Release 0.6

June 1, 2006

OpenSim Developer's Guide

The National Center for
Physics-Based Simulation of
Biological Structures
at Stanford

Acknowledgements

OpenSim is a part of SimTK and the Simbios project 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 at http://nihroadmap.nih.gov/bioinformatics.

Authors

Frank C. Anderson, Ph.D.

Ayman Habib, Ph.D.

Peter Loan, M.S.

Scott L. Delp, Ph.D.

Intended Audience

This user's guide is intended for users of OpenSim. OpenSim is an object-oriented modeling framework written in C++ for the simulation, control, and analysis of neuromusculoskeletal systems. User's of OpenSim are likely to range from programmer's and engineers who model the neuromusculoskeletal system to scientists and clinicians who use simulation to investigate the biomechanics of movement. This manual, because it largely details the software engineering aspects of OpenSim, is likely to be of greater interest to programmers and engineers. Although not required, a working knowledge of SIMM from Musculographics, Inc. and familiarity with the C++ programming language is recommended.

Trademarks & Copyright

Notes

OpenSim 0.6 is an alpha release not yet intended for general public use. As such, the documentation and source code for OpenSim is undergoing heavy revision and bug fixing.

During the coming months, a series of releases will be rolled out (0.6, 0.7, 0.8, and 0.9), leading up to a full public release with OpenSim1.0. Until Release 1.0, access to OpenSim source code, libraries, and executables will be restricted to OpenSim project members. As of Release 0.6, there are a total of 20 project member. A substantial number of these members are from institutions outside Stanford University, including the University of Wisconsin-Madison, the University of Texas at Austin, the National Institutes of Health, and the Royal Veterinary College of the University of London. The number of OpenSim project members is expected to grow, and the OpenSim development team will be actively seeking experts in biomechanical simulation to become new members to help test, augment, and refine OpenSim.

While OpenSim source code and downloads are currently restricted to project members, the documentation for OpenSim has been made accessible to the general public. We do this to generate interest in OpenSim, document the growing capabilities of OpenSim, and solicit feedback. Those interested in OpenSim are encouraged to download the documentation and send questions and feedback to the project administrators, Frank C. Anderson and Ayman Habib. The project administrators can be contacted through the OpenSim project on SimTK.org.

If you would like to joint the development team, you may direct inquiries also to the project administrators (see above). Keep in mind that, especially during the early releases of OpenSim, the number of members will be kept small in order to make the testing process more manageable. As OpenSim becomes a more hardened and robust framework, the number of project members will be allowed to grow.

Feedback on this Developer's Guide is welcomed!

Table of Contents

Chapter 1

What is OpenSim?

OpenSim is an object-oriented modeling framework written in C++ for the simulation, control, and analysis of the neuromusculoskeletal system. The neuromusculoskeletal system is modeled in OpenSim using algebraic and ordinary differential equations.

Relation to SimTK

SimTK, the Simulation Toolkit, is part of the Simbios National Center Center for Biomedical Computation funded by the National Institutes of Health. The purpose of SimTK is to enable groundbreaking biomedical research by providing open access to high-quality tools for modeling and simulating biological structures.

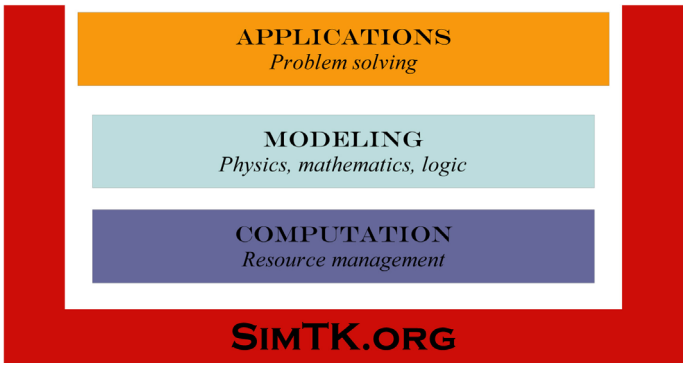

Figure 1.1: Organization of tools on SimTK.org.

The tools on SimTK.org consist of low-level computational tools, modeling frameworks that express the physics, mathematics, and logic of biological structures, and applications that help clinicians, scientists, and engineers solve problems (Fig. 1.1).

OpenSim is a modeling-layer toolset within SimTK (Fig. 1.2). It is build on top of computational tools that include numerical integrators, optimizers, and multibody

dynamics engines such as SD/Fast. Future releases of OpenSim will use the CVODE numerical integrator and the Simbody Multibody Dynamics Toolset, two computational tools available on SimTK.org. The Gait Workflow is an application built on top of OpenSim. It consists of a suite of executables for generating and analyzing subject-specific, muscle-actuated simulations of gait.

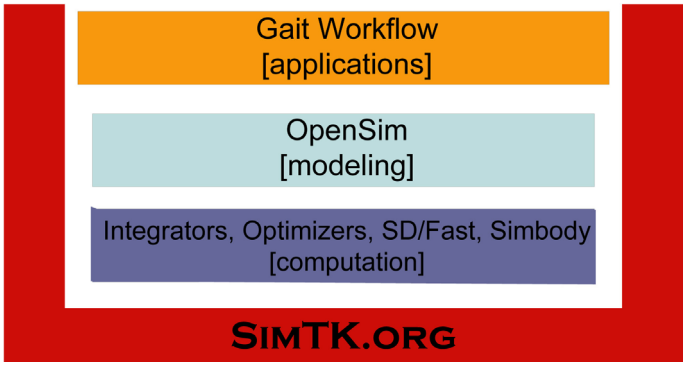

Figure 1.2: Relation of OpenSim to SimTK .

The source code, compiled binaries, and documentation for OpenSim are available on SimTK.org under the project title OpenSim. The source code in OpenSim is freely available and distributable under the MIT License.

Compatibility with SIMM

SIMM (Software for Interactive Musculoskeletal Modeling) from Musculographics, Inc. is a widely-used software application for biomechanical analysis, surgical planning, and ergonomics. The joint (*.jnt) and muscle files (*.msl) used by SIMM to describe models of the musculoskeletal system can be converted into OpenSim models (*.osim) using a conversion program called `simmToOpenSim.exe`, and brought into the OpenSim modeling framework.

OpenSim augments the functionality of SIMM and the SIMM Dynamics Pipeline by providing advanced simulation and control capabilities. In addition, the object-oriented, modular design of OpenSim allows users to extend its functionality and share functionality with other OpenSim users.

Architecture & Design

OpenSim is designed to be modular, extensible, portable, and fast. Functionality is provided mainly in two libraries (Fig. 1.3). The Tools Library provides support classes for mathematics, file input/output, curve fitting, and storage of simulation results. The Simulation Library provides classes for conducting advanced

numerical simulation. These classes include a simulation manager for high-level administration, a fast and robust variable-step numerical integrator, and a set of layered classes that define the functionality of a dynamic model.

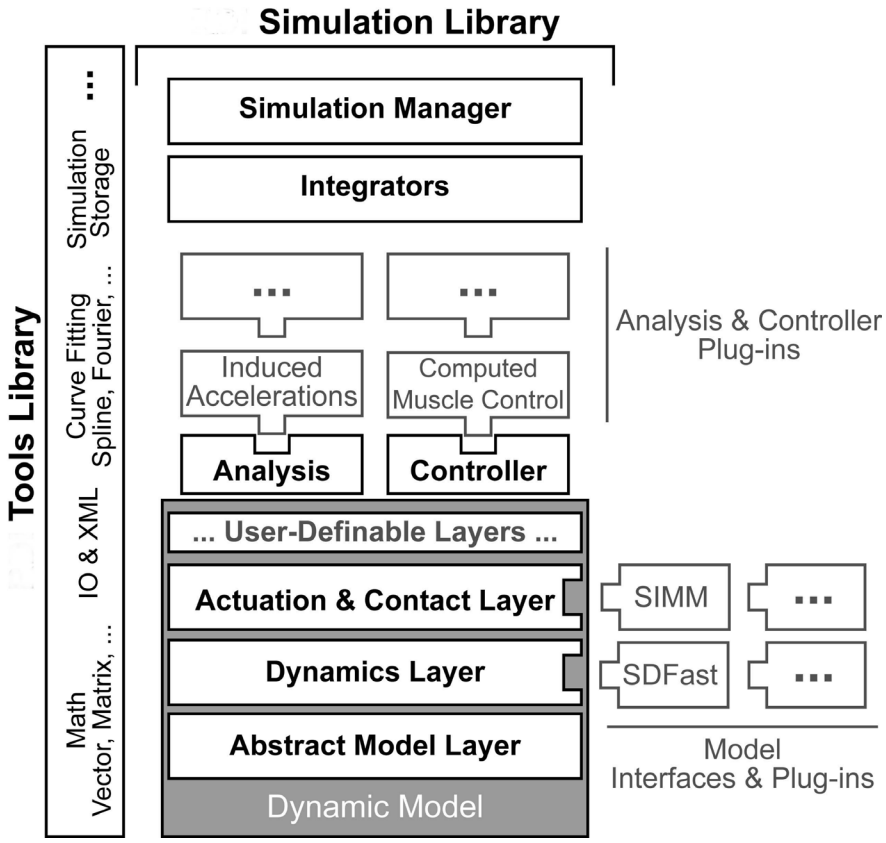

Figure 1.3: Schematic of the OpenSim Architecture.

The base layer of a dynamic model specifies a common abstract interface for all dynamic models. It provides access to a wide range of model information and functionality. The second layer is the dynamics layer. It wraps the equations of motion for a model and, importantly, allows independence from any particular dynamics engine. The dynamics layer currently supports dynamic models generated with SDFast (Parametric Technology Corp.), and there are plans to extend support to the Simbody Multibody Dynamics Toolset. Simbody is a fast, recursive dynamics engine that will eliminate the compile step that is needed by SDFast. The third layer is the actuation and contact layer. It handles application of actuator and contact forces to the model. There are several of basic actuation and contact classes provided in OpenSim. In addition, there is full support for musculoskeletal models with muscles from the SIMM Dynamics Pipeline by Musculographics, Inc.

OpenSim makes extensive use of modern plugin technology through the *virtual* mechanism in C++. Users can develop their own plugin controllers, analyses, muscle models, contact models, and other key neuromusculoskeletal constructs. The framework allows plugins to be shared across users without the need to alter or recompile source code. Currently, about a dozen analysis plugins are available for analyzing simulations.

OpenSim incorporates callback functionality that allows for highly flexible simulation scenarios. For example, callbacks allow perturbation analyses to be conducted or haptic devices to be integrated within a simulation. The modular, object-oriented architecture facilitates compact code writing and helps reduce errors. Use of Extensible Markup Language (XML) for documents makes file IO robust and portable. Finally, because OpenSim is written ialmost exclusively in ANSI C++, it is fast and portable across most computer operating systems including Windows, MacOS X, Linux, and other flavors of Unix.

Applications written on top of OpenSim

OpenSim itself is not a software application, but a modeling framework and set of tools on top of which applications can be built. Release 0.6 of OpenSim does include an application written on top of OpenSim referred to as the ***Gait Workflow***. The Gait Workflow is a suite of executables for generating and investigating subject-specific, muscle-actuated simulations of gait.

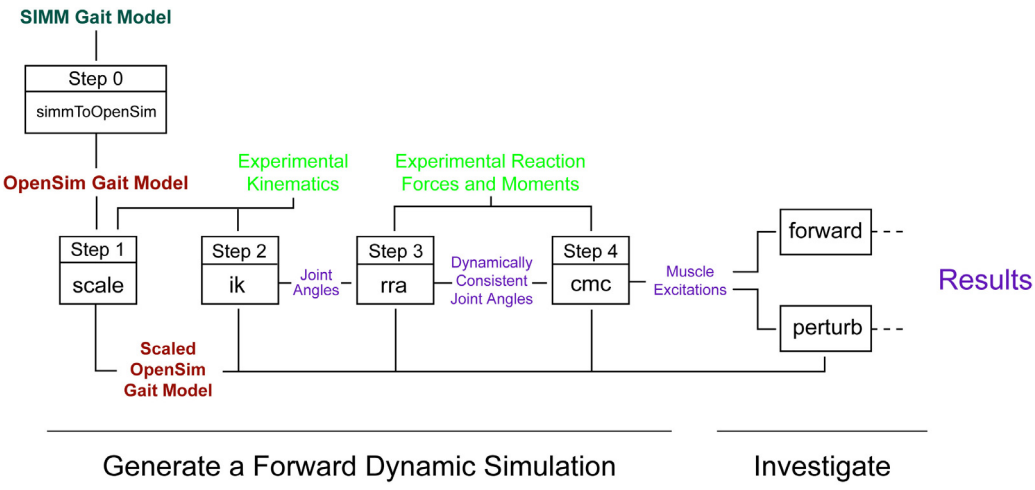

Figure 1.4: Schematic of the Gait Workflow.

The Gait Workflow takes as input experimental gait data (kinematics and ground reaction forces) and a generic musculoskeletal model and generates a forward dynamic simulation that accurately reproduces the input gait pattern. The workflow is comprised of 4 main steps, each of which is carried out by a command-line

executable. If necessary, before beginning the workflow, a SIMM gait model (described by a SIMM joint and muscle file) is converted to an OpenSim gait model by running `simmToOpenSim.exe` (Step 0). In Step 1, the generic gait model is scaled to a particular subject by running `scale.exe`. In Step 2, an inverse-kinematics problem is solved to find the set of joint angles that best reproduce the input kinematics by running `ik.exe`. In Step 3, a residual reduction algorithm is applied to render the joint angles more dynamically consistent with the ground reaction forces by running `rra.exe`. Finally, in Step 4, Computed Muscle Control (CMC) is used to solve for muscle excitations that drive the gait model to track the processed kinematics by running `cmc.exe`. Once a forward dynamic simulation is generated, additional executables can be used to investigate the simulation. The executable `forward.exe` can be used to run analyses on the generated simulation. The executable `perturb.exe` can be used to run perturbations for quantifying muscle function.

Documentation & Downloads

In addition to this User's Guide, other documentation and downloads are available with Release 0.6 on SimTK.org under the OpenSim Project. While the downloads and source code of OpenSim are currently restricted for testing and refinement, the documