Simbody™

User's Manual

Early Access Release 0.7

August 2007

Website: SimTK.org/home/simbody

SimTK Simbody™ 0.7

Early Access User's Guide

Michael Sherman

DRAFT Version 12, December 10, 2007

Abstract

We discuss the SimTK toolset for multibody dynamics (a.k.a. internal/torsion/relative coordinate modeling, rigid body mechanics). This toolset, called "SimTK Simbody," is intended to be useful in coarse grain molecule modeling, neuromuscular gait simulation, and many other biologically relevant models. SimTK Simbody is structured as a library, not a standalone application, and is intended to work easily from programs written in a variety of languages and styles. We provide some basic definitions and background, and then present specifics of the current pre-release version of Simbody (0.7) and where it is headed for version 1.0. [This document is a work in progress.]

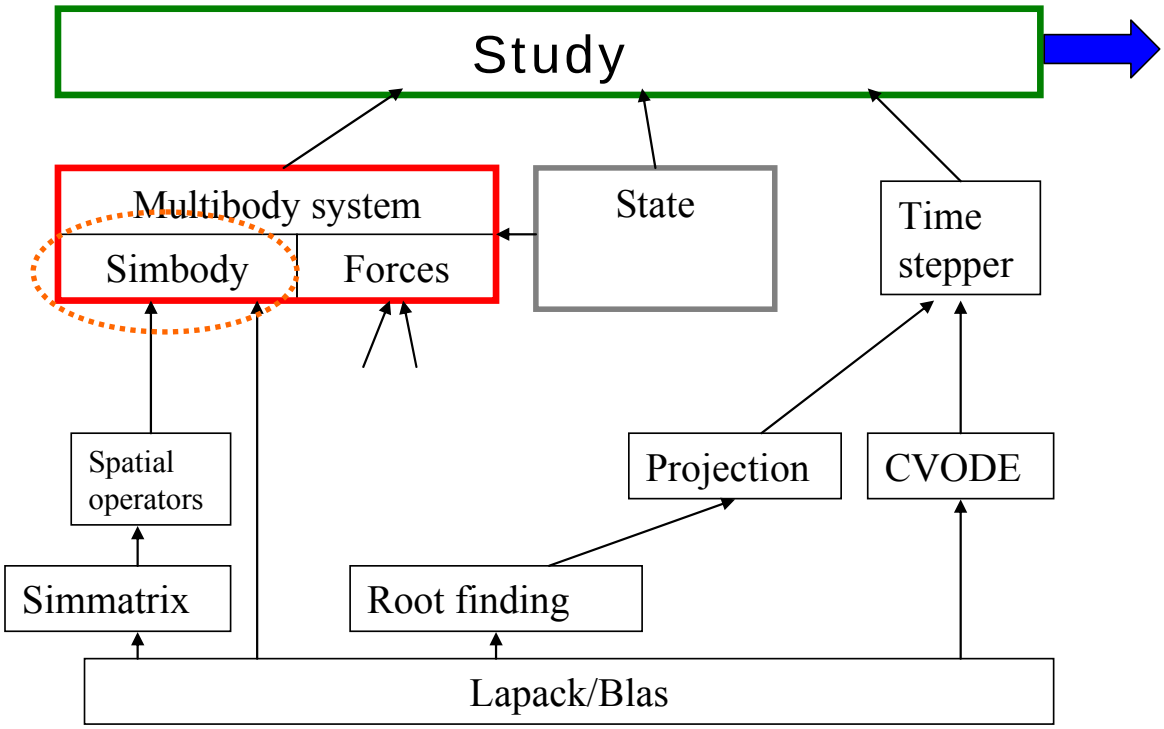
1 Preface

SimTK Simbody provides a powerful multibody mechanics capability for use in biosimulation. It is designed for use by programmers who are not experts in multibody mechanics. Simbody provides a sophisticated, robust, high performance, open source option for mechanical simulation, as well as the additional functionality and performance needed for effective modeling of large molecular systems in internal coordinates. It is accessible through a stable API* to programmers who work in a variety of languages. The early access 0.7 API is object-oriented C++. C and FORTRAN APIs and wrappers for interpretive languages like Java and Python are planned. The full capability of this package will be built up in layers over time; this document covers the "early access" capabilities and discusses future directions.

A complete multibody mechanics simulation (a molecular dynamics simulation of a protein/RNA interaction, for example) requires many layers. At the lowest level are basic numerical methods like linear algebra, numerical integration, nonlinear root finding, and optimization. Next up is the multibody mechanics computation, which is where Simbody fits in. Alongside that are domain-specific force computations. Above that is a modeling layer for use in constructing these systems, and above that a user interface that

* API: "Application Programming Interface," i.e., a programming library.

provides model building, execution, and visualization of results. The figure below shows Simbody's small but significant place in the SimTK framework.



At this pre-release stage, Simbody's API is not yet stable, and most users would find it difficult to use. Simbody 1.0 will provide a more palatable and stable API.

Document conventions

In order to allow myself the pleasure of delivering the occasional opinionated diatribe, while permitting the easily offended reader to avoid them, I have placed a "pontification warning" symbol like the one at the left at the beginning of such sections in the text. The end of these sections is marked with the "off my soapbox" symbol to the right.

The symbol to the left is used to highlight sections which summarize earlier material.

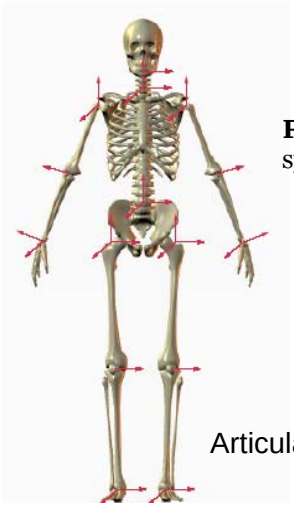
This one is used to mark discussions of capabilities which are planned but not yet implemented.

2 Background

This is general material hopefully providing enough background for the rest of the document to make sense. Even for those familiar with multibody dynamics, it is probably worth reading to see how we are characterizing it for the broad uses it will serve for SimTK users.

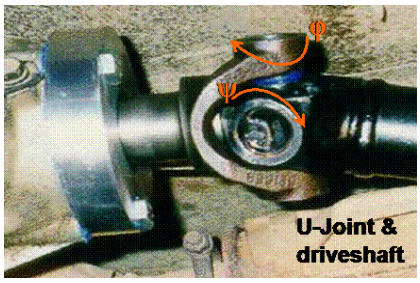
2.1 What is "multibody dynamics"?

Multibody mechanics (of which multibody dynamics is a subset) is the field studying the classical mechanical properties, especially motion, of systems of *bodies* interconnected by *joints*, influenced by *forces*, and restricted by *constraints*. The key feature of a system that makes it suitable for multibody treatment is the observation that its motion is *localized*, that is, it is well-described as a set of independently identifiable parts which undergo large motion with respect to one another, but are themselves rigid or nearly rigid. Figure 1 shows some examples of the breadth of applicability of multibody mechanics, which has been used effectively to model machines, skeletal motion and gait, coarse-grained biopolymers, and many other systems relevant to a wide variety of scientific and engineering disciplines.

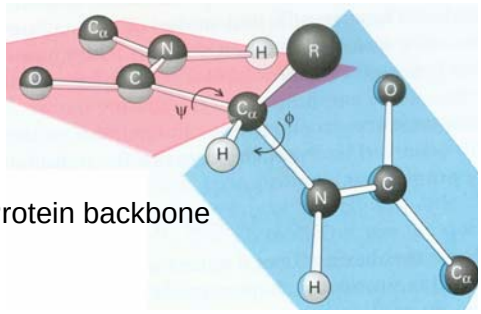


Articulated skeleton

Figure 1: Some multibody systems.



Mechanical U-joint



Protein backbone

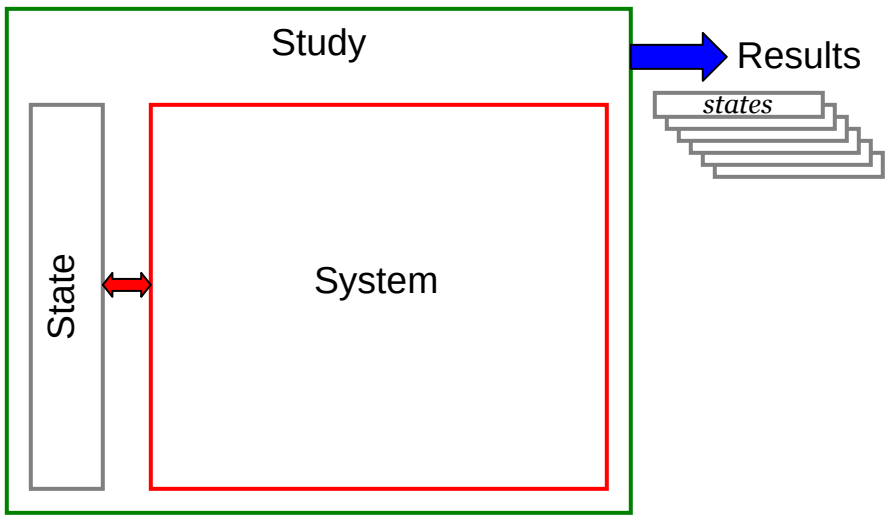
Multibody mechanics is a *generalization* of several more-familiar modeling methods. It includes as special cases, for example, systems of point masses represented in Cartesian coordinates (e.g. molecular dynamics models) and systems of freely moving extended bodies (typically, rigid bodies) and these can be intermixed into systems which also contain bodies whose motion is defined with internal (relative) coordinates, that is, with respect to one another rather than with respect to the Cartesian frame. Multibody mechanics should be viewed as a basic numerical capability fundamental to any simulation system. It is in the same category as, say, a linear algebra library, not an end-user application. Simbody is for use by modelers and application developers as a basic building block. Computational researchers working to improve multibody simulation methods can use Simbody as a baseline source of correct answers for debugging and as a performance baseline to demonstrate the superiority of their new methods.

2.2 Structure of a simulation in SimTK

The figure below shows the primary objects involved in computational simulation of a physical system in SimTK, the infamous "three *S*'s of simulation": *System*, *State*, and *Study*. Here's our first equation:

Simulation(SimTK) = System + State + Study

A *System* is a computational embodiment of a mathematical model of the physical world. A System typically comprises several interacting, separately meaningful subsystems. A System contains models for physical objects and the forces that act on them and specifies a set of variables whose values can affect the System's behavior. However, the System itself is an unchangeable, state-free ("const") object. Instead, the values of its variables are stored in a separate object, called a *State*, more about which below.* Finally, a *Study* couples a System and one or more States, and represents a computational experiment intended to reveal something about the System. By design, the results of *any* Study can be expressed as a State value or set of State values which satisfies some pre-specified criteria, along with results which the System can calculate directly from those State values. Such a set of State values is often called a *trajectory*.



It is important to note that our notion of "state" is somewhat more general than the common use of the term. By state, we mean *everything* variable

* We will frequently use "state" (lower case) to refer to the *values* stored within a State object. This isn't as confusing as it might seem—even if we get the capitalization wrong the meaning will be obvious from context.

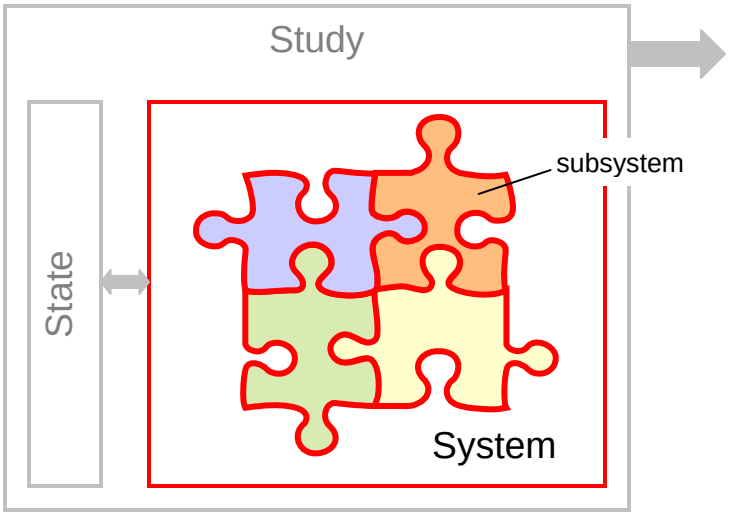
about a System. That includes not only the traditional continuous time, position and velocity variables, but also discrete variables, memory of past events, modeling choices, and a wide variety of parameters that we call *instance variables*. The System's State has entries for the values of all of these variables.

This design allows the conceptually simple model depicted above to express every kind of investigation one may wish to perform. Here are some examples. The simplest Study merely asks the System to evaluate itself using values taken from a particular State. More interestingly, a dynamic Study produces a series of time, position and velocity State values which result from solving the classical dynamic equations representing Newton's 2nd law, F=ma. An energy minimization is a Study which seeks values for the State's position coordinates at which an energy calculation yields its minimum value. A Monte Carlo simulation is a Study yielding a series of states which satisfy an appropriate probability distribution. Design studies, also used for parameter fitting, are Studies which find values for instance variables such as lengths, masses, material properties, or coefficients which meet specified criteria. Modeling Studies select among models or algorithmic choices to improve defined measures of behavior, such as accuracy, stability, or execution speed. And so on. Since we know that *all* System variability is contained in the State, we can guarantee that any answers you seek regarding the System can be expressed in terms of state values, provided that a corresponding System is available to interpret them.

At this stage in Simbody's development, System and State are implemented, but there is very little of the Study layer except in spirit! There are some numerical methods classes provided which are best thought of as prototype Studies, but there will a more substantial Study framework by 1.0.

2.3 Structure of a System

Looking a little closer at a System, you will find that it is composed of a set of interlocking pieces, which we call *subsystems*.



In this jigsaw puzzle analogy, you can think of the System as providing the "edge pieces" which frame the subsystems into a complete whole.

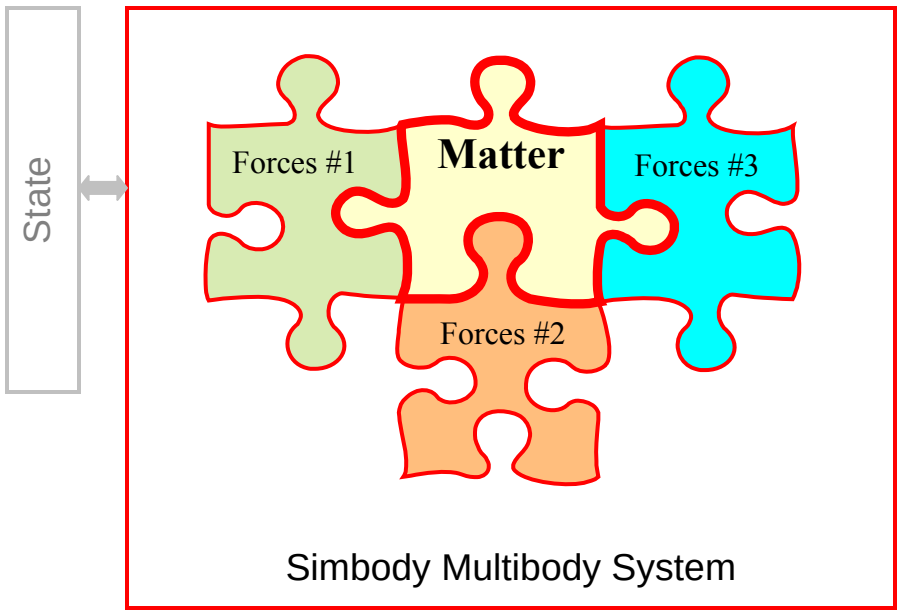
In general any subsystem of a System may have its own state variables, as can the System itself. The System ensures that its subsystems' state needs are provided for within the overall System's State. The calculations performed by subsystems are interdependent in the sense of having interlocking computational dependencies. However, these dependencies can always be untangled by performing computations in "stages" as will be discussed below. It is the System's responsibility to properly sequence its subsystems through the stages.

2.4 Structure of a multibody system

Let's look at Simbody in this context. Simbody provides *one* computational component (one puzzle piece) of a complete multibody mechanics System. Simbody's piece manages the representation of interconnected massive objects (that is, bodies interconnected by joints). Simbody can use this representation to perform computations which permit a wide variety of useful Studies to be performed. For example, given a set of applied forces, Simbody can very efficiently solve a generalized form of Newton's 2nd law F=ma. On the other hand, Simbody is agnostic about the forces F, which come from domain-specific models. That is, Simbody fully understands the concept of *forces*, and knows exactly what to do with them, but hasn't any idea where

they might have come from. Muscle contraction? Molecular electrostatic interactions? Galactic collisions? Whatever.

A complete System thus consists both of the matter subsystem implemented by Simbody, and user-written or application-provided force subsystems. So for a multibody system, the general SimTK System described above is specialized to look something like this:



Although both the Simbody Matter subsystem and the force subsystems require state variables, as discussed above any SimTK System (including of course a Multibody System) is a stateless object once constructed. Its subsystems collectively define the System's parameterization, but the parameter values themselves are stored externally in a separate State object.

The force and mechanical subsystems are computationally interlocked. For example, a user-provided force will typically depend on position and velocity information (kinematics) returned by the Simbody subsystem, while accelerations (dynamics) calculated by Simbody will in turn depend on the values of the forces. Section 2.5 provides details on how these interlocking computations are performed.

2.5 Computation – realization of the State

This section provides some details about how computations are performed in the System-State-Study architecture described in Section 2.2.

During a Study, the System is used to *realize* a State. By *realize* we mean the process of taking a new set of values from a State and performing the computations that those new values enable. A simple example would be to take new position coordinate values from a State and use them to calculate new spatial locations for the bodies, and then distances between designated points on different bodies. Realizing a State enables three kinds of computations: *responses*, *operators*, and *solvers*, defined next.

2.5.1 Responses, operators, and solvers

A *response* is a numerical result which can be computed knowing only the values in the State. The above calculation of distance from position coordinates is an example of a response. An *operator* is a computation which requires knowledge of certain State variables, but then can be applied repeatedly to other input data (i.e., data not from the State) to produce numerical results. For example, once we know positions and velocities from the State, we can realize an operator which, when applied to a set of forces, efficiently calculates the accelerations that would be produced by those forces. Neither responses nor operators make changes to the State. A *solver*, on the other hand, both reads from and writes to the State. A given solver requires certain values from the State, and may make use of those values or responses and operators calculated from them. It then performs a computation which updates the State in some well-defined way. The simplest kind of solver is a method which just sets a particular State variable to a given value. A more elaborate example is a solver which takes current positions from the State and modifies them to find the nearest set of positions which satisfies particular constraints.

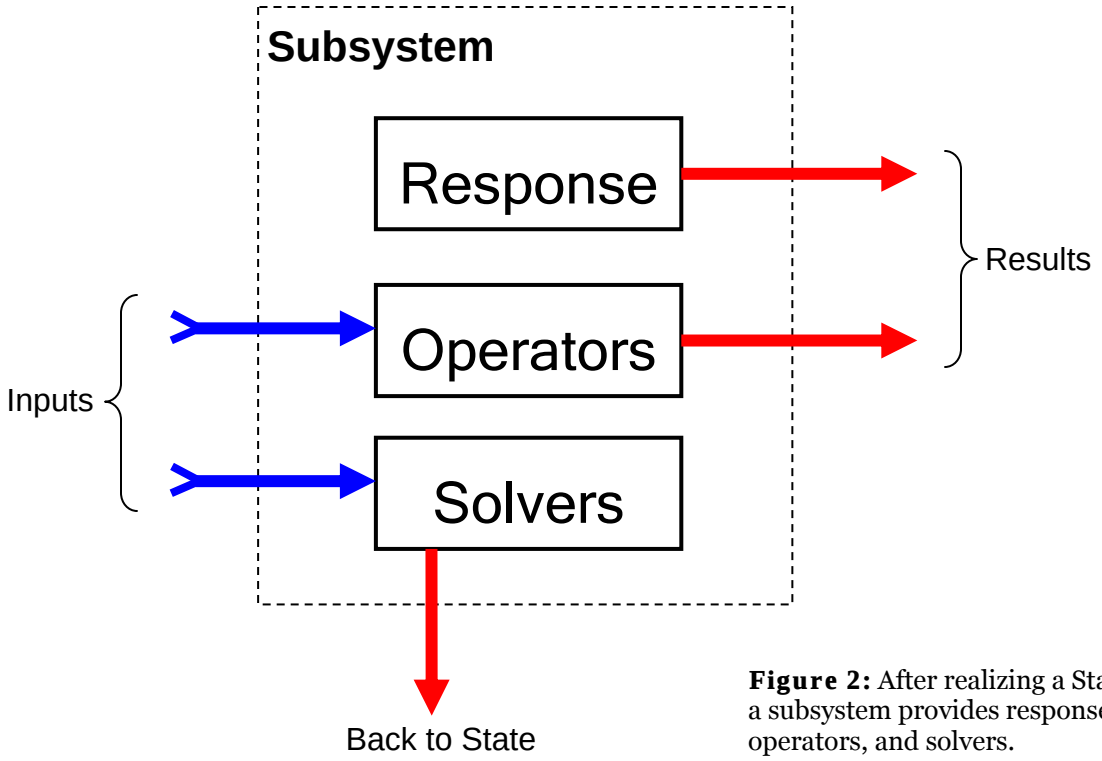


Figure 2: After realizing a State, a subsystem provides responses, operators, and solvers.

2.5.2 Caching of computed results

Realizing a State may require a large amount of expensive computation, and the computed results are typically used many times in calculation of subsequent results. Consequently it is crucial that these computations not be re-done once calculated for a particular set of State values. Given that a System is a read-only object, and that realization results are associated with a particular State, the obvious place to store these results is in the State object. That way when a Study provides a State to a System, all previously-calculated results are available as well, and one may be certain that those results were calculated using the values from the supplied State. This eliminates the possibility of bugs in which values computed at one state are incorrectly used as results at a different state. That is an extremely common error in simulation programs and is very difficult to fix, primarily because it often goes completely unnoticed. Errors of this type are hidden by the fact that sequentially-produced states tend to differ very little, making the computed values only a “little bit” wrong.

To take a brief pontification opportunity, I want to emphasize in the strongest possible terms that "little" bugs in simulation programs do not leave them "nearly" valid the way, say, small measurement errors affect real-world experiments. Simulation software is the most nonlinear thing in existence—one wrong bit in a billion can completely destroy any relevance it might have had to the real world. The resulting simulation results, unfortunately, may still appear plausible, especially where human intuition is of limited use such as with molecular systems. And statistical reduction methods used to calculate physical properties from a simulation (e.g., population distributions, free energies, radii of gyration, transition times, etc.) are almost certain to turn meaningless garbage into "intriguing" results which "should be researched further."

Although cached results are stored in the State object, it is important to note that those results (that is, responses, operators, and solvers) are not *logically* part of the system state. They are simply intermediate calculations which have been derived from the state, and can easily be discarded and re-created when necessary. They are needed only for efficient computation using the System-State-Study architecture, and so can be viewed as "merely" a hint. They exist as a kind of shadow behind the actual state variables, whose values do matter. We call this shadowy construct the *realization cache*, or more often, just the *cache*.

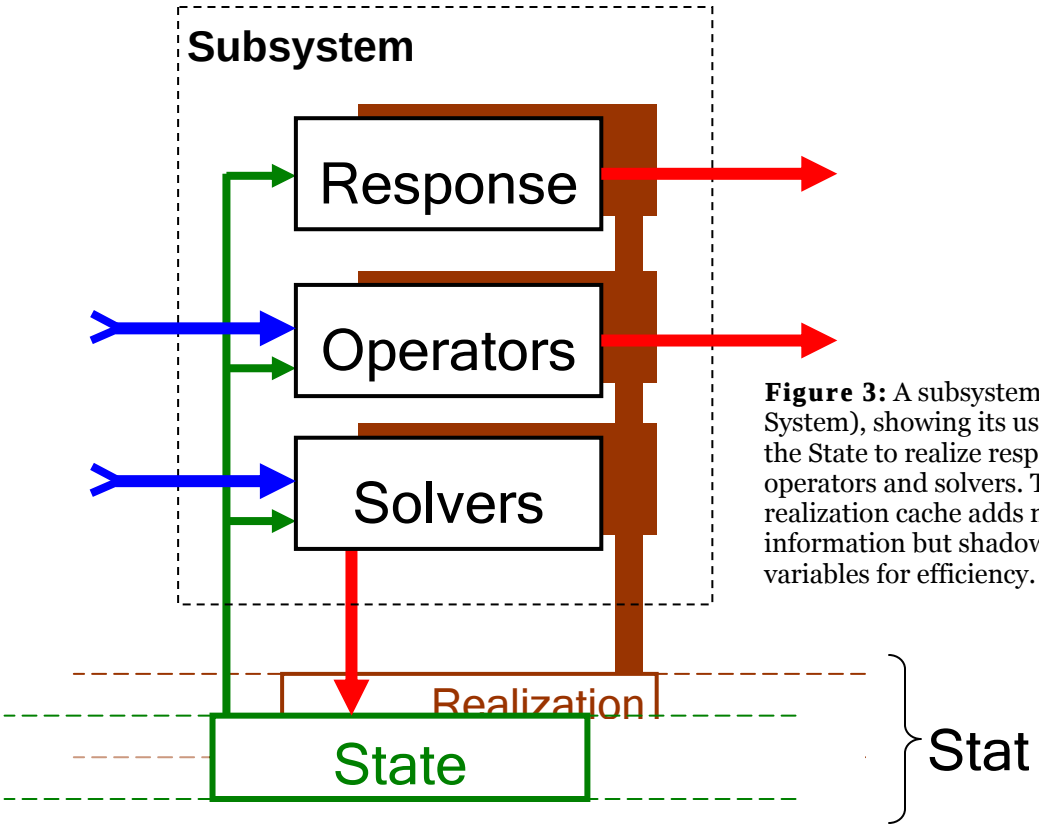


Figure 3: A subsystem (part of a System), showing its use of part of the State to realize responses, operators and solvers. The realization cache adds no new information but shadows the state variables for efficiency.

Figure 3 combines the concepts just described. It shows a subsystem (one of the pieces making up a System) and how its responses, operators, and solvers make use of the realization cache. Note that responses require no input other than the State, while operators and solvers can have additional inputs (the blue arrows in the figure). Operators and solvers then differ by the disposition of their outputs (red arrows), with only solvers' output able to update the State.

To summarize briefly: A System (or subsystem) by itself is stateless once constructed. The values of state variables stored in a particular State object completely determine the behavior of the System. That behavior is produced by realizing the State. The results of realization, which are responses, operators, and solvers, are stored in a hidden cache which is physically contained in the State object, but is not logically part of the state in the sense that cache values are not permitted to alter the behavior of the System, except for the speed with which it can perform computations involving that State.

2.5.3 Computing in stages

State variables are naturally ordered in *stages*, and realization of a State is done one stage at a time, in order. This structure allows interdependencies among the subsystems in the System, without requiring any subsystem to know any internal details of other subsystems. Of specific relevance for Simbody, user-supplied forces depend on values provided by the Simbody multibody subsystem (such as positions and velocities), but Simbody dynamic calculations (e.g., accelerations) likewise depend on the user-supplied forces. Thus complete realization of a State requires sequences like (1) the SimbodyMatterSubsystem realizes its "Position" stage, then (2) each force subsystem independently realizes *its* Position stage to calculate position-dependent forces (repeat for Velocities), and then (3) SimbodyMatterSubsystem realizes its accelerations (reactions) using computations cached by the force subsystems. This staging approach allows a composite System computation to be performed efficiently from isolated subsystems, with each subsystem mediating access to its own state variables and cache.

