Simbody™

Theory Manual

Release 3.7

December, 2019

SimTK Simbody™ 3.7
Theory Manual

Michael Sherman

Stanford University

December, 2019

Simbody™ is part of SimTK, the open source biosimulation toolkit originating from Simbios, the NIH National Center for Physics-Based Simulation of Biological Structures at Stanford, funded under 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.

Simbody home page: https://simtk.org/home/simbody

Simbody source code: https://github.com/simbody

Simbios home page: http://simbios.stanford.edu

That handsome devil on the cover is my hero, Sir Isaac Newton.

Preface

SimTK Simbody provides a powerful multibody mechanics capability for use in biosimulation and robotics. 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, including biomechanical 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 C++ API. The full capability of this package, including bindings for other languages, will be built up in layers over time; this document covers the current capabilities and discusses future directions.

A complete multibody mechanics simulation (a biomechanical gait simulation or a molecular dynamics simulation of a protein/RNA interaction, for example) requires many layers. At the lowest level are hardware-dependent, computationally intense "inner loop" numerical methods like basic linear algebra and molecular force field computations. Built on those are numerical mathematics methods like numerical integration, nonlinear root-finding, optimization, and higher-level linear algebra. The next layer supports physics and mechanics and includes Simbody's multibody dynamics capability, as well as a variety of models for forces and constraints.

Simbody's distribution contains multibody dynamics, generally useful force and constraint models, a crude visualizer, and the lower-layer software it needs to run efficiently. However, that is still by no means a complete simulation system since it does not include domain-specific modeling or a GUI. Thus in any complete simulation tool there are typically two more layers built above Simbody: a modeling layer and a user interface that provides model building and editing, execution, and visualization of results.

Consequently, Simbody itself is intended for use by primarily by programmers who are writing domain-specific modeling tools and/or user interfaces and would like to incorporate high-quality multibody mechanics into their work. Some current open source examples are:

- OpenSim™, a musculoskeletal modeling layer and GUI developed at Stanford, http://opensim.stanford.edu,

- Gazebo, a robot simulator http://gazebosim.org from the Open Source Robotics Foundation (http://www.osrfoundation.org), and

- MMB (MacroMoleculeBuilder) developed by Samuel Flores, https://simtk.org/home/rnatoolbox.

MMB makes use of the Molmodel™ internal-coordinate molecular modeling layer for Simbody, https://simtk.org/home/molmodel.

How to use this document

This Simbody Theory Manual contains background information on simulation in general and multibody dynamics in particular, as well as detailed mathematical theory discussion and literature references describing Simbody's implementation. Generally the equation-dense parts are kept separate from the overview, so readers who want just one or the other can skim over large chunks of material.

This is not a programming manual or user guide so you will not find detailed information here about using Simbody in a program. For that, see the latest Simbody User Guide, Simbody Advanced Programming Guide, and Doxygen API documentation available from the Simbody distribution project at https://simtk.org/home/simbody (go to the Documents tab). Simbody also provides a public forum there where you can get help. Source code is maintained on GitHub at https://github.com/simbody.

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.

Table of Contents

1 Background

This is general material hopefully providing enough background for the rest of the document to make sense. Even if you are familiar with multibody dynamics, it is probably worth reading this section to see how we characterize it for Simbody users.

1.1 What is "multibody dynamics"?

Multibody mechanics (of which multibody *dynamics* is a component) 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.
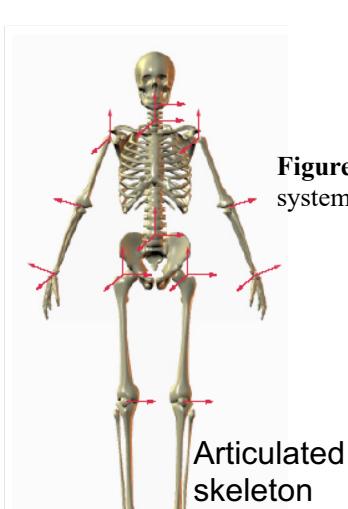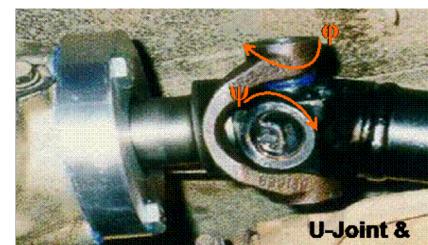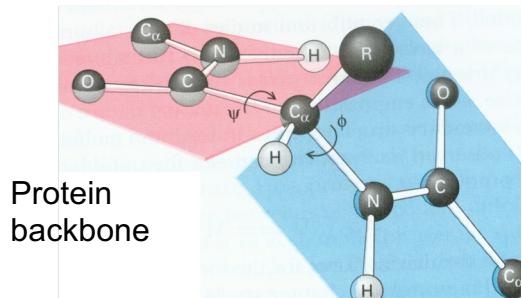
Figure 1: Some multibody systems.

Mechanical U-joint

Articulated skeleton

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. However, Simbody itself is not a research project; it is intended instead as a reliable, industrial-grade tool on which researchers may depend.

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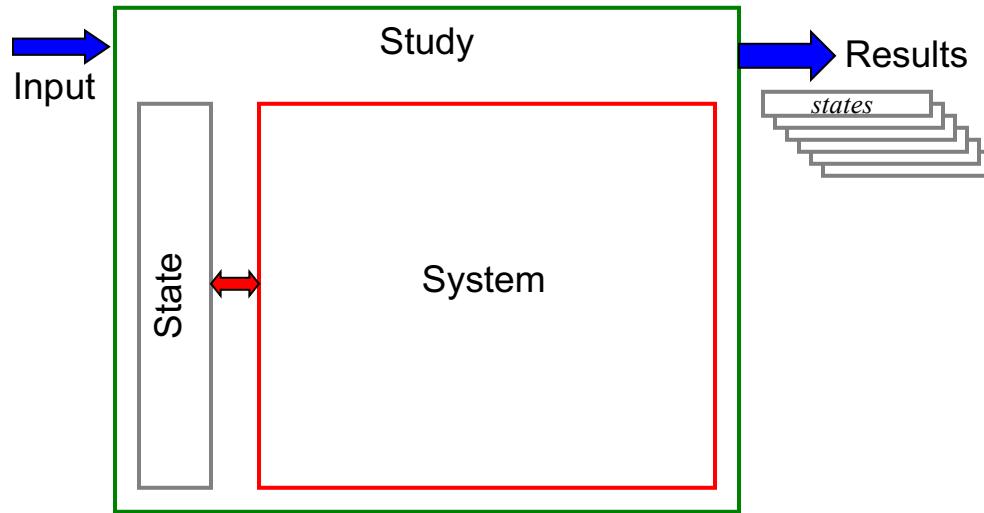
The figure below shows the primary objects involved in computational simulation of a physical system in Simbody, the infamous "three *S*'s of simulation": *System*, *State*, and *Study*. Here's our first equation:

$$\text{Simulation(Simbody)} = \text{System} + \text{State} + \text{Study}$$

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 some-

---

[*] 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.

thing 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*.
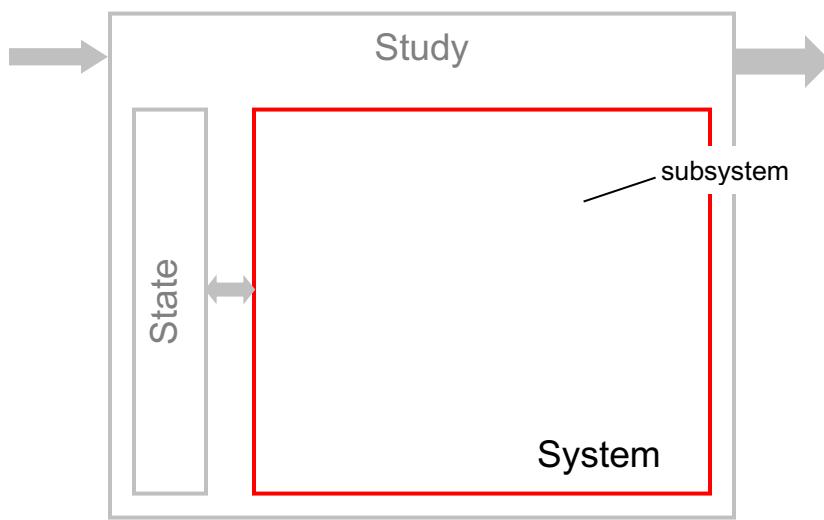
It is important to note that our notion of "state" is somewhat more general than the common use of the term. By state, we mean *everything* variable about a System. That includes not only the traditional continuous time, position and velocity variables, but also discrete variables, memory of past events, 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.

1.3 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.
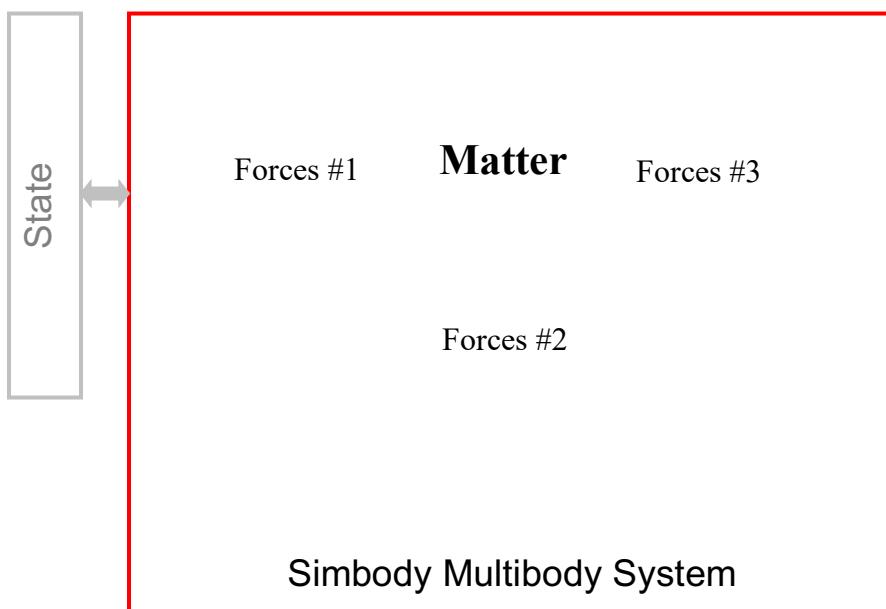
Note that by design this is *not* a hierarchical structure. It is a flat partitioning of a System into a small number of Subsystems. In a higher-level modeling layer, one would expect to find hierarchical models, which are a powerful way to represent the physical world. However, computational resources are flat, not hierarchical, and the System/Subsystem scheme is a

computational device, not a modeling system. The intent is that a modeling layer (or user program) assembles a System from a small library of Subsystems just at the point when it is ready to perform resource-intense computations.

1.4 Structure of a <u>multibody</u> system

Simbody provides some computational components (puzzle pieces) of a complete multibody mechanics System. Simbody's primary piece, the SimbodyMatterSubsystem, 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, and force subsystems that may be Simbody-provided, user-written, or application-provided. So for a multibody system, the general 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 System (including of course a MultibodySystem) 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 matter subsystem, while accelerations (dynamics) calculated by Simbody will in turn depend on the values of the forces. Section 7.1 provides details on how these interlocking computations are performed.

2 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 to define a multibody system are: coordinate frame, body, mobilizer, constraint, and force. We'll also discuss the fundamental ideas of kinematics and dynamics of a multibody system.

2.1 Coordinate frames

We define a *coordinate frame* (synonyms: *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 F_x, F_y, F_z and follow a right-handed ("dextral") convention so that $F_z = F_x \times F_y$. We label frame F's origin point F_O.

Coordinate frames are used for measuring things. We can express the location of a point P in frame F, for example, by measuring the vector $\mathbf{r}$ from F's origin to P, that is $\mathbf{r} = P{-}F_O$, 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 $^F\mathbf{r} = (r_x, r_y, r_z) = (\mathbf{r} \cdot F_x, \mathbf{r} \cdot F_y, \mathbf{r} \cdot F_z)$. 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 the expressed-in frame. Here's a picture:
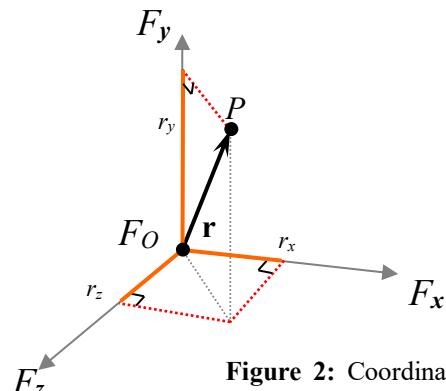
Figure 2: Coordinate frame F, and how to express the location of a point P in F.

Note that **r** is a unique physical quantity (the vector from F_O to P) but its measure numbers would be different if it were expressed in a different frame. In general we will use the $^F[Q]$ notation to indicate that some physical quantity Q is being expressed in frame F, whenever the frame is not already obvious.

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 $^F[F_x] = (1,0,0)$, $^F[F_y]=(0,1,0)$, $^F[F_z]=(0,0,1)$ and its origin point is $^F[F_O]=(0,0,0)$. 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*. (Roboticists often call the same concept a *link*.)

2.2 Bodies (links)

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

8

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 a part of a doughnut?[*] In any case the infinite extent of bodies will turn out to be very convenient when we start connecting them together.

We call the ordered set of all bodies in a multibody system B[, with the ith body designated as $B[i] \in B$. $B[i]$'s body frame is $B[i]$ with origin $B_O[i]$. 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 G_O (we'll usually drop the frame in this case and just say O) and fixed orthogonal directions $\mathbf{x} \triangleq G_x, \mathbf{y} \triangleq G_y, \mathbf{z} \triangleq G_z$. By convention, we identify ground with the "0th" 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 point B_O is the station whose measure numbers when expressed in B are (0,0,0), and its axes are the directions with measure numbers (1,0,0), (0,1,0), and (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.

---

[*] 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.

For practical purposes we assign each body a fixed property called that body's *mass structure*. The possible mass structures are: (1) ground, (2) massless, (3) particle (inertialess), (4) line (inertialess in one direction), (5) rigid body, and (6) flexible body.

2.3 Mobilizers

A *mobilizer* connects a body to its unique parent body,[*] and provides the relative *mobility* (degrees of freedom or "dofs") allowed between those bodies. Mobility expresses the *permitted* motion of a body's frame with respect to its parent's frame. Don't confuse this with the idea of *constraining* the motion of otherwise free bodies—in Simbody, bodies start out with no mobility at all, meaning that the body's frame and its parent's frame are locked together and would stay that way forever. Thus there is no motion to be constrained. Instead, Mobilizers are used to *grant* to a body the ability to move relative to its parent, allowing translation and/or rotational motion of the body frame and providing a parameterization of that motion. We call these unrestricted, parent-relative degrees of freedom a body's *mobilities*. The unique Ground body has no parent and no mobility.

2.3.1 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. We reserve the term "joint" to refer to the physical-world concept of that name, as illustrated in Figure 3. 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.

---

[*] Recall that Ground is a body.

10

Figure 3: a mechanism with four joints; at most three can be
mobilizers unless you break one of the bodies.

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. In particular, for a case like the one illustrated in Figure 3, there is a nice way to make the mobilizers correspond to the joints—break the loop by cutting one of the bodies instead of the more intuitive means of breaking one of the joints. The split body's mass properties should be divide 50/50 between the two halves (don't use a massless body – that risks making the tree part of the system singular which is not allowed in Simbody). Then a Weld constraint is added to weld the two halves back together into the original body. With this approach all the joints are mobilizers so are treated uniformly, with the motion of every joint being represented explicitly by mobilities. The alternative of modeling one of the joints with a 5-constraint equation Pin constraint can work but is unappealing; one of the joints is then modeled differently, and there are no coordinates in the system directly corresponding to the motion of that joint.

One case where it is reasonable to split a loop at the joint is when that joint is a Ball joint modeled with quaternions – in that case you can use a Ball constraint rather than a Ball mobilizer and easily obtain the quaternions from the relative orientations of the two con-

strained bodies. Other than that, though, we recommend splitting loops by cutting bodies rather than joints.

2.3.2 Types of Mobilizers

The most common 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 could 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. However, the Simbody mobilizer concept is extensible in the sense that arbitrarily complicated ones can be constructed without the use of constraints. There is, in fact, already a screw mobilizer that has only a single generalized coordinate and no constraints, but can only represent screw

motion. A more elaborate, data-driven example is a subject-specific knee joint which can be built as a 1-dof mobilizer so that a single unconstrained coordinate is used to represent the complicated coordinated rotational and translational motion of a knee.

2.3.3 Mobility space

A body can have from 0 to 6 relative mobilities (degrees of freedom) with respect to its parent body. 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 axles. 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).

2.3.4 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 coordinates are sometimes referred to as "internal coordinates," "relative coordinates," or "torsion coordinates."

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 $\dot{u}$ and refer to as *generalized accelerations*. 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 configura-

---

* *Generalized* here refers to the use of the mobility space basis where the meaning of the coordinates and speeds depends on the definition of the associated mobility. They can represent translations, rotations, or more general motions. We similarly use *generalized forces* to mean both forces, torques, or more general loads which are applied along or about mobilities.

tion, 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 sets of mobilizer parameters for body B.

2.3.5 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. Typically, structural mechanics methods are used to aggregate large nearly-rigid subsystems into deformable bodies with "assumed mode" linear deformations.

We do not provide direct support for deformable bodies yet in Simbody, but 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. Reference 1 describes how deformable bodies can be incorporated into a computational methodology like Simbody's.

2.4 Constraints

As discussed above, Simbody's mobilizers allow construction of a very small mobility space to represent all possible motion of the bodies of a multibody system. However, we will often find that even this reduced space is substantially larger than our known solution space. For example, in a multibody system where the joints form closed loops like the one shown in Figure 3, mobility space would permit solutions where the loops are not closed. 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 (or net mobility) 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, make 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.

Constraints may represent arbitrary restrictions on the generalized coordinates and generalized speeds, and linear restrictions on accelerations. Each Simbody constraint generates 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 internal forces and moments. These forces act in the same manner as the applied forces described below—they can act on bodies or along mobilities (joint axes), and as with applied forces they can

---

[*] When a multibody system represents a mechanism, the net number of degrees of freedom after accounting for constraints is conventionally called the mechanism's "mobility." We use the terms mobility and mobility space exclusively to mean the number of degrees of freedom in the *unconstrained* system. We'll say "net dofs" or "net mobility" whenever we're referring to the post-constraint leftovers.

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.

2.5 Forces

By forces we mean to include both forces and moments (torques).[*] Force vectors can be applied to the multibody system at any station on a body and moment vectors can be applied to any body (or implemented as pairs of forces). Scalar forces or moments can also be applied to the system's mobilities, that is, directly along the generalized speeds; these are called *generalized forces* or *mobility forces*. All systems of forces and moments can be reduced to an equivalent set of generalized forces, and Simbody provides an operator 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 configuration are called conservative forces, and are the gradient of some potential energy function. Non-conservative forces dependent on time and velocity exist as well but may not contribute to potential energy.

2.6 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

---

[*] 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.

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. Such problems arise, for example, when fitting a reduced-coordinate molecular model to a set of atom positions determined with X-ray crystallography. Simbody provides an Assembler solver that can solve many common inverse kinematics problems. 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.

2.7 *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 generalized forces f). *Inverse dynamics* (also called *prescribed motion*) attempts to determine what set of generalized forces explains a given set of generalized accelerations. In practice it is often useful to specify some accelerations and some forces and calculate the remaining unknowns.

Given this definition of dynamics, advancing the state through time to produce a trajectory, or searching through the state to satisfy particular objectives, are higher-level operations ("Studies") which are facilitated by the multibody dynamics capabilities described here. SimTK::TimeStepper and SimTK::Integrator objects are designed to employ Simbody dynamics calculations to generate a trajectory through time.

3 Basic Simbody numerical types

This chapter presents the basic numerical types used repeatedly in the Simbody API and in theory discussions. We'll present both the mathematical notation and definitions for these objects and the C++ classes used to manipulate them programmatically.

3.1 Vectors and Matrices

Simbody makes use of lower-level SimTK toolsets to simplify its interface and internals. The SimTK general purpose Simmatrix™ package (part of the SimTKcommon library that is part of Simbody) 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 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.

3.2 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 *positions*, *rotations*, and *transforms* (*xforms*) respectively, which locate these objects with respect to an existing frame.

3.2.1 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 position vector which would take the frame's origin to the station. A position 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.

3.2.2 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.

3.2.3 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 $^{from}R^{to}$ to represent the orientation of the "to" frame (the right superscript) measured with respect to the "from" frame (the left super-script), like this:

$$^{G}R^{B} \equiv \left(^{G}\left[B_x\right] \quad ^{G}\left[B_y\right] \quad ^{G}\left[B_z\right] \right)$$

(B_x is the x-direction unit vector of frame B, with measure numbers expressed in B's frame, while the operator $^{F}\left[\bullet\right]$ indicates that the measure numbers of some physical quantity are re-expressed in coordinate frame F.) So the symbol $^{G}R^{B}$ 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 $^{G}R^{B}$ 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 Simma-trix "~" operator to indicate matrix transpose: $\sim^{G}R^{B} = {}^{B}R^{G}$. As an example, if you have a rotation $^{G}R^{B}$ and a vector $^{B}\mathbf{v}$ expressed in B, you can re-express that same vector in G like this: $^{G}\mathbf{v} = {}^{G}R^{B}\bullet{}^{B}\mathbf{v}$. To go the other direction, we can write $^{B}\mathbf{v} = {}^{B}R^{G}\bullet{}^{G}\mathbf{v} = \sim^{G}R^{B}\bullet{}^{G}\mathbf{v}$. 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 $^{G}R^{B}$ and $^{G}R^{C}$ we can get $^{B}R^{C}$ as $^{B}R^{C} = {}^{B}R^{G}\bullet{}^{G}R^{C} = \sim^{G}R^{B}\bullet{}^{G}R^{C}$. Note that the "~" operator has a high precedence like unary "–" so $\sim^{G}R^{B}\bullet{}^{G}R^{C}$ is $(\sim^{G}R^{B})\bullet{}^{G}R^{C}$, not $\sim({}^{G}R^{B}\bullet{}^{G}R^{C})$.

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 $^{B}R^{G}\bullet{}^{G}\mathbf{v}$ and $\sim^{G}R^{B}\bullet{}^{G}\mathbf{v}$ are exactly equivalent in both meaning and performance:

the cost is 15 floating point operations (three inline dot products), with no wasted data copying or subroutine calls.

3.2.4 Transforms

Transforms combine a rotation and a position (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$^{\text{th}}$ element is 0) or points (4$^{\text{th}}$ element is 1), or composed using matrix multiplication. We define a type `Transform` which conceptually represents transforms as follows:

$$^{G}X^{B} \triangleq \begin{pmatrix} ^{G}[B_x] & ^{G}[B_y] & ^{G}[B_z] & ^{G}p^{B} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(The notation $^{G}p^{B} \equiv {}^{Go}p^{Bo}$, 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 $^{B}X^{G}$.[*]

---

[*] Note that this is actually a different definition for the "~" operator than is normally 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.

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:

$$
{}^{G}X^{B} \equiv \left(\left(\begin{array}{c} {}^{G}R^{B} \end{array} \right) \left(\begin{array}{c} {}^{G}p^{B} \end{array} \right) \\ \left(\begin{array}{cccc} 0 & 0 & 0 & 1 \end{array} \right) \right)
$$

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, ${}^{G}X^{B} = \left({}^{G}R^{B} \mid {}^{G}p^{B} \right)_{3\times4}$. 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, 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 `Transforms` (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.

3.3 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 3✕3 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 3✕3 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.

3.3.1 Spatial Notation

We also build on the Simmatrix types to define some specialized vectors and matrices useful in mechanics. Following Jain and Rodriguez5, 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 2✕2 matrix of ordinary 3✕3 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 sub-vector 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:

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

where ω is an angular velocity vector, v a linear velocity, β an angular acceleration[*], a is a linear acceleration, μ a moment (torque), and f is a force. Each of these elements is an ordinary 3-vector (`Vec3`). Sadly, orientation is not a vector quantity so we can't use an analogous `SpatialVec` $P \triangleq \begin{pmatrix} \theta \\ p \end{pmatrix}$ to represent configuration (orientation and position) of a rigid body (that is, of a reference frame). However, it can be useful to think of position this way in some circumstances.

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

$$V = {}^{G}V^{B} \triangleq \begin{pmatrix} {}^{G}\omega^{B} \\ {}^{G}v^{B_O} \end{pmatrix}, \quad A = {}^{G}A^{B} \triangleq \begin{pmatrix} {}^{G}\beta^{B} \\ {}^{G}a^{B_O} \end{pmatrix}, \quad F = {}^{G}F^{B} \triangleq \begin{pmatrix} {}^{G}\mu^{B} \\ {}^{G}f^{B_O} \end{pmatrix}$$

For spatial position, instead of the fanciful P we use the `Transform` class described above, where

$$ {}^{G}X^{B} = \left({}^{G}R^{B} \mid {}^{G_O}p^{B_O} \right) $$

with rotation matrix R playing the role of P's θ.

The above notation and somewhat atypical use of Greek symbols was chosen so that there would be an obvious way to represent these using the restrictive typographical capability of a

---

[*] We use β rather than the more conventional α for angular acceleration because a and α are too similar in many fonts, and we can use b in code rather than spelling out `alpha`.

programming language. For Greek letters we use the correspondence w=ω, b=β, m=μ, q=θ, so we can represent the above symbols in code with

```
V=[w,v], A=[b,a], F=[m,f], P=[q,p], X=[R,p]
```

(Although as mentioned above there is no actual P like this, orientation angles and quaternions are part of the generalized coordinates q so this notation is conceptually right even if pragmatically flawed.)

3.3.2 Cross product matrix

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

$$\mathbf{v}_\times = \begin{pmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{pmatrix} \tag{3.1}$$

Note that the matrix is skew-symmetric, so $\mathbf{v}_\times^\mathsf{T} = -\mathbf{v}_\times$.

We will occasionally make use of the following identities:

$$(\mathbf{v}+\mathbf{w})_\times = \mathbf{v}_\times + \mathbf{w}_\times \tag{3.2}$$

$$\mathbf{v}_\times \mathbf{w}_\times = \mathbf{w}\mathbf{v}^\mathsf{T} - \mathbf{w}^\mathsf{T}\mathbf{v}\mathbf{1}_3 \tag{3.3}$$

$$\mathbf{v}_\times^2 \triangleq \mathbf{v}_\times^\mathsf{T}\mathbf{v}_\times = -\mathbf{v}_\times\mathbf{v}_\times = \mathbf{v}^\mathsf{T}\mathbf{v}\mathbf{1}_3 - \mathbf{v}\mathbf{v}^\mathsf{T} \tag{3.4}$$

($\mathbf{1}_3$ is a 3x3 identity matrix.) Note that $\mathbf{v}_\times^\mathsf{T}\mathbf{v}_\times$ is a symmetric matrix with non-negative diagonal elements. Spelled out in scalars,

$$\mathbf{v}_\times^2 \triangleq \mathbf{v}_\times^\mathsf{T}\mathbf{v}_\times = \begin{pmatrix} v_y^2 + v_z^2 & \mathsf{T} & \mathsf{T} \\ -v_x v_y & v_x^2 + v_z^2 & \mathsf{T} \\ -v_x v_z & -v_y v_z & v_x^2 + v_y^2 \end{pmatrix} \tag{3.5}$$

where T indicates that the element is the same as the transposed one. This can also be viewed as the inertia (or gyration) matrix of a unit-mass particle located at $\mathbf{v}$, measured about the origin.

$$\mathbf{U} \cdot \mathbf{v}_\times \cdot \mathbf{U}^\mathsf{T} = (\mathbf{U} \cdot \mathbf{v})_\times,$$

$$\text{where } \mathbf{U}_{3\times3} \text{ is orthogonal.} \tag{3.6}$$

Since rotation matrices are orthogonal, equation (3.6) is particularly useful when transforming spatial quantities from one frame to another.

$$\dot{\mathbf{v}}_\times = (\dot{\mathbf{v}})_\times \tag{3.7}$$

where the overdot indicates a derivative with respect to time taken in some frame understood from context.

Identity (3.7) is primarily useful to allow us to write $\dot{\mathbf{v}}_\times$ unambiguously without concern for the typographical details of the overdot placement.

3.3.3 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 $\mathcal{J}$. It is convenient to view the inertia tensor as the product of the mass and a gyration tensor $\mathcal{G}$, such that $\mathcal{J}=m\mathcal{G}$. Then a spatial inertia matrix M can be written as a spatial gyration matrix (giving the mass distribution) scaled by the total mass:

$$M \triangleq m \begin{pmatrix} \mathcal{G} & p_\times \\ -p_\times & \mathbf{1}_3 \end{pmatrix}$$

For the spatial inertia matrix M_B of a body B about its origin B_O we have $\mathbf{p} = {}^{B_O}p^{B_C}$ so

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

Note that when the spatial mass properties are given about the center of mass B_C we have $\mathbf{p} = 0$ so

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

Where the central gyration matrix is

27

$$\mathcal{G}_B^{B_C} = \mathcal{G}_B - (\mathbf{p}^\mathsf{T}\mathbf{p}\mathbf{1}_3 - \mathbf{p}\mathbf{p}^\mathsf{T}) = \mathcal{G}_B - p_\times^\mathsf{T} p_\times = \mathcal{G}_B - p_\times^2$$

using the parallel axis theorem and then cross product matrix identities (3.4).

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

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

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

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

(Because $\mathbf{v} = \mathbf{v}^C + p_\times \boldsymbol{\omega}$ and $\mathcal{G} = \mathcal{G}^C + p_\times^\mathsf{T} p_\times$.) A body's kinetic energy (a scalar) is calculated from spatial momentum like this:

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

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

$$KE = \tfrac{1}{2}V^\mathsf{T} MV = \tfrac{1}{2}V^\mathsf{T} P = \tfrac{1}{2}m(\boldsymbol{\omega}^\mathsf{T}\mathcal{G}\boldsymbol{\omega} + \mathbf{v}^2 + 2\boldsymbol{\omega}^\mathsf{T} p_\times \mathbf{v})$$
$$= \tfrac{1}{2}V^{C\mathsf{T}} M^C V^C = \tfrac{1}{2}V^{C\mathsf{T}} P^C = \tfrac{1}{2}m(\boldsymbol{\omega}^\mathsf{T}\mathcal{G}^C\boldsymbol{\omega} + \mathbf{v}^{C2})$$

These can be shown equivalent by substituting $\mathcal{G}^C = \mathcal{G} - p_\times^\mathsf{T} p_\times$ and $\mathbf{v}^C = \mathbf{v} - p_\times \boldsymbol{\omega}$ into the last expression.

