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

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

$$\mathbf{G}\dot{u} + \mathbf{b} = 0$$

User's Guide

Version 2.0

December, 2009

Website: https://simtk.org/home/simmath

SimTK Simmath™ 2.0

User's Guide

Michael Sherman, Jack Middleton, Peter Eastman

Abstract

We discuss the SimTK toolset for high-performance basic mathematical algorithms. This includes matrix factoring, numerical integration and differentiation, discrete time stepping, constrained and unconstrained optimization, random number generation, and polynomial root finding.

1 Introduction

SimTK Simmath is part of the SimTK Core toolset and provides a powerful set of open source numerical methods for performing basic mathematical algorithms that arise during physically-based simulation, with a particular emphasis on biological structures. Simmath is accessible through a stable API* to programmers who work in a variety of languages. The 2.0 API is

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

object-oriented C++. C and FORTRAN APIs and wrappers for interpretive languages like Java and Python are planned. The full capability of this package will be built up in layers over time; this document covers the current capabilities and discusses future directions.

1.1 Simmath ancestry

Simmath is built on the proverbial shoulders of giants. It inherits code from many public domain sources, and contains custom SimTK code as well. The Simmath 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.

Simmath depends on several other core SimTK tools, including SimTK LAPACK and Simmatrix.

1.2 Document conventions

In order to allow ourselves the pleasure of delivering the occasional opinionated diatribe, while permitting the easily offended reader to avoid them, we 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 our soapbox" symbol to the right.

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

2 Background

This is general material hopefully providing enough background for the rest of the document to make sense.

2.1 What are "numerical methods"?

Numerical methods are techniques for solving mathematical problems which are typically difficult or impossible to solve analytically. The techniques implemented in Simmath can be either iterative or closed techniques which find a solution in a fixed number of steps. All numerical methods are subject to errors produced either by round off or by approximations made in the computations.

2.2 What can you do with Simmath?

Simmath provides tools for solving optimization, numerical integration and differentiation problems as well as common dense linear algebra problems such as solutions to linear systems of equations, linear least squares, eigenvalue and singular value problems. It also provides discrete time stepping, random number generation, and polynomial root finding.

2.3 SimTK software conventions and style

TODO: Binary compatibility, error handling, etc. etc.

In code examples, we use blue for language keywords, red for SimTK-defined symbols, green for comments and black for everything else. Code samples will be in `typewriter font`, with output from running the examples shown in **`bold typewriter font`**.

All SimTK-defined C++ symbols are in the `SimTK` namespace, meaning that a SimTK class like `Optimizer` is really named `SimTK::Optimizer`. You can use the full name or introduce the unadorned one into your software via C++ `using` statements, either "`using namespace SimTK;`" to introduce all SimTK-defined symbols, or more narrowly-targeted statements like "`using SimTK::Optimizer;`" which allows use of only the name "`Optimizer`" without the prefix.

In the (rare) cases where it is not possible to use a namespace, such as with preprocessor macro names, we always start the name with the characters "`SimTK_`", capitalized exactly as shown. We use the same convention for our C interfaces, since there are no namespaces in C. This can make for some long

names, but we feel it is more important not to introduce conflicts with existing code. A user of SimTK code is always free to define some shorter names to use in invoking the longer ones.

2.4 Scalars, vectors and matrices

Simmath makes use of the lower-level SimTK core module Simmatrix which provides support for `SimTK::Vector` and `SimTK::Matrix` classes, and high-performance basic operations on those objects. For a full description of the Simmatrix package, see the SimTK Simmatrix User's Guide [URL]. However, here is a quick description providing enough information to understand the examples in this document.

The SimTK-defined numerical data types appearing in the Simmath APIs are these (dropping the SimTK:: prefix for brevity):

`Real`	A single scalar value represented as a floating point number at default precision (currently always `double`). This is a `typedef`, not a separate class.
`Vector`	A resizable *m*x1 column vector of `Real` elements.
`RowVector`	A resizable 1x*n* row vector of `Real` elements.
`Matrix`	A resizable *m*x*n* matrix of `Real` elements (*m* rows, *n* columns)

All standard arithmetic operators are overloaded so that they work as expected on data of type `Vector` and `Matrix`, with strict attention paid to the need for conformable dimensions. The "~" (tilde) operator is overloaded to indicate the transpose operation, so that for example the expression `~m` means m^T for a `Matrix m`. A column of a `Matrix` has type `Vector` (*m*x1) which is distinct from the type of a row, which is a `RowVector` (1x*n*). The transpose of a `Vector` has type `RowVector` and vice versa.

As in C, the square brackets operator `[]` is used for indexing, and all indexing is 0-based. So `v[`*i*`]` or `r[`*j*`]` selects the i^{th} element of a `Vector` or `RowVector`, and `m[`*i*`]` selects the i^{th} row (a `RowVector`) from a `Matrix`. Unlike C, round

brackets `()` are also available for indexing, with the same effect when applied to `Vector` or `RowVector` but selecting a *column* rather than a *row* when applied to a `Matrix`. That is, `m(`*j*`)` selects the *j*th column of a `Matrix m` (a `Vector`). The operator `m(`*i*`,`*j*`)` selects the *i*-*j*th element of `m`.* All indexing operations produce "lvalues," that is, they can appear on the left hand side of an assignment operator and the assignment will affect the original object.

3 Linear algebra

Linear algebra techniques solve problems involving one or more equations whose coefficients are linear. The set of equations for a linear system is expressed as a SimTK matrix. Many times the most effective technique, both in terms of speed and accuracy is to factor the matrix into one or more matrices which have special properties from which the solution can be easily found. For example the FactorLU class, which is discussed in more detail in the following section, is used to solve the general problem: Ax=b where A is a matrix, b is the right hand side vector and x is a vector of unknowns. To find x, the matrix A is first factored into two matrices L and U. L is a lower triangular matrix with ones on the diagonal and zeros above the diagonal and U is an upper triangular matrix with all zeros below the diagonal. Once L and U have been computed x can be found easily by back substitution of U with the right hand side vector b. The vector x could have also been found by computing the inverse of A and then multiplying it by the right hand side vector b. However this is usually twice as slow as computing L and U and back substituting and is subject to more round-off errors.

---

* You can also select elements with the more "C-like" construct `m[`*i*`][`*j*`]`, but that requires two operations (one to select the row and one to index that row). The `m(`*i*`,`*j*`)` form is substantially more efficient and should be used instead. Better yet, use vector and matrix operators rather than accessing individual elements whenever possible.

