *q*'s	`SDASSEMBLE` or `SDROOT`
8. compute compatible initial *u*'s	`SDINITVEL` or `SDROOT`
9. find roots of nonlinear equations, if desired	`SDROOT`
10. perform dynamic (motion) analysis	
• initialize derivatives	`sduderiv`
for each communication interval	
• integrate equations across interval	`SDVINTEG`, `SDFINTEG`
• compute system momen., energy, mass	`SDMOM`, `SDSYS`
• compute reaction forces	`SDREAC`
• output quantities of interest	
next communication interval	
next analysis	
next system variant	
11. check for usage errors	`SDPRINTERR`

For simple examples of the usage of the `SDROOT()`, `SDFINTEG()`, and `SDVINTEG()` routines, examine the implementations of `SDASSEMBLE()`, `SDINITVEL()`, `SDMOTION()`, etc., in a generated Simplified Analysis File.

R15.3 Structure of User Written Routines

Below are outlines of how the user written routines are constructed, as well as some examples.

R15.3.1 The `sduforce()` Routine

The user-written `sduforce()` routine should be declared like this:

```
subroutine sduforce(t,q,u)
parameter (NQ=..., NU=...)
double precision t,q(NQ),u(NU)
```

The basic structure of the routine is as follows:

1.	obtain any needed kinematic information	`SDPOS`, `SDVEL`,...
2.	apply point forces and body torques	`SDPOINTF`, `SDBODYT`
3.	apply hinge torques	`SDHINGET`

Example

This example shows an `sduforce()` routine implementing a spring between two points and a damper on a joint axis.

The spring is stretched between point `(1,2,3)` on body 1 (`ARM`) and point `(1,1,0)` on body 2 (`CHASSIS`). The spring constant is `K` = 10, and the natural length of the spring is `L` = 5. The damper is acting on the first axis of joint 3. The damping constant is `C` = 2. Note that this system must not be run in a configuration in which the two spring connection points are coincident, since it is impossible to determine the force direction in that case.

```
      subroutine sduforce(t,q,u)
      double precision t,q(*),u(*)
      double precision K,L,C,dist,stretch,
     1  armpt(3),chpt(3),armptg(3),chptg(3),frcg(3),frc(3)
      integer GROUND, ARM, CHASSIS, SDINDX
      parameter (K=10., L=5., C=2., GROUND=0, ARM=1, CHASSIS=2)
      data armpt/1d0,2d0,3d0/, chpt/1d0,1d0,0d0/


c obtain the ground frame location of the points
      call SDPOS(ARM,armpt,armptg)
      call SDPOS(CHASSIS,chpt,chptg)


c compute the force vector in gnd frame, in arm->chassis direction
      do 10 i=1,3
 10     frcg(i) = (chptg(i)-armptg(i))
      dist = sqrt(frcg(1)**2+frcg(2)**2+frcg(3)**2)
      stretch = dist - L
      do 15 i=1,3
 15     frcg(i) = K*stretch*frcg(i)/dist
```

```
c convert force into body frames and apply to the bodies
c    body1: force is positive
       call SDTRANS(GROUND,frcg,ARM,frc)
       call SDPOINTF(ARM,armpt,frc)
c    body2: force is negative
       call SDTRANS(GROUND,frcg,CHASSIS,frc)
       do 20 i=1,3
 20      frc(i) = -frc(i)
       call SDPOINTF(CHASSIS,chpt,frc)

c now apply the damping to joint 3, axis 1
       call SDHINGET(3,1, -C*u(SDINDX(3,1)))
       return
       end
```

C Language

```
sduforce(t,q,u)
double t,q[NQ],u[NU];
```

R15.3.2 The `sdumotion`() Routine

The user-written `sdumotion()` routine should be declared like this:

```
subroutine sdumotion(t,q,u)
parameter (NQ=..., NU=...)
double precision t,q(NQ),u(NU)
```

The basic structure of the routine is as follows:

1.	obtain any needed kinematic information	`SDPOS`, `SDVEL`,...
2.	enable/disable prescribed motion	`SDPRES`
3.	provide prescribed pos, vel, accel as desired	`SDPRESPOS`, `SDPRESVEL`,...

In this example, we prescribe the motion of axis 1 of joint 2 to have an initial position of 0.1 radians and a constant velocity of 10 radians/sec.

```
subroutine sdumotion(t,q,u)
double precision t,q(*),u(*),INITPOS,VEL
parameter (INITPOS=.1D0, VEL=10d0)

call SDPRESPOS(2,1, INITPOS+VEL*t)
call SDPRESVEL(2,1, VEL)
call SDPRESACC(2,1, 0d0)

return
end
```

In this example, we prescribe the acceleration of axis 1 of joint 2 to follow a sinusoidal function of time. The desired period is 0.1s, and the amplitude is 10.

```
  subroutine sdumotion(t,q,u)
  double precision t,q(*),u(*),PERIOD,AMP,twopi,radps
  parameter (PERIOD=.1d0, AMP=10d0)

  twopi = 2d0*acos(-1d0)
  radps = twopi/PERIOD
  call SDPRESACC(2,1, AMP*sin(rasps*t))

  return
  end
```

The above example does not specify the position or velocity of the prescribed joint, so these will simply be the values obtained as the acceleration is integrated, which may be subject to numerical drift. These can be stabilized if desired by specifying the velocity and position as functions of time as well. Knowledge of the initial position and velocity is required, so we assume here that they are passed in common.

```
  subroutine sdumotion(t,q,u)
  double precision
  t,q(*),u(*),PERIOD,AMP,twopi,radps,initpos,initvel
  parameter (PERIOD=.1d0, AMP=10d0)
  common /initcond/ initpos,initvel

  twopi = 2d0*acos(-1d0)
  radps = twopi/PERIOD
  call SDPRESACC(2,1, AMP*sin(radps*t))
  call SDPRESVEL(2,1, initvel - AMP*cos(radps*t)/radps)
  call SDPRESPOS(2,1, initpos+initvel*t -
1                     AMP*sin(radps*t)/(radps**2))

  return
  end
```

C Language

```
  sdumotion(t,q,u)
  double t,q[NQ],u[NU];
```

R15.3.3 The `sduderiv`() Routine

The user-written `sduderiv()` routine should be declared like this:

```
  subroutine sduderiv(t,state,dstate,param,status)
  parameter (NEQ=...)
  double t,state(NEQ),dstate(NEQ),param(*)
  integer status
```

The basic structure of the routine is as follows:

1.	compute kinematic information	`SDSTATE`
2.	apply forces	`sduforce`
3.	specify prescribed motions	`sdumotion`
4.	compute system derivatives	`SDDERIV`
5.	compute other needed derivatives	
6.	check for constraint errors	`SDPERR`, `SDVERR`, `SDAERR`
7.	check for other errors or events	

Here is the basic routine, for a system with 11 *q*'s and 10 *u*'s:

```
subroutine sduderiv(t,state,dstate,param,status)
integer NQ,NU,NEQ
parameter(NQ=11,NU=10,NEQ=NQ+NU)
double precision t,state(NEQ),dstate(NEQ),param(*)
integer status

call SDSTATE(t,state,state(NQ+1))
call sduforce(t,state,state(NQ+1))
call sdumotion(t,state,state(NQ+1))
call SDDERIV(dstate,dstate(NQ+1))
status = 0

return
end
```

If you have constraints, you may wish to monitor constraint errors here and report a non-zero status if there is a problem. Here we'll assume that there are 7 constraints, and the desired constraint tolerance is passed in `param(1)`.

```
subroutine sduderiv(t,state,dstate,param,status)
integer NQ,NU,NEQ,NC
parameter(NQ=11,NU=10,NEQ=NQ+NU,NC=7)
double precision t,state(NEQ),dstate(NEQ),param(*),errs(NC)
integer status,i

call SDSTATE(t,state,state(NQ+1))
call sduforce(t,state,state(NQ+1))
call sdumotion(t,state,state(NQ+1))
call SDDERIV(dstate,dstate(NQ+1))
```

```
      status = 1
      call SDAERR(errs)
      do 10 i=1,NC
10       if (abs(errs(i)) .gt. param(1)) return
      call SDVERR(errs)
      do 20 i=1,NC
20       if (abs(errs(i)) .gt. param(1)) return
      call SDPERR(errs)
      do 30 i=1,NC
30       if (abs(errs(i)) .gt. param(1)) return
      status = 0

      return
      end
```

C Language

```
sduderiv(t,state,dstate,param,status)
double t,state[NEQ],dstate[NEQ],param[];
int *status;
```

R15.3.4 The `sdueval`() Routine

The user-written `sdueval()` routine for use with the root finder `SDROOT()` should be declared like this:

```
subroutine sdueval(vars,param,resid)
parameter (NVAR=..., NFUNC=...)
double precision vars(NVAR),param(*),resid(NFUNC)
```

`NVAR` is the number of variables and `NFUNC` the number of (possibly nonlinear) functions to be simultaneously driven to zero by adjustment of the variables. The basic structure of the routine is as follows:

1. evaluate each function at the current value of `vars`
2. report these values back in the array `resid`

A warning for Fortran users: Fortran does not normally allow subroutines to call themselves recursively. If your `sdueval()` routine itself calls `SDROOT()`, either directly or indirectly, then you cannot use `SDROOT()` with that routine. Instead, use the "prefix" option of SD/FAST to generate another copy of `SDROOT()` but with a different name. For example, if you specify a prefix of `SD2`, SD/FAST will generate a routine by the name of `SD2ROOT()` which can be used in conjunction with your `sdueval()` routine which calls `SDROOT()`. Routines generated by SD/FAST which call `SDROOT()` are: `SDASSEMBLE()`, `SDINITVEL()`, `SDSTATIC()`, and `SDSTEADY()`. You will need a second version of `SDROOT()` if your `sdueval()` routine calls any of these. An example of this is given in Tutorial 4.

Example

A pendulum is supported by a pin joint which has a torsional spring. With the pin coordinate at 0, the pendulum is straight down. The spring is attached at 90^{o} (that is, it would exert no torque if the pendulum were sticking straight out to the right). Consequently, it exerts a positive torque when the pendulum hangs straight down. Gravity acts downward. The problem to solve is: what spring stiffness `k` is required to make the equilibrium position for the pendulum 40^{o}?

This problem has a single variable, the spring stiffness. The pin joint acceleration will serve as a function which measures how close to equilibrium the system is — if we place the pendulum at 40^{o} with zero velocity and see no acceleration, then the system must be in equilibrium. The `sdueval()` routine for this problem is shown below.

```
      subroutine sdueval(vars,param,resid)

      integer NVAR,NFUNC,NQ,NU
      parameter (NVAR=1, NFUNC=1, NQ=1, NU=1)
      double precision vars(NVAR),param(*),resid(NFUNC),
     1    q(NQ),u(NU),qdot(NQ),udot(NU),pi,dtr,k,stretch
      pi = acos(-1d0)
      dtr = pi/180d0


c put the pendulum at +40 degrees
      q(1) = 40d0 * dtr
      u(1) = 0d0


c the spring is stretched down from 90 degrees
      stretch = -(90d0*dtr - q(1))
      k = vars(1)


c compute and return the resulting joint acceleration
      call SDSTATE(0d0,q,u)
      call SDHINGET(1,1, k*stretch)
      call SDDERIV(qdot,udot)
      resid(1) = udot(1)

      return
      end
```

A complete program for solving this problem using the above `sdueval()` routine with the root finder is shown next.

```
      program pendulum

      integer NQ,NU,NVAR,NFUNC,NFNV,IWSZ,JWSZ,DWSZ,RWSZ
      parameter (NQ=1,NU=1,NVAR=1,NFUNC=1,NFNV=NFUNC+NVAR,
     1  IWSZ=4*NFNV,JWSZ=NFUNC*NVAR,DWSZ=2*NFNV*NFNV,RWSZ=9*NFNV)


      double precision vars(NVAR),param(1),fret(NFUNC),
     1                 jw(JWSZ),dw(DWSZ),rw(RWSZ)
      integer lock(NVAR),iw(IWSZ),fcnt,err
      external sdueval


      call SDINIT


c set initial guess for k, and leave k free
      vars(1) = 10d0
      lock(1) = 0


c solve for the desired k
      call SDROOT(sdueval,vars,param,NFUNC,NVAR,0,lock,1d-6,0d0,
     1            100,jw,dw,rw,iw,fret,fcnt,err)
      print *,'SDROOT returned err=',err,' after ',fcnt,' evals'

      print *,'k=',vars(1),' N-m/rad with residual accel ',fret(1)

      stop
      end
```

C Language

```
sdueval(vars,param,resid)
double vars[NVAR],param[],resid[NFUNC];
```

R16 Reference Configuration

SD/FAST accepts the description of a multibody system and develops equations of motion for the system. To do this, the system description must convey information regarding mass properties of the bodies, location of the hinge points, orientation of the bodies, and orientation of joint axes. The configuration of the system is parameterized in terms of scalars called generalized coordinates. Knowledge of the system coordinates completely specifies the location of each body and its orientation in the inertial frame. This specification is done by measuring the system configuration with respect to a particular configuration called the *reference configuration*. The multibody system is described to SD/FAST in its reference configuration, in which all generalized coordinates are defined to be zero.

R16.1 Choose Spanning Tree

The multibody system can have tree structure or can have one or more closed loops. In the case of loops, the user must cut each loop to form a tree-structured system (which includes, or *spans*, all the bodies in the system) and a set of loop joints, one per loop. This can be done in one of two ways: a loop can be cut at a joint or at a point fixed in a body. When a loop is cut at a joint the number of joints in the resulting system is reduced by one. The removed joint becomes a loop joint in the system. When a body is cut, the number of bodies in the system is increased. The original system is recovered by "welding" the two bodies back together. SD/FAST supports a "weld" joint useful for this purpose.

Splitting a body allows some arbitrariness in the choice of point at which to cut, and the resulting mass properties of the sub-parts. The only rule is that the two bodies should have a total mass equal to the original mass, and a composite inertia equal to the original part inertia. An easy way to accomplish this is to choose the mass center of the part as the point at which to split. The two parts can then each be assigned half the original part mass, and half the original part inertia, provided the mass center of each sub-part is defined to be at the original part mass center. Another way to think of this is to imagine two copies of the original part, each made of material of half the density of the original. The original part is built by superimposing the two bodies (allowing them to occupy the same space!), and then welding some point of the bodies. The two joints which form the loop in question are each assigned to a different member of the pair. Other points fixed in the original body can be arbitrarily assigned to either member of the pair.

The choice of spanning tree can have an influence on the execution speed of the resulting generated equations of motion. It is best to cut a loop so as to minimize the maximum length of any chain of bodies. That is, it is better to cut a loop in the middle than at either end. This reduces the amount of coupling among the bodies and simplifies the equations. The choice of whether to cut at a joint or in a body also has performance implications. The equations are always simpler if a joint is cut since that reduces the number of degrees of freedom in the tree system. And if you have a choice of different types of joints at which to cut, always pick the one with the greater number of degrees of freedom. Cutting in a body leaves all the joints in the tree system, and adds six constraint equations to effect the weld joint. Still, when the performance difference[14] is not a major issue, cutting a body means that all the joints can be treated uniformly, avoiding the need to deal with loop joint pseudo coordinates. The choice is yours. See Section R11.1 on page R-49 for more discussion.

R16.2 Select Zero Configuration

Once a tree has been chosen, the multibody system is deposited in inertial space (that is, relative to a ground frame) in any arbitrary configuration. There are some considerations to follow, which will be discussed below, but essentially there are no restrictions on the user's choice. The reference configuration does not restrict the motion of the system, or imply that eventual motion studies will begin at the reference, or even operate near the reference. After depositing the system in inertial space, the user should choose some point fixed in the inertial (ground) frame as a reference point (origin), and fix a coordinate basis in the inertial frame. An image of the inertial coordinate basis is then fixed in each body, so that in the reference configuration, all the body-fixed frames have the same orientation as the inertial frame. As the bodies move, and the joints rotate, the body frames do not remain parallel to each other, but the coordinates of points fixed in any body and the body central inertia matrix are constant when expressed in the local body frame.

All vectors contained in the SD/FAST System Description File and all the body inertia matrices are measured in the reference configuration. Any connection points between a body and ground are specified by giving the vector from the chosen reference point (the ground "origin") to the connection point.

When SD/FAST generates equations of motion for the multibody system the choice of reference configuration has some influence over the efficiency of the resulting equations. This is because the system geometry and mass properties enter into the equations of motion as parameters. The system geometry is the specification for each axis vector, and the vectors from the body mass centers to the hinge points. The mass properties include the mass of each body and the central inertia matrix of each body. Choices for the reference configuration which allow the SD/FAST symbol manipulator to simplify the equations are those which reduce the number of non-zero parameters. For example, orienting bodies so that the inertially-fixed coordinate directions are parallel to the body central principal directions causes the inertia matrix to have only three inertia scalars rather than the usual six scalars. Likewise, positioning bodies to align joint axes with coordinate directions, or to cause the `bodytojoint` or `inbtojoint` vectors to lie along coordinate directions will simplify the equations. It is usually true as well that the simplest reference configuration is also the easiest for the user to specify. When performing an analysis, the user can specify initial conditions to bring the multibody system into any allowable configuration.

---

14. The additional cost of the body-splitting approach depends strongly on the available alternatives. For example, if you would otherwise have to cut a loop pin joint you would remove only one equation from the tree system and add five constraint equations. This is not much better than adding the weld, which would leave the tree system as is and add six constraints. On the other hand, a cut ball joint removes three equations and adds only three constraints, and a cut sixdof joint removes six equations and adds none!

As mentioned above, the multibody system may possess loops of bodies, which must be broken to define the reference. There is no requirement when choosing the reference configuration that loop joints be in an assembled state. The user is free to not only break the loop, but to then move the ends of the loops away from each other. The user must simply designate loop closure points (the inboard and outboard hinge points) and the type of loop joint desired, but the closure points do not need to be coincident in the reference configuration. Likewise, orientation constraints associated with loop joints do not have to be satisfied in the reference configuration. After creating the equations of motion, SD/FAST provides an analysis routine (`SDASSEMBLE()`) whose purpose is to assemble the system. The system can be defined in an unassembled state, which makes the system specification much easier to perform.

R17 Set and Get System Parameters

This section documents the set of routines used for setting the system parameters which were specified with question marks in the System Description File; for retrieving the current value of any system parameter; and for retrieving invariant topological information about the system.

R17.1 Set Parameters

In the SD/FAST System Description File, mass properties and system geometry can be specified as variable, by providing their values with a "`?`", either alone or with a default value. Any parameters which have been specified this way may be modified by the routines described below. Parameters whose values were provided as constants may not be modified and any attempt to do so will cause an error to be posted for `SDPRINTERR()`. After any of these parameter-modifying routines has been called, `SDINIT()` must be called before any subsequent analysis can be performed. (Multiple calls to these routines may be made before the `SDINIT()` call.)

Every System Description File parameter which was specified with a lone question mark (no default) *must* have a value provided with one of these routines before the call to `SDINIT()`, or `SDINIT()` will post an error.

```
SUBROUTINE SDMASS(BODY,MASS)
INTEGER BODY
DOUBLE PRECISION MASS

SUBROUTINE SDINER(BODY,INERTIA)
INTEGER BODY
DOUBLE PRECISION INERTIA(3,3)

SUBROUTINE SDBTJ(JOINT,BODYTOJOINT)
INTEGER JOINT
DOUBLE PRECISION BODYTOJOINT(3)

SUBROUTINE SDITJ(JOINT,INBTOJOINT)
INTEGER JOINT
DOUBLE PRECISION INBTOJOINT(3)

SUBROUTINE SDPIN(JOINT,PINNO,PIN)
INTEGER JOINT,PINNO
DOUBLE PRECISION PIN(3)
```

In each of the above routines, the supplied parameter is of the appropriate dimension for the quantity being set. Note that entire vectors and the whole inertia matrix are set with a single call. Any elements of the vector or matrix which correspond to parameters which were not specified with question marks are simply ignored and need not be set prior to the call. If *none* of the elements was specified with a question mark, an error will be posted.

`SDINER()` looks only at the upper right triangle of `INERTIA` and sets the off-diagonal lower triangle elements equal to the corresponding upper triangle elements.

For loop joints, `SDPIN()` treats the inboard pins as pin numbers 1, 2, and 3, `inbref` as pin no. 4, `bodypin` as pin no. 5, and `bodyref` as pin no. 6. Attempts to set values for nonexistent pins or reference lines are ignored and an error indication is posted.

C language

```
sdmass(body,mass)
int body;
double mass;

sdiner(body,inertia)
int body;
double inertia[3][3];

sdbtj(joint,bodytojoint)
int joint;
double bodytojoint[3];

sditj(joint,inbtojoint)
int joint;
double inbtojoint[3];

sdpin(joint,pinno,pin)
int joint,pinno;
double pin[3];
```

As always, the body and joint numbers begin at 0 in C (it doesn't make sense to supply ground, body -1, here). In addition, the pin number parameter to `sdpin()` varies from 0 to 5 in C, not 1 to 6 as described above.

R17.2 Get Parameters

The routines in this section are used to access the current values of system parameters. If these were specified with "?" in the input file, the returned values will be the ones most recently supplied via the corresponding routines described above.

```
SUBROUTINE SDGETMASS(BODY,MASS)
INTEGER BODY
DOUBLE PRECISION MASS

SUBROUTINE SDGETINER(BODY,INERTIA)
INTEGER BODY
DOUBLE PRECISION INERTIA(3,3)

SUBROUTINE SDGETBTJ(JOINT,BODYTOJOINT)
INTEGER JOINT
DOUBLE PRECISION BODYTOJOINT(3)
```

```
SUBROUTINE SDGETITJ(JOINT,INBTOJOINT)
INTEGER JOINT
DOUBLE PRECISION INBTOJOINT(3)

SUBROUTINE SDGETPIN(JOINT,PINNO,PIN)
INTEGER JOINT,PINNO
DOUBLE PRECISION PIN(3)
```

For each routine, the returned argument is a floating point value of the appropriate type and dimension.

For loop joints, `SDGETPIN()` returns the inboard pins as pin numbers 1, 2, and 3, inbref as pin no. 4, bodypin as pin no. 5, and bodyref as pin no. 6. Attempts to get values for nonexistent pins or reference lines are ignored and an error indication is posted.

C language

```
sdgetmass(body,mass)
int body;
double *mass;

sdgetiner(body,inertia)
int body;
double inertia[3][3];

sdgetbtj(joint,bodytojoint)
int joint;
double bodytojoint[3];

sdgetitj(joint,inbtojoint)
int joint;
double inbtojoint[3];

sdgetpin(joint,pinno,pin)
int joint,pinno;
double pin[3];
```

As always, the body and joint numbers begin at 0 in C (it doesn't make sense to supply ground, body -1, here). In addition, the pin number parameter to `sdgetpin()` varies from 0 to 5 in C, not 1 to 6 as described above.

R17.3 Get System Topology

The routines in this section return topological (integer) information about the system as a whole, about the joints and about the constraints. These routines can be used to write generic simulation codes usable with any SD/FAST-generated simulation. Note that most of this information is available in the generated Information File as well.

```
SUBROUTINE SDINFO(INFO)
INTEGER INFO(50)

SUBROUTINE SDJNT(JOINT,INFO,SLIDER)
INTEGER JOINT,INFO(50),SLIDER(6)

SUBROUTINE SDCONS(CONSNO,INFO)
INTEGER CONSNO,INFO(50)
```

The meaning of each of the returned values in the different `INFO` arrays for the three routines, and the `SLIDER` array for `SDJNT()` are shown below.

SDINFO()

`INFO(1)`	`GROUNDED`	1 if system is grounded, 0 if free-flying
`(2)`	`NBOD`	number of bodies (and tree joints) in the system
`(3)`	`NDOF`	number of tree hinge degrees of freedom (ign. pres)
`(4)`	`NC`	total number of constraints
`(5)`	`NLOOP`	number of loop joints in the system
`(6)`	`NLDOF`	number of loop hinge degrees of freedom (ignoring prescribed)
`(7)`	`NLC`	number of loop constraints (ign. pres)
`(8)`	`NB`	number of 6dof and ball joints in the tree system
`(9)`	`NLB`	number 6dof and ball joints which are loop joints
`(10)`	`NPRESC`	total number of prescribed motion constraints
`(11)`	`NUSERC`	number of user constraints
`(12)`	`RANK`	number constraints not obviously redundant (<=nc)
`(13..50)`		not currently used

Several quantities displayed in the Information File are not directly returned by `SDINFO()`. These can be computed from the above quantities as follows:

```
NQ=NDOF+NB, NU=NDOF, NLQ=NLDOF+NLB, NLU=NLDOF, NJNT=NBOD+NLOOP
```

SDJNT()

`INFO(1)`	`TYPE`	1=pin,2=ujoint,3=gimbal,4=ball,5=slider,6=free,7=cylindrical, 8=planar,9=weld,10=bushing,11=bearing, 20=rplanar,21=rfree,22=rbushing,23=rbearing
`(2)`	`ISLOOP`	0=tree joint, 1=loop joint
`(3)`	`INB`	body number of inboard (first) body
`(4)`	`OUTB`	body number of outboard (second) body
`(5)`	`NDOF`	no. dof provided by joint (ignoring prescribed)
`(6)`	`NC`	no. constraints used to implement jt (ignoring prescribed)
`(7)`	`NP`	no. prescribed or run-time prescribed hinges
`(8)`	`FIRSTQ`	coord or pseudo-coord index of 1st axis (in `q`,`lq`,`u`,`lu`)
`(9)`	`BALLQ`	for ball or 6dof, index of 4th Euler param (in `q` or `lq`)
`(10)`	`FIRSTM`	for any loop joint, `MULT` index of 1st multiplier
`(11)`	`FIRSTP`	if any prescribed motion, `MULT` index of 1st multiplier
`(12..50)`		not currently used
`SLIDER(1..6)`		0 if corresponding axis is rotational, 1 if slider, undefined for `SLIDER(i)`, `i` > `NDOF`

SDCONS()

INFO(1)	TYPE	1=user
(2)	MULT	mult index of multiplier for this constraint
(3..50)		not currently used

C language

```
sdinfo(info)
int info[50];

sdjnt (joint,info,slider)
int joint,info[50],slider[6];

sdcons(consno,info)
int consno,info[50];
```

Note also that all the indices shown above are numbered from 1. In C, these indices are numbered from 0 so you should subtract 1 from each index shown above. The values for `TYPE` returned by `SDJNT()` and `SDCONS()` are as shown, however.

R18 Simplified Analysis Routines

SD/FAST generates routines for performing several common analyses without having to use the more general (and more numerous) underlying routines and analysis tools. The analyses that can be performed this way are limited forms of assembly analysis, initial velocity analysis, static analysis, motion analysis, and steady motion analysis. The following restrictions apply:

1. Forces and prescribed motions (if any) must have been supplied via user-written `sduforce()` and `sdumotion()` subroutines. If your analysis program is organized differently, the Simplified Analysis Routines cannot be used, although all of the General Analysis Routines are available (see Section R8).
2. The only constraints that can be met by the Simplified Assembly and Initial Velocity Analyses are those that are also maintained during motion analysis, that is, loop joint constraints, prescribed motion constraints, and user constraints. In addition, individual state variables can be set to particular values and locked at those values. To impose other conditions to be met by these analyses, the `SDROOT()` nonlinear equation solving tool described in Section R8.2 (or any other such tool you may have at your disposal) should be used explicitly.
3. Simplified Motion Analysis assumes that the only state variables are the hinge positions and velocities, and further that the state array consists of the q's followed immediately by the u's. (For a more detailed discussion of the system state, see Section R20.) If you have additional state variables, such as those modeling control elements, you should use the supplied integrators `SDVINTEG()` or `SDFINTEG()` or any other integration method you may have available, such as those provided by ACSL, Simulink, Matrix-X, Easy-5, or IMSL.
4. Simplified Static and Steady Motion analyses operate by trying to find configurations (and velocities, in the case of Steady Motion) which yield zero accelerations. Depending on initial conditions, these routines may discover unstable solutions. You may be able to define better functions to minimize for your system, such as potential energy for static analysis. In that case you should use `SDROOT()` or other nonlinear equation solving methods rather than the simplified routines.

Routines in this section are placed in the Simplified Analysis File ("`_sar`" or "`_s`" file). It is not necessary to generate and compile these routines if you do not plan to use them. However, they are often quite useful even when putting together a complicated analysis, so they are generated by default.

It is important to stress that these routines are provided for convenience only. It is by no means necessary to use them; nor is it particularly difficult to use the more general underlying analysis tools. Furthermore, the SD/FAST routines are designed to be used with *any* analysis tools, including simulation languages, not just the tools provided with SD/FAST.

R18.1 Assembly Analysis

```
SUBROUTINE SDASSEMBLE(T,STATE,LOCK,TOL,MAXEVALS,FCNT,ERR)
DOUBLE PRECISION T,STATE(NQ+NU),TOL
INTEGER LOCK(NU), MAXEVALS,FCNT,ERR

ERR=0 => success
    1 => failed
    2 => ran out of time
```

`SDASSEMBLE()` attempts to find a set of hinge positions q (in the first `NQ` elements of `STATE`) which meet all the position constraints so that the largest error in any one constraint is at or below `TOL`. Position constraints are (1) loop joint constraints (assembly constraints), (2) prescribed motion constraints, and (3) user defined position constraints. Only the first `NQ` elements of `STATE` are examined and changed. Any elements of `STATE` after that are neither examined nor changed.

The analysis proceeds incrementally from the passed-in initial `STATE`, using a nonlinear root finding method which involves repeated calls to `SDSTATE()` and, if there is any prescribed motion in the problem, the user-written `sdumotion()` routine. Position constraint errors are evaluated using `SDPERR()` (which, if there are user constraints, will mean a call to the user-written `sduperr()` routine). The passed-in `MAXEVALS` parameter limits the number of calls to `SDSTATE()` which may be made by `SDASSEMBLE()` in searching for the answer. The returned `FCNT` says how many calls actually were made. This may be somewhat higher than `MAXEVALS`.

There is no guarantee that `SDASSEMBLE()` will find a solution even if there is one. If it fails, and you feel that there actually is a solution, you should provide `SDASSEMBLE()` with a better initial guess in the passed-in `STATE`.

If `LOCK(i)` is nonzero, the corresponding q_i (in `STATE(i)`) will not be changed by `SDASSEMBLE()`. For ball joints, although the q's are Euler parameters, the locks really correspond to body-fixed 1-2-3 Euler angles (see Section R6). For example, if the first joint is a ball then setting `LOCK(2)` will restrict the rotations to those which are reachable by a rotation around axis 1 followed by a rotation around axis 3. However, the initial and final rotations will be represented in `STATE` by four Euler parameters, all of which may change during the analysis. Only the three locks in locations corresponding to the first three Euler parameters are needed. Thus only `NU` locks are needed even though there are `NQ` q's in `STATE`.

If the motion of a particular joint is prescribed, it is a good idea to set the corresponding q to its correct prescribed value for time `T`, and then lock that q to remove it from consideration during the analysis. If you start a q a long distance from its correct prescribed value, it may slow convergence or even prevent a solution from being found.

If `ERR`=0 on return, `STATE` contains q's which bring all the position errors to `TOL` or below. Otherwise, it contains the best set of q's encountered during the analysis, that is, the configuration which produced the smallest maximum position error. If `ERR`=1, `SDASSEMBLE()` had either hit a local minimum or was improving so slowly that there is no point continuing. If `ERR`=2, `SDASSEMBLE()` was still making some progress (although it may have been very slow) when it exceeded `MAXEVALS` calls to

SDSTATE(). In that case, another call to SDASSEMBLE() may result in further improvement.

After calling SDASSEMBLE(), a call to SDPERR() can be used to check how well the initial positions have been met. SDASSEMBLE() guarantees that the last call it made to SDSTATE() was made at the final value of the hinge positions, so you do not need to call SDSTATE() prior to calling SDPERR() or other routines, such as SDPOS().

R18.2 Initial Velocity Analysis

```
SUBROUTINE SDINITVEL(T,STATE,LOCK,TOL,MAXEVALS,FCNT,ERR)
DOUBLE PRECISION T,STATE(NQ+NU),TOL
INTEGER LOCK(NU), MAXEVALS,FCNT,ERR

ERR=0 => success
    1 => failed
    2 => ran out of time
```

SDINITVEL() attempts to find a set of hinge velocities *u* (in the final NU elements of STATE) which meet all the velocity constraints so that the largest error in any one constraint is at or below TOL. Velocity constraints are (1) loop joint constraints (that is, the velocities must not cause the loop joints to unassemble), (2) prescribed motion velocity constraints, and (3) user defined velocity constraints.

Note: a successful SDASSEMBLE() analysis should have been performed before attempting an SDINITVEL() analysis.

The analysis proceeds incrementally from the passed-in initial velocities *u*. The initial positions *q* (in the first NQ elements of STATE) are not changed by the analysis. This is a linear problem so the solution (if there is one) will always be found. If there is no solution, a least squares solution will be found. Repeated calls are made to SDSTATE() and, if there is any prescribed motion in the problem, the user-written sdumotion() routine. Velocity constraint errors are evaluated using SDVERR() (which, if there are user constraints, will mean a call to the user-written sduverr() routine).

If LOCK(i) is nonzero, the corresponding u_i (in STATE(NQ+i)) will not be changed by SDINITVEL(). The passed-in MAXEVALS parameter limits the number of calls to SDSTATE() which may be made by SDINITVEL() in searching for the answer. The returned FCNT says how many calls actually were made. This may be somewhat higher than MAXEVALS.

If ERR=0 on return, STATE contains a set of hinge velocities *u* which bring all the velocity errors to or below TOL. Otherwise, it contains the best set of velocities encountered during the analysis, that is, the velocities which produced the smallest maximum velocity error. If ERR=1, there are no values for the unlocked *u*'s which, while maintaining the locked *u*'s, meet all the velocity constraints. If ERR=2, SDINITVEL() was still making progress when it exceeded MAXEVALS calls to SDSTATE(). In that case, another call to SDINITVEL() may result in further improvement.

After calling `SDINITVEL()`, a call to `SDVERR()` can be used to check how well the initial velocities have been met. `SDINITVEL()` guarantees that the last `SDSTATE()` call it made was at the final `STATE` value, so `SDVERR()`, `SDVEL()` and other routines that depend on `SDSTATE()` can be called immediately after return from `SDINITVEL()`.

R18.3 Static Analysis

```
SUBROUTINE SDSTATIC(T,STATE,LOCK,CTOL,TOL,MAXEVALS,FCNT,ERR)
DOUBLE PRECISION STATE(NQ+NU),CTOL,TOL
INTEGER LOCK(NU), MAXEVALS,FCNT,ERR

ERR=0 => success
    1 => failed
    2 => ran out of time
```

`SDSTATIC()` attempts to find a set of hinge positions q which constitute a static configuration for the system, that is, one in which all the accelerations are 0 when the hinge velocities u are as supplied in the last `NU` elements of `STATE`. `SDSTATIC()` maintains the position constraints to tolerance `CTOL`, and searches for a solution in which the maximum hinge acceleration ($\dot{u}$) is at or below `TOL`.

Note: a successful `SDASSEMBLE()` analysis should have been performed to tolerance at least `CTOL` before attempting an `SDSTATIC()` analysis. Velocity errors should also be small, although this is usually guaranteed by setting all the velocities to 0 before the `SDSTATIC()` call.

The analysis proceeds incrementally from the q's in the first `NQ` elements of the passed-in initial `STATE`, using a nonlinear root finding method which involves repeated calls to `SDSTATE()`, the user-written `sduforce()` and `sdumotion()` routines, and `SDDERIV()`. Position constraint errors are evaluated using `SDPERR()` (which, if there are user constraints, will mean a call to the user-written `sduperr()` routine). Accelerations are evaluated with `SDDERIV()` which, if there are user constraints, will call the user-written `sduaerr()` and `sduconsfrc()` routines.

Normally, all velocities are set to zero for static analysis. `SDSTATIC()` does allow more generality in that non-zero velocities can be passed in; however, no attempt is made to maintain velocity constraints. Only the hinge coordinates (q's) are varied to minimize the hinge accelerations — the hinge velocities (u's) are not changed. If you pass in non-zero velocities you must be certain that velocity errors will remain zero in any configuration. Normally, that means that all the velocities in parts of the system which form closed loops are zero.

There is no guarantee that `SDSTATIC()` will find a solution even if there is one. If it fails, and you feel that there actually is a solution, or if it finds an undesired solution (such as an unstable equilibrium), you should provide `SDSTATIC()` with a better initial guess in the passed-in q's. It is sometimes helpful to run a motion analysis (perhaps with added damping) for a while to let the system move "downhill" before beginning the static analysis.

The passed-in `MAXEVALS` parameter limits the number of calls to `SDSTATE()` which may be made by `SDSTATIC()` in searching for the answer. The returned `FCNT` parameter says how many calls actually were made. This may be somewhat larger than `MAXEVALS`.

If `LOCK(i)` is nonzero, the corresponding q_i (in `STATE(i)`) will not be changed by `SDSTATIC()`. For ball joints, the same considerations apply as for `SDASSEMBLE()` (see page R-99). If the problem contains prescribed motion, you can speed up the analysis somewhat by locking the corresponding q's.

If `ERR`=0 on return, `STATE` contains a set of q's which bring all the hinge accelerations to or below `TOL` with the velocities as supplied. Otherwise, `STATE` contains the best configuration encountered during the analysis, that is, the configuration which produced the smallest maximum hinge acceleration. If `ERR`=1, `SDSTATIC()` had either hit a local minimum or was improving so slowly that there is no point continuing. If `ERR`=2, `SDSTATIC()` was still making some progress (although it may have been very slow) when it exceeded `MAXEVALS` calls to `SDSTATE()`. In that case, another call to `SDSTATIC()` may result in further improvement.

After calling `SDSTATIC()`, a call to `SDDERIV()`, `SDACC()`, etc., can be used to check how well the static configuration has been obtained. `SDSTATIC()` guarantees that the last calls to `SDSTATE()` and `SDDERIV()` were made with the final state, so you do not need to call these routines again before calling routines such as `SDACC()` or `SDREAC()`. The position errors will always be at most `CTOL` if they were met to that tolerance when `SDSTATIC()` was called.

R18.4 Steady Motion Analysis

```
SUBROUTINE SDSTEADY(T,STATE,LOCK,CTOL,TOL,MAXEVALS,FCNT,ERR)
DOUBLE PRECISION STATE(NQ+NU),CTOL,TOL
INTEGER LOCK(NU+NU),MAXEVALS,FCNT,ERR

ERR=0 => success
    1 => failed
    2 => ran out of time
```

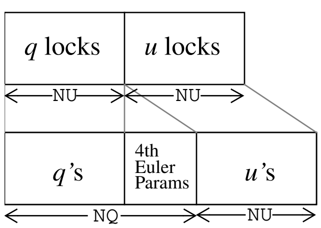
`SDSTEADY()` attempts to find a set of hinge positions q (in the first `NQ` elements of `STATE`) and hinge velocities u (the last `NU` elements of `STATE`) which constitutes a "steady" configuration for the system, that is, one in which all the hinge axis accelerations are 0. Unlike `SDSTATIC()`, `SDSTEADY()` maintains both the position constraints and velocity constraints to tolerance `CTOL`, and will modify both the hinge positions and velocities in search of a solution.

Note: a successful `SDASSEMBLE()` and `SDINITVEL()` analysis should have been performed to tolerance at least `CTOL` before attempting an `SDSTEADY()` analysis.

A successful return from `SDSTEADY()` does not necessarily mean that a solution that is steady over time has been found, only that the accelerations at the given time are 0. A simple example is a pendulum swinging around its pivot in a gravity field — its hinge acceleration will be 0 at the top and bottom of its stroke, but there is no steady solution since its velocity changes with time. A motion analysis (e.g., with `SDMOTION()`) should be performed after `SDSTEADY()` to verify that the solution is still steady after some time has elapsed. If not, there may be no steady solution, or you may have to start `SDSTEADY()` with a state that is closer to the desired solution.

The analysis proceeds incrementally from the passed-in initial state, using a nonlinear root finding method which involves repeated calls to `SDSTATE()`, the user-written `sduforce()` and `sdumotion()` routines, and `SDDERIV()`. Constraint errors are evaluated using `SDPERR()` and `SDVERR()` which, if there are user constraints, will mean calls to the user-written `sduperr()` and `sduverr()` routines. Accelerations are evaluated with `SDDERIV()` which, if there are user constraints, will call the user-written `sduaerr()` and `sduconsfrc()` routines.

There is no guarantee that `SDSTEADY()` will find a solution even if there is one. If it fails, and you feel that there actually is a solution, or if it finds an undesired solution, you should provide `SDSTEADY()` with a better initial guess in the passed-in `STATE`. The passed-in `MAXEVALS` parameter limits the number of calls to `SDSTATE()` which may be made by `SDSTEADY()` in searching for the answer. The returned `FCNT` parameter says how many calls actually were made. This may be somewhat larger than `MAXEVALS`.



If `LOCK(i)` is nonzero, the corresponding *q* or *u* in `STATE` will be left unchanged by `SDSTEADY()`. If there are no ball (or sixdof) joints in the problem `LOCK(i)` corresponds directly to `STATE(i)`. With ball joints, `LOCK` corresponds to a "compressed" state in which all the ball joint Euler parameters (4 per ball) have been converted to Euler angles (3 per ball). That is, while `STATE` is `NQ+NU` long, `LOCK` will be just 2*`NU` long, with the first `NU` elements corresponding to `STATE(1)` to `STATE(NU)` and the second `NU` elements corresponding to `STATE(NQ+1)` to `STATE(NQ+NU)`. Hinges with prescribed motion can be locked to improve analysis performance.

If `ERR`=0 on return, `STATE` contains *q*'s and *u*'s which brings all the hinge accelerations to or below `TOL`. Otherwise, `STATE` contains the best set of *q*'s and *u*'s encountered during the analysis, that is, the set which produced the smallest maximum hinge acceleration. If `ERR`=1, `SDSTEADY()` had either hit a local minimum or was improving so slowly that there is no point continuing. If `ERR`=2, `SDSTEADY()` was still making some progress (although it may have been very slow) when it exceeded `MAXEVALS` calls to `SDSTATE()`. In that case, another call to `SDSTEADY()` may result in further improvement.

After calling `SDSTEADY()`, a call to `SDDERIV()`, `SDACC()`, etc., can be used to check how well the steady configuration has been obtained. `SDSTEADY()` guarantees that the last call to `SDSTATE()` and `SDDERIV()` was made with the final state, so these routines do not need to be called again before calling routines such as `SDACC()` or `SDREAC()`. The position and velocity errors will always be at most `CTOL` if they were met to that tolerance when `SDSTEADY()` was called.

R18.5 Motion Analysis

```
SUBROUTINE SDMOTION(T,STATE,DSTATE,DT,CTOL,TOL,FLAG,ERR)
DOUBLE PRECISION T,STATE(NQ+NU),DSTATE(NQ+NU),DT,CTOL,TOL
INTEGER FLAG,ERR

ERR=0 => success
    1 => warning: went over a step
    2 => can't continue (lock-up)
    3 => can't continue (constraint violated)

SUBROUTINE SDFMOTION(T,STATE,DSTATE,DT,CTOL,FLAG,ERREST,ERR)
DOUBLE PRECISION T,STATE(NQ+NU),DSTATE(NQ+NU),DT,CTOL,ERREST
INTEGER FLAG,ERR

ERR=0 => success
    1 => warning: constraint violated
```

SDMOTION() integrates the state from time T to T+DT using a variable-timestep integrator. Local error is controlled not to exceed TOL, provided the system accelerations contain no impulses (sharp spikes whose duration is small compared to the integration interval). Global errors are not controlled, but in practice tend to be similar to the local error. If you are concerned about global error, the best approach is to rerun the integration at a tighter tolerance and retain only the digits that do not change between the two runs.

SDFMOTION() integrates from T to T+DT using a fixed step (fourth order Runge-Kutta, or RK4) integrator. An estimate of the integration error introduced in the step is returned in ERREST. ERREST is similar in meaning to SDMOTION()'s TOL parameter. That is, ERREST is returned as the TOL value which would just have been met at this step size.

Note: successful SDASSEMBLE() and SDINITVEL() analyses should have been performed before attempting an SDMOTION() or SDFMOTION() analysis.