3.3.4 Spatial rotation, shifting, and transform

Objects of type Rotation and Transform have equivalent interpretations as spatial matrices:

$$\text{spatial rotation} \qquad {}^{A}R^{B} \equiv \begin{pmatrix} {}^{A}R^{B} & 0 \\ 0 & {}^{A}R^{B} \end{pmatrix} \tag{3.8}$$

$$\text{spatial shift} \qquad {}^{P}S^{Q} \equiv \begin{pmatrix} 1 & 0 \\ {}^{P}p_{\times}^{Q} & 1 \end{pmatrix} \tag{3.9}$$

$$\text{spatial transform} \quad {}^{A}X^{B} \equiv {}^{A_{O}}S^{B_{O}} \, {}^{A}R^{B} = \begin{pmatrix} {}^{A}R^{B} & 0 \\ {}^{A_{O}}p_{\times}^{B_{O}} \, {}^{A}R^{B} & {}^{A}R^{B} \end{pmatrix} \tag{3.10}$$

Then a spatial vector or spatial matrix can be rotated, shifted, or transformed using these matrices, their inverses, and their duals (inverse transpose). From the definitions above you can check that swapping the superscripts produces the inverse of each of these matrices:

$$
\begin{aligned}
{}^{B}R^{A} &\equiv \begin{pmatrix} {}^{B}R^{A} & 0 \\ 0 & {}^{B}R^{A} \end{pmatrix} & &= ({}^{A}R^{B})^{\mathsf{T}} & &= ({}^{A}R^{B})^{-1} \\[2mm]
{}^{Q}S^{P} &\equiv \begin{pmatrix} 1 & 0 \\ {}^{Q}p_{\times}^{P} & 1 \end{pmatrix} & &= \begin{pmatrix} 1 & 0 \\ -{}^{P}p_{\times}^{Q} & 1 \end{pmatrix} & &= ({}^{P}S^{Q})^{-1} \\[2mm]
{}^{B}X^{A} &\equiv \begin{pmatrix} {}^{B}R^{A} & 0 \\ {}^{B_{O}}p_{\times}^{A_{O}} \, {}^{B}R^{A} & {}^{B}R^{A} \end{pmatrix} & &= \begin{pmatrix} {}^{B}R^{A} & 0 \\ -{}^{B}R^{A} \, {}^{A_{O}}p_{\times}^{B_{O}} & {}^{B}R^{A} \end{pmatrix} & &= ({}^{A}X^{B})^{-1}
\end{aligned}
\tag{3.11}
$$

The dual operators are given by transposing the inverses, so

$$
\begin{aligned}
{}^{A}R^{*B} &\triangleq ({}^{A}R^{B})^{-\mathsf{T}} = ({}^{B}R^{A})^{\mathsf{T}} & &= {}^{A}R^{B} \\[2mm]
{}^{P}S^{*Q} &\triangleq ({}^{P}S^{Q})^{-\mathsf{T}} = ({}^{Q}S^{P})^{\mathsf{T}} & &= \begin{pmatrix} 1 & {}^{P}p_{\times}^{Q} \\ 0 & 1 \end{pmatrix} \\[2mm]
{}^{A}X^{*B} &\triangleq ({}^{A}X^{B})^{-\mathsf{T}} = ({}^{B}X^{A})^{\mathsf{T}} & &= \begin{pmatrix} {}^{A}R^{B} & {}^{A_{O}}p_{\times}^{B_{O}} \, {}^{A}R^{B} \\ 0 & {}^{A}R^{B} \end{pmatrix}
\end{aligned}
\tag{3.12}
$$

Defined this way, operators R, S, and X apply to spatial vectors in the "motion" basis, like velocities and accelerations. The dual operators $R^{*}(=R)$, S^{*}, and X^{*} apply to spatial vectors in the "force" basis, like forces and impulses. These definitions follow Featherstone[2] but have the reverse sense from Jain[3] and Schwieters[4] where the force basis is primary and the motion basis is dual (their ϕ operator is our S^{*} operator).

Using these definitions you can rotate, shift, or transform a spatial inertia matrix (rigid or articulated) M like this:

$$^{A}R *^{B}M_{B}{}^{B}R^{A} = (^{B}R^{A})^{\mathsf{T}}M_{B}{}^{B}R^{A} = {}^{A}R^{B}M_{B}{}^{B}R^{A}$$

$$^{A}S *^{B}M_{B}{}^{B}S^{A} = (^{B}S^{A})^{\mathsf{T}}M_{B}{}^{B}S^{A}$$

$$^{A}X *^{B}M_{B}{}^{B}X^{A} = (^{B}X^{A})^{\mathsf{T}}M_{B}{}^{B}X^{A}$$

See Chapter 12 for details.

4 Constructing a Simbody multibody system

The Simbody API (application programmer interface) 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 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.

4.1 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 possible mass structures are: (1) ground, (2) massless, (3) particle (inertialess), (4) line, (5) rigid body, and (6) flexible 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. A flexible body has a mass distribution that is not constant in the body's frame.

4.2 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 2 for an example.)

- Mass properties for the body, with the center of mass location measured from B_O and expressed in *B*, and the inertia (actually the unit inertia or gyration matrix $\mathcal{G}$) measured about B_O and expressed in *B*.

- The mobilizer's moving 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*, and whether to reverse the interpretation of the generalized coordinates.

Figure 4 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 $\mathcal{J}_B = m_B \mathcal{G}_B$ about B_O 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."
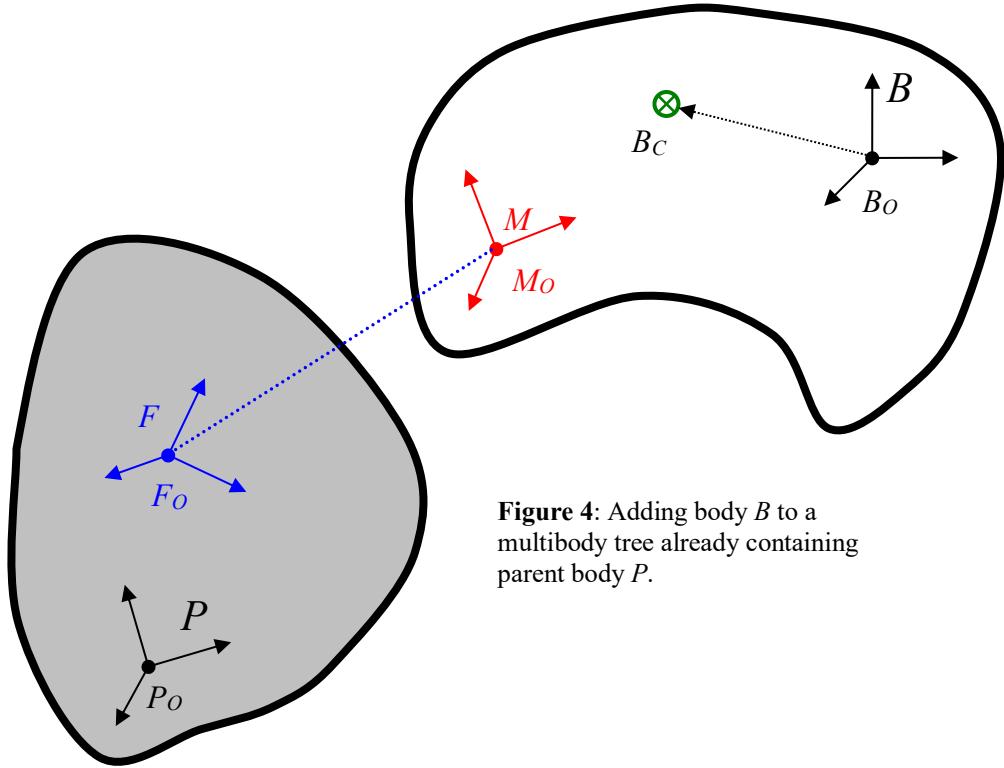
Figure 4: Adding body B to a multibody tree already containing parent body P.

4.2.1 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 5 shows the reference configuration for the mobilizer defined in Figure 4. For any mobilizer type, the values of the generalized coordinates q express a transform ${}^{F}X^{M}$ which gives the

---

[*] 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).

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 their reference configuration.
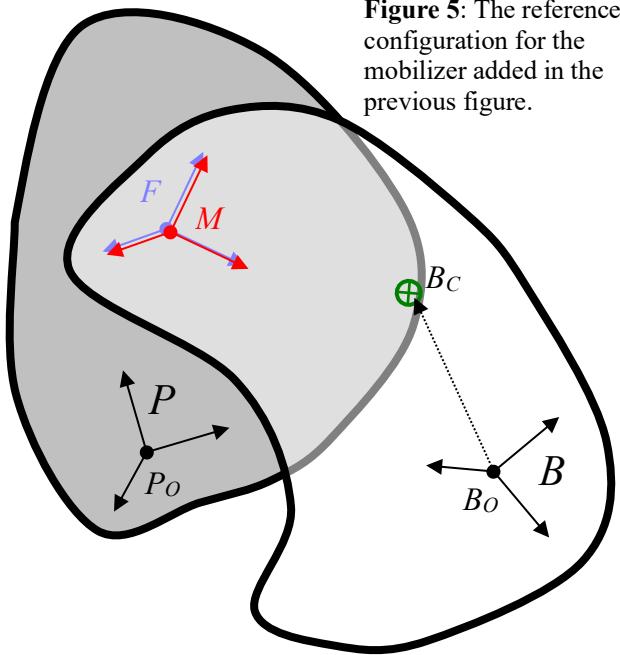
Figure 5: The reference configuration for the mobilizer added in the previous figure.

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.

4.3 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 supports a variety of built-in Constraints, and arbitrary user-defined Constraints. Some examples of built-ins 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 mobilized body index 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 A_O and expressed in A while P_B is measured from B_O 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 $^B X^{F_B}$. 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 will not 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. Simbody provides a variety of numerical methods that can be used to drive constraint violations to below a desired tolerance, at which point the associated constraints will be considered to be satisfied. After that, valid numerical studies maintain the constraint equations, and thus satisfy the Constraints, as they advance from step to step.

4.4 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 and it is easily extended.

5 Theory for Mobilizers

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 $0 \leq n \leq 6$.

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 $n_q \geq n$ 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 at the time it is added to the multibody tree: 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 ${}^P X^F$.
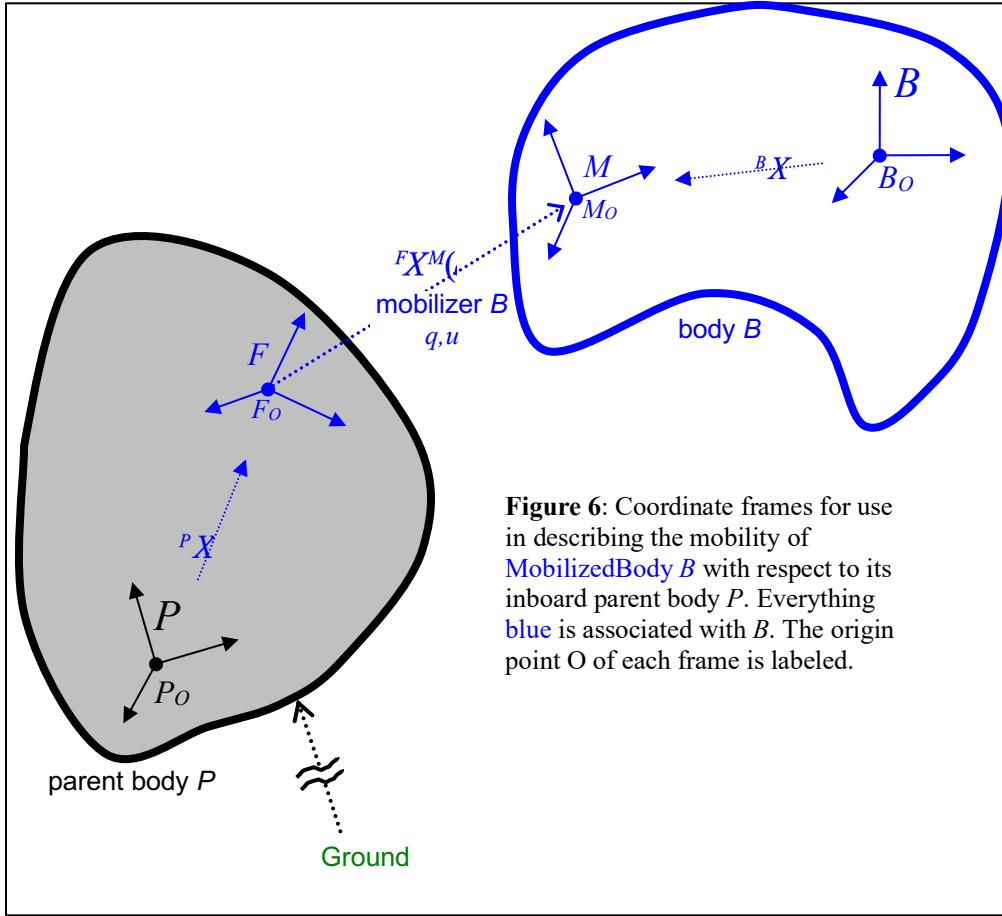
Figure 6: Coordinate frames for use in describing the mobility of MobilizedBody B with respect to its inboard parent body P. Everything blue is associated with B. The origin point O of each frame is labeled.

These are the equations that define the generalized coordinates and speeds. (The overdot notation below indicates time derivatives that are taken in the F frame.)

$$^{F}X^{M}(q) \triangleq \left(^{F}R^{M}(q) \mid ^{F}p^{M}(q) \right) \tag{5.1}$$

$$^{F}V^{M}(q,u) \triangleq \begin{pmatrix} ^{F}V_{\omega}^{M} \\ ^{F}V_{v}^{M} \end{pmatrix} = \begin{pmatrix} ^{F}\omega^{M} \\ ^{F}v^{M} \end{pmatrix} = {}^{F}\mathbf{H}^{M}(^{F}X^{M}(q))u \tag{5.2}$$

$$^{F}A^{M}(q,u,\dot{u}) \triangleq {}^{F}\dot{V}^{M} = \begin{pmatrix} ^{F}\beta^{M} \\ ^{F}a^{M} \end{pmatrix} = {}^{F}\mathbf{H}^{M}\dot{u} + {}^{F}\dot{\mathbf{H}}^{M}u \tag{5.3}$$

$$\dot{q} = \mathbf{N}(q)u \tag{5.4}$$

$$\mathrm{n}(q) = 0 \tag{5.5}$$

Note that X and $\mathbf{H}$ cannot be chosen arbitrarily, because the time derivative of X is related to V:

$$\begin{aligned} ^{F}\dot{X}^{M} &\triangleq \left(^{F}\dot{R}^{M} \mid ^{F}\dot{p}^{M} \right) \\ &= \left(^{F}\omega_{\times}^{M} \cdot {}^{F}R^{M} \mid ^{F}v^{M} \right) \\ &= \left((^{F}V_{\omega}^{M})_{\times} \cdot {}^{F}R^{M} \mid ^{F}V_{v}^{M} \right) \end{aligned} \tag{5.6}$$

This implies relationships that must hold among X, $\mathbf{H}$, and $\mathbf{N}$:

$$\begin{aligned} ^{F}\dot{R}^{M} &= {}^{F}\omega_{\times}^{M} \cdot {}^{F}R^{M} = (^{F}\mathbf{H}_{\omega}^{M}u)_{\times} \cdot {}^{F}R^{M} \\ &= \frac{\partial {}^{F}R^{M}}{\partial q}\dot{q} = \frac{\partial {}^{F}R^{M}}{\partial q}\mathbf{N}u \end{aligned} \tag{5.7}$$

$$\begin{aligned} ^{F}\dot{p}^{M} &= {}^{F}v^{M} = {}^{F}\mathbf{H}_{v}^{M}u \\ &= \frac{\partial {}^{F}p^{M}(q)}{\partial q}\dot{q} = \frac{\partial {}^{F}p^{M}(q)}{\partial q}\mathbf{N}u \end{aligned} \tag{5.8}$$

where $\mathbf{H}_{\omega}$ and $\mathbf{H}_{v}$ are the upper and lower $3\times n$ partitions of hinge matrix $\mathbf{H}$. Intuitively, this is stating the requirement that the spatial velocity produced from u by the action of $\mathbf{H}$ is the time derivative of the spatial position and orientation produced from q by the nonlinear function $X(q)$, with matrix $\mathbf{N}$ serving to mediate between u and $\dot{q}$. Note from (5.2) that $\mathbf{H}$ depends only on the transform (spatial position) represented by the set of q's, not on the definitions of the individual q's.

Equation (5.5) specifies additional constraints that q must satisfy if there are not enough equations (5.4) to uniquely specify q. That occurs when there are more q's than u's, typically because q is a quaternion in which case $\mathrm{n}(q)$ is the quaternion normalization constraint.

5.1 Reverse mobilizers

Because a tree of mobilized bodies must be ordered parent→child along each branch going away from Ground, the role of parent and child may be reversed from the desired sense for some mobilizers. When the generalized coordinates and speeds are intended to have a particular physical meaning, we would like to preserve that meaning even when exchanging the roles of parent and child. For example, if you make a "knee" mobilizer with the generalized coordinate being knee flexion, you would like to preserve that meaning regardless of whether the femur or tibia is the parent body.

So we will at times be given the mobilizer specification from frame M on body B to frame F on body P, but a mobilizer specification must go from F to M. That is, we're given $^{M}X^{F}$, $^{M}\mathbf{H}^{F}$, $^{M}\dot{\mathbf{H}}^{F}$, and $\mathbf{N}$ for a mobilizer in frame M fixed to the child body B (with time derivatives taken in M), but we want $^{F}X^{M}$, $^{F}\mathbf{H}^{M}$, $^{F}\dot{\mathbf{H}}^{M}$, and $\mathbf{N}$ describing the mobilizer with identical generalized coordinates and speeds in frame F fixed on the parent body (and with time derivatives taken in F).

$\mathbf{N}$ is easy enough—since we want q and u to retain their original meanings, $\mathbf{N}$ must stay the same. Almost as easy, $^{F}X^{M} = (^{M}X^{F})^{-1} = \left(^{F}R^{M} \mid -^{F}R^{M} \cdot ^{M}p^{F} \right)$. But we're going to have to work to get the other two matrices. First $^{F}\mathbf{H}^{M}$:

From Equation (5.2) (with the frames swapped) we have

$$^{M}V^{F} = \begin{pmatrix} ^{M}\omega^{F} \\ ^{M}v^{F} \end{pmatrix} = \begin{pmatrix} ^{M}\mathbf{H}_{\omega}^{F} \\ ^{M}\mathbf{H}_{v}^{F} \end{pmatrix} u \tag{5.9}$$

We want to find $^{F}\mathbf{H}^{M}$ such that

$$^{F}V^{M} = \begin{pmatrix} ^{F}\omega^{M} \\ ^{F}v^{M} \end{pmatrix} = \begin{pmatrix} ^{F}\mathbf{H}_{\omega}^{M} \\ ^{F}\mathbf{H}_{v}^{M} \end{pmatrix} u \tag{5.10}$$

For the moment we are going to leave the expressed-in frame as M and work only with the physical quantities. We'll re-express at the end. We know that $^{F}\omega^{M} = -^{M}\omega^{F}$ so it follows that $^{F}\mathbf{H}_{\omega}^{M} = -^{M}\mathbf{H}_{\omega}^{F}$. But the linear velocity cannot simply be negated. We have

$$ {}^{M}v^{F} \triangleq {}^{M}\dot{p}^{F} \triangleq \frac{{}^{M}d}{dt} {}^{M}p^{F} \tag{5.11}$$

$$ {}^{F}v^{M} \triangleq {}^{F}\dot{p}^{M} \triangleq \frac{{}^{F}d}{dt} {}^{F}p^{M} \tag{5.12}$$

Since ${}^{F}p^{M} = -{}^{M}p^{F}$ we can substitute into (5.12) and get

$$
\begin{aligned}
{}^{F}v^{M} &= -\frac{{}^{F}d}{dt} {}^{M}p^{F} \\
&= -({}^{M}\dot{p}^{F} + {}^{F}\omega^{M} \times {}^{M}p^{F}) \\
&= -({}^{M}v^{F} + {}^{M}p^{F} \times {}^{M}\omega^{F}) \\
&= -({}^{M}\mathbf{H}_{v}^{F} + {}^{M}p_{\times}^{F} {}^{M}\mathbf{H}_{\omega}^{F})u
\end{aligned}
\tag{5.13}
$$

Since ${}^{F}v^{M} = {}^{F}\mathbf{H}_{v}^{M}u$ we see that

$$ \left[{}^{F}\mathbf{H}_{v}^{M} \right]_{M} = -({}^{M}\mathbf{H}_{v}^{F} + {}^{M}p_{\times}^{F} {}^{M}\mathbf{H}_{\omega}^{F}) \tag{5.14}$$

In the above analysis we left the quantities expressed in the M frame as emphasized in equation (5.14) (although we took the derivative in F). Re-expressing in the F frame completes the computation of ${}^{F}\mathbf{H}^{M}$:

$$ {}^{F}\mathbf{H}^{M} = -{}^{F}R^{M} \begin{pmatrix} {}^{M}\mathbf{H}_{\omega}^{F} \\ {}^{M}\mathbf{H}_{v}^{F} + {}^{M}p_{\times}^{F} {}^{M}\mathbf{H}_{\omega}^{F} \end{pmatrix} \tag{5.15}$$

We can differentiate equation (5.15) in F to get ${}^{F}\dot{\mathbf{H}}^{M}$:

$$
\begin{aligned}
{}^{F}\dot{\mathbf{H}}^{M} = &-{}^{F}\omega_{\times}^{M} \cdot {}^{F}R^{M} \begin{pmatrix} {}^{M}\mathbf{H}_{\omega}^{F} \\ {}^{M}\mathbf{H}_{v}^{F} + {}^{M}p_{\times}^{F} {}^{M}\mathbf{H}_{\omega}^{F} \end{pmatrix} \\
&-{}^{F}R^{M} \begin{pmatrix} {}^{M}\dot{\mathbf{H}}_{\omega}^{F} \\ {}^{M}\dot{\mathbf{H}}_{v}^{F} + {}^{M}p_{\times}^{F} {}^{M}\dot{\mathbf{H}}_{\omega}^{F} + {}^{M}\dot{p}_{\times}^{F} {}^{M}\mathbf{H}_{\omega}^{F} \end{pmatrix}
\end{aligned}
\tag{5.16}
$$

Substituting from (5.12) and (5.15) gives this form for ${}^{F}\dot{\mathbf{H}}^{M}$:

$$ {}^{F}\dot{\mathbf{H}}^{M} = {}^{F}\omega_{\times}^{M} \cdot {}^{F}\mathbf{H}^{M} - {}^{F}R^{M} \begin{pmatrix} {}^{M}\dot{\mathbf{H}}_{\omega}^{F} \\ {}^{M}\dot{\mathbf{H}}_{v}^{F} + {}^{M}p_{\times}^{F} {}^{M}\dot{\mathbf{H}}_{\omega}^{F} + {}^{M}v_{\times}^{F} {}^{M}\mathbf{H}_{\omega}^{F} \end{pmatrix} \tag{5.17}$$

Algorithmically, we can avoid duplicate computations by separating the two rows of (5.15), and rearranging to use already-computed terms:

$$
{}^{F}\mathbf{H}_{\omega}^{M} = -{}^{F}R^{M}\,{}^{M}\mathbf{H}_{\omega}^{F}
\tag{5.18}
$$

$$
\begin{aligned}
{}^{F}\mathbf{H}_{v}^{M} &= -{}^{F}R^{M}\,{}^{M}\mathbf{H}_{v}^{F} - {}^{F}R^{M}\,{}^{M}p_{\times}^{F}\,{}^{M}\mathbf{H}_{\omega}^{F} \\
&= -{}^{F}R^{M}\,{}^{M}\mathbf{H}_{v}^{F} - {}^{F}R^{M}\,{}^{M}p_{\times}^{F}\left(-{}^{M}R^{F}\,{}^{F}\mathbf{H}_{\omega}^{M}\right) \\
&= -{}^{F}R^{M}\,{}^{M}\mathbf{H}_{v}^{F} + \left({}^{F}R^{M}\,{}^{M}p^{F}\right)_{\times}{}^{F}\mathbf{H}_{\omega}^{M} \\
&= -\left({}^{F}R^{M}\,{}^{M}\mathbf{H}_{v}^{F} + {}^{F}p_{\times}^{M}\,{}^{F}\mathbf{H}_{\omega}^{M}\right)
\end{aligned}
\tag{5.19}
$$

using identity (3.6) in the second-to-last step.

To create an algorithmic version of (5.17), differentiate equations (5.18) and (5.19) in F to get ${}^{F}\dot{\mathbf{H}}^{M}$ in terms of already-computed quantities:

$$
\begin{aligned}
{}^{F}\dot{\mathbf{H}}_{\omega}^{M} &= -{}^{F}R^{M}\,{}^{M}\dot{\mathbf{H}}_{\omega}^{F} - {}^{F}\omega_{\times}^{M}\,{}^{F}R^{M}\,{}^{M}\mathbf{H}_{\omega}^{F} \\
&= -{}^{F}R^{M}\,{}^{M}\dot{\mathbf{H}}_{\omega}^{F} + {}^{F}\omega_{\times}^{M}\,{}^{F}\mathbf{H}_{\omega}^{M}
\end{aligned}
\tag{5.20}
$$

$$
\begin{aligned}
{}^{F}\dot{\mathbf{H}}_{v}^{M} &= -\left({}^{F}R^{M}\,{}^{M}\dot{\mathbf{H}}_{v}^{F} + {}^{F}\omega_{\times}^{M}\,{}^{F}R^{M}\,{}^{M}\mathbf{H}_{v}^{F} + {}^{F}p_{\times}^{M}\,{}^{F}\dot{\mathbf{H}}_{\omega}^{M} + {}^{F}v_{\times}^{M}\,{}^{F}\mathbf{H}_{\omega}^{M}\right) \\
&= -\left({}^{F}R^{M}\,{}^{M}\dot{\mathbf{H}}_{v}^{F} - {}^{F}\omega_{\times}^{M}\left({}^{F}\mathbf{H}_{v}^{M} + {}^{F}p_{\times}^{M}\,{}^{F}\mathbf{H}_{\omega}^{M}\right)\right. \\
&\quad \left. + {}^{F}p_{\times}^{M}\,{}^{F}\dot{\mathbf{H}}_{\omega}^{M} + {}^{F}v_{\times}^{M}\,{}^{F}\mathbf{H}_{\omega}^{M}\right) \\
&= -\left({}^{F}R^{M}\,{}^{M}\dot{\mathbf{H}}_{v}^{F} - {}^{F}\omega_{\times}^{M}\,{}^{F}\mathbf{H}_{v}^{M}\right. \\
&\quad \left. + {}^{F}p_{\times}^{M}\,{}^{F}\dot{\mathbf{H}}_{\omega}^{M} + \left({}^{F}v_{\times}^{M} - {}^{F}\omega_{\times}^{M}\,{}^{F}p_{\times}^{M}\right){}^{F}\mathbf{H}_{\omega}^{M}\right)
\end{aligned}
\tag{5.21}
$$

Or, collecting terms

$$
\begin{aligned}
{}^{F}\dot{\mathbf{H}}^{M} &= -{}^{F}R^{M}\,{}^{M}\dot{\mathbf{H}}^{F} + {}^{F}\omega_{\times}^{M}\,{}^{F}\mathbf{H}^{M} \\
&\quad - \begin{pmatrix} 0 \\ {}^{F}p_{\times}^{M}\,{}^{F}\dot{\mathbf{H}}_{\omega}^{M} + \left({}^{F}v_{\times}^{M} - {}^{F}\omega_{\times}^{M}\,{}^{F}p_{\times}^{M}\right){}^{F}\mathbf{H}_{\omega}^{M} \end{pmatrix}
\end{aligned}
\tag{5.22}
$$

Simbody uses equations (5.18), (5.19), (5.20), and (5.22) in that order to perform these computations.

5.2 Mobilizers in body frames

At times it is more convenient to deal with the mobilizer hinge matrix 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 hinge matrix $\mathbf{H}$ defined above by the constant transforms ${}^{P}X^{F}$ and ${}^{M}X^{B}$ depicted in Figure 6. 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}:

$$ {}^{F}\mathbf{H}^{B} \triangleq \frac{\partial {}^{F}V^{B}}{\partial u} = \phi^{\mathsf{T}} \left({}^{F} \left[{}^{M}p^{B} \right] \right) \cdot {}^{F}\mathbf{H}^{M} \tag{5.23}$$

where

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

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

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:

$$ {}^{P}\mathbf{H}^{B} \triangleq \frac{\partial {}^{P}V^{B}}{\partial u} = {}^{P}R^{F} \cdot {}^{F}\mathbf{H}^{B} \tag{5.24}$$

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

$$ {}^{P}\dot{\mathbf{H}}^{B} \triangleq \frac{{}^{P}d}{dt} {}^{P}\mathbf{H}^{B} = {}^{P}R^{F} \cdot {}^{F}\dot{\mathbf{H}}^{B} = {}^{P} \left[{}^{F}\dot{\mathbf{H}}^{B} \right] \tag{5.25}$$

where

$$ {}^{F}\dot{\mathbf{H}}^{B} \triangleq \frac{{}^{F}d}{dt} {}^{F}\mathbf{H}^{B} = \dot{\phi}^{\mathsf{T}} \left({}^{F} \left[{}^{M}p^{B} \right] \right) \cdot {}^{F}\mathbf{H}^{M} + \phi^{\mathsf{T}} \left({}^{F} \left[{}^{M}p^{B} \right] \right) {}^{F}\dot{\mathbf{H}}^{M} \tag{5.26}$$

and

$$ \dot{\phi} \left({}^{A} \left[\mathbf{v}^{B} \right] \right) = \begin{pmatrix} 0 & \left({}^{A}\omega^{B} \times {}^{A} \left[\mathbf{v}^{B} \right] \right)_{\times} \\ 0 & 0 \end{pmatrix} \tag{5.27}$$

These matrices are related to the hinge matrix $\mathbf{H}^{*}$ in reference 3 as follows:

$$\mathbf{H}^* \triangleq \frac{\partial {}^G\left[{}^P V^B\right]}{\partial u} = {}^G\left[{}^P \mathbf{H}^B\right] = {}^G R^P \cdot {}^P \mathbf{H}^B \tag{5.28}$$

$$\begin{aligned}
\dot{\mathbf{H}}^* \triangleq \frac{{}^G d}{dt}\mathbf{H}^* \quad &= {}^G \dot{R}^P \cdot {}^P \mathbf{H}^B + {}^G R^P \cdot {}^P \dot{\mathbf{H}}^B \\
&= {}^G \omega_\times^P \cdot {}^G\left[{}^P \mathbf{H}^B\right] + {}^G\left[{}^P \dot{\mathbf{H}}^B\right]
\end{aligned} \tag{5.29}$$

Note that ${}^P \mathbf{H}^B$ is not *shifted* to Ground to form $\mathbf{H}^*$, but only *re-expressed* in Ground. 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.)