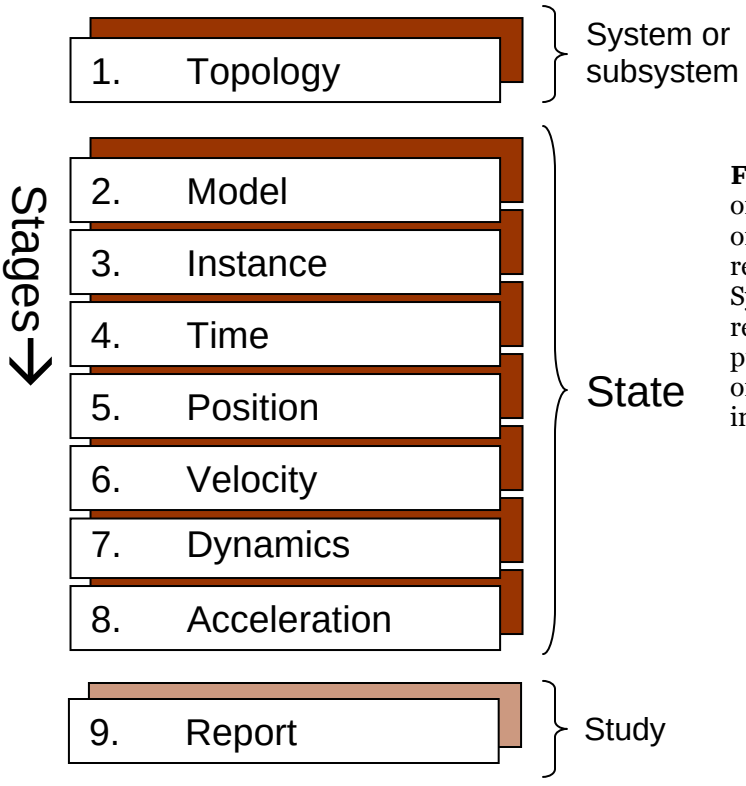


Figure 4: The conceptual organization of a computation into ordered stages. A given stage is fully realized by *each* subsystem in a System before the next stage is realized by *any* subsystem. For many purposes, construction of the (read only) System can be viewed as the initial stage of computation.

With the set of concepts described above, which at this point may seem disconcertingly abstract, we are ready to see a simple concrete example, after which we'll move on to the nuts and bolts of expressing a multibody system for efficient computation in Simbody.

3 A simple example: 2d pendulum

That's enough background now for us to work a simple example, although not everything needed for this example has been presented. We're going to build the system depicted below, a one-body pendulum swinging in a plane passively under gravity. There's not much we can investigate here, but we can at least try to replicate some classical results. In particular, for a pendulum consisting of a point mass m suspended from a massless rod of length d in a gravitational field of strength g, for modest swing amplitudes the period ω is independent of both the amplitude and the mass: $\omega = 2\pi\sqrt{d/g}$. So we'll build a Simbody model of this, run a dynamic study, and measure the period we get at two different amplitudes. As an example, at the Earth's surface if d is half a meter, then ω=1.42s (g=9.8 m/s^2).

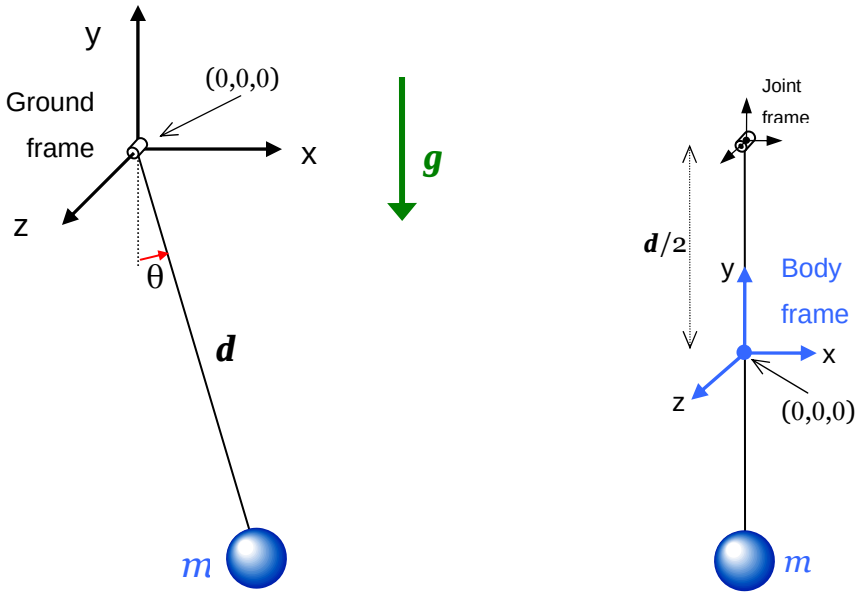


Figure 5: A simple point-mass pendulum pinned to ground at the origin. For small amplitudes, the period ω should be $\omega = 2\pi\sqrt{d/g}$. The parameterization of the pendulum body is shown on the right. We arbitrarily put the body frame in the middle, so the pin joint's frame is located at $(0,+d/2,0)$ and the mass at $(0,-d/2,0)$. A pin joint is defined to align the z axes of the two frames it connects, which in this case are the ground frame and the joint frame drawn at the top of the body.

3.1 The main program

(OBSOLETE – needs to be rewritten.) Below is the `main()` of a short C++ program which defines and simulates the above system. The Simbody piece is hiding in the `MySimbodyPendulum` class which will be shown later.

```
int main(int argc, char** argv) {
    try { // If anything goes wrong, an exception will be thrown.
        XXX
    }
    catch (const exception& e) {
        printf("EXCEPTION THROWN: %s\n", e.what());
        exit(1);
    }
}
```

3.2 Results

We ran the above program twice, with initial angles 10 degrees and 20 degrees, for 5 times the expected period, and then fed the output to Matlab to make a plot, shown below.

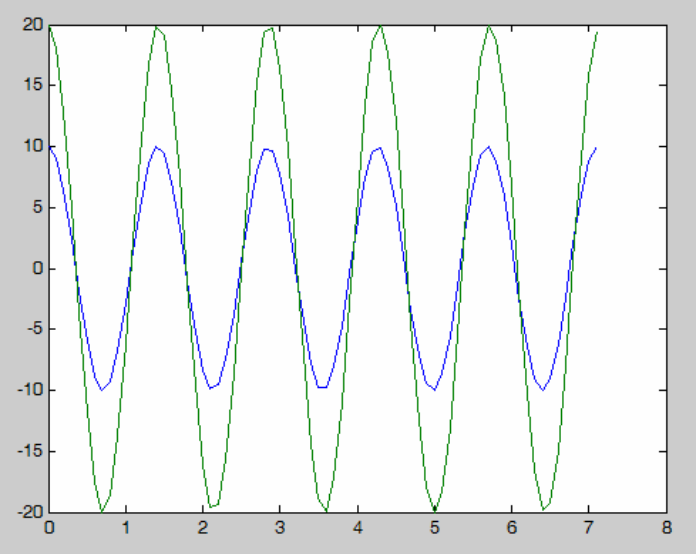


Figure 6: Output from two runs of the pendulum example, starting at 10 (blue) and 20 (green) degrees. The vertical axis is degrees, horizontal is seconds. Period should be 1.42s, independent of amplitude.

It is hard to take a precise reading from this graph, but it certainly *looks* like a period of around 1.4s, and it does seem to be the same for the two amplitudes. You can try this yourself and run at a higher precision with more output points to get as much accuracy as you want.

3.3 Building the Simbody model

`(OBSOLETE - needs to be rewritten.)`

We will now turn to more detail about multibody mechanics, so we can see what goes into a `SimbodyMatterSubsystem`.

4 Fundamental concepts of multibody mechanics

There are only a few general concepts required to completely specify a multibody system. These are closely related to physical concepts for which the reader is likely already to have a good intuition. This is both blessing and curse, since our intuitive understanding of these concepts is almost, but not quite, general enough or precise enough to serve as a basis for general simulation. Nevertheless we will plunge ahead using familiar concepts, adding precise definitions and suitable generalizations where needed.

The concepts we'll need are: coordinate frame, body, mobilizer, constraint, and force.

4.1 Coordinate frames

We define a *coordinate frame* (syn: *reference frame* or just *frame*) F to be a set of three mutually orthogonal directions (called axes) and a point (called the frame's origin). We will denote the axes as unit vectors $\mathbf{x}^F$,$\mathbf{y}^F$,$\mathbf{z}^F$ and follow a right-handed ("dextral") convention so that $\mathbf{z}^F=\mathbf{x}^F\times\mathbf{y}^F$. In our notation, a *right* superscript "F" indicates a physical quantity which is attached to, or fixed in, frame F. We label frame F's origin point $\square^F$.

Coordinate frames are used for measuring things. We can express the location of a point $\mathcal{P}$ in frame F, for example, by constructing the vector $\mathbf{r}=\mathcal{P}-\square^F$ (which points from $\square^F$ to $\mathcal{P}$) and then expressing it in frame F by writing down the components of $\mathbf{r}$ in each of the three axis directions. These numerical values are called the *measure numbers* of $\mathbf{r}$ in F denoted ${}^F\mathbf{r}={}^F[r_x,r_y,r_z] \equiv {}^F[\mathbf{r}\cdot\mathbf{x}^F,\mathbf{r}\cdot\mathbf{y}^F,\mathbf{r}\cdot\mathbf{z}^F]$. That is, the measure numbers are the scalars obtained by taking the dot product of a vector with each of the three axis directions of a frame. We use a *left* superscript to indicate the "measured in" frame (the frame defining the physical quantity of interest), and unless otherwise specified that is also the "expressed in" frame (the frame whose axes are used to decompose the measured quantity into scalars). Note that the same vector will have different measure numbers when expressed in different frames. Here's a picture:

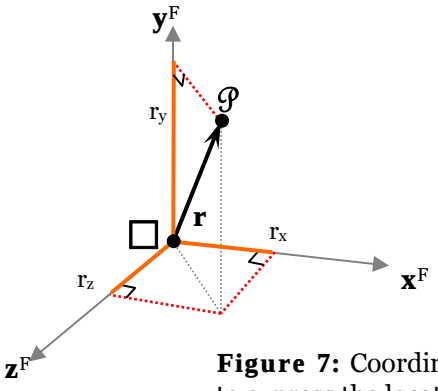


Figure 7: Coordinate frame F, and how to express the location of a point $\mathcal{P}$ in F.

I suspect the above has not been much of a stretch for most readers, since this is a perfectly ordinary example of a conventional coordinate frame. Possibly the notation and the term "measure number" are new, but everyone is

familiar with these concepts. We are just being excruciatingly precise in distinguishing the physical quantities of direction and location from their expression in a particular frame of reference.

This next idea may seem a bit odd if you haven't encountered it before: the concept of a frame makes perfect sense even if we can't say where it is or which way its axes are pointing. Once we have a frame F, for example, like the one defined above, we can start measuring things in frame F without the slightest idea how F is placed with respect to other things. We can even measure frame F in itself—the measure numbers of its axes are ${}^F\mathbf{x}^F={}^F[1,0,0]$, ${}^F\mathbf{y}^F={}^F[0,1,0]$, ${}^F\mathbf{z}^F={}^F[0,0,1]$ and its origin point is ${}^F\square^F={}^F[0,0,0]$. In a sense F defines its own self-consistent universe without reference to anything else. Note that this universe extends infinitely in every direction. In multibody mechanics we have another name for such an independent universe: a *body*.

4.2 Bodies

Fundamentally, a body B is just a moving reference frame, called the *body frame* B. You probably aren't used to thinking of a body this way! We will shortly connect this back to more intuitive "body" concepts like mass and geometry; however, it is the *frame* that is a body's most fundamental characteristic. One implication of this is that a body extends infinitely in all directions. Before you completely reject this idea, answer this question: is the hole part of the doughnut?* In any case the infinite extent of bodies will turn out to be very convenient when we start connecting them together.

The i^{th} body is B_i and its body frame is B_i with origin $\square^{Bi}$. In practice we'll only have to talk about a few bodies at a time so we can use different letters for them and avoid subscript bloat. In particular, body G is the distinguished body *Ground* representing the inertial (non-accelerating, non-rotating) reference frame.† The ground frame provides a global origin $\square^G$ (we'll usually

---

* Thanks to Paul Mitiguy for the doughnut analogy.

† Other names sometimes used for the ground frame are: Cartesian frame, Newtonian frame, world frame, inertial frame, laboratory frame, and experimenter's frame.

drop the frame superscript in this case and just say □) and fixed orthogonal directions **x**, **y**, **z**. By convention, we identify ground with the "0^{th}" body, that is, $B_0 \equiv G$.

Bodies typically have associated features which can be measured in and expressed in the body frame. These include other frames, *directions* (unit vectors) and *stations* (point locations). The body frame B origin is the station whose measure numbers when expressed in the body frame are ${}^B[0,0,0]$, and its axes are the directions with measure numbers ${}^B[1,0,0]$, ${}^B[0,1,0]$, and ${}^B[0,0,1]$. Mass properties include the total mass (a scalar), the center of mass (a station, represented numerically by a vector), and an inertia tensor (numerically a 3×3 symmetric matrix) which expresses rotational inertia about a particular station. When the inertia tensor is defined about the center of mass it is called the *central inertia*. For rigid bodies, mass properties are constant; for deformable bodies (not presently supported by Simbody) the mass is constant but the center of mass and inertia will be seen to vary when measured in the body frame.

4.3 Mobilizers

A *mobilizer* connects a body to its unique parent body,* and defines the relative mobility (degrees of freedom or "dofs") allowed between those bodies. These are often imprecisely called "joints"; we reserve the term "joint" to refer to the physical-world concept of that name, as illustrated in Figure 8.

* Recall that Ground is a body.

Figure 8: a mechanism with four joints; at most three can be mobilizers

A mobilizer is one way to implement a joint, but not all joints are mobilizers. For example, when a joint forms a loop as in the figure, it *reduces* the total mobility, requiring implementation as a constraint rather than a mobilizer. While the physical system is uniquely described in terms of its bodies and joints, in general there will be many ways to decompose that system into mobilizers and constraints for purposes of building a Simbody model.

There can be from 0 to 6 relative mobilities (degrees of freedom) between a pair of bodies. The parameterization of these mobilities is a set of *generalized coordinates q* representing the mobilizer's configuration, and *generalized speeds u* representing the mobilizer's motion.

The three fundamental mobilizer types are sliding, torsion, and orientation. A *sliding mobilizer* (syn: prismatic) provides a single degree of freedom representing translation along a defined axis, and adds a single coordinate with units of length to the system's set of generalized coordinates. A *torsion mobilizer* (syn: pin) provides a single degree of freedom representing rotation about a defined axis and adds a single generalized coordinate with angular units. An *orientation mobilizer* (syn: ball, spherical) permits unrestricted relative orientation between its pair of bodies, that is, three degrees of freedom and at least three corresponding generalized coordinates (for dynamics these require a four-element quaternion).

Most other mobilizer types can be viewed as compositions of the three basic types. For example, a *Cartesian mobilizer* is a composition of three sliding

joints with orthogonal axes and thus permits unrestricted relative translation (three degrees of freedom) between the bodies it connects. A *free mobilizer* is a composition of a Cartesian and an orientation mobilizer and permits six degrees of freedom (completely unrestricted motion) between its bodies. A free mobilizer serves to introduce independent rigid bodies into the system and simply provides a convenient reference frame and corresponding coordinates with which to express their motion. Note that, like all other mobilizers, a free mobilizer can be placed between any two bodies—it does not have to connect a body to ground. This allows very convenient relative coordinates to be used for collections of independent bodies. For example, one can express a protein domain that carries its local waters and ions along with it when it is moved kinematically.

Complex joints can be built up from mobilizers and constraints (see below). A "screw joint" for example can be composed of a coaxial sliding and torsion mobilizer, providing one translational and one rotational coordinate, plus a constraint enforcing a defined relationship (the screw's "pitch") between the time derivatives of these coordinates. The Simbody mobilizer concept is extensible (internally only, in this release) in the sense that arbitrarily complicated ones can be constructed. For example, a knee joint could be built as a 1-dof mobilizer so that a single unconstrained coordinate would be used to represent the complicated coordinated motion of a knee.

4.4 Constraints

Constraints may represent arbitrary restrictions on the generalized coordinates and generalized speeds, and linear restrictions on accelerations. Constraints arise, for example, if the body/joint connectivity graph contains a loop as in Figure 8. Constraints generate one or more *constraint equations*. Each independent constraint equation removes one degree of freedom from the system. In this sense constraints are the complement of mobilizers, whose generalized speeds each *add* one degree of freedom to the system. And in fact any n-dof mobilizer can be represented instead as a free mobilizer plus $6-n$ constraint equations.

Constraints among the moving bodies of a physical system act by introducing non-working internal forces and moments. These forces act in the same

manner as the applied forces described below—they can act on bodies or along joint axes, and as with applied forces they can always be reduced to a system of forces acting only along the mobilities. The only difference between constraint forces and externally applied forces is that the constraint forces are unknown and must be solved for simultaneously with the system accelerations.

4.5 Forces

By forces we mean *generalized forces* which include both forces and torques (moments).[*] Force vectors can be applied to the multibody system at any body station and moment vectors can be applied to any body (or implemented as pairs of forces). Scalar forces or torques can also be applied directly to the system's mobilities, that is, directly along the generalized speeds. All systems of forces can be reduced to an equivalent set acting only along the mobilities, and Simbody provides an operation which efficiently performs this useful conversion.

Forces can be functions of time, position, velocity, and their own internal states. They may be local effects or result from spatially distributed fields or a constant gravitational field, or act pairwise between distant stations (e.g. atoms) in the system. Forces which depend only on time and configuration are called conservative forces, and are the gradient of some potential energy function. Non-conservative forces may depend on velocities as well.

5 Simbody Matter Subsystem reference guide

The lowest-level Simbody API (application programmer interface), which is all we offer in the current release, assumes that the caller has made all modeling decisions and simply wants to perform calculations on the model. The primary decisions to be made are (1) how the physical model is to be decomposed into a particular set of rigid bodies, (2) what kinds of mobilizers

[*] The term *loads* is often used as an alternative with less ambiguity. However we will continue to use the more familiar term forces, usually meaning both forces and torques.

are to be used to interconnected them in a tree structure, and (3) what constraints, if any, should be present to restrict the allowable mobility. A variety of higher-level automated modelers for specific domains can be provided which can make these decisions and then use the low-level interface. Concepts involved in the API will be described in terms of convenient C++ objects first, and then we will discuss the equivalents for C and FORTRAN access which necessarily require more language-specific details.

5.1 Vectors and Matrices

Simbody makes use of lower-level SimTK toolsets to simplify its interface and internals. The (as yet unreleased) SimTK general purpose Simmatrix™ package (https://simtk.org/home/SimTKcommon) is used to handle basic vector and matrix objects. We follow the Simmatrix convention of using names containing "Vector" and "Matrix" to refer to large objects of variable dimension, and names containing "Vec" and "Mat" to mean small, fixed-size objects of known dimension. The basic types we use most are the fixed-size `Vec3` and `Mat33` types and the variable length `Vector` type. We use the basic Simmatrix types to build up a set of specialized vectors and matrices of particular use in manipulating physical objects, as described in the next sections.

5.2 Geometry

We provide a small set of specialized types for dealing with geometric quantities of interest in multibody dynamics. This is not intended to be a general purpose geometry package. For example, we happily assume that all geometry of interest is 3D.

Given the fundamental existence of a rigid body frame B, we are primarily interested in *stations*, *directions*, and other frames fixed in B. These are represented by *translations*, *rotations*, and *transforms* (*xforms*) respectively, which locate these objects with respect to an existing frame.

5.2.1 Stations

Stations are simply points which are fixed in a particular reference frame (i.e., they are "stationary" in that frame). They are specified by the translation

vector which would take the frame's origin to the station. A translation is represented by a Simmatrix `Vec3` type. Simbody does not provide an explicit `Station` class; `Vec3`'s are adequate whenever a station is to be specified.

5.2.2 Directions

Directions are unit vectors, which are `Vec3`s with the additional property that their lengths are always 1. We define a class `UnitVec3` which behaves identically to `Vec3` in most respects but restricts the ways in which values can be assigned to ensure that the length is always 1. This has concrete performance benefits because this unit length guarantee means that we can track length-preserving operations at compile time and avoid unnecessary normalization checks, or worse, unnecessary normalizations which are very expensive.

5.2.3 Rotations

There are many ways to express 3D rotations. Examples are: pitch-roll-yaw, azimuth-elevation-twist, axis-angle, and quaternions. Many others are in common use. Each way of writing orientation has its own quirks and complexities. However, all of these are equivalent to a 3x3 matrix, called a *rotation matrix* (synonyms: orientation matrix, direction cosine matrix). Rotation matrices have a particularly simple definition and straightforward physical interpretation, and are very easy to work with. At the API level, Simbody uses the rotation matrix as a least common denominator, embodied in a class `Rotation`. `Rotation` provides a set of methods which can be used to construct a rotation matrix from a wide variety of commonly-used rotation schemes.

Rotation matrices are simply 3x3 matrices whose three columns are mutually perpendicular directions (unit vectors) representing the axes of one coordinate frame, expressed in another. These are represented internally in objects of type `Rotation` as an ordinary Simmatrix `Mat33`, and behave identically except that their construction and assignment is restricted to ensure that certain properties are maintained. Those properties are: each column and row is a unit vector, the columns are mutually perpendicular, and the rows are mutually perpendicular, forming a right-handed set. That means

that the third column (row) is the positive cross product of the first two columns (rows). Such a matrix is orthogonal; hence its transpose is its inverse. Its determinant is +1, meaning that it is a pure rotation and not a reflection or scaling operation.

We use the symbol R with left and right superscripts ${}^{\text{From}}\text{R}^{\text{To}}$ to represent the orientation of the "to" frame (the right superscript) measured with respect to the "from" frame (the left superscript), like this:

$$ {}^{G}\text{R}^{B} \equiv \left[{}^{G}\mathbf{x}^{B} \quad {}^{G}\mathbf{y}^{B} \quad {}^{G}\mathbf{z}^{B} \right] $$

$$ {}^{G}\text{R}^{B} \equiv \left(\left[\mathbf{x}^{B} \right]_{G} \quad \left[\mathbf{y}^{B} \right]_{G} \quad \left[\mathbf{z}^{B} \right]_{G} \right) $$

(Remember that the notation $\mathbf{v}^{B}$ indicates a (column) vector quantity $\mathbf{v}$ fixed to reference frame B, while the operator $[\square]_{F}$ indicates that the measure numbers of some physical quantity are given in coordinate frame F.) So the symbol ${}^{G}\text{R}^{B}$ should be read "the axes of frame B expressed in frame G," or "the orientation of frame B in G," or just "B in G." We never use "R" alone for a rotation matrix; that is a recipe for certain disaster. Instead, we always provide the two frames. (When under tight typographical restrictions, as in source code, we write ${}^{G}\text{R}^{B}$ as `R_GB`.) Using this notation, one can simply match up superscripts to rotate vectors or compose rotations. Also, since these are orthogonal, the inverse of a rotation matrix is just its transpose, which serves simply to swap the superscripts. Using the Simmatrix "~" operator to indicate matrix transpose, $\sim{}^{G}\text{R}^{B} = {}^{B}\text{R}^{G}$. As an example, if you have a rotation ${}^{G}\text{R}^{B}$ and a vector $[\mathbf{v}]_{B}$ expressed in B, you can re-express that same vector in G like this: $[\mathbf{v}]_{G} = {}^{G}\text{R}^{B} \bullet [\mathbf{v}]_{B}$. To go the other direction, we can write $[\mathbf{v}]_{B} = {}^{B}\text{R}^{G} \bullet [\mathbf{v}]_{G} = \sim{}^{G}\text{R}^{B} \bullet [\mathbf{v}]_{G}$. As a C++ code fragment, this can be written

```
Rotation R_GB;  //orientation of frame B in G
Vec3     v_G;   //a vector expressed in G
…
Vec3     v_B = ~R_GB*v_G; //re-express v_G in frame B
```

Composition of rotations is similarly accomplished by lining up superscripts (subject to order reversal with the "~" operator). So given ${}^{G}\text{R}^{B}$ and ${}^{G}\text{R}^{C}$ we can get ${}^{B}\text{R}^{C}$ as ${}^{B}\text{R}^{C} = {}^{B}\text{R}^{G} \bullet {}^{G}\text{R}^{C} = \sim{}^{G}\text{R}^{B} \bullet {}^{G}\text{R}^{C}$. Note that the "~" operator has a high precedence like unary "-" so $\sim{}^{G}\text{R}^{B} \bullet {}^{G}\text{R}^{C}$ is $(\sim{}^{G}\text{R}^{B}) \bullet {}^{G}\text{R}^{C}$, not $\sim({}^{G}\text{R}^{B} \bullet {}^{G}\text{R}^{C})$.

As is typical for Simmatrix operations on small quantities, the transpose operator is actually just a change in point of view and involves no computation or copying of data. That is, the operations ${}^{B}R^{G} \bullet [\mathbf{v}]_G$ and ${}^{\sim G}R^{B} \bullet [\mathbf{v}]_G$ are exactly equivalent in both meaning and performance: the cost is 15 floating point operations (three dot products), with no wasted data copying or subroutine calls.

5.2.4 Transforms

Transforms combine a rotation and a translation and are used to define the configuration of one frame with respect to another. (Recall that we consider a frame to consist of both a set of axes and an origin point.) We represent a frame B's configuration with respect to another frame G by giving the measure numbers in G of each of B's axes, and the measure numbers in G of the vector from G's origin point to B's origin point, for a total of 4 vectors, which can be interpreted as a 3x3 `Rotation` (see above) followed by the origin point location (a `Vec3`). Following computer graphics convention, we call this object a *transform* (abbreviated *xform*) and conceptually augment the axes and origin point to create a 4x4 linear operator which can be applied to augmented vectors (4^{th} element is 0) or points (4^{th} element is 1), or composed using matrix multiplication. We define a type `Transform` which conceptually represents transforms as follows:

$$ {}^{G}X^{B} \equiv \left(\begin{bmatrix} \mathbf{x}^B \\ 0 \end{bmatrix}_G \quad \begin{bmatrix} \mathbf{y}^B \\ 0 \end{bmatrix}_G \quad \begin{bmatrix} \mathbf{z}^B \\ 0 \end{bmatrix}_G \quad \begin{bmatrix} {}^{G}p^{B} \\ 1 \end{bmatrix}_G \right) $$

(Recall that ${}^{G}p^{B} \equiv {}^{O_G}p^{O_B}$, that is, the vector from the origin of the G frame to the origin of the B frame.) Note that we use the symbol X for transforms, with superscripts ${}^{From}X^{To}$ so ${}^{G}X^{B}$ means "the transform from frame G to frame B," or "frame B measured from and expressed in frame G." Another way to interpret ${}^{G}X^{B}$ is that it represents the operations that must be performed on G to bring it into alignment with B (a rotation and a translation). Then as for rotation matrices described above, we can interpret ${}^{G}X^{B} \bullet {}^{B}X^{C}$ as a composition

of operators yielding ${}^{G}X^{C}$, and $\sim{}^{G}X^{B}$ is defined to yield the inverse transform ${}^{B}X^{G}$.*

The above transform matrix can be considered a matrix of four columns as shown: three augmented vectors and an augmented point. An alternate, and entirely equivalent, way to view this is as a rotation matrix, translation vector, and an extra row:

$$ {}^{G}X^{B} \equiv \begin{pmatrix} \begin{pmatrix} & & \\ & {}^{G}R^{B} & \\ & & \end{pmatrix} & \begin{pmatrix} \\ {}^{G}p^{B} \\ \end{pmatrix} \\ \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} & 1 \end{pmatrix} $$

In our implementation, the physical layout of a Simbody `Transform` is just the three columns of the rotation matrix followed immediately in memory by the translation vector, that is, ${}^{G}X^{B} \equiv \left({}^{G}R^{B} \mid {}^{G}p^{B} \right)_{3\times 4}$. There is no need for the fourth row to be stored in memory since it is always the same.

Given a `Transform`, you can work with it as though it were a 4x4 matrix, or work directly with the rotation matrix *R* and translation vector *p* individually, without having to make copies. Although a transform defined this way is not orthogonal, its inverse is easy to apply with no additional calculation. As described above, we overload the normal matrix transpose operator "~" to recast a `Transform` to its inverse so that either the transform or its inverse can be used conveniently in an expression, for example, ${}^{B}X^{C} = \sim{}^{G}X^{B} \bullet {}^{G}X^{C}$. As is typical using Simmatrix objects, this inverse operator is just a change of point of view at zero cost, so the total cost is the same in either direction. For example, to transform a point measured and expressed in one frame to the equivalent one re-measured and re-expressed in another frame costs one 3x3 matrix-vector multiply and one addition of 3-vectors per transformed point,

---

* Note that this is actually a different definition for the "~" operator than is used in Simmatrix, since the inverse of a transform is not simply its transpose. However, the analogy with ~R (which is both the transpose and inverse of rotation matrix R), combined with the lack of any practical use for the transpose of a transform, makes this use of "~" very attractive and natural to use in practice.

for a total of 18 floating point operations (flops), and the cost is the same if we transform it back using a `Transform` inverse. A straightforward implementation of a 4x4 transform (i.e., as an actual 4x4 matrix times a 4-vector) would require 28 floating point operations per transformed point. Composition of `Transform`s (using the '*' operator for matrix multiply) is done in 63 flops but would take 112 using a 4x4 matrix multiply. Thus `Transform` provides the convenience of a 4x4 transformation matrix at substantially lower cost.