These routines are normally called repeatedly in a loop to obtain a time history of system motion. The first call should be made with FLAG=1. This causes all internal data structures to be initialized and causes the routines not to depend on the values currently in DSTATE. At the end of the first call, the routines set FLAG to zero, update T, and put valid derivatives in DSTATE. On subsequent calls, they assume that DSTATE contains the valid derivative of STATE at T. Anytime a change has been made which invalidates DSTATE, set FLAG=1 again. (This is not necessary, however, if DT=0, since the invalid DSTATE will not be referenced.)

Position and velocity constraint errors are stabilized using Baumgarte's method, feeding the constraint position and velocity errors back to the acceleration equations (see Section R4.2). SDMOTION() returns with ERR=3 if any of these constraints are violated by more than CTOL. SDMOTION() will return with T just before the step at which the constraints would have been violated (thus the constraints are not actually violated at the point where SDMOTION() returns). SDFMOTION() returns ERR=1 in this case which is just a warning. It always integrates across the entire interval and simply reports the constraint violation as a warning. If you find that constraints are violated during the simulation, the Baumgarte constants should be increased and/or the integration tolerance TOL (or the step size DT for SDFMOTION()) should be decreased and the analysis should be re-run from the beginning.

SDMOTION() has additional error returns of 1 and 2. ERR=1 means that it reduced the step size to its smallest allowable value, without being able to reduce the error to below TOL. This normally corresponds to integrating over a step in some function, which usually produces acceptable errors. However, if the step was unexpected it is likely to indicate something wrong with the way the problem is set up. The integration proceeds over the whole interval even if ERR=1 is returned. It is simply a warning. ERR=2, on the other hand, is more severe. It indicates that the relative error did not improve on the other side of the minimum-size step. This generally means that the integration cannot proceed, possibly because some function is leaping towards infinity. This is often an indication of a lock-up configuration having been reached, so that constraints are about to become inconsistent. In this case the returned T is that just before the first (very small) step for which the errors were not going to be below TOL.

Regardless of the error return value, SDFMOTION() always integrates across the entire interval, so T has always moved to T+DT on return. SDMOTION() halts integration early if the error return is 2 or 3; otherwise, it integrates across the full interval.

After a call to either routine, you may call any of the data access routines (SDPOS(), SDANGACC(), etc.) to obtain information about the system at time T+DT. That is, these routines guarantee that the last calls to SDSTATE() and SDDERIV() were made at the final state, so you do not need to call them again before calling routines which depend on them. The returned DSTATE is always the one valid for the returned T and STATE.

C language

```
sdassemble(t,state,lock,tol,maxevals,fcnt,err)
double t,state[NQ+NU],tol;
int lock[NU],maxevals,*fcnt,*err;

sdinitvel(t,state,lock,tol,maxevals,fcnt,err)
double t,state[NQ+NU],tol;
int lock[NU],maxevals,*fcnt,*err;

sdstatic(t,state,lock,ctol,tol,maxevals,fcnt,err)
double t,state[NQ+NU],ctol,tol;
int lock[NU],maxevals,*fcnt,*err;

sdsteady(t,state,lock,ctol,tol,maxevals,fcnt,err)
double t,state[NQ+NU],ctol,tol;
int lock[2*NU],maxevals,*fcnt,*err;

sdmotion(t,state,dstate,dt,ctol,tol,flag,err)
double *t,state[NQ+NU],dstate[NQ+NU],dt,ctol,tol;
int *flag,*err;

sdfmotion(t,state,dstate,dt,ctol,flag,errest,err)
double *t,state[NQ+NU],dstate[NQ+NU],dt,ctol,*errest;
int *flag,*err;
```

R19 System Description File

This section describes in detail the rules for construction of an SD/FAST System Description File. It does not explain *why* you might want to put a particular construct into a System Description File, just *how* to do so. The purpose of each construct is explained in the Tutorials and in the Reference Sections for the individual topics. A complete System Description File is shown at the end of this section. This information is summarized in the SD/FAST Quick Reference Guide, Section Q1 on page R-2

In the material below, the following conventions are followed for defining syntax:

- Text in `typewriter font` represents a literal string which must appear in the System Description File at the indicated point. Upper and lower case letters are considered indistinguishable except in a prefix string (see below).
- Angle brackets around text in *italics font* indicate that the text is a description of the element which should appear at that point, rather than the literal element. For example, <*bodyname*>.
- Square brackets around a construct indicate that it is optional. For example,

  [`this is optional stuff`].
- Curly braces around a construct indicate that it is required. For example,

  { `this must appear` }.
- The "or" bar indicates that a choice is to be made, for example

  [`choice1` | `choice2`].
- A "*" following any construct indicates repetition (0 or more if the construct is enclosed in square brackets, 1 or more if curly braces. For example,

  [`zero or more of these`]*

  { `at least one of these` }*

R19.1 System Description File Elements

Their are ten individual elements of the System Description File: *keywords*, *names*, *joint types*, *real numbers*, *integers*, *logicals*, *question marks*, *prefix string*, *whitespace*, and *comments*. Below, each of these elements is described in detail.

1. Keywords

Keywords are predefined sequences of letters and digits, having special meaning to SD/FAST when they appear in the appropriate places in the input file. A keyword may require a value, in which case the keyword is followed by an equal sign and the value. A complete list of SD/FAST-recognized keywords and the values they expect is shown in Table R-3.

2. Names

Names are user-chosen sequences of up to 32 letters, digits, and underscores (_). A name must begin with a letter except for names predefined by SD/FAST, which begin with a dollar sign (`$`). Two kinds of names are currently used: body names and user constraint names. The only predefined name is `$ground`, the name for the ground body. Otherwise any name may be used — it does not matter if the name duplicates an SD/FAST keyword.

Table R-3 Keywords

Preamble keywords	Data expected
`gravity`	*vector* (3 *real numbers*)
`language`	`fortran,c`
`prefix`	*prefix string*
`single/double`	none
Body and joint keywords	
`body`	*name* (of body)
`mass`	*real number*
`inertia`	*3×3 matrix* (3 or 9 *real numbers)*
`inboard (inb)`	*name* (of body)
`joint`	`weld,pin,ujoint,gimbal,ball,` `slider,cylinder,[r]planar,` `[r]bearing,[r]free,[r]bushing`
`bodytojoint (btj)`	*vector*
`inbtojoint (itj)`	*vector*
`pin`	*vector*
`prescribed`	none, or *logical* (`1` or `0`) for each joint dof
Loop joint only keywords	
`bodypin`	*vector*
`inbref`	*vector*
`bodyref`	*vector*
User constraint keywords	
`constraint`	*name*
`constraints`	*integer* (number of user constraints)

3. Joint Types

Joint types are keywords which are recognized only after the keyword `joint`. These indicate which type of joint is being defined, for example `joint=slider`.

4. Real Numbers

Real numbers (i.e., floating point numbers) are specified as follows:

[+|−][<*digits*>][`.` [<*digits*>]][`e` [+|−] <*digits*>]

that is, an optional sign followed by a number like `1`, `1.`, `.1`, or `1.1`, followed optionally by an exponent. Note that only the letter “`e`” (upper or lower case) can be used to introduce the exponent — “`d`” is not allowed. Nevertheless, the numbers are interpreted as double precision.

5. Integers

Integers are simply strings of digits (unsigned). These are used only to provide the number of user constraints in the system.

6. Logicals

Logicals are represented by the integer `1` (true) and `0` (false). These are used only after the `prescribed` keyword to say for each hinge of a joint whether that hinge is to follow prescribed motion.

7. Question Marks

```
mass = ?

bodytojoint= 2.5? 3.72? 0

prescribed = 1?

inertia = 1.5  .5  .6

            ?   2  .7

            ?   ?   3
```

Question marks are used to indicate for a particular system parameter that the generated equations should allow variation of that parameter at run time. A question mark can be used instead of or in addition to the specification of a real number or a logical. When a value is supplied, the question mark is placed immediately after the number, with no intervening spaces, e.g. "`24.2?`". In this case the value is a default which can be changed at run time.

As a special case to avoid duplicate entry, if a question mark appears in an off-diagonal position of a full (9 numbers) inertia matrix (which must be symmetric), and the corresponding symmetric term is not a question mark, then the question mark is simply replaced by the symmetric term — no variability is indicated in that case.

8. Prefix String

A prefix string is provided following the keyword `prefix`. It defines a string of characters which is to be prepended to the names of all the generated subroutines. The string should be composed only of characters which will produce legitimate routine names in the target language, although SD/FAST will not check. In C, the case (upper or lower) of the letters in the string is preserved. By default, the prefix string is "`sd`".

9. Whitespace

Whitespace (blank, newline, tab, and form feed) may appear between any elements in the file without altering the meaning. Whitespace may not appear inside any keyword, name, or number. When a variable parameter is specified with a default (e.g., "`mass=10?`") no whitespace may appear between the number and the question mark.

10. Comments

Comments are introduced by the "`#`" character and continue to the end of the line. The meaning of the System Description File is the same as if the "`#`" and all the characters following it on the same line were removed.

R19.2 System Description File Structure

The SD/FAST System Description File is divided conceptually into four sections, all but one of which is optional. The sections are provided as shown below. A description of each section follows.

```
[ <preamble> ]
<body and tree joint section>
[ <loop joint section> ]
[ <constraint section> ]
```

Preamble

This optional section consists of specifications which affect the system as a whole. There are four possible specifications here: (1) gravity, (2) output language, (3) prefix for generated routines, and (4) single or double precision selection. Except for gravity, these specifications can also be set from the command line when SD/FAST is invoked. Examples:

```
gravity = 0 -9.8? 0
language = c
prefix = mod1
single
```

Language, prefix and precision can also be given on the SD/FAST command line; see Section R7.

Body And Tree Joint Section

Recall that the system must be modeled as a tree of bodies with a “root” at `$ground`, plus some loop joints (if needed) to produce closed topological loops. In the body and tree joint section, each of the bodies is described, beginning with a body which is connected to `$ground`. (A free-flying system, that is, one which does not specify a connection to `$ground`, has an implicit “free joint” connection to `$ground`.) The mass and inertia properties are given and the inboard joint type and location are provided. Each body/joint description is entered like this:

```
body = <bodyname1>  inboard = <bodyname2>  joint = <jointtype>
  mass = <real>
  inertia = <real><real><real> |  <real><real><real>
                                  <real><real><real>
                                  <real><real><real>
  bodyToJoint = <vector>
  inbToJoint  = <vector>
  [pin = <vector>]*
  [prescribed [ = {1|0}* ]]
```

The `body`, `inboard`, and `joint` parameters should be provided in the order shown (they do not have to be on the same line, however). The remaining parameters may be listed in any order which is convenient. *<bodyname2>*, the name of the inboard body, must either be `$ground` or the name of a body which was defined earlier in the System Description File.

See Section R13 for definitions and restrictions which apply to the `mass` and `inertia` parameters.

The number of pin vectors provided must match the number of axes in the specification of the *<jointtype>* being defined. That is, no pins for a weld or ball joint; one pin for a pin, slider, or cylinder joint; two pins for a U-joint; and three pins each for a gimbal, planar, bearing, bushing or free (6dof) joint. In joints with both translational and rotational axes (cylinder, planar, bearing, and bushing), the pins for the sliding axes are defined first unless the joints are reversed. See Section R11 for more information.

If the “=” appears after `prescribed` the number of `1`’s and `0`’s following it must be equal to the number of degrees of freedom provided by the joint being defined.

Loop Joint Section

Loop joints are specified in a manner similar to tree joints. However, both the inboard and outboard bodies must already have been defined in the tree section, and no mass properties are required or allowed.

```
body = <bodyname1>  inboard = <bodyname2>  joint = <jointtype>
   bodyToJoint = <vector>
   inbToJoint  = <vector>
   [pin = <vector>]*
   [bodypin = <vector>]
   [inbref  = <vector>]
   [bodyref = <vector>]
   [prescribed [ = {1|0}* ]]
```

The `body`, `inboard`, and `joint` parameters should be provided in the order shown (they do not have to be on the same line, however). The remaining parameters may be listed in any order which is convenient. *<bodyname1>*, the name of the “outboard” body, and *<bodyname2>*, the name of the “inboard” body, must either be `$ground` or the names of bodies which were defined earlier in the tree section of the System Description File.

The number of pin vectors provided must match the number of axes in the specification of the *<jointtype>* being defined. The number required is the same as described under tree joints above, except that a weld joint requires one pin. `Bodypin`, `inbref`, and `bodyref` may only be specified if allowed by the specification of *<jointtype>* when used as a loop joint. The *<jointtype>* specification will also place some perpendicularity restrictions on the various pins and reference lines. See Section R11 for more information.

If the “=” appears after `prescribed` the number of `1`’s and `0`’s following it must be equal to the number of degrees of freedom provided by the joint being defined.

Constraint Section

Currently the only type of constraint which may be specified in this section is the User Constraint. These can be given individual names, or you can simply specify how many user constraints there are. In that case, SD/FAST will choose names for the constraints, like `user_1`.

The syntax is:

```
constraint = <name> | constraints = <integer>
```

These lines may be repeated as many times as necessary. If the `constraints=`*<integer>* form appears more than once, the total number of user constraints will be the sum of the specified values.

R19.3 Example

The following is an example of a complete SD/FAST System Description File for a four-bar crank and rocker mechanism. This file has a preamble, three tree joints and a loop joint. There is no constraint section.

preamble

body and tree joints

loop joints

```
# A 4-bar linkage (crank and rocker)
#                                                y|
#                 connect                         |
#              *------------o                     |______ x
#              |            |                    /
#              *            | rocker           z/
#        crank |            |
#              o------------o               * = loop joint
#                  __|__
#                 //////
#
# Reference configuration is as shown.  Note that system is not
# assembled.  All joints, including the loop joint, are pins.
# The crank has length 1, the rocker has length 2, and the connecting
# bar and ground baseline each have length 3.  The crank is attached to
# ground at the origin.
#
gravity = 0 -9.8 0

body = crank inb = $ground joint = pin
  mass = 1                        inertia = 1 0 1
  bodytojoint = 0 -.5 0           inbtojoint = 0 0 0     pin = 0 0 1
body = rocker inb = $ground joint = pin
  mass = 2                        inertia = 2 0 2
  bodytojoint = 0 -1 0            inbtojoint = 3 0 0     pin = 0 0 1
body = connect inb = rocker joint = pin
  mass = 3                        inertia = 0 3 3
  bodytojoint = 1.5 0 0           inbtojoint = 0 1 0     pin = 0 0 1

# This is the loop joint.
body = connect inb = crank joint = pin
  bodytojoint = -1.5 0 0          inbtojoint = 0 .5 0    pin = 0 0 1
```

R20 System State

SD/FAST makes use of a state vector to represent the position and velocities of a multibody system. The elements of the state vector are called generalized coordinates and generalized speeds. They are given the symbols q and u, respectively. The generalized coordinates determine the configuration of the multibody system. The location of every material point in the system can be computed from knowledge of the generalized coordinates. In particular, the location of every body's mass center and the orientation of each body in inertial space is determined by the generalized coordinates. The generalized speeds determine the motion of the multibody system. Given the configuration of the system, the velocity of every material point in the system is determined by the generalized speeds. In particular, the linear velocity of every body mass center, and the angular velocity of every body is determined by the generalized speeds.

R20.1 Number of Coordinates and Speeds

SD/FAST introduces coordinates and speeds that are associated with motion permitted by the joints in the system. For example, a pin joint permits one degree of freedom between the connected bodies, and the joint rotation angle and rotation rate are part of the SD/FAST state vector. In describing the multibody system to SD/FAST, the user must specify a spanning tree plus loop closures (see Section R16.1). The joints included in the spanning tree determine the number of generalized coordinates and generalized speeds. The number of coordinates `NQ` is a simple function of the number and type of each joint in the tree system:

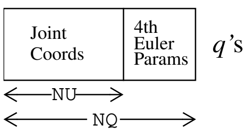
`NQ` = *# of pins* + *# of sliders* + 2 * *# of cylinders* + 2 * *# of U-joints* +
3 * *# of gimbals* + 3 * *# of planars* + 4 * *# of bearings* +
6 * *# of bushings* + 4 * *# of balls* + 7 * *# of frees*

The number of generalized speeds is a similar function of the number and type of each tree joint:

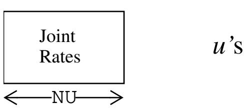
`NU` = *# of pins* + *# of sliders* + 2 * *# of cylinders* + 2 * *# of U-joints* +
3 * *# of gimbals* + 3 * *# of planars* + 4 * *# of bearings* +
6 * *# of bushings* + 3 * *# of balls* + 6 * *# of frees*

Note that the difference between the number of coordinates and the number of speeds is due only to ball joints (including ball joints that make up free joints). Ball joints contribute four elements to the coordinate array, but only three elements to the speed array. The coordinates used to describe all joints except ball joints are joint rotation angles for rotational degrees of freedom, and sliding displacements for sliding degrees of freedom. Ball joint orientation is described using a set of four Euler parameters (see Section R6). The speeds used to describe all joints except balls are the rate of change of the joint coordinates, namely rotation rate and sliding rate. Ball joint motion is described using the three measure numbers of the relative angular velocity vector of the connected bodies, expressed in the outboard body frame.

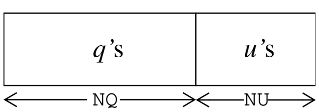
R20.2 Organization of the State Array



The NQ elements in the coordinate state array *q* are labelled in the order in which tree joints are defined in the input System Description File. (This is documented for each system in the generated Information File, as described in Section R9.) The coordinates *q* for each joint are ordered after the coordinates for the previous joint, except for ball-containing joints (ball, free, and rfree joints). Ball joint coordinates (four Euler parameters, see Section R6) are split into a group of three elements and a single scalar. The first three Euler parameters are placed immediately after the coordinates for the previous joint, just as for any other joint. The fourth Euler parameter is grouped with all the other fourth Euler parameters in the system at the end of the coordinate array. Thus the coordinate array has in its first NU elements the first three Euler parameters for all ball joints, plus all other system coordinates, followed by NQ−NU fourth Euler parameters.



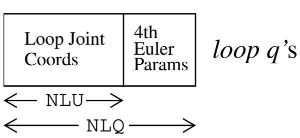
The NU elements in the speeds state array *u* are the collection of generalized speeds, arranged in the same order as the tree joints in the System Description File. Again, the generated Information File provides the details for each individual system. Because of the splitting of the Euler parameters into two groups as described above, the generalized speeds for a particular joint begin at the same location within the *u* array as do the generalized coordinates in the *q* array.



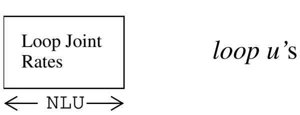
Composite State Array

Low-level SD/FAST-generated routines deal separately with *q*'s and *u*'s. Some of the higher-level analysis routines (for example, `SDASSEMBLE()` and `SDMOTION()`) take as input a composite state array which is formed by simply concatenating the coordinate and speed arrays. The first NQ elements of the resulting array are the coordinates *q*, and the last NU elements are the speeds *u*.

R20.3 Pseudo States for Loop Joints



Only tree joints contribute elements of the *q* and *u* arrays. However, user applications may require the same coordinate information for loop joints as for tree joints. SD/FAST provides the values of *pseudo-coordinates* and *pseudo-speeds* corresponding to loop joints via the generated routine `SDPSEUDO()`, discussed below. The loop *q*'s are organized similarly to the tree *q*'s, and the loop *u*'s are organized similarly to the tree *u*'s. The primary difference between the loop states and the tree states is that the tree states can be assigned initial conditions. The values of the loop *q*'s and *u*'s are completely determined by the tree variables. In essence, the relative rotation or translation of all loop joints is computed in terms of the tree states. The values of the loop variables are then found by decomposing the relative rotation and translation into a coordinate description.



Pseudo-coordinates for rotational loop joint axes are always reported in the range $-\pi$ to π, with the exception of the middle rotational axis of a loop gimbal-containing joint which is restricted to the range $-\pi/2$ to $\pi/2$. To track the actual number of rotations about a loop joint axis, you should integrate the pseudo-coordinate derivatives as described below. In most cases this is unnecessary.

There are four SD/FAST-generated routines which deal explicitly with the loop joint pseudo-state variables.

```
SUBROUTINE SDPSEUDO(LQ,LU)
DOUBLE PRECISION LQ(NLQ),LU(NLU)

SUBROUTINE SDPSQDOT(LQDOT)
DOUBLE PRECISION LQDOT(NLQ)

SUBROUTINE SDPSUDOT(LUDOT)
DOUBLE PRECISION LUDOT(NLU)

SUBROUTINE SDPSSTATE(LQ)
DOUBLE PRECISION LQ(NLQ)
```

`SDPSEUDO()` returns loop joint pseudo-coordinates in `LQ` and `LU`. These are valid any time after `SDSTATE()` has been called.

`SDPSQDOT()` and `SDPSUDOT()` return the derivatives of the loop joint pseudo coordinates. `SDPSQDOT()` may be called any time after `SDSTATE()` while `SDPSUDOT()` is callable only after `SDDERIV()` has been called.

The pseudo-state variables are organized similarly to the state variables `Q` and `U`. `LQ` contains the loop joints' hinge position coordinates, with loop ball joints represented by four Euler parameters. The fourth Euler parameters are in elements `LQ(NLU+1)` to `LQ(NLQ)`, while the first three are left in slots of `LQ` which correspond to the angular velocity of that same joint in `LU`. This way the numbering for `LQ` and `LU` matches except for the fourth Euler parameters, and `SDINDX()` can be used to index both `LQ` and `LU` in the (*joint*, *axis*) format.

The `LQ`'s are calculated from body orientations and do not include any "tumbling" information. Rotational (non-slider, non-ball) `LQ`'s will always range between the limits specified above regardless of how many times the joint may have rotated. If it is desired to track number of rotations, the `LQDOT`'s returned by `SDPSQDOT()` should be integrated and the resulting `LQ`'s reported instead of the `LQ`'s returned here.

An optional call to `SDPSSTATE()` may (immediately) follow the `SDSTATE()` call to replace the SD/FAST-calculated values of the loop q's with user-calculated values. This is needed only when prescribed motion is specified at a rotational (i.e., non-slider, non-ball) loop joint hinge and the prescribed motion is outside the limited range returned by `SDPSEUDO()`. Only the elements of `LQ` corresponding to prescribed rotational hinges need be filled in; the rest are ignored. The correct values for these loop q's should be calculated by integrating the loop $\dot{q}$'s returned by `SDPSQDOT()`. This is rarely necessary, however. In most cases you can simply choose a different joint as the loop joint so that you can prescribe motion at a tree joint. Or, just prescribe the joint velocity. See Section R14.

C language

```
sdpseudo(lq,lu)
double lq[NLQ], lu[NLU];

sdpsqdot(lqdot)
double lqdot[NLQ];

sdpsudot(ludot)
double ludot[NLU];

sdpsstate(lq)
double lq[NLQ];
```

R20.4 Invalid State Arrays

Normally, the generalized coordinates and speeds in a state array may be set to any numerical values. These values might not satisfy system constraints, but they still constitute a legitimate state for passing in to `SDSTATE()`. (That is very important when the type of analysis you are doing is one which tries to find a state which *does* satisfy the constraints, for example assembly analysis.) However, there are three conditions which are considered an invalid state. These are (1)gimbal lock, (2) badly unnormalized Euler parameters, and (3) singular mass matrix. A state exhibiting either of these conditions cannot be used for further analysis.

The `SDSTATE()` routine will post an error if either of these conditions is detected. See the discussion on that routine in Section R3.2 for more information.

R20.5 Accessing Elements of the State Array

The specific assignment of joint axis coordinates and speeds to elements of the state array is given in the generated Information File for each system (see Section R9). To increase program flexibility and clarity, and to reduce the potential for error, we recommend that user programs not use this information when accessing individual elements of the state array. Instead, SD/FAST provides an integer function `SDINDX()` which maps a (*joint number, axis number*) pair to a state array index. This function can be used directly as an index for the state arrays, so the actual index number does not have to appear in your program at all. If the joint number is provided using a variable or parameter name rather than as a numerical constant, your program will be easier to maintain if model changes are made.

```
FUNCTION SDINDX(JOINT,AXIS)
INTEGER JOINT,AXIS,SDINDX
```

`SDINDX()` is useful for converting from the (*joint, axis*) specification of a coordinate or speed to its location in the `Q`, `U`, `QDOT` or `UDOT` arrays. In addition, it can be used to access elements of a composite `STATE` or `DSTATE` (state derivatives) array. For example, say joint 4 is a U-joint which is the inboard joint of body `ANTENNA`. The initial pin angles are 1.2 and 1.5 radians and the initial velocities are zero. `SDINDX()`

can be used to initialize either individual coordinate and speed arrays or a composite state array as follows:

```
parameter (NQ=…, NU=…, ANTJNT=4)

q(SDINDX(ANTJNT,1)) = 1.2            state(SDINDX(ANTJNT,1)) = 1.2
q(SDINDX(ANTJNT,2)) = 1.5            state(SDINDX(ANTJNT,2)) = 1.5
u(SDINDX(ANTJNT,1)) = 0.             state(NQ+SDINDX(ANTJNT,1)) = 0.
u(SDINDX(ANTJNT,2)) = 0.             state(NQ+SDINDX(ANTJNT,2)) = 0.
```

If the specified joint is a ball or sixdof joint, the 4th Euler parameter can be located by specifying “axis” 4 (for balls) or “axis” 7 (for free joints). `SDINDX()` also returns correct indices into the pseudo-state variables `LQ`, `LU`, `LQDOT` and `LUDOT` when the passed-in joint is a loop joint. (Loop joints are numbered starting after the last tree joint.) An out-of-range joint or axis number produces an undefined return value and posts an error condition.

R20.6 Example

Suppose we want to assign initial conditions to the coordinates and speeds of a two-body spacecraft. The system is modeled with a free joint connection between the first body (called `BUS`) and ground, plus a pin joint between the second body (called `ROTOR`) and the first. From the generated Information File, we would find that the first three coordinates are sliding displacements, followed by three Euler parameters, followed by the pin joint rotation, followed by the fourth Euler parameter. The speeds are three sliding velocities, then three angular velocity components, followed by the pin joint rotation rate. The above description is “canonical” in the sense that SD/FAST will make this choice for any two body spacecraft with a pin joint. There is nothing problem-specific about the order or definition of the state variables for such a two-body system.

The following program fragment illustrates how a user program can create the `Q` and `U` array by using the integer function `SDINDX()`. The initial values of the coordinates and speeds are read in. The orientation read in is provided as Euler angles rather than the required Euler parameters. The Euler angles are converted to Euler parameters by the generated routine `SDANG2ST()` (see Section R6).

```
      parameter (NQ=8,NU=7,BUS=1,ROTOR=2)
      double precision t,q(NQ),u(NU)
      integer SDINDX
      double precision x0,y0,z0,th1,th2,th3,phi
      double precision v1,v2,v3,w1,w2,w3,phidot

      read (5,*) x0,y0,z0,th1,th2,th3,phi
      read (5,*) v1,v2,v3,w1,w2,w3,phidot
      call SDINIT
c
c initialize the coordinate array q and convert
c
      q(SDINDX(BUS,1)) = x0
      q(SDINDX(BUS,2)) = y0
      q(SDINDX(BUS,3)) = z0
      q(SDINDX(BUS,4)) = th1
      q(SDINDX(BUS,5)) = th2
      q(SDINDX(BUS,6)) = th3
      q(SDINDX(ROTOR,1)) = phi
      call SDANG2ST(q,q)
c
c initialize the speed array u
c
      u(SDINDX(BUS,1)) = v1
      u(SDINDX(BUS,2)) = v2
      u(SDINDX(BUS,3)) = v3
      u(SDINDX(BUS,4)) = w1
      u(SDINDX(BUS,5)) = w2
      u(SDINDX(BUS,6)) = w3
      u(SDINDX(ROTOR,1)) = phidot

      t = 0d0
      call SDSTATE(t,q,u)
      ...
```

R21 Units of Measure

The equations of motion generated by SD/FAST are unitless. That is, there is no way to tell from the SD/FAST input System Description File what system of units is being used. The user must use a system which makes dimensional sense (i.e., Newton's Law is valid!). Common systems of units are SI (meters, kilograms, Newtons, seconds), CGS (centimeters, grams, dynes, seconds), English (feet, slugs, pounds, seconds). Inertia will then have units of *mass**length*^{2}, torque is measured in *force**length*, etc. Gravity at sea level has the value 9.80665 m/s^2, 980.665 cm/s^2, and 32.174 ft/s^2.

Angular measure is always in radians.

R22 Usage Errors

There are a large number of simple errors which can easily be made in the use of the SD/FAST-generated routines. For example, an illegal body number might be passed to `SDPOS()`. Or, a call to `SDACC()` might be made before the derivatives have been calculated with `SDDERIV()`. The user might forget to include a call to `SDINIT()` after modifying a system parameter.

Most errors of this sort can easily be detected by the SD/FAST routines themselves. Each routine performs some checks to be sure it has been called correctly. If not, the routine "posts" an error notification, which will remain posted until explicitly cleared. (Only the first error posted is remembered, in case there are multiple usage errors.) At some convenient point in the user's program (usually right at the end) a call can be made to an SD/FAST routine which will check for a posted error and return or print out the error message. Should a message be displayed, the programmer can use it to locate the incorrect call and fix it.

Note that this error handling facility is primarily for *debugging*, concerned with proper usage of the SD/FAST-generated routines. It is not used to detect run time events such as failure to find a static configuration, system lockup, or constraint violation during integration — these conditions are reported separately by the appropriate routines.

Table R-4 SD/FAST Usage Errors

Error No.	Meaning
1	a tree joint pin axis was zero
2	the 1st inboard pin for a loop joint was zero
3	the 2nd inboard pin for a loop joint was zero
4	the 3rd inboard pin for a loop joint was zero
5	an inboard reference line was zero
6	a set of loop joint axes was not right handed
7	a loop joint bodypin was zero
8	a loop joint body reference line was zero
9	1st/2nd pins in a loop joint not perpendicular
10	2nd/3rd pins in a loop joint not perpendicular
11	1st/3rd pins in a loop joint not perpendicular
12	a loop jt pin and inbref were not perpendicular
13	a bodypin and bodyref were not perpendicular
14	Euler parameters were far from normalized
15	illegal body number
16	illegal joint number
17	illegal axis number
18	illegal axis number for this joint
19	tried to set non-variable (i.e., non-?) parameter
20	prescribed motion was neither 0 (off) or 1 (on)
21	illegal user constraint number
22	SDINIT must be called first
23	SDSTATE must be called first
24	SDDERIV must be called first
25	a gravity ? parameter is unspecified
26	a ? mass is unspecified
27	a ? inertia is unspecified
28	a ? tree jt pin is unspecified
29	a ? tree bodyToJoint vector is unspecified
30	a ? tree inbToJoint vector is unspecified
31	a ? prescribed tree jt axis is unspecified
32	the stabvel ? parameter is unspecified
33	the stabpos ? parameter is unspecified
34	a ? loop jt inboard pin is unspecified
35	a ? loop jt inbref is unspecified
36	a ? loop jt bodypin is unspecified
37	a ? loop jt bodyref is unspecified
38	a ? loop jt bodyToJoint vector is unspecified
39	a ? loop jt inbToJoint vector is unspecified
40	a ? prescribed loop jt axis is unspecified
41	Dynamics & Library File serial nos. differ
42	Dynamics & Analysis File gen. times differ
43	A tree gimbal joint is in gimbal lock
44	A loop gimbal joint is in gimbal lock
45	Bad tree coordinate number
46	Can’t rotate about a zero vector
47	Singular mass matrix

Table R-5 SD/FAST Routine Numbers

Routine No.	Name	Routine No.	Name
1	SDGRAV	32	SDACC
2	SDMASS	33	SDANGACC
3	SDINER	34	SDMULT
4	SDBTJ	35	SDAERR
5	SDITJ	36	SDINDX
6	SDPIN	37	SDPRES
7	SDINIT	38	SDSTAB
8	SDSTATE	39	SDGETGRAV
9	SDPSSTATE	40	SDGETMASS
13	SDPRESACC	41	SDGETINER
14	SDPRESVEL	42	SDGETBTJ
15	SDPRESPOS	43	SDGETITJ
10	SDHINGET	44	SDGETPIN
11	SDPOINTF	45	SDGETPRES
12	SDBODYT	46	SDGETSTAB
17	SDDERIV	47	SDINFO
16	SDRESID	48	SDJNT
18	SDPSEUDO	49	SDCONSB
19	SDMOM	50	SDASSEMBLE
20	SDSYS	51	SDINITVEL
21	SDPOS	52	SDSTATIC
22	SDVEL	53	SDSTEADY
23	SDORIENT	54	SDMOTION
24	SDANGVEL	55	SDFMOTION
25	SDTRANS	56	SDEQUIVHT
26	SDPERR	57	SDMASSMAT
27	SDVERR	58	SDFRCMAT
28	SDPSQDOT	59	SDREL2CART
29	SDPSUDOT	60	SDCOMPTRQ
30	SDGETHT	61	SDFULLTRQ
31	SDREAC	62	SDVROT

```
SUBROUTINE SDERROR(ROUTINE,ERRNO)
INTEGER ROUTINE, ERRNO

SUBROUTINE SDPRINTERR(FILE)
INTEGER FILE

SUBROUTINE SDCLEARERR()
```

`SDERROR()` sets `ERRNO` to 0 if no usage errors have occurred since the beginning of the program or since the last call to `SDCLEARERR()`. Otherwise, it returns the error number of the first error which occurred and the routine number of the routine in which the error was detected. An exhaustive list of the returned error numbers is given in Table R-4. The complete list of routine numbers for the routines that can return errors using this facility is given in Table R-4.

`SDPRINTERR()` prints the routine name and a description of the error to the file number indicated by its argument. This is the same information as would be returned by an `SDERROR()` call, although in a human-readable form. If no error has been posted, nothing is printed.

`SDCLEARERR()` clears any indication that an error has occurred, so that the next error to occur will be the one reported by `SDERROR()` or `SDPRINTERR()`.

The simplest use of this facility is to place a call to `SDPRINTERR()` at the end of your program, just before you terminate. Or, you may want to place a call at the end of each analysis you perform. If you have not made any errors in using the generated routines, there will be no error reported. If any errors have been made, the first one encountered will be reported. You should then examine your call(s) to the reported routine to see what is wrong. If there are multiple calls, you may have to add additional `SDPRINTERR()` calls earlier in the program to track down which one is failing. Once they have all been fixed, you can go back to having just one `SDPRINTERR()` call at the end. You should always leave in at least this one call to guard against errors which may be introduced in the future or which may not have been caught due to unexecuted branches of the program.

`SDERROR()` is used when you want to modify the behavior of your program as a consequence of an error having occurred. The most common behavior modification is to print out a message and terminate.

C language

```
#include <stdio.h>

sdprinterr(file)
FILE *file;

sderror(routine,errno)
int *routine, *errno;

sdclearerr();
```

R23 Usage with ACSL

ACSL (Advanced Continuous Simulation Language) is a commercial product marketed by MGA, Inc. It is just one of many analysis tools which can be used with SD/FAST. Other popular analysis tools such as Matrix-X, Pro-Matlab, Simulink, and Easy5 can be used equally as well with SD/FAST. SD/FAST does not contain any special interface to ACSL; rather, SD/FAST is designed to interface easily with other programs. We show use with ACSL here since it is one of the more popular environments employed by SD/FAST users. This can also serve as a suggestive example for use of SD/FAST in other environments, since the interface method is similar.

R23.1 Choice of Precision

Please note that we do not recommend use of single precision SD/FAST-generated code on machines whose single precision floating point numbers occupy fewer than 56 bits, especially for systems with constraints. Therefore we suggest that ACSL always be used in double precision mode.

R23.2 Linking SD/FAST Routines to ACSL

Normally, you will link your ACSL program with only the SD/FAST-generated Dynamics File and Library File, using a command like this (on the Sun):

```
acsl myprog -lib "myprog_dyn.o sdlib.o"
```

A similar command would be used on other computer systems. See your System Administrator for advice.

If you want to use any of the SD/FAST-generated Simplified Analysis Routines like `SDASSEMBLE()` and `SDINITVEL()`, you should link in the Simplified Analysis File (e.g. `myprog_sar.o`) as well. Note that these routines expect to be able to access `sduforce()` and `sdumotion()` user-written Fortran routines. That means you will need to provide at least stubs for these routines at the end of your ACSL program if you want to link successfully with the Simplified Analysis File.

Instead of using `SDASSEMBLE()`, another way to perform assembly analysis with ACSL is to use its `ANALYZ 'TRIM'` feature. In this case, make sure there are no forces acting on the system and that the Baumgarte stabilization constants are non-zero (see Section R4.2). Then the only cause of accelerations is the Baumgarte stabilization feedback constants. `ANALYZ 'TRIM'` will try to make those accelerations go to zero, which drives the system towards assembly.

R23.3 ACSL Program Structure

Here is the basic skeleton for an ACSL program using SD/FAST:

```
PROGRAM

INITIAL
1. set variable system parameters              SDMASS,SDPIN,...
2. initialize system                           SDINIT
3. set stabilization parameters if desired     SDSTAB
4. perform assembly and initial velocity       SDASSEMBLE
   analysis if needed                          SDINITVEL
END

DERIVATIVE
5. compute forces and motions                  (ACSL code)

PROCEDURAL
     6. set time and state                     SDSTATE
     7. apply forces                           SDHINGET,SDBODYT,SDPOINTF
     8. prescribe motions                      SDPRESPOS,...
     9. compute derivatives                    SDDERIV
END

10. integrate (ACSL integ or intvc)
END

TERMINATE
11. check for usage errors                     SDPRINTERR
END

END

         <provide any needed Fortran code here>
```

R23.4 Examples

Here is an example of a two-body spring-mass system with variable masses. There are no constraints. This assumes that ACSL is running in double precision mode or the equations have been generated in single precision (but see Section R23.1 above).

```
PROGRAM
    'initial masses, spring constants, and simulation stop time'
    constant mass1 = 10., mass2 = 10., k1 = 5., k2 = 5., tstop = 5.

INITIAL
    call SDMASS(1, mass1)
    call SDMASS(2, mass2)
    call SDINIT
END

DERIVATIVE
    real qic(2), uic(2), q(2), u(2), qdot(2), udot(2)
    constant qic = 0.,0.,  uic = 1., 1.     $ 'initial conditions'
    f1 = -k1*q(1)                           $ 'calc. spring forces'
    f2 = -k2*q(2)
    PROCEDURAL(qdot,udot=t,q,u,f1,f2)       $ 'compute derivatives'
        call SDSTATE(t,q,u)
        call SDHINGET(1,1,f1)
        call SDHINGET(2,1,f2)
        call SDDERIV(qdot,udot=)
    END
    u = intvc(udot,uic)                     $ 'integrate'
    q = intvc(qdot,qic)
    termt(t .ge. tstop)
END

TERMINAL
   call SDPRINTERR(6)
END

END
```

The following more complex example shows use of the Simplified Analysis Routines `SDASSEMBLE()` and `SDINITVEL()` to set the initial conditions for a motion analysis. In addition, we use the SD/FAST-generated constraint error monitoring routines to track the degree to which constraints are violated during the motion analysis. We let the Baumgarte stabilization constants be set at runtime.

With the structure shown below, the ACSL user can at runtime restart the motion analysis using the ACSL `REINIT` command, which copies the current state into the initial conditions. Then the assembly and velocity analyses in the `INITIAL` section will serve to "touch up" the mechanism; that is, they will change the state only enough to bring the constraint errors below the tolerance `INTTOL`.

The system being simulated is the four-bar crank and rocker mechanism whose System Description File is shown in Section R19.3 on page R-112. There is a damper at the crank-to-ground joint. We're assuming that ACSL is running in double precision mode, or that the SD/FAST routines have been generated in single precision (but see Section R23.1 above).

```
PROGRAM
   real qic(3), uic(3)
   constant tstop = 2., damper = 3.   $ 'run time & damping constant'

   'These are initial guesses at the initial conditions.  The first'
   'must be met; the other two will be adjusted to be compatible.'
   constant qic=0.,0.,0.,   uic=1., 1., -1.

   'inttol is max tol for constraint errs after assembly & velocity'
   'analyses.  oktol says max err tolerated during motion analysis.'
   constant inttol=1e-6,    oktol = 1e-4

INITIAL
   real state(6)
   integer lock(3),fcnt,err,i
   constant lock = 1, 0, 0,   a=1.   $ 'a is Baumgarte constant'

   call SDINIT
   call SDSTAB(2.*a, a*a)

   do 5 i=1,3                        $ 'initialize state to qic,uic'
      state(i) = qic(i)
 5 .. state(3+i) = uic(i)

   call SDASSEMBLE(fcnt,err=t,state,lock,inttol,1000)
   print 10,err,fcnt
10 .. format(' sdassemble returned err=',i5,' fcnt=',i5)

   call SDINITVEL(fcnt,err=t,state,lock,inttol,1000)
   print 20,err,fcnt
20 .. format(' sdinitvel returned err=',i5,' fcnt=',i5)

   do 25 i=1,3                       $ 'set ICs to analysis results'
      qic(i) = state(i)
25 .. uic(i) = state(3+i)

   print 30,'qic=',qic
   print 30,'uic=',uic
30 .. format(' ', a5, 3f12.5)
END
```

```
DERIVATIVE
   real q(3), u(3), qdot(3), udot(3), maxerr, errs(5), trq
   trq = -damper * u(1)         $ 'calc. damping torque'

   PROCEDURAL(qdot,udot,maxerr=t,q,u,trq)
      call SDSTATE(t,q,u)          $ 'compute derivatives'
      call SDHINGET(1,1,trq)
      call SDDERIV(qdot,udot=)

      'check for excessive constraint errors'
      maxerr = 0.
      call SDVERR(errs=)
      do 40 i=1,5
40 ..    if (abs(errs(i)) .gt. maxerr) maxerr = abs(errs(i))
      call SDPERR(errs=)
      do 50 i=1,5
50 ..    if (abs(errs(i)) .gt. maxerr) maxerr = abs(errs(i))
   END

   u = intvc(udot,uic)          $ 'integrate'
   q = intvc(qdot,qic)

   if (maxerr .gt. oktol) print 60,maxerr
60 .. format (' max constraint error too big =', f12.5)

   termt((t .ge. tstop) .or. (maxerr .gt. oktol))
END

TERMINAL
   call SDPRINTERR(6)
END

END

       subroutine sduforce(t,q,u)
c This stub is present to avoid linker errors when linking with
c the Simplified Analysis Routines in the `_sar' file.
       double precision t,q(3),u(3)
       return
       end
```

R24 User Constraints

SD/FAST implements two types of constraints automatically: loop joint constraints and prescribed motion constraints. In addition to these, it provides a general mechanism called *user constraints* by which a user can implement arbitrary constraints. Examples of constraints which can be implemented this way are gears, screws, pulleys, distance constraints, tracks, pin-in-slot, and roll-without-slipping constraints. In general, any holonomic (position) constraint or non-holonomic (velocity) constraint can be implemented using user constraints.

The method used here allows specification of the constraint in a way in which you can account for the physical implementation of the constraint. This allows correct computation of the reaction forces both within the constraining components and elsewhere in the system.

The basic idea is that you write (1) a set of routines which, given the current system positions, velocities, and accelerations can return the error in the position, velocity and acceleration user constraints; and (2) a routine which, given the current state and SD/FAST-calculated Lagrange multipliers applies the appropriate forces to the system.

Although user written, the names of these routines must be exactly as shown below, since SD/FAST-generated code will call them by these names. (If the prefix has been changed from "`sd`" to something else, it should be changed here as well.)

```
subroutine sduperr(t,q,errs)
double precision t,q(NQ),errs(NUSERC)

subroutine sduverr(t,q,u,errs)
double precision t,q(NQ),u(NU),errs(NUSERC)

subroutine sduaerr(t,q,u,udot,errs)
double precision t,q(NQ),u(NU),udot(NU),errs(NUSERC)

subroutine sduconsfrc(t,q,u,mults)
double precision t,q(NQ),u(NU),mults(NUSERC)
```

The basic structure of the `sduperr()`, `sduverr()` and `sduaerr()` routines is as follows:

1. obtain any needed information — `SDPOS`, `SDVEL`, `SDACC`,...
2. calculate and return constraint errors

The basic structure of the `sduconsfrc()` routine is as follows:

1. obtain any needed information — `SDPOS`, `SDVEL`,...
2. apply hinge torques, point forces, body torques — `SDHINGET`,`SDPOINTF`, `SDBODYT`

To write a user constraint, first decide whether your constraint is holonomic (a position constraint) or non-holonomic (a velocity constraint). For holonomic constraints, you will need to write constraint error equations for positions, velocities, and accelerations. For non-holonomic constraints, you can set all the position errors to zero and write equations only for velocity and acceleration errors. Acceleration error functions must always be the time derivatives of their corresponding velocity error functions. For holonomic constraints, velocity error functions must also be the time derivatives of their corresponding position error functions.

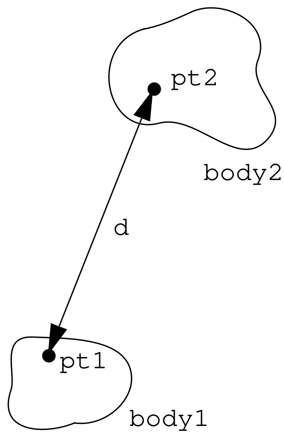
You can have both holonomic and non-holonomic constraints in the same problem. For holonomic constraints, write your position error function in `sduperr()` first, then differentiate each error function twice to produce your velocity and acceleration error functions (in `sduverr()` and `sduaerr()`, resp.). For non-holonomic constraints, set the position errors in `sduperr()` to 0. Write the velocity constraint error function and put it in `sduverr()`. Then differentiate once to produce the acceleration error function for `sduaerr()`.

For the force-application routine `sduconsfrc()`, the meaning of the passed-in multipliers can be obtained by examining your `sduverr()` function. In general, a valid velocity constraint error function can be written as a linear function of point velocities, body angular velocities, and hinge axis velocities (*u*'s). That is, the point velocities and body angular velocities will each appear dot-multiplied by a vector, and the hinge velocities will appear multiplied by a scalar. Application of suitable vector identities may be required to get the error function into this form. Once in this form, the `sduverr()` equations can be used directly to construct the `sduconsfrc()` routine, as follows.

For a particular velocity error function, call the associated multiplier passed-in to `sduconsfrc()` *m*. Now consider a term of the velocity error function involving the velocity of a point multiplied by some vector **v**. For each such term, a call to `SDPOINTF()` should be made in `sduconsfrc()` specifying that point and using *m****v** as the force. For each term consisting of the angular velocity of a body multiplied by some vector **w**, a call to `SDBODYT()` should be made specifying that body and using *m****w** as the torque. For each term consisting of a hinge velocity multiplied by some scalar *s*, call `SDHINGET()` specifying that same hinge and using *m***s* as the torque (or force, in case of a slider). Be sure to take signs into account when producing these forces. In most cases, these applied forces represent significant physical quantities such as the meshing force of gears or the tension in a rod used to enforce a distance constraint.

The following example may help to clarify the above. Further examples are available as Application Notes. If you are still not sure how to proceed, you may wish to contact Symbolic Dynamics for support.

Example



A Distance Constraint

The following example shows a distance constraint. The point `pt1` on `body1` is constrained to remain a distance `d` from point `pt2` on `body2`. `Body1`, `body2`, `pt1`, `pt2`, and `d` are passed in common.

```
      subroutine sduperr(t,q,errs)

c This routine computes the position error (perr) for the
c distance constraint. If d is the desired distance and
c p1 and p2 are the current global frame locations of
c points pt1 and pt2, then

c    perr = ((p1-p2)*(p1-p2) - d**2)/2

      double precision t,q(*),errs(1)
      integer body1,body2,i
      double precision pt1(3),pt2(3),d,p1(3),p2(3),dot
      common /const/ body1,body2,pt1,pt2,d

      call SDPOS(body1,pt1,p1)
      call SDPOS(body2,pt2,p2)
      dot = 0d0
      do 10 i=1,3
10        dot = dot + (p1(i)-p2(i))**2
      errs(1) = 0.5d0*(dot-d**2)

      return
      end

      subroutine sduverr(t,q,u,errs)

c This routine computes the velocity error (verr) for the distance
c constraint, which must be the derivative of the position
c error function above.  If v1 and v2 are  the current global
c frame velocities of points pt1 and pt2, then

c    verr = (v1-v2)*(p1-p2)

      double precision t,q(*),u(*),errs(1)
      integer body1,body2,i
      double precision pt1(3),pt2(3),d,p1(3),p2(3),v1(3),v2(3)
      common /const/ body1,body2,pt1,pt2,d

      call SDPOS(body1,pt1,p1)
      call SDPOS(body2,pt2,p2)
      call SDVEL(body1,pt1,v1)
      call SDVEL(body2,pt2,v2)
      errs(1) = 0d0
      do 10 i=1,3
10       errs(1) = errs(1) + (v1(i)-v2(i))*(p1(i)-p2(i))

       return
       end
```

```
      subroutine sduaerr(t,q,u,udot,errs)

c This routine computes the acceleration error (aerr) for the
c distance constraint, which must be the derivative of the velocity
c error function above. If a1 and a2 are the current global frame
c accelerations of points pt1 and pt2, then

c    aerr = (a1-a2)*(p1-p2) + (v1-v2)*(v1-v2)

      double precision  t,q(*),u(*),udot(*),errs(1)
      integer body1,body2,i
      double precision  pt1(3),pt2(3),d,p1(3),p2(3)
      double precision  v1(3),v2(3),a1(3),a2(3)
      common /const/ body1,body2,pt1,pt2,d

      call SDPOS(body1,pt1,p1)
      call SDPOS(body2,pt2,p2)
      call SDVEL(body1,pt1,v1)
      call SDVEL(body2,pt2,v2)
      call SDACC(body1,pt1,a1)
      call SDACC(body2,pt2,a2)
      errs(1) = 0d0
      do 10 i=1,3
10       errs(1) = errs(1) + (a1(i)-a2(i))*(p1(i)-p2(i))
     1             + (v1(i)-v2(i))**2

      return
      end

      subroutine sduconsfrc(t,q,u,mults)

c This routine applies appropriate forces to the two points to cause
c the distance between them to remain as desired.  By inspection of
c the verr function above, we see the terms involving velocities are:
c          v1*(p1-p2)       and   -v2*(p1-p2)
c Thus, if we call the passed-in multiplier m, the forces to be
c applied are:
c          f1 = m*(p1-p2)   and    f2 = -m*(p1-p2) = -f1
c f1 is to be applied to pt1 and f2 is to be applied to pt2. Note that
c these forces are in the global frame and must be converted into the
c appropriate body-local frame before being applied.

      double precision  t,q(*),u(*),mults(1)
      integer body1,body2,i
      double precision pt1(3),pt2(3),d,p1(3),p2(3),frc(3)
      common /const/ body1,body2,pt1,pt2,d

      call SDPOS(body1,pt1,p1)
      call SDPOS(body2,pt2,p2)
      do 10 i=1,3
10       frc(i) = mults(1)*(p1(i)-p2(i))
      call SDTRANS(0,frc,body1,frc)
      call SDPOINTF(body1,pt1,frc)
      do 20 i=1,3
20       frc(i) = -frc(i)
      call SDTRANS(body1,frc,body2,frc)
      call SDPOINTF(body2,pt2,frc)

      return
      end
```

C language

```
sduconsfrc(t,q,u,m)
double t,q[NQ],u[NU],m[NUSERC];

sduaerr(t,q,u,udot,errs)
double t,q[NQ],u[NU],udot[NU],errs[NUSERC];

sduverr(t,q,u,errs)
double t,q[NQ],u[NU],errs[NUSERC];

sduperr(t,q,errs)
double t,q[NQ],errs[NUSERC];
```

R25 Vector Library Routines

In the SD/FAST Library File is a set of routines for manipulating 3-element vectors. These routines are not problem-specific at all; they are just generic vector manipulation routines which may come in handy while using SD/FAST. There is no requirement to use these routines ever if you prefer to work some other way.

In the description below "`ivec`" indicates an input vector and "`ovec`" an output vector. For any of these routines, the output vector can be the same Fortran or C array as one of the input vectors with no ill effects.

Vector Library Routines

Routine	Description
`subroutine SDVROT(ivec,rvec,theta,ovec)` `double precision ivec(3),rvec(3),theta,ovec(3)`	Rotate a vector `ivec` around a vector `rvec` by an angle `theta`, and put the resulting vector in `ovec`.
`function SDVDOT(vec1,vec2)` `double precision sdvdot,vec1(3),vec2(3)`	Compute the dot product `vec1*vec2`.
`function SDVNORM(vec)` `double precision sdvnorm,vec(3)`	Compute the norm (length) of vector `vec`.
`subroutine SDVCOPY(ivec,ovec)` `double precision ivec(3),ovec(3)`	Vector copy. Set `ovec` = `ivec`.
`subroutine SDVSET(s1,s2,s3,ovec)` `double precision s1,s2,s3,ovec(3)`	Set `ovec` = [`s1`,`s2`,`s3`].
`subroutine SDVADD(vec1,vec2,sum)` `double precision vec1(3),vec2(3),sum(3)`	Vector addition. Set `sum` = `vec1` + `vec2`.
`subroutine SDVSUB(vec1,vec2,diff)` `double precision vec1(3),vec2(3),diff(3)`	Vector subtraction. Set `diff` = `vec1` – `vec2`.
`subroutine SDVMUL(sclr,ivec,prod)` `double precision sclr,ivec(3),prod(3)`	Scalar multiply. Set `prod` = `sclr` * `ivec`.
`subroutine SDVAXPY(sclr,ivec1,ivec2,ovec)` `double precision sclr,ivec1(3),ivec2(3),ovec(3)`	Scalar multiply and vector add "$a\mathbf{x}+\mathbf{y}$". Set `ovec` = `sclr` * `ivec1` + `ivec2`.
`subroutine SDVCROSS(vec1,vec2,ovec)` `double precision vec1(3),vec2(3),ovec(3)`	Cross product. Set `ovec` = `ivec1` × `ivec2`.

C language

```
sdvrot(ivec,rvec,theta,ovec)
double ivec[3],rvec[3],theta,ovec[3];

double sdvdot(vec1,vec2)
double vec1[3],vec2[3];

double sdvnorm(vec)
double vec[3];

sdvcopy(ivec,ovec)
double ivec[3],ovec[3];

sdvset(s1,s2,s3,ovec)
double s1,s2,s3,ovec[3];

sdvadd(vec1,vec2,sum)
double vec1[3],vec2[3],sum[3];

sdvsub(vec1,vec2,diff)
double vec1[3],vec2[3],diff[3];

sdvmul(sclr,ivec,prod)
double sclr,ivec[3],prod[3];

sdvaxpy(sclr,ivec1,ivec2,ovec)
double sclr,ivec1[3],ivec2[3],ovec[3];

sdvcross(vec1,vec2,ovec)
double vec1[3],vec2[3],ovec[3];
```

SD/FAST Quick Reference Guide

This is a very condensed reference guide intended for knowledgeable users. Please read the referenced sections for details and clarification.

You may wish to photocopy the next four pages in this section and place them near your computer for convenient reference.

Q1 Description File (See Section R19 on page R-107)

```
# Any text following a '#' is considered a comment
```

Preamble (optional)

```
[ gravity = <vector> ]
[ language = { fortran|c } ]
[ prefix = <string> ]
[ single | double ]
```

Body (1 or more)

```
body = <name>  inboard = { $ground|<name> }  joint = <jointtype>
  mass = <real>
  inertia = { <real><real><real>  |<real><real><real>
                                   <real><real><real>
                                   <real><real><real> }
  bodyToJoint = <vector>
  inbToJoint  = <vector>
 [ pin = <vector> ]*
 [ prescribed [ = { 1[?]|0[?]|? }* ] ]
```

Loop Joint (0 or more)

```
body = <name>  inboard = { $ground|<name> }  joint = <jointtype>
  inbToJoint = <vector>   bodyToJoint = <vector>
 [ pin = <vector> ]*
 [ bodypin = <vector> ]
 [ inbref = <vector> ]
 [ bodyref = <vector> ]
 [ prescribed [ = { 1[?]|0[?]|? }* ] ]
```

User Constraint (0 or more)

```
constraint = <name> | constraints = <integer>
```

Notes: [ ] = optional { } = required * = repeat | = choice

<*real*> = [+|−] [<*digits*>] [. [<*digits*>]] [e [+|−] <*digits*>] [?] ? = parameter

<*jointtype*> = weld, pin, ujoint, gimbal, ball, slider, cylinder, [r]planar, [r]bearing, [r]bushing, [r]free

<*vector*> = <*real*> <*real*> <*real*>

<*name*> = user-chosen sequence up to 32 letters, digits, and _. Must start with a letter.

<*string*> = user-chosen sequence of characters legal to user's compiler.

$ground = predefined name for inertial frame.

Q2 Command Line Options (See Section R7.1 on page R-30)

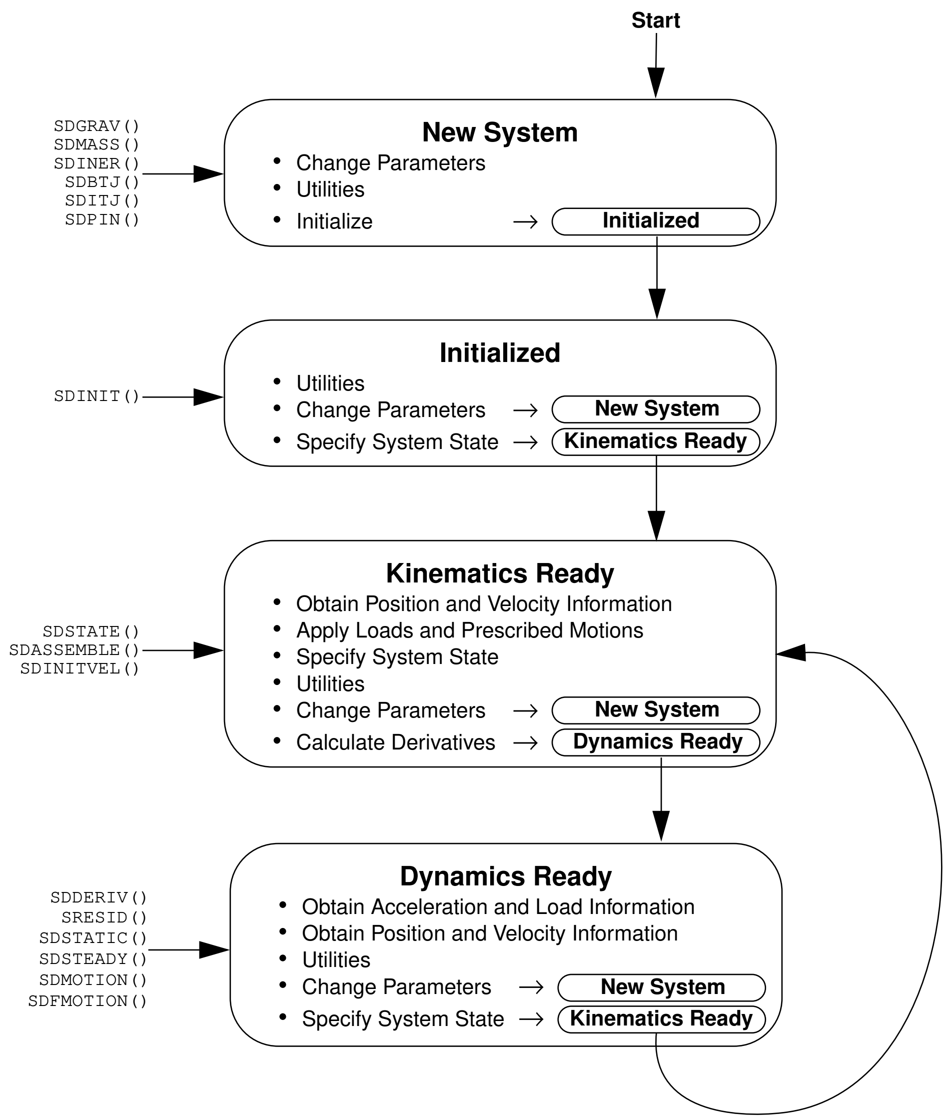
```
sdfast [-snbv] [-l <language>] [-p <prefix>] [-g dsile] [<infile> [<basename>]]
```

- −s single precision
- −n use Order(N) formulation
- −b break up Dynamics file into several smaller pieces
- −v verbose (−vv very verbose, also echoes <*infile*>)
- −l <*language*> = {fortran | c}
- −p <*prefix*> = user-chosen sequence of characters legal to user's compiler.
- −g generate = { [d] [s] [i] [l] | e } (dyn, sar, info, lib) or (everything)
- <*infile*> = System Description File name (legal file name)
- <*basename*> = Base for output file names (any text string part of a legal file name)

Notes: [ ] = optional { } = required | = choice

Q3 SD/FAST Computational Stages (See Section R3.1 on page R-12)

This diagram below shows the ordering dependencies among the SD/FAST-generated routines. The individual routines which cause entry into each of the four computational stages are shown to the left of each stage box. Only routines in certain classes can be called in a given stage. These classes are shown inside each box. Routines in certain classes cause a change to a different stage; these are also shown inside the boxes. The typical computational flow in a simulation is shown by arrows. For a list of the members of each class, see Section Q4 following. Note that routines in the Utilities class can be called any time, resulting in no change of stage. Routines in the Change Parameters class can also be called any time, but always leave you in the **New System** stage in the diagram.



Q4 SD/FAST Routines

dimensions

NC	total # of constraints
NEQ	# state eqns (= *NQ*+*NU*)
NFUNC	# functions to zero
NJNT	total # tree & loop jts
NLQ	# loop jnt pos coords lq
NLU	# loop jnt rate coords lu
NQ	# tree jnt pos coords q
NU	# tree jnt rate coords u
NUS	user-specified length
NUSERC	# of user constraints
NVAR	# variables in functions

These are most user-callable routines generated by SD/FAST, grouped into eight classes, followed by user-written routines. Parameters are documented as shown in the sample below. Unless explicitly indicated, parameters are input only, type real (single or double as specified when generated), dimension one (scalar). A brief description and Reference Section page number is given for each routine.

routine name — input parameters — *i*=integer type — ↑=output parameters

SDSAMPLE (F$^{()}$, RIN, INTIN^i, ↕RINOUT$_{N,3}$, ↑ROUT, ↑INTOUT$^i_3$)

()=callable function parameter — ↕=input/output parameter — dimensions

Class	Routine	Description
Change Parameters	SDBTJ (JOINT^i, BODYTOJOINT$_3$)	Set loc. of outboard hinge point, R-93
	SDGRAV (GRAV$_3$)	Set gravity, R-9
	SDINER (BODY^i, INERTIA$_{3,3}$)	Set inertia matrix for a body, R-93
	SDITJ (JOINT^i, INBTOJOINT$_3$)	Set loc. of inboard hinge point, R-93
	SDMASS (BODY^i, MASS)	Set mass for a body, R-93
	SDPIN (JOINT^i, PINNO^i, PIN$_3$)	Set joint axis orientation, R-93
Initialize	SDINIT ()	Init. at start or after param. change, R-13
Specify System State	SDSTATE (T, Q$_{NQ}$, U$_{NU}$)	Specify time and system state, R-13
	SDASSEMBLE (T, ↕STATE$_{NEQ}$, LOCK$^i_{NU}$, TOL, MAXEVALS^i, ↑FCNT^i, ↑ERR^i)	Adj. q's to satisfy position constraints, R-99
	SDINITVEL (T, ↕STATE$_{NEQ}$, LOCK$^i_{NU}$, TOL, MAXEVALS^i, ↑FCNT^i, ↑ERR^i)	Adj. u's to satisfy velocity constraints, R-100
Calculate Derivatives	SDDERIV (↑QDOT$_{NQ}$, ↑UDOT$_{NU}$)	Calculate state derivatives, R-15
	SDRESID (QDOT$_{NQ}$, UDOT$_{NU}$, MULT$_{NC}$, ↑RESID$_{NQ+NU+NC}$)	Calc. err. in DAE est. of derivs & mults, R-15
	SDSTATIC (T, ↕STATE$_{NEQ}$, LOCK$^i_{NU}$, CTOL, TOL, MAXEVALS^i, ↑FCNT^i, ↑ERR^i)	Adj. q's to find static configuration, R-101
	SDSTEADY (T, ↕STATE$_{NEQ}$, LOCK$^i_{2*NU}$, CTOL, TOL, MAXEVALS^i, ↑FCNT^i, ↑ERR^i)	Adj. q's and u's to find steady config., R-102
	SDMOTION (↕T, ↕STATE$_{NEQ}$, ↕DSTATE$_{NEQ}$, DT, CTOL, TOL, ↕FLAG^i, ↑ERR^i)	Integrate state from t to t+dt, R-104
	SDFMOTION (↕T, ↕STATE$_{NEQ}$, ↕DSTATE$_{NEQ}$, DT, CTOL, ↕FLAG^i, ↑ERREST, ↑ERR^i)	Integ. state from t to t+dt w/ fixed step, R-104
Apply Loads and Prescribed Motions	SDBODYT (BODY^i, TORQUE$_3$)	Apply body-fixed torque, R-10
	SDHINGET (JOINT^i, AXIS^i, TORQUE)	Apply joint axis load (force or torque), R-10
	SDPOINTF (BODY^i, POINT$_3$, FORCE$_3$)	Apply body-fixed force to pt. on body, R-10
	SDPRESACC (JOINT^i, AXIS^i, ACCEL)	Prescribe joint axis acceleration, R-77
	SDPRESVEL (JOINT^i, AXIS^i, VEL)	Prescribe joint axis velocity, R-77
	SDPRESPOS (JOINT^i, AXIS^i, POS)	Prescribe joint axis position, R-77
	SDPSSTATE (LQ$_{NLQ}$)	Repl. pres. lq's with integrated values, R-114
Obtain Position and Velocity Information	SDANGVEL (BODY^i, ↑ANGVEL$_3$)	Inertial angular velocity, in body frame, R-70
	SDMOM (↑LM$_3$, ↑AM$_3$, ↑KE)	Inertial linear & ang. mom., kin. energy, R-71
	SDORIENT (BODY^i, ↑DIRCOS$_{3,3}$)	Body orientation with respect to ground, R-70
	SDPERR (↑PERRS$_{NC}$)	Position constraint errors, R-18
	SDPOS (BODY^i, POINT$_3$, ↑LOC$_3$)	Inertial position of any point on body, R-70
	SDPSEUDO (↑LQ$_{NLQ}$, ↑LU$_{NLU}$)	Return pseudo-state variables, R-114
	SDPSQDOT (↑LQDOT$_{NLQ}$)	Return pseudo-q derivatives, R-114
	SDTRANS (FRBODY^i, FVEC$_3$, TOBODY^i, ↑TVEC$_3$)	Xform vector from one frame to another, R-72
	SDSYS (↑MTOT, ↑CM$_3$, ↑ICM$_{3,3}$)	System mass, ctr. of mass, inertia mat., R-71
	SDVEL (BODY^i, POINT$_3$, ↑VEL$_3$)	Inertial velocity of any point on body, R-70
	SDVERR (↑VERRS$_{NC}$)	Velocity constraint errors, R-18

Obtain Acceleration and Load Information

Routine	Arguments	Description
SDACC	(BODY^i,POINT$_3$,↑ACC$_3$)	Inertial accel. of any point on body, R-70
SDAERR	(↑AERRS$_{NC}$)	Acceleration constraint errors, R-18
SDANGACC	(BODY^i,↑ANGACC$_3$)	Inertial angular accel. in body frame, R-70
SDGETHT	(JOINT^i,AXIS^i,↑HINGET)	Get applied or computed hinge load, R-46
SDMULT	(↑MULTS$_{NC}$,↑RANK^i,↑MULTMAP$^i_{NC}$)	Constraint multiplier information, R-22
SDPSUDOT	(↑LUDOT$_{NLU}$)	Return pseudo-u derivatives, R-114
SDREAC	(↑FORCES$_{NJNT,3}$,↑TORQUES$_{NJNT,3}$)	Outb. body reaction loads, body frame, R-45

Utility Routines

Routine	Arguments	Description
SDANG2DC	(A1,A2,A3,↑DIRCOS$_{3,3}$)	Cnvt. 1-2-3 Euler angles to dir. cos., R-72
SDANG2ST	(STANG$_{NU}$,↑ST$_{NQ}$)	Cnvt. Euler angles in state to quats, R-27
SDCLEARERR	()	Clear posted usage error (if any), R-119
SDCONS	(CONSNO^i,↑INFO$^i_{50}$)	Get info. about a constraint, R-95
SDDC2ANG	(DIRCOS$_{3,3}$,↑A1,↑A2,↑A3)	Cnvt. dir. cos. to 1-2-3 Euler angles, R-72
SDDC2QUAT	(DIRCOS$_{3,3}$,↑E1,↑E2,↑E3,↑E4)	Cnvt. direction cosine to quaternions, R-72
SDERROR	(↑ROUTINE^i,↑ERRNO^i)	Return posted usage error no. (if any), R-119
SDFINTEG		Fixed-step RK4 integrator, R-37
	(F$^{()}$,↕T,↕Y$_{NEQ}$,↕DY$_{NEQ}$,↕PARAM$_{NUS}$,DT,NEQ^i,WORK$_{4*NEQ}$,↑ERREST,↑FSTATUS^i)	
	F(T,Y$_{NEQ}$,↑DY$_{NEQ}$,↕PARAM$_{NUS}$,↑STATUS^i)	
SDGETBTJ	(JOINT^i,↑BODYTOJT$_3$)	Get loc. of outboard hinge point, R-94
SDGETGRAV	(↑GRAV$_3$)	Get current value of gravity vector, R-9
SDGETINER	(BODY^i, ↑INERTIA$_{3,3}$)	Get a body's inertia matrix, R-94
SDGETITJ	(JOINT^i,↑INBTOJT$_3$)	Get loc. of inboard hinge point, R-94
SDGETMASS	(BODY^i, ↑MASS)	Get a body's mass, R-94
SDGETPIN	(JOINT^i,PINNO^i,↑PIN$_3$)	Get joint axis orientation, R-94
SDGETPRES	(JOINT^i,AXIS^i,↑PRES^i)	Find out if prescribed motion on or off, R-77
SDGETSTAB	(↑VEL,↑POS)	Get current Baumgarte constants, R-19
↑SDINDX^i	(JOINT^i,AXIS^i)	Function returning q or u indx of jt axis, R-116
SDINFO	(↑INFO$^i_{50}$)	Get general info. about system, R-95
SDJNT	(JOINT^i,↑INFO$^i_{50}$,↑SLIDER$^i_6$)	Get information about a joint, R-95
SDLSSLV		Linear equations least-squares solver, R-40
	(NR^i,NC^i,NRA^i,NCA^i,NDES^i,MAPR$^i_{NRA}$,MAPC$^i_{NCA}$,TOL, DW$_{2*(NRA+NCA)**2}$,RW$_{4*(NRA+NCA)}$,IW$^i_{3*(NRA+NCA)}$,W$_{NR,NC}$,B$_{NR}$,↑X$_{NC}$)	
SDNORMST	(ST$_{NQ}$,↑NORMST$_{NQ}$)	Normalize quaternions in state, R-28
SDPRES	(JOINT^i,AXIS^i,PRES^i)	Turn prescribed motion on or off, R-77
SDPRINTERR	(FILE^i)	Print posted usage error (if any) to file, R-119
SDQUAT2DC	(E1,E2,E3,E4,↑DIRCOS$_{3,3}$)	Cnvt. quaternions to direction cosine, R-72
SDROOT		Nonlinear equations solver, R-39
	(F$^{()}$,↕VARS$_{NVAR}$,↕PARAM$_{NUS}$,NFUNC^i,NVAR^i,NDES^i,LOCK$^i_{NVAR}$,RTOL,DTOL,MAXEVAL^i, JW$_{NFUNC*NVAR}$,DW$_{2*(NFUNC+NVAR)**2}$,RW$_{9*(NFUNC+NVAR)}$,IW$^i_{4*(NFUNC+NVAR)}$, ↑FRET$_{NFUNC}$,↑FCNT^i,↑ERR^i)	
	F(VARS$_{NVAR}$,↕PARAM$_{NUS}$,↑RESID$_{NFUNC}$)	
SDST2ANG	(ST$_{NQ}$,↑STANG$_{NU}$)	Cnvt. quats in state to Euler angles, R-27
SDSTAB	(VEL,POS)	Set Baumgarte constants, R-19
SDVINTEG		Variable-step RK4 integrator, R-35
	(F$^{()}$,↕T,↕Y$_{NEQ}$,↕DY$_{NEQ}$,↕PARAM$_{NUS}$,DT,↕STEP,NEQ^i,TOL,WORK$_{6*NEQ}$,↑ERR^i,↑WHICH^i)	
	F(T,Y$_{NEQ}$,↑DY$_{NEQ}$,↕PARAM$_{NUS}$,↑STATUS^i)	

User-Written Routines

Routine	Arguments	Description
sduaerr	(T,Q$_{NQ}$,U$_{NU}$,UDOT$_{NU}$,↑ERRS$_{NUSERC}$)	Calc. user accel. constraint errors, R-128
sduconsfrc	(T,Q$_{NQ}$,U$_{NU}$,MULT$_{NUSERC}$)	Apply user constraint reaction loads, R-128
sduderiv		Calculate derivatives for integrator, R-85
	(T,Y$_{NEQ}$,↑DY$_{NEQ}$,↕PARAM$_{NUS}$,↑STATUS^i)	
sdueval		Evaluate residual error for root finder, R-87
	(VARS$_{NVAR}$,↕PARAM$_{NUS}$,↑RESID$_{NFUNC}$)	
sduforce	(T,Q$_{NQ}$,U$_{NU}$)	Apply forces and torques, R-83
sdumotion	(T,Q$_{NQ}$,U$_{NU}$)	Apply prescribed motions, R-84
sduperr	(T,Q$_{NQ}$,↑ERRS$_{NUSERC}$)	Calc. user position constraint errors, R-128
sduverr	(T,Q$_{NQ}$,U$_{NU}$,↑ERRS$_{NUSERC}$)	Calc. user velocity constraint errors, R-128

Index

L

M

N