3.1 Solving Linear Systems (SimTK::FactorLU)

3.2 Linear Least Squares (SimTK::FactorQTZ)

3.3 Singular Value Decomposition (SimTK::FactorSVD)

3.4 Eigen Values (SimTK::Eigen)

4 Numerical differentiation (SimTK::Differentiator)

Say you are in possession of software which is able to calculate the value of a function $\mathbf{f}(\mathbf{y})$ where in general $\mathbf{f}$ and $\mathbf{y}$ are vectors of lengths n_f and n_y respectively. You would like to obtain the $n_f \times n_y$ Jacobian, or matrix of partial derivatives, $\mathbf{J} = \partial \mathbf{f} / \partial \mathbf{y}$, evaluated at some $\mathbf{y}=\mathbf{y}_0$. By far the best way to obtain $\mathbf{J}$, if you can do it, is to write a new piece of software which calculates $\mathbf{J}$ by analytical differentiation of the equations which underlie the original code for $\mathbf{f}$. This yields $\mathbf{J}$ calculated to machine precision in the shortest possible amount of CPU time. An alternative is automatic differentiation (e.g. ADIFOR or complex step derivatives), which operates on $\mathbf{f}$'s source code to produce source for $\mathbf{J}$. These also give full machine precision although are typically much more expensive computationally than a hand-derived computation.

However, analytic or automatic differentiation can be difficult in practice, and at times only an approximation of $\mathbf{J}$ is needed, so one might hope for a numerical method which could approximate $\mathbf{J}$ at run time given only the code for $\mathbf{f}$. Such methods are called "numerical differentiation" and Simmath provides a `Differentiator` class for performing this operation, using methods which attempt to balance approximation error and roundoff error to provide a good estimate of $\mathbf{J}$ with a reasonable amount of computation.

To use `Differentiator`, one supplies the function to be differentiated by deriving a concrete object from one of `Differentiator`'s abstract function classes. The most general class is `JacobianFunction`, which handles the problem as described above. There are also simpler classes specialized for scalar-valued functions: `GradientFunction` for scalar functions $f(\mathbf{y})$ of

multiple parameters and `ScalarFunction` for ordinary scalar functions $f(y)$ of one scalar variable. These classes are nested within Differentiator, so their actual names are:

```
Differentiator::JacobianFunction
Differentiator::GradientFunction
Differentiator::ScalarFunction
```

These all derive from a common base class `Differentiator::Function`, which does not normally appear directly in user programs, but provides services which are common to all functions.

4.1 Example

Here is a simple example in which the user function is the scalar function $f(x)=\sin(\omega x)$ where ω is some specified constant. We want to evaluate the derivative $f'(x)=\mathrm{d}f/\mathrm{d}x$ at a particular value x. The correct analytical answer is $f'(x)=\omega\cos(\omega x)$, which we can use to check the approximation.

Below is a complete program that uses the `Differentiator` class to compute an approximate derivative of the above function and compare it with the analytical solution. The program's output is shown in bold at the end.

```
#include "simmath/Differentiator.h"
#include <cstdio>
#include <cmath>
#include <exception>
using SimTK::Real;
using SimTK::Differentiator;

// user-written class
class SinOmegaX : public Differentiator::ScalarFunction {
public:
    SinOmegaX(Real omega) : w(omega) { }

    // Must provide this virtual function.
    int f(Real x, Real& fx) const {
        fx = std::sin(w*x);
        return 0; // success
    }
private:
    const Real w;
};

int main () {
try
  { const Real w=3;
    SinOmegaX      sinwx(w);
    Differentiator dsinwx(sinwx);

    const Real x = 1.234;
    Real exact  = w*std::cos(w*x);
    Real approx = dsinwx.calcDerivative(x);

    std::printf("exact =%16.12f\n", exact);
    std::printf("approx=%16.12f err=%.3e\n",
        approx, std::abs((approx-exact)/exact));

    return 0;
  }
catch (std::exception& e)
  { std::printf("FAILED: %s\n", e.what());
    return 1;
  }
}
exact = -2.541115991807
approx= -2.541115573494 err=1.646e-007
```

Next we will look at the `Differentiator` class and its nested class `Differentiator::ScalarFunction`.

TBD

4.2 Accuracy considerations

Accurate calculation of the result **J** requires knowledge of the relative accuracy ε_f which the function **f** is computed. Normally we assume that **f** is being computed exactly to within machine precision. However, if **f** itself is being approximated in some way you can supply an estimate of ε_f in the

`Function` class and the numerical differentiation will proceed taking that into account.

4.3 Available algorithms

Currently there are two available algorithms: *forward difference* and *central difference*. Forward difference computes a first-order estimate of **J**, with accuracy of $\varepsilon_J \approx \sqrt{\varepsilon_f}$ at a cost of one evaluation of **f** per variable $y_i \in \mathbf{y}$. Central difference computes a second-order estimate of **J**, with accuracy $\varepsilon_J \approx \varepsilon_f^{\frac{2}{3}}$, at a cost of two evaluations of **f** per variable. In contrast, automatic differentiation using ADIFOR or complex step derivatives provides **J** to full accuracy ε_f, at a cost of about three evaluations of **f** per variable. Analytic differentiation also provides full accuracy, but in some cases can be obtained *much* more efficiently.

To put some numbers on this, if the original **f** is calculated with relative accuracy $\varepsilon_f \approx 10^{-14}$, typical for complicated double precision computations, then the forward difference method would give **J** with relative accuracy no better than $\varepsilon_J \approx 10^{-7}$. Central difference could achieve accuracy $\varepsilon_J \approx 4 \times 10^{-10}$. These represent best-case numbers, and while often the methods will come close to these levels, in practice these simple algorithms can do much worse with no warning. In contrast an analytic or automatic differentiation method will reliably yield $\varepsilon_J \approx \varepsilon_f$.

5 Time stepping and numerical integration (SimTK::TimeStepper, SimTK::Integrator)

Simulation of a biological system often entails solving for the trajectory of that system as it moves through time. We start at a known point of the trajectory (that is, a known time and state) called the *initial conditions*, and evolve the trajectory through time in accordance with a mathematical model. The trajectory is computed as a sequence of steps, where the time and state at the end of each step is used to create the initial conditions for the next step. This procedure is called *time stepping*. *Numerical integration* is a crucial ingredient of time stepping used to advance through "smooth" intervals of the overall trajectory.

Mathematical models for biological systems in general comprise both continuous and discrete equations. Such systems are referred to in the simulation literature as *hybrid systems*. The job of a time stepper is to advance the hybrid system through time, in accordance with its mathematical model, generating output when appropriate. The job of the time stepper's numerical integrator is twofold: (1) to correctly advance the trajectory during intervals in which only the continuous part of the model is changing, and (2) to detect *events* which indicate that the discrete part of the model may need updating. The integrator returns control to the time stepper when either (1) a specified *report time* has been reached, or (2) an event has been detected.

The integrator must achieve a user-specified level of accuracy during the continuous interval, and chooses its internal step sizes accordingly. During a single interval, the integrator may take many internal steps or may take a single internal step which reaches or even passes the report time. In the latter case an accurate, interpolated trajectory point is returned to the time stepper. It is possible for a single internal integrator step to satisfy several user-requested report times; the system's continuity is uninterrupted by reporting.