5.3 Mechanics

Some additional specialized quantities arise in mechanics for dealing with mass properties, which consist of a mass, center of mass, and inertia matrix for each body. Mass is a simple scalar and center of mass just a point so we do not define special classes for them. Inertia, however, is a tensor quantity (a 3x3 matrix) which is expected to exhibit certain properties. Among these, it is symmetric, and the values of its elements must satisfy certain relationships. In addition, there are common operations on inertias which can be most efficiently and conveniently provided with a distinct inertia class. So we provide a class `Inertia` which is stored physically as a 3x3 symmetric matrix, i.e., a Simmatrix `SymMat33` containing six real-valued numbers. This behaves like an ordinary matrix for read-only operations but its construction and assignment is restricted to enforce physical relevance, and additional operations are provided, such as shifting inertia taken about one point to the equivalent inertia about another point.

For convenience we combine all the mass properties into a `MassProperties` class, which contains a mass, a center of mass location, and an inertia matrix. Note that there is implicitly a reference frame in whose axes the vector and tensor are expressed, and from whose origin the center of mass location and inertia distribution are measured.

5.4 Spatial Notation

We also build on the Simmatrix types to define some specialized vectors and matrices useful in mechanics. Following Jain and Rodriguez,[13] we use *spatial notation* which combines translational and rotational quantities into a single

object. Using Simmatrix we define the convenient type `SpatialVec` to mean a stacked vector of two ordinary 3-vectors, and `SpatialMat` to mean a 2x2 matrix of ordinary 3x3 matrices, that is

```
typedef Vec<2,Vec3>    SpatialVec;
typedef Mat<2,2,Mat33> SpatialMat;
```

Note that these convenient types have well-defined interpretations as packed arrays of real numbers, which means they have equivalent descriptions in C and FORTRAN, which we'll address later. There is zero overhead in C++ for using the more expressive types.

The first subvector of a `SpatialVec` is always the rotational component, and the second is the translational one. Some examples of spatial vectors: spatial velocity V, spatial acceleration A, and spatial force F, defined like this:

$$V \equiv \begin{pmatrix} \omega \\ v \end{pmatrix}, \quad A \equiv \begin{pmatrix} \beta \\ a \end{pmatrix}, \quad F \equiv \begin{pmatrix} \tau \\ f \end{pmatrix}$$

where ω is an angular velocity vector, v a linear velocity, β an angular acceleration, a is a linear acceleration, τ a torque (moment), and f is a force. Each of these elements is an ordinary 3-vector (`Vec3`). Unless otherwise indicated, all quantities are measured with respect to the ground frame G, and linear quantities are referred to the body origin. That is, the default symbols above represent

$$V = {}^G V^B \equiv \begin{pmatrix} {}^G\omega^B \\ {}^G v^{O_B} \end{pmatrix}, \quad A = {}^G A^B \equiv \begin{pmatrix} {}^G\beta^B \\ {}^G a^{O_B} \end{pmatrix}, \quad F = {}^G F^B \equiv \begin{pmatrix} {}^G\tau^B \\ {}^G f^{O_B} \end{pmatrix}$$

Orientation is not a vector quantity, so we can't use a `SpatialVec` to represent configuration (orientation and location) of a rigid body (that is, of a reference frame). Instead, we use the `Transform` class described above to represent spatial configuration, with

$${}^G X^B \equiv \left({}^G R^B \;\middle|\; {}^{O_G} p^{O_B} \right)$$

For any vector quantity $\mathbf{v}$, we use the notation $\mathbf{v}_\times$ to indicate a 3x3 skew-symmetric cross product matrix such that for any vector $\mathbf{w}$, $\mathbf{v}_\times \cdot \mathbf{w} = \mathbf{v} \times \mathbf{w}$. Spelled out in scalars, the cross product matrix is

$$\mathbf{v}_\times = \begin{pmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{pmatrix}$$

Note that $\mathbf{v}_\times^{\mathsf{T}} = -\mathbf{v}_\times$.

5.4.1 Spatial mass properties

The mass properties of a rigid body conventionally consist of the body's mass m, the mass center location $\mathbf{p}$, and its inertia tensor $\mathbf{I}$. It is convenient to view the inertia tensor as the product of the mass and a gyration tensor $\mathcal{G}$, such that $\mathbf{I}=m\mathcal{G}$. Then a spatial inertia matrix M can be written as a spatial gyration matrix (giving the mass distribution) scaled by the total mass:

$$M \equiv m\begin{pmatrix} \mathcal{G} & \mathbf{p}_\times \\ -\mathbf{p}_\times & \mathbf{1}_3 \end{pmatrix}$$

The spatial inertia matrix of a body B about its origin O_B is then

$$M_B = m_B\begin{pmatrix} \mathcal{G}_B & {}^{O_B}p_\times^{C_B} \\ -{}^{O_B}p_\times^{C_B} & \mathbf{1}_3 \end{pmatrix}$$

Note that when the spatial mass properties are given about the center of mass C_B we have

$$M_B^{C_B} = m_B\begin{pmatrix} \mathcal{G}_B^{C_B} & 0 \\ 0 & \mathbf{1}_3 \end{pmatrix}$$

where $\mathcal{G}_B^{C_B} = \mathcal{G}_B + (\mathbf{p}\mathbf{p}^{\mathsf{T}} - \mathbf{p}^{\mathsf{T}}\mathbf{p}\mathbf{1}) = \mathcal{G}_B + \mathbf{p}_\times\mathbf{p}_\times$, $\mathbf{p} = {}^{O_B}p^{C_B}$ is the central inertia.

If we have the spatial velocity V^C also referred to the center of mass, i.e. $V^C = \begin{pmatrix}\boldsymbol{\omega}^{\mathsf{T}} & \mathbf{v}^{C\mathsf{T}}\end{pmatrix}^{\mathsf{T}}$, then we can define another spatial vector quantity, spatial momentum of a body "referred to" its center of mass:

$$P^C \equiv M^C V^C = m\begin{pmatrix} \mathcal{G}^C & 0 \\ 0 & \mathbf{1} \end{pmatrix}\begin{pmatrix} \boldsymbol{\omega} \\ \mathbf{v}^C \end{pmatrix} = m\begin{pmatrix} \mathcal{G}^C\boldsymbol{\omega} \\ \mathbf{v}^C \end{pmatrix}$$

In the more general (and typical) case where the body origin $O_B \neq C_B$ we compute spatial momentum the same way with the result being the spatial momentum referred to O_B, which is *not* the same quantity:

$$P \equiv MV = m\begin{pmatrix} \mathcal{G} & \mathbf{p}_\times \\ -\mathbf{p}_\times & \mathbf{1} \end{pmatrix}\begin{pmatrix} \boldsymbol{\omega} \\ \mathbf{v} \end{pmatrix} = m\begin{pmatrix} \mathcal{G}\boldsymbol{\omega} + \mathbf{p}\times\mathbf{v} \\ \mathbf{v} + \boldsymbol{\omega}\times\mathbf{p} \end{pmatrix} = P^C + m\begin{pmatrix} \mathbf{p}\times\mathbf{v}^C \\ 0 \end{pmatrix}$$

A body's kinetic energy (a scalar) is calculated from spatial momentum like this:

$$KE = \tfrac{1}{2}V^T MV = \tfrac{1}{2}V \cdot P = \tfrac{1}{2}m(\boldsymbol{\omega} \cdot \mathcal{G}\boldsymbol{\omega} + \mathbf{v}^2 + \rho)$$
$$\rho \equiv (\boldsymbol{\omega} \cdot \mathbf{p}\times\mathbf{v} + \mathbf{v} \cdot \boldsymbol{\omega}\times\mathbf{p}) = 2\mathbf{v} \cdot \boldsymbol{\omega}\times\mathbf{p}$$

Note that although the angular momentum must be referred to a specific point, kinetic energy is independent of that point. That is

$$KE = \tfrac{1}{2}V^{\mathsf{T}} MV \quad = \tfrac{1}{2}V \cdot P \quad = \tfrac{1}{2}m(\boldsymbol{\omega} \cdot \mathcal{G}\,\boldsymbol{\omega} + \mathbf{v}^2 + 2\mathbf{v} \cdot \boldsymbol{\omega}\times\mathbf{c})$$
$$= \tfrac{1}{2}V^{C\mathsf{T}} M^C V^C = \tfrac{1}{2}V^C \cdot P^C = \tfrac{1}{2}m(\boldsymbol{\omega} \cdot \mathcal{G}^C\boldsymbol{\omega} + \mathbf{v}^{C\,2})$$

5.4.2 Re-expressing spatial quantities

For any quantity Q we use the notation $[Q]_Z$ to mean "Q re-expressed in frame Z." Note that this never changes the physical quantity being represented, just the frame in which the measure numbers of that quantity are expressed. If Q is a vector currently expressed in frame A, then $[Q]_Z \triangleq {}^Z R^A \cdot Q$. If M is a tensor (matrix) currently expressed in frame A, then $[M]_Z \triangleq {}^Z R^A \cdot M \cdot \left({}^Z R^A\right)^{\mathsf{T}} = {}^Z R^A \cdot M \cdot {}^A R^Z$. We use similar definitions for spatial quantities. If Q is a *spatial* vector currently expressed in frame A, then we define ${}^Z R^A \cdot Q \triangleq \begin{pmatrix} {}^Z R^A & 0 \\ 0 & {}^Z R^A \end{pmatrix} \cdot Q$. For example,

$$\left[{}^A V^B\right]_Z = {}^Z R^A \cdot {}^A V^B \triangleq \begin{pmatrix} {}^Z R^A & 0 \\ 0 & {}^Z R^A \end{pmatrix} {}^A V^B = \begin{pmatrix} {}^Z R^A \cdot {}^A\omega^B \\ {}^Z R^A \cdot {}^A v^{O_B} \end{pmatrix} = \begin{pmatrix} \left[{}^A\omega^B\right]_Z \\ \left[{}^A v^{O_B}\right]_Z \end{pmatrix}.$$

To re-express a spatial inertia matrix M from frame A to frame Z, write

$$[M]_Z = {}^Z R^A \cdot M \cdot ({}^Z R^A)^{\mathsf{T}}$$

$$= \begin{pmatrix} {}^Z R^A & 0 \\ 0 & {}^Z R^A \end{pmatrix} M \begin{pmatrix} {}^A R^Z & 0 \\ 0 & {}^A R^Z \end{pmatrix} = m \begin{pmatrix} {}^Z R^A \mathcal{G} {}^A R^Z & ({}^Z R^A \mathbf{p})_\times \\ -({}^Z R^A \mathbf{p})_\times & \mathbf{1}_3 \end{pmatrix}$$

$$= m \begin{pmatrix} [\mathcal{G}]_Z & [\mathbf{p}]_{Z\times} \\ -[\mathbf{p}]_{Z\times} & \mathbf{1}_3 \end{pmatrix}$$

where we have made use of the fact that if U is a 3x3 orthogonal matrix, then $\mathrm{U} \cdot \mathbf{v}_\times \cdot \mathrm{U}^{\mathsf{T}} = (\mathrm{U} \cdot \mathbf{v})_\times$. (Recall that rotation matrices are orthogonal.)

5.4.3 Rigid body shifting of spatial quantities

Rigid body shifting is used during processing of the multibody tree to transfer the effect of inboard kinematic quantities (velocities and accelerations) in an outboard direction, and to shift applied forces from outboard bodies in an inward direction. The rigid body shift matrix ${}^P\phi^Q$ is used to shift a spatial force acting at point *Q* to the equivalent spatial forces acting at point *P*. The transpose of that matrix ${}^Q\phi^{*P} \equiv \left({}^P\phi^Q\right)^{\mathsf{T}}$ is used to shift a velocity or acceleration of point P to the equivalent velocity or acceleration of point Q. Then the operators are

$${}^P\phi^Q \equiv \begin{pmatrix} 1 & {}^P p_\times^Q \\ 0 & 1 \end{pmatrix} \qquad \text{and} \qquad {}^Q\phi^{*P} \equiv \left({}^P\phi^Q\right)^{\mathsf{T}} \equiv \begin{pmatrix} 1 & 0 \\ -{}^P p_\times^Q & 1 \end{pmatrix}$$

so that $$F^P = {}^P\phi^Q \cdot F^Q = \begin{pmatrix} \tau + {}^P p^Q \times f^Q \\ f^Q \end{pmatrix}$$

and $$V^Q = {}^Q\phi^{*P} \cdot V^P = \begin{pmatrix} \omega \\ v^P + \omega \times {}^P p^Q \end{pmatrix}.$$

To shift a spatial inertia matrix about a point Q to another point P of the same rigid body, use

$$M^P = {}^P\phi^Q \cdot M^Q \cdot {}^Q\phi^{*P} = m_B \begin{pmatrix} 1 & {}^P p_\times^Q \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{G}^Q & {}^Q p_\times^C \\ -{}^Q p_\times^C & \mathbf{1}_3 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -{}^P p_\times^Q & 1 \end{pmatrix}$$

$$= m_B \begin{pmatrix} \mathbf{G}^Q - {}^P p_\times^Q {}^P p_\times^Q & {}^P p_\times^C \\ -{}^P p_\times^C & \mathbf{1}_3 \end{pmatrix}$$

Note that there is no mention of expressed-in frame; the shifting operators assume that all quantities are expressed in the same frame.

5.5 Topology

In describing the "matter" side of a multibody system, the most fundamental property is the system *topology*. By topology we mean just these properties:

- A set of bodies (that is, reference frames). One distinguished body *Ground* is always present.
- The *mass structure* of each body. The five currently-supported mass structures are: (1) ground, (2) massless, (3) particle, (4) line, and (5) rigid body.
- For each body except Ground, a unique "parent" body with respect to which the body's mobility will be defined. This leads to a tree topology for the system as a whole, with the ground body at its root.
- A set of *topological constraints*, that is, constraints which are always present and active. These can impart a closed-loop topology to the system as a whole.

A body's mass structure defines the most general form that the body's mass properties can take on. Ground and massless bodies have only a single predefined set of mass properties: infinite and zero respectively. Particles can take only a point mass, and never have inertia about that point. A line body can be thought of as a linear arrangement of particles, and thus has mass, a meaningful center of mass along the line, and equal central inertias in two directions perpendicular to the line, but none about the line. A rigid body (representing a mass distribution on a surface or in a volume) can have a full inertia.

5.6 Mobility

Mobility expresses the allowable motion of a body's frame with respect to its parent's frame. Bodies start out with no mobility at all, meaning that the body's frame and its parent's frame are coincident and will stay that way forever. In Simbody, *Mobilizers* are used to provide between zero and six

independent degrees of freedom between a body and its parent, allowing translation and/or rotational motion of the body frame with respect to its parent. We call these unrestricted degrees of freedom a body's *mobilities* with respect to its parent. The unique Ground body has no mobility.

Summing the mobilities of each body in a multibody system, the total of n mobilities defines an n-dimensional *mobility space* for the multibody system. The n mobilities are independent by construction and thus form a basis for mobility space. Only configurations in *mobility* space are representable by the multibody system. Typically there are many conceivable configurations which simply cannot be expressed. For example, consider a system composed of Ground and one moving body, a wheel, having a single mobility with respect to Ground consisting of just a rotation about a fixed axis. One can imagine a configuration in which the wheel is removed from the axis, but the chosen multibody system simply can't express that. With just one coordinate, an angle, we can only talk about rotations of the wheel about an axis. Additional mobilities would have to have been granted to the wheel in order to express more general configurations.

This ability to limit the mobility space of a multibody system is extremely powerful if you happen to know something about the space containing the solutions of interest to you. To continue the above example, if you are a car designer rather than a crash-test engineer, then you know that correct solutions to your vehicle simulation problems will always exhibit wheels that are attached to their axes. Solutions in that smaller space are much easier to find than solutions in the much larger space where wheels may be found anywhere. Similarly, in molecular mechanics if you know that certain groups of atoms are always observed to move together as rigid bodies, problems are much easier to solve in a reduced space in which only those groupings can be expressed, than one in which the atoms *could* be anywhere. We know that correct solutions would always "rediscover" the known groupings (at great, and unnecessary, expense).

However, we will often find that even mobility space is substantially bigger than our known solution space. Instead, we would like to focus on a lower-dimensional subspace of mobility space, called *constrained space*. The

dimensionality of constrained space is the net number of degrees of freedom possessed by the multibody system. So a multibody system's net degrees of freedom can be smaller than the sum of its bodies' individual mobilities.

One might wish simply to redefine the mobility of the bodies so that only constrained space can be expressed (that is, mobility space=constrained space), and that is a very good thing to do if you can. Unfortunately, in general constrained space cannot be parameterized directly. Instead we create a system with a small but convenient-to-define mobility space, and then add a set of Constraints whose satisfaction implicitly defines the constrained space.

5.6.1 Parameterization of mobility

The mobilities of the bodies in a multibody system, taken together, define its mobility space. However, we must choose a particular parameterization of this space (that is, a basis) in order to express a particular configuration and motion of the multibody system and this choice is not unique. Conveniently, body mobilities are mutually independent so we may choose the parameterization for each body separately. The set containing all these parameters is then the parameterization of mobility for the multibody system as a whole.

The independence of body mobilities localizes the parameterization issue to the Mobilizer for each body. Each Mobilizer must define two sets of scalar parameters to express particular values for its mobilities, one set to specify the relative positioning (configuration) and the other to specify the relative velocity (motion) between the parent and child bodies. Parameters used for positioning are conventionally called *generalized coordinates*; parameters for velocity are called *generalized speeds*.[*] The symbol q is used to represent a vector of generalized coordinates, and u is a vector of generalized speeds.

---

[*] *Generalized* here refers to the inclusion of both translational and rotational coordinates. We similarly use *generalized forces* to mean both forces and torques.

In Simbody, the number of a body's generalized speeds u is *always* the same as that body's mobility—e.g., if a body has five degrees of freedom with respect to its parent, then it will also have five u's. The u's are thus mutually independent. u's have interpretations with direct physical meaning, and the system equations of motion are written in terms of the time derivatives of u, which we denote $\dot{u}$. The generalized coordinates q, on the other hand, must at times be chosen for convenience or computational stability and do not always map directly to physical quantities, so in general $\dot{q} \neq u$. In fact, for many bodies there will be more q's than u's in which case the q's are not always independent. However, the interdependence among a body's q's is always a localized relationship among only those q's, and never involves other bodies. At any particular configuration, there is always a linear, invertible relationship between $\dot{q}$ and u, and each Mobilizer provides the necessary conversions. As a specific example, during dynamic calculations Simbody Mobilizers that permit unrestricted relative orientation between a body and its parent use four quaternions to stably represent the orientations, while the three generalized speeds are just the elements of the relative angular velocity vector. The four quaternions must satisfy a normalization constraint, leaving only the expected three degrees of freedom for the four coordinates.

For the whole multibody system, the generalized speeds are aggregated in a vector whose length is the sum of the mobilities of each body. This vector is the set of generalized speeds for the multibody system and is also designated u. A vector q aggregating the individual bodies' generalized coordinates forms the generalized coordinates for the whole multibody system. Together, q and u constitute the instantaneous state of the matter component of a multibody system. It will usually be clear from context whether we are referring to the coordinates of the whole system or just one body, but if we need to be specific we use q^B and u^B to indicate the mobilizer parameters for body B.

5.6.2 Mobilizers are not joints

When describing a multibody system, a *joint* is a higher-level (more abstract) concept than a Mobilizer, although they are easily confused. In general, joints are implemented as a combination of Mobilizers and Constraints, and may also introduce force elements (e.g. friction or soft stops). It is possible to

create topological loops with joints but not with Mobilizers, as the latter are restricted to connections between bodies and their unique parents. So a Mobilizer can only *add* degrees of freedom to a system, while a joint may add or remove them.

5.7 Bodies and their Mobilizers

The primary Simbody representation of matter is a multibody tree, that is, a tree-structured collection of interconnected bodies, which we call a `SimbodyMatterSubsystem`. On initial construction, a `SimbodyMatterSubsystem` contains just a single body, the inertial frame Ground (body 0) which is the root of the multibody tree. To add a body B to an existing `SimbodyMatterSubsystem`, you will need to be able to specify the following properties:

- The parent body P (with body frame P), which must already be in the multibody tree.
- A reference frame (axes and origin) for the body (this is implicit, but you need to have it in mind). We call that the body frame B. (See Figure 5 for an example.)
- Mass properties for the body, with the center of mass location measured from O_B and expressed in B, and the inertia measured about O_B and expressed in B.
- The mobilizer's mobilized frame M attached to B. You must be able to express M's configuration on B as a transform ${}^{B}X^{M}$ from B to M.
- The mobilizer's fixed frame F, attached to P, which will be connected to M by the mobilizer. You must be able to express F's configuration on P as a transform ${}^{P}X^{F}$ from P to F.
- The kind of mobilizer to be used to connect B to its parent body P.

Figure 9 shows a body B being added to a tree already containing its parent P. Not shown are the body's mass m_B, its inertia $\mathbf{I}_B$ about O_B and the transforms ${}^{B}X^{M}$ and ${}^{P}X^{F}$.

When this information is supplied to the appropriate Simbody method, the new body becomes part of the growing tree, and a unique, small integer body number is assigned which can be used to refer to the body later. The specified mobilizer is the unique *inboard* mobilizer of body B, that is, the mobilizer which connects it to a body which is closer (in a graph path-length sense) to the Ground body. When defining the sense (sign) of mobilizer coordinates later we will refer to the frame F on P as the "fixed" frame, and frame M on B as the "moving" or "mobilized" frame, although these terms are arbitrary and do not imply anything of physical significance except when P is ground in which case it really is "fixed."

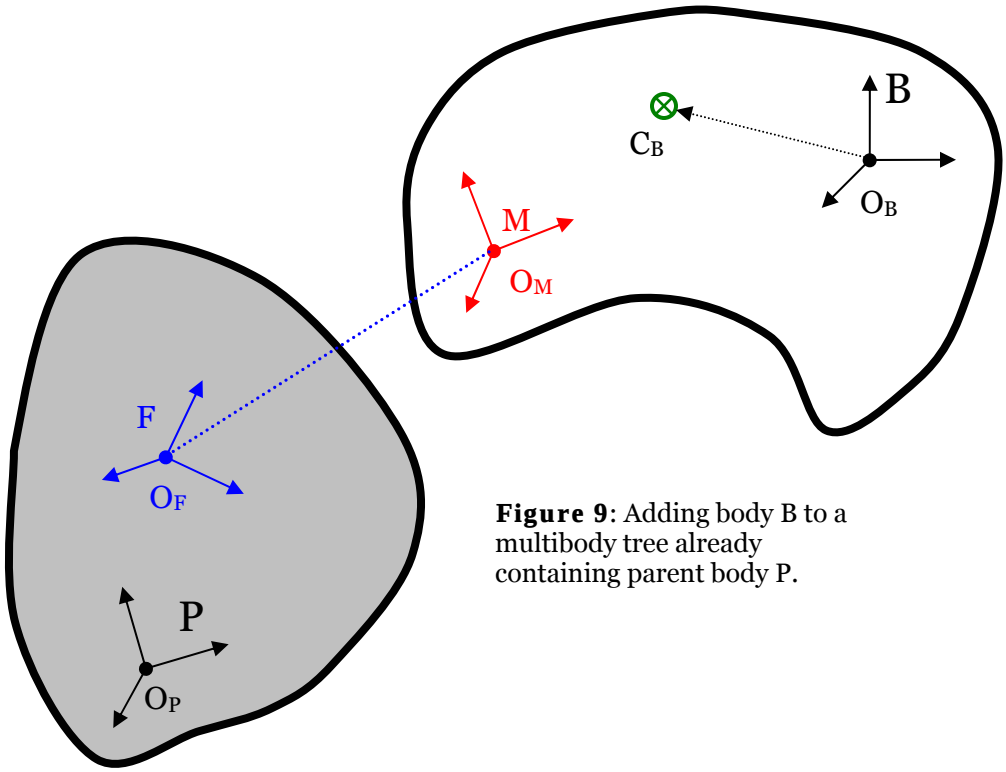


Figure 9: Adding body B to a multibody tree already containing parent body P.

5.7.1 The reference configuration

The frames M and F are used to define a *reference configuration* for each body with respect to its parent. For most mobilizers, that is the configuration

in which M and F are overlaid, and corresponds to a value of zero for the mobilizer's generalized coordinates*. Figure 10 shows the reference configuration for the mobilizer defined in Figure 9. For any mobilizer type, the values of the generalized coordinates q express a transform ${}^{F}X^{M}$ which gives the current location and orientation of the M frame, measured from and expressed in the F frame. The definition of each mobilizer type specifies the meaning of each of the q's for that mobilizer and the kinds of transforms that can be expressed. For example, a Cartesian mobilizer would permit arbitrary translation of M, but its axes must remain forever aligned with those of F. A ball (spherical) mobilizer's coordinates express the complementary motion in which the origins of the two frames must remain coincident forever, but the orientation of M can be arbitrary with respect to F. A sliding mobilizer permits translation along one axis only, and a torsion (pin) mobilizer permits only rotation about a single axis. Other mobilizers permit various combinations of rotation and translation, with the extremes being the Free mobilizer which permits all possible motion (six degrees of freedom) and the Weld (im)mobilizer which permits no motion at all (zero degrees of freedom).

Regardless of the mobilizer type, setting all the coordinates to zero expresses that the M and F frames are in the reference configuration.

* Certain sets of mobilizer coordinates may define their own "zero" which does not necessarily correspond to numerical values of zero for all coordinates. For example, zero ("no rotation") for a quaternion is a four-vector (1,0,0,0).

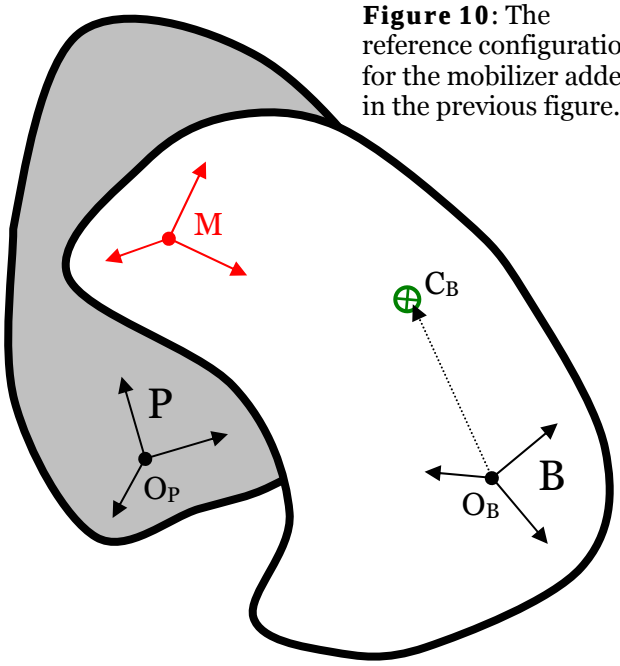


Figure 10: The reference configuration for the mobilizer added in the previous figure.

Those users familiar with SD/FAST's reference configuration should note that the above is a different method for defining the reference configuration. It is in fact the opposite approach: SD/FAST requires the bodies to be entered already in the reference configuration, and then defines the mobilizer (SD/FAST joint) frames from the reference configuration. We think it is much more natural to express the joint frames separately in their bodies' frames, and then define the reference configuration from the joint frames. It is always possible to choose mobilizer frames to reproduce the ones used by SD/FAST if you want, but it is no longer necessary to calculate them that way.

5.8 Constraints

Constraints in Simbody are the complement of Mobilizers. Mobilizers *add* mobility to a multibody system; Constraints *reduce* mobility by introducing one or more *constraint equations*. Mobilizers are local, granting degrees of freedom to a single body, while Constraints are global and remove degrees of freedom from the multibody system as a whole by introducing restrictions on the allowable relationships among the generalized coordinates, speeds, or accelerations. A simple example is a distance constraint which says that a particular point fixed on one body must always be at a certain distance *d* from a point fixed on another body. If those bodies are far apart in the graph of the

multibody topology, this simple restriction is actually expressing a complicated relationship that must hold among the mobility coordinates of *many* bodies. As mentioned earlier, it is much more efficient to define less mobility in the first place than to grant the bodies their freedom and then take it away later! However, as with the distance constraint above, that is not always possible or convenient, so we have Constraints.

In the same way that a single Mobilizer may introduce several *mobilities*, a single Constraint may generate multiple *constraint equations*. Unlike mobilities, which are globally independent, the constraint equations generated by Constraints may be mutually interdependent making some of the constraints ineffective, redundant or inconsistent. A trivial example of a redundant constraint would be adding the same Constraint twice—nothing changes since mobility coordinates which satisfy the first Constraint also satisfy the second. An example of an ineffective constraint would be restricting the distance between a point on the outside of a wheel and the point of the parent at the wheel's center. If the specified distance is equal to the wheel's radius, the single mobility automatically meets this restriction at all times and the system has the same net mobility with or without the restriction. Changing the required distance to anything other than the wheel's radius creates an inconsistent constraint which can never be satisfied by *any* setting of the mobility coordinates.

Simbody initially supports only a small selection of Constraints. These are: Rod (distance) Constraint, Ball (coincident points) Constraint, and Weld (coincident frames) Constraint. A Rod constraint generates one constraint equation which maintains a user-specified constant, non-zero separation distance between a station on one body (that is, a point fixed on the body) and a station on another body, as measured along the line between the two stations. Each nonredundant distance constraint removes one degree of freedom from the system. A Ball or "coincident points" constraint generates three constraint equations which together hold a station from each of two distinct bodies together at the same location in space, i.e., at a separation distance of zero, exactly like a Ball joint. A nonredundant Ball constraint thus *removes* three translational degrees of freedom from the system (all translation between the two points), while a Ball mobilizer *adds* three

rotational degrees of freedom (all rotation about the connected points). A Weld constraint maintains frames (both location and orientation) from each of two bodies coincident in space, generating six constraint equations and thus removing six degrees of freedom from the system. Weld Constraints are the primary means by which we take a system that has loop topology and make it a tree—we cut one of the bodies in two to break the loop and then weld the two halves back together with a Weld constraint.

The information needed for adding one of the above Constraints to a Simbody multibody system is as follows:

- Two distinct bodies A and B. Either one (but not both) may be Ground. Both bodies must already be part of the multibody tree and are identified by the body number that was returned at the time they were added.
- (Distance or Coincident Points Constraint) A station point P^A fixed on body A and station point P^B fixed on body B. These are measured and expressed in the bodies' local frames, that is, P^A is measured from O_A and expressed in A while P^B is measured from O_B and expressed in B. The measure numbers of these vectors are thus constant during simulation.
- (Weld Constraint) A frame F^A fixed on body A and a frame F^B fixed on body B. These are expressed in the bodies' local frames, that is, F^A is given by a transform ${}^A X^{F^A}$ while F^B is given by transform ${}^A X^{F^B}$. The measure numbers of the transforms are thus constant.
- For a Rod (constant distance) Constraint you also need to supply a scalar distance. This is the physical separation $d=|P^B-P^A|$ between the stations that you would like Simbody to maintain at all times. This separation must be significantly larger than zero; zero distance between stations is obtained using a Ball Constraint rather than a Rod Constraint.

Note that nonredundant constraints cannot be satisfied by arbitrary values of the mobility coordinates. Prior to a simulation, you must find an initial set of generalized coordinates q and speeds u that satisfies all the constraint

equations. Occasionally this can be done by inspection or hand calculation, but in general it is a difficult nonlinear problem to be solved numerically prior to beginning a simulation (this is called *assembly analysis* for q and *velocity analysis* for u). Given any set of mobility coordinates q and u, Simbody can efficiently calculate the constraint equation violations those entail. A variety of numerical methods can then be used to drive those constraint violations to below a desired tolerance, at which point the associated constraints will be satisfied. After that, valid numerical studies maintain the constraint equations, and thus satisfy the Contraints, as they advance from step to step.

5.9 Generalized forces

We can apply forces to bodies, or directly to the mobility coordinates represented by the generalized speeds u. In general these include both linear and rotational forces (torques). Forces applied to mobilities are called *generalized forces* or *mobility forces*. Forces applied to the bodies are called *spatial forces* or *body forces*. There is always a unique set of mobility forces equivalent to any set of body forces, in the sense that both sets will produce the same accelerations. Calculating this equivalent set is an important Simbody capability, since the equations of motion are written in terms of the mobilities, while forces are typically known in terms of their effects on the bodies.

It is important to note that calculation of applied forces is not limited to the force types provided Simbody. Force calculation is a domain-specific modeling issue; Simbody's job is to provide the information needed by the modeler to calculate the forces, and then to respond to those forces in accordance with Newton's laws of motion. For convenience, the Simbody distribution does include a set of basic force subsystems to use in calculating simple forces such as gravity, springs, and atomic forces; however, this is by no means an exhaustive set.

5.10 Kinematics

Kinematics is usually defined as the study of motion without regard to mass or force. In practice, however, it is entirely concerned with the mapping between the mobility coordinates and spatial positions, velocities, and

accelerations of the bodies. The mobility coordinates and speeds uniquely determine the spatial quantities so the mapping in that direction is fast and direct; this is called forward kinematics. Given a q we can immediately say where all the bodies are; with q and u we can say how they are moving; and with q, u, and $\dot{u}$ (where $\dot{u}$ is the time derivative of u) we can say how they are reacting (accelerating). The reverse direction is called inverse kinematics and is more difficult unless all bodies have been given unrestricted mobility (i.e., they are "free"). Given a set of observed spatial kinematic quantities, the goal of inverse kinematics is to find the "best fit" mobility coordinates and speeds that satisfy both the observations and the constraint equations generated by the multibody systems' own Constraints.

5.11 Dynamics

Dynamics is concerned with the relationship between forces and accelerations at a fixed value of the state variables q and u. This is determined by Newton's second law, f=ma. *Forward dynamics* attempts to calculate accelerations and internal constraint forces, given a set of applied forces (which is equivalent to some set of joint forces). *Inverse dynamics* attempts to determine what set of joint forces explains a given set of joint accelerations. In practice it is often useful to specify some accelerations and some forces and calculate the remaining unknowns.

It is important to understand that the scope of the Simbody multibody tree (the "matter" subsystem) includes only the instantaneous evaluations performed at a particular system state. Advancing the state through time to produce a trajectory, or searching through the state to satisfy particular objectives, are higher-level operations which are facilitated by the multibody capabilities described here. They will be discussed elsewhere.

5.12 Equations

These are the equations represented by the Simbody multibody tree.

$$\dot{q} = \mathbf{Q}u \tag{1}$$

$$\mathbf{M}\dot{u} + \mathbf{G}^{\mathsf{T}}\lambda = \mathbf{f} - \mathbf{f}_{\text{bias}} \tag{2}$$

$$\mathbf{G}\dot{u} = \mathbf{b} \tag{3}$$

Equation (1) (kinematic differential equations) relates the generalized coordinate derivatives to the generalized speeds via an invertible linear transform represented by block diagonal matrix **Q**. This is typically an identity mapping except for some orientation coordinates.

Equations (2) and (3) together constitute the dynamic equation in the presence of constraints, showing the unknown accelerations $\dot{u}$ and the unknown internal constraint force multipliers λ. Here **M**=**M**(q) is the system mass matrix, **G=G**(q) is the constraint matrix which couples the $\dot{u}$'s and serves to map the constraint multipliers into mobility forces. Together with **b**=**b**(t,q,u), **G** provides the constraints among the $\dot{u}$'s which must be satisfied in the solution. Some readers may prefer to think of equations 2 and 3 as a single matrix equation like this:

$$\begin{bmatrix} \mathbf{M} & \mathbf{G}^{\mathsf{T}} \\ \mathbf{G} & 0 \end{bmatrix} \begin{pmatrix} \dot{u} \\ \lambda \end{pmatrix} = \begin{pmatrix} \mathbf{f} - \mathbf{f}_{\text{bias}} \\ \mathbf{b} \end{pmatrix} \qquad (2a)$$

Equation 2a clarifies that the vector $(\dot{u}, \lambda)^{\mathsf{T}}$ comprises a single set of unknowns to be solved for simultaneously when the right hand side is known.

f=**f**(t,q,u) represents all applied forces and torques mapped to equivalent joint forces. $\mathbf{f}_{\text{bias}}$(q,u) includes velocity-dependent Coriolis and gyroscopic effects, and is zero if u=0. We partition **f** as follows:

$$\mathbf{f} = \mathbf{f}_{\text{mob}} + \mathrm{J}^{\mathsf{T}} \bullet \mathbf{F}_{\text{body}} \qquad (4)$$

Here $\mathbf{f}_{\text{mob}}$ and $\mathbf{F}_{\text{body}}$ are the user-supplied system of forces and torques, while the kinematic Jacobian J=J(q) is managed internally by Simbody. $\mathbf{F}_{\text{body}}$ is an nbx1 "stacked" vector of spatial forces consisting of one element per body (that is, the per-body net result of all the forces and torques applied to each body), where each element is a 6-element spatial vector combining body torque and force as described above. $\mathbf{F}_{\text{body}}$ collects user-applied forces (such as Cartesian forces on atoms). If gravity has been specified, then $\mathbf{F}_{\text{body}}$ includes the spatial forces resulting from a uniform acceleration field **g** (fixed in the Ground frame) acting through the centers of mass of each body. $\mathbf{f}_{\text{mob}}$ is an nx1 vector of user-supplied scalar forces applied directly to the mobilities, such as would be used for bonded forces in a molecular model. $\mathrm{J}^{\mathsf{T}}\bullet$ is the

Cartesian-to-internal conversion operator, conceptually an *n*x*nb* matrix of spatial vectors that maps spatial forces to their equivalent mobility forces (some authors call J the matrix of "partial velocities"). In practice the $J^T \bullet$ operator is an O(*n*) algorithm, and J is never formed explicitly in Simbody.

5.13 SimbodyMatterSubsystem API

Here is a draft of the low-level API for Simbody, in C++. The highest level object is the `SimbodyMatterSubsystem` (defined in the `SimTK` namespace).

```
class SimbodyMatterSubsystem {
public:
TODO
};
```

5.14 Operator form of Simbody interface

$$\begin{aligned} \dot{q} &= \mathbf{Q}u \\ \mathbf{M}\dot{u} &= \mathbf{f} - \mathbf{G}^T\lambda \\ \mathbf{G}\dot{u} &= \mathbf{b} \\ \mathbf{v}(t,q,u) &= 0 \\ \mathbf{p}(t,q) &= 0 \\ \dot{z} &= \dot{z}(t,q,u,z) \end{aligned} \qquad \mathbf{f} = \mathbf{f}_{\text{mob}} + \mathrm{J}^T \bullet (\mathbf{F}_{\text{body}} + G - C)$$

$$\begin{bmatrix} \mathbf{M} & \mathbf{G}^T \\ \mathbf{G} & 0 \end{bmatrix} \begin{pmatrix} \dot{u} \\ \lambda \end{pmatrix} = \begin{pmatrix} \mathbf{f} \\ \mathbf{b} \end{pmatrix}$$

$\dot{u} = \dot{u}_0 - \dot{u}_C$, where $\dot{u}_0 = \mathbf{M}^{-1}\mathbf{f}$, $\dot{u}_C = \mathbf{M}^{-1}\mathbf{G}^T\lambda$, and $\lambda = (\mathbf{G}\mathbf{M}^{-1}\mathbf{G}^T)^+(\mathbf{G}\dot{u}_0 - \mathbf{b})$

$$\dot{q}(u) = \mathbf{Q}_{[q]} u$$
$$\dot{u}_0(f) = \mathbf{M}^{-1}{}_{[q]} f$$
$$\lambda(\varepsilon) = (\mathbf{G}\mathbf{M}^{-1}\mathbf{G}^{\mathrm{T}})^{+}{}_{[q]}\varepsilon$$
$$f_\lambda(\lambda) = \mathbf{G}^{\mathrm{T}}\lambda$$
$$\varepsilon_a(\dot{u}) = \mathbf{G}_{[q]}\dot{u} - \mathbf{b}_{[t,q,u]}$$
$$\dot{u}_c(f) = \mathbf{M}^{-1} f_\lambda(\lambda(\varepsilon_a(\dot{u}_0(f))))$$
$$\dot{u}(f) = \dot{u}_0(f) - \dot{u}_c(f)$$
$$\varepsilon_v(u) = \mathbf{v}_{[t,q]}(u)$$
$$\varepsilon_p(q) = \mathbf{p}_{[t]}(q)$$
$$\dot{z} = \dot{z}(t,q,u,z)$$

$$G_{body}(q) = g \cdot \ldots$$
$$C_{body}[q](u) = u^T \cdot \ldots$$
$$f_{body}(\mathbf{F}_{body}) = J_{[q]} \cdot \mathbf{F}_{body}$$
$$\ddot{q}(\dot{u}) = \dot{\mathbf{Q}}_{[q,u]} u + \mathbf{Q}_{[q]}\dot{u}$$

$$\varepsilon_k(\hat{\dot{q}}) = \hat{\dot{q}} - \mathbf{Q}_{[q]} u$$
$$\varepsilon_d(\hat{\dot{u}}) = \mathbf{M}_{[q]}\hat{\dot{u}} + \mathbf{G}^{\mathrm{T}}(\mathbf{G}\mathbf{M}^{-1}\mathbf{G}^{\mathrm{T}})^{+}{}_{[q]}\varepsilon_a(\hat{\dot{u}}) - \mathbf{f}_{[t,q,u,z]}$$
$$\varepsilon_a(\hat{\dot{u}}) = \mathbf{G}_{[q]}\hat{\dot{u}} - \mathbf{b}_{[t,q,u]}$$
$$\varepsilon_v(\hat{u}) = \mathbf{v}_{[t,q]}(\hat{u})$$
$$\varepsilon_p(\hat{q}) = \mathbf{p}_{[t]}(\hat{q})$$
$$\varepsilon_z(\hat{\dot{z}}) = \hat{\dot{z}} - \dot{z}_{[t,q,u,z]}$$

$$\varepsilon_{dae} = \begin{bmatrix} \mathbf{M} & \mathbf{G}^{\mathrm{T}} \\ \mathbf{G} & 0 \end{bmatrix} \begin{pmatrix} \hat{\dot{u}} \\ \hat{\lambda} \end{pmatrix} - \begin{pmatrix} \mathbf{f} \\ \mathbf{b} \end{pmatrix}$$

The Simbody subsystem follows the response/operator/solver scheme described elsewhere. Arguments in brackets indicate the stage at which the operator is available; other symbols are the runtime arguments.

Operator	Stage	Method	Description
$\dot{q} = \mathbf{Q}_{[q]} u$	`Position`	`void calcQDot(State, Vector u, Vector& qdot)`	Convert generalized speeds to generalized coordinate time derivatives.

$\ddot{q} = \dot{\mathbf{Q}}_{[q,u]}\dot{u} + \mathbf{Q}_{[q]}\dot{u}$	Velocity	void calcQDotDot(State, Vector udot, Vector& qdotdot)	Convert generalized speed time derivatives to generalized coordinate 2nd time derivatives.
$f_{\mathrm{a}} = \mathbf{M}_{[q]}a$ $f_{\mathrm{c}} = \mathbf{G}^{\mathsf{T}}_{[q]}\lambda$ $f_{\mathrm{bias}} = \boldsymbol{\tau}_{[q,u]}$ $f_{\mathrm{inv}} = f_{\mathrm{a}} + f_{\mathrm{c}} + f_{\mathrm{bias}}$	Dynamics	void calcMa(State, Vector a, Vector& f)	Inverse dynamics. Can use as residual (implicit) form of equations : $f_{\underset{[q,u]}{\mathrm{inv}}}(\dot{u},\lambda) - f_{\underset{[t,q,u]}{\mathrm{applied}}} = 0$
$a = \mathbf{M}^{-1}_{[q]}f$ $\dot{u}_{\mathrm{tree}} = \mathbf{M}^{-1}_{[q]}(f - f_{\underset{[q,u]}{\mathrm{bias}}})$ $\dot{u}_{\mathrm{loop}} = \dot{u}_{\mathrm{tree}} - \dot{u}_{\mathrm{cons}}$ $\dot{u}_{\mathrm{cons}} = \mathbf{M}^{-1}\mathbf{G}^{\mathsf{T}}\lambda(\varepsilon_{\mathrm{a}}(\dot{u}_{\mathrm{tree}}))$	Dynamics	void calcMInverseF(State, Vector f, Vector& a) void calcTreeUdot(State, Vector f, Vector& udot)	Forward dynamics.
$\varepsilon_{\mathrm{a}} = \mathbf{G}_{[q]}a + \mathbf{b}_{[t,q,u]}$	Dynamics	void calcAccelerationConstraintErr (State, Vector a, Vector& aerr)	Maps accelerations a to the acceleration constraint errors they entail.
$\varepsilon_{\mathrm{a}} = \begin{bmatrix}\ddot{\mathbf{p}}\\ \dot{\mathbf{v}}\\ \mathbf{a}\end{bmatrix}_{[t,q,u,\dot{u}]}$	Acceleration	const Vector& getAccelerationConstraintErr(State)	Maps accelerations $\dot{u}$ to the acceleration constraint errors they entail.
$\varepsilon_{\mathrm{v}} = \begin{bmatrix}\dot{\mathbf{p}}\\ \mathbf{v}\end{bmatrix}_{[t,q,u]}$	Velocity	const Vector& getVelocityConstraintErr(State)	Given a set of generalized speeds u, return the velocity constraint errors they entail.
$\varepsilon_{\mathrm{p}} = \mathbf{p}_{[t,q]}$	Position	const Vector& getPositionConstraintErr(State)	Given a set of generalized coordinates q, return the position constraint errors they entail.

$\lambda = (\mathbf{G}\mathbf{M}^{-1}\mathbf{G}^{\mathsf{T}})^{+}_{[q]}\varepsilon_{\mathrm{a}}$	Dynamics	`void calcMultipliers(State, Vector aerr, Vector& lambda)`	Given a set of acceleration constraint violations, calculate the multipliers needed to eliminate them.
$f = \mathbf{J}^{\mathsf{T}}_{[q]}F$	Position	`void calcTreeEquivalentForces(State, Vector_<SpatialVec> bodyForces, Vector& jointForces)`	Given a set of body forces and torques, convert them to hinge forces ignoring constraints.
$V = \mathbf{J}_{[q]}u$	Position		Given a set of generalized speeds, compute the equivalent spatial velocities of each body.
ke=$\mathbf{ke}[q](u)$	Position	`Real calcKineticEnergy(State, Vector u)`	Given a set of generalized speeds, calculate the resulting kinetic energy.

6 Simbody Force Subsystems reference guide

Simbody comes with a predefined set of commonly-used force subsystems. Each of these is an independent, self-contained set of related features, and users may add their own force subsystems as well.

6.1 Uniform Gravity subsystem

6.2 General Force Elements subsystem

6.3 Hertz/Hunt and Crossley contact model subsystem

Simbody comes with a force subsystem class called HuntCrossleyContact. This section describes the theory behind it.

6.3.1 Motivation

Most engineers, physicists and computer scientists are introduced to contact problems using the concept of *coefficient of restitution*. The idea presented is

that when two objects collide, they will rebound in a predictable way with the rebound velocity being a known fraction *e* of the impact velocity.

Unfortunately, it is rarely mentioned that this concept is only usable in the most limited cases. Many difficulties arise trying to apply this in a multibody dynamics context; in particular the presence and motion of the other bodies and the forces applied to them (which change constantly and are not know in advance) change the rebound velocity. Also, it is well known in the field of contact mechanics (and to anyone who has watched closely as a ball bounces) that the coefficient of restitution is very sensitive to the impact velocity. In fact, in contact mechanics the normal way to approximate the coefficient of restitution is $e = 1 - cv_i$ for small impact velocity v_i, where *c* is a material property. An enormous amount of empirical data supports that—at low velocities, normal materials have a coefficient of restitution that drops linearly with impact velocity. The classic work in this field is reference 3. Even with this improvement to the functional form of *e*, the results are rarely applicable outside the realm of freely falling bodies. In multibody dynamics, the coefficient of restitution is something to be *computed*, along with the rest of the system's motion, not something that can be known in advance!

To obtain usable results in a multibody context, we need a method that can calculate *forces* produced during contact, rather than impulsive velocity changes. That permits contact to be treated as yet another force among the many that influence the behavior of multibody systems, ensuring that accurate (or at least reasonable!) behavior will result. Only once you can obtain physically correct results with *some* model, should an optimization like "treat contact as an instantaneous event" be attempted, and even then one might wonder if it is worth the effort.

6.3.2 The model

This model is based on Hertz theory of elastic contact,[1] and the Hunt and Crossley model for damping.[2] The idea is to predict contact behavior during a dynamic simulation working only from material properties and geometry.

This is a frictionless model but it can be used as a starting point for several useful frictional models.

To apply Hertz theory, we need two linearly elastic materials in non-conforming contact, where the dimensions of the contact patch are small compared to the curvatures, and small compared to the overall dimensions of the object. Hertz theory can be used for general curved shapes (including cylinders) provided they can be approximated by paraboloids at the contact point; however, we will discuss only sphere-sphere and sphere-halfspace contact here. For Hunt and Crossley, the impact velocities should be small enough not to cause permanent yielding of the materials. Within these regimes, the model produces a good match for empirical data, such as that found in reference 3. Outside these limits, the model can still produce surprisingly useful results when fit to experimental data, because the *form* of the model has a structure which captures the most significant aspects of contact for many purposes. It is especially well-suited for soft contacts such as are common in biology, even though those are well out of the range that the rigorous theory presented here can support. I speculate that it works well in most applications because the results of interest don't usually depend on precise details of contact, only that it behaves in a qualitatively correct manner. As an example, if you get a stiffness parameter too low, the model will compensate by allowing more deformation with the result being that you get the same forces (such as are needed to keep a foot going through the floor) although the precise deformation of the foot and floor are not obtained. This may be acceptable for researchers who are more interested in studying some other aspect of the model, knee flexion for example.

For the rest of this section, please refer to Figure 11 which defines the geometry of contact. We will consider a collision between two bodies, B_1 and B_2, in which a sphere attached to B_1 contacts a sphere or halfspace attached to B_2. During the collision (which will occur over an extended period of time, not impulsively), our goal will be to determine instantaneous values for the contact force f, the contact patch orientation $\mathbf{n}$ and radius a, and a unique contact point P at which we can apply equal and opposite forces to the two contacting bodies. We will be given the spatial locations and velocities of the *undeformed* geometric objects in contact, and will easily be able to determine

the total deformation x that must have occurred because of the apparent overlap between the undeformed objects. However, in order to find the contact point P and the compression rates of each body (needed to compute dissipation), we have to determine the *individual* deformations x_1 of B_1 and x_2 of B_2, where $x = x_1 + x_2$ and $\dot{x} = \dot{x}_1 + \dot{x}_2$.

The thin lines in the figure are intended to show the undeformed shape while the thicker lines give a (crude) depiction of the deformed shape. Note the assumption that the contact patch is planar, circular of radius a, centered at P and oriented with normal **n** pointing towards body B_1. With these conventions, the scalar force f that we will calculate (applied equal and opposite to the two bodies at P) is always positive, and the vector force $f\mathbf{n}$ is applied to body B_1 at P, while we apply $-f\mathbf{n}$ to body B_2 at P. From the diagram one might think it doesn't matter where along the line between the centers we apply the force. However, it is important to keep in mind that the colliding objects are in general only *attached* to a larger body—they do not constitute the whole body. That means the applied force is also generating moments on the bodies, and those moments depend critically on exactly where the force is applied.

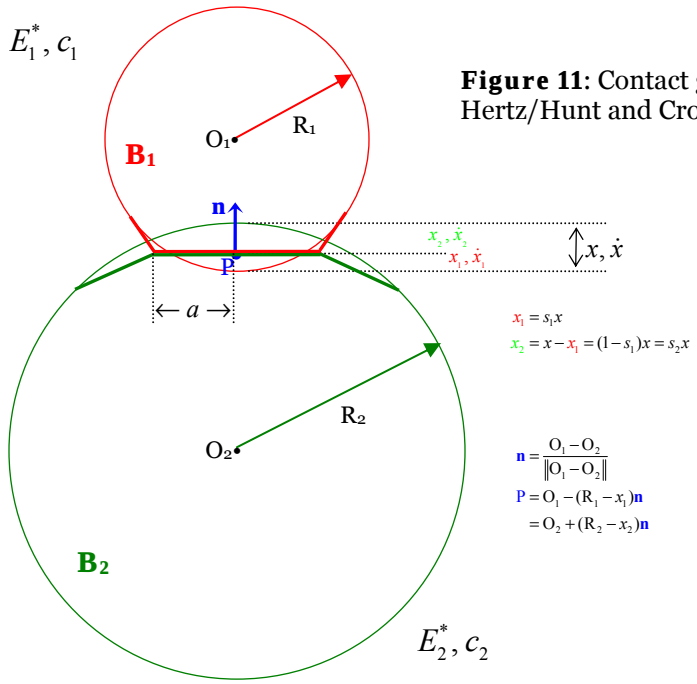


Figure 11: Contact geometry for the Hertz/Hunt and Crossley model.

We expect to be given the following material properties for each body:

Property	Symbol	Units	Comments
Radius of curvature	R	Length	Measured at the contact point
Young's modulus	E	Pressure	stress/strain = (force/unit area) / (% deformation)
Poisson's ratio	ν	Unitless ratio	ratio of transverse contraction to deformation (0–½ for normal materials); related to preservation of volume during strain; rubber has ν=½
Dissipation coefficient	c	1/velocity	–slope of coef. of restitution vs. velocity at low velocities; i.e., coef. of restitution $e=1-cv_i$ for impact velocity v_i

For our purposes, we combine Young's modulus E and Poisson's ratio v into a single "stiffness" property called the plane-strain modulus $E^*=E/(1-v^2)$. This is measured as the pressure per unit area induced by a fractional deformation (strain). The MKS unit is Pascals which are Newtons/m^2. Below are some typical values as ballpark figures only; please don't rely on them. (Note that the stiffnesses are given in *giga*pascals, i.e. 10^9 N/m^2!)

Material	Young's modulus E (GPa)	Poisson's ratio ν (unitless)	Plane-strain modulus E* (GPa)	Dissipation coefficient (s/m)
Rubber	0.01	0.5	.0133	0.05?
Bacteriophage capsid	2	0.4(?)	2.4	?
Nylon	3	0.4	3.6	?
Lead	14	0.42	17	0.4?
Concrete	25	0.15	25.6	?
Steel	200	0.3	220	0.08?
Diamond	1100	0.2	1150	?

Of these, Young's modulus and Poisson's ratio can be obtained easily from handbooks for most materials, but the dissipation coefficient is harder to get. It would be very useful to relate this to standard properties such as hardness and yield stress (if that's possible) but for now it has to be measured or estimated as the slope of the coefficient of restitution-vs.-velocity curve at low velocities. References 2 and 3 provide or imply some values for c, but they should be taken with a grain of salt. Note that this situation is still better than the standard approach of supplying a coefficient of restitution e directly—at least c is a material property so can be expected to produce correct behavior over a range of velocities.

Hertz contact theory says the relationship between force f_{Hz} and displacement x depends only on the relative curvature R of the two bodies at the contact point, and on an effective plane strain modulus E^*, and the contact patch radius a is an even simpler function

$$f_{\text{Hz}} = \tfrac{4}{3}\sqrt{R}E^* x^{3/2}, \quad a = \sqrt{R}\,x^{1/2}$$

Hunt and Crossley start with the above formula for f_{Hz} and add a dissipation term:

$$f_{\text{HC}} = f_{\text{Hz}}(1 + \tfrac{3}{2}c\dot{x})$$

where c is an effective dissipation coefficient combining the material properties of the two contacting materials.

Note that although the materials are assumed linear, the force-displacement relationship is nonlinear because of the changing geometry during contact. This complicates the calculation of the effective stiffness E^*. The literature seems to suggest $E^* = E_1^* E_2^* / (E_1^* + E_2^*)$ but this would be inconsistent with the Hertz relationship, by the following reasoning. First, the relative curvature is a geometric property and is straightforward to calculate: $R = R_1 R_2 / (R_1 + R_2)$. Looking at the figure, note that the contact situation depicted should be indistinguishable from one in which B_1 (the top, red body) had met an infinitely rigid halfspace, with a displacement of x_1 instead of x, provided that B_1's radius were R instead of R_1. The effective stiffness in that

case would be just the stiffness E_1^* of B_1. Hertz theory would then give $f_1 = \frac{4}{3}\sqrt{R}E_1^* x_1^{3/2}$. By the same reasoning, we can view B_1 as a rigid half space and see that the force on B_2 (with radius changed to R) would be unchanged at $f_2 = \frac{4}{3}\sqrt{R}E_2^* x_2^{3/2}$. But the forces must be the same on both bodies and the same as $f = \frac{4}{3}\sqrt{R}E^* x^{3/2}$. Recalling that $x = x_1 + x_2$, we now have enough information to write E^* in terms of E_1^* and E_2^*:

$$E_1^* x_1^{3/2} = E_2^* x_2^{3/2} = E^* \left(x_1 + x_2\right)^{3/2}$$
$$\Rightarrow E_1^{*2/3} x_1 = E_2^{*2/3} x_2 = E^{*2/3} \left(x_1 + x_2\right)$$
$$\Rightarrow E^* = \left(\frac{E_1^{*2/3} E_2^{*2/3}}{E_1^{*2/3} + E_2^{*2/3}} \right)^{\frac{3}{2}}$$

Note that this combining formula is close, but not identical, to $E^* = E_1^* E_2^* / (E_1^* + E_2^*)$. The general scheme is that if your force/displacement dependency has an exponent n, as in $f = kx^n$, then the combining scheme for the material stiffness is

$$E^* = \left(\frac{E_1^{*1/n} E_2^{*1/n}}{E_1^{*1/n} + E_2^{*1/n}} \right)^n$$

We can now rearrange this for our case where n=3/2 to determine how x is split into x_1 and x_2 given the stiffnesses of the materials, the result we need to determine the contact point location P:

$$x_1 = \left(\frac{E^*}{E_1^*} \right)^{\frac{2}{3}} x = \frac{E_2^{2/3}}{E_1^{2/3} + E_2^{2/3}} x$$
$$x_2 = \left(\frac{E^*}{E_2^*} \right)^{\frac{2}{3}} x = \frac{E_1^{2/3}}{E_1^{2/3} + E_2^{2/3}} x = x - x_1$$

By inspection, the time derivatives $\dot{x}_1$ and $\dot{x}_2$ are split in the same ratios, which gives us a way to define an equivalent dissipation coefficient for $\dot{x}$: $c = c_1 s_1 + c_2 \left(1 - s_1\right)$, where $s_1 = E_2^{2/3} / \left(E_1^{2/3} + E_2^{2/3}\right)$. To summarize, here are the combining rules we use:

$$R = \frac{R_1 R_2}{R_1 + R_2}, \quad E^* = \left(\frac{E_1^{*2/3} E_2^{*2/3}}{E_1^{*2/3} + E_2^{*2/3}} \right)^{\frac{3}{2}}$$

$$s_1 = \frac{E_2^{2/3}}{E_1^{2/3} + E_2^{2/3}}, \quad s_2 = \frac{E_1^{2/3}}{E_1^{2/3} + E_2^{2/3}} = 1 - s_1$$

$$x_1 = s_1 x, \quad x_2 = s_2 x$$

$$c\dot{x} = c_1 \dot{x}_1 + c_2 \dot{x}_2 \Rightarrow c = c_1 s_1 + c_2 s_2$$

Now we can apply the Hunt and Crossley model, which starts with Hertz contact and adds a dissipation term:

$$\begin{aligned} f &= \max(f_{HC}, 0) \\ &= \max(\tfrac{4}{3}\sqrt{R} E^* x^{3/2} (1 + \tfrac{3}{2} c\dot{x}), 0) \end{aligned}$$

The max() is needed only when an active force is "yanking" two contacting bodies apart; the force will never be negative in normal contact/response conditions (see reference 4 for proof). The "yanking" situation corresponds to pulling the bodies apart faster than they can undeform.

6.3.3 Extension to include Friction

TBD

Friction models need to know the normal force, and sometimes the contact patch dimensions, and the Hertz/Hunt and Crossley model provides those.

We hope to provide a simple, continuous model with functionality like that described in reference 5, which is able to accurately model sticking, pre-sliding, and sliding friction behavior and exhibit empirically observed Stribeck, Coulomb and viscous friction effects *without* adding intermittent constraints to the multibody model and event detection to the numerical methods.

6.4 DuMM — Molecular mechanics force field

Simbody comes with a force subsystem class called `DuMMForceFieldSubsystem`, which we'll abbreviate "`DuMM`" below. This is

intended to provide a straightforward implementation of conventional molecular mechanics force fields, for use in experimenting with rigid-body molecule models, and to serve as sample code for someone who would like to write or port a good molecular mechanics force field for Simbody. It is *not* intended for production work!

6.4.1 Background

Molecular mechanics (MM) uses classical approximations of molecular interactions. It is thus suited only for circumstances in which quantum effects are not dominant; in practice that means simulations which do not form or break covalent bonds between atoms. Fortunately this includes a lot of biologically interesting behavior such as binding, aggregation, protein folding, and other cases where molecules rearrange rather than form or break.

Atomic force models are conventionally divided into two categories: bonded and non-bonded. Bonded forces act between or among covalently-bound "neighbor" atoms. Since each atom can form only a small number of bonds, the number of bonded interactions is $O(n_a)$ in the number of atoms n_a. Non-bonded forces, on the other hand, represent interactions between each atom and all the other atoms. These are electronic in nature and comprise Coulomb forces and van der Waals forces. Because the number of such forces is $O(n_a^2)$, these terms dominate the computational cost of the force field for all but the smallest systems.

6.4.2 Basic concepts

The primary concepts supported by DuMM are the force field, molecule, and body. The resulting model permits matter to be coarse-grained (that is, large bodies interconnected by mobilizers and constraints) while retaining detailed atomic forces and geometry. The same methods are used to produce systems from ones where atoms are free to move anywhere in Cartesian space, to systems where all the atoms move together as a rigid body, to anything in between. Different molecules or pieces of molecules can be modeled at different granularity in the same simulation.

6.4.2.1 Force field

The force field provides broad *atom classes* providing van der Waals parameters for particular elements in particular covalent environments. All bonded terms are specified in terms of these atom classes. A larger set of *charged atom types* is defined which combine atom classes with particular partial charges. Each atom in the molecule is classified as a particular charged atom type, which implicitly provides the partial charge, van der Waals parameters, and element. Then the force field provides bonded terms for stretch, bend, and torsion, defined as a pair, triple, or quad of atom classes.

The force field definition includes a few global parameters as well, such as how to scale charge and van der Waals forces for closely-bonded atoms, and how to mix van der Waals parameters for dissimilar atom classes.

6.4.2.2 Molecules

Molecules are built from three concepts: atoms, bonds, and clusters. The only information required in the definition of an atom is its charged atom type as described above. An integer *atomId* is assigned and returned to the caller, so that every atom in the system has a unique atomId. A bond connects a pair of atoms, with at most one bond allowed between any pair.

A cluster is a rigid grouping of atoms. When a cluster is defined it is assigned a unique *clusterId*, which is returned to the caller as a handle for future references to that cluster. Each cluster has its own reference frame, like a body, and when initially created a cluster consists only of that reference frame. Whenever an atom is placed in a cluster, it is given a station (position) with respect to that cluster's reference frame. Clusters may be placed within larger clusters, in which case a Transform is used to specify the configuration (location and orientation) of the child cluster's reference frame with respect to the parent cluster's frame. An atom may appear only once within a cluster or any of its subclusters. However, an atom may be placed in multiple clusters as long as those clusters are independent.

Once a cluster has been populated with atoms, it can calculate its own mass properties which can then be used in the construction of bodies.

6.4.2.3 Bodies

Once molecules have been constructed by adding atoms and bonds and then partitioning the atoms into clusters, a mapping of the atoms to `SimbodyMatterSubsystem` bodies can be made. Bodies serve as a "top level" cluster, and atoms and clusters can be attached to bodies. Any time an atom is attached to a body it is given a station in the body's reference frame, and a cluster is given a configuration (Transform).

Note that mass properties are not automatically determined by attaching atoms and clusters to bodies. Rather, bodies must have mass properties assigned at the time they are defined in the `SimbodyMatterSubsystem`. Typically, the mass properties as calculated by clusters, and the masses of individual atoms, will be used in calculating the appropriate mass properties but that is not required.

Once the bodies are assigned, `DuMMForceFieldSubsystem` will figure out which of its atoms are on different bodies, and consequently which of the bonded terms cross bodies. Bonded and nonbonded terms that act only within a single body are ignored.

There is no automatic mapping of mobilizer coordinates to bonds, and in fact there is not necessarily any direct mapping possible. Optionally, you may assign particular mobilities to any of the cross-body bonded terms (such as a sliding mobility to a bond stretch term or a rotating mobility to a bond torsion angle). Bonded terms which depend directly on mobilities can be calculated very efficiently, and it can be very convenient to have a coordinate which corresponds directly to a bonded term. (TODO: bond mapping not implemented yet).

6.4.3 Units

There are a number of molecular mechanics unit systems in popular use. DuMM supports a single "native" unit system but provides conversions to and from the others. The native unit system is sometimes called "MD units" and is defined by the following units: length in nanometers (nm, 10^{-9} m), mass in daltons (Da, g/mol, atomic mass units), and time in picoseconds (ps, 10^{-12} seconds). Angles are measured as unitless radians. In this set of units, a

typical bond has a length of about 0.15 nm, a hydrogen atom has mass about 1 Da, and substantial motion occurs on a scale of about 1 ps.

This is a particularly appealing set of units because when combined consistently into energy (mass x length^2/time^2) we get energy per mole in g-nm^2/ps^2=10^{3}kg-m^2/s^2 =1kJ. That is, our energy unit is 1 kilojoule/mol which is one of the energy units popular among molecular mechanics practitioners. (Our consistent unit of force is then the kJ/nm = 1 Da-nm/ps^2.)

The other popular unit system, perhaps somewhat more chemist-friendly than ours, is the kcal-Ångströms (KA) system. It uses the kilocalorie (kcal) for energy, where 1 kcal = 4.184 kJ, and the Ångström (Å, 0.1 nm) for length (those are both exact conversions), degrees for angles, and ps for time. However, there is no reasonable consistent set of units in which energy is measured in kcals, so there is always a conversion involved in this system.* The DuMM subsystem provides alternate methods dealing directly in kcals, Ångstroms, and degrees so that users who think better in KA units can continue to do so, hopefully resulting in a smaller chance of errors being made. Whenever we use these nonstandard units we include "KA" in the method and argument names; any time no unit system is specified you may assume we are using MD units as described above. And no matter which methods were called initially, anyone who looks at internal data should be aware that our internal units are kJ, nm, ps, and radians.

6.4.4 Defining a force field

TODO

6.4.5 Defining the molecules

TODO

---

* Typically, energy is calculated in the consistent unit of decajoules/mol (Da-A^2/ps^2) and then divided by 418.4 when no one is looking.

6.4.6 Defining bodies and attaching the molecule to them

TODO

6.4.7 Running a simulation

TODO

6.4.8 Theory

TODO

7 Other Simbody Subsystems reference guide

TODO

7.1 Visualization subsystem

8 Simbody Systems reference guide

TODO

8.1 MultibodySystem

8.2 MolecularMechanicsSystem

9 Simbody Studies reference guide

The system described by equations (11.16)–(11.19) is an overdetermined system since there are more equations than unknowns. The n_q+n+n_z+m unknowns are q,u,z and λ. The first line of equation (11.16) provides only n_u independent equations, but the second adds n_{quat} more for a total of n_q kinematic equations. Then equation (11.17) provides $n+m$* with the first line and n_z more with the second line. That leaves (11.18) and (11.19) as $2m_p+m_v$

* When **G** doesn't have full row rank (meaning some of the constraints are redundant or inconsistent), we introduce other conditions to select the "best" solution for the underdetermined λ. Specifically, we choose the value for λ that minimizes $|\lambda|_2$ in the redundant situation, and the value which minimizes the 2-norm of the residual error in equation (11.17) if the constraints are (slightly) inconsistent.

"extra" equations. These equations define the position and velocity constraint manifolds on which the solutions $q(t)$ and $u(t)$ are expected to lie (that is, the values of q and u should always satisfy those equations). If equations (11.16) and (11.17) could be integrated perfectly, the solutions would indeed stay on the manifolds since they start out that way and equations (11.16) and (11.17) satisfy the constraint derivatives. However, truncation error inherent in methods for approximate numerical integration allows the solution to drift away from the manifolds. The "extra" equations can be employed rigorously to eliminate this drift, and in fact improve the solution overall, using the method of *coordinate projection*[9] to be discussed below. But first we have to take a short detour to discuss scaling of variables, since that will be required to make rigorous notions like "improving the solution" or "making a small change."

9.1 Scaling, tolerance, and accuracy

A multibody system is modeled using a set of state variables, and a set of differential and algebraic equations that those variables must satisfy. There are many mathematically equivalent ways to model the same system, and some of the modeling choices to be made are arbitrary. Some examples are: choice of units for various quantities; choice of which quantities to treat as independent and which dependent; and choice of which body is to serve as the base body for a chain. However, the resulting physically equivalent models are not *numerically* equivalent so can affect the actual solutions we obtain when doing computations, which are necessarily approximate. Such computations involve strong tradeoffs between CPU time and accuracy, so are typically performed to a level of accuracy chosen by the user based on his or her requirements. The goal of scaling is to ensure that an accuracy specification (e.g., "1% accuracy") can be applied in a physically meaningful way so that the behavior of a study is not dominated by arbitrary modeling choices. That is, we would like a given accuracy specification to yield the same physical results for all the physically equivalent models, regardless of any arbitrary choices that may have been made during construction of those models.

There are two ways in which arbitrary modeling choices interact with accuracy requirements. These are: (1) *scaling* of system state variables, and (2) *tolerance* for errors in the algebraic constraints. Our goal is to be able to determine a physically meaningful "unit change" to each state variable, and a physically meaningful "unit error" for each algebraic constraint. Then when solving the system equations we can define "accuracy" to mean calculation of state variables to some fraction of that unit change, and satisfaction of algebraic equations to some fraction of that unit error. We deal with multiple variables and equations by defining a scalar norm representing "overall change" and "overall error" and then requiring our computations to maintain those norms at or below the requested accuracy.

9.1.1 Scaling

An important practical consideration for any multibody formulation is that the state variables y={q,u,z} vary widely in scaling, by which we mean the degree to which a change in the numerical value of a state variable affects a physically meaningful quantity. There are several causes for the uneven scaling of state variables. To begin with, they are expressed in different units—q's are typically lengths, angles, or quaternions; u's are typically length/time or angle/time; z's can be anything at all. Scaling differences are even more pronounced in internal coordinate formulations like Simbody's, since the effect of a state variable depends strongly on its position in the multibody tree. A change Δq to an angular coordinate near the system base will have a much larger effect (on almost anything you might care to measure) than the identical change made to a coordinate which rotates only a lone terminal body.

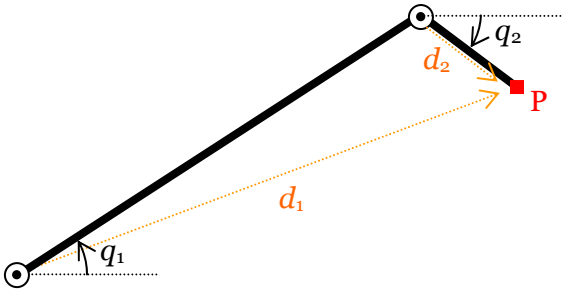


Figure 12: scaling of q1 and q2 are very different with respect to the position of the end point P.

Figure 12 depicts this situation. If we hope to achieve a given level of accuracy with regard to the positioning of the end point P marked with a red square, we need to calculate q_1 more accurately than q_2. For example, say d_1=10 nm and d_2=0.5 nm. Then an error ε=0.1 radians in q_1 induces an error of $d_1\varepsilon$=1 nm in P's location, while that same error in q_2 induces only $d_2\varepsilon$=0.05 nm of positioning error. In this situation we say that state q_1 has more *weight* than q_2 with respect to the location of P. Scaling is thus the process of assigning a numerical weight $w_i \geq 0$ to each state variable y_i. States which are "more important" get a numerically larger weight, so that we know for each state variable what size perturbation would produce a unit change in the physical system.

Even in the simple example of Figure 12 it is clear that the relative and absolute weighting of state variables are not constant but change as a function of system configuration. Thus weights may need to be recalculated periodically as a system moves during a study. Fortunately, in practice scaling does not need to be done perfectly to yield substantial improvements over unscaled variables. That permits us to treat weights as constants in the discussion to follow; in practice they are updated only occasionally.

Say we want to make the "smallest" state change Δy that will satisfy some physically-meaningful condition, perhaps to reposition point P to a specified nearby location. It is unlikely that we mean "smallest" in the sense of a norm like $\|\Delta y\|_2$ on the unscaled numerical values of the state variables, since those numerical values to some degree reflect arbitrary modeling choices as discussed above. More likely we mean something like the smallest motion of the bodies, least change in energy, or some other physical consideration. This can be achieved by defining "smallest" in terms of a weighted norm $\|\Delta y\|_W$, where the weights on each individual state entry reflect its scaling with respect to the physical parameter of interest. For this purpose we define a diagonal weighting matrix **W**, where the i^{th} diagonal element is w_i, the "unit weight" of state variable y_i. For example, referring again to Figure 12, if we want to scale by the geometric consequences of the state variables on P's location we could use w_1=weight(q_1)=d_1 and w_2=weight(q_2)=d_2. Then we would define W as follows:

$$\mathbf{W} \equiv \begin{bmatrix} d_1 & 0 \\ 0 & d_2 \end{bmatrix}$$

Then we define the weighted norm $\|\Delta \mathrm{y}\|_{\mathrm{W}} \equiv \|\mathbf{W}\Delta \mathrm{y}\|_2$. The ideal weighting matrix would be such that, for a given scalar property of interest $P(\mathrm{y})$, we would have

$$\frac{\partial P(\mathrm{y})}{\partial y_i} = w_i, \forall i .$$

Another way to think of this is to imagine an alternate set of state variables $\hat{\mathrm{y}} \equiv \mathbf{W}\mathrm{y}$ such that $\partial P(\mathrm{y})/\partial \hat{\mathrm{y}}_i = 1$ for all i (with $\mathrm{dy}=\mathbf{W}^{-1}\mathrm{d}\hat{\mathrm{y}}$). That is, a unit change in the numerical value of *any* state variable in $\hat{\mathrm{y}}$ produces a unit change of the physical quantity *P*.

We will primarily make use of the related RMS norm

$$\|\Delta \mathrm{y}\|_{\mathrm{WRMS}} \equiv \frac{1}{\sqrt{n_{\mathrm{y}}}}\|\Delta \mathrm{y}\|_{\mathrm{W}}$$

where n_{y} is the number of state variables, because the RMS norm does not grow with the problem size. Now if we let ε_{y} represent the vector of n_{y} unweighted error estimates introduced by a step in the numerical solution for $\mathrm{y}(t)$, we can provide a precise meaning for the notion "solve for $\mathrm{y}(t)$ to an accuracy α": $\|\varepsilon_{\mathrm{y}}\|_{\mathrm{WRMS}} \le \alpha$ at each step during the study.

We will assume below that an n_{y}xn_{y} weighting matrix $\mathbf{W}=\mathrm{diag}(w_i)$ is available for the state y. We'll defer discussion of how to compute **W** until later, however we note that the weightings on q's cannot be independent of the weightings on their corresponding u's, since we know that $\dot{q} = \mathbf{Q}(q)u$ (and $u = \mathbf{Q}^{+}(q)\dot{q}$). We can compute weighted least squares configuration changes about a nominal configuration using "instant coordinates" $\hat{q}$, which have the property that $\dot{\hat{q}} = u$. Then the corresponding change to the real q's is $\Delta q = \mathbf{Q}\,\Delta\hat{q}$.

Weights are thus provided on the mobilities u, which are physical quantities, rather than on the generalized coordinates q which can be chosen somewhat

arbitrarily. So we work only with the $n_u x n_u$ diagonal weighting matrix $\mathbf{W}_u$; there is no separate $\mathbf{W}_q$. Note that if there are auxiliary state variables z they will have independent weights $\mathbf{W}_z$.

9.1.2 Tolerance

The nonlinear algebraic equations defining the system, such as equations (11.13) and (11.14), cannot be solved exactly by a numerical computation. Instead, they will be met with some residual error. We would like to keep that error below a specified "tolerance" level during a study. As with the state variable scaling problem above, we have to deal with the issue that there are many separate algebraic equations, and the errors they produce will not be measured in the same units. This is especially important when mixing position (holonomic) and velocity (nonholonomic) constraints, since velocity constraint errors need to be in units comparable to the time derivatives of the position constraint errors. Also, if any acceleration-only constraints are provided their errors must be in units comparable to the 2nd time derivative of the holonomic constraint errors.

The formulation used by Simbody ensures that the acceleration-level constraints are solved to machine precision at the same time we solve for the accelerations. So our numerical integration methods do not need to deal with acceleration constraint tolerances; we'll just take what we get from solving system (11.17). However, when there are redundant constraints the tolerances can affect how the constraint forces are distributed, although the accelerations are still unique. In any case we do need to actively control the errors in velocity, position, and quaternion normalization constraints. As with state variables, we want to be able to provide a consistent physical meaning for a statement like "solve the constraint equations to 1% accuracy."

To do this we define a set of tolerances $t_i>0$, one for each of the m_p position (holonomic) and m_v velocity (nonholonomic) and m_a acceleration-only constraint equations, and define a diagonal constraint weighting matrix $\mathbf{T}$ whose i^{th} diagonal element is $1/t_i$. Each t_i should represent the violation that is to be considered a "unit violation" of the i^{th} constraint ("unit" doesn't necessarily mean "small"). Note that "tolerance" has the inverse sense to "weight"—while a larger weight means "more important" a larger tolerance

means "less important," which is why we invert them in **T** to create weights. We sometimes refer to these reciprocal tolerances "constraint weights." If we need to refer separately to the position and velocity tolerances, we'll consider **T** to be partitioned into diagonal submatrices $\mathbf{T}=\{\mathbf{T}_p,\mathbf{T}_v,\mathbf{T}_a\}$. Unlike relative weights of state variables, which can change as the state variable values change, we expect **T** to remain fixed once specified since tolerances are absolute quantities. Now define ε_c as the vector containing the current, unweighted error for each constraint equation. We can calculate a norm like $\|\mathbf{T}\varepsilon_c\|_2$ which treats all constraint errors uniformly. We'll call this the tolerance norm and write it $\|\varepsilon_c\|_T$. In practice we will use the RMS norm $\|\varepsilon_c\|_{TRMS} = \frac{1}{\sqrt{n_c}}\|\varepsilon_c\|_T$ to remove effects due just to problem size, and define the phrase "meeting tolerance to accuracy α" to mean $\|\varepsilon_c\|_{TRMS} \le \alpha$.

To summarize, we now have a way to define what is meant by solving a multibody system to a given accuracy, say α=0.1%. We will have defined a locally-constant weighting matrix **W** on changes to the state variables u and z (and implying a weighting on changes to q) and a constant reciprocal tolerance matrix **T** on the absolute errors in the constraint equations. **W** defines a "unit change" for each state variable, and **T** defines a "unit error" for each constraint equation. Then we have solved a trajectory to an accuracy α =0.1% (for example) when both

$$\|\varepsilon_y\|_{WRMS} \le 0.001 \text{ and } \|\varepsilon_c\|_{TRMS} \le 0.001$$

hold for each step of the solution.

Although *constraint* accuracy is maintained throughout a simulation, it is important to emphasize that we define accuracy of the *state variables* as a *local* phenomenon. Many multibody systems are inherently chaotic, meaning that their long term behavior is arbitrarily sensitive to initial conditions and numerical errors and hence not predictable. Only local measures of accuracy make sense for such systems. One may think of this as ensuring that the simulation accurately simulates *some* system which is very similar to the one under study. Without such accuracy control there is no guarantee that *any* such system is being simulated.

9.2 Coordinate projection

Given an arbitrary value for the state variables, some or all of the constraint equations may fail to be satisfied. Since accelerations are computed quantities rather than states, we can always calculate them to satisfy the acceleration constraint equations. However, since t, q, and u are independent states we may find position and velocity constraints are not satisfied. In cases where the equations are expected to be *arbitrarily* far from being satisfied (typically prior to the start of a study), we may need special analyses to attempt to find values which satisfy the constraints. However, during a dynamic simulation it will typically be the case that the constraints will *almost* be satisfied, meaning that q and u just need to be "cleaned up" a little. This cleaning up process can be thought of as taking state variables which have left the required constraint manifold and *projecting* them back to the manifold via the shortest path (smallest change in a weighted norm) we can make.

We define

$$\varepsilon_{\mathrm{n}}(q) = \mathbf{n}(q) = \begin{pmatrix} |\mathrm{n}_1(q)| - 1 \\ \vdots \\ |\mathrm{n}_{nquat}(q)| - 1 \end{pmatrix} \tag{9.1}$$

$$\varepsilon_{\mathrm{p}}(q) = \mathbf{p}(\mathrm{t}, q) \tag{9.2}$$

$$\varepsilon_{\mathrm{v}}(u) = \begin{bmatrix} \dot{\mathbf{p}} \\ \mathbf{v} \end{bmatrix}(\mathrm{t}, \mathrm{q}, u) \tag{9.3}$$

$$\varepsilon_{\mathrm{a}}(\dot{u}) = \begin{bmatrix} \ddot{\mathbf{p}} \\ \dot{\mathbf{v}} \\ \mathbf{a} \end{bmatrix}(\mathrm{t}, \mathrm{q}, \mathrm{u}, \dot{u}) \tag{9.4}$$

where $\mathrm{t}, \mathrm{q}, \mathrm{u}$ are fixed at their current values. Note that these are unweighted errors; ε_{p} and ε_{v} need to be normalized using the tolerance matrix **T** discussed in section 9.1.2 above. Then we would like to find the smallest change to q that will drive ε_{p} to 0, and the smallest change to u that will drive ε_{v} to 0. Those "smallest" changes correspond to a least squares projection in the *weighted* (W norm) direction, normal to the constraint manifold, for which a theorem given in reference 9 guarantees that this projection also *improves*

the solution to the differential equations. See section 9.1.1 for a discussion of the W norm. ε_a is satisfied exactly when we solve equation (11.17), and ε_n is always satisfied simply by normalizing the quaternions $\mathrm{n}_i \subset q$, which is a 2-norm projection that can be done separately from everything else.

The projection equations are underdetermined, nonlinear equations, but we expect to be close to a solution so they can be solved efficiently with Newton iteration or similar methods. For example, the full Newton steps would be

$$\bar{\mathbf{P}}(q^{(i)})_{\Delta}\hat{q}_{WLS} = \mathbf{T}_{\mathrm{p}}\varepsilon_{\mathrm{p}}(q^{(i)}), \quad q^{(i+1)} = q^{(i)} - \mathbf{Q}\mathbf{W}_{\mathrm{u}}^{-1}{}_{\Delta}\hat{q}_{WLS} \tag{9.5}$$

$$\forall \mathrm{n}_k \subset q: \ \mathrm{n}_k^{\mathrm{final}} = \mathrm{n}_k^{(last)} \big/ |\, \mathrm{n}_k^{(last)} \,| \tag{9.6}$$

$$\bar{\mathbf{V}}(q^{\mathrm{final}})_{\Delta}u_{WLS} = \begin{bmatrix} \mathbf{T}_{\mathrm{p}}\varepsilon_{\mathrm{p}} \\ \mathbf{T}_{\mathrm{v}}\varepsilon_{\mathrm{v}} \end{bmatrix}(u^{(i)}), \quad u^{(i+1)} = u^{(i)} - \mathbf{W}_{\mathrm{u}}^{-1}{}_{\Delta}\hat{u}_{WLS} \tag{9.7}$$

where

$$\bar{\mathbf{P}}(q) = \mathbf{T}_{\mathrm{p}}\, \mathbf{P}(q)\, \mathbf{W}_{\mathrm{u}}^{-1}$$

and

$$\bar{\mathbf{V}}(q) = \begin{bmatrix} \mathbf{T}_{\mathrm{p}}\mathbf{P}(q)\mathbf{W}_{\mathrm{u}}^{-1} \\ \mathbf{T}_{\mathrm{v}}\mathbf{V}(q)\mathbf{W}_{\mathrm{u}}^{-1} \end{bmatrix}.$$

We iterate (9.5) until we have calculated a final value $q^{(last)}$ that satisfies the holonomic constraint equations (9.2) to within a specified tolerance, then using equation (9.6) project the quaternions in $q^{(last)}$ via their normalization constraints (9.1). That gives us q^{final} which satisfies all the constraints (9.1) and (9.2). We then iterate (9.7) with $\bar{\mathbf{V}}$ calculated at q^{final} while solving for the final velocity value u^{final} which satisfies the velocity constraints (9.3). Note that we must perform a least squares solution to the linear system at each iteration, and that the diagonal weighting matrices $\mathbf{W}_{\mathrm{u}}$, $\mathbf{T}_{\mathrm{p}}$, and $\mathbf{T}_{\mathrm{v}}$ are constant during the iteration.

Normalizing a quaternion as in equation (9.6) is the least squares projection of the four-dimensional quaternion onto its constraint manifold, a three-dimensional sphere of unit radius. However, quaternion projection is done in the *unweighted* norm since it is a constraint on the numerical values of the quaternion elements unrelated to the physical effect of those elements. By

construction, the physical effect of a change in the *length* of a quaternion in Simbody is zero. Note also that there are no velocity or acceleration constraints corresponding to the quaternion normalization constraint, because those constraints are satisfied exactly by the quaternion derivatives we calculate from the generalized speeds *u*.

A very similar problem arises when we have a vector in the *q* or *u* basis, and we would like to remove the component of that vector which is normal to the constraint manifold, in the weighted norm. For example, when an integrator has computed a pre-projection absolute error estimate vector $\varepsilon_y = \{\varepsilon_q, \varepsilon_u, \varepsilon_z\}$ in its computation of state variables y={q,u,z}, we know that performing the above constraint projection will remove the component of the error in the weighted constraint-normal direction (for proof, see ref. 9 and ref. 6, §3.8.2), and also the component of error along the length of quaternions. So we can now reduce that error estimate by subtracting out any component it might have had in the directions we just fixed, which may allow us to take a bigger step. In that case the projections are

$$\bar{\mathbf{P}}(q^{\text{final}})\varepsilon_{wq}^{\perp} = \bar{\mathbf{P}}(q^{\text{final}})\mathbf{W}_u\mathbf{Q}^{+}\varepsilon_q, \quad \bar{\varepsilon}_q = \varepsilon_q - \mathbf{Q}\mathbf{W}_u^{-1}\varepsilon_{wq}^{\perp} \tag{9.8}$$

$$\hat{\varepsilon}_{n_k} = \bar{\varepsilon}_{n_k} - (\bar{\varepsilon}_{n_k} \cdot n_k^{\text{final}})n_k^{\text{final}} \tag{9.9}$$

$$\bar{\mathbf{V}}(q^{\text{final}})\varepsilon_{wu}^{\perp} = \bar{\mathbf{V}}(q^{\text{final}})\mathbf{W}_u\varepsilon_u, \quad \hat{\varepsilon}_u = \varepsilon_u - \mathbf{W}_u^{-1}\varepsilon_{wu}^{\perp} \tag{9.10}$$

Again we need to find least-squares solutions to the underdetermined systems (9.8) and (9.10). Then we set $\hat{\varepsilon}_y = \{\hat{\varepsilon}_q, \hat{\varepsilon}_u, \varepsilon_z\}$ as the new (absolute, unweighted) error estimate. Note that these use the same (final) iteration matrices as above with a different right hand side. Equations (9.8) and (9.10) are linear systems so no iteration is needed. After this projection the integrator should use the revised estimate $\hat{\varepsilon}_y$ as its error estimate instead of the original estimate ε_y (using the **W** norm).

We can use a pseudoinverse to find the least squares solution at each step. The pseudoinverse $\mathbf{A}^+$ of an *m*x*n* matrix **A**, with *m*<=*n* and full row rank (i.e. *rank*(**A**)=*m*) is given by $\mathbf{A}^+=\mathbf{A}^T(\mathbf{A}\mathbf{A}^T)^{-1}$, although computing A^+ that way can be numerically inaccurate. Using an SVD or faster complete orthogonal

factorization (QTZ) we can compute a numerically well-conditioned pseudoinverse even in the case of redundant constraints, i.e., $rank(\mathbf{A})<m$.

Looking now at the weighted holonomic position constraint iteration matrix $\bar{\mathbf{P}} = \mathbf{T}_p\mathbf{P}\mathbf{W}_u^{-1}$, we see that the pseudo inverse we need is

$$(\mathbf{T}_p\mathbf{P}\mathbf{W}_u^{-1})^+$$

The corresponding velocity constraint projection is

$$\bar{\mathbf{V}}^+ = \begin{bmatrix} \mathbf{T}_p\mathbf{P}\mathbf{W}_u^{-1} \\ \mathbf{T}_v\mathbf{V}\mathbf{W}_u^{-1} \end{bmatrix}^+$$

When there are no non-holonomic constraints, the velocity projection is just

$$\bar{\mathbf{V}}^+ = (\mathbf{T}_p\mathbf{P}\mathbf{W}_u^{-1})^+$$

The constraint projections can be performed sequentially (for proof, see ref. 6, §3.8.3). First, with t fixed at $\hat{t}$ we must find some $q = \bar{q}$ that satisfies the holonomic constraint equations to within a specified tolerance. Then we normalize the quaternions in $\bar{q}$ (which by construction cannot affect any of the holonomic constraints) and call the result $\hat{q}$. After that we freeze q at $\hat{q}$ and proceed to find u that satisfies the velocity constraint equations to within a specified tolerance. Note that the holonomic velocity constraint equations (i.e., first time derivatives of the holonomic constraint equations) and nonholonomic constraint equations must be dealt with simultaneously since they can be coupled.

9.2.1 What about zero-weight state variables? [TBD]

When a weight w_i=0 exactly, we can't allow it in the above scheme because we depend on $\mathbf{W}$ to be invertible. (This isn't a problem for tolerances; they are required to be positive.) For numerical reasons in practice we will set weights to zero whenever they fall below some threshold, typically expressed as some fraction of the largest weight. A zero weighting means that changes to that state variable are "free" in the sense that they have no effect on any physical quantities of interest. Such variables are never considered in the integrator's step size choice; any error estimates for them are acceptable.

It is possible that zero-weight state variables can be used in solving the constraint equations. If so, this is a very good thing since the constraints can be solved without affecting the dynamics. In that case the state variables are really algebraic variables and the integrator is merely supplying a good initial guess for use in solving the algebraic equations.

Sherm speculation: split the system into two smaller ones. Solve the constraints as best as can be done using only the zero-weighted variables. Then freeze those variables and use the new values to calculate the weighted constraint matrix for the other variables; solve those in the normal way to get rid of the remaining error. Note that the reduced systems may not have full row rank even if the original ones did, because we might have deleted the only columns of **P** with non-zeroes in them, or created some other kind of linear dependency.

How should these systems be iterated? Should we revisit the free ones after changing the expensive ones?

9.3 Simplified equations

For use with a generic coordinate projection integrator, the Simbody equations can be viewed in the following simplified form:

$$\textit{differential eqns.} \qquad \dot{y} = f(t, y) \tag{9.11}$$

$$\textit{algebraic eqns.} \qquad c(t, y) = 0 \tag{9.12}$$

$$\textit{initial conditions} \qquad c(t_0, y_0) = 0 \tag{9.13}$$

with the guarantee that equation (9.11) is solved in such a way that any new constraints introduced by time-differentiating equation (9.12) are satisfied automatically; that is, $\dot{c}(t, y, f(t, y)) = 0$ whenever equations (9.11) and (9.12) are satisfied. There is an n_yxn_y diagonal weighting matrix **W** and an n_cxn_c diagonal tolerance matrix **T**, and corresponding norms as discussed in section 9.1 above. To solve this system to accuracy α, the following two conditions must be satisfied at each integration step:

$$\left\| \varepsilon_y \right\|_{\text{WRMS}} \le \alpha \tag{9.14}$$

$$\left\| c(t, y) \right\|_{\text{TRMS}} \le \alpha \tag{9.15}$$

where the ε_y are the post-projection local state errors introduced by an integration step. When conditions (9.14) and (9.15) are met, the integrator can accept the step.

9.4 Modal analysis and implicit integration

In this section we discuss the related needs of modal analysis (that is, normal modes in internal coordinates) and implicit integration. Both of these require that the system equations of motion be differentiated with respect to the generalized coordinates and speeds. That is we want to calculate the dynamic, internal coordinate Jacobian

$$\mathbf{J} = \begin{bmatrix} \mathbf{J}_{qq} & \mathbf{J}_{qu} & \mathbf{J}_{qz} \\ \mathbf{J}_{uq} & \mathbf{J}_{uu} & \mathbf{J}_{uz} \\ \mathbf{J}_{zq} & \mathbf{J}_{zu} & \mathbf{J}_{zz} \end{bmatrix} = \begin{bmatrix} \partial \dot{q}/\partial q & \partial \dot{q}/\partial u & \partial \dot{q}/\partial z \\ \partial \dot{u}/\partial q & \partial \dot{u}/\partial u & \partial \dot{u}/\partial z \\ \partial \dot{z}/\partial q & \partial \dot{z}/\partial u & \partial \dot{z}/\partial z \end{bmatrix} \tag{5}$$

Modal analysis is typically done with all speeds set to zero, so only the submatrix $\mathbf{J}_{uq}$ is of interest. If q is such that the system is stable (at a local energy minimum), then the eigenvalues of this matrix are the normal modes of the system about that equilibrium point and the corresponding eigenvectors are the modal basis (that is, they represent the coordinated motion involved in each of the normal modes).

Given the system equations of motion, note that one can easily obtain an approximation to **J** by perturbing the state variables (this is called a finite difference approximation to **J**). Simbody 1.0 should, at a minimum, support that method. However, it is both inaccurate and extremely expensive to compute. Finite differencing loses about half the available precision, and requires O(n) calculations of the system accelerations to form an $n \times n$ matrix. In molecular dynamics straightforward force calculations are typically O(n^2), so this can mean the Jacobian calculation is a prohibitive O(n^3). In any case the force calculations are very expensive and doing O(n) of them to get a half-accurate Jacobian is not a very good deal. Analytical methods exist which allow $\mathbf{J}_{uq}$ to be calculated from the spatial force derivatives (energy Hessian),

to full accuracy and in much less time, with the total calculation being $O(n^2)$. Note that this is within a constant factor of optimal for filling in a matrix with n^2 elements.

If possible, Simbody 1.0 should include a good modern method for calculating **J** analytically, but if that can't be done it should at least provide an interface designed to support such a calculation in the next release.

For implicit integration the required matrix is the full **J** (with nonzero velocities) rather than just $\mathbf{J}_{uq}$. However, that is not much worse. Calculating the $\mathbf{J}_{uq}$ submatrix is by far the most difficult part since it involves the Hessian of the potential energy and (formally) the partial derivatives of the mass matrix *inverse* with respect to the q's.

9.5 Root finding and optimization

The needed computations here depend on the kind of problems being solved. They typically require Jacobians of various calculations with respect to the generalized coordinates and speeds. **J** as defined above can be very useful for minimizations involving search for equilibria. For satisfying constraints, the partial derivatives of the constraint equations (11.13) and (11.14) are required. Simbody 1.0 should provide access to these matrices, which are needed internally anyway.

Root finding problems can be difficult when the coordinates are constrained, so it is convenient to define a new set of fully-independent coordinates. In particular, Simbody 1.0 should do this at least for the case where the only constraints are the quaternion normalization conditions. It is easy to create a localized 3-coordinate representation for orientation about a current set of q's which will remain valid even for large perturbations. Reduced sets of coordinates for more general constraints may have limited validity ranges and have to be recalculated periodically during a root finding or optimization run.

10 Simbody Reporters reference guide

TODO

10.1 VTK Reporter

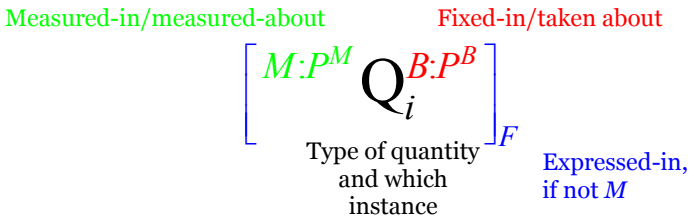
TODO

11 Simbody theory

Some readers may find this more-technical discussion helpful in defining the specific approach we have in mind; others will find it confusing and perhaps somewhat irrelevant and are invited to skip it!

11.1 Notation for multibody theory

When discussing physical quantities that arise in multibody dynamics, we must be very precise. We need to describe exactly what quantity we mean, how it was measured, and in what coordinate system we have decided to express the result. In the worst case, this can result in a complicated forest of super- and subscripts, however there are defaults which cover most cases. Here is the worst-case, fully-decorated symbol:



The type of quantity (the central black symbol) is the only required piece. The right subscript picks out a particular instance, so B_i might be the i^{th} body. Here are the symbols we conventionally use for particular quantities:

G	`G`	The unique Ground body, and the inertial (Cartesian) reference frame fixed to it. Technically this is a MobilizedBody, although it doesn't do a lot of moving.
B B_i	`B` `Bi`	The mobilized body under discussion. The same symbol is used to mean the body frame associated with that body.
P P_B	`P` `Pb`	The parent (inboard) body of the mobilized body under discussion, or the parent of a particular body *B*.

M M_B	`M` `Mb`	The mobilized frame for the body under discussion, or for a particular mobilized body *B*. The M frame is fixed to the body, and is related to the body frame by the constant transform ${}^{B}X^{M}$.
F F_B	`F` `Fb`	The fixed (reference) frame for the mobilized body under discussion, or for a particular body *B*. The F frame is fixed to the *parent* body P, and is related to the parent's body frame by the constant transform ${}^{P}X^{F}$.
O_F	`Of`	The origin point of some frame *F*.
C_B	`Cb`	The mass center (a point) of some body *B*. By default this is the vector from the B origin to the mass center, expressed in B.
m_B	`mb`	The mass of some body *B*.
I_B	`Ib`	The inertia of body B. By default this is taken about the B origin and expressed in the B frame.
${}^{A}R^{B}$	`R_AB`	The 3x3 rotation matrix whose columns are the B frame's axes expressed in the A frame.
${}^{R}p^{S}$ ${}^{A}p^{B}$	`p_RS` `p_AB`	The translation vector from point R to point S, expressed in the same frame as R. If R or S are the names of bodies or coordinate frames, the origins of those frames are used as the points; that is, ${}^{A}p^{B} = {}^{O_A}p^{O_B}$.
${}^{A}X^{B}$	`X_AB`	The *spatial transform* (rotation and translation) expressing frame B in frame A. ${}^{A}X^{B} = \left({}^{A}R^{B} \mid {}^{A}p^{B}\right)$.
${}^{A}V^{B}$ ${}^{A}V^{B:Q}$	`V_AB` `V_AQ`	The *spatial velocity* of frame B in A. This includes the angular velocity of B in A and the linear velocity of O_B in A as a stacked pair of vectors expressed in A. A different point *Q* (fixed in B) can be specified as shown in which case the linear velocity is of *Q* in A rather than of O_B. B:Q can be considered a coordinate frame parallel to B but with its origin shifted to Q.
MORE TODO		

The right superscript defines the physical quantity by specifying the frame to which a physical quantity is attached, and optionally a point other than the frame's origin to which the physical quantity is referred. The inertia of body B, taken about B's origin would be I_B, but if the inertia were instead taken about B's center of mass point C_B, the symbol would be $I_B^{C_B}$.

The left superscript specifies how we are to take the measurement of the physical quantity. Typically this is just a frame F, so that the measurement is done with respect to that frame's coordinate system and from the frame's origin, and by default the resulting measure numbers are expressed in F. However, a "measured about" point can be provided which is different from the origin. As an example, if body B's center of mass point C_{B} is to be measured in the local frame of another body A, we would write ${}^{\mathrm{A}}p^{C_{\mathrm{B}}}$ (a vector from body A's origin to body B's center of mass point). If instead we want the vector from A's center of mass to B's, the symbol would be ${}^{\mathrm{A}:C_{\mathrm{A}}}p^{C_{\mathrm{B}}}$ or more simply ${}^{C_{\mathrm{A}}}p^{C_{\mathrm{B}}}$ where the expressed-in frame A is inferred from C_{A}. In both cases the vector would be expressed in A. If instead it was to be expressed in the ground frame G, we would write $\left[{}^{C_{\mathrm{A}}}p^{C_{\mathrm{B}}}\right]_G = {}^{G}R^{A} \cdot {}^{C_{\mathrm{A}}}p^{C_{\mathrm{B}}}$.

Time derivatives with respect to the expressed-in frame are denoted with an overdot. For example

$$\begin{aligned}\left[{}^{C_{\mathrm{A}}}\dot{p}^{C_{\mathrm{B}}}\right]_G &\triangleq \frac{{}^{G}d}{dt}\left(\left[{}^{C_{\mathrm{A}}}p^{C_{\mathrm{B}}}\right]_G\right)\\ &= \left[{}^{C_{\mathrm{A}}}\dot{p}^{C_{\mathrm{B}}}\right]_G + \left[{}^{C_{\mathrm{A}}}p^{C_{\mathrm{B}}}\right]_{\dot{G}}\\ &= \left[{}^{C_{\mathrm{A}}}\dot{p}^{C_{\mathrm{B}}}\right]_G + {}^{G}\omega^{A} \times \left[{}^{C_{\mathrm{A}}}p^{C_{\mathrm{B}}}\right]_G\end{aligned}$$

where for any quantity Q expressed in frame A and an arbitrary frame B:

$$\begin{aligned}\left[{}^{A}Q\right]_B &\triangleq {}^{B}R^{A} \cdot {}^{A}Q\\ \left[{}^{A}Q\right]_{\dot{B}} &\triangleq {}^{B}\dot{R}^{A} \cdot {}^{A}Q = ({}^{B}\omega_{\times}^{A} \cdot {}^{B}R^{A}) \cdot {}^{A}Q\\ &= {}^{B}\omega_{\times}^{A} \cdot \left[{}^{A}Q\right]_B\\ {}^{A}\dot{Q} &\triangleq \frac{{}^{A}d}{dt}{}^{A}Q\\ \left[{}^{A}\dot{Q}\right]_B &\triangleq \frac{{}^{B}d}{dt}\left[{}^{A}Q\right]_B = \left[{}^{A}\dot{Q}\right]_B + \left[{}^{A}Q\right]_{\dot{B}}\\ &= \left[{}^{A}\dot{Q}\right]_B + {}^{B}\omega_{\times}^{A} \cdot \left[{}^{A}Q\right]_B\end{aligned}$$

11.2 Components of a multibody model

All mass and geometric features of the system are associated with the bodies (with "Ground" viewed as an immobile body). Each body is associated with a unique *mobilizer*, which defines how that body may move relative to its parent body. Thus large scale motion is permitted only by mobilizers, whose *mobilities* (degrees of freedom) define *generalized coordinates* describing the system configuration in terms of relative translations and orientations of the bodies they interconnect, and *generalized speeds* describing the relative motion (velocities) of those bodies. Generalized coordinates are sometimes referred to as "internal coordinates," "relative coordinates," or "torsion coordinates."

Forces (more properly *generalized forces*) include both forces and moments (torques) and may be applied to bodies or directly along the mobility coordinates.

Constraints express algebraic restrictions on the allowed values of the generalized coordinates and speeds. One may reasonably think of constraints as "infinitely strong" forces. We distinguish two sets of constraints: *topological constraints* which are always present, and *intermittent constraints*, which may be added or removed as needed.

As a practical matter, we consider bodies, mobilizers, and topological constraints to be the fundamental features of a multibody system, together defining the system's *topology* which is invariant. A change in the number of bodies, connectivity or types of mobilizers, or connectivity or types of topological constraints results in a new multibody system. Forces and intermittent constraints, on the other hand, can be added, changed, and removed from a multibody system without changing its identity. This does not imply that topology must remain fixed during an investigation, just that a topology change is a more significant operation than a change in forces or non-topological constraints.

11.3 A comment on deformable (flexible) bodies

In general, the bodies of a multibody system do not have to be rigid. It is sometimes desirable to allow the bodies themselves to undergo small internal

motions, called *deformations*. These add a new set of independent coordinates to the overall system coordinates and speeds, but we distinguish them from the generalized coordinates and generalized speeds introduced by mobilizers and refer to them instead as *deformation coordinates* and *deformation rates*. Various techniques can be used to determine the appropriate representation of deformable bodies. Such bodies can be used, for example, to supply "ring pucker" coordinates for molecules rather than modeling the mobility of every bond individually. Or, the techniques of structural mechanics can be used to aggregate large nearly-rigid subsystems into deformable bodies with "assumed mode" linear deformations.

We will not support deformable bodies in Simbody 1.0, but will allow for adding them in the future (e.g. by not building in an assumption that a body's center of mass is in a fixed location in the body frame). In the meanwhile it is always possible to model body flexibility by partitioning the body into mobilizer-connected rigid bodies, with internal forces and constraints modeling the deformation behavior.

11.4 Kinematics

Kinematics is the study of motion in the absence of mass and force effects. In practice, it refers to the mapping between generalized coordinates and speeds and their spatial counterparts. For example, given values for the generalized coordinates, one should be able to obtain (cheaply) positions and orientations for bodies and spatial (Cartesian) locations of any stations (e.g. atoms). In the other direction, one should be able to solve for the set of generalized coordinates and speeds which most closely reproduces a given set of spatial configurations and velocities.

Kinematic results available in Simbody 1.0 should be sufficient to permit the solution of kinematic problems such as finding the set of generalized coordinates which best approximates a given set of spatial locations. Such problems arise, for example, when fitting a reduced-coordinate molecular model to a set of atom positions determined with X-ray crystallography. More generally, there is a broad assortment of useful initial condition analyses which must be performed prior to the start of a dynamic analysis, and these are based on kinematic calculations.

11.5 Dynamics

Dynamics refers to the relationship between forces and motion. There are two flavors: forward dynamics, in which forces are known and motion calculated, and inverse dynamics where motion is known and forces are to be calculated. Various combinations of known and unknown forces and motions are possible. Simbody supports both of these operations and provide access to the basic O(n) operators that manipulate the associated quantities.

Note that Simbody itself focuses on instantaneous dynamics, that is, the relationship between forces and accelerations at a particular time and state. This capability is designed to be used in conjunction with numerical methods, primarily numerical integrators, to advance the time and state. These numerical methods exist independently of Simbody, however good methods can be very difficult to construct, so the SimTK Core TimeStepper and Integrator classes have been provided which are designed to work efficiently with Simbody.

11.6 Equations of motion

Given the above description, we can write down the system of equations defining a multibody system. A few conventions: We use n and subscripted n's to count quantities related to coordinates (mobilities or degrees of freedom) and m and subscripted m's to count constraint equations. We use overdot to represent differentiation with respect to time. We use a right superscript to denote a quantity which applies only to a particular body or its mobilizer.

The equations of motion will be written in terms of the set of n generalized speeds $u = \bigcup_{\mathrm{B}} u^{\mathrm{B}}$, and the n_{q} generalized coordinates $q = \bigcup_{\mathrm{B}} q^{\mathrm{B}}$, where u^{B} and q^{B} are the (disjoint for each B) sets of n^{B} speeds and $n_{\mathrm{q}}^{\mathrm{B}}$ coordinates which

arise from the presence of body B's mobilizer.* Thus we have for the total number of u's and q's $n_u = n = \sum_B n^B$ and $n_q = \sum_B n_q^B$. Typically there will also be a set of differential equations associated with force models which must be integrated along with the matter model's generalized coordinates and speeds; we'll call these n_z auxiliary state variables z. In general a system will also include discrete-time (difference) equations and associated discrete states but we'll only consider the continuous system here.

The total number n of mobilities in a multibody system is just the sum of the bodies' individual mobilities, that is $n = \sum_B n^B$. Note that n is the number of *unconstrained* system mobilities; the net number of degrees of freedoms after constraints will be $n_{net} = n - m_{net}$ where $m_{net} \le m$ is the number of *independent* constraint equations generated by the system's constraints.

Generalized speeds u are fundamentally related to the physics of the system, while generalized coordinates q are chosen primarily to facilitate good numerical behavior during computation. Thus the number of generalized speeds introduced by a mobilizer is always the same as the number of mobilities, that is, $n_u^B = n^B$ so that the generalized speeds are always mutually independent. The number of generalized coordinates $n_q^B \ge n^B$ so the coordinates q^B may not be independent. In Simbody, that occurs only when a mobilizer uses a quaternion to represent unrestricted orientation. For convenience we introduce the symbol n_{quat}^B defined as follows:

$$n_{quat}^B \equiv \begin{cases} 1, & \text{if mobilizer B uses a quaternion} \\ 0, & \text{otherwise} \end{cases} \qquad (11.1)$$

* We use n, representing the mobilizer's of degrees of freedom, rather than n_u to count generalized speeds, since there is necessarily the same number of generalized speeds as degrees of freedom.

Then the total number of quaternions in the system is $n_{\text{quat}} = \sum_{\text{B}} n_{\text{quat}}^{\text{B}}$.

It should be emphasized that our presentation of the equations of motion below is a *formal* description. It would be extremely inefficient to set up and solve the equations in the form they are presented here (although many lesser codes do that). The techniques of Order(N) multibody dynamics provide the solution of these equations without ever requiring their explicit formation.

11.6.1 Unconstrained systems

In a system with no constraints, the equations of motion are

$$\dot{q} = \mathbf{Q}(q)u \tag{11.2}$$

$$\mathbf{n}(q) = 0 \tag{11.3}$$

$$\mathbf{M}(q)\dot{u} = \mathbf{f}(t,q,u,z) - \mathbf{f}_{\text{bias}}(q,u) \tag{11.4}$$

$$\dot{z} = \dot{z}(t,q,u,z) \tag{11.5}$$

Here $\mathbf{M}_{n\times n}$ is a symmetric, positive definite mass matrix which captures all the inertial properties of the system in its current configuration, and $\mathbf{f}_{n\times 1}$ is the set of all applied force and torques (including gravity) mapped into an equivalent set of n generalized forces acting along the mobilities. $\mathbf{f}_{\text{bias}}$ is equivalent to the forces representing velocity-induced coriolis acceleration and gyroscopic terms. ($\mathbf{f}_{\text{bias}}$ is quadratic in u, and is zero if u=0.) $\mathbf{Q}_{n_{\text{q}}\times n}$ is a block diagonal, invertible mapping between generalized speeds and generalized coordinate derivatives. In practice this is used to convert angular velocities to scaled quaternion derivatives or to Euler angle derivatives. The rectangular system of equations represented by (11.2) has rank only n ($\le n_{\text{q}}$), leaving quaternion lengths undetermined, so we need n_{quat} additional normalization conditions represented by (11.3) to ensure a unique solution for trajectory $q(t)$. Note that although equation (11.3) is formally a set of constraints, we consider this an unconstrained system since these constraints do not affect the physical solution.

Equation (11.4) is just a version of Newton's second law F=ma, relating forces to accelerations. The z's are n_{z} additional state variables whose values can affect the forces, which may themselves be modeled as differential equations.

z's cannot *directly* affect positions and velocities, although of course they do affect accelerations which will *ultimately* affect velocities and then positions.

Formally, we can solve equation (11.4) for the accelerations $\dot{u}$ with

$$\dot{u} = \mathbf{M}^{-1}(\mathbf{f} - \mathbf{f}_{\text{bias}}) \tag{11.6}$$

By formally we mean, "don't take this literally!" There is always special structure to **M** that can be exploited such that the accelerations can be calculated directly in O(n) time, while a literal matrix inversion would take O(n^3) time and be prohibitive for large systems. Even *forming* **M** would take O(n^2) time since it has n^2 elements, so Simbody neither forms nor factors **M** while solving equation (11.6).

As an extreme example, consider the special case of a molecular system modeled with n_a point mass atoms and Cartesian coordinates, so that $n=3n_a$. **M** is then a diagonal matrix of dimension $3n_a \times 3n_a$ with the atomic masses (each repeated three times) arrayed along the diagonal. The q's are the Cartesian coordinates, and the u's are the Cartesian velocities so $n_q=n_u$, **Q** is an identity matrix, and $\dot{q} = u$. $\mathbf{f}_{\text{bias}}$ is always zero for this system. **f** is simply the Cartesian forces acting on each coordinate of each atom, typically resulting from taking the gradient of the potential energy function. This represents a set of $3n_a$ uncoupled scalar equations for the Cartesian accelerations of each atom, which can clearly be solved in O(n)!

In a more general multibody system **M** will be dense as a result of coupling produced by the internal coordinates. Use of quaternions for orientation results in there being more q's than u's and **Q** is no longer identity. However, equation (11.6) provides the solution for the accelerations in this case just as well, and the special structure of multibody systems permits a solution in O(n) time regardless of the amount of coupling in **M**.

11.6.2 Constrained systems

Constraints introduce unknown forces and torques into the system. Constraints are introduced, for example, if there are topological loops created by the set of bodies and joints. The constraint forces involve additional unknowns (along with accelerations). We call these unknowns Lagrange

multipliers and represent them as a vector λ of length m. These are mapped to mobility forces with a coupling matrix $\mathbf{G}$ and thus modify acceleration equation (11.4) like this:

$$\mathbf{M}\dot{u} + \mathbf{G}^{\mathrm{T}}\lambda = \mathbf{f} - \mathbf{f}_{\text{bias}} \tag{11.7}$$

$$\mathbf{G}\dot{u} = \mathbf{b} \tag{11.8}$$

where $\mathbf{G}_{m\times n}=\mathbf{G}(q)$ and $\mathbf{b}_{m\times 1}=\mathbf{b}(t,q,u)$, m is the number of constraints and $n=n_{\mathrm{u}}$ is the number of generalized speeds. Equations (11.7) and (11.8) are a system of $n+m$ equations in $n+m$ unknowns ($\dot{u}$ and λ) so can be solved for the accelerations that satisfy the constraint equations. The solution of this system makes use of the unconstrained result from equation (11.6). Note that because we can directly solve for $\dot{u}$ and eliminate λ, this is still just an ordinary differential equation, with $\dot{u} = \dot{u}(t,q,u,z)$.*

Equation (11.8) is written in terms of linear constraints on the accelerations $\dot{u}$. However, in most cases constraints are known only at the configuration level, that is, as nonlinear algebraic relationships which must hold among the q's or among quantities fully determined by the q's. A constraint like "these two atoms must be a certain distance apart at all times" would be an example. In other cases the constraints may be expressed at the velocity level as restrictions on u. In these cases we time-differentiate the constraints twice or once, resp., until we have corresponding acceleration constraints, and then use them in equation (11.8), along with any constraints which may have been defined directly at the acceleration level.

Following this procedure yields correct accelerations, but with approximate numerical integration of those accelerations the original position or velocity constraints will not remain satisfied over time. In practice, any constraints that are not actively enforced will gradually drift apart during a dynamic simulation. To address this, we must keep the original algebraic constraints in

---

* Knocking equations (11.7) and (11.8) around a little, one can verify that $\dot{u} = \dot{u}_0 - \dot{u}_C$, where $\dot{u}_0 = \mathbf{M}^{-1}(\mathbf{f} - \mathbf{f}_{\text{bias}})$, $\dot{u}_C = \mathbf{M}^{-1}\mathbf{G}^{\mathrm{T}}\lambda$, and $\lambda = (\mathbf{G}\mathbf{M}^{-1}\mathbf{G}^{\mathrm{T}})^{-1}(\mathbf{G}\dot{u}_0 - \mathbf{b})$. In general the constraint matrix $\mathbf{G}$ can be singular, so there may be no solution, or an unlimited number of solutions, in which case least squares solutions for λ are typically used.

the problem and solve them along with the ODE (11.7), (11.8). That results in a system of mixed differential and algebraic equations, known as a DAE. Equations (11.9)-(11.15) shows the complete set of equations, including the set of auxiliary, unconstrained differential equations in z which may be required in the computation of forces.

$$\dot{q} = \mathbf{Q}(q)u \tag{11.9}$$

$$\mathbf{n}(q) = 0 \tag{11.10}$$

$$\mathbf{M}(q)\dot{u} + \mathbf{G}^{\mathrm{T}}\lambda = \mathbf{f}(t,q,u,z) - \mathbf{f}_{\mathrm{bias}}(q,u) \tag{11.11}$$

$$\mathbf{a}(t,q,u,\dot{u}) \equiv \mathbf{A}\dot{u} - \mathbf{b}_{\mathrm{a}}(t,q,u) = 0 \tag{11.12}$$

$$\mathbf{v}(t,q,u) = 0 \tag{11.13}$$

$$\mathbf{p}(t,q) = 0 \tag{11.14}$$

$$\dot{z} = \dot{z}(t,q,u,z) \tag{11.15}$$

Constraint coupling matrix $\mathbf{G}_{m\times n}$ is obtained from equations (11.12)-(11.14) as discussed below.

Equations (11.9)-(11.15) show the system as it is defined to Simbody, including all the constraints that must be obeyed during a dynamic simulation, starting with initial conditions t_0, $q(t_0)$, $u(t_0)$, $z(t_0)$ such that constraint equations (11.10), (11.13), and (11.14) are satisfied.

The function $\mathbf{a}_{m_{\mathrm{a}}\times 1}$ specifies m_{a} *acceleration* (index 1) constraints, which are required to be linear in the accelerations $\dot{u}$, with m_{a}xn coefficient matrix **A**. These have application, for example, in some models of Coulomb friction[7] and in producing simulations which must track measured accelerations or reaction forces.

The function $\mathbf{v}_{m_{\mathrm{v}}\times 1}$ specifies m_{v} *nonholonomic* (velocity, index 2) constraints (usually, but not necessarily, linear or quadratic in u). These include, for example, “non slip” constraints like gears and rolling contact, and constraints involving kinetic energy. The m_{v} time derivatives $\dot{\mathbf{v}}$ of the nonholonomic constraints **v** must also be obeyed since, like **a**, they restrict the allowable values of $\dot{u}$ and in general they will be coupled to **a**.

The function $\mathbf{p}_{m_{\mathrm{p}}\times 1}$ specifies m_{p} *holonomic* (position, index 3) constraints, which are arbitrarily nonlinear in t and q. The m_{p} time derivatives $\dot{\mathbf{p}}$, and m_{p}

second time derivatives $\ddot{\mathbf{p}}$ must also be obeyed since they impose restrictions on u and $\dot{u}$, respectively, and in general will be coupled to **v** and **a**.

Then **a**, $\dot{\mathbf{v}}$, and $\ddot{\mathbf{p}}$ together constitute the acceleration-level constraints, so we have $m=m_a+m_v+m_p$ the total number of constraints at the acceleration level.

The system of equations (11.9)-(11.15) contains $n_q+n_u+n_z+m$ equations in the $n_q+n_u+n_z+m$ unknowns q,u,z and λ, and should thus yield a unique solution for the resulting trajectories $q(t)$, $u(t)$, $z(t)$ and $\lambda(t)$, given consistent t_0, $q(t_0)$, $u(t_0)$, and $z(t_0)$ to start with. Unfortunately, obtaining that solution is easier said than done! Numerical analysts describe a system like this as a Differential Algebraic Equation (DAE) system of index 3, for which few entirely satisfactory solution methods exist. For a survey of methods, see reference 8. For Simbody we advocate the method known as *coordinate projection*,[9] which is very accurate and reliable in practice. We also support the more conventional but less robust technique called Baumgarte stabilization,[10] and Simbody is flexible enough to allow most other methods to be used as well. In the next section we'll discuss how we go about solving equations (11.9)-(11.15).

11.6.3 Dynamic simulation solution method

The previous section glossed over some details of the system formulation that we'll need to deal with here. Let's first revisit the several types of constraint equations. *Holonomic constraint equations* **p** (equation (11.14)) are those that are expressed at the q (position) level and represent meaningful physical properties of the system. Holonomic constraint equations involve only state variables at position stage or below, that is, q, t, parameters (instance variables), and modeling choices. Holonomic constraint equations can be differentiated once to produce *holonomic velocity constraint equations* $\dot{\mathbf{p}}$, and again to produce *holonomic acceleration constraint equations* $\ddot{\mathbf{p}}$.

Nonholonomic constraint equations **v** (equation (11.13)) are those that are directly expressed in terms of system velocities, that is, at the u level, and also represent meaningful physical properties. Typical examples are "non slip" conditions like rolling or gears, but these can also include more global restrictions such as a conservation of energy constraint. Nonholonomic

constraint equations involve state variables at the velocity stage and below, which includes the entire list given above for holonomic constraints plus the generalized speeds u. Nonholonomic constraint equations can be differentiated once to produce *nonholonomic acceleration constraint equations* $\dot{\mathbf{v}}$.

Acceleration constraint equations **a** (equation (11.12)) are those which are directly specified in terms of the system accelerations $\dot{u}$, or quantities which are linearly related to accelerations such as reaction forces or constraint forces. Like holonomic and nonholonomic constraints these are physically meaningful constraints.

Also at position level are *quaternion normalization constraints* **n**, each of which involves only the coordinates of a single mobilizer and is present for numerical reasons rather than physical. These are produced by mobilizers which use quaternions to permit unrestricted orientation. Simbody's implementation ensures that violation of quaternion normalization constraints has *no physical effect* on the system. That is, a change to q which serves only to satisfy a quaternion normalization constraint is not permitted to cause any change to the system configuration. Quaternion normalization constraints exist only to reduce the number of degrees of freedom of a mobilizer's four quaternions down to the three physical rotational degrees of freedom represented by its three u's.

Unlike the holonomic and nonholonomic constraints, there are no constraints at the velocity or acceleration level corresponding to the quaternion normalization constraint. Equation (11.9) constructs the quaternion derivatives in terms of the three independent u's, ensuring by construction that the velocity-level constraints are satisfied.

As an aside, note that the system equations include a block diagonal invertible linear mapping between the u's and the time derivatives of the q's: $\dot{q} = \mathbf{Q}(q)u$. Although this is a rectangular matrix, it is invertible. Note that when the quaternion normalization constraints are not satisfied exactly, the 4x3 blocks $\mathbf{Q}_i$ on the diagonal of $\mathbf{Q}$ which correspond to quaternion q_i will be scaled by $|q_i|$ so that the resulting $\dot{q}$ is the derivative of the *unnormalized* quaternion.

Table 1 provides details of the mathematical structure for each of the four types of Simbody constraints, and defines the symbols we'll refer to below.

Here is the system in the form we actually solve in Simbody:

kinematics

$$\begin{aligned} \dot{q} &= \mathbf{Q}u \\ \mathbf{n}(q) &= 0 \end{aligned} \tag{11.16}$$

dynamics

$$\begin{aligned} \begin{bmatrix} \mathbf{M} & \mathbf{G}^{\mathrm{T}} \\ \mathbf{G} & 0 \end{bmatrix} \begin{bmatrix} \dot{u} \\ \lambda \end{bmatrix} &= \begin{bmatrix} \mathbf{f} - \mathbf{f}_{\mathrm{bias}} \\ \mathbf{b} \end{bmatrix} \\ \dot{z} &= \dot{z}(t, q, u, z) \end{aligned} \tag{11.17}$$

velocity manifold

$$\begin{aligned} \mathbf{P}u - \mathbf{c} &= 0 \\ \mathbf{v}(t, q, u) &= 0 \end{aligned} \tag{11.18}$$

position manifold

$$\mathbf{p}(t, q) = 0 \tag{11.19}$$

We are given initial conditions t_0, $q(t_0)$, $u(t_0)$ such that equations (11.18) and (11.19) are satisified, as well as initial values $z(t_0)$, and are asked in a dynamic simulation study to solve for $q(t)$, $u(t)$, and z(t) for $t_0 \le t \le t_{\mathrm{final}}$.

quaternion normalization (index 2)	position level only	$\mathbf{n}(q) : q_i^{\mathsf{T}} q_i = \mathbf{1}$	
holonomic constraint equations (index 3)	position $\mathbf{p} = 0$	$\mathbf{p}(t,q) = 0$	
	velocity $\dot{\mathbf{p}} = 0$ (index 2)	$\mathbf{P}u - \mathbf{c}(t,q) = 0$	$\mathbf{P} = \frac{\partial \dot{\mathbf{p}}}{\partial u} = \frac{\partial \mathbf{p}}{\partial q}\mathbf{Q}$, $\mathbf{c} = -\frac{\partial \mathbf{p}}{\partial t}$
	acceleration $\ddot{\mathbf{p}} = 0$ (index 1)	$\mathbf{P}\dot{u} - \mathbf{b}_{\mathrm{p}}(t,q,u) = 0$	$\mathbf{b}_{\mathrm{p}} = \dot{\mathbf{c}} - \dot{\mathbf{P}}u$
nonholonomic constraint equations (index 2)	velocity $\mathbf{v} = 0$	$\mathbf{v}(t,q,u) = 0$	
	acceleration $\dot{\mathbf{v}} = 0$ (index 1)	$\mathbf{V}\dot{u} - \mathbf{b}_{\mathrm{v}}(t,q,u) = 0$	$\mathbf{V} = \frac{\partial \mathbf{v}}{\partial u}$, $\mathbf{b}_{\mathrm{v}} = -\left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{V}u\right)$
acceleration constraint equations (index 1)	acceleration $\mathbf{a} = 0$	$\mathbf{A}\dot{u} - \mathbf{b}_{\mathrm{a}}(t,q,u) = 0$	$\mathbf{A} = \frac{\partial \mathbf{a}}{\partial \dot{u}}$
all index 1 constraints	collect contributions from all of the above	$\mathbf{G}\dot{u} - \mathbf{b} = 0$	$\mathbf{G} = \begin{bmatrix} \mathbf{P} \\ \mathbf{V} \\ \mathbf{A} \end{bmatrix}$, $\mathbf{b} = \begin{bmatrix} \mathbf{b}_{\mathrm{p}} \\ \mathbf{b}_{\mathrm{v}} \\ \mathbf{b}_{\mathrm{a}} \end{bmatrix}$

Table 1: the four types of constraint equations dealt with by Simbody.

11.7 Equations for general mobilizer

A Simbody Mobilizer defines the permitted mobility of a body B with respect to a more-inboard (closer to Ground) body P, called its parent body. A given mobilizer provides *n* mobilities (degrees of freedom) for body B with respect to body P, with $0 \le n \le 6$.

Each body has a unique parent so there is a one-to-one correspondence between bodies and mobilizers; in Simbody we call the combination of a body with its unique mobilizer a *MobilizedBody*. The permitted mobility is described in terms of *n* scalar velocity coordinates *u* (called *generalized speeds*), and $n_{\mathrm{q}} \ge n$ scalar position coordinates *q* (called *generalized coordinates*). The time derivatives of the generalized speeds serve as the

generalized accelerations $\dot{u}$. The meanings of these quantities are defined by the following equations, which express the body's allowed motion with respect to its parent in terms of q and u. This relative motion is defined using a pair of coordinate frames associated with the MobilizedBody B: the unique mobilizer "moving" frame M attached to B with constant transform ${}^{B}X^{M}$ and "fixed" frame F attached to P with constant transform ${}^{P}X^{F}$.

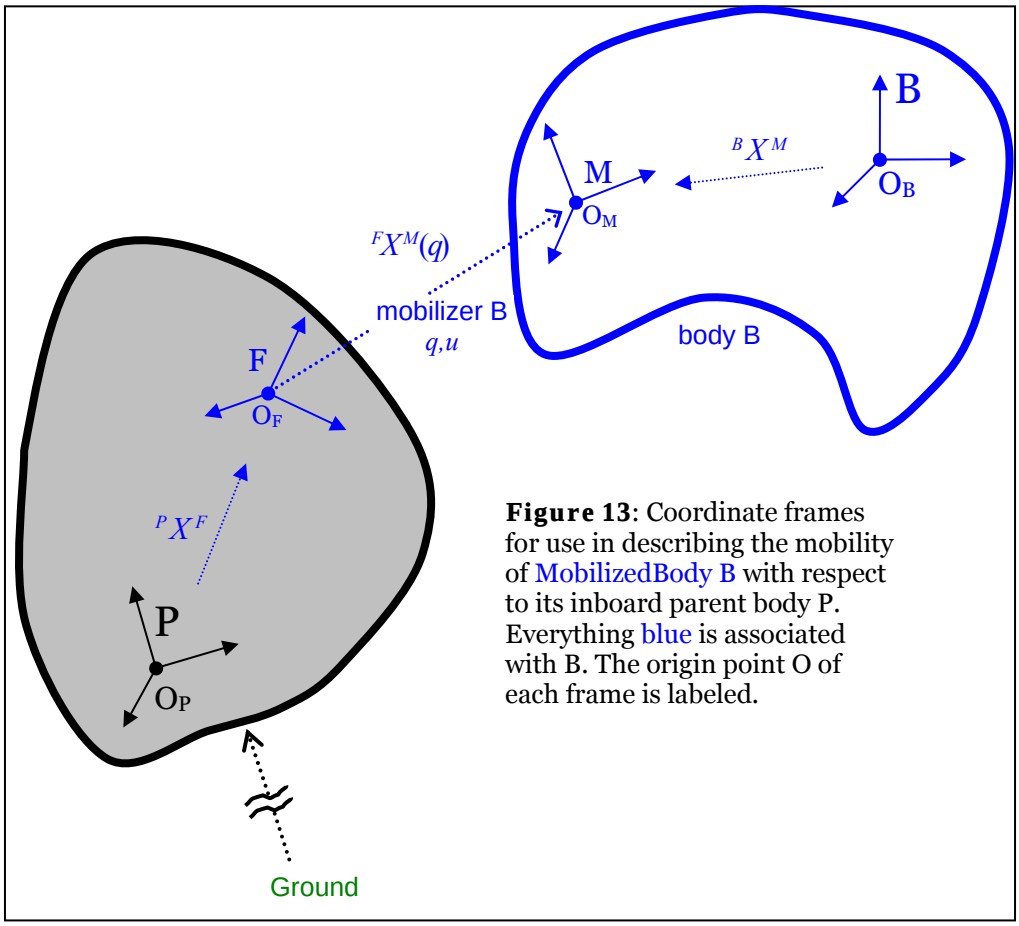


Figure 13: Coordinate frames for use in describing the mobility of MobilizedBody B with respect to its inboard parent body P. Everything blue is associated with B. The origin point O of each frame is labeled.

$$ {}^{F}X^{M} = \left({}^{F}R^{M} \mid {}^{F}p^{M} \right) = X(q) = \left(R(q) \mid p(q) \right) \quad (11.20) $$

$$ {}^{F}V^{M} = \begin{pmatrix} {}^{F}\omega^{M} \\ {}^{F}v^{M} \end{pmatrix} = \mathbf{J}(X(q))u \quad (11.21) $$

$$ {}^{F}A^{M} = {}^{F}\dot{V}^{M} = \begin{pmatrix} {}^{F}\beta^{M} \\ {}^{F}a^{M} \end{pmatrix} = \mathbf{J}\dot{u} + \dot{\mathbf{J}}u \quad (11.22) $$

$$ \dot{q} = \mathbf{Q}(q)u \quad (11.23) $$

$$ \mathrm{n}(q) = 0 \quad (11.24) $$

where

$$ {}^{F}\dot{X}^{M} = \left({}^{F}\dot{R}^{M} \;\middle|\; {}^{F}\dot{p}^{M} \right) = \left({}^{F}\omega^{M} \times \mathrm{R}(q) \;\middle|\; {}^{F}v^{M} \right) \qquad (11.25) $$

This implies a relationship that must hold among X(q), **J**, and **Q**:

$$ \begin{aligned} \dot{\mathrm{R}}(q) &= \frac{\partial \mathrm{R}(q)}{\partial q}\mathbf{Q}u = \mathbf{J}_{\omega}u \times R(q) \\ \dot{\mathrm{p}}(q) &= \frac{\partial \mathrm{p}(q)}{\partial q}\mathbf{Q}u = \mathbf{J}_{v}u \end{aligned} \qquad (11.26) $$

where $\mathbf{J}_{\omega}$ and $\mathbf{J}_{v}$ are the upper and lower 3xn partitions of **J**. Intuitively, this is stating the requirement that the spatial velocity produced from u by the action of **J** is the time derivative of the spatial position and orientation produced from q by the nonlinear function X(q), with matrix **Q** serving to mediate between u and $\dot{q}$. Note that **J** depends only on the transform (spatial position) represented by the set of q's, not on the definitions of the individual q's.

At times it is more convenient to deal with the mobilizer Jacobian describing the allowed motion of the body frame B with respect to the parent body's frame P, rather than between the two mobilizer frames. This is related to the Jacobian defined above by the constant transforms ${}^{P}X^{F}$ and ${}^{M}X^{B}$. First, perform a rigid body shift of the spatial velocity from M's origin outward to B's, using the kinematic shift operator ϕ^{T}:

$$ {}^{F}\mathbf{J}^{B} \;\square\; \frac{\partial\, {}^{F}V^{B}}{\partial u} = \phi^{\mathsf{T}}\left(\left[{}^{M}p^{B} \right]_{F} \right) \cdot {}^{F}\mathbf{J}^{M} \qquad (11.27) $$

where

$$ \left[{}^{M}p^{B} \right]_{F} = {}^{F}R^{M} \cdot {}^{M}p^{B} $$

$$ \phi\left(\left[\mathbf{v}^{B} \right]_{A} \right) = \begin{pmatrix} 1 & \left(\left[\mathbf{v}^{B} \right]_{A} \right)_{\times} \\ 0 & 1 \end{pmatrix} $$

Note that although we are shifting from one point on body B to another, the effect is time varying since we are expressing the shift vector in the parent body using the cross-mobilizer rotation matrix ${}^{F}R^{M}(q)$.

Next, re-express the resulting spatial velocity (currently in F) to P:

$$ {}^{P}\mathbf{J}^{B} \equiv \frac{\partial\, {}^{P}V^{B}}{\partial u} = {}^{P}R^{F} \cdot {}^{F}\mathbf{J}^{B} \tag{11.28} $$

This transformation involves only a constant rotation matrix, and the translation of the reference frame from F to P doesn't affect the velocity.

The time derivative taken in P is then

$$ {}^{P}\dot{\mathbf{J}}^{B} \equiv \frac{{}^{P}d}{dt}\ {}^{P}\mathbf{J}^{B} = {}^{P}R^{F} \cdot {}^{F}\dot{\mathbf{J}}^{B} = \left[{}^{F}\dot{\mathbf{J}}^{B} \right]_{P} \tag{11.29} $$

where

$$ {}^{F}\dot{\mathbf{J}}^{B} \equiv \frac{{}^{F}d}{dt}\ {}^{F}\mathbf{J}^{B} = \dot{\phi}^{\mathsf{T}}(\left[{}^{M}p^{B} \right]_{F}) \cdot {}^{F}\mathbf{J}^{M} + \phi^{\mathsf{T}}(\left[{}^{M}p^{B} \right]_{F}) {}^{F}\dot{\mathbf{J}}^{M} \tag{11.30} $$

and

$$ \dot{\phi}(\left[\mathbf{v}^{B} \right]_{A}) = \begin{pmatrix} 0 & \left({}^{A}\omega^{B} \times \left[\mathbf{v}^{B} \right]_{A} \right)_{\times} \\ 0 & 0 \end{pmatrix} \tag{11.31} $$

These matrices are related to the hinge matrix $\mathbf{H}^{\mathsf{T}}$ in reference 13 as follows:

$$ \mathbf{H}^{\mathsf{T}} \equiv \frac{\partial \left[{}^{P}V^{B} \right]_{G}}{\partial u} = \left[{}^{P}\mathbf{J}^{B} \right]_{G} = {}^{G}R^{P} \cdot {}^{P}\mathbf{J}^{B} \tag{11.32} $$

$$ \begin{aligned} \dot{\mathbf{H}}^{\mathsf{T}} \equiv \frac{{}^{G}d}{dt}\mathbf{H}^{\mathsf{T}} &= {}^{G}\dot{R}^{P} \cdot {}^{P}\mathbf{J}^{B} + {}^{G}R^{P} \cdot {}^{P}\dot{\mathbf{J}}^{B} \\ &= {}^{G}\omega^{P}_{\times} \cdot \left[{}^{P}\mathbf{J}^{B} \right]_{G} + \left[{}^{P}\dot{\mathbf{J}}^{B} \right]_{G} \end{aligned} \tag{11.33} $$

Note that **H** is not *shifted* to ground, but only *rotated* (re-expressed). That is, it still represents motion of B with respect to P (not with respect to G), however it has been re-expressed in the Ground frame. (Time derivatives are taken in the frame indicated by the expressed-in frame of the differentiated quantity.)

11.7.1 Defining a Custom Mobilizer in Simbody

[TODO: this is currently a design specification, not a description]

To specify a mobilizer for Simbody's use requires that the mobilizer's implementor provide the following information and computations:

- The number n of mobilities (degrees of freedom) provided by the mobilizer, which is also the number of generalized speeds u, and the number n_q of generalized coordinates q.

```
void getNumMobilities
    (const State&, // already realized through Stage::Model
     int& n, int& nq) const;
```

- A method for projecting the q's onto the mobilizer's local constraint manifold. In practice, this means normalizing quaternions if they are present in q. If there are no constraints on the q's, this routine doesn't need to be supplied; by default it will do nothing.

```
template<int nq>
void normalizeQ
      (const State&, // already realized through Stage::Model
       Vec<nq>& q);   // in/out: for quaterion part of q, set q = q/|q|
```

  Note that this routine does not normalize the q's present in the State; instead it works with a separately-supplied q argument. (That means it is an *operator*, not a *solver*.) The supplied State (which is `const`) is used only for modeling information, to discover whether the selected model involves constrained q's. The q argument may well be a reference to the q's in the State, but it doesn't have to be.

  Important note: this method must perform a *normal* projection of q onto the local position constraint manifold (that is, the q's must be changed by the least possible amount in a 2-norm sense). Normalization of quaternions is such a projection.

- The configuration function X(q) that maps the generalized coordinates q to a transform (rotation and translation) that expresses the orientation and position of frame M as measured in frame F.

```
template<int nq>
void calcMobilizerTransform
     (const State&,  // already realized through Stage::Instance
      const Vec<nq>& q,
      Transform&     X_FM) const;
```

  Important note for when there are quaternions in q: it is required that the above routine return the *identical* transform regardless of whether the quaternions are normalized. That is, the routine should behave as though it normalizes the quaternion prior to generating the transform.

- The kinematic Jacobian $\mathbf{J}$, which is a 6xn matrix mapping the generalized speeds u to the effect each one has on the spatial velocity of M in F. Simbody requires both $\mathbf{J}(\mathrm{X}(q))$ and its time derivative

$$\dot{\mathbf{J}}(\mathrm{X}(q),\mathrm{V}(q,u)) \equiv \frac{{}^{F}d}{dt}\mathbf{J}(\mathrm{X}(q)) = \sum_{j=1}^{4}\frac{\partial \mathbf{J}}{\partial X_j}\dot{X}_j \quad (X_j \text{ is } j^{\text{th}} \text{ column of } X)$$

$$\text{where } \dot{X} = \left(\mathbf{J}_\omega u \times R(q) \mid \mathbf{J}_v u\right) = \left(\omega \times R(q) \mid v\right)$$

$$\text{with } \mathbf{J} \equiv \begin{pmatrix}\mathbf{J}_\omega \\ \mathbf{J}_v\end{pmatrix}, \quad \mathrm{V} \equiv \begin{pmatrix}\omega \\ v\end{pmatrix} = \begin{pmatrix}\mathbf{J}_\omega \\ \mathbf{J}_v\end{pmatrix} u$$

  as explicit 6xn matrices (actually 2xn `Vec3`'s):

```
template<int n>
void calcMobilizerJacobian
     (const State&,   // already realized through Stage::Position; includes X_FM
      Mat<2,n,Vec3>& J_FM) const;

template<int n>
void calcMobilizerJacobianDot
     (const State&,  // already realized through Stage::Velocity; includes u
      Mat<2,n,Vec3>& JDot_FM) const;
```

  Note that $\mathbf{J}$ and $\dot{\mathbf{J}}$ should not depend explicitly on q, but rather only on the cross-mobilizer transform X(q) represented by q. That is because the choice of q's is somewhat arbitrary (e.g. quaternions vs. rotation angles), but all equivalent sets of q's must generate the same $\mathbf{J}$. Then $\dot{\mathbf{J}}$ depends on u only through V(q,u)=$\mathbf{J}u$.

- The matrix **Q** which maps from generalized speeds u to time derivatives $\dot{q}$ of generalized coordinates. Simbody requires both $\mathbf{Q}(q)$ and its time derivative $\dot{\mathbf{Q}}(q,u)$, in the form of methods which can form $\dot{q} = \mathbf{Q}u$ given u, and $\ddot{q} = \mathbf{Q}\dot{u} + \dot{\mathbf{Q}}u$ given $\dot{u}$ (in that case u is taken from the State):

```
template<int n, int nq>
void calcQDotFromU
     (const State&, // already realized through Stage::Position; includes q
      const Vec<n>& u,
      Vec<nq>&      qDot) const;

template<int n, int nq>
void calcQDotDotFromUDot
     (const State&, // already realized through Stage::Velocity; includes q,u
      const Vec<n>& udot,
      Vec<nq>&      qDotDot) const;
```

  Simbody provides utilities for the common cases of angular velocity-to-quaternion derivative and angular velocity-to-rotation angle derivatives. For most other cases, $\dot{q} = u$ and $\ddot{q} = \dot{u}$.

  Important note for when there are quaternions in q: unlike `calcMobilizerTransform()` above, `calcQDot()` and `calcQDotDot()` must behave *differently* when the quaternion is unnormalized. That is, they *must not* normalize q before calculating $\mathbf{Q}$ and $\dot{\mathbf{Q}}$.

- Coordinate-independent mobilizer initialization routines. The meaning of these is as follows: if the mobilizer can represent the specified configuration or spatial velocity, then the q's or u's are set appropriately. If not, the q's or u's are set to some approximation of the desired result, with the particular approximation decided by the author of the mobilizer.

```
template<int nq>
void setQToFitTransform
     (const State&, // already realized through Stage::Instance
      const Transform& X_FM,
      Vec<nq>&         q) const;
```

```
template<int n, int nq>
void setUToFitVelocity
     (const State&, // already realized through Stage::Instance; includes q
      const SpatialVec& V_FM,
      Vec<n>&           u) const;
```

- Optional: decorative geometry suitable for visualizing this mobilizer.

11.7.2 Mobilizer examples

TODO

11.8 Equations for general constraints

A Simbody Constraint C is modeled with a set of m^C scalar constraint equations which restrict the allowable values for mobilizer coordinates by enforcing algebraic relationships among them or their time derivatives. Constraints are usually written to *directly* affect only a very small number n_b of bodies and n_m of mobilizers, typically one, two, or three, which we call the *constrained bodies* and *constrained mobilizers*. For efficient processing, Simbody must know the complete set $\{B_k^C, M_l^C\}$ of n_b^C constrained bodies and n_m^C constrained mobilizers for each Constraint C. The set of constrained bodies and mobilizers is considered topological information and is thus frozen after the Constraint is specified.

The set of mobilities which can appear in the corresponding constraint equations consists of all the mobilities u_m^C associated with the constrained mobilizers, plus all mobilities u_b^C which can affect the relative motions of any the constrained bodies. Note that while the number of mobilities associated with a mobilizer is very small, the number which may affect a set of constrained bodies can be *much* larger, potentially including all the mobilities on the paths from the constrained bodies back to Ground.

To avoid unnecessarily including a large number of mobilities in the constraint calculations for a Constraint C, Simbody searches the multibody tree from the constrained bodies in the inboard direction (towards Ground) to find the *outmost common ancestor* A^C, which is the most-outboard (highest numbered) body shared by the inboard paths of all the constrained bodies. Ground can always serve as A if no other common body can be found.

We call the path from the k^{th} constrained body inward to A^C the k^{th} *branch* of the Constraint; these branches may overlap and may also overlap with constrained mobilizers. We call the set of all generalized speeds on the k^{th} branch $u^C_{\text{b},k}$, with $u^C_{\text{b}} = \bigcup_k u^C_{\text{b},k}$; the complete set of generalized speeds which can affect Constraint C is then $u^C = \{u^C_{\text{m}}, u^C_{\text{b}}\}$. These are the Constraint's $n^C = |u^C|$ *participating mobilities*. The n^C_{q} *participating coordinates* are similarly defined as $q^C_{\text{b}} = \bigcup_k q^C_{\text{b},k}$ and $q^C = \{q^C_{\text{m}}, q^C_{\text{b}}\}$, with $n^C_{\text{q}} = |q^C|$ and $n^C_{\text{q}} \geq n^C$.

Figure 14 depicts these quantities for a single Constraint C with three constrained bodies. The figure does not show the m^C constraint equations that this Constraint generates; m^C can't be determined just from the number of constrained bodies. However, it does show how the body-affecting mobilities u^C_{b} are determined. Note that the mobilizers for the two black highlighted bodies are shared by branches 0 and 1.

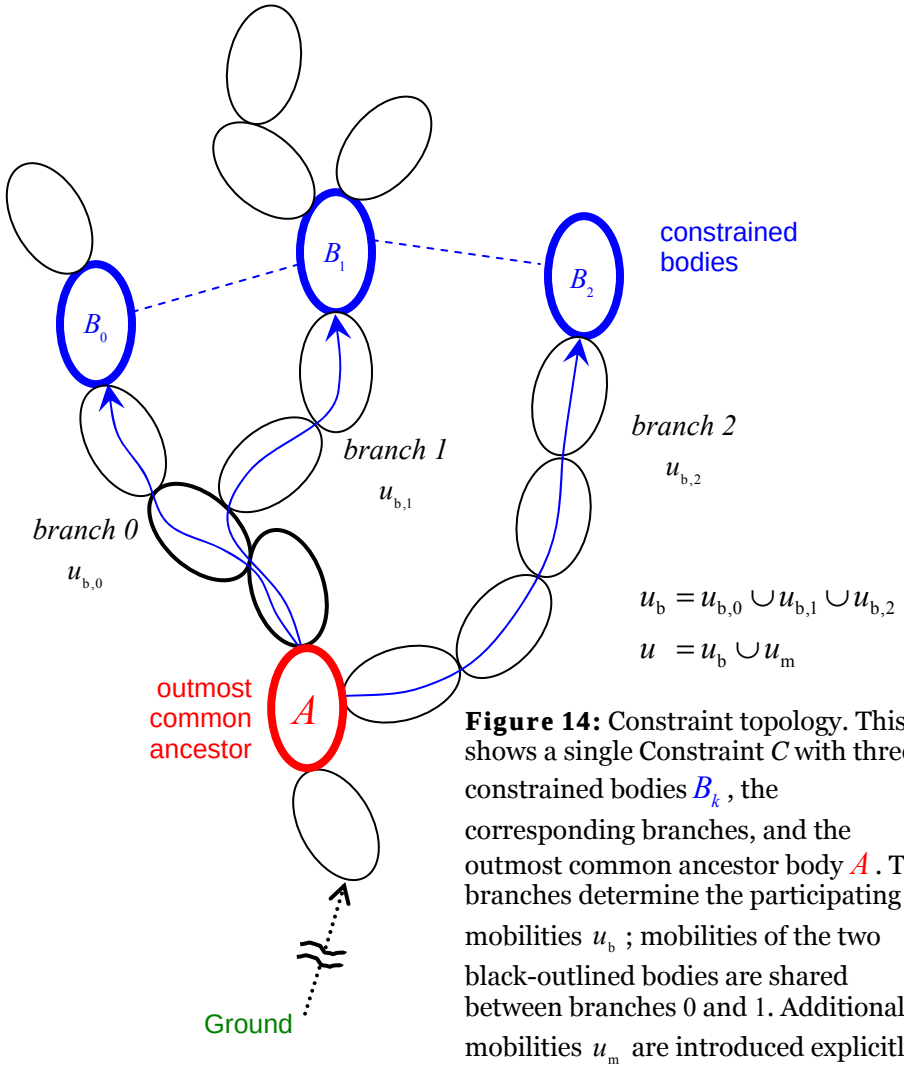


Figure 14: Constraint topology. This shows a single Constraint C with three constrained bodies B_k, the corresponding branches, and the outmost common ancestor body A. The branches determine the participating mobilities u_b; mobilities of the two black-outlined bodies are shared between branches 0 and 1. Additional mobilities u_m are introduced explicitly for constrained mobilizers.

For the rest of this section we'll drop the superscript "C" except when necessary for clarity. Without the C, our Constraint generates m constraint equations in n mobilities.

The most fundamental constraint equation is a relationship among the accelerations (the n participating generalized speed derivatives $\dot{u}$), called an acceleration constraint. Every Simbody constraint equation ultimately restricts accelerations, and these m acceleration constraint equations form part of the dynamical equations of motion. The i^{th} acceleration constraint equation has the following form:

$$g_i(t,q,u,\dot{u}) \equiv \mathbf{g}_i \dot{u} - b_i(t,q,u) = 0 \tag{11.34}$$

Where g_i is a scalar function, $\mathbf{g}_i$=$\mathbf{g}_i(q)$ is a row vector of length n, and b_i is a scalar function. Every defined Constraint must provide a method for efficiently evaluating its m scalar acceleration error functions g_i. For constraint equations involving only constrained *mobilizers* this can be done directly in terms of the mobilities u_m. But in the case of participating mobilities u_b due to constrained *bodies*, the constraints are not normally known explicitly as in (11.34), but rather in terms of some physical consequence of $\dot{u}_b$, such as body accelerations. The user-written routine is expected to calculate the error in those terms in constant time, with the physical consequences of $\dot{u}_b$ having been supplied by Simbody after an O(n) computation.

Similarly, the meaning of the Lagrange multipliers λ is given by

$$f_i(q,\lambda_i) \equiv \mathbf{g}_i^T \lambda_i \tag{11.35}$$

where f_i is a column vector function giving the n generalized forces generated by the scalar multiplier λ_i allocated to the i^{th} constraint equation. Every defined Constraint must provide a method for efficiently calculating its forces given its m multipliers λ. Again, except for participating coordinates due to constrained mobilizers, this is normally not known explicitly in generalized forces as in equation (11.35), but in terms of forces and torques applied to bodies. The user-written routine is written as a constant-time function in those terms, and then Simbody converts the result to generalized forces with a single O(n) computation.

Constraint equations may differ in the level at which they are first defined: position, velocity, or acceleration. When a constraint equation is introduced at the position level (such constraints are called *holonomic* and are typically nonlinear), it is differentiated once to yield a (linear) constraint on velocities, and again to yield a (linear) constraint on accelerations. When a constraint is first introduced at the velocity level (a *nonholonomic* constraint, which can be nonlinear in velocities) it is differentiated once to yield a (linear) constraint on accelerations. A constraint which appears *only* at the acceleration level (an *acceleration-only* constraint; not common) is required by Simbody to be

linear in the generalized accelerations $\dot{u}$. Here are the equations defining each of the three types of constraint equation:

holonomic (position) constraints $(0 \le j < m_\mathrm{p})$

$$\mathrm{p}_j(t,q) = 0 \tag{11.36}$$

$$\Rightarrow \quad \dot{\mathrm{p}}_j(t,q,u) \quad \mathbf{p}_j u - c_j(t,q) = 0 \tag{11.37}$$

$$\Rightarrow \quad \ddot{\mathrm{p}}_j(t,q,u,\dot{u}) \quad \mathbf{p}_j \dot{u} - b_{\mathrm{p},j}(t,q,u) = 0 \tag{11.38}$$

nonholonomic (velocity) constraints $(0 \le j < m_\mathrm{v})$

$$\mathrm{v}_j(t,q,u) = 0 \tag{11.39}$$

$$\Rightarrow \quad \dot{\mathrm{v}}_j(t,q,u,\dot{u}) \quad \mathbf{v}_j \dot{u} - b_{\mathrm{v},j}(t,q,u) = 0 \tag{11.40}$$

acceleration-only constraints $(0 \le j < m_\mathrm{a})$

$$\mathrm{a}_j(t,q,u,\dot{u}) \quad \mathbf{a}_j \dot{u} - b_{\mathrm{a},j}(t,q,u) = 0 \tag{11.41}$$

where the row vectors are

$$\mathbf{p}_j(q) = \frac{\partial \ddot{\mathrm{p}}_j}{\partial \dot{u}} = \frac{\partial \dot{\mathrm{p}}_j}{\partial u} = \frac{\partial \mathrm{p}_j}{\partial q}\mathbf{Q}^C \tag{11.42}$$

$$\mathbf{v}_j(q) = \frac{\partial \dot{\mathrm{v}}_j}{\partial \dot{u}} = \frac{\partial \mathrm{v}_j}{\partial u} \tag{11.43}$$

$$\mathbf{a}_j(q) = \frac{\partial \mathrm{a}_j}{\partial \dot{u}} \tag{11.44}$$

and $\mathbf{Q}^C$ is an $n_\mathrm{q}^C \times n^C$ matrix assembled from a subset of the rows and columns of the global $\mathbf{Q}$ such that $\dot{q}^C = \mathbf{Q}^C u^C$. Note that the equations marked with blue arrows are implied by differentiation of the modeled constraint equations; they are not independent. These add another $2m_\mathrm{p}+m_\mathrm{v}$ equations to the m^C modeled ones.

All m_p rows $\mathbf{p}_j$ stacked together form matrix $\mathbf{P}^C$, and all m_v rows $\mathbf{v}_i$ form matrix $\mathbf{V}^C$, which together are used for initial satisfaction of position and velocity constraints, as well as for constraint projection during numerical integration. All m_a rows $\mathbf{a}_j$ together form matrix $\mathbf{A}^C$, and $\mathbf{P}^C$,$\mathbf{V}^C$,$\mathbf{A}^C$ stacked together form the m^C rows of constraint matrix $\mathbf{G}^C = \begin{bmatrix} \mathbf{P}^C \\ \mathbf{V}^C \\ \mathbf{A}^C \end{bmatrix}$ as discussed above. Note that each of the m^C rows $\mathbf{g}_i$ in (11.34) is actually one of the rows $\mathbf{p}_j$, $\mathbf{v}_j$, or $\mathbf{a}_j$.

11.8.1 Explicit calculation of constraint matrices

For efficient calculation of constraint forces and for performing constraint projections, Simbody needs to be able to efficiently calculate matrix-vector products involving the constraint matrices and their transpose. We expect to be able to calculate either $\mathbf{G}v$ or $\mathbf{G}^T w$ in O(n+m) time, where $\mathbf{G}$ is m×n and v and w are conformant column vectors. (Note that a straightforward matrix multiply would be O(nm), much more expensive.) The Constraint writer is required to provide implementations of virtual methods which can be used to perform these operations efficiently.

With the O(n+m) matrix-vector multiplies available, Simbody can calculate the constraint matrices **P**, **V**, and **A** (collectively **G**) explicitly in constant time per element. By making m calls to the provided routines, m×n matrices can be calculated in O(nm+m^2)=O(nm)* time which is within a constant factor of optimal.

Regardless of whether a constraint equation is initially specified at position, velocity, or acceleration level it will contribute a row **g** to the acceleration constraint matrix **G** above, which will also be a row of **P**, **V**, or **A**. So all the terms we need can be obtained by examining the constraint equation's error function once it has been expressed at the acceleration level, that is, equations (11.38), (11.40), or (11.41). Taken together, these equations are just the equations (11.34), that is, $\mathrm{g}_i(t,q,u,\dot{u}) \equiv \mathbf{g}_i\dot{u} - b_i(t,q,u) = 0$. So the rows of the explicit matrices we need are just $\partial \mathrm{g}_i(t,q,u,\dot{u})/\partial\dot{u}$. An alternative is to use the constraint force functions (11.35) which can equivalently provide a column of $\mathbf{g}_i^T$ at O(n) cost per column. Simbody thus calculates the constraint matrices a row at a time by m^C repeated calls to the O(n^C) constraint force function (11.35), yielding an explicit $\mathbf{G}^C$ matrix to machine precision in $O(m^C n^C)$ operations, which is within a constant factor of optimal since the

* because $m \le n$, $mn+m^2 \le 2mn$

matrix has $m^C n^C$ elements. The m^C scalars $\mathbf{b}^C = \begin{bmatrix} b_0 \\ \vdots \\ b_{m^C-1} \end{bmatrix}$ from each Constraint form the vector **b** in the same equation, and can if necessary be determined explicitly in O(n) time using equation (11.34) with all $\dot{u}$'s set to zero.

As discussed above, it is rare that an acceleration constraint equation will be conveniently written directly in terms of the generalized accelerations $\dot{u}$ (prescribed motion is an exception). Instead, it will be written in terms of physically meaningful acceleration-derived quantities involving the constrained bodies. These may be complicated expressions, but they are always built from the following fundamental quantities:

- the accelerations of points and angular accelerations of vectors fixed on the constrained bodies, relative to the ancestor or to other constrained bodies in this Constraint, or
- the cross-mobilizer accelerations directly in terms of the generalized accelerations $\dot{u}$ of the constrained mobilizers.

Simbody's Constraint base class provides utilities to efficiently obtain the constrained bodies' accelerations relative to the outmost common ancestor A given a set of $\dot{u}$'s (for fixed q and u), relative velocities given u's (for fixed q), and relative positions given q's. The user's constraint equation error functions are written using these utilities.

11.8.2 Defining a Custom Constraint in Simbody

[TODO: this is currently a design specification, not a description of existing code.]

To specify a Constraint for Simbody's use requires that the Constraint's implementor provide the following information and computations, by implementing virtual methods of the `Constraint::Custom` class:

- The set of constrained MobilizedBodies, that is, those which are referenced explicitly in the constraint error functions. Simbody uses this information to determine the set of participating mobilizers

whose coordinates can potentially affect this constraint. This is topological information, so no State is passed in.

```
ConstrainedBodyIndex
    addConstrainedBody(const MobilizedBody& B);
int getNumConstrainedBodies() const; // nb
const MobilizedBody&
    getConstrainedBody(ConstrainedBodyIndex) const;

ConstrainedMobilizerIndex
    addConstrainedMobilizer(const MobilizedBody& M);
int getNumConstrainedMobilizers() const; // nm
const MobilizedBody&
    getConstrainedMobilizer(ConstrainedMobilizerIndex) const;
```

If a constraint directly involves a mobilizer coordinate (i.e., q, u, or $\dot{u}$), the MobilizedBody containing that mobilizer should be added to the list as a constrained mobilizer. Constrained mobilizers are numbered from zero to $n_m - 1$, and referenced using the unique integer index type `ConstrainedMobilizerIndex`.

For constraint equations involving bodies rather than mobilities, add the MobilizedBody containing each of the bodies as a constrained body. These are numbered from zero to $n_b - 1$, and accessed using the unique integer index type `ConstrainedBodyIndex`. Simbody will collect this information during `realizeTopology()`. After that, if there are any constrained bodies the following method in the `Constraint` parent class can be called:

```
const MobilizedBody& getOutmostCommonAncestor() const;
```

- The number m_p of holonomic (position), m_v of nonholonomic (velocity), and m_a of acceleration-only constraint equations generated by this Constraint. The total number m of acceleration constraints and corresponding Lagrange multipliers λ is $m=m_p+m_v+m_a$.

```
void getNumConstraintEquations
   (const State&, // already realized through Stage::Model
    int& mp, int& mv, int& ma) const;
```

Simbody will collect this information during `realizeModel()`. After that the following method from the parent class is available for use by the Constraint implementation:

```
const Array<int>& // nC of these
getParticipatingMobilities(const State&) const;
                                  // already realized through Stage::Model
```

- A method for calculating the m acceleration constraint errors.

```
template<int m>
void calcAccelerationErrors
     (const State&, // already realized through Stage::Acceleration
      Vec<m>&       err) const;
```

- A method which converts a set of m Lagrange multipliers λ to the equivalent set of body and mobilizer forces.

```
template<int m>
void applyConstraintForces
     (const State&, // already realized through Stage::Position
      const Vec<m>&        multipliers,
      Vector_<SpatialVec>& bodyForces,
      Vector&              mobilityForces) const;
```

- A method for calculating the m_p+m_v velocity constraint errors, and one for the m_p position constraint errors.

```
template<int mp, int mv>
void calcVelocityErrors
     (const State&, // already realized through Stage::Velocity
      Vec<mp+mv>&   err) const;

template<int mp>
void calcPositionErrors
     (const State&, // already realized through Stage::Position
      Vec<mp>&      err) const;
```

- Optional: decorative geometry suitable for visualizing this constraint.

11.8.3 Base class services

The following methods are provided by Constraint::Custom to facilitate writing constraint equations where the outmost ancestral body A acts in place of "Ground" for the Constraint's constrained bodies. For each constrained body B_k, and each constrained mobilizer frame M_k with associated frame F_k fixed to its parent (inboard) body, we need access to the following quantities:

bodies: $^{A}X^{B_k}$, $^{A}V^{B_k}$, $^{A}A^{B_k}$

mobilizers: ${}^{F_k}X^{M_k}$, ${}^{F_k}V^{M_k}$, ${}^{F_k}A^{M_k}$, q^{M_k}, u^{M_k}, $\dot{u}^{M_k}$

Any of the above can appear in `calcAccelerationErrors()`, position and velocity quantities can appear in `calcVelocityErrors()`, and position quantities only can appear in `calcPositionErrors()` or `applyConstraintForces()`.

11.8.4 Constraint examples

TODO

12 Release notes for Simbody 0.7

12.1 What do we need in Simbody 1.0?

Basic capabilities (all operations are fast, O(n) operations unless otherwise stated):

- Given a set of observed point locations (e.g., atomic positions for a polymer or marker locations for a biomechanical system), and an internal coordinate (multibody) model, find the set of internal coordinates that best represents the given point locations (inverse kinematics).
- Given a multibody model and values for its generalized (relative) coordinates and speeds, provide the corresponding spatial locations and velocities (kinematics).
- Given values for generalized coordinates, speeds and accelerations, calculate the internal (joint) forces which would have produced those accelerations (inverse dynamics).
- Given a set of spatial forces, calculate the equivalent system of forces which act only at the generalized coordinates.

- Calculate accelerations in internal coordinates. That is, calculate instantaneous generalized speed time derivatives. (forward dynamics)
- Support both general model building (specify bodies and joints) and specialized modeling for constructing protein and nucleic acid multibody models from, e.g., pdb files.
- Calculate matrices and operators needed for Operational Space Control (kinematic Jacobian, partial velocities).
- Calculate matrices needed for calculating normal modes in internal coordinates (also needed for implicit integration). (dynamic Jacobian) This is a relatively expensive operation – inherently O(n^2) but may be even worse in the 1.0 release.
- Provide interfaces to Simbody which are callable from C++, C, and Fortran.
- Stateless design for compatibility with modeling layer.
- Provide an SD/FAST-compatible interface, both for ease of conversion and to facilitate building tests which compare Simbody results and performance with the same problem solved with SD/FAST. This should include most SD/FAST functionality including joint types, application of forces, and addition of constraints.
- Provide (or at least suggest) numerical methods, not strictly a part of the Simbody toolset, which are useful for manipulating multibody models. This includes time integration, modal analysis, root finding, initial condition analysis, and solving impulse problems.
- Provide suitable documentation enabling users to use Simbody effectively. Note that all users are programmers since Simbody is an API (callable library) rather than an application.

12.2 What are we leaving out in 1.0?

Rewrite …

- All bodies will be rigid. Deformability, if desired, will be achieved by using multiple rigid bodies interconnected with appropriate joints, constraints, and forces.
- Matrices needed for implicit integration and normal modes may be calculated by numerical differencing in 1.0, with analytic methods to follow later.
- Model building capability will be limited, in the sense that convenient, domain-specific modeling building facilities will not be ready. Fully general, "lowest common denominator" model building will be available, but will require a greater level of multibody knowledge that is ultimately necessary. As an example, an automated protein-to-torsion-angle model mapping a pdb file directly to a multibody model is feasible and useful, but probably will not be done by Simbody 1.0. Instead, early application writers may have to parse their own molecules, decide how to split them into bodies, and then call the basic multibody modeling tools to create the model.
- Constrained systems will use a method which involves factoring a potentially singular matrix whose size is the number of constraints. This yields an extremely robust solution, but would be a performance problem in highly constrained systems. This is likely insignificant in most biological systems but will come up sometimes and improvements are possible.
- Integration with existing applications (e.g. Gromacs) will be absent or incomplete.
- Performance will be suboptimal at first release.
- Wrappers for interpreted languages like Java and Python may not be available in Simbody 1.0. There will be C, Fortran, and C++ access only. We do intend to support these later.
- SD/FAST emulation will not be perfect, but should be close enough to make transition straightforward.

Acknowledgments

Simbody is built on the proverbial shoulders of giants. It inherits code from the public domain IVM molecular modeling module written at the NIH and kindly provided by Charles Schwieters[11] and the TAO robotics simulation and control code which was placed into open source and contributed to SimTK by Arachi Corporation.[12] It inherits ideas from earlier efforts such as SD/FAST and Imagiro. In turn, these packages were based on fundamental work in aerospace, robotics, and molecular dynamics by Dan Rosenthal and Michael Sherman at Symbolic Dynamics and Protein Mechanics, by Abhi Jain and Guy Rodriguez at JPL and Cal Tech,[13] and in Oussama Khatib's lab at Stanford. The Simbody effort is intended to bring the best of these ideas together (and avoid some earlier mistakes) in a form that is practical for use in physics-based simulation of biological structures over a wide range of scales.

I thank Charles Schwieters for writing IVM, providing the source code, and helping me understand the code. I am grateful to Bill Mydlowec for discovering the IVM paper in the Journal of Magnetic Resonance and pointing it out to me. Thanks also to Oussama Khatib, James Warren, K.C. Chang, and Diego Ruspini for help in obtaining TAO, and to Michael Levitt and Vijay Pande for their guidance through the thicket of biomolecular simulation. The Tinker molecular mechanics code, and in particular its author Jay Ponder, were very helpful in constructing the DuMM subsystem, whose dummness is not Jay's fault.

I thank Dan Rosenthal for patiently teaching me everything I know (plus much more which I promptly forgot) about the fascinating field of multibody dynamics, and Linda Petzold for inspiring me with her deep knowledge and intense enjoyment of the equally fascinating field of numerical integration and specifically for helping me learn to solve the system of equations (11.9)-(11.15).

This work was funded by the National Institutes of Health through the NIH Roadmap for Medical Research, Grant U54 GM072970. Information on the National Centers for Biomedical Computing can be obtained from http://nihroadmap.nih.gov/bioinformatics.

References

[1] Johnson, K.L. *Contact Mechanics*, Cambridge University Press (1985). Chapter 4, especially section 4.2.

[2] Hunt, K.H.; Crossley, F.R.E. Coefficient of restitution interpreted as damping in vibroimpact. *ASME Journal of Applied Mechanics, Series E* 42:440-445 (1975).

[3] Goldsmith, W. *Impact*, London: Arnold (1960).

[4] Marhefka, D.W.; Orin, D.E. Simulation of Contact Using a Nonlinear Damping Model, *Proc. Of the 1996 IEEE Intl. Conf. on Robotics and Automation*, Minneapolis, Minnesota (1996).

[5] Dupont, P.; Hayward, V.; Armstrong, B.; Altpeter, F. Single state elastoplastic friction models. *IEEE Trans. On Automatic Control*, 47(5):787-792 (2002).

[6] von Schwerin, R. *Multibody System Simulation: numerical methods, algorithms, and software*. Springer-Verlag Lecture Notes in Computational Science and Engineering (1999).

[7] Mitiguy, P.C.; Banerjee, A.K. Efficient simulation of motions involving Coulomb friction. *J. Guidance, Control, and Dynamics* 22(1):78-86 (1999).

[8] Ascher, U.M.; Chin, H.; Petzold, L.R.; Reich, S. Stabilization of constrained mechanical systems with DAEs and invariant manifolds. *Mechanics of Structures and Machines* 23(2):135-157 (1995).

[9] Eich, E. Convergence results for a coordinate projection method applied to mechanical systems with algebraic constraints. *SIAM J. on Numerical Analysis* 30(5):1467-1482 (1993).

[10] Baumgarte, J. Stabilization of constraints and integrals of motion in dynamic systems. *Computer Methods in Applied Mechanics and Engineering* 1:1–16 (1972).

[11] Schwieters, C.D.; Clore, G.M. Internal Coordinates for Molecular Dynamics and Minimization in Structure Determination and Refinement. *J. Magnetic Resonance* 152:288-302 (2001).

[12] Chang, K.S.; Khatib, O. Efficient Algorithm for Extended Operational Space Inertia Matrix. *Proc. of the 1999 IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems* (1999). The TAO source code is available in its original form under an unrestrictive license on Simtk.org here: https://simtk.org/home/tao_de, and in a more complete environment as a commercial product from Arachi: http://www.arachi.com.

[13] Jain, A.; Vaidehi, N.; Rodriguez, G. A Fast Recursive Algorithm for Molecular Dynamics Simulation. *J. Comput. Phys.* 106(2):258-268 (1993).