6 Theory for 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 kth constrained body inward to A^C the kth *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 kth branch $u_{b,k}^C$, with $u_b^C = \bigcup_k u_{b,k}^C$; the complete set of generalized speeds which can affect Constraint C is then $u^C = \{u_m^C, u_b^C\}$. These are the Constraint's $n^C = |u^C|$ *participating mobilities*. The n_q^C *participating coordinates* are similarly defined as $q_b^C = \bigcup_k q_{b,k}^C$ and $q^C = \{q_m^C, q_b^C\}$, with $n_q^C = |q^C|$ and $n_q^C \geq n^C$.

Figure 7 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_b^C are determined. Note that the mobilizers for the two black highlighted bodies are shared by branches 0 and 1.

$$u_b = u_{b,0} \cup u_{b,1} \cup u_1$$
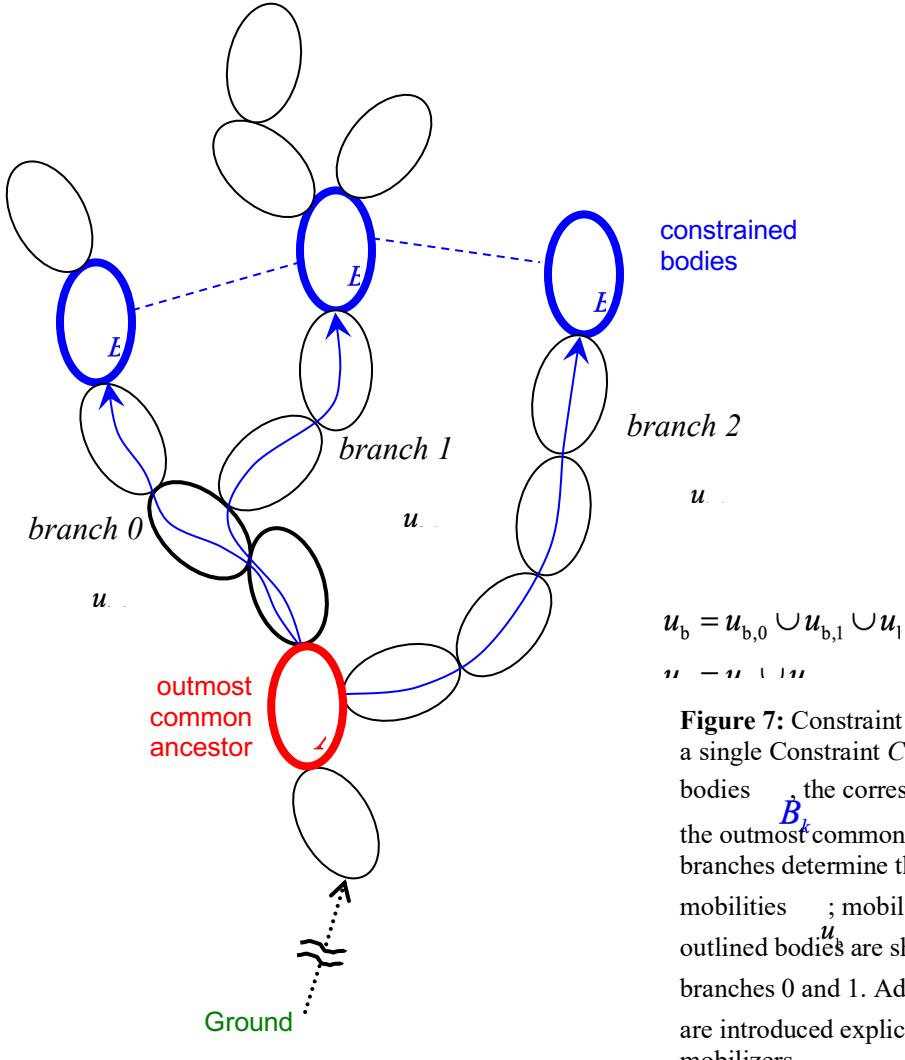
Figure 7: Constraint topology. This shows a single Constraint C with three constrained bodies B_k, the corresponding branches, and the outmost common ancestor body A. The branches determine the participating mobilities u_b; mobilities of the two black-outlined bodies are shared between branches 0 and 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 ith acceleration constraint equation has the following form:

$$g_i(t,q,u,\dot{u}) \triangleq \mathbf{g}_i \dot{u} - b_i(t,q,u) = 0 \tag{6.1}$$

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 (6.1) but rather in terms of some physical consequence of $\dot{u}_{\text{b}}$, 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 $\dot{u}_{\text{b}}$ having been supplied by Simbody after an O(*n*) computation.

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

$$f_i(q,\lambda_i) \triangleq \mathbf{g}_i^T \lambda_i \tag{6.2}$$

where f_i is a column vector function giving the *n* generalized forces generated by the scalar multiplier λ_i allocated to the ith 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 (6.2), 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 $(0 \le j < m_p)$

$$\mathrm{p}_j(t,q) = 0 \tag{6.3}$$

$$\Rightarrow \qquad \dot{\mathrm{p}}_j(t,q,u) \triangleq \qquad \mathbf{p}_j u - c_j(t,q) = 0 \tag{6.4}$$

$$\Rightarrow \qquad \ddot{\mathrm{p}}_j(t,q,u,\dot{u}) \triangleq \quad \mathbf{p}_j \dot{u} - b_{\mathrm{p},j}(t,q,u) = 0 \tag{6.5}$$

nonholonomic (velocity) constraints $(0 \le j < m_v)$

$$\mathrm{v}_j(t,q,u) = 0 \tag{6.6}$$

$$\Rightarrow \qquad \dot{\mathrm{v}}_j(t,q,u,\dot{u}) \triangleq \quad \mathbf{v}_j \dot{u} - b_{\mathrm{v},j}(t,q,u) = 0 \tag{6.7}$$

acceleration-only constraints $(0 \le j < m_a)$

$$\mathrm{a}_j(t,q,u,\dot{u}) \triangleq \quad \mathbf{a}_j \dot{u} - b_{\mathrm{a},j}(t,q,u) = 0 \tag{6.8}$$

where the row vectors are

$$\mathbf{p}_j(q) = \frac{\partial \ddot{\mathrm{p}}_j}{\partial \dot{u}} = \frac{\partial \dot{\mathrm{p}}_j}{\partial u} = \frac{\partial \mathrm{p}_j}{\partial q} \mathbf{N}^C \tag{6.9}$$

$$\mathbf{v}_j(q) = \frac{\partial \dot{\mathrm{v}}_j}{\partial \dot{u}} = \frac{\partial \mathrm{v}_j}{\partial u} \tag{6.10}$$

$$\mathbf{a}_j(q) = \frac{\partial \mathrm{a}_j}{\partial \dot{u}} \tag{6.11}$$

and the remainder terms produced by differentiation are

$$c_j(t,q) = -\frac{\partial \mathrm{p}_j}{\partial t} \tag{6.12}$$

$$b_{\mathrm{p},j}(t,q,u) = \dot{c}_j - \dot{\mathbf{p}}_j u \tag{6.13}$$

$$b_{\mathrm{v},j}(t,q,u) = -\left(\frac{\partial \mathrm{v}_j}{\partial t} + \frac{\partial \mathrm{v}_j}{\partial q} \dot{q} \right) \tag{6.14}$$

$\mathbf{N}^C$ is an $n_q^C \times n^C$ matrix assembled from a subset of the rows and columns of the global $\mathbf{N}$ such that $\dot{q}^C = \mathbf{N}^C u^C$. Note that the equations marked with blue arrows are implied by differentiation of the modeled constraint equations; they are not independent. These add another $2m_p + m_v$ 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 dis-

cussed above. Note that each of the m^C rows $\mathbf{g}_i$ in (6.1) is actually one of the rows $\mathbf{p}_j$, $\mathbf{v}_j$, or $\mathbf{a}_j$.

6.1 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 transposes. We expect to be able to calculate both $\mathbf{G}v$ and $\mathbf{G}^\mathrm{T}w$ in O($n+m$) time, where $\mathbf{G}$ is $m\mathsf{X}n$ and v and w are conformant column vectors. (Note that a straightforward matrix multiply would be O(nm), much more expensive.) Simbody uses the methods that define the constraint in combination with O(n) operators to perform these computations efficiently.

With the O($n+m$) matrix-vector multiplies available, Simbody can calculate the constraint matrices $\mathbf{P}$, $\mathbf{V}$, and $\mathbf{A}$ (collectively $\mathbf{G}$) explicitly in constant time per element. By making m calls to the provided routines, $m\mathsf{X}n$ matrices can be calculated in O($nm+m^2$)=O(nm)[*] time which is within a constant factor of optimal if you have to form these matrices.

Regardless of whether a constraint equation is initially specified at position, velocity, or acceleration level it will contribute a row $\mathbf{g}$ to the acceleration constraint matrix $\mathbf{G}$ above, which will also be a row of $\mathbf{P}$, $\mathbf{V}$, or $\mathbf{A}$. 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 (6.5), (6.7), or (6.8). Taken together, these equations are just the equations (6.1), that is, $\mathrm{g}_i(t,q,u,\dot{u}) \triangleq \mathbf{g}_i \dot{u} - b_i(t,q,u) = 0$. 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 (6.2)

---

[*] because $m{\leq}n$, $mn+m^2{\leq}2mn$

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 (6.2), 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 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 $\mathbf{b}$ in equation (6.1), and can if

necessary be determined explicitly in O(n) time using equation (6.1) 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.

7 State representation and realization

The State concept was presented in Section 1.2. In this section we will take a closer look at how we represent the state of a Simbody System and how we operate on that state when performing a Study.

7.1 Computation – realization of the State

This section provides some details about how computations are performed in the System-State-Study architecture described in Section 1.2.

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.

7.1.1 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.
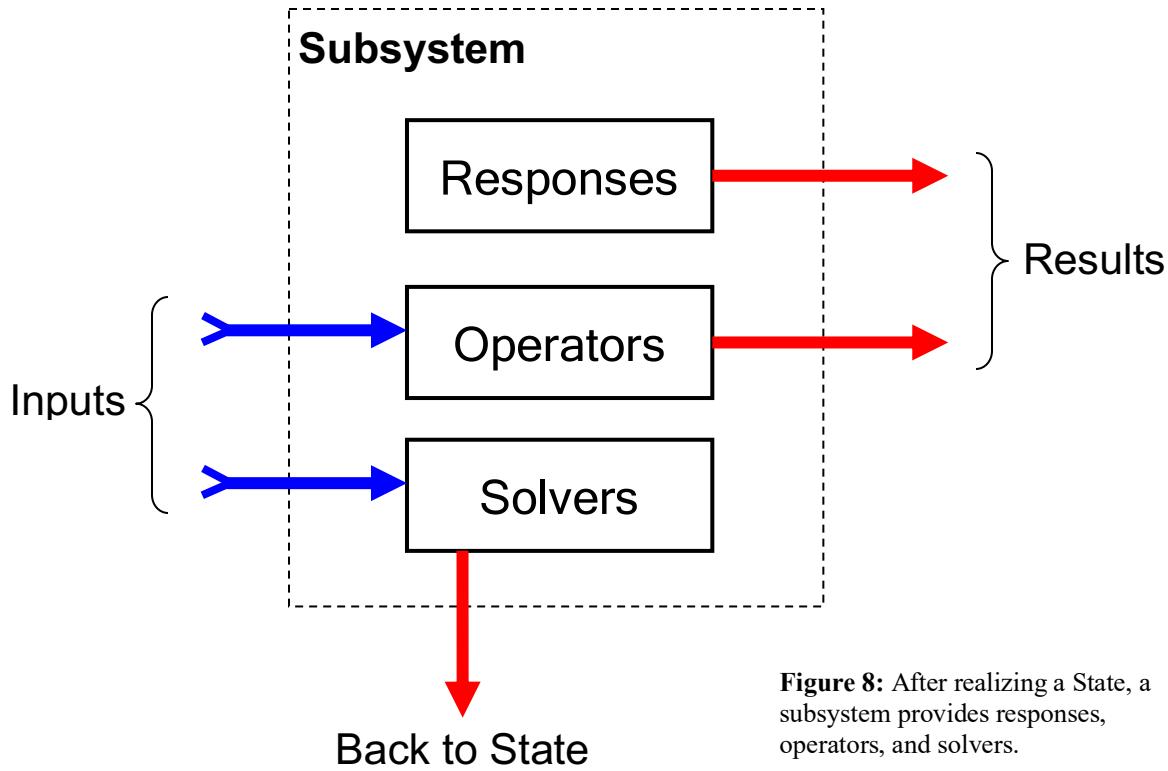
Figure 8: After realizing a State, a subsystem provides responses, operators, and solvers.

7.1.2 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*.
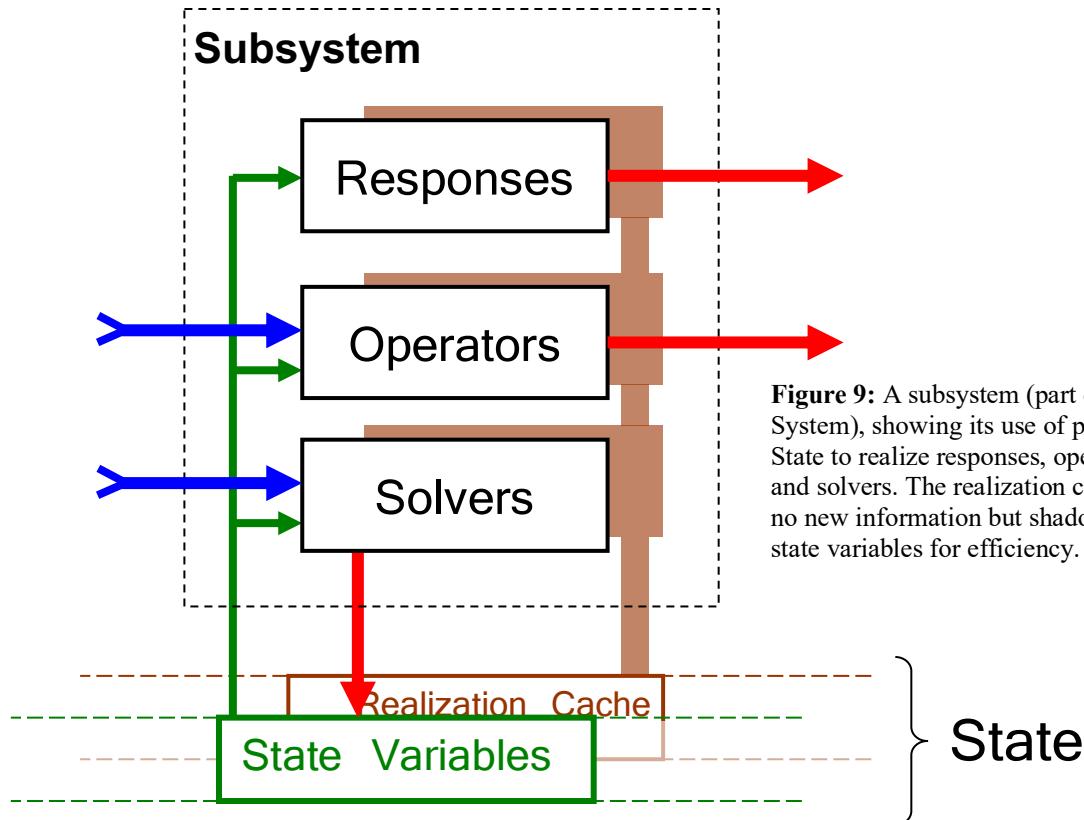
Figure 9: A subsystem (part of a System), showing its use of part of the State to realize responses, operators and solvers. The realization cache adds no new information but shadows the state variables for efficiency.

Figure 9 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.

56

7.1.3 Computing in stages

The computations performed by a System in realizing a State are naturally ordered in *stages*, and realization is done one stage at a time, in order. For example, one must compute the positions of the bodies before computing forces that may depend on those positions. This structure allows for 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.
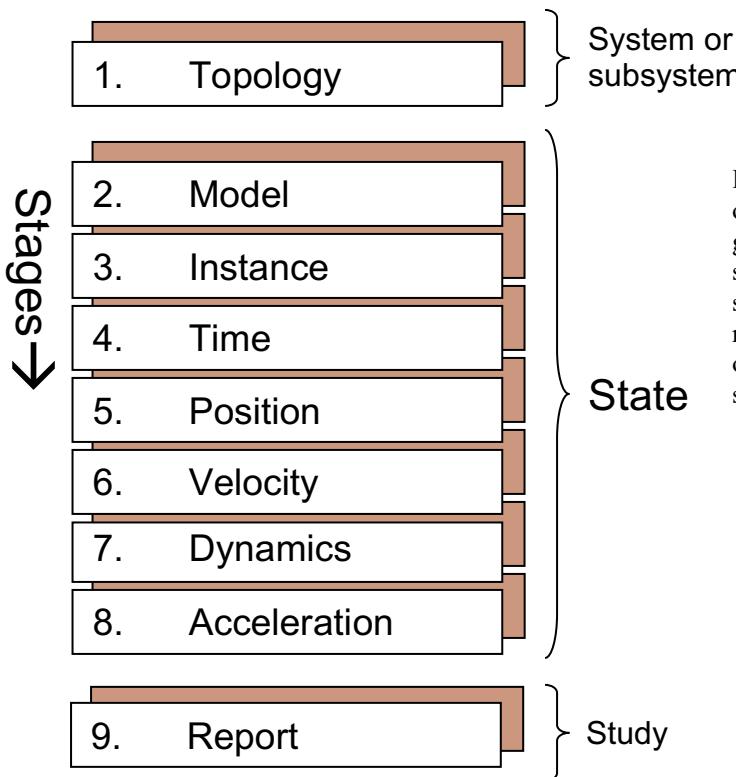
Figure 10: The conceptual organization of a computation into ordered stages. A given stage is fully realized by *each* subsystem in a System before the next stage is realized by *any* subsystem. For many purposes, construction of the (read only) System can be viewed as the initial stage of computation.

7.2 State variables

The complete state of a Simbody System is represented by a set S of broadly-defined state variables. Physically, S is stored in a SimTK State object, where it is partitioned into time $t \in S$, and two disjoint subsets $x, y \subseteq S$, so that $S \equiv \{t\} \cup x \cup y$. During time stepping, t is the independent variable; x and y both contain dependent state variables but differ in the types of variables they can contain. Partition y contains conventional real-valued, "smooth" state variables q, u, and z representing system kinematic and dynamic quantities, while partition x contains state variables of arbitrary value types including boolean, integer, and structured types of any complexity.

7.2.1 State partitioning by stage

State variables may be usefully partitioned by the computation stages they affect; we call the lowest affected computation stage the state variable's *stage*. As shown in Figure 10 stages that can be affected by state variables in S are: model, instance, time, position, velocity, force, acceleration, and report. (Topology stage is only affected by the contents of the System, not its State.) We denote state partitions with these effects S^{model}, S^{instance}, S^{time}, S^{pos}, S^{vel}, S^{force}, S^{acc}, S^{report} with

$$
\begin{aligned}
S^{\text{time}} &\triangleq x^{\text{time}} \cup \{t\} \\
S^{\text{pos}} &\triangleq x^{\text{pos}} \cup y^{\text{pos}} &= x^{\text{pos}} \cup q \\
S^{\text{vel}} &\triangleq x^{\text{vel}} \cup y^{\text{vel}} &= x^{\text{vel}} \cup u \\
S^{\text{force}} &\triangleq x^{\text{force}} \cup y^{\text{force}} &= x^{\text{force}} \cup z
\end{aligned}
\tag{7.1}
$$

For the other stages, only variables in the x partition can be used, so we have $S^{\text{model}} \equiv x^{\text{model}}$, $S^{\text{instance}} \equiv x^{\text{instance}}$, and $S^{\text{report}} \equiv x^{\text{report}}$,

Partition y consists only of position-, velocity-, and force-stage variables as shown above. For convenience, we provide more conventional names for those partitions:

$$
\begin{aligned}
q \equiv y^{\text{pos}}, \quad u \equiv y^{\text{vel}}, \quad z \equiv y^{\text{force}} \\
\text{with } y = q \cup u \cup z
\end{aligned}
\tag{7.2}
$$

q and u are the sets of generalized coordinates and speeds associated with the System's mobilized bodies. z is a set of generalized dynamic variables typically belonging to force and

control elements in the System. z's can directly affect forces, but not positions or velocities of bodies.

7.3 State resources

The SimTK::State object manages a collection of resources including state variables and cache entries needed for realization. Resources are allocated at the request of the various Subsystems, and the State object keeps them organized by Subsystem. In addition, certain kinds of state resources are aggregated into global resources pools which represent the System as a whole. Global pools support efficient numerical manipulation of the System's state as a whole in circumstances where the Subsystem composition is irrelevant.

Every State resource has an allocation stage, meaning that that resource is allocated during realization of that stage, and unallocated if that stage is invalidated later. That means that the set of resources in a System's State can vary based on the current stage, and the settings of state variables that were allocated earlier.

Resources that are always present for a given System are allocated at Topology stage. That includes the set of state variables which can affect modeling options. Then realizing Model stage causes more resources to be allocated, reflecting the modeling choices made. Some of those resources may be state variables representing parameters of the chosen model. Once those are set, realizing Instance stage allocates the final set of resources needed for further computation. No state resources may be allocated later than Instance stage; after that only the values of existing state resources may be changed.

Here are the resources supported by a State object.

Resource	Allocation	Description	Resource notes
Built-in resources	Topology	Every State has some built-in resources, allocated at Topology stage. Note that the System stage can never be higher than the lowest Subsystem stage.	t, t_{prev} System current stage Per-Subsystem current stages Low-water mark
Dynamic variable group (q,u,z)	Topology Model *size*: Instance	A contiguous array of real-valued "conventional" state variables. These are pooled into a global array y, which is itself partitioned into global q, u, and z arrays. An additional state variable holds the size of the group and can be set at Instance stage.	$\dot{y} = \{\dot{q}, \dot{u}, \dot{z}\}$, $\ddot{q}$ $y_{prev} = \{q_{prev}, u_{prev}, z_{prev}\}$ *sizes* Subsystem-private group Id Pool slots assignment

Structured variable x	Topology Model	A private variable belonging to a single subsystem and able to contain a designated value type of any kind, from boolean flag to arbitrary object. Has "invalidates" stage; must be allocated before that.	Previous value x_{prev} Subsystem-private Id only
Constraint group (qerr, uerr, udoterr, mults)	Topology Model Instance m: Instance	Allocates cache entries which are contiguous arrays of m scalars for holding the current constraint errors and multiplier values. An additional state variable is allocated to hold the size m of the group which can be set at Instance stage.	$y_{err} = \{q_{err}, u_{err}\}$ $\dot{u}_{err}, \lambda$ m Subsystem-private group Id Pool slots assignment
Event trigger group	Topology Model Instance $size$: Instance	Allocates contiguous array of scalar cache entries for event trigger function values, in global pool. The number of entries can be changed up to Instance stage.	$e, \quad e_{prev}$ # of triggers Subsystem-private group Id Pool slots assignment
Event group	Topology Model Instance $size$: Instance	Allocates a group of event ids for a subsystem, and a corresponding global pool of System-unique Ids.	Subsystem-private group Id # events
Cache entry	Topology Model Instance	Private variable belonging to a single subsystem. Can hold any designated object type. Has depends-on stage and "is valid" flag. Must be allocated before depends-on stage.	Subsystem-private Id only

The State maintains mapping information for the global resources so that one can determine which particular entity is the owner of an entry in a global pool, for any local entity where in the global pool it may be found.

7.4 Allocation of state resources

State variables are allocated by the various elements of a System. Here are the System elements and the kinds of state variables they allocate:

7.4.1 Mobilized bodies

There are n_B mobilized bodies $B[j]$ (including Ground) .

Each $B[j]$ represents a unique body and its mobilizer providing $0 \leq n[j] \leq 6$ unconstrained mobilities (degrees of freedom). The ground body $G \equiv B[0]$, and $n[0]=0$. $u[j]$ and $\dot{u}[j]$ are sets of $n[j]$ scalar generalized speeds and corresponding generalized accelerations defined by $B[j]$'s mobilizer. $q[j]$ and $\dot{q}[j]$ are the mobilizer's $n_q[j]$ generalized coordinates and their time derivatives, with $n_q[j] \geq n[j]$. q and u are related via an $n_q[j] \times n[j]$ kinematic coupling

matrix $\mathbf{N}[j]$ such that $\dot{q}[j] = \mathbf{N}[j]u[j]$. There may be a local quaternion normalization constraint $\mathbf{n}[j]$ defined, where $\mathbf{n}[j]$ depends only on $q[j]$.

For the System as a whole, we define ordered sets $u = \bigcup_j u[j]$ and $q = \bigcup_j q[j]$, and sets of their elementwise time derivative variables $\dot{u}$ and $\dot{q}$. Sizes are $n \triangleq |u| = \sum_j n[j]$ and $n_q \triangleq |q| = \sum_j n_q[j]$. We also define block diagonal $n_q \times n$ matrix $\mathbf{N} = \mathrm{diag}(\mathbf{N}[j])$ so that $\dot{q} = \mathbf{N}u$.

7.4.2 Dynamic variables z

There are n_D scalar dynamic variable sets $z[i] \in z$.

Any element in the System may allocate one or more sets of $n_z[i]$ scalar dynamic variables $z[i]$ and their corresponding time derivative variables $\dot{z}[i]$. We collect these into n_z-element aggregate ordered sets z and $\dot{z}$.

7.4.3 Structured variables d

There are n_d structured-value variables $d[i] \in d$.

These variables can be allocated by any element of the System. Each one holds an object of a particular type, but that type is arbitrary and is different for each variable. These can be as simple as boolean flags or integers to arbitrarily complex objects.

7.4.4 Constraints

There are n_C constraints $C[i]$ which can restrict the mobility of the bodies.

Each constraint defines a set of $m[i]$ constraint equations $g[i]$. These are classified as position (holonomic), velocity (nonholonomic), and acceleration constraint equations and a single constraint can generate equations at different levels. $g[i]$ is then partitioned into corresponding subsets $p[i], v[i], a[i] \subset g[i]$ of cardinality $m_p[i], m_v[i], m_a[i]$ such that $m[i] = m_p[i] + m_v[i] + m_a[i]$. We also define aggregate sets $\mathrm{P}, \mathrm{V}, \mathrm{A}$ and $\mathrm{G} = \mathrm{P} \cup \mathrm{V} \cup \mathrm{A}$ with cardinalities m_p, m_v, m_a and m, resp.; $m = m_p + m_v + m_a$.

A position constraint equation $p_k \in \mathrm{P}$ has the general form $q_k : p_k(t; q_k) = 0$ where $q_k \subseteq q$, that is, it defines an implicit algebraic relationship among a subset of the elements of q, possibly with an explicit dependence on time. If the set q_k contains just one element ($|q_k| = 1$

), then p_k defines q_k explicitly as $q_k = q_k(t)$; this is called a "prescribed position". Each position constraint p_k requires an entry in the global *qerr* pool in the State, a corresponding entry in the *uerr* pool for the time derivative equation $\dot{p}_k = 0$, an entry in *udoterr* for $\ddot{p}_k = 0$, and a λ slot in the multiplier pool.

Similarly, a velocity constraint equation $v_k \in V$ is $u_k : v_k(t, q_k; u_k) = 0$ where $u_k \subseteq u$, an implicit relationship among a subset of generalized speeds with explicit dependence on t and q. Note that there is not necessarily any correspondence between the sets q_k and u_k in a velocity constraint; u_k can depend on time and any set of q's. If $|u_k| = 1$ then v_k defines u_k explicitly as $u_k = u_k(t, q_k)$, this is called a "prescribed velocity". Each velocity constraint v_k requires an entry in the global *uerr* pool in the State, an entry in *udoterr* for $\dot{v}_k = 0$, and a λ slot in the multiplier pool.

Finally, an acceleration constraint equation $a_k \in A$ is $\dot{u}_k : a_k(t, q_k, u_k; \dot{u}_k) = 0$ where $\dot{u}_k \subseteq \dot{u}$, an implicit relationship among a subset of generalized accelerations with explicit dependence on t, q, and u. Note that there is not necessarily any correspondence between the sets u_k and $\dot{u}_k$ in a velocity constraint; $\dot{u}_k$ can depend on time and any sets of q's and u's. If $|\dot{u}_k| = 1$ then a_k defines $\dot{u}_k$ explicitly as $\dot{u}_k = \dot{u}_k(t, q_k, u_k)$, this is called a "prescribed acceleration". Each acceleration constraint a_k requires an entry in the global *udoterr* pool in the State, and a λ slot in the multiplier pool.

62

8 Equations of motion

In this chapter we'll present the equations of motion represented by a Simbody System. By equations of motion we mean the equations that determine the instantaneous rates of change for the state variables. Integrating those rates of change into a trajectory through time is a different topic and will be discussed in Chapter 10.

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, in the Ground frame unless otherwise specified. We use a right superscript to denote a quantity which applies only to a particular body or its mobilizer.

As detailed in section 7.4.1, the equations of motion will be written in terms of the set of n generalized speeds $u = \bigcup_b u[b]$ where b ranges over all the mobilized bodies, and the n_q generalized coordinates $q = \bigcup_b q[b]$, where $u[b]$ and $q[b]$ are the disjoint sets of $n[b]$ speeds and $n_q[b]$ coordinates for each body $B[b]$, which arise from the presence of its mobilizer.[*] 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 equations and associated discrete states (we call those "slow" variables) 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 freedom after constraints will be $n_{net} = n - m_{net}$ where $m_{net} \leq m$ is the number of *independent* acceleration-level constraint equations generated by the system's constraints.

---

[*] 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.

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 so that the generalized speeds are always mutually independent. The number of generalized coordinates $n_q[b] \geq n[b]$ 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 $n_{\text{quat}}[b]$ defined as follows:

$$n_{\text{quat}}[b] \equiv \begin{cases} 1, & \text{if mobilizer B}[b] \text{ uses a quaternion} \\ 0, & \text{otherwise} \end{cases} \tag{8.1}$$

Then the total number of quaternions in the system is $n_{\text{quat}} = \sum_b n_{\text{quat}}[b]$.

It should be emphasized that our presentation of the equations of motion below is a *formal* description, rather than a computational algorithm. 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.

8.1 Unconstrained dynamic systems

In a system with no constraints and where all state variables q, u, and z are defined dynamically by differential equations, the equations of motion are

$$\dot{q} = \mathbf{N}(q)u \tag{8.2}$$

$$\mathbf{n}(q) = 0 \tag{8.3}$$

$$\mathbf{M}(q)\dot{u} = \mathbf{f}_{\text{app}}(t,q,u,z) - \mathbf{f}_{\text{bias}}(q,u) \tag{8.4}$$

$$\dot{z} = \dot{z}(t,q,u,z,\dot{u}) \tag{8.5}$$

Then a time stepper study seeks to find trajectories $q(t), u(t), z(t)$ where

$$q(t) = q(t_0) + \int_{\tau=t_0}^{t} \dot{q}(\tau) \, d\tau \qquad (8.6)$$

$$u(t) = u(t_0) + \int_{\tau=t_0}^{t} \dot{u}(\tau) \, d\tau \qquad (8.7)$$

$$z(t) = z(t_0) + \int_{\tau=t_0}^{t} \dot{z}(\tau) \, d\tau \qquad (8.8)$$

Simbody forms equations (8.2)–(8.5) analytically while equations (8.6)–(8.8) must be solved numerically. Time stepping is discussed in detail in Chapter 10.

Here $\mathbf{M}_{n \times n}$ is a symmetric, positive definite mass matrix in mobility space (u-space) which captures all the inertial properties of the system in its current configuration, and $\mathbf{f}_{app}$ ($n \times 1$) is the set of all applied force and torques (including gravity) mapped into an equivalent set of n generalized forces acting along the mobilities. $\mathbf{f}_{bias}$ ($n \times 1$) is equivalent to the forces representing velocity-induced coriolis acceleration and gyroscopic terms. ($\mathbf{f}_{bias}$ is quadratic in u, and is zero if $u=0$.) We partition the applied forces $\mathbf{f}_{app}$ as follows:

$$\mathbf{f}_{app} = \mathbf{f}_{mob} + J^{\mathsf{T}} \bullet \mathbf{F}_{body} \qquad (8.9)$$

Here $\mathbf{f}_{mob}$ and $\mathbf{F}_{body}$ are the user-supplied system of forces and torques, while the kinematic Jacobian $J=J(q)$ is managed internally by Simbody. $\mathbf{F}_{body}$ is an $n_B \times 1$ "stacked" vector of spatial forces consisting of one element per body (that is, the per-body net result of all the forces and moments applied to each body), where each element is a 6-element spatial vector combining body forces and moments as described in section 3.3.1. $\mathbf{F}_{body}$ collects user-applied forces and moments that act on bodies (such as Cartesian forces on atoms). If gravity has been specified, then $\mathbf{F}_{body}$ also includes the spatial forces resulting from the gravitational model. $\mathbf{f}_{mob}$ is an $n \times 1$ vector of user-supplied scalar forces applied directly to the mobilities, such as would be used for bonded forces in a molecular model. $J^{\mathsf{T}} \bullet$ is the Cartesian-to-internal conversion operator, conceptually an $n \times n_B$ 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 $J^{\mathsf{T}} \bullet$ operator is an O(n) algorithm, and J is never formed explicitly in Simbody (unless you ask for it).

$\mathbf{N}_{n_q \times n}$ is a block diagonal, invertible mapping between generalized speeds and generalized coordinate derivatives. In practice this is mostly used to convert angular velocities to scaled quaternion derivatives or to Euler angle derivatives. The rectangular system of equations

represented by (8.2) has rank only n ($\leq n_q$), leaving quaternion lengths undetermined, so we need n_{quat} additional normalization conditions represented by (8.3) to ensure a unique solution for trajectory $q(t)$. Note that although equation (8.3) is formally a set of constraints, we consider this an unconstrained system since these constraints do not affect the physical solution.

Equation (8.4) 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. Note that z's time derivatives $\dot{z}$ can depend on $\dot{u}$ but not vice versa.

Formally, we can solve equation (8.4) for the accelerations $\dot{u}$ with

$$\dot{u} = \mathbf{M}^{-1}(\mathbf{f}_{\text{app}} - \mathbf{f}_{\text{bias}}) \tag{8.10}$$

By formally we mean, "don't actually do it that way!" There is always special structure to $\mathbf{M}$ 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* $\mathbf{M}$ would take $O(n^2)$ time since it has (roughly) $n^2/2$ unique elements, so Simbody neither forms nor factors $\mathbf{M}$ while solving equation (8.10).

As an extreme example, consider the special case of a molecular system modeled with n_a point mass atoms and Cartesian coordinates, so that $n=3n_a$. $\mathbf{M}$ 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 x,y,z coordinates, and the u's are the Cartesian velocities so $n_q=n$ ($=n_u$), $\mathbf{N}$ is an identity matrix, and $\dot{q} = u$. $\mathbf{f}_{\text{bias}}$ is always zero for this system. $\mathbf{f}_{\text{app}}$ 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 $\mathbf{M}$ 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 $\mathbf{N}$ is no longer identity and in fact not even square. However, equation (8.10) 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 $\mathbf{M}$.

8.2 Constrained systems

Constraints introduce unknown forces and moments 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 the non-prescribed 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 (8.4) like this:

$$\mathbf{M}\dot{u} + \mathbf{G}^{\mathrm{T}}\lambda = \mathbf{f}_{\text{app}} - \mathbf{f}_{\text{bias}} \tag{8.11}$$

$$\mathbf{G}\dot{u} = \mathbf{b} \tag{8.12}$$

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 constraint equations and $n=n_{\text{u}}$ is the number of generalized speeds. Equations (8.11) and (8.12) 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 (8.10). Note that because we can directly solve for $\dot{u}$ and eliminate λ, this is still just an ordinary differential equation, with $\dot{u} = \dot{u}(t,q,u,z)$.[*]

8.3 Unconstrained systems with prescribed, fast, and slow variables

A system may have motion where some or all of the generalized accelerations $\dot{u}$ are known as functions of time and state, but the corresponding generalized forces are unknown. In this

---

[*] Knocking equations (8.11) and (8.12) around a little, one can verify that $\dot{u} = \dot{u}_0 - \dot{u}_C$, where $\dot{u}_0 = \mathbf{M}^{-1}(\mathbf{f}_{\text{app}} - \mathbf{f}_{\text{bias}})$, $\dot{u}_C = \mathbf{M}^{-1}\mathbf{G}^{\mathrm{T}}\lambda$, and $\lambda = (\mathbf{G}\mathbf{M}^{-1}\mathbf{G}^{\mathrm{T}})^{-1}(\mathbf{G}\dot{u}_0 - \mathbf{b})$. In general the constraint matrix $\mathbf{G}\mathbf{M}^{-1}\mathbf{G}^{\mathrm{T}}$ can be singular, so there may be no solution, or an unlimited number of solutions, in which case Simbody will provide a least squares solution for λ.

case we'll partition the generalized accelerations $\dot{u}$ as $\dot{u} = \{\dot{u}_p, \dot{u}_f\}$ where $\dot{u}_p$ are the n_p prescribed accelerations and $\dot{u}_f$ are the n_f free accelerations driven by forces. (For notational convenience we'll treat these as though all prescribed variables follow any dynamic ones, although that ordering is not required in practice.) We then use τ $(n_p \times 1)$ to represent the unknown generalized forces associated with prescribed generalized accelerations $\dot{u}_p$. Any accelerations that are associated with fast or slow variables are prescribed to be zero.

Equation (8.4) is then replaced with equation (8.13) where there are additional unknowns for the generalized forces that implement the prescribed motions:

$$\mathbf{M}(q)\begin{pmatrix}\dot{u}_f \\ \dot{u}_p\end{pmatrix} + \begin{pmatrix}0_f \\ \tau\end{pmatrix} = \mathbf{f}_{\text{app}}(t,q,u,z) - \mathbf{f}_{\text{bias}}(q,u) \tag{8.13}$$

$$\dot{u}_p[i] = \begin{cases} \dot{u}_p[i](t,q_{pf},u_{pf}), & \text{upd}[i] = pres \\ 0, & \text{upd}[i] = fast \text{ or } slow \end{cases} \tag{8.14}$$

$$\dot{z}[i] = \dot{z}[i](t,q,u,z,\dot{u},\tau) \tag{8.15}$$

For a subset of the prescribed accelerations $\dot{u}_p$, we may also know the corresponding generalized speeds u_p analytically as a function of time and position, otherwise the generalized speeds will be free variables u_f produced by numerical integration of $\dot{u}_p$. Similarly, for a subset of the prescribed speeds we will also know prescribed coordinates q_p as a function of time only; otherwise the coordinates will be part of the free position variables q_f.

The trajectory equations (8.6) and (8.7) still hold for the free and prescribed dynamic variables, but only the free variables q_f, u_f, z need to be solved via numerical integration:

$$q(t) = \begin{pmatrix} q_f(t) \\ q_p(t) \\ q_{fast}(t) \\ q_{slow}(t) \end{pmatrix} = \begin{pmatrix} q_f(t_0) + \int_{\tau=t_0}^{t} \dot{q}_f(\tau)d\tau \\ q_p(t) \\ \text{relax}(q_{fast}) \\ q_{slow}^{(k)} \end{pmatrix} \tag{8.16}$$

$$u(t) = \begin{pmatrix} u_f(t) \\ u_p(t) \\ u_{fast}(t) \\ u_{slow}(t) \end{pmatrix} = \begin{pmatrix} u_f(t_0) + \int_{\tau=t_0}^{t} \dot{u}_f(\tau)d\tau \\ u_p(t, q_{pf}) \\ \text{relax}(q_{fast}; u_{fast}) \\ u_{slow}^{(k)} \end{pmatrix} \tag{8.17}$$

$$z(t) = z(t_0) + \int_{\tau=t_0}^{t} \dot{z}(\tau)d\tau \tag{8.18}$$

Note again that the partitioning of q, u, and $\dot{u}$ into free and prescribed variables can differ because a higher-level coordinate (position or velocity) may need to be produced by integration even if a lower one (velocity or acceleration) is prescribed.

To solve for the dynamic unknowns $\dot{u}_f$ and τ, we view the mass matrix and right hand side as being composed of partitions corresponding to the free and prescribed variables like this:

$$\begin{pmatrix} \mathbf{M}_{ff} & \mathbf{M}_{fp} \\ \mathbf{M}_{fp}^{\mathsf{T}} & \mathbf{M}_{pp} \end{pmatrix} \begin{pmatrix} \dot{u}_f \\ \dot{u}_p \end{pmatrix} + \begin{pmatrix} 0_f \\ \tau \end{pmatrix} = \begin{pmatrix} \mathbf{f}_f \\ \mathbf{f}_p \end{pmatrix} - \begin{pmatrix} \mathbf{f}_{\text{bias},f} \\ \mathbf{f}_{\text{bias},p} \end{pmatrix} \tag{8.19}$$

Multiplying out the blocks and moving the known quantities to the right hand side gives

$$\dot{u}_p = \dot{u}_p(t, q, u) \tag{8.20}$$

$$\mathbf{M}_{ff}\dot{u}_f = \mathbf{f}_f - \mathbf{f}_{\text{bias},f} - \mathbf{M}_{fp}\dot{u}_p \tag{8.21}$$

$$\tau = \mathbf{f}_p - \mathbf{f}_{\text{bias},p} - \mathbf{M}_{fp}^{\mathsf{T}}\dot{u}_f - \mathbf{M}_{pp}\dot{u}_p \tag{8.22}$$

These equations can be solved recursively in O(n) time by the method described in reference 5.

8.4 Constrained systems with prescribed motion

Prescribed motion is similar to a constraint, and in fact one of Simbody's built-in Constraint types implements prescribed motion that way. There are two differences between a constraint-based prescribed motion and the direct form described here:

1. Direct prescribed motion takes priority over any constraints. That is, first prescribed motion is satisfied, and then constraints are satisfied by modifying only the unprescribed variables.

2. Direct prescribed motion can be implemented much more efficiently. In fact, adding prescribed motion this way speeds up computation rather than slowing it down, because inverse dynamics is faster than forward dynamics.

With prescribed motion some of the generalized accelerations $\dot{u}$ are known explicitly as functions of time, q, and u. As was done in section 8.3, we'll partition the $\dot{u}$'s and generalized forces into two groups, subscripting those associated with prescribed motion with a p, and the free (force-driven) ones with an f: $\dot{u} = \{\dot{u}_f, \dot{u}_p\}$, $\mathbf{f} = \{\mathbf{f}_f, \mathbf{f}_p\}$ with $\dot{u}_f, \lambda$ and τ being the unknown dynamic quantities. Substituting into equations (8.11) and (8.12) gives

$$\mathbf{M}\begin{pmatrix} \dot{u}_f \\ \dot{u}_p \end{pmatrix} + \mathbf{G}^\mathsf{T}\lambda + \begin{pmatrix} 0_f \\ \tau \end{pmatrix} = \mathbf{f} - \mathbf{f}_{\text{bias}} \tag{8.23}$$

$$\mathbf{G}\begin{pmatrix} \dot{u}_f \\ \dot{u}_p \end{pmatrix} = \mathbf{b} \tag{8.24}$$

Now we'll partition $\mathbf{M}$ and $\mathbf{G}$ into blocks corresponding to the free and prescribed variables as follows:

$$\mathbf{M} = \begin{pmatrix} \mathbf{M}_{ff} & \mathbf{M}_{fp} \\ \mathbf{M}_{fp}^\mathsf{T} & \mathbf{M}_{pp} \end{pmatrix}, \quad \mathbf{G} = \begin{pmatrix} \mathbf{G}_f & \mathbf{G}_p \end{pmatrix} \tag{8.25}$$

Partitioning the right hand sides analogously, the equations of motion are now

$$\dot{u}_p = \dot{u}_p(t, q, u) \tag{8.26}$$

$$\begin{pmatrix} \mathbf{M}_{ff} & \mathbf{M}_{fp} \\ \mathbf{M}_{fp}^\mathsf{T} & \mathbf{M}_{pp} \end{pmatrix}\begin{pmatrix} \dot{u}_f \\ \dot{u}_p \end{pmatrix} + \begin{pmatrix} \mathbf{G}_f^\mathsf{T} \\ \mathbf{G}_p^\mathsf{T} \end{pmatrix}\lambda + \begin{pmatrix} 0_f \\ \tau \end{pmatrix} = \begin{pmatrix} \mathbf{f}_f \\ \mathbf{f}_p \end{pmatrix} - \begin{pmatrix} \mathbf{f}_{\text{bias},f} \\ \mathbf{f}_{\text{bias},p} \end{pmatrix} \tag{8.27}$$

$$\begin{pmatrix} \mathbf{G}_f & \mathbf{G}_p \end{pmatrix}\begin{pmatrix} \dot{u}_f \\ \dot{u}_p \end{pmatrix} = \mathbf{b} \tag{8.28}$$

Multiplying the blocks out and moving the known terms to the right hand side gives

$$\dot{u}_p = \dot{u}_p(t, q, u) \tag{8.29}$$

$$\mathbf{M}_{ff}\dot{u}_f + \mathbf{G}_f^{\mathsf{T}}\lambda = \hat{\mathbf{f}}_f \tag{8.30}$$

$$\mathbf{G}_f\dot{u}_f = \hat{\mathbf{b}} \tag{8.31}$$

$$\tau = \hat{\mathbf{f}}_p - \mathbf{G}_p^{\mathsf{T}}\lambda \tag{8.32}$$

where

$$\hat{\mathbf{f}}_f = \mathbf{f}_f - \mathbf{f}_{\text{bias},f} - \mathbf{M}_{fp}\dot{u}_p \tag{8.33}$$

$$\hat{\mathbf{b}} = \mathbf{b} - \mathbf{G}_p\dot{u}_p \tag{8.34}$$

$$\hat{\mathbf{f}}_p = \mathbf{f}_p - \mathbf{f}_{\text{bias},p} - \mathbf{M}_{fp}^{\mathsf{T}}\dot{u}_f - \mathbf{M}_{pp}\dot{u}_p \tag{8.35}$$

Equations (8.30) and (8.31) are solved for $\dot{u}_f$ and λ in the same manner as equations (8.11) and (8.12), using the known value of $\dot{u}_p$ to evaluate the right hand sides in equations (8.33) and (8.34). Then the resulting values for $\dot{u}_f$ and λ are substituted into equations (8.35) and (8.32), giving the final unknown τ.

$$\begin{pmatrix} \dot{u}_{f,0} \\ \tau_0 \end{pmatrix} = dyn(\mathbf{f}, \dot{u}_p) = \begin{pmatrix} \mathbf{M}_{ff}^{-1}\hat{\mathbf{f}}_f \\ \hat{\mathbf{f}}_p(\dot{u}_{f,0}) \end{pmatrix} \tag{8.36}$$

$$(\mathbf{G}_f\mathbf{M}_{ff}^{-1}\mathbf{G}_f^{\mathsf{T}})\lambda = aerr\begin{pmatrix} \dot{u}_{f,0} \\ \dot{u}_p \end{pmatrix} = \mathbf{G}_f\dot{u}_{f,0} - \hat{\mathbf{b}} \tag{8.37}$$

$$\begin{pmatrix} \dot{u}_f \\ \tau \end{pmatrix} = dyn(\mathbf{f} - \mathbf{G}^{\mathsf{T}}\lambda, \dot{u}_p) = \begin{pmatrix} \mathbf{M}_{ff}^{-1}(\hat{\mathbf{f}}_f - \mathbf{G}_f^{\mathsf{T}}\lambda) \\ \hat{\mathbf{f}}_p(\dot{u}_f) - \mathbf{G}_p^{\mathsf{T}}\lambda \end{pmatrix} \tag{8.38}$$

As before the only matrix we must form and factor explicitly is the $m\mathsf{x}m$ possibly-singular matrix $\mathbf{G}_f\mathbf{M}_{ff}^{-1}\mathbf{G}_f^{\mathsf{T}}$, which takes worst case $O(m \cdot n_f)$ time to form and $O(m^3)$ time to factor, with the worst case occurring when all the constraints are coupled. All the other matrix-vector multiplies can be performed with recursive $O(n)$ and $O(m)$ operators and the pre-scribed constraints do not contribute to this matrix.

The operators we have available for performing the above calculations are as follows. Prescribed motion provides direct calculation of $\dot{u}_p(t,q,u)$ and applied forces, converted to equivalent generalized forces $\mathbf{f}(t,q,u,z) = \begin{pmatrix} \mathbf{f}_f \\ \mathbf{f}_p \end{pmatrix}$ are available also.

From the multibody system we have these two O(n) operators:

$$dyn(\mathbf{f},\dot{u}_p) = \begin{pmatrix} \dot{u}_f \\ \tau \end{pmatrix} = \begin{pmatrix} \mathbf{M}_{ff}^{-1}(\mathbf{f}_f - \mathbf{f}_{\text{bias,f}} - \mathbf{M}_{fp}\dot{u}_p) \\ \mathbf{f}_p - \mathbf{f}_{\text{bias},p} - \mathbf{M}_{fp}^{\mathsf{T}}\dot{u}_f - \mathbf{M}_{pp}\dot{u}_p \end{pmatrix} \tag{8.39}$$

$$minv(\mathbf{f}) = \begin{pmatrix} \dot{u}_f \\ 0_p \end{pmatrix} = \begin{pmatrix} \mathbf{M}_{ff}^{-1} & 0 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} \mathbf{f}_f \\ \mathbf{f}_p \end{pmatrix} \tag{8.40}$$

From the system of constraints we also have two O(n) operators:

$$aerr(\dot{u}) = \begin{pmatrix} \mathbf{G}_f & \mathbf{G}_p \end{pmatrix}\begin{pmatrix} \dot{u}_f \\ \dot{u}_p \end{pmatrix} - \mathbf{b} \tag{8.41}$$

$$frc(\lambda) = \begin{pmatrix} \mathbf{f}_{\text{cons},f} \\ \mathbf{f}_{\text{cons},p} \end{pmatrix} = \begin{pmatrix} \mathbf{G}_f^{\mathsf{T}} \\ \mathbf{G}_p^{\mathsf{T}} \end{pmatrix}\lambda \tag{8.42}$$

These are used to calculate all the above terms efficiently, especially the $m \times m$ matrix $\mathbf{W}_f = \mathbf{G}_f \mathbf{M}_{ff}^{-1}\mathbf{G}_f^{\mathsf{T}}$ which we calculate by noting

$$\mathbf{W}_f = \mathbf{G}_f \mathbf{M}_{ff}^{-1}\mathbf{G}_f^{\mathsf{T}} = \begin{pmatrix} \mathbf{G}_f & \mathbf{G}_p \end{pmatrix}\mathbf{X}$$

$$\text{where} \qquad \mathbf{X}_{n \times m} = \begin{pmatrix} \mathbf{M}_{ff}^{-1}\mathbf{G}_f^{\mathsf{T}} \\ 0_{pm} \end{pmatrix} = \begin{pmatrix} \mathbf{M}_{ff}^{-1} & 0_{fp} \\ 0_{pf} & 0_{pp} \end{pmatrix}\begin{pmatrix} \mathbf{G}_f^{\mathsf{T}} \\ \mathbf{G}_p^{\mathsf{T}} \end{pmatrix} \tag{8.43}$$

Our goal is to calculate columns of $\mathbf{X}$ one column at a time and then use the $aerr()$ operator to calculate a column of $\mathbf{W}_f$.

First make one call to the $dyn()$ operator (8.39) to calculate $\dot{u}_{f,0}$ in equation (8.36). Then make one call to the $aerr()$ operator (8.41) with a zero argument to calculate the bias term $\mathbf{b}$:

$$aerr_0 = aerr\begin{pmatrix} 0_f \\ 0_p \end{pmatrix} = -\mathbf{b} \tag{8.44}$$

Then use a call to equation (8.42) to form columns of $\mathbf{G}^{\mathsf{T}}$ explicitly, one at a time:

$$\mathbf{G}_i^{\mathsf{T}} = frc(\lambda^i) = \mathbf{G}^{\mathsf{T}}\lambda^i \tag{8.45}$$

where λ^i is a unit vector of length m with the ith element 1 and the rest 0.

We then call the *minv*() operator in equation (8.40), followed by a call to *aerr*() to calculate a column $\mathbf{X}_i$:

$$\mathbf{X}_i = \begin{pmatrix} \mathbf{M}_{f\!f}^{-1}\mathbf{G}_i^{\mathsf{T}} \\ \mathbf{0}_p \end{pmatrix} = minv(\mathbf{G}_i^{\mathsf{T}}) \tag{8.46}$$

and then $\qquad \mathbf{W}_f(i) = \mathbf{G}_f\mathbf{X}_i = aerr(\mathbf{X}_i) - aerr_0 \tag{8.47}$

to build up $\mathbf{W}_f = \mathbf{G}_f\mathbf{M}_{f\!f}^{-1}\mathbf{G}_f^{\mathsf{T}}$ in $O(m \cdot n)$ operations as needed to solve equation (8.37) for a least squares value for λ. That is used to calculate constraint forces $\mathbf{G}^{\mathsf{T}}\lambda$ using the *frc*() operator again. These are subtracted from the applied forces for a final call to the *dyn*() operator to solve equation (8.38) for $\dot{u}_f$ and τ.

8.5 Constrained systems as specified to Simbody

Equations (8.12) and (8.31) are written in terms of linear constraints on the accelerations $\dot{u}$. However, in most cases general 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 (8.12) or (8.31), along with any constraints which may have been defined directly at the acceleration level. Similarly in the case of prescribed motions given at the position or velocity level, we differentiate twice or once and then use the result as equation (8.29).

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 the problem and solve them along with the ODE (8.11), (8.12). That results in a system of mixed differential and algebraic equations, known as a DAE. Equations (8.48)-

(8.54) 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.

$$\dot{q} = \mathbf{N}(q)u \qquad (8.48)$$

$$\mathbf{n}(q) = 0 \qquad (8.49)$$

$$\mathbf{M}(q)\dot{u} + \mathbf{G}^{\mathrm{T}}\lambda = \mathbf{f}(t,q,u,z) - \mathbf{f}_{\text{bias}}(q,u) \qquad (8.50)$$

$$\mathbf{a}(t,q,u;\dot{u}) = 0 \qquad (8.51)$$

$$\mathbf{v}(t,q;u) = 0 \qquad (8.52)$$

$$\mathbf{p}(t;q) = 0 \qquad (8.53)$$

$$\dot{z} = \dot{z}(t,q,u,z) \qquad (8.54)$$

Constraint coupling matrix $\mathbf{G}_{m \times n}$ is obtained from equations (8.51)-(8.53) as discussed below. Note also that the constraint equations (8.51)-(8.53) combine both general constraints and prescribed motion constraints, each subject to different restrictions on the allowable forms for the constraint functions. We are showing only dynamic variables here; there are also relaxation conditions for fast variables and discrete conditions for slow variables.

Equations (8.48)-(8.54) 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 (8.49), (8.52), and (8.53) are satisfied.

The function $\mathbf{a}_{m_a \times 1}$ in equation (8.51) specifies m_a *acceleration* (index 1) constraints, which are required to be linear in the accelerations $\dot{u}$, with $m_a \times n$ coefficient matrix $\mathbf{A}$. These have application, for example, in some models of Coulomb friction[6] and in producing simulations which must track measured accelerations or reaction forces. It is common to prescribe an acceleration to zero to temporarily lock a joint.

The function $\mathbf{v}_{m_v \times 1}$ in equation (8.52) 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 $\mathbf{v}$ must also be obeyed since, like $\mathbf{a}$, they restrict the allowable values of $\dot{u}$ and in general they will be coupled to $\mathbf{a}$. Prescribed velocity is often used to force a body to rotate at a constant rate.

The function $\mathbf{p}_{m_\mathrm{p} \times 1}$ in equation (8.53) 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 $\mathbf{v}$ and $\mathbf{a}$.

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

The system of equations (8.48)-(8.54) contains $n_\mathrm{q} + n_\mathrm{u} + n_\mathrm{z} + m$ equations in the $n_\mathrm{q} + n_\mathrm{u} + n_\mathrm{z} + m$ 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 7. For Simbody we advocate and actively support the method known as *coordinate projection*,[8] which is very accurate and reliable in practice. It is also possible to use the popular but less robust technique called Baumgarte stabilization[9]. In the next section we'll discuss how we go about solving equations (8.48)-(8.54).

8.6 Unilateral constraints

Normally a constraint generates whatever constraint forces are necessary in order to satisfy the acceleration-level expression of that constraint. Some constraints are limited, however, in the forces they can generate. Most commonly this occurs when constraints are used to represent contact between bodies. In those cases, the constraint can produce "pushing" forces but not "pulling" forces. Other common examples are joint stops, ratchets, and ropes. These constraints are capable only of producing forces λ of one sign, and their characteristic constraint equations are inequalities rather than the equalities shown in equations (8.51)–(8.53). By convention we'll require that all the constraint errors and λ are nonnegative; if they are actually nonpositive we can negate them so that we only need to consider nonnegativity here. Now when solving for the constraint forces λ we must first determine which of the unilateral constraints are active. These extra unknowns (that is, whether each constraint is active) are resolved by a complementarity condition that must hold

for every acceleration-level unilateral constraint equation: if the force is nonzero then the constraint error must be zero (constraint is active), or if the constraint error is nonzero then the force must be zero (constraint is inactive). Thus the product $g_i\lambda_i = 0$ for every unilateral constraint equation i, where g_i is the acceleration-level constraint error (i.e., it is a_i, $\dot{v}_i$, or $\ddot{p}_i$ depending on the kind of constraint). So we have the conditions

$$g_i \geq 0, \quad \lambda_i \geq 0, \quad g_i\lambda_i = 0 \quad \forall i \tag{8.55}$$

Combined with the always-active acceleration-level equality constraint equations, (8.55) is in the form of a *mixed complementarity problem* (MCP). In cases where g_i and λ_i are linear functions, this is a *(mixed) linear complementarity problem* (LCP) for which very efficient solution methods exist. In practice, all our inequality constraints will be linear except for friction, which can be quadratic.

An enabled, unilateral holonomic constraint will be in one of these seven states:

State	Conditions	Action
violated	$p < -tol$	Illegal; must correct.
separated	$p > tol$	Ignore constraint.
separating	$\|p\| \leq tol, \quad \dot{p} > vtol$	
inactive	$\|p\| \leq tol, \quad \|\dot{p}\| \leq vtol, \quad \ddot{p} \geq 0$ and $\lambda = 0$	
impacting	$\|p\| \leq tol, \quad \dot{p} < -vtol$ Or, impact declared by time stepper.	Perform impulsive MCP; new $\dot{p} \geq -vtol$ → separating or candidate.
candidate	$\|p\| \leq tol, \quad \|\dot{p}\| \leq vtol, \quad \ddot{p}, \lambda$ unknown	Perform MCP to determine $\ddot{p}, \lambda$
active	$\|p\| \leq tol, \quad \|\dot{p}\| \leq vtol, \quad \ddot{p} = 0$ and $\lambda > 0$	Obey constraint.

Holonomic unilateral constraints can generate impulses, that is, discontinuous changes to the velocity state variables due to impacts. The response to an impact requires user specification of the amount of dissipation that should occur; this results in a coefficient of restitution $0 \leq e \leq 1$ that is used to impulsively change the velocities. If the resulting rebound velocity is small enough, the constraint becomes a candidate to become active; otherwise the surfaces separate after impact.

An enabled, unilateral nonholonomic constraint will be in one of these five states:

State	Conditions	Action
violated	$v < -vtol$	Illegal; must correct.
separated	$v > vtol$	Ignore constraint.
inactive	$|v| \leq vtol, \quad \dot{v} \geq 0$ and $\lambda = 0$	
candidate	$|v| \leq vtol, \quad \dot{v}, \lambda$ *unknown*	Perform MCP to determine $\dot{v}, \lambda$
active	$|v| \leq vtol, \quad \dot{v} = 0$ and $\lambda > 0$	Obey constraint.

No state variable discontinuity occurs when a unilateral nonholonomic constraint becomes active, although the acceleration will have a jump.

An enabled, unilateral acceleration-only constraint will be in one of these three states:

State	Conditions	Action
candidate	a, λ *unknown*	Perform MCP to determine a, λ
inactive	$a \geq 0$ and $\lambda = 0$	Ignore constraint.
active	$a = 0$ and $\lambda > 0$	Obey constraint.

Note that a unilateral acceleration-only constraint is always a candidate and can never be violated.

8.6.1 Solving for impacts

When we have determined that an impact has occurred, we need to make a discontinuous change to the system's generalized speeds to avoid violating constraints. These speed changes result from applying an impulse to the system. An impulse is the integral of the forces applied during the collision, which is considered to be infinitesimally fast for rigid contact so that the system configuration (value of q) does not change during the impact. Further, during an impact we assume that all applied forces are insignificant except contact forces, so only those contribute to the impulse integral. Contact forces result either from the enforcement of constraints (including unilateral and bilateral constraints and mobilizers), or from the application of sliding friction forces.

Impact events have a coefficient of restitution e that determines how much energy is dissipated during the collision. When $e=1$, the collision is conservative, and when $e=0$ it is maximally dissipative. In between some fraction of the energy is lost and there are three common ways to interpret e: as the ratio of rebound speed to impact speed (Newton's coefficient); as the ratio of expansion impulse to compression impulse (Poisson's coefficient); and as the square root of the ratio of the final energy to the initial energy (Stronge's coefficient). These are all equivalent in simple collisions, but for multibody systems Newton's interpretation will often produce non-physical behavior. Poisson's produces consistent behavior though Stronge claims that only his interpretation can be guaranteed not to add energy. It is not clear whether Stronge's interpretation can be implemented in practice, however, so we use Poisson's.

The basic procedure for a frictionless impact is as follows:

1. Determine that a contact event has occurred at time t^- that would violate a constraint c.

2. Determine the collision velocity that must be eliminated to avoid constraint violation, call it v_c^- (< 0). If v_c^- is below a small "capture velocity" we set coefficient of restitution $e=0$, otherwise we use a supplied value or velocity-dependent calculation for e.

3. Compression: calculate the consistent impulse that would just drive v_c to zero, call that I. (By consistent we mean that it satisfies the equations of motion and all constraint conditions.)

4. Expansion: Apply the impulse $(1+e)I$ and calculate the resulting change in generalized speeds Δu.

5. Update $u^+ = u^- + \Delta u$ and use that to calculate the post-impact velocity v_c^+.

6. If $v_c^+ = 0$ (to a tolerance), examine its time derivative a_c^+. If $a_c^+ < 0$ activate constraint c.

Assume first that only constraint c is unilateral and that all other constraints are active. We need to integrate the equations of motion over the infinitesimal interval t^- to t^+. Since time and state don't change during the impact, only contact force-dependent terms can change during the integration interval. That leaves us with these impulsive equations of motion:

$$\Delta u_p = 0 \tag{8.56}$$

$$\begin{bmatrix} \mathbf{M}_{ff} & \mathbf{M}_{fp} \\ \mathbf{M}_{fp}^{\mathsf{T}} & \mathbf{M}_{pp} \end{bmatrix} \begin{pmatrix} \Delta u_f \\ 0 \end{pmatrix} + \begin{bmatrix} \mathbf{G}_f^{\mathsf{T}} & \mathbf{c}_f^{\mathsf{T}} \\ \mathbf{G}_p^{\mathsf{T}} & \mathbf{c}_p^{\mathsf{T}} \end{bmatrix} \lambda_I + \begin{pmatrix} 0_f \\ \tau_I \end{pmatrix} = \begin{bmatrix} f_{I,f} \\ f_{I,p} \end{bmatrix} \tag{8.57}$$

$$\begin{bmatrix} \mathbf{G}_f & \mathbf{G}_p \\ \mathbf{c}_f & \mathbf{c}_p \end{bmatrix} \begin{pmatrix} \Delta u_f \\ 0 \end{pmatrix} = \begin{bmatrix} 0 \\ -v_c^- \end{bmatrix} \tag{8.58}$$

where the unknowns are shown in red. These are Δu_f, the change in generalized speeds, λ_I, the constraint impulses, and τ_I, the prescribed motion impulses. The f_I are the sliding friction impulses applied during the impact. We have added rows to the constraint matrix corresponding to the new constraint c, with right hand side set to the change we want to make in the constraint violation velocity. For Poisson restitution, we will always want to calculate the impulse that will make $v_c^+ = 0$. Note that the prescribed motions in (8.56), applied and inertial forces in (8.57), and the right hand side of (8.58) are all zero here, but otherwise these are the same as the acceleration-level equations of motion with the unknowns relabeled. Thus we can solve these equations using the same operators and techniques as for the original equations. Multiplying these out we get

$$\mathbf{W}\lambda_I = \begin{bmatrix} 0 \\ \begin{bmatrix} \mathbf{G}_f \\ \mathbf{c}_f \end{bmatrix} \mathbf{M}_{ff}^{-1} f_{I,f} + v_c^- \end{bmatrix} \tag{8.59}$$

$$\Delta u_f = \mathbf{M}_{ff}^{-1} (f_{I,f} - \begin{bmatrix} \mathbf{G}_f^{\mathsf{T}} & \mathbf{c}_f^{\mathsf{T}} \end{bmatrix} \lambda_I) \tag{8.60}$$

$$\tau_I = f_{I,p} - \mathbf{M}_{fp}^{T} \Delta u_f - \begin{bmatrix} \mathbf{G}_p^{\mathsf{T}} & \mathbf{c}_p^{\mathsf{T}} \end{bmatrix} \lambda_I \tag{8.61}$$

where
$$\mathbf{W} = \begin{bmatrix} \mathbf{G}_f \\ \mathbf{c}_f \end{bmatrix} \mathbf{M}_{ff}^{-1} \begin{bmatrix} \mathbf{G}_f^{\mathsf{T}} & \mathbf{c}_f^{\mathsf{T}} \end{bmatrix} \tag{8.62}$$

Note that we do not need to determine the prescribed motion impulses in order to solve for the constraint impulses and changes to the generalized speeds. As before, $\mathbf{W}$ may be singular so we choose a least squares solution to λ_I; any choice for λ_I that solves (8.59) will produce the same Δu_f.

8.6.2 Sliding friction forces and impulses

Using Coulomb's friction law results in sliding friction forces or impulses of the form

$$\mathbf{F}_{slip} = \sigma \mathbf{e} \tag{8.63}$$

with sliding friction multiplier σ a nonnegative scalar and $\mathbf{e}$ a unit vector in the slip direction (or impending slip direction). The sliding friction multiplier is defined as

$$\sigma = \mu N \tag{8.64}$$

where scalar $N \geq 0$ is a normal force or impulse magnitude and $\mu \geq 0$ is a scalar (often velocity-dependent) representing the coefficient of friction. The dimensionality of slip direction $\mathbf{e}$ determines the dimensionality and meaning of the friction force $\mathbf{F}_{slip}$ – for example, $\mathbf{e}$ may be a scalar (a generalized speed), a translational 2-vector (planar friction), or a rotational 3-vector such as the angular velocity of a ball joint. These would produce friction forces that are: a generalized force, a planar force, or a 3d torque, respectively.

8.6.2.1 Coefficient of friction

The coefficient of friction μ in general may be velocity-dependent. Pure Coulomb friction switches discontinuously between two values, a static coefficient of friction μ_s used when sticking, and a dynamic coefficient $\mu_d \leq \mu_s$ used when sliding. Continuous schemes may blend these values based on velocity. Coefficients of friction may also change with time and position, but must not depend on accelerations or constraint forces. A typical form for sliding friction (where $\mathbf{v}$ is the slip velocity vector) is:

$$\mu = \mu_d + \mu_v |\mathbf{v}| \tag{8.65}$$

where μ_v is a viscous coefficient of friction. For any given pair of contact surfaces, these coefficients must be calculated via some combination of the two sets of surface properties.

8.6.2.2 Normal force

N may depend nonlinearly (quadratically) on mobilizer or constraint reactions, which are in turn dependent on the calculated accelerations $\dot{u}$ and Lagrange multipliers λ. Because these may then depend on friction forces, the equations of motion can become mildly nonlinear and some iteration may be required to find a compatible set of accelerations, multipliers, and sliding friction forces for a given set of active constraints. During impact, similar iteration may be required to find compatible velocity changes, impulse multipliers, and impulsive sliding friction forces. We expect this iteration usually to converge very rapidly for several reasons: the nonlinearity is very mild, at most the magnitude of a vector; friction forces are

80

orthogonal to *N*'s application direction so coupling must be indirect through the multibody system; and we generally have a very good starting guess from the normal forces at the previous time step. Nevertheless the coupling can be strong at times and there are circumstances in which convergence cannot be obtained without an impulsive change to velocities (see Painlevé's paradox).

8.6.2.3 Sliding friction with linear dependence on multipliers

When each sliding friction multiplier σ_j is a linear function of a constraint multiplier $\lambda_{j'}$, as is the case for point contact between surfaces, the equations of motion remain linear, although with a switching condition (a discrete nonlinearity). There are two cases, depending on whether the constraint multiplier belongs to a bilateral or a unilateral constraint, as illustrated below.
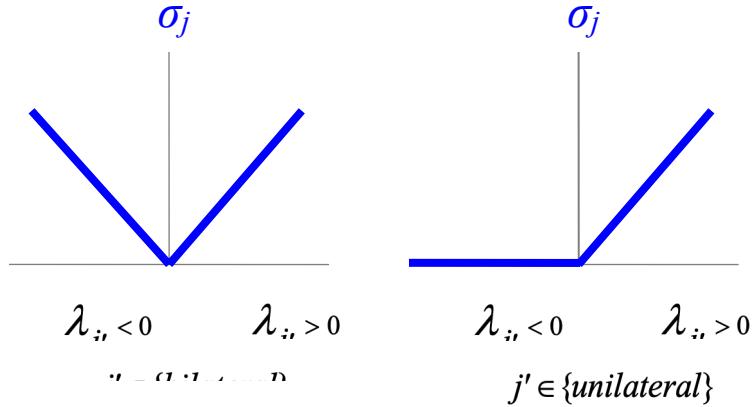
Figure 11: Dependence of sliding force magnitude σ on normal constraint multiplier λ.

Friction force multipliers are required to be nonnegative, but the constraint multipliers they depend on are signed. This is true even for unilateral constraints here, because we are assuming a given active set (all active constraints are treated as bilateral). If that results in violation of a unilateral constraint (i.e., a negative constraint multiplier or stiction beyond its limit), a higher-level algorithm will revise the set of active constraints; we do not make that choice here.

The friction force multipliers are given by

$$\sigma_j = \begin{cases} \mu_j |\lambda_{j'}|, & j' \in \{bilateral\} \\ \mu_j \max(\lambda_{j'},0), & j' \in \{unilateral\} \end{cases} \qquad (8.66)$$

where j' is the index of the constraint multiplier on which sliding friction force j depends. There are a total of k sliding forces and m multipliers (and n generalized speeds). For convenience we'll arrange the $j \rightarrow j'$ mapping in a kxm matrix $\mathbf{T}$, which is all zero except that element $\mathbf{T}[j,j']=1$ for $j=1..k$. Then $\mathbf{T}\lambda$ selects just the k multipliers for which there are sliding forces and reorders them if necessary. ($\mathbf{T}$ can select the same multiplier more than once if several sliding forces depend on it, but that is unlikely.)

We can view this as a collection of linear equations, only one of which is active for each j, depending on the sign of $\lambda_{j'}$:

$$\sigma_j = \begin{cases} \mu_j \lambda_{j'}, & \lambda_{j'} \geq 0 \\ -\mu_j \lambda_{j'}, & \lambda_{j'} < 0 \text{ and } j' \in \{bilateral\} \\ 0, & \lambda_{j'} < 0 \text{ and } j' \in \{unilateral\} \end{cases} \qquad (8.67)$$

We don't know in advance what the signs of the multipliers will be, so we now have k additional unknowns $d_j = \text{sense}(\lambda_{j'}) \in \{1,-1,0\}$ giving the sense ("direction") for each sliding force equation, defined

$$d_j = \text{sense}(\lambda_{j'}) = \begin{cases} \text{sign}(\lambda_{j'}), & j' \in \{bilateral\} \\ \max(\text{sign}(\lambda_{j'}),0), & j' \in \{unilateral\} \end{cases} \qquad (8.68)$$

where sign(x) is 1 if $x \geq 0$, -1 otherwise. Note that d does not represent sliding directions; it is just the sense of the relationship between the normal constraint multipliers and their sliding force multipliers. With that definition we have

$$\sigma_j = d_j \mu_j \lambda_{j'} \quad (\geq 0) \qquad (8.69)$$

Assemble the sense scalars into a kxk diagonal matrix $\mathbf{D}=\mathbf{diag}(d_j)$. Given the right values for d, we can write the sliding friction magnitudes as the set of linear equations

$$\sigma = \mathbf{D\mu T}\lambda \quad (\geq 0) \qquad (8.70)$$

where σ is a kx1 column vector containing the sliding friction force multipliers and $\mathbf{\mu}=\mathbf{diag}(\mu_j)$ is a kxk diagonal matrix. Now let $\mathbf{S}^T$ be the matrix mapping those magnitudes to

generalized forces (that is, $\mathbf{S}=\mathbf{S}(t,q,u)$ is a kxn matrix formed from the slip directions $\mathbf{v}$ and kinematic Jacobian). Then the equations of motion are

$$\begin{aligned}
\mathbf{M}\dot{u} + \mathbf{G}^T\lambda + \mathbf{S}^T\sigma &= f \\
\mathbf{G}\dot{u} &= b \\
\sigma &= \mathbf{D\mu T}\lambda \\
d &= \text{sense}(\mathbf{T}\lambda), \quad \mathbf{D} = \mathbf{diag}(d)
\end{aligned}$$

$$(8.71)$$

Given fixed values for d, the first three equations above define a linear subproblem for the unknowns $\dot{u}, \lambda, \sigma$. Substituting the third line of (8.71) into the first eliminates σ:

$$\begin{aligned}
\mathbf{M}\dot{u} + (\mathbf{G} + \mathbf{T}^T\mathbf{D\mu S})^T\lambda &= f \\
\mathbf{G}\dot{u} &= b
\end{aligned}$$

$$(8.72)$$

Then multiplying through by the inverse mass matrix and substituting eliminates $\dot{u}$:

$$\begin{aligned}
\mathbf{W}\lambda &= \mathbf{GM}^{-1}f - b \\
\text{where} \quad \mathbf{W} &= \mathbf{GM}^{-1}(\mathbf{G} + \mathbf{T}^T\mathbf{D\mu S})^T
\end{aligned}$$

$$(8.73)$$

This system is likely to be underdetermined, but the least-squares value of λ is the unique solution we want:

$$\lambda = \mathbf{W}^+(\mathbf{GM}^{-1}f - b) \qquad (8.74)$$

where superscript + indicates pseudoinverse. This is identical to Simbody's usual calculation of λ except for the new term $\mathbf{T}^T\mathbf{D\mu S}$, which makes $\mathbf{W}$ non-symmetric. That doesn't matter for us since we can't easily exploit symmetry anyway here due to the potential for redundancy. Now we can recalculate d and check whether we used the right set of equations. If so, then $\sigma = \mathbf{D\mu T}\lambda$ are the correct, nonnegative sliding force multipliers. If not we'll have to revise d and re-solve the modified linear system.

In the case where we have chosen d incorrectly, we will have errors in σ due to having chosen the wrong linear equation from (8.67). For bilateral constraints we will have had $d=1$ when it should have been -1 or vice versa; for unilateral constraints we will have had $d=0$ when it should have been 1 or vice versa. That means an incorrect choice leads to twice the error for bilateral constraints as for unilateral ones. You can see that easily in Figure 12 below.
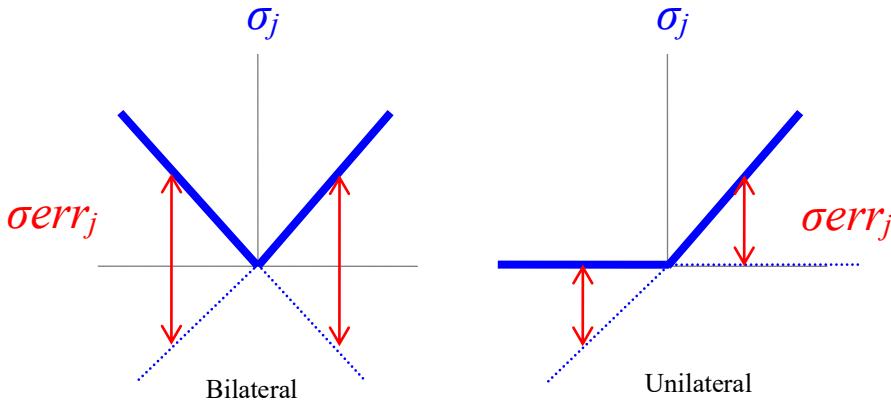
Figure 12: The error in a sliding friction force multiplier when the wrong linear equation is chosen for it. The dotted blue lines show the incorrect values of σ. Note that the error is twice as large for bilateral constraints as for unilateral ones.

We propose the following algorithm to solve this system [TODO: is there a better way?]:

1. Assume we are given an initial set of signs d_j (these will typically have been saved from the previous time step).

2. Solve for λ using (8.74), then define *derr* such that $derr_j = \left| d_j - \text{sense}(\lambda_{j'}) \right| \in \{0,1,2\}$. This will be 2 for an incorrect bilateral constraint, 1 for an incorrect unilateral constraint, and zero if d_j was correct.

3. If *derr*=0 we are done since we used the correct set of linear equations. Go to step 5.

4. Otherwise, from all j for which $derr_j \neq 0$, choose the one for which $\sigma err_j = derr_j \mu_j \left| \lambda_{j'} \right|$ is largest, then revise d_j using (8.68), and repeat from step 2. [TODO: is this the right way to choose j?] Note that we only revise one sign at a time since all the unknowns are coupled in general—changing more than one at a time can cause failure to converge. [TODO: what can we say about convergence and uniqueness?]

5. Verify that we have a solution. If not, the system is inconsistent which means there must be an impulsive velocity change to change a sliding contact into a sticking one; the impact handler should be invoked.

This same set of equations can be used for impact also, after relabeling, removing non-impulsive forces, and inserting the velocity error to be removed from normal and stiction constraints:

$$\mathbf{M}\Delta u + \mathbf{G}^T \pi + \mathbf{S}^T \sigma = 0$$
$$\mathbf{G}\Delta u = -v_{err}$$
$$\sigma = \mathbf{D}\boldsymbol{\mu}\mathbf{T}\pi \tag{8.75}$$
$$d = \text{sense}(\mathbf{T}\pi), \quad \mathbf{D} = \mathbf{diag}(d)$$

where the unknowns are the set of impulse multipliers π (and σ), the velocity change Δu, and the signs d.

8.6.2.4 *Sliding friction with nonlinear dependence on multipliers*

A bilateral constraint implemented with several multipliers may have sliding friction that is dependent on a normal force that is the magnitude of a vector. Examples are a 2-multiplier point-on-line constraint (with sliding friction along the line), or joint-like constraints such as a 3-multiplier ball joint constraint (sliding friction torque vector opposing angular velocity) or 5-multiplier pin joint constraint (sliding friction generalized torque about rotation axis). In these cases the normal force scalar of interest is the norm of a vector which is some projection of the constraint reaction force. Then we can write the magnitude $\sigma_j \geq 0$ of the jth sliding friction force as

$$\sigma_j = \mu_j N_j(\vec{\lambda}_{j'})$$
$$\text{where} \quad N_j(\vec{\lambda}_{j'}) = \left\| \mathbf{A}_j \vec{\lambda}_{j'} \right\|_2 \tag{8.76}$$

Subscript j' is now the set of multiplier indices on which σ_j depends and $\vec{\lambda}_{j'}$ is a vector, typically with 2 or 3 elements; conceivably it could contain up to 5. $\mathbf{A}_j = \mathbf{A}_j(q)$ provides rotation, scaling, and projection as needed. Note that $N_j(0)=0$; you can't have a sliding friction force without a normal force. However, $\mathbf{A}_j$ is not necessarily full rank so $\mathbf{A}_j \vec{\lambda}_{j'}$ may be zero even if $\vec{\lambda}_{j'} \neq 0$.

Now the equations of motion become:

$$\mathbf{M}\dot{u} + \mathbf{G}^T \lambda + \mathbf{S}^T \sigma = f$$
$$\mathbf{G}\dot{u} = b \tag{8.77}$$
$$\sigma = \boldsymbol{\mu}N(\lambda)$$

where each element of vector function N is one of the N_j and $\boldsymbol{\mu} = \mathbf{diag}(\mu_j)$. Given an estimate λ^i for λ, we can approximate N locally to first order as

$$N(\lambda^i + \Delta\lambda) = N(\lambda^i) + N_\lambda^i \Delta\lambda + O(\Delta\lambda^2)$$

$$\text{where} \quad N_\lambda^i \triangleq \left.\frac{\partial N}{\partial \lambda}\right|_{\lambda=\lambda^i} \quad (k \times m) \tag{8.78}$$

From (8.76) we can determine the general form for a block of the derivative:

$$\left.\frac{\partial N_j}{\partial \vec{\lambda}_{j'}}\right|_{\vec{\lambda}_j=\vec{\lambda}^i} = \frac{\mathbf{A}_j^T \mathbf{A}_j \vec{\lambda}^i}{\left\|\mathbf{A}_j \vec{\lambda}^i\right\|_2} \tag{8.79}$$

Note that this is undefined when $\mathbf{A}_j \vec{\lambda}^i = 0$, requiring special handling there.

Define $\sigma^i = \boldsymbol{\mu} N(\lambda^i)$. Assume we have an initial estimate λ^0, perhaps saved from the previous time step, and let $\sigma^0 = \boldsymbol{\mu} N(\lambda^0)$ be our initial estimate of the sliding friction magnitudes. We can refine this estimate using Newton iteration, terminating when

$$\left\|\sigma^{i+1} - \sigma^i\right\|_\infty \leq tol \tag{8.80}$$

for some *tol* based on user-requested accuracy. Substituting into (8.77) gives

$$\begin{aligned} \mathbf{M}\ddot{u}^{i+1} + \mathbf{G}^T \lambda^{i+1} + \mathbf{S}^T \sigma^{i+1} &= f \\ \mathbf{G}\ddot{u}^{i+1} &= b \\ \sigma^{i+1} &\approx \sigma^i + \boldsymbol{\mu} N_\lambda^i \Delta\lambda \\ \text{where} \quad \lambda^{i+1} &= \lambda^i + \Delta\lambda \end{aligned} \tag{8.81}$$

The unknowns are $\ddot{u}$ and $\Delta\lambda$. Rearranging, we get

$$\begin{aligned} \mathbf{M}\ddot{u}^{i+1} + (\mathbf{G} + N_\lambda^{i\,T} \boldsymbol{\mu} \mathbf{S})^T \Delta\lambda &= f^i \\ \mathbf{G}\ddot{u}^{i+1} &= b \\ \text{where} \quad f^i &= f - \mathbf{G}^T \lambda^i - \mathbf{S}^T \sigma^i \end{aligned} \tag{8.82}$$

We can solve now as for the linear case:

$$\begin{aligned} \Delta\lambda &= \mathbf{W}^{i+}(\mathbf{G}\mathbf{M}^{-1} f^i - b) \\ \text{where now} \quad \mathbf{W}^i &= \mathbf{G}\mathbf{M}^{-1}(\mathbf{G} + N_\lambda^{i\,T} \boldsymbol{\mu} \mathbf{S})^T \end{aligned} \tag{8.83}$$

That gives us λ^{i+1} so we can calculate $\sigma^{i+1} = \boldsymbol{\mu} N(\lambda^{i+1})$ and check for convergence in (8.80). If we converged, then the solution is $[\ddot{u}, \lambda, \sigma] = [\ddot{u}^{i+1}, \lambda^{i+1}, \sigma^{i+1}]$.

Note that the above describes a full Newton iteration, where the Jacobian is recalculated with each iteration. Because of the mild nonlinearity, we are likely to be able to converge quickly with a modified Newton, using only N_λ^0 and $\mathbf{W}^0$ unless one of the λ changes sign. However, as mentioned above, Jacobian N_λ will be undefined at $\mathbf{A}\lambda=0$.

8.6.2.5 Stiction to sliding transition

When a contact was sticking but first begins to slide, there is no slip velocity to use to determine the sliding force direction. Instead we want the sliding force to oppose the (unknown) *impending* slip direction. One reasonable approach is to record the last known stiction force direction and use that as the initial sliding force direction. Provided that the last stiction force and first sliding force are about the same, this is likely to be a good direction. However, it is possible that the stiction force exceeded its limit by a lot, and the coefficient of static friction may be much larger than the coefficient of dynamic friction, so there may be a large change in magnitude at the stiction→sliding transition. This may cause a change in direction that could make the final stiction force direction differ substantially from the direction of impending slip.

A single friction element can be viewed as having three distinct "phase":

1. Stiction. The sliding velocity is zero; the magnitude and direction of the stiction force that will keep the sliding acceleration at zero are both unknown. This is implemented as a set of workless constraint equations. If the magnitude exceeds $\mu_s N$, we transition to impending slip.

2. Impending slip. The velocity is currently zero but the acceleration is not constrained. The magnitude is known as a function of the (unknown) normal force ($\mu_d N$), and the direction should oppose the (unknown) acceleration.

3. Sliding. The magnitude is known as a function of the (unknown) normal force, and the direction is known as a function of state (it opposes the sliding velocity).

Here are the equations of motion with an impending slip friction element:

$$
\begin{aligned}
\mathbf{M}\dot{u} + \mathbf{G}^T \lambda + \mathbf{S}^T \sigma &= f \\
\mathbf{G}\dot{u} &= b \\
\sigma_i &= \mu_d N \mathbf{e}_i \\
\mathbf{e} &= \mathbf{a}/|\mathbf{a}|
\end{aligned}
\tag{8.84}
$$

where $\mathbf{a}$ is the friction-space (usually tangential) acceleration direction which is linearly dependent on the unknowns $\dot{u}$. We can get a good initial guess for the direction $\mathbf{e}$ using the last stiction force direction: $\mathbf{e} = -\boldsymbol{\lambda}/|\boldsymbol{\lambda}|$

8.6.3 Event detection for unilateral constraints

Once the set of active constraints has been determined, a Simbody simulation proceeds continuously with an unchanged active set until a contact event is detected. Then one of two event handlers is invoked: either the Impact Handler, or the Contact Handler.

The Impact Handler is invoked when a velocity-level unilateral constraint condition would be violated, requiring a discontinuous change in velocities. There are two ways this can happen. Most commonly, a holonomic unilateral constraint like contact or a joint stop reaches its limit with a negative velocity, meaning further simulation would cause violation of the limit. Alternatively, failure to converge a sliding friction contact force requires an impulsive transition from sliding to sticking.

The Contact Handler is invoked when an acceleration-level, or equivalently reaction force-level, unilateral constraint condition would be violated. For example, change of a unilateral contact's normal force from positive to negative requires invocation of the Contact Handler. The solution involves only changes to the active constraint set; no changes to velocity occur except very small projections of the active velocity constraints to satisfy velocity tolerances. The Contact Handler is also invoked as the last step of the Impact Handler, to determine the active set after the instantaneous velocity change has been completed. Figure 13 outlines this flow of control during a time stepping study.
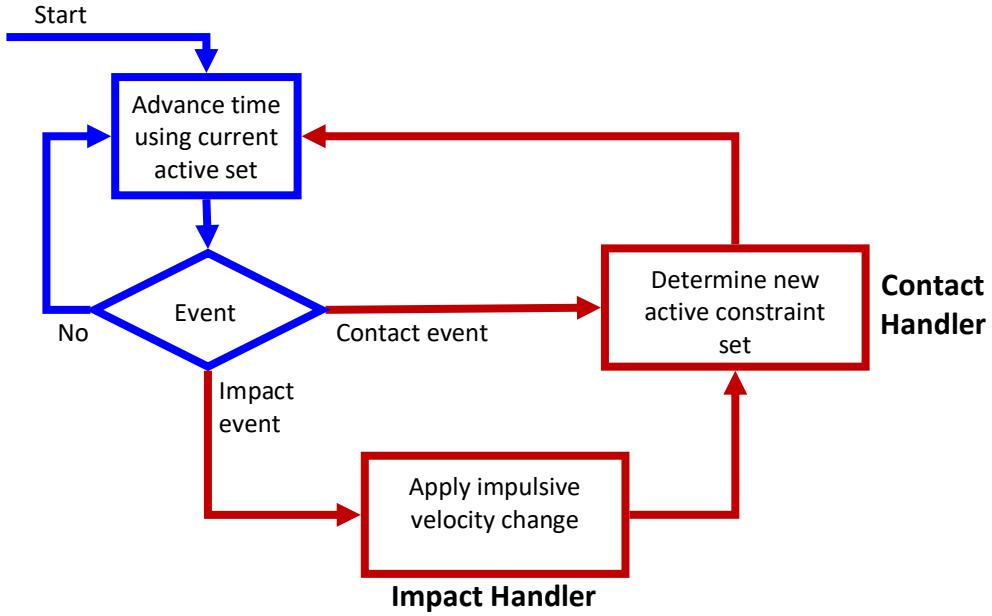
Figure 13: Flow of control during time stepping with unilateral contact and friction constraints. Time stepping begins with a feasible active set. Then the normal time stepping sequence is in blue and does not involve recalculation of the active set.

Although one or more particular constraint condition violations will trigger an event, the handlers work with all relevant constraints simultaneously without giving the triggering event special consideration.

TODO: Impact events: penetration, failure to converge sliding friction normal forces.

TODO: Contact events: liftoff (normal force goes negative), contact (normal acceleration goes negative), stiction release (force exceeds $\mu_s N$), slip-to-stick (slip velocity stops or reverses).

8.7 Dynamic 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** in equation (8.53) 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* $\ddot{\mathbf{p}}$.

The *nonholonomic constraint equations* $\mathbf{v}$ in equation (8.52) 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* $\dot{\mathbf{v}}$.

The *acceleration constraint equations* $\mathbf{a}$ in equation (8.51) 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 the *quaternion normalization constraints* $\mathbf{n}$ in equation(8.49), 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 mobilities u.

Unlike the holonomic and nonholonomic constraints, there are no constraints at the velocity or acceleration level corresponding to the quaternion normalization constraint. Equation (8.48) constructs the quaternion derivatives in terms of the three independent u's, ensuring by construction that the velocity-level constraints are satisfied.

Note that the system equations include a block diagonal invertible linear mapping between the u's and the time derivatives of the q's, shown in equation (8.48): $\dot{q} = \mathbf{N}(q)u$. Although $\mathbf{N}$ is a rectangular matrix, it is invertible. Note that when the quaternion normalization constraints are not satisfied exactly, the 4x3 blocks $\mathbf{N}_i$ on the diagonal of $\mathbf{N}$ 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.

Here is the system in the form we actually solve in Simbody, with slow (discrete) variables not shown:

$$
\begin{array}{c}
\dot{q} = \mathbf{N}u \\
\mathbf{n}(q) = 0
\end{array}
\tag{8.85}
$$

kinematics

dynamics
$$
\begin{pmatrix} \mathbf{M} & \mathbf{G}^{\mathsf{T}} \\ \mathbf{G} & 0 \end{pmatrix} \begin{pmatrix} \dot{u} \\ \lambda \end{pmatrix} = \begin{pmatrix} \mathbf{f} - \mathbf{f}_{\text{bias}} \\ \mathbf{b} \end{pmatrix}
\tag{8.86}
$$
$$
\dot{z} = \dot{z}(t, q, u, z)
$$

velocity manifold
$$
\begin{array}{c}
\mathbf{P}u - \mathbf{c} = 0 \\
\mathbf{v}(t, q; u) = 0
\end{array}
\tag{8.87}
$$

position manifold
$$
\mathbf{p}(t; q) = 0
\tag{8.88}
$$

relax fast variables
$$
\mathbf{r}(t, y_{dyn}; d_f, y_f) = 0
\tag{8.89}
$$

We are given initial condition $S(t_0) = \{t_0, d(t_0), y(t_0)\}$ such that equations (8.87)–(8.89) are satisfied, and are asked in a time stepping study to solve for $d(t)$, $y(t)$ for $t_0 \le t \le t_{\text{final}}$.

Showing prescribed motion and fast variables:

kinematics
$$
\begin{array}{c}
\dot{q} = \mathbf{N}u \\
\mathbf{n}(q) = 0
\end{array}
\tag{8.90}
$$

dynamics
$$
\begin{pmatrix} \mathbf{M} & \mathbf{G}^{\mathsf{T}} \\ \mathbf{G} & 0 \end{pmatrix} \begin{pmatrix} \dot{u} \\ \lambda \end{pmatrix} = \begin{pmatrix} \mathbf{f} - \mathbf{f}_{\text{bias}} \\ \mathbf{b} \end{pmatrix}
\tag{8.91}
$$

auxiliary
$$
\dot{z} = \dot{z}(t, q, u, z, \dot{u}, \lambda, \tau)
\tag{8.92}
$$

$$\text{constraint matrix} \quad \left\{ \begin{aligned} & \mathbf{g} \triangleq \mathbf{G}\begin{pmatrix} \dot{u}_p \\ \dot{u}_f \end{pmatrix} - \mathbf{b} = 0 \\[2em] & \mathbf{G} = \begin{pmatrix} 1_p & 0_f \\ \mathbf{G}_{fp} & \mathbf{G}_{ff} \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} 1_f & 0_f \end{pmatrix} \\ \begin{pmatrix} \mathbf{P}_{fp} \\ \mathbf{V}_{fp} \\ \mathbf{A}_{fp} \end{pmatrix} \begin{pmatrix} \mathbf{P}_{ff} \\ \mathbf{V}_{ff} \\ \mathbf{A}_{ff} \end{pmatrix} \end{pmatrix} \end{aligned} \right. \tag{8.93}$$

$$\text{position manifold} \quad \left\{ \quad \mathbf{p} = \begin{pmatrix} q_p - \mathbf{P}_p(t) \\ \mathbf{p}_f(t, q_p; q_f) \end{pmatrix} = 0 \right. \tag{8.94}$$

$$\text{velocity manifold} \quad \left\{ \begin{aligned} & \dot{\mathbf{p}} = \begin{pmatrix} u_{q_p} - \mathbf{c}_p \\ \mathbf{P}_f u - \mathbf{c}_r \end{pmatrix} = 0, \qquad u_{q_p} = \mathbf{N}^{-1}\dot{q}_p \tag{8.95} \\[1.5em] & \mathbf{v} = \begin{pmatrix} u_p - \mathbf{v}_p(t, q) \\ \mathbf{v}_f(t, q, u_p; u_f) \end{pmatrix} = 0 \tag{8.96} \end{aligned} \right.$$

$$\text{acceleration manifold} \left\{ \begin{aligned} & \ddot{\mathbf{p}} = \begin{pmatrix} \dot{u}_{q_p} - \mathbf{b}_{q_p} \\ \mathbf{P}_f \dot{u} - \mathbf{b}_{q_f} \end{pmatrix} = 0 \tag{8.97} \\[1.5em] & \dot{\mathbf{v}} = \begin{pmatrix} \dot{u}_{u_p} - \mathbf{b}_{u_p} \\ \mathbf{V}_f \dot{u} - \mathbf{b}_{u_f} \end{pmatrix} = 0 \tag{8.98} \\[1.5em] & \mathbf{a} = \begin{pmatrix} \dot{u}_p - \mathbf{a}_p(t, q, u) \\ \mathbf{a}_f(t, q, u, \dot{u}_p; \dot{u}_f) \triangleq \mathbf{A}_f \dot{u} - \mathbf{b}_f \end{pmatrix} = 0 \tag{8.99} \end{aligned} \right.$$

$$\text{relax fast variables} \left\{ \begin{aligned} & \mathbf{r}^{\text{time}} & (t, d, y_{dyn}; & \quad d_{fast}^{\text{time}}) & = 0 \\ & \mathbf{r}^{\text{pos}} & (t, d, y_{dyn}; & \quad d_{fast}^{\text{pos}}, q_{fast}) & = 0 \\ & \mathbf{r}^{\text{vel}} & (t, d, y_{dyn}, q_{fast}; & \quad d_{fast}^{\text{vel}}, u_{fast}) & = 0 \\ & \mathbf{r}^{\text{force}} & (t, d, y_{dyn}, q_{fast}, u_{fast}; & \quad d_{fast}^{\text{force}}, z) & = 0 \\ & \mathbf{r}^{\text{acc}} & (t, d, y; & \quad d_{fast}^{\text{acc}}, \dot{u}_{fast}, \dot{z}) & = 0 \end{aligned} \right. \tag{8.100}$$

Relaxation constraints (8.100) should be solved in stage order as shown, so that once fast variables have been calculated at one stage they can be dependencies in a later stage. Note that a relaxation solver at *any* stage may involve repeated state realizations through Acceleration stage; the relaxation stage just determines which of the fast variables may be modified.

92

For compactness, equation (8.100) doesn't show the always-present dependency on y_{slow}, and doesn't show which variables in the state's d partition can actually serve as dependencies; the rule is: any d variable is allowed except those in d_{fast} which haven't yet been computed in the relaxation sequence. As with the dynamic constraints in (8.94)-(8.99), relaxation constraints will in general consist of a mix of explicit and implicit equations, but we're not breaking them out since there is no special treatment for them outside the relaxation solver itself.

9 Scaling and 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; definitions of generalized coordinates and speeds; 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 significant tradeoffs between CPU time and accuracy, so we usually want to use the loosest accuracy that is adequate for our purposes. 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) scaling of 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 Relative vs. absolute accuracy

For some quantities we are satisfied to achieve *relative* accuracy, e.g. a result that is within 1% of the correct value. In that case absolute errors of larger magnitude are acceptable when the numerical magnitude of a quantity is larger. However, one must be careful to avoid

situations in which a physically-*meaningless* change in magnitude of a measured quantity would result in a physically-*significant* difference in accuracy or behavior. For example, rotations are cyclical quantities, but the angular variables that define them can grow without bound, adding 2π with each revolution. There is no configuration change associated with those factors of 2π and no reason that the absolute angle should be determined less accurately as the revolution count increases. Less obviously, translational coordinates can also introduce physically-insignificant numerical changes. Consider a block on a sliding joint that hits a stiff spring stop at $x=1$ and a simulation that predicts its behavior as it encounters the stop. If you now lengthen the block's travel and put the stop at $x=1000$, it would be incorrect to calculate x any less accurately because that could drastically change the calculation of the spring force, with significant consequences for the block's predicted behavior. Common sense says a block should behave the same way in one place as another. Similarly, any change in the spatial location of a system involves changes to translational variables, yet should not affect the precision of results. Hence Simbody solves all configuration coordinates q to an absolute accuracy level independent of their current values. Note that this does mean that there can be a performance cost for choosing units far from unity; we deem that a more reasonable consequence than producing bad answers.

Unlike configuration variables, velocity variables u may reasonably be solved to relative accuracies in most cases[*]. Arbitrary constants in q are eliminated by differentiation, so do not appear in $\dot{q}$ or u. For auxiliary state variables z relative accuracy may or may not be appropriate; Simbody assumes that relative accuracy for z is sufficient unless told otherwise.

When a u or z is near zero, relative accuracy becomes too strict. In that case an absolute accuracy requirement should be used instead. The absolute accuracy requirement for a variable represents a "good enough" level that is acceptable no matter what fractional error of the current value that represents. For example, say accuracy $\alpha=0.1\%$ (10^{-3}) has been requested for a forward dynamics study. If the current value of a variable x is 10^{-6} units, an error

---

[*] This may not be acceptable for u's that are involved in constraints, since constraint errors generally involve the *differences* between velocities.

estimate no larger than 10^{-9} units would be required to produce a relative accuracy of 0.1%, requiring a very small step size. However, if the absolute error requirement for x were 10^{-5} units, then an error estimate as large as 10^{-5} units would be acceptable, despite that being 1000% of the current value.

9.2 *Weighting and absolute accuracy*

An important practical consideration for any multibody formulation is that the state variables $y = \{q, u, z\}$ vary widely in effect, or *weight*, by which we mean the degree to which a unit change in the numerical value of a state variable affects a physically meaningful quantity of interest. There are several causes for the uneven weights 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. Weighting 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 unit change δq_i 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.
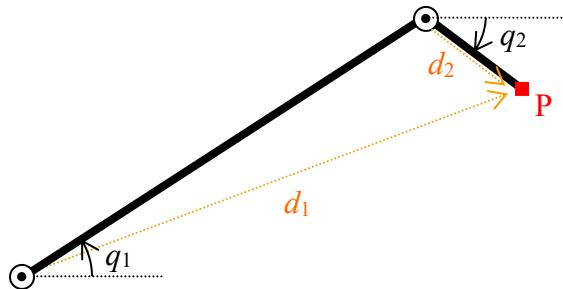
Figure 14: Weighting of q_1 and q_2 are very different with respect to the position of the end point P.

Figure 14 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. If we consider a unit change of P to be 1nm, we can define a corresponding "unit change" of q_1 in this case to be δq_1=0.1 radians while δq_2=2 radians. Each of those unit changes will move P by 1nm. States which are "more important" will have a numerically smaller unit change. Note that "unit change" does not mean "small change." Rather, this is the quantity to which the requested accuracy α is applied.

Even in the simple example of Figure 14 it is clear that the weights 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 weighting does not need to be done perfectly to yield substantial improvements over un-scaled variables. That permits us to treat weights as constants in the discussion to follow; in practice they are updated only occasionally. Also, in practice there is no single quantity P that we can use to neatly define unit changes for each of the generalized coordinates.

This concept of "unit change" $\delta y_i > 0$ for each state variable y_i is used to define the "always good enough" absolute error requirement $\alpha \cdot \delta y_i$ for use when relative error $\alpha \cdot y_i$ is not allowed or too stringent. We can then define a diagonal weighting matrix $\mathbf{W}$, where the ith diagonal element is $w_i = 1/\delta y_i$, the "unit weight" of state variable y_i. For example, referring again to Figure 14, if we want to scale by the geometric consequences of the state variables on P's location we could use $w_1 = 1/\delta q_1 = d_1$ and $w_2 = 1/\delta q_2 = d_2$. Then we would define $\mathbf{W}$ as follows:

$$\mathbf{W} \triangleq \begin{pmatrix} d_1 & 0 \\ 0 & d_2 \end{pmatrix}$$

In the above discussion we have oversimplified the choice of weights for q by choosing a system in which $\dot{q} = u$. In general, however, we have $\dot{q} = \mathbf{N}(q)u$ (and $u = \mathbf{N}^+(q)\dot{q}$). That means the effective weights on q contain arbitrary factors due to choice of coordinates in addition to physical considerations. However, given weights on u (which are physical) we can easily determine the corresponding $\dot{q}$ weights. So instead of determining weights from $\partial P/\partial q$ we instead look at $\partial \dot{P}/\partial u$. Thus "unit change" estimates are provided for 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 nxn diagonal weighting matrix $\mathbf{W}_u$; there is no separate $\mathbf{W}_q$. Instead, it may be calculated as

$$W_q = NW_u N^+ \tag{9.1}$$

which ensures that the appropriate kinematic relationship holds between the weighted velocity variables, that is $\dot{q}_w = N(q)u_w$ where $\dot{q}_w = W_q\dot{q}$ and $u_w = W_u u$. Note that although W_u is diagonal, positive, and constant, W_q is block-diagonal, signed, and q-dependent since $N=N(q)$. (Note that although $N^+N = I$, $NN^+ \neq I$.) If there are auxiliary state variables z they will have independent weights W_z. Then we have

$$W = \begin{pmatrix} NW_u N^+ & 0 & 0 \\ 0 & W_u & 0 \\ 0 & 0 & W_z \end{pmatrix} \tag{9.2}$$

Even when $W_u=I$, W_q can be significant for orientation coordinates. Some configurations of quaternions have one or more coordinates with zero weight (because they act only to change the length of the quaternion and thus produce no rotation). Euler angle coordinates near singular configurations have one coordinate whose weight approaches infinity; proper weighting ensures the simulation remains accurate in near-singular configurations, although possibly with a significant performance penalty.

We can combine relative accuracy requirements with the absolute "unit change" estimates into a matrix E to form the *error norm*:

$$E_q = NW_u N^+$$

$$E = \begin{pmatrix} E_q & & \\ & E_u & \\ & & E_z \end{pmatrix} \quad \text{where} \quad E_u = \begin{cases} \min(W_{u,i}, 1/|u_i|), & u_i \text{ solved to relative accuracy} \\ W_{u,i}, & \text{otherwise} \end{cases} \tag{9.3}$$

$$E_z = \begin{cases} \min(W_{z,i}, 1/|z_i|), & z_i \text{ solved to relative accuracy} \\ W_{z,i}, & \text{otherwise} \end{cases}$$

Now let e_y represent the vector of n_y unweighted error estimates introduced by a step in the numerical solution for $y(t)$. Then $f_y = Ee_y$ is the vector giving for each state variable the unitless fractional error represented by the value in e_y. We can define the error 2-norm $\|e_y\|_E \triangleq \|f_y\|_2$ and error ∞-norm $\|e_y\|_{E\infty} \triangleq \|f_y\|_\infty$. We don't use the 2-norm directly, but instead use the related RMS norm

$$\|e_y\|_{ERMS} \triangleq \frac{1}{\sqrt{n_y}} \|e_y\|_E$$

where n_y is the number of state variables, because the RMS norm does not grow with the problem size. (The ∞-norm is already size-independent.) Now we can provide a precise meaning for the notion "solve for $y(t)$ to an accuracy α": either $\left\| e_y \right\|_{\text{ERMS}} \leq \alpha$ or (stricter) $\left\| e_y \right\|_{\text{E}\infty} \leq \alpha$ at each step during the study. Typically we'll require that q, u, and z norms separately meet the required accuracy to avoid mixing dissimilar variables in the RMS case.

9.3 Scaling of constraint errors

The nonlinear algebraic equations defining the system, such as equations (8.52) and (8.53), 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 weighting 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 (8.86). However, if you are using your own methods you might find it useful to scale the acceleration errors. 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 unit constraint errors $t_{\text{p}}, t_{\text{v}}, t_{\text{a}} > 0$, one for each of the m_{p} position (holonomic) and m_{v} velocity (nonholonomic) and m_{a} acceleration-only constraint equations. Each of these has appropriate units for its constraint equation, e.g. angle or length for a holonomic constraint, speed for a nonholonomic constraint, and accelerations for acceleration-only constraints. Each t_i should represent the violation that is to be considered a "unit

violation" of the ith constraint ("unit" doesn't necessarily mean "small"). By default each unit constraint error is 1, meaning 1 length unit, 1 radian, 1 radian/time unit, etc.

To estimate tolerances for holonomic first and second derivatives, and nonholonomic derivatives, we use a System-defined characteristic time scale parameter t_c. This parameter gives a typical "time of interest" over which we expect to see significant behavior. This should be roughly the rate at which you could expect interesting reporting data, *not* necessarily the highest frequency in the system. For example, given the time scale, we consider an appropriate unit velocity error for a distance constraint derivative to be one unit distance error per t_c. A biomechanical or robotic system might have a timescale of 100ms, a biomolecular system 100fs, and an orbital problem might have a timescale of 10 or 100s. Somewhat counterintuitively, a long time scale puts more stringent requirements on velocity constraints since they have much longer to drift in that case. Our default time scale is 0.1 time units.

Now we can define diagonal constraint weighting matrices $\mathbf{T}_p$, $\mathbf{T}_{pv}$, $\mathbf{T}_{pva}$ whose diagonal elements are the weights for the combined constraints at position, velocity, and acceleration levels (that is, velocity level includes holonomic derivatives, etc.). Note that "unit error" has the inverse sense to "weight"—while a larger weight means "more important" a larger unit error means "less important," so we invert unit errors in $\mathbf{T}$ to create weights. Here is how the constraint weighting matrices are defined:

$$
\begin{aligned}
&\mathbf{T}_p = \left(\operatorname{diag}(1./t_p) \right), \quad \mathbf{T}_v = \left(\operatorname{diag}(1./t_v) \right), \quad \mathbf{T}_a = \left(\operatorname{diag}(1./t_a) \right) \\
&\mathbf{T}_{pv} = \begin{pmatrix} t_c \mathbf{T}_p & \\ & \mathbf{T}_v \end{pmatrix} \\
&\mathbf{T}_{pva} = \begin{pmatrix} t_c \mathbf{T}_{pv} & \\ & \mathbf{T}_a \end{pmatrix} = \begin{pmatrix} t_c^2 \mathbf{T}_p & & \\ & t_c \mathbf{T}_v & \\ & & \mathbf{T}_a \end{pmatrix}
\end{aligned}
\qquad (9.4)
$$

Unlike relative weights of state variables, which can change as the state variable values change, we expect $\mathbf{T}$ to remain fixed once specified since tolerances are absolute quantities. Now define $\varepsilon_p, \varepsilon_{pv}, \varepsilon_{pva}$ as the vectors containing the current, unweighted errors for each constraint equation at each level. We can now calculate the tolerance norms $\left\| \varepsilon_p \right\|_{\text{T}} = \left\| \mathbf{T}_p \varepsilon_p \right\|_2$ and $\left\| \varepsilon_p \right\|_{\text{T}\infty} = \left\| \mathbf{T}_p \varepsilon_p \right\|_\infty$ and similarly for *pv* and *pva*. These treat all constraint errors uniformly.

As in the previous section, in practice we will use the RMS norm $\left\|\varepsilon_p\right\|_{\text{TRMS}} = \frac{1}{\sqrt{m_p}}\left\|\varepsilon_p\right\|_{\text{T}}$ or

∞-norm $\left\|\varepsilon_p\right\|_{\text{T}\infty}$ rather than the 2-norm to remove effects due just to problem size, and define the phrase "meeting tolerance to accuracy α" to mean

$$\max(\left\|\mathbf{T}_p\varepsilon_p\right\|_{norm}, \left\|\mathbf{T}_{pv}\varepsilon_{pv}\right\|_{norm}) \le \alpha \tag{9.5}$$

where *norm*=RMS or (stricter) *norm*=∞.

9.4 Scaling at the acceleration level

Acceleration-level constraint weightings $\mathbf{T}_{pva}$ do not matter when using Simbody's standard formulation since accelerations are calculated to machine precision. Even when the constraints are redundant accelerations are uniquely determined, however in that case the constraint forces are not unique. Simbody will calculate a least-squares solution to the constraint forces which tends to spread the load evenly among the redundant constraints. You can in theory provide a weighting $\mathbf{W}_\lambda$ to influence the norm in which that least squares calculation is performed. However, that is not the same as the acceleration error weighting $\mathbf{T}_{pva}$. Instead, $\mathbf{W}_\lambda$ is used to capture the relative stiffness of the compliant elements that are being modeled rigidly with constraints. This is a rather obscure nuance and is unlikely to be useful in practice – if you really care about how the forces are distributed you will probably want to use compliant elements. However, here is a proof that acceleration-level weighting has no effect except when there are redundant constraints.

Scaling on the constraint forces would be incorporated as follows. First, we want to solve for the scaled multipliers (using just $\mathbf{W}$ here to represent $\mathbf{W}_\lambda$):

$$M\dot{u} + G^{\text{T}}(\mathbf{W}\lambda_W) = f \quad \Rightarrow \quad M\dot{u} + (\mathbf{W}G)^{\text{T}}\lambda_W = f \tag{9.6}$$

($\mathbf{W}$ is symmetric.) This shows how the scaling transfers to the constraint Jacobian, so the acceleration constraint equation is scaled like this:

$$(\mathbf{W}G)\dot{u} = \mathbf{W}b \tag{9.7}$$

Note that once we have λ_W we can unscale this for use with the original constraints via

$$\lambda = \mathbf{W}^{-1}\lambda_W. \tag{9.8}$$

Multiplying (9.6) through by M^{-1} and then $\mathbf{W}G$ and substituting from (9.7) gives:

102

$$(\mathbf{W}GM^{-1}G^{\mathsf{T}}\mathbf{W})\lambda_W = \mathbf{W}(GM^{-1}f - b) \tag{9.9}$$

In the underdetermined case we can solve for a least squares λ_W using the pseudoinverse:

$$\lambda_W = (\mathbf{W}GM^{-1}G^{\mathsf{T}}\mathbf{W})^+ \, \mathbf{W}(GM^{-1}f - b) \tag{9.10}$$

But if $GM^{-1}G^{\mathsf{T}}$ has full rank then so does the scaled version, and the pseudoinverse is just the ordinary matrix inverse so we can rewrite (9.10) as

$$
\begin{aligned}
\lambda_W &= (\mathbf{W}GM^{-1}G^{\mathsf{T}}\mathbf{W})^{-1} \, \mathbf{W}(GM^{-1}f - b) \\
&= \mathbf{W}^{-1}(GM^{-1}G^{\mathsf{T}})^{-1}\mathbf{W}^{-1} \, \mathbf{W}(GM^{-1}f - b) \\
&= \mathbf{W}^{-1}(GM^{-1}G^{\mathsf{T}})^{-1}(GM^{-1}f - b) \\
&= \mathbf{W}^{-1}\lambda_U
\end{aligned}
\tag{9.11}
$$

where λ_U is just the multiplier we would have calculated without scaling. Unscaling λ_W using equation (9.8) gives $\lambda = \mathbf{W}\lambda_W = \mathbf{W}\mathbf{W}^{-1}\lambda_U = \lambda_U$, that is, the final λ is just the unscaled one. That proves that weighting the acceleration constraints and multipliers has no effect for systems with full rank.

9.5 Accuracy

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 $\mathbf{W}$ on changes to the state variables u and z (and implying a weighting on changes to q) and a constant reciprocal tolerance matrix $\mathbf{T}$ on the absolute errors in the constraint equations, appropriately time-scaled. $\mathbf{W}$ defines a "unit change" for each state variable, and $\mathbf{T}$ defines a "unit error" for each constraint equation. We also have information that defines what we consider "full scale" for each state variable, so that we can interpret accuracy as a request for precision as a fraction of the full scale value. The block diagonal matrix $\mathbf{E}$ combines relative scaling with the absolute unit changes in $\mathbf{W}$ to define the error norm.

Then we have solved a trajectory to an accuracy α =0.1% (for example) when both

$$
\begin{aligned}
&\max(\lVert\mathbf{E}_q e_q\rVert_{Enorm}, \lVert\mathbf{E}_u e_u\rVert_{Enorm}, \lVert\mathbf{E}_z e_z\rVert_{Enorm}) \le 0.001 \\
\text{and} \quad &\max(\lVert\mathbf{T}_p \varepsilon_p\rVert_{Tnorm}, \lVert\mathbf{T}_{pv} \varepsilon_{pv}\rVert_{Tnorm}) \qquad\qquad \le 0.001
\end{aligned}
\tag{9.12}
$$

hold for each step of the solution, where *Enorm* and *Tnorm* are RMS or ∞ norms.

Although *constraint* accuracy is maintained throughout a simulation, it is important to emphasize that accuracy of the *state variables* is 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.

10 Time Stepping

The system described by equations (8.85)–(8.88) 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 (8.85) provides only n independent equations, but the second adds n_{quat} more for a total of n_q kinematic equations. Then equation (8.86) provides $n+m^*$ with the first line and n_z more with the second line. That leaves (8.87) and (8.88) as $2m_p+m_v$ "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 (8.85) and (8.86) could be integrated perfectly, the solutions would indeed stay on the manifolds since they start out that way and equations (8.85) and (8.86) 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*[8] to be discussed below. We'll make use here of the scaling, tolerance, and accuracy theory from Chapter 9.

10.1 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

---

[*] When $\mathbf{G}$ 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 (8.86) if the constraints are (slightly) inconsistent.

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

$$\varepsilon_n(q) = \mathbf{n}(q) = \begin{pmatrix} |\mathbf{q}_1| - 1 \\ \vdots \\ |\mathbf{q}_{nquat}| - 1 \end{pmatrix} \tag{10.1}$$

$$\varepsilon_p(q) = \mathbf{p}(\mathrm{t}; q) \tag{10.2}$$

$$\varepsilon_{pv}(u) = \begin{bmatrix} \dot{\mathbf{p}} \\ \mathbf{v} \end{bmatrix}(\mathrm{t}, \mathrm{q}; u) \tag{10.3}$$

$$\varepsilon_{pva}(\dot{u}) = \begin{bmatrix} \ddot{\mathbf{p}} \\ \dot{\mathbf{v}} \\ \mathbf{a} \end{bmatrix}(\mathrm{t}, \mathrm{q}, \mathrm{u}; \dot{u}) \tag{10.4}$$

where arguments before the semicolon are fixed at their current values. Note that these are unweighted errors; ε_p and ε_{pv} need to be weighted using the constraint weighting matrices $\mathbf{T}_p$ and $\mathbf{T}_{pv}$ discussed in section 9.3. When some of the q and u values are known due to pre-scribed motion (see section 8.3), then $q=\{q_p, q_f\}$ and $u=\{u_p, u_f\}$ and we are only able to modify the free variables q_f and u_f to satisfy the violated constraints.

Then we would like to find the smallest change to q_f that will drive ε_p to 0, and the smallest change to u_f that will drive ε_{pv} to 0. Those "smallest" changes correspond to a least squares projection in a weighted direction we call the *error norm* (E norm), normal to the constraint manifold, for which a theorem given in reference 8 guarantees that this projection also *improves* the solution to the differential equations. See section 9.1 for a discussion of the E norm. ε_{pva} is satisfied exactly when we solve equation (8.86), and ε_n is always satisfied simply by normalizing the quaternions $\mathbf{n}_k \subset q$, which is a 2-norm projection that can be done separately from everything else, although no projection is done for quaternions that are contained in q_p.

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. To avoid an excess of subscripts we will drop the subscript "*f*" here, but keep in mind that we

106

are working only with free variables, and matrix columns corresponding to prescribed variables are removed. All constraint equations remain, however, even as the number of variables available to solve them is reduced by prescribed motion.

The full Newton steps in projection are

$$\left. \begin{aligned} \bar{\mathbf{P}}_{q}(q^{(i)}) \Delta q_{ELS} &= \mathbf{T}_{p} \varepsilon_{p}(q^{(i)}) \\ q^{(i+1)} &= q^{(i)} - \mathbf{E}_{q}^{+} \Delta q_{ELS} \end{aligned} \right\} \text{iterate} \qquad (10.5)$$

$$\forall \mathbf{n}_{k} \subset q^{(last)} : \quad \mathbf{n}_{k}^{\text{final}} = \mathbf{n}_{k}/|\mathbf{n}_{k}| \qquad (10.6)$$

$$\left. \begin{aligned} \bar{\mathbf{V}}(q^{\text{final}}) \Delta u_{ELS} &= \begin{bmatrix} t_{c} \mathbf{T}_{p} \varepsilon_{\dot{p}}(u^{(i)}) \\ \mathbf{T}_{v} \varepsilon_{v}(u^{(i)}) \end{bmatrix} \\ u^{(i+1)} &= u^{(i)} - \mathbf{E}_{u}^{-1} \Delta u_{ELS} \end{aligned} \right\} \text{iterate} \qquad (10.7)$$

where

$$\bar{\mathbf{P}}_{q}(q) = \mathbf{T}_{p} \, \mathbf{P}_{q}(q) \, \mathbf{E}_{q}^{+} \qquad (10.8)$$

$$\bar{\mathbf{V}}(q) = \begin{bmatrix} t_{c} \mathbf{T}_{p} \mathbf{P}(q) \\ \mathbf{T}_{v} \mathbf{V}(q) \end{bmatrix} \mathbf{E}_{u}^{-1} \qquad (10.9)$$

$$\mathbf{E}_{q} = \mathbf{N} \mathbf{W}_{u} \mathbf{N}^{+} \quad (\Rightarrow \mathbf{E}_{q}^{+} = \mathbf{N} \mathbf{W}_{u}^{-1} \mathbf{N}^{+}) \qquad (10.10)$$

$$\text{and} \quad \mathbf{P}_{q}(q) = \mathbf{P}(q) \mathbf{N}^{+}. \qquad (10.11)$$

(Columns of $\bar{\mathbf{P}}_{q}$ and $\bar{\mathbf{V}}$ corresponding to prescribed variables are removed.)

We iterate (10.5) until we have calculated a final value $q^{(last)}$ that satisfies the holonomic constraint equations (10.2) to within a specified tolerance, then using equation (10.6) project the quaternions in $q^{(last)}$ via their normalization constraints (10.1). That gives us q^{final} which satisfies all the constraints (10.1) and (10.2). We then iterate (10.7) with $\bar{\mathbf{V}}$ calculated at q^{final} while solving for the final velocity value u^{final} which satisfies the velocity constraints (10.3). Note that we must perform a least squares solution to the linear system at each iteration, and that the diagonal weighting matrices $\mathbf{E}_{u}$, $\mathbf{T}_{p}$, and $\mathbf{T}_{v}$ must be constant during the iteration (although $\mathbf{E}_{q}$ may change a little). Typically, $\mathbf{P}$ and $\mathbf{V}$ are block-structured matrices (and weighting preserves that structure). For efficiency, uncoupled blocks should be treated separately in equations (10.5) and (10.7).

Normalizing a quaternion as in equation (10.6) 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 an *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 $e_y = \{e_q, e_u, e_z\}$ in its computation of integrated 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. 8 and ref. 10, §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

$$\left.\begin{aligned}\bar{\mathbf{P}}_q(q^{\text{final}})e_{wq}^{\perp} &= \bar{\mathbf{P}}_q(q^{\text{final}})\mathbf{E}_q e_q \\ e_q' &= e_q - \mathbf{E}_q^{+} e_{wq}^{\perp}\end{aligned}\right\} \text{ no iteration} \tag{10.12}$$

$$\forall e_{\mathbf{q}_k}' \subset e_q' : \hat{e}_{\mathbf{q}_k} = e_{\mathbf{q}_k}' - (e_{\mathbf{q}_k}' \bullet \mathbf{n}_k^{\text{final}})\mathbf{n}_k^{\text{final}} \quad \text{quaternions} \tag{10.13}$$

$$\left.\begin{aligned}\bar{\mathbf{V}}(q^{\text{final}})e_{wu}^{\perp} &= \bar{\mathbf{V}}(q^{\text{final}})\mathbf{E}_u e_u \\ \hat{e}_u &= e_u - \mathbf{E}_u^{-1} e_{wu}^{\perp}\end{aligned}\right\} \text{ no iteration} \tag{10.14}$$

(Here as above we are working only with the error slots corresponding to free variables, not prescribed ones.)

Again we need to find least-squares solutions to the underdetermined systems (10.12) and (10.14). Then we set $\hat{e}_y = \{\hat{e}_q, \hat{e}_u, \hat{e}_z\}$ 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 (10.12) and (10.14) are linear systems so no iteration is needed, and again block

108

structure in **P** and **V** should be exploited for efficiency. After this projection the integrator should use the revised estimate $\hat{e}_y$ as its error estimate instead of the original estimate e_y (using the **W** norm).

We can use a pseudoinverse to find the least squares solution at each step. The pseudoinverse A^+ of an $m{\times}n$ matrix A, with $m{<=}n$ and full row rank (i.e. $rank(A){=}m$) is given by $A^+{=}A^T(AA^T)^{-1}$, 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., $rank(A){<}m$.

Looking now at the weighted holonomic position constraint iteration matrix $\bar{\mathbf{P}}_q$ in equation (10.8) we see that the pseudo inverse we need is

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

The corresponding velocity constraint projection from (10.9) is

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

When there are no non-holonomic constraints, we have $\bar{\mathbf{V}} = \bar{\mathbf{P}}$ the velocity projection is just

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

The constraint projections can be performed sequentially (for proof, see ref. 10, §3.8.3). First, with t fixed at $\hat{t}$ we must find some $q = q^{(last)}$ that satisfies the holonomic constraint equations to within a specified tolerance. Then we normalize the quaternions in $q^{(last)}$ (which by construction cannot affect any of the holonomic constraints) and call the result q^{final}. After that we freeze q at q^{final} 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.

10.2 Simplified equations

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

differential eqns.	$\dot{y} = f(t, y)$	(10.15)
algebraic eqns.	$c(t, y) = 0$	(10.16)
initial conditions	$c(t_0, y_0) = 0$	(10.17)

with the guarantee that equation (10.15) is solved in such a way that any new constraints introduced by time-differentiating equation (10.16) are satisfied automatically; that is, $\dot{c}(t, y, f(t, y)) = 0$ whenever equations (10.15) and (10.16) are satisfied. There is an $n_y \times n_y$ block-diagonal weighting matrix $\mathbf{E}$ and an $n_c \times n_c$ diagonal tolerance matrix $\mathbf{T}$, and corresponding norms as discussed in Chapter 9. To solve this system to accuracy α, the following two conditions must be satisfied at each integration step:

$$\left\| \varepsilon_y \right\|_{\text{ERMS}} \leq \alpha \tag{10.18}$$

$$\left\| c(t, y) \right\|_{\text{TRMS}} \leq \alpha \tag{10.19}$$

where the ε_y are the post-projection local state errors introduced by an integration step. When conditions (10.18) and (10.19) are met, the integrator can accept the step.

10.3 Update rates for state variables

During a time-stepping study, Simbody supports three distinct update rates for time-varying variables in S : *slow (discrete)*, *dynamic*, and *fast*. Slow variables are updated only occasionally upon occurrence of events and never change during a continuous interval. Dynamic variables follow differential equations, evolve smoothly during continuous intervals, and have well-defined time derivatives. Fast variables are those whose response may be considered instantaneous on the scale of the dynamic variables. The dynamic system is thus always in quasi-static equilibrium with respect to the fast variables, whose values may be determined by difference equations, equilibrium conditions, or algorithmically. Fast variables change continuously in response to dynamic ones but can affect dynamic time evolution only indirectly by affecting the dynamic variables' derivatives.

Only variables in the State's y partition can be dynamic. So we can partition the state by update rate as follows:

$$\begin{aligned}
S_{slow} &\triangleq d_{slow} \cup y_{slow} \\
S_{fast} &\triangleq d_{fast} \cup y_{fast} \\
S_{dyn} &\equiv y_{dyn}
\end{aligned} \qquad (10.20)$$

where

$$\begin{aligned}
y_{slow} &\triangleq q_{slow} \cup u_{slow} \\
y_{dyn} &\triangleq q_{dyn} \cup u_{dyn} \cup z \\
y_{fast} &\triangleq q_{fast} \cup u_{fast} \\
\text{with} \quad y &= y_{slow} \cup y_{dyn} \cup y_{fast}
\end{aligned} \qquad (10.21)$$

There also exists a set of time derivatives $\dot{y} = \dot{q} \cup \dot{u} \cup \dot{z}$, with

$$\begin{aligned}
\dot{y}_{slow} &= \dot{y}_{fast} = 0 \\
\dot{y}_{dyn} &= \frac{d}{dt} y_{dyn}
\end{aligned} \qquad (10.22)$$

Note that $\dot{y}_{slow}$ is zero because y_{slow} is constant during a step, while $\dot{y}_{fast}$ is zero for the opposite reason: y_{fast} changes so quickly that by the time we look it has already reached equilibrium and is thus no longer changing. There are no derivatives defined for variables in x, which are not permitted to be dynamic (although they can be fast or slow).

When we want to denote variables which affect a particular stage and also have a particular update rate, we combine the sub- and superscripts, for example, $S_{fast}^{pos} \triangleq S^{pos} \cap S_{fast} = d_{fast}^{pos} \cup q_{fast}$. Note that a variable's stage effect is an inherent property of that variable, while the rate at which it is to be updated is a run time choice that may be different under different circumstances.

At run time we must choose for every mobilizer:

- How motion is driven: forces, acceleration, velocity, or position
- The rate at which the driven motion is updated: slow (discrete), dynamic, or fast. A force-driven mobilizer is always dynamic.

When mobilizer j is updated dynamically, it is driven by differential equations relating position $q[j]$ and velocity $u[j]$ to acceleration $\dot{u}[j]$. Normally, $\dot{u}[j]$ is calculated as a response

to forces, with $u[j]$ and then $q[j]$ calculated from $\dot{u}[j]$ by numerical integration. In that case the dynamic mobilizer j is said to be *free*. When motion is known analytically, the mobilizer is said to be *prescribed*. Both free and prescribed mobilizers are considered *dynamic* mobilizers since they are both defined by time-varying differential equations. If instead the driven motion is fast or slow, we have a *fast* or *slow* mobilizer, resp. All mobilities of a multi-dof mobilizer must have the same update rate (that is, slow, prescribed, free, or fast). Note that for any mobilizer that is driven at acceleration or velocity level, higher levels are determined by numerical integration and are thus free.

There are some restrictions on the use of "fast" motion, because fast variables may not directly affect dynamic ones: (1) If a mobilized body is fast, then all its children and descendents in the multibody tree must also be fast. (2) If a constraint involves any fast body or mobility, that makes it a *fast constraint* that is not part of the dynamic system. Instead it is solved taking the dynamic variables as fixed, with only the fast variables varying.

Driven by	Update rate	$\dot{u}$	u	q
forces	dynamic (free)	$\dot{u}_f\left(t,q,u,z,\dot{u}_p\right)$	$u_f = \int dt$	$q_f = \iint dt^2$
acceleration	dynamic (prescribed)	$\dot{u}_p(t,q_{pr},u_{pr})$	$u_f = \int dt$	$q_f = \iint dt^2$
velocity	dynamic (prescribed)	$\dot{u}_p = d/dt$	$u_p(t,q_{pr})$	$q_f = \int dt$
velocity	fast	$\dot{u}_p = 0$	u_{fast}: algebraic	$q_f = \int dt$
velocity	slow	$\dot{u}_p = 0$	u_{slow}: event-driven	$q_f = \int dt$
position	dynamic (prescribed)	$\dot{u}_p = d^2/dt^2$	$u_p = d/dt$	$q_p(t)$
position	fast	$\dot{u}_p = 0$	$u_p = 0$	q_{fast}: algebraic
position	slow	$\dot{u}_p = 0$	$u_p = 0$	q_{slow}: event-driven

Structured variables d that can affect time-varying quantities can be evaluated either continuously (fast) or at discrete times (slow); variables that affect model or instance parameters or are just used for reporting can't be continuous. Here the continuous variables cannot be defined by differential equations; they are always algebraic (or algorithmic).

	$d^{\text{model,instance,report}}$	$d^{\text{time,pos,vel,dyn,acc}}$
fast	n/a	d_{fast}: algebraic
slow	d_{slow}: event-driven	d_{slow}: event-driven

Also note that "end of step" can be an event, so event-driven (slow) variables can be updated as frequently as every step. However, they will not be updated during integration stages and cannot affect integrator step size selection in the current step.

10.3.1 Coupling

Variables which are mutually coupled must be solved simultaneously which can be computationally expensive. In many cases, modeling considerations dictate that some variables are "stronger" than others so coupling is only in the direction from driving variables to driven ones, lending an ordering to the computation that avoids some simultaneity. Simbody supports four levels of coupling for computations at a given realization stage:

1. Slow (discrete): these variables are event driven so are not updated at all during a continuous interval. Their values are taken as given at the beginning of a continuous interval and can affect all subsequent calculations but are themselves unaffected.

2. Dynamic, prescribed: these are q, u, z, or $\dot{u}$ variables defined by differential equations with explicit analytic solutions based only on already-available values. For example: positions as a function of time. This could represent an extremely high-bandwidth controller, or a locked joint. Prescribed accelerations are not affected by forces.

3. Dynamic, free: these are q, u, or z variables calculated by numerical integration. Corresponding accelerations $\dot{u}$ can additionally be coupled by algebraic constraint equations, which equations can depend on prescribed accelerations. They can thus be driv-

en by prescribed variables at the same level (e.g. prescribed velocity can drive free velocity) but not vice versa. A dynamic position can affect a prescribed velocity since that's a later stage. $\dot{z}$ variables cannot be coupled via constraints.

4. Fast (quasi-static): these variables are calculated from the values of dynamic variables (free or prescribed). They can affect forces so will change the behavior of free variables indirectly, through subsequent accelerations or by interaction with the numerical integration method's error test.

Free dynamic variables y_f are approximated numerically by an integration method, then any of them that are subject to constraint equations may need to be projected back to the constraint manifold via the System's `project()` solvers. Prescribed dynamic variables y_p are simply set to their appropriate values via the System's `prescribe()` solvers, once their dependencies are available. Finally, we must solve for the fast variables d_{fast} and y_{fast} using the System's `relax()` solver.

Depending on the operation we are performing, it will be convenient to partition S in a variety of different ways.

Variable type	$\Sigma=\{d,t,y\}$ $y=\{q,u,z\}$ stage(t)=Time stage(q)=Position stage(u)=Velocity stage(z)=Force stage(x) varies	scalar variables y	q	Generalized coordinates (positions)
			u	Generalized speeds (velocities)
			z	Force variables
		d		structured variables
		t		time
Simulation	$\Sigma=\{\Sigma^{def}, t, \Sigma^{var}, \Sigma^{report}\}$ $\Sigma^{def}=\{\Sigma^{topo}, \Sigma^{model}, \Sigma^{instance}\}$ $\Sigma^{var}=\{\Sigma^{pos}, \Sigma^{vel}, \Sigma^{force}, \Sigma^{acc}\}$	Σ^{def}		variables that define the modeled system: topology, model, instance
		t		time
		Σ^{var}		variant properties of the modeled system: position, velocity, forces, acceleration
		Σ^{report}		"output" states for reporting
Time stepping	$\Sigma_{pres}=\{q_p, u_p\}$ $\Sigma_{free}=\{q_f, u_f, z\}$ $\Sigma_{fast}=\{q_{fast}, u_{fast}, d_{fast}\}$	continuous variables	Σ_{pres}	prescribed analytically
			Σ_{free}	determined by numerical integration
			Σ_{fast}	Quasi-static variables determined algebraically after Σ_{pres} and Σ_{free}

	$\Sigma_{slow}=\{q_{slow},\ u_{slow},\ d_{slow}\}$	Σ_{slow}	Discrete "slow" variables (updated occasionally upon event occurrence)

10.4 How to take a time step

Here we deconstruct a single time step into its constituent parts.

Special handling is required at the start of a time stepping study. Time and all state variables must be given initial values, and then these must be modified to satisfy all constraints, starting with prescribed motion. The resulting state must be fully realized using to-be-updated values for discrete variables, as though we had just completed a zero-length step. Then we can take the first step as though it were any arbitrary step.

1. Handle events as needed, updating discrete states, realize through Acceleration stage.

2. Take trial step of continuous system (with projection), with success metric.

3. If metric unacceptable, reduce step and return to 2.

4. Check for Time-stage event triggers and handle them; <= Instance stage change is a restart meaning later event triggers are forgotten.

5. Realize(Positions); check for Position-stage event triggers and handle them; <= Instance stage change is a restart.

6. Repeat for Velocity, Force and Acceleration-stage event triggers.

7. If simulation is not done, go to step 1 to begin the next step.

Taking a trial step of the continuous system (we just updated discrete variables). Continuous system consists of dynamic (prescribed and free), and "fast" variables q_{fast}, u_{fast}, z_{fast}, and d_{fast}. There are a series of integrator stage evaluations and then a final setting of the state variables in which differential variables are projected to their constraint manifold.

All steps begin like this:

- If there are pending triggered events (at t^+):
    - Set $t=t^+$. Call time-triggered handlers to change slow variables; re-evaluate $\dot{y},e,c$; revise list of pending events
    - Repeat for pending position, velocity, dynamics, acceleration-triggered events

- If there are any beginning-of-step handlers, call them and re-evaluate. This cannot cause any other events to occur.
- Set $t^0=t$, $y^0=y$; invalidate d^0
- $\dot{y} = \texttt{realize[A]}(d,t,y)$
- Set $\dot{y}^0 = \dot{y}$
- Next step size to attempt is h

Fixed step explicit Euler (DAE step only):

- Only stage:
  $\rightarrow y'_{free} = y^0_{free} + h \cdot \dot{y}^0_{free}$
  Set $t = t^0 + h$; $\texttt{realize[T]}(t)$
  $q_{pres} = \texttt{prescribeQ}(t)$
  $q_{free}: \texttt{projectQ}(q_{pres}, q'_{free})$
  $u_{pres} = \texttt{prescribeU}(t, q_{pres}, q_{free})$
  $u_{free}: \texttt{projectU}(u_{pres}, u'_{free})$
    Test convergence of projection. (bad: shrink h goto$\rightarrow$)
  $d_{fast}, y_{fast}: \texttt{relax[TPVFA]}(t, y, \dot{y}(d_{fast}, y_{fast}); d_{fast}, y_{fast})$
    Test convergence of relaxation. (bad: shrink h goto$\rightarrow$)
- (Error test OK.) Check for events. If any:
  - Localize event(s) to window $w = (t^-, t^+]$
  - Back up the state to $t = t^-$:
    $y'_{free}(t^-) = \texttt{interpolate}(t^-, y^0_{free}, y'_{free})$
    $y_{pres}, y_{free} = \texttt{presAndProjQU}(t^-, y'_{free})$
    $x_{fast}, y_{fast}: \texttt{relax[TPVFA]}(t^-, y, \dot{y}(d_{fast}, y_{fast}); d_{fast}, y_{fast})$
  - (Projection and relaxation must converge or there is something seriously wrong.)
- Successful, event-free step taken to time t. May be events pending for time t^+.

Explicit trapezoid (w/explicit Euler error estimator):

- First stage:
  $\rightarrow y^1_{free} = y^0_{free} + h \cdot \dot{y}^0_{free}$
  $t = t^0 + h$; $\texttt{realize[T]}(t)$
  $y^1_{pres} = \texttt{prescribeQU}(t, y^1_{free})$
  $d^1_{fast}, y^1_{fast}: \texttt{relax[TPVFA]}(t, y^1, \dot{y}(d_{fast}, y_{fast}); d_{fast}, y_{fast})$
    Test convergence of relaxation. (bad: shrink h goto$\rightarrow$)
- Final stage:

$$\dot{y}^1 = \texttt{realize[A]}(t, d^1, y^1)$$

$$y'_{free} = y^0_{free} + \tfrac{h}{2}(\dot{y}^0_{free} + \dot{y}^1_{free})$$

$$\varepsilon'_{free} = y'_{free} - y^1_{free}$$

$$y_{pres}, y_{free}, \varepsilon_{free} = \texttt{presAndProjQU}(y_{pres}, y'_{free}, \varepsilon'_{free})$$

Test $\|\varepsilon_{free}\|$ and convergence of projection. (bad: shrink h goto$\rightarrow$)

$$d_{fast}, y_{fast} : \texttt{relax[TPVFA]}(t, y, \dot{y}(d_{fast}, y_{fast}); d_{fast}, y_{fast})$$

Test convergence of relaxation. (bad: shrink h goto$\rightarrow$)

- (Error test OK.) Check for events. If any:
  - Localize event(s) to window $w = (t^-, t^+]$
  - Back up the state to $t = t^-$:

    $$y'_{free}(t^-) = \texttt{interpolate}(t^-, y^0_{free}, y'_{free})$$

    $$y_{pres}, y_{free} = \texttt{presAndProj[PVF]}(t^-, y'_{free})$$

    $$x_{fast}, y_{fast} : \texttt{relax[TPVFA]}(t^-, y, \dot{y}(x_{fast}, y_{fast}); x_{fast}, y_{fast})$$

  - (Projection and relaxation must converge or there is something seriously wrong.)
- Successful, event-free step taken to time t. May be events pending for time t^+.

Verlet

- First stage:

  $$\rightarrow q'_{free} = q^0_{free} + h \cdot \dot{q}^0_{free} + \tfrac{h^2}{2} \cdot \ddot{q}^0_{free}$$

  $$t = t^0 + h; \quad \texttt{realize[T]}(t)$$

  $$q_{pres}, q_{free} = \texttt{presAndProjQ}(t, q'_{free});$$

  $$\texttt{realize[P]}(t, q)$$

  $$u_{pres} = \texttt{prescribeU}(t, q_{pres}, q_{free})$$

  $$u^1_{free} = u^0_{free} + h \cdot \dot{u}^0_{free}; \quad z^1 = z^0 + h \cdot \dot{z}^0$$

  $$\texttt{realize[V]}(t, q, u)$$

  $$d^1_{fast}, q_{fast} : \texttt{relax[TP]}(t, y^1_{dyn}, \dot{y}(d_{fast}, y_{fast}); d_{fast}, q_{fast})$$

  $$k = 1$$

- Loop until u and z are converged:

  $$d^k_{fast}, uz^k_{fast} : \texttt{relax[VFA]}(t, y^k_{dyn}, \dot{y}(x_{fast}, y_{fast}); x_{fast}, uz_{fast})$$

  $$u^k, \dot{z}^k = \texttt{realize[VFA]}(t, q, u^k, z^k) \quad \textcolor{red}{\text{expensive once}}$$

  $$++k$$

  $$u^k_{free} = u^0_{free} + \tfrac{h}{2}(\dot{u}^0_{free} + \dot{u}^{k-1}_{free}); \quad z^k = z^0 + \tfrac{h}{2}(\dot{z}^0 + \dot{z}^{k-1})$$

  $$\delta_u = \|u^k - u^{k-1}\|; \quad \delta_z = \|z^k - z^{k-1}\|$$

  If δ not converged, continue looping.

$$u'_{free} = u^k_{free}; \quad z' = z^k$$

- u and z have converged to 2nd order

$$\dot{q}'_{free} = \mathbf{N} u'_{free}$$

Compute low order estimates:

$$q''_{free} = q^0_{free} + \tfrac{h}{2}(\dot{q}^0_{free} - \dot{q}'_{free}); \quad u''_{free} = u^1_{free}; \quad z'' = z^1$$

$$\varepsilon'_{free} = y''_{free} - y'_{free} \quad \text{error estimate}$$

$$\varepsilon_{free,q} = \texttt{projErrEstQ}(\varepsilon'_{free,q})$$

$$uz_{free}, \varepsilon_{free,uz} = \texttt{presAndProjU}(y_{pres}, uz'_{free}, \varepsilon'_{free,uz})$$

  Test $\left\| \varepsilon_{free} \right\|$ and convergence of projection. (bad: shrink h goto→)

$$d_{fast}, uz_{fast} : \texttt{relax[VFA]}(t, y_{dyn}, \dot{y}(d_{fast}, y_{fast})); d_{fast}, uz_{fast})$$

  Test convergence of relaxation. (bad: shrink h goto→)

- Successful continuous step to t. Now check for events.

10.4.1 Setting the values of prescribed variables

Prescribed variables $y_p \subset y$ and $\dot{y}_p \subset \dot{y}$ are defined by analytically-known explicit func-tions. These include $q_p(t)$, $u_p(t, q_{dyn})$, $\dot{u}_p(t, q_{dyn}, u_{dyn})$. ($q_{dyn} \triangleq q_{pres} \cup q_{free}$, etc.) Note that earlier-stage prescribed and dynamic variables can affect later-stage prescribed ones.

A System defines `prescribeQ()` and `prescribeU()` solvers which sets the values for that stage's prescribed variables, taking values of earlier-stage variables as given. Be sure to call them in this order: `prescribeQ(), projectQ(), prescribeU(), projectU()`.

10.4.2 Relaxation of fast variables

After the values of time and the differential variables $y_{pres}, y_{free} \subset y$ have been calculated by an integrator, the fast variables $y_{fast} \subset y$ and $d_{fast} \subset d$ need to be allowed to respond to the new values, to attain a new quasi-static equilibrium. We call this process "relaxation" alt-hough in general a System is permitted to use any method to solve for these variables, provided the correct order of dependencies is followed. A System provides a `relax`(*stage*, *tol*) solver which will drive a particular stage's fast variables to their new equilibrium, solving the associated equations to the given tolerance.

The time-stepping relaxation stages and the variables relaxed at each stage are as follows:

1. d_{fast}^{time}:   `relax(Time)`
2. $q_{fast}, d_{fast}^{\text{pos}}$:   `relax(Position)`
3. $u_{fast}, d_{fast}^{\text{vel}}$:   `relax(Velocity)`
4. d_{fast}^{force}:   `relax(Force)`
5. d_{fast}^{acc}:   `relax(Acceleration)`

A System may also support relaxation of report-stage variables d^{report} and definition-stage variables d^{model} and d^{instance}, but these are not relevant for time stepping.

11 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.

11.1 General Force Subsystem

Simbody comes with a force subsystem called GeneralForceSubsystem. It can be used to add a variety of standard forces to a system, such as linear springs and dampers. It also provides a mechanism for adding user defined forces.

11.2 Hertz/Hunt and Crossley contact model subsystem

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

11.2.1 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 13. 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.

11.2.2 The model

This model is based on Hertz theory of elastic contact,[11] and the Hunt and Crossley model for damping.[12] 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 13. 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 15 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 $\mathbf{n}$ 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.
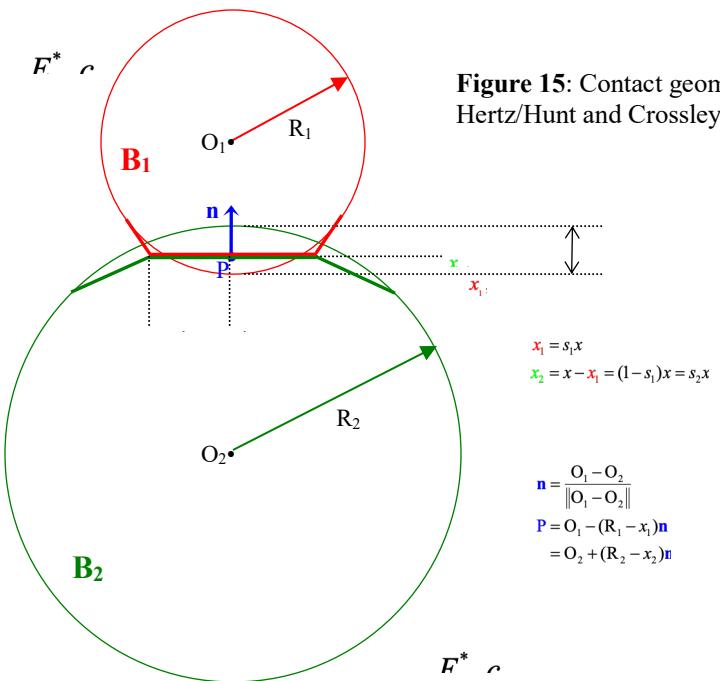
Figure 15: Contact geometry for the Hertz/Hunt and Crossley model.

$x_1 = s_1 x$
$x_2 = x - x_1 = (1 - s_1)x = s_2 x$

$\mathbf{n} = \dfrac{O_1 - O_2}{\|O_1 - O_2\|}$

$P = O_1 - (R_1 - x_1)\mathbf{n}$
$ = O_2 + (R_2 - x_2)\mathbf{n}$

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 $e = 1 - c v_i$ 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	Poisson's ratio ν	Plane-strain	Dissipation coefficient

	(GPa)	(unitless)	modulus E* (GPa)	(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 12 and 13 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

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

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

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

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 half-space, 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 $\dot{x}$: $c = c_1 s_1 + c_2 (1 - s_1)$, 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:

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

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 14 for proof). The "yanking" situation corresponds to pulling the bodies apart faster than they can undeform.

11.2.3 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 15, 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.

11.3 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!

11.3.1 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.

11.3.2 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.

11.3.2.1 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.

11.3.2.2 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.

11.3.2.3 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).

130

11.3.3 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 pico-seconds (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 $\times$ length^2/time^2) we get energy per mole in g-nm^2/ps^2=10^3kg-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.

---

[*] 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.

11.3.4 Defining a force field

TODO

11.3.5 Defining the molecules

TODO

11.3.6 Defining bodies and attaching the molecule to them

TODO

11.3.7 Running a simulation

TODO

11.3.8 Theory

TODO

132

12 Appendix: derivations

This collects detailed derivations for and discussion about some of the results presented earlier.

12.1 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:

$$
{}^{F}\left[{}^{M}\mathbf{Q}_{i}^{B} \right]
$$

"From" point or frame — "To" point or frame — Expressed-in, if not M — Type of quantity and which instance

The type of quantity (the central black symbol) is the only required piece. The right subscript conveys the particular instance being measured. The superscripts are bodies, frames, or points. When useful, points are considered to be the origin of a frame that is parallel to the point's body frame, but with the origin shifted to the point. That frame is then considered the "measured in" and "expressed in" frame unless otherwise stated.

Here are the symbols we conventionally use for particular quantities, shown in nice typography and then the crude equivalent we have to use in code.

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 mobilizer frame for the mobilized 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 ${}^P X^F$.
F_O	Fo	The origin point of some frame F.
B_C	Bc	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.
$\mathcal{G}_B$	Gb	The gyration matrix of body B. By default this is taken about the B origin and expressed in the B frame.
$\mathcal{J}_B$	Jb	The inertia of body B, where $\mathcal{J}_B = m_B \mathcal{G}_B$. 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.
${}^R p^S$ ${}^A p^B$ ${}^G[{}^A p^B]$	p_RS p_AB p_AB_G	The position 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, ${}^A p^B = {}^{A_O} p^{B_O}$. If the position vector is expressed in a frame other than the "from" point's frame, we use the bracket notation shown.
${}^A X^B$	X_AB	The *spatial transform* (rotation and translation) expressing frame B in frame A. ${}^A X^B = \left({}^A R^B \mid {}^A p^B \right)$.
${}^A V^B$ ${}^A V^Q$	V_AB V_AQ	The *spatial velocity* of frame B in A, expressed in A. This includes the angular velocity of B in A and the linear velocity of B_O 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 B_O. Q can be considered a coordinate frame parallel to B but with its origin shifted to Q.

$^{A}A^{B}$ $^{A}A^{Q}$	A_AB A_AQ	The *spatial acceleration* of frame B in A, expressed in A. This includes the angular acceleration of B in A and the linear acceleration of B_O 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 acceleration is of Q in A rather than of B_O. Q can be considered a coordinate frame parallel to B but with its origin shifted to Q.
<mark>MORE</mark> <mark>TODO</mark>		

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 $\mathcal{J}_B$, but if the inertia were instead taken about B's center of mass point B_C, the symbol would be $\mathcal{J}_B^{B_C}$.

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 B_C is to be measured in the local frame of another body A, we would write $^{A}p^{B_C}$ (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 $^{A_C}p^{B_C}$ where the expressed-in frame A is inferred from A_C. 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 $^{G}\left[^{A_C}p^{B_C} \right] = {}^{G}R^{A} \cdot {}^{A_C}p^{B_C}$.

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

$$^{G}\left[{}^{A_C}\dot{p}^{B_C} \right] \triangleq \frac{{}^{G}d}{dt}\left({}^{G}\left[{}^{A_C}p^{B_C} \right] \right)$$

$$= {}^{G}\left[{}^{A_C}\dot{p}^{C_B} \right] + {}^{\dot{G}}\left[{}^{A_C}p^{B_C} \right]$$

$$= {}^{G}\left[{}^{A_C}\dot{p}^{C_B} \right] + {}^{G}\omega^{A} \times {}^{G}\left[{}^{A_C}p^{B_C} \right]$$

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

$$^B[^AQ] \triangleq {}^BR^A \cdot {}^AQ$$

$$^B[^AQ]^{\cdot} \triangleq {}^B\dot{R}^A \cdot {}^AQ = ({}^B\omega^A_\times \cdot {}^BR^A) \cdot {}^AQ$$

$$= {}^B\omega^A_\times \cdot {}^B[^AQ]$$

$$^A\dot{Q} \triangleq \frac{^Ad}{dt} {}^AQ$$

$$^B[^A\dot{Q}] \triangleq \frac{^Bd}{dt} {}^B[^AQ] = {}^B[^A\dot{Q}] + {}^B[^AQ]^{\cdot}$$

$$= {}^B[^A\dot{Q}] + {}^B\omega^A_\times \cdot {}^B[^AQ]$$

12.2 Re-expressing spatial quantities

For any quantity Q we use the notation $^Z[Q]$ 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 v is a vector currently expressed in frame A, then $^Z[v] \triangleq {}^ZR^A \cdot v$. If M is a tensor (matrix) currently expressed in frame A, then $^Z[M] \triangleq {}^ZR^A \cdot M \cdot ({}^ZR^A)^\mathsf{T} = {}^ZR^A \cdot M \cdot {}^AR^Z$. We use similar definitions for spatial quantities. If V is a *spatial* vector currently expressed in frame A, then we define $^ZR^A \cdot V \triangleq \begin{pmatrix} {}^ZR^A & 0 \\ 0 & {}^ZR^A \end{pmatrix} \cdot V$. For example,

$$^Z[^AV^B] = {}^ZR^A \cdot {}^AV^B \triangleq \begin{pmatrix} {}^ZR^A & 0 \\ 0 & {}^ZR^A \end{pmatrix} {}^AV^B = \begin{pmatrix} {}^ZR^A \cdot {}^A\omega^B \\ {}^ZR^A \cdot {}^Av^{B_O} \end{pmatrix} = \begin{pmatrix} {}^Z[^A\omega^B] \\ {}^Z[^Av^{B_O}] \end{pmatrix}.$$

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

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

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

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

where we have made use of identity (3.6), the fact that if U is a 3x3 orthogonal matrix, then $U \cdot \mathbf{v}_\times \cdot U^T = (U \cdot \mathbf{v})_\times$. (Recall that rotation matrices are orthogonal.)

12.3 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 and spatial inertias from outboard bodies in an inward direction. The rigid body shift matrix ${}^P S^Q$ is used to shift a spatial motion vector (e.g., velocity or acceleration) at point Q to the equivalent spatial motion vector acting at point P. The transpose of that matrix ${}^Q S*^P \triangleq \left({}^P S^Q\right)^T$ is used to shift a spatial force vector (e.g., force or impulse) acting at point P to the equivalent spatial force acting at point Q, and both forms are used when shifting inertias. The operators are

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

so that

$$ V^P \triangleq \begin{pmatrix} \omega \\ v^P \end{pmatrix} = {}^P S^Q \cdot V^Q = \begin{pmatrix} \omega \\ v^Q - \omega \times {}^P p^Q \end{pmatrix} $$

and

$$ F^Q \triangleq \begin{pmatrix} \mu \\ f^Q \end{pmatrix} = {}^Q S*^P \cdot F^P = \begin{pmatrix} \mu - {}^P p^Q \times f^P \\ f^P \end{pmatrix}. $$

12.3.1 Rigid body shift of rigid body spatial inertia

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

137

$$M^P = {}^PS*^Q \cdot M^Q \cdot {}^QS^P = m_B \begin{pmatrix} 1 & {}^Pp^Q_\times \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \mathcal{G}^Q & {}^Qp^C_\times \\ -{}^Qp^C_\times & \mathbf{1}_3 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -{}^Pp^Q_\times & 1 \end{pmatrix}$$

$$= m_B \begin{pmatrix} \mathcal{G}^Q - \left({}^Qp^C_\times\right)^2 + \left({}^Pp^C_\times\right)^2 & {}^Pp^C_\times \\ -{}^Pp^C_\times & \mathbf{1}_3 \end{pmatrix}$$

$$= m_B \begin{pmatrix} \mathcal{G}^C + \left({}^Pp^C_\times\right)^2 & {}^Pp^C_\times \\ -{}^Pp^C_\times & \mathbf{1}_3 \end{pmatrix}$$

$$= m_B \begin{pmatrix} \mathcal{G}^P & {}^Pp^C_\times \\ -{}^Pp^C_\times & \mathbf{1}_3 \end{pmatrix}$$

where ${}^Pp^C = {}^Pp^Q + {}^Qp^C$.

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

${}^Pp^C = {}^Pp^Q + {}^Qp^C$	3
$\left({}^Qp^C_\times\right)^2$, $\left({}^Pp^C_\times\right)^2$	22 (11 each)
$\mathcal{G}^P$	12 (2 adds)
TOTAL	37 flops

12.4 Inversion of rigid body spatial inertia

In case you find yourself with a need to invert a rigid body spatial inertia, you can do it very efficiently. We have symmetric, positive definite

$$M = m \begin{pmatrix} \mathcal{G} & p_\times \\ -p_\times & \mathbf{1}_3 \end{pmatrix} \tag{12.1}$$

Then its inverse is also a symmetric matrix

$$M^{-1} = \frac{1}{m} \begin{pmatrix} \Gamma & * \\ p_\times\Gamma & p_\times\Gamma p_\times^T + \mathbf{1}_{3\times3} \end{pmatrix} \tag{12.2}$$

where $\Gamma = (\mathcal{G} - p_\times^2)^{-1}$

So even though this is a 6x6 matrix, only a symmetric 3x3 need be inverted.

This can be a very useful optimization for lone rigid bodies on free joints to Ground, where the joint matrix H is a constant identity matrix in Ground. In that case the mobility space

mass matrix $D = H^\mathsf{T} P H$ (see below) is just the body's rigid body spatial inertia M re-expressed in Ground and is itself a rigid body spatial inertia.

Because of the orthogonality of rotation matrices, we also have

$$\left[M^B\right]_F^{-1} \triangleq ({}^F R^B M^B \, {}^B R^F)^{-1} = {}^F R^B (M^B)^{-1} \, {}^B R^F \triangleq \left[(M^B)^{-1}\right]_F \qquad (12.3)$$

which may permit the inverse spatial inertia to be usefully precalculated in some cases.

Note that an articulated body inertia is a general symmetric matrix so there is no "trick" way to invert it as there is for a rigid body spatial inertia.

12.5 Articulated body inertia

An articulated body inertia (ABI) matrix $P(q)$ contains the spatial inertia properties that a body appears to have when it is the free base body of an articulated multibody tree in a given configuration q. Despite the complex relative motion that occurs within a multibody tree, at any given configuration q there is still a linear relationship between a spatial force F applied to a point of the base body and the resulting acceleration A of that body and that point: $F = P(q)A + c$, where c is a velocity-dependent inertial bias force. P is thus analogous to a rigid body spatial inertia (RBI), but for a body which has other bodies connected to it by joints which are free to move.

An ABI P is a symmetric 6x6 spatial matrix, consisting of 2x2 blocks of 3x3 matrices, similar to the RBI. However, unlike the RBI which has only 10 independent elements, all 21 non-repeated elements of P are significant. For example, the apparent mass of an articulated body depends on which way you push on it, and in general there is no well-defined center of mass. This is a much more expensive matrix to manipulate than an RBI. In Simbody's formulation, we only work with ABIs in the Ground frame, so there is never a need to rotate or re-express them. (That is achieved by rotating RBIs prior to using them to construct the ABIs.) Thus only shifting operations need be performed when transforming ABIs from body to body. Cheap rigid body shifting is done when moving an ABI within a body or across a prescribed mobilizer; otherwise we have to perform an articulated shift operation which is quite expensive. For a full discussion of the properties of articulated body inertias, see

Section 7.1 (pp. 119-123) of Roy Featherstone's excellent 2008 book, Rigid Body Dynamics Algorithms.2

As a spatial matrix, an articulated body inertia is composed these of 3x3 subblocks:

$$P^B = \begin{pmatrix} \mathcal{J} & \mathcal{F} \\ \mathcal{F}^{\mathsf{T}} & \mathcal{M} \end{pmatrix} \tag{12.4}$$

Here $\mathcal{M}$ is the mass distribution (symmetric), $\mathcal{F}$ is the first mass moment distribution (full), and $\mathcal{J}$ is an inertia (second mass moment, symmetric).

12.5.1 Rigid body shift of articulated body inertia

Rigid body shifting of ABIs is done when an ABI is shifted across a prescribed (or welded) mobilizer. This is done with a rigid body shift operator $^P S^B$ as above. Here we're given an ABI P^B expressed in the Ground frame, taken about the origin B_O of body B. We would like to shift it to the origin P_O of its parent body's frame P, crossing the prescribed mobilizer connecting B to P.

$$P^P = {}^P S^{*B} \cdot P^B \cdot {}^B S^P = \begin{pmatrix} 1 & p_\times \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \mathcal{J} & \mathcal{F} \\ \mathcal{F}^{\mathsf{T}} & \mathcal{M} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -p_\times & 1 \end{pmatrix}$$
$$= \begin{pmatrix} \mathcal{J}' & \mathcal{F}' \\ \mathcal{F}'^{\mathsf{T}} & \mathcal{M} \end{pmatrix} \tag{12.5}$$

where

$$\mathcal{F}' = \mathcal{F} + p_\times \mathcal{M} \tag{12.6}$$

$$\mathcal{J}' = \mathcal{J} + p_\times \mathcal{F}^{\mathsf{T}} - \mathcal{F}' p_\times \tag{12.7}$$

Note that symmetry of $\mathcal{J}$ is preserved in equation (12.7) because

$$p_\times \mathcal{F}^{\mathsf{T}} - \mathcal{F}' p_\times = p_\times \mathcal{F}^{\mathsf{T}} - \mathcal{F} p_\times - p_\times \mathcal{M} p_\times$$
$$= \left\{ p_\times \mathcal{F}^{\mathsf{T}} + (p_\times \mathcal{F}^{\mathsf{T}})^{\mathsf{T}} \right\} - \left\{ p_\times^{\mathsf{T}} \mathcal{M} p_\times \right\} \tag{12.8}$$

and each of the quantities in {} is symmetric. Therefore we need only calculate the lower halves of $p_\times \mathcal{F}^{\mathsf{T}}$ and $\mathcal{F}' p_\times$ which, if done carefully, requires fewer calculations than calculating the symmetric terms in (12.8) directly would. Flop counts are:

$p_\times \mathcal{M}$ (full)	24

$\mathcal{F}' = \mathcal{F} + p_\times \mathcal{M}$	9
half of $p_\times \mathcal{F}^\mathsf{T} + \mathcal{F}' p_\times$	33
$\mathcal{J}'$	6
TOTAL	72 flops

This is a little less than twice as expensive as a rigid body shift of a *rigid* body inertia, which isn't bad considering there are more than twice as many meaningful elements in an ABI than an RBI. Unfortunately, this is the easy case!

12.5.2 Articulated shift of an articulated body inertia

This is the single most expensive operation in the Simbody kernel. We have the articulated body inertia P^B of a child body B which we would like to shift to its parent body A, but accounting for the movable (free, non-prescribed) mobilizer connecting B to A. That means that prior to a final rigid body shift from child to parent we have to remove the inertia projected on the current directions of the mobilities, since the parent can't "feel" that inertia through the floppy mobilizer in those directions. The projection of the ABI onto the mobilities is $P' = P^B H D^{-1} H^\mathsf{T} P^B$ and the final result we're looking for is $P^A += {}^A S *^B (P^B - P')^B S^A$. Here $D = H^\mathsf{T} P^B H$ is a symmetric, positive definite $n \times n$ mobility-space mass matrix; H is the $6 \times n$ hinge matrix associated with the connecting mobilizer.

		Pin/slider joint	Free joint
$(P^B H)_{6 \times n}$	$66n$	66	396
sym: $D_{n \times n} = H^\mathsf{T}(P^B H)$	$\frac{11}{2} n^2 + \frac{11}{2} n$	11	231
sym: D^{-1} (use Cholesky)	$\frac{5}{6} n^3 + n/, / = 10\ (?)$	10	240
$G_{6 \times n} = (P^B H) D^{-1}$	$12n^2 - 6n$	6	396
sym: $P'_{6 \times 6} = G(P^B H)^\mathsf{T}$	$42n - 21$	21	231
sym: $P'' = P^B - P'$	21	21	21
sym: $P''' = {}^A S *^B P''{}^B S^A$	72 (see above)	72	72
sym: $P^A += P'''$	21	21	21
total	$\frac{5}{6} n^3 + 17.5 n^2 + 117.5 n + 93$	228	1608

n	0	1	2	3	4	5	6
total/per dof	93	228	404/202	625/209	896/224	1222/245	1608/268

This is unlikely to be the optimal computation of an articulated shift but it is the best I could come up with for now. The above is a general-case treatment; there are many special cases that could be exploited some of which are discussed below.

In addition to P^A, several of the intermediate quantities in the above calculation are needed for subsequent operations, such as calculating accelerations. These are D, D^{-1}, and G.

12.5.3 Terminal bodies and base bodies

In a typical multibody tree, a substantial fraction of the bodies are terminal, meaning they have no outboard children, or base, meaning their parent body is Ground. For a terminal body we have $P^B = {}^G\!\left[M^B\right]$, that is, the articulated body inertia is just the rigid body inertia of B, re-expressed in the Ground frame (and that is still a rigid body inertia matrix). We would like to use that fact to reduce the op count required for the articulated shift of P^B to its parent.

Although P^B in the terminal-body case is an ordinary rigid body inertia, from inspecting the above table it is not obvious how to get much out of that except a modest savings in calculating $P^B H$ because D, D^{-1}, and G still need to be calculated and P', P'', P''', and P^A are all articulated body inertias. A few possible special cases:

- If H is constant when expressed in the body frame B, then $D = H^\mathsf{T}(P^B H) = (H^B)^\mathsf{T} M^B H^B$ is constant. (It doesn't matter what frame you calculate in since D is in mobility space which is its own coordinate system.) Thus D and D^{-1} can be precalculated. Unfortunately it is much more common for H to be constant in the *parent* body frame than in the child.

- For base bodies we don't need to update the parent's (Ground's) articulated body inertia since that is already infinite. Thus P', P'', P''', and P^A calculates are all unnecessary.

- Particles (lone point masses) and spherical rigid bodies are common special cases that can be handled extremely efficiently since their spatial inertia matrices are constant under rotation.

- A lone, free rigid body is a common case that can be handled efficiently. Here H can be made an identity matrix in Ground or in the body frame. With H identity in Ground, D is a rigid body spatial inertia, which can be inverted very efficiently (see section 12.4). With H identity in the body, D is a constant as discussed above so can be precalculated.

12.6 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

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

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 $\mathbf{J}$ by perturbing the state variables (this is called a finite difference approximation to $\mathbf{J}$). 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 $\mathbf{J}$ 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 $\mathbf{J}$ (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.

12.7 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. $\mathbf{J}$ as defined above can be very useful for minimizations involving search for equilibria. For satisfying constraints, the partial derivatives of the constraint equations (8.52) and (8.53) are required and Simbody can provide those analytically.

Root finding problems can be difficult when the coordinates are constrained, so it is convenient to define a new set of fully-independent coordinates. 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, using the $\mathbf{N}$ and $\mathbf{N}^{-1}$ operators provided by Simbody to work with the kinematic coupling matrix $\mathbf{N}$. 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.

12.8 Operator form of Simbody interface

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
$\dot{q} = \mathbf{N}_{[q]}u$	Position	void calcQDot(State, Vector u, Vector& qdot)	Convert generalized speeds to generalized coordinate time derivatives.
$\ddot{q} = \dot{\mathbf{N}}_{[q,u]}\dot{u} + \mathbf{N}_{[q]}\dot{u}$	Velocity	void calcQDotDot(State, Vector udot, Vector& qdotdot)	Convert generalized speed time derivatives to generalized coordinate 2nd time derivatives.
$f_{\mathrm{a}} = \mathbf{M}_{[q]}a$ $f_{\mathrm{c}} = \mathbf{G}_{[q]}^{\mathsf{T}}\lambda$ $f_{\mathrm{bias}} = \boldsymbol{\tau}_{[q,u]}$ $f_{\mathrm{inv}} = f_{\mathrm{a}} + f_{\mathrm{c}} + f_{\mathrm{bias}}$	Force	void calcMa(State, Vector a, Vector& f)	Inverse dynamics. Can use as residual (implicit) form of equations : $f_{\mathrm{inv}}\underset{[q,u]}{}(\dot{u},\lambda) - f_{\mathrm{applied}}\underset{[t,q,u]}{} = 0$
$a = \mathbf{M}_{[q]}^{-1}f$ $\dot{u}_{\mathrm{tree}} = \mathbf{M}_{[q]}^{-1}(f - f_{\mathrm{bias}}\underset{[q,u]}{})$ $\dot{u}_{\mathrm{loop}} = \dot{u}_{\mathrm{tree}} - \dot{u}_{\mathrm{cons}}$ $\dot{u}_{\mathrm{cons}} = \mathbf{M}^{-1}\mathbf{G}^{\mathsf{T}}\lambda(\varepsilon_{\mathrm{a}}(\dot{u}_{\mathrm{tree}}))$	Force	void calcMInverseF(State, Vector f, Vector& a) void calcTreeUdot(State, Vector f, Vector& udot)	Forward dynamics.
$\varepsilon_{\mathrm{a}} = \mathbf{G}_{[q]}a + \mathbf{b}_{[t,q,u]}$	Force	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.

$\mathcal{E}_v = \begin{bmatrix} \dot{\mathbf{p}} \\ \mathbf{v} \end{bmatrix}_{[t,q,u]}$	Velocity	`const Vector& getVelocityCon-` `straintErr(State)`	Given a set of generalized speeds u, return the velocity constraint errors they entail.
$\mathcal{E}_p = \mathbf{p}_{[t,q]}$	Position	`const Vector& getPositionCon-` `straintErr(State)`	Given a set of generalized coordinates q, return the position constraint errors they entail.
$\lambda = (\mathbf{G}\mathbf{M}^{-1}\mathbf{G}^{\mathsf{T}})^{+}_{[q]}\mathcal{E}_a$	Force	`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 calcTreeEquivalentForc-`
		`es(State, Vector_<SpatialVec>`
		`bodyForces, Vector& joint-`
		`Forces)`
			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.

12.9 Misc

This is material that should probably be removed.

Quaternion normalization	position level only	$\mathbf{n} = 0$	$\mathbf{n}_i(q) = q_i^\mathsf{T} q_i - 1$	
Prescribed motion prescribed coordinates $\overline{q} \subseteq q$ $\overline{u} \subseteq u$	holonomic (index 3) Local to each prescribed mobilizer *i*.	**Given: position**	$\overline{q}_i = \mathbf{q}_i(t)$	
		velocity (index 2)	$\overline{u}_i = \mathbf{u}_{\mathrm{p},i}(t, \overline{q}_i)$	$\mathbf{u}_{\mathrm{p},i}(t, \overline{q}_i) = \mathbf{N}_i^{-1}(\overline{q}_i)\dot{\mathbf{q}}_i(t)$
		acceleration (index 1)	$\dot{\overline{u}}_i = \mathbf{m}_{\mathrm{p},i}(t, \overline{q}_i, \overline{u}_i)$	$\mathbf{m}_{\mathrm{p},i}(t, \overline{q}_i, \overline{u}_i) = \dot{\mathbf{u}}_{\mathrm{p},i}(t, \overline{q}_i)$
	nonholo-nomic (index 2)	**Given: velocity**	$\overline{u} = \mathbf{u}_{\mathrm{v}}(t, q)$	
		acceleration (index 1)	$\dot{\overline{u}} = \mathbf{m}_{\mathrm{v}}(t, q, u)$	$\mathbf{m}_{\mathrm{v}}(t, q, u) = \dot{\mathbf{u}}_{\mathrm{v}}(t, q)$
	acceleration only (index 1)	**Given: acceleration**	$\dot{\overline{u}} = \mathbf{m}_{\mathrm{a}}(t, q, u)$	
General constraints free coordinates $\hat{q} \subseteq q$ $\hat{u} \subseteq u$	holonomic (index 3)	**Given: position** $\mathbf{p}(\hat{q}) = 0$	$\mathbf{p}(t, \overline{q}, \hat{q}) = 0$	
		velocity $\dot{\mathbf{p}}(\hat{u}) = 0$ (index 2)	$\hat{\mathbf{P}}\hat{u} - \mathbf{c}(t, q, \overline{u}) = 0$	$\hat{\mathbf{P}} = \dfrac{\partial \dot{\mathbf{p}}}{\partial \hat{u}} = \dfrac{\partial \mathbf{p}}{\partial \hat{q}}\hat{\mathbf{N}}$ $\overline{\mathbf{P}} = \dfrac{\partial \dot{\mathbf{p}}}{\partial \overline{u}} = \dfrac{\partial \mathbf{p}}{\partial \overline{q}}\overline{\mathbf{N}}$ $\mathbf{c} = -\left(\dfrac{\partial \mathbf{p}}{\partial t} + \overline{\mathbf{P}}\overline{u}\right)$
		acceleration $\ddot{\mathbf{p}}(\hat{u}) = 0$ (index 1)	$\hat{\mathbf{P}}\dot{\hat{u}} - \mathbf{b}_{\mathrm{p}}(t, q, u, \dot{\overline{u}}) = 0$	$\mathbf{b}_{\mathrm{p}} = \dot{\mathbf{c}} - \dot{\hat{\mathbf{P}}}\hat{u}$
	Nonholo-nomic (index 2)	**Given: velocity** $\mathbf{v}(\hat{u}) = 0$	$\mathbf{v}(t, q, \overline{u}, \hat{u}) = 0$	
		acceleration $\dot{\mathbf{v}}(\dot{\hat{u}}) = 0$ (index 1)	$\hat{\mathbf{V}}\dot{\hat{u}} - \mathbf{b}_{\mathrm{v}}(t, q, u, \dot{\overline{u}}) = 0$	$\hat{\mathbf{V}} = \dfrac{\partial \mathbf{v}}{\partial \hat{u}}$ $\overline{\mathbf{V}} = \dfrac{\partial \mathbf{v}}{\partial \overline{u}}$ $\mathbf{b}_{\mathrm{v}} = -\left(\dfrac{\partial \mathbf{v}}{\partial t} + \dfrac{\partial \mathbf{v}}{\partial q}\mathbf{N}u + \overline{\mathbf{V}}\dot{\overline{u}}\right)$
	acceleration only (index 1)	**Given: acceleration** $\mathbf{a}(\dot{\hat{u}}) = 0$	$\hat{\mathbf{A}}\dot{\hat{u}} - \mathbf{b}_{\mathrm{a}}(t, q, u, \dot{\overline{u}}) = 0$	Note that $\mathbf{a}$ must be linear in $\dot{\hat{u}}$.
All index 1 constraints	collect contributions from all the shaded rows above		$\dot{\overline{u}} = \mathbf{m}(t, q, u)$ $\hat{\mathbf{G}}\dot{\hat{u}} - \mathbf{b}(t, q, u, \dot{\overline{u}}) = 0$	$\hat{\mathbf{G}} = \begin{bmatrix} \hat{\mathbf{P}} \\ \hat{\mathbf{V}} \\ \hat{\mathbf{A}} \end{bmatrix}$ $\mathbf{b} = \begin{bmatrix} \mathbf{b}_{\mathrm{p}} \\ \mathbf{b}_{\mathrm{v}} \\ \mathbf{b}_{\mathrm{a}} \end{bmatrix}$ $\mathbf{m} = \begin{bmatrix} \mathbf{m}_{\mathrm{p}} \\ \mathbf{m}_{\mathrm{v}} \\ \mathbf{m}_{\mathrm{a}} \end{bmatrix}$

Table 1: the three classes of constraint equations dealt with by Simbody.

Quaternion normalization	position level only	$\mathbf{n} = 0$	$\mathbf{n}_i(q) = q_i^{\mathsf{T}} q_i - 1$	
Prescribed motion prescribed coordinates $q_{qp} \subseteq q_{up}$ $u_{up} \subseteq u_p$ $\dot{u}_p \subseteq \dot{u}$	holonomic (index 3) Local to each prescribed mobilizer i.	**Given: position**	$q_{qp,i} = \mathbf{q}_i(t)$	
		velocity (index 2)	$u_{qp,i} = \mathbf{u}_{\mathrm{p},i}(t, q_{qp,i})$	$\mathbf{u}_{\mathrm{p},i}(t, q_{qp,i}) = \mathbf{N}_i^{-1}(q_{qp,i})\dot{\mathbf{q}}_i(t)$
		acceleration (index 1)	$\dot{u}_{qp,i} = \mathbf{m}_{\mathrm{p},i}(t, q_{qp,i}, u_{qp,i})$	$\mathbf{m}_{\mathrm{p},i}(t, q_{qp,i}, u_{qp,i}) = \dot{\mathbf{u}}_{\mathrm{p},i}(t, q_{qp,i})$
	nonholonomic (index 2)	**Given: velocity**	$u_{up} = \mathbf{u}_{\mathrm{v}}(t, q)$	
		acceleration (index 1)	$\dot{u}_{up} = \mathbf{m}_{\mathrm{v}}(t, q, u)$	$\mathbf{m}_{\mathrm{v}}(t, q, u) = \dot{\mathbf{u}}_{\mathrm{v}}(t, q)$
	acceleration only (index 1)	**Given: acceleration**	$\dot{u}_p = \mathbf{m}_{\mathrm{a}}(t, q, u)$	
General constraints free coordinates $q_{qf} = q - q_{qp}$ $u_{uf} = u - u_{up}$ $\dot{u}_f = \dot{u} - \dot{u}_p$	holonomic (index 3)	**Given: position** $\mathbf{p}(q_{qf}) = 0$	$\mathbf{p}(t, q_{qp}, q_{qf}) = 0$	
		velocity $\dot{\mathbf{p}}(u_{qf}) = 0$ (index 2)	$\mathbf{P}u_{qf} - \mathbf{c}(t, q, u_{qp}) = 0$	$\mathbf{P} = \dfrac{\partial \dot{\mathbf{p}}}{\partial u_{qf}} = \dfrac{\partial \mathbf{p}}{\partial q_{qf}} \mathbf{N}_{qf}$ $\mathbf{P}_p = \dfrac{\partial \dot{\mathbf{p}}}{\partial u_{qp}} = \dfrac{\partial \mathbf{p}}{\partial q_{qp}} \mathbf{N}_{qp}$ $\mathbf{c} = -\left(\dfrac{\partial \mathbf{p}}{\partial t} + \mathbf{P}_p u_{qp} \right)$
		acceleration $\ddot{\mathbf{p}}(\dot{u}_{qf}) = 0$ (index 1)	$\mathbf{P}\dot{u}_{qf} - \mathbf{b}_{\mathrm{p}}(t, q, u, \dot{u}_{qp}) = 0$	$\mathbf{b}_{\mathrm{p}} = \dot{\mathbf{c}} - \dot{\mathbf{P}}u_{qf}$
	Nonholonomic (index 2)	**Given: velocity** $\mathbf{v}(u_{uf}) = 0$	$\mathbf{v}(t, q, u_{up}, u_{uf}) = 0$	
		acceleration $\dot{\mathbf{v}}(\dot{u}_{uf}) = 0$ (index 1)	$\mathbf{V}\dot{u}_{uf} - \mathbf{b}_{\mathrm{v}}(t, q, u, \dot{u}_{up}) = 0$	$\mathbf{V} = \dfrac{\partial \mathbf{v}}{\partial u_{uf}} \qquad \mathbf{V}_p = \dfrac{\partial \mathbf{v}}{\partial u_{up}}$ $\mathbf{b}_{\mathrm{v}} = -\left(\dfrac{\partial \mathbf{v}}{\partial t} + \dfrac{\partial \mathbf{v}}{\partial q} \mathbf{N}u + \mathbf{V}_p \dot{u}_{up} \right)$

148

	acceleration only (index 1)	**Given: acceleration** $\mathbf{a}(\dot{u}_f) = 0$	$\mathbf{A}\dot{u}_f - \mathbf{b}_a(t,q,u,\dot{u}_p)$ $= 0$	Note that $\mathbf{a}$ must be linear in $\dot{u}_f$.
All index 1 constraints	collect contributions from all the shaded rows above		$\dot{u}_p = \mathbf{m}(t,q,u)$ $\mathbf{G}\dot{u}_f - \mathbf{b}(t,q,u,\dot{u}_p)$ $= 0$	$\mathbf{G} = \begin{bmatrix} \mathbf{P} \\ \mathbf{V} \\ \mathbf{A} \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} \mathbf{b}_p \\ \mathbf{b}_v \\ \mathbf{b}_a \end{bmatrix} \quad \mathbf{m} = \begin{bmatrix} \mathbf{m}_p \\ \mathbf{m}_v \\ \mathbf{m}_a \end{bmatrix}$

Table 2: the three classes of constraint equations dealt with by Simbody.

13 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 Schwieters4 and the TAO robotics simulation and control code which was placed into open source and contributed to SimTK by Arachi Corporation.[16] 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,[3] 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.

14 References

[1] Jain, A.; Rodriguez, G. Recursive flexible multibody system dynamics using spatial operators. *J. Guidance, Control, and Dynamics* 15(6):1453-66 (1992).

[2] Featherstone, R. *Rigid Body Dynamics Algorithms*, Springer (2008).

[3] Jain, A.; Vaidehi, N.; Rodriguez, G. A fast recursive algorithm for molecular dynamics simulation. *J. Comput. Phys.* 106(2):258-268 (1993).

[4] 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).

[5] Jain, A.; Rodriguez, G. Recursive dynamics algorithm for multibody systems with prescribed motion. *J. Guidance, Control, and Dynamics* 16(5):830-837 (1993).

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

[7] 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).

[8] 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).

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

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

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

[12] 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).

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

[14] 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).

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

[16] 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.