When an event is detected, the integrator determines the precise time at which that event occurred, completes the current continuous interval and then returns control to the time stepper. The time stepper invokes the hybrid system's *event handler*, which in general will make discontinuous changes to both continuous and discrete variables. Handling events sets up a new set of initial conditions for the next continuous integration interval; however, those conditions will be different than those at the end of the preceding interval. In practice we expect the continuous segments to be long and the events relatively rare, although it is possible to define a system which has no continuous phases at all and simply advances time discrete event to discrete event.

To clarify this process, Figure 1 shows a continuous segment of a time stepping trajectory, bracketed by discontinuous events. Please refer to the figure for the following discussion. Return of control from the integrator to

the time stepper is represented by tall green lines; internal integrator steps are represented by short red lines. *

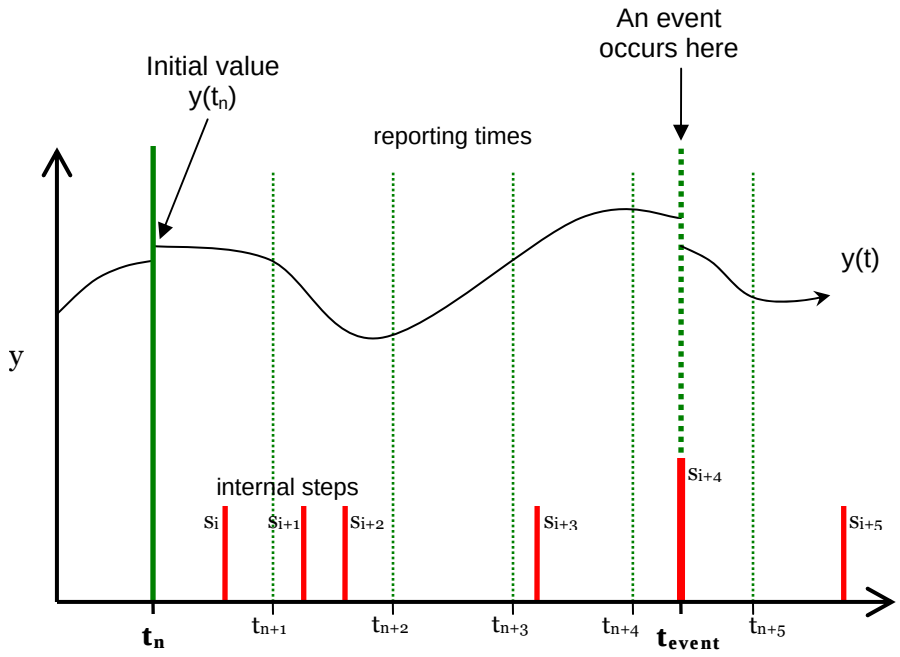

Figure 1: Numerical integration over a continuous segment of a time-stepping trajectory. The continuous region starts at t_n and terminates at t_{event}. Each tall green line represents a return of control to the time stepper: thin dotted ones are caller-requested reporting times; the thick dotted green line is an unrequested return of control just prior to event occurrence and just after event handling.

Starting with the initial condition at t_n (solid green line), the time stepper initiates integration by asking for a report at time t_{n+1}. For accuracy reasons, it takes the integrator two internal steps s_i and s_{i+1} before it reaches t_{n+1}, which it overshoots somewhat, so it returns control to the time stepper at time t_{n+1} with an interpolated value for the state. The integrator's internal context is maintained between calls, however, so when the time stepper asks for the next report time t_{n+2} the integrator is able to start up where it left off (at the end of step s_{i+1}). It then takes two more steps s_{i+2} and s_{i+3} before passing the

* The figure does not show the internal trial steps that an error controlled integrator would have taken and rejected along the way; only the accepted steps are shown.

next report time at t_{n+2}. It returns control with an interpolated state at t_{n+2} and is then able to return again immediately when asked for t_{n+3} without taking another internal step, because step s_{i+3} was long enough to cross two report times.

The time stepper than asks for report time t_{n+4}. However, during the next internal step (s_{i+4}), an event is detected at time t_{event}, terminating that internal step. But before the event can be reported, the integrator must first return control to the time stepper with an interpolated trajectory point at report time t_{n+4}, since that occurred before the event. The time stepper next requests report time of t_{n+5}, but the integrator returns prematurely at t_{event} (actually slightly prior) telling the time stepper that an event is about to occur. The time stepper then invokes the event handler to produce a discontinuously-modified state at t_{event}, which is used to restart the integrator. The next integrator call successfully reaches reporting time t_{n+5}, which is achieved by the integrator's internal step s_{i+5}.

Report times can be seen to have no effect on the evolution of the system.* That is, an integrator will take the same sequence of internal steps regardless of the requested report times during a simulation. Instead, the size of internal steps is determined by accuracy requirements (with more accuracy requiring smaller steps) and the occurrence of events. Internal steps typically involve expensive computation, while the interpolations necessary to satisfy report times are relatively cheap.

Each internal step represents an irreversible advancement of the continuous solution in time. An error-controlled integrator may also take internal trial steps which are ultimately rejected as not being sufficiently accurate. An important consequence of these trial steps is that, from the point of view of the system being integrated, time will be seen to move forward and backward before being irreversibly advanced. That means that the system (which is of course called during trial steps) must not depend on time increasing

* In practice this is not strictly true since an integrator may perform a little "rounding off" of internal steps to avoid having to interpolate over very small slivers of time.

monotonically. Examples of some common behaviors which are *not* permitted are: the setting of flags, updating of neighbor lists, or any other discrete state change not under control of the time stepper.

An integrator maintains only a small, sliding window of trajectory information, just enough to allow interpolation to reporting intervals between internal steps and, for some integrators, recent-past information used to predict the near-future part of the trajectory. Other than that, the past is forgotten except as it is explicitly represented in the current values of state variables. Of course an application program may save the trajectory for as long as desired, but the integrator has little need for past state once time has advanced.

5.1 The continuous system

The continuous part of a SimTK System provides the following equations:

$$\dot{y} = f(d;t,y) \tag{5.1}$$

$$0 = c(d;t,y) \tag{5.2}$$

$$0 = e(d;t,y) \tag{5.3}$$

Here t is time, y contains the continuous state variables, and d contains the discrete state variables which are constant during a continuous interval. $\dot{y} = dy/dt$ is the time derivative of the continuous state variables y. Below, we will usually drop the d from the function arguments for brevity, but it is important to remember that the functions can change behavior between continuous intervals.

During an interval in which d is constant, equation (5.1) is a set of ordinary differential equations with the special property that its exact solution lies on the manifold defined by the set of algebraic equations (5.2). This class of system is called a *differential equation on a manifold*. The above property is equivalent to the statement $c(t,y) = 0 \Rightarrow \dot{c}(t,y,\dot{y}) = 0$ whenever $\dot{y}$ is calculated using equation (5.1); that is, whenever y lies in the manifold then $f(t,y)$ returns $\dot{y}$ tangent to the manifold. Thus perfect integration of equation (5.1) would automatically maintain equation (5.2); it is only because

we cannot integrate (5.1) perfectly that we must deal with the constraint manifold explicitly.

Equation (5.3) is a set of scalar-valued *event trigger* functions designed to change sign precisely when the corresponding event occurs; these will be discussed in the next section.

At the beginning of a continuous interval I of a trajectory we are given initial conditions t_0^I, $y_0^I = y(t_0^I)$, and d^I which satisfy the algebraic constraints, that is,

$$0 = c(d^I; t_0^I, y_0^I) \tag{5.4}$$

We then calculate the starting value for the event triggers:

$$e_0^I = e(d^I; t_0^I, y_0^I) \tag{5.5}$$

Our numerical integration methods then ensure that (5.2) remains satisfied as the trajectory evolves (and d^I stays constant) during the continuous interval I, and the system remains continuous as long as

$$\text{sign}[e(t, y_t)] = \text{sign}[e_0^I] \tag{5.6}$$

for all event triggers, with $t > t_0^I$ and sign[x] a vector whose elements are -1, 0, 1 depending in the obvious way on the signs of the elements of its vector argument x.

5.2 The discrete system

The discrete variables d are updated by the time stepper only at specific times or upon occurrence of specific events detected by the integrator. The occurrence of events is detected using the set of scalar-valued event trigger functions (5.3). The integrator's task is to isolate the precise time at which a sign change is first seen and return control to the caller with the internal state advanced just to that point. That is, the integrator declares that an event has occurred at time t when

$$\text{sign}[e(t, y_t)] \neq \text{sign}[e(t - \varepsilon, y_{t-\varepsilon})] \tag{5.7}$$

for some suitably small interval ε and following the notation described for equation (5.6).

It is likely that not all sign changes are significant for the system being simulated. For example, an event may be triggered only upon a "rising" transition ($\text{sign}[e_i(t)] > \text{sign}[e_i(t-\varepsilon)]$), or "falling" transition ($\text{sign}[e_i(t)] < \text{sign}[e_i(t-\varepsilon)]$), or transition to zero or away from zero, for some event trigger i.

When a triggering transition occurs, the integrator returns control to the time stepper with a status indicating that an event is about to occur (at a time just prior to the event occurrence). The states just prior to and just at the trigger detection are available. But the state at which the event trigger was detected is not immediately suitable for further time stepping, because it is in an inconsistent condition. Instead, an action must be performed (called *handling* the event) which may include discrete (discontinuous) updates to the state. Integration can then be resumed with a modified set of initial conditions for the next step. So the time stepper invokes the system's event handler on the state in the condition where the event(s) have triggered. The handler modifies the state discontinuously, creating a valid state in which the event(s) are no longer triggering, and then the integrator is restarted at the event occurrence time with the "fixed up" state.

Here is a simple example of an event trigger function. Suppose we want to record in a boolean discrete variable d_{flag} whether two points p_1 and p_2 of our system ever came closer than a distance r_0 during a simulation, such that once d_{flag} is true it remains true for the rest of the simulation. The setting of that flag could, for example, affect subsequent forces or visualization, or could simply be reported at the end. Assume the system provides a function r=`dist`$(y,\text{p}_1,\text{p}_2)$ which, given the current state, calculates the distance r between two points. Consider the scalar function

$$e_i(y) = \texttt{dist}(y, \text{p}_1, \text{p}_2) - r_0 \tag{5.8}$$

as a candidate trigger for this event. It passes through zero just as the points reach a distance of exactly r_0 apart. We can provide this function to the integrator, and request that control be returned when it passes through zero. When our event handler is invoked a result of this trigger, it sets d_{flag} to true and then returns control to the integrator.

You may already have noticed several problems with using equation (5.8) as a trigger. It passes through zero both on approach and on separation. It will never pass through zero if the points start out close together and stay that way. It will continue to interrupt control uselessly long after d_{flag} has been set. What we really want is a trigger that will occur only when the points are approaching, and only if d_{flag} hasn't already been set, like this:

$$e_i(y) = \begin{cases} d_{\text{flag}}, & 0 \\ !d_{\text{flag}}, & \texttt{dist}(y, \text{p}_1, \text{p}_2) - r_0 \end{cases} \quad \text{(falling only)} \tag{5.9}$$

Then the remaining problem is how to deal with points that are already less than r_0 apart at the beginning. That is not a transition event and cannot be handled as part of an event trigger. Instead, an initial conditions analysis must be done prior to beginning the simulation to make sure that if the system starts out with the points close together, the initial state will already have d_{flag} set.

The zero crossings of continuous event trigger functions will be isolated quickly; discontinuous event triggers are allowed but have to be localized by "binary chopping" which is more expensive. The next section discusses in detail how we localize events.

5.2.1 Event localization

Events occur instantaneously, at a particular moment in time, say t_{event}. Numerical integrators, on the other hand, advance time in a series of finite-width steps. In general it is prohibitively expensive (not to mention impossible) to find t_{event} exactly. Instead we ask the integrator to *localize* the event to within a small time interval $(t_{\text{low}}, t_{\text{high}}]$, which we call the *event window*. We are certain that the event window contains the actual time of occurrence, with $t_{\text{low}} < t_{\text{event}} \le t_{\text{high}}$. That is, we are sure that the event has *not* occurred by t_{low}, and that it *has* occurred by t_{high}, but we're not sure of the precise value of t_{event}. With well-behaved event trigger functions the integrator can provide a reasonably good guess as to the exact time; otherwise the best guess is that t_{event} is in the middle of the window.

Finite-width localization windows create a likelihood that multiple events will occur within the same window. We cannot say with certainty in what order these events occurred, so for precise treatment they must be considered simultaneous. If an approximate ordering is acceptable then the integrator's t_{event} guesses can be used to order the events within the window, but even those may be identical for some events, and in any case the integrator cannot guarantee that the events actually occurred in the order they appear when sorted by estimated time of occurrence. If more precise information is required, then the localization window must be made narrower, at the cost of increased computation time.

Once an event has been localized to an acceptable tolerance, the integrator's `stepTo()` method will return control to the time stepper at time t_{low}, with a status indicating that the current state is the last one before an event occurs. Generally that marks the end of a continuous interval. The time stepper will next invoke the system's event handler on the state at t_{high} with an indication of which events occurred within the event window. Note that this state at t_{high} is inconsistent in some way; that is, it contains events that have triggered but have not been processed. Thus it is *not* a legitimate point along the system's trajectory and should never be returned to the caller. For discussion we label this "improper" state t_{high}^{-}. The event handler will correct the state discontinuously (potentially updating both discrete and continuous variables), creating a modified state with time still at t_{high} but not triggering any events. We label the modified state t_{high}^{+}. Once the system's handler returns, the state t_{high}^{+} can be output as part of the trajectory and used as the initial condition for the next continuous interval. Thus the time stepper generates the sequence of legitimate trajectory points t_{low}, just prior to event occurrence, immediately followed by t_{high}^{+} which is the time at which the event(s) are defined to have occurred, but after the event handler has modified the state to deal with those events. Consecutive intervals I and I+1 will consist of trajectory points

$$[t_0^I, t_1^I, \ldots, t_n^I, t_{\text{low}}^I](t_{\text{high}}^{-})[t_{\text{high}}^{+} \equiv t_0^{I+1}, t_1^{I+1}, \ldots, t_m^{I+1}, t_{\text{low}}^{I+1}]$$

with round brackets indicating that t_{high}^{-} is not part of the trajectory.

5.2.2 Event handlers

Event handlers are solvers which are able to take a state in an "event(s) triggered" condition, resolve the event(s), set up appropriate event triggers for the next interval, and report back to the time stepper the degree to which the system continuity has been altered, so that the integrator can be reinitialized appropriately.

An event handler can also indicate that the simulation should be terminated, in which case the time stepper will return the final state to its caller and disallow further time stepping.

5.2.3 Other event types

Not all events have to be localized. There are several special case events:

- events which are simply a known function of *t* ("scheduled events")
- "end of step" updates ("time advanced events")
- external events (e.g., clock time, user interrupt)
- termination (e.g. reached final time)

Scheduled events are handled outside the integrator simply by allowing the time stepper to specify with each `stepTo()` call a maximum time to which the integrator may advance. When this time is reached, control is returned to the time stepper. The time stepper can then declare that a scheduled event has occurred, call the system's event handler, and reinitialize the integrator if continuity has been violated.

Time advanced events occur whenever the integrator has advanced time irreversibly, that is, at the end of every successful internal integration step. These are generally restricted to discrete variable updates which do not affect the continuous system, such as min/max values used only for reporting. Normally the integrator does not return control at the end of a step; however the hybrid system can request that if necessary, in which case "end of step" is treated like any other event.

External events are typically handled through the end-of-step event mechanism. For example, the end of step function can look at the clock, keyboard events, etc. and respond accordingly.

Finally, a termination event occurs either when the integrator advances to a designated "final time" for the simulation, or when an event handler indicates that the simulation must be terminated for some other reason.

5.2.4 Event trigger transition details

This section probably covers more than you really want to know about event triggers. But if you run into some mysterious behavior regarding your event trigger functions at some point, you may want to come back to this section.

Normally event triggers are used to detect zero *crossings*. That is, the event in question occurs when the trigger's value goes from negative to positive or vice versa at some moment in time. Zero itself has no special meaning, and in a mathematical world it would never actually be encountered when localizing to a finite-width time window. However, in a computational world landing exactly on zero during localization is not only possible but certain to happen now and again. (See Figure 2 for an illustration.) This leads to the possibility of seeing a -1→0 sign transition while localizing a -1→1. This can cause difficulties since the next step will see a 0→1 transition which could trigger a spurious second event.

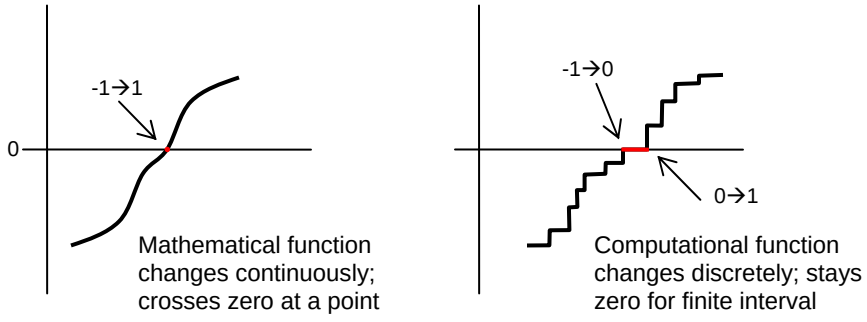

Figure 2: a continuous event trigger function appears discrete in finite precision arithmetic.

An Integrator must take great care not to report two events in the above situation, since there is only a single zero crossing despite appearances.

On the other hand, many event trigger functions are most naturally expressed discretely, often as boolean functions which toggle between "false" and "true" when an event occurs. Others may be designed to have three states such as

"above," "below," and "on the surface." In either case, these triggers will have a significant "zero" state which is not an artifact of finite precision arithmetic. In those circumstances the sign transitions -1→0 and 0→1 can be meaningful and the caller may need to be notified at each transition. Figure 3 shows two examples.

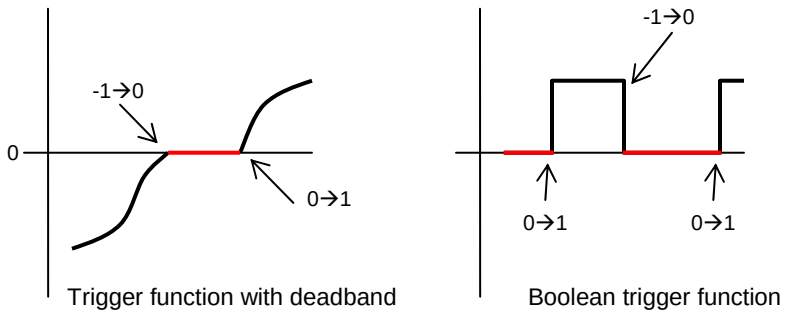

Figure 3: two intentionally-discrete event trigger functions.

To allow the modeler to distinguish between accidental and purposeful zeroes, the System can optionally provide information about each of the event trigger functions. This information can include a specification of which sign transitions are considered significant for each trigger function. If only -1→1 and/or 1→-1 are specified, then we consider the event trigger function to be continuous and the integrator will only report a single event as the trigger passes through zero, even if it accidentally lands exactly on zero during localization. On the other hand, if the System tells us that zero is a significant value, then we assume that the trigger function has a discrete zero "zone" (or "deadband") and we will report transitions to and from zero if they are seen during localization. Table 1 below shows how observed transitions are treated based on whether rising or falling transitions have been specified, and whether zero is significant. Rising and falling transitions are disjoint so these specifications can be combined to allow a single event trigger to detect both rising and falling transitions.

The first column in Table 1 lists the possible specifications that a System can make for the transitions of interest (rising/falling; continuous/discrete). The second column lists the transitions that might be observed initially, *prior* to

localization. Typically, an initial step will be much wider than a localization interval so a -1→1 or 1→-1 transition is most likely to occur. The third column shows for each pre-localization observation the possible post-localization transitions (transitions to and from zero always localize to the same transition). The final column shows what transition will be reported back to the caller as having been seen. Note that although a wider set of transitions may be recognized during localization, the final report must come from the set that the system specified as interesting. For example, if a -1→0 transition is seen after localization, but the system only watches for -1→1, then we'll report -1→0 as an instance of -1→1.

Table 1: handling of sign transitions

Significant sign transitions		Transition seen before localization	Localized transition	Reported transition
Rising -1→1	continuous trigger	-1→1	any rising	-1→1
		-1→0	(unchanged)	-1→1
		0→1	*no event*	
	discrete zero adds -1→0, 0→1	-1→1	any rising	report localized transition
		-1→0	(unchanged)	-1→0
		0→1	(unchanged)	0→1
Falling 1→-1	continuous trigger	1→-1	any falling	1→-1
		1→0	(unchanged)	1→-1
		0→-1	*no event*	
	discrete zero adds 1→0, 0→-1	1→-1	any falling	report localized transition
		1→0	(unchanged)	1→0
		0→-1	(unchanged)	0→-1

Notes: (1) For a continuous event trigger function that accidentally hits zero exactly, the transition *to* zero is treated as a zero crossing, but the subsequent transition *from* zero is ignored. (2) For a discrete-zero event trigger, if the function doesn't stay zero long enough then the zero transitions may be missed and a zero crossing (i.e., -1→1 or 1→-1) will be reported instead.

5.3 Accuracy, scaling, tolerances

The time stepper is capable of achieving different levels of accuracy when simulating a particular system, with more accuracy requiring more computation time. Our goal is to allow application programmers to deliver to

their end users the ability to control the accuracy level with a single, physically meaningful scalar value α, a fraction in the range $0<\alpha\leq 1$. A typical value is α=0.001 (0.1%) meaning that the integrator should produce results which are "within 0.1%" of the "perfect answer." (These phrases are in quotations because they are much easier to say than to define precisely!) Equivalently, one can interpret α as specifying the number of significant digits n_d desired in the results, with $n_d = -\log_{10} \alpha$. For example, $\alpha = 1\text{e-}5 \Rightarrow n_d = 5$.

This goal is achievable only imperfectly, since in general accuracy is difficult to specify, to measure, to control, and even to define precisely. Reasonable users can disagree about exactly what they mean by "1% accuracy." Nevertheless we feel it is important to provide a single "knob" for a user to turn that delivers "more accuracy" at higher cost or "less accuracy" at lower cost, in a way that at least attempts to capture what at typical user might mean by these terms. The alternative of exposing the many complex issues involved to the end user assumes a kind of specialized expertise that would severely limit our intended audience. Instead, we expose these troubles to the application programmer in a way that allows us to collect information that can be used to define the Holy Grail control α.

The primary difficulty we encounter is that the variables and equations defining the user's system are not evenly weighted. We expect that in most cases the programmer who is using SimTK::TimeStepper will know something useful about the various weights that will help us define α. To accommodate that, we allow specification of four kinds of scaling information:

- the characteristic time scale τ
- a weight w_i for each continuous state variable y_i
- a tolerance t_i for each of the constraint errors c_i
- a localization time window width l_i for each event e_i

Each of these quantities is defined below to be a property of the system (model), independent of the accuracy α with which the system is being simulated. The idea is to define for each quantity a "unit error" to which the accuracy requirement can then be applied. The integrator treats these values as constant during each integration step. However, we expect the state

variable weights w_i to change (slowly) as the trajectory (time and state) evolves, so the integrator will request updated weights from time to time during a continuous interval, according to some criteria. Time scale, constraint tolerances, and localization requirements, on the other hand, are expected to be independent of trajectory and are fixed once the instance variables are frozen prior to the start of a continuous interval within the simulation.

5.3.1 Time Scale

There is a characteristic time scale τ of the system which the programmer may provide to convey to the integrator the smallest "time of interest" for the study. This should be the minimum time interval over which "interesting" changes are expected to occur. The integrator may use this for selecting the initial step size and defaults for other scaling information may depend on τ.

5.3.2 Weights and constraint tolerances

We are given a set of weights $w_i \geq 0$ for each y_i and a set of tolerances $t_i > 0$ for each of the constraint errors c_i. Let $\mathbf{W}$ be a diagonal matrix with the w_i's on its diagonal, and $\mathbf{T}$ be a diagonal matrix with the reciprocal tolerances $1/t_i$ on its diagonal (we also call these reciprocal tolerances "constraint weights"). Then given the fractional accuracy specification α (e.g. α=0.1%), the integrator is required to solve for the trajectory $y(t)$ such that each integration step maintains the local error $|\mathbf{W}\cdot\varepsilon_y|_{\mathrm{RMS}} \leq \alpha$ and constraint error $|\mathbf{T}\cdot c(t,y)|_{\mathrm{RMS}} \leq \alpha$ at all times. Here ε_y is the vector of estimated absolute errors in each state variable y, as estimated by the integrator for a trial step under consideration.

5.3.3 Event localization window

Event localization requires for each event trigger function e_i a "unit" localization requirement l_i (in units of the system's time scale τ) which is then narrowed by the accuracy requirement α so that the integrator localizes event e_i to a time interval of width $\varepsilon \leq \alpha\tau l_i$. Thus when the integrator reports that a set E of events has triggered in the event window $(t_{\mathrm{low}}, t_{\mathrm{high}}]$, it guarantees that

$$t_{\mathrm{high}} - t_{\mathrm{low}} \leq \alpha\tau l_i, \quad \forall i \in \mathrm{E}.$$

5.3.4 Default accuracy, scaling and tolerances

The defaults are as follows:

accuracy	$\alpha = 0.001$
time scale	$\tau = 1$
weights	$w_i = 1, \quad 0 \le i < n_y$
constraint tolerances	$t_i = 0.1, \quad 0 \le i < n_c$
localization windows	$l_i = 0.1, \quad 0 \le i < n_e$

We also support a number of different schemes for defining the weights. The conventional "*rtol*, *atol*" scheme is achieved by defining

$$w_i = \frac{1}{\max(|y_i|, u_i)}$$

where u_i represents "one unit" of error in y_i (a typical value would be 0.1). This is then equivalent to $rtol=\alpha$ and $atol_i = \alpha u_i$.

6 Numerical optimization (SimTK::Optimizer)

The `SimTK::Optimizer` class can be used to numerically solve various optimization problems. Optimization methods can be used to find the optimal solution to some problem. For example, what set of coordinates would give the minimum energy configuration for this molecule?

The algorithms implemented in the `Optimizer` class search for points which are a minimum of a, user supplied, objective function. The algorithms start from some initial point and iteratively search for points which reduce the objective function. The algorithms terminate when the objective function stops decreasing. Note that the minimum found by the Optimizers is sometime a local minimum and not a global minimum. Therefore, the point which the algorithm starts searching from is important. This is illustrated in Figure 2 below.

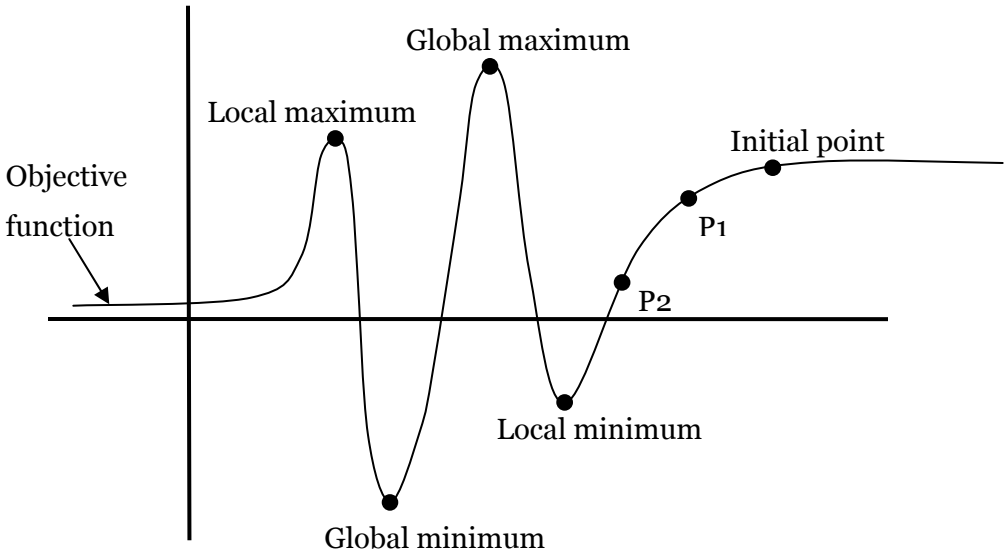

Figure 2: The optimizer begins searching along the objective function starting at the Initial point. It moves first to point P1 then point P2 which reduce the objective function. Eventually the optimizer terminates at the Local minimum when it cannot reduce the objective function any further.

Note that if our goal was to find the maximum of some objective function f(x). We could still use the existing optimizers by using –f(x) as our objective function.

Sometimes the optimization problem has constraints on the objective function.

Consider the simple example below:

The following objective function:

$$(x_1-5)^2+(x_2-1)^2$$

is subject to two constraints:

C1: $x_1 - x_2^2 \geq 0$

C2: $-x_1 + x_2 + 2 \geq 0$

Has an optimal solution at: X = (4, 2)

Figure 3 shows a graph of this problem. The two constraints (C1, C2) bound the space of possible solutions to a feasible region which is shown in gray. The optimal solution to the problem must be a point within this feasible region which has the smallest value of the objective function.

The two dashed circles indicate the constant value contours of our objective function. The smaller circle has a value of 1.0 and the larger circle has a value of 4.0.

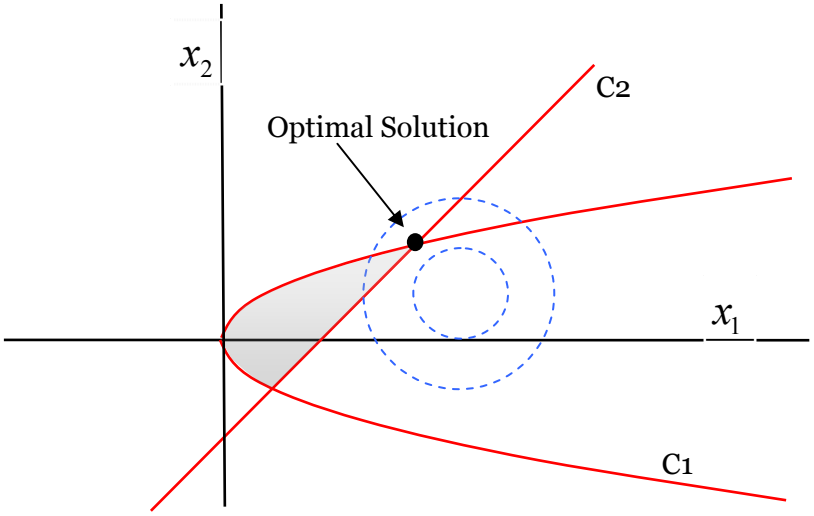

Figure 3: Graph of problem to finding optimal solution to the problem described above. The two constraints are shown in red and the feasible region is show in grey. The contours of the objective function are shown as dashed blue circles.

Optimization problems are solved in SimTK using two classes, the `Optimizer` class, and the `OptimizerSystem` class. The `Optimizer` class allocates an optimizer and sets the options for the optimizer. The `OptimizerSystem` class is used to describe the optimization problem.

`OptmizerSystem` is an abstract class which has methods for computing the objective function, the gradient of the objective function, the constraints, and the constraint Jacobian. The application needs to define a concrete class that subclasses from `OptimizerSystem`. As a minimum the concrete class must

supply an implementation of the `objectiveFunction()` method which computes the objective function for a given set of parameters. An instance of the concrete class is passed as an argument for the constructor of the `Optimizer` class. The Optimizer class uses the information from the `OptimizerSystem` concrete class to select the best optimizer algorithm and to allocate any workspace that the algorithm may need. The application can set various parameters for the optimizer such as convergence tolerance.

Once the Optimizer has been instantiated and the options set the solution can be computed by calling the Optimizer class's `optimize()` method. `Optimizer.optimize()` takes a `SimTK::Vector` which sets the initial point the optimizer will begin searching from. If the optimizer is able to find a solution it will return the parameters for the optimal solution in the `Vector`. `Optimizer.optimize()` will also return the optimal value of the objective function.

The code for using the `Optimizer` class to solve this problem is shown below.

```
#include "Simmath.h"
#include "Optimizer.h"
#include <cstdio>
#include <exception>
using SimTK::Real;
using SimTK::OptimizerSystem;
using SimTK::Optimizer;


// user-written class
class ProblemSystem : public OptimizerSystem {
public:

    // Must provide this virtual function.
   int objectiveFunc(  const Vector &coefficients, const bool
new_coefficients, Real& f ) const {
      const Real *x;
      int i;

      x = &coefficients[0];

      f = (x[0] - 5.0)*(x[0] - 5.0) + (x[1] - 1.0)*(x[1] - 1.0);
      return( 0 );
   }
   int gradientFunc( const Vector &coefficients, const bool
new_coefficients, Vector &gradient ) const{
      const Real *x;

      x = &coefficients[0];

     gradient[0] = 2.0*(x[0] - 5.0);
     gradient[1] = 2.0*(x[1] - 1.0);
```

```
      return(0);


  }
  /*
  ** Method to compute the value of the constraints.
  ** Equality constraints are first followed by the any inequality
constraints
  */
  int constraintFunc( const Vector &coefficients, const bool
new_coefficients, Vector &constraints)  const{
      const Real *x;

      x = &coefficients[0];
      constraints[0] = x[0] - x[1]*x[1];
      constraints[1] = x[1] - x[0] + 2.0;

      return(0);
  }


  /*
  ** Method to compute the Jacobian of the constraints.
  **
  */
  int constraintJacobian( const Vector& coefficients, const bool
new_coefficients, Matrix& jac)  const{
      const Real *x;

      x = &coefficients[0];
      jac(0,0) =  1.0;
      jac(0,1) = -2.0*x[1];
      jac(1,0) = -1.0;
      jac(1,1) =  1.0;


      return(0);
  }


   ProblemSystem( const int numParams, const int numConstraints) :

         OptimizerSystem( numParams, numConstraints ) {}

};
main() {

    Real f;
    int i;

    /* create the system to be optimized */
    ProblemSystem sys(NUMBER_OF_PARAMETERS, NUMBER_OF_CONSTRAINTS );

    Vector results(NUMBER_OF_PARAMETERS);


    /* set initial conditions */
    results[0] = 5.0;
    results[1] = 5.0;

   try {

      Optimizer opt( sys );
```

```
      opt.setConvergenceTolerance( .0000001 );

      /* compute  optimization */
      f = opt.optimize( results );
   }
   catch (const std::exception& e) {
      std::cout << "ConstrainedOptimization.cpp Caught exception:" <<
std::endl;
      std::cout << e.what() << std::endl;
   }


    printf("Optimal Solution: f = %f   parameters = %f %f
\n",f,results[0],results[1]);


}
```

```
Optimal Solution: f = 2.00000 parameters = 3.99998 1.999998
```

Sometimes the optimization problem has constraints on the parameters of the objective function. For example, the potential energy of a protein may be expressed as a function of the bond angles of the protein. Steric clashes may limit the bond angles of a protein to a range of valid values. Therefore minimum energy configuration of the protein would be constrained by these angles. It may be easier to express these constraints as limits on the values of our objective function parameters. This would cause the optimizer to search for a minimum only from within this feasible region.

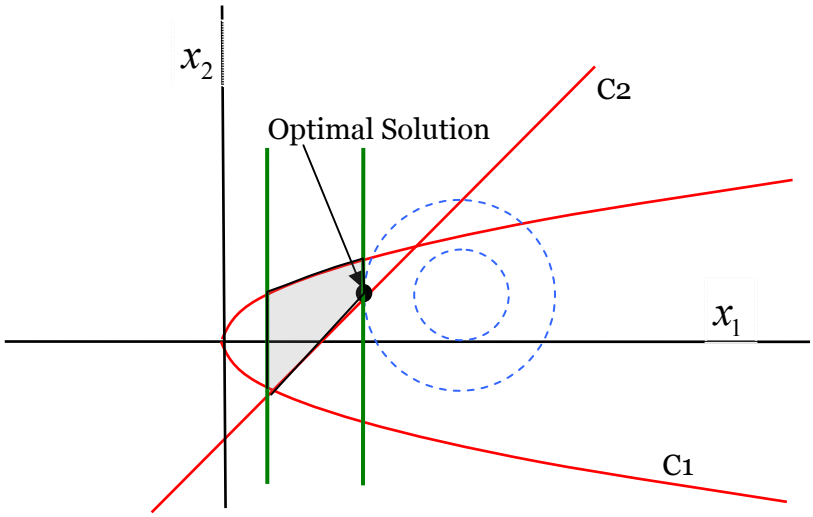

Figure 4: Limits $1.0 \le x_1 \le 3.0$ are shown in green. The two constraints are shown in red and the feasible region is show in grey. The contours of the objective function are shown as dashed blue circles.

For example, if the parameter x_1 in our example had limits of $1.0 \le x_1 \le 3.0$ our problem would look like the graph in Figure 4. These limits could be expressed by adding the following code:

```
Vector lower_limits(NUMBER_OF_PARAMETERS);
Vector upper_limits(NUMBER_OF_PARAMETERS);

/* set limits on the parameters */
   lower_limits[0] =  1.0;
   upper_limits[0] =  3.0;
   lower_limits[1] = -2e19;
   upper_limits[1] =  2e19;
```

```
    sys.setParameterLimits( lower_limits, upper_limits );
```

```
Optimal Solution: f = 4.000000   parameters = 3.000000 1.000073
```

7 Other Mathematical Tools

SimTK provides a number of other tools for solving common mathematical problems. Examples include generating random numbers and finding the roots of polynomials. These tools are described below.

7.1 Random Numbers (SimTK::Random)

There are many cases where it is necessary to generate a set of random numbers, such as to drive a Monte Carlo simulation or to provide random initial conditions for a set of dynamics simulations. Most programming environments provide a random number generator, but they often are poorly suited to scientific applications. If the random number generator is not to bias the results of a simulation, it must have excellent statistical properties in terms of the distribution of values, correlation between successive values, and the length of the sequence it generates.

An algorithm for generating random numbers is more accurately known as a "pseudo-random number generator", because it is deterministic. The sequences of numbers it generates may appear random, but if you reset it to its initial condition (or create a new random number generator instance), it will produce exactly the same sequence of numbers. If you need several random number generators that each produces a different random sequence, you can do this by initializing each one with a different "seed" value. Every possible seed value corresponds to a different sequence of random numbers that (in the case of a good generator) is independent of every other one.

The `SimTK::Random` class is based on the SIMD-oriented Fast Mersenne Twister (SFMT) library. It provides excellent statistical properties, fast performance, and a very long sequence (2^{19937}-1).

Never instantiate `SimTK::Random` directly. Instead, create an instance of one if its two subclasses, `SimTK::Random::Uniform` and `SimTK::Random::Gaussian`. These classes generate numbers according to

uniform and Gaussian distributions, respectively. You can specify the minimum and maximum of a uniform distribution, and the mean and standard deviation of a Gaussian distribution. For example, to generate a sequence of numbers uniformly distributed between 0 and 100, you would write:

```
Random::Uniform random(0.0, 100.0);
// Each time you call getValue(), it will return a different value.
Real nextValue = random.getValue();
```

7.2 Roots of Polynomials (SimTK::PolynomialRootFinder)

This class provides static methods for finding the roots of polynomials. There are specialized methods for quadratic and cubic polynomials, as well as general methods for polynomials of arbitrary degree. In each case, there are methods for polynomials with both real and complex coefficients.

There are two different algorithms used by this class. The specialized methods for quadratic polynomials calculate the roots by explicit evaluation of the quadratic formula. They use the evaluation method described in section 5.6 of "Numerical Recipes in C++, Second Edition", by Press, Teukolsky, Vetterling, and Flannery. In addition, the method for quadratic polynomials with real coefficients performs an extra check to detect when the discriminant is zero to within machine precision. This helps to prevent round-off error from producing a tiny imaginary part in a multiple root.

The methods for cubic and arbitrary degree polynomials use the Jenkins-Traub method, as implemented in the classic RPOLY and CPOLY functions. This is an iterative method that provides rapid convergence and high accuracy in most cases. For details, see

Jenkins, M. A. and Traub, J. F. (1972), Algorithm 419: Zeros of a Complex Polynomial, Comm. ACM, 15, 97-99.

Jenkins, M. A. (1975), Algorithm 493: Zeros of a Real Polynomial, ACM TOMS, 1, 178-189.

As an example of using this class, the following code finds the roots of $x^3-6x^2+11x-6$:

```
Vec4 coefficients(1.0, -6.0, 11.0, -6.0);
Vec<3,Complex> roots;
PolynomialRootFinder::findRoots(coefficients, roots);
cout << "Roots: " << roots << endl;
```

which produces the output

```
Roots: ~[(1,0),(2,0),(3,0)]
```

Notice that the coefficients are specified in order of descending powers. Also notice that the roots are always returned as complex numbers, even if the coefficients are real. This is because a polynomial with real coefficients can still have complex roots.

Acknowledgments

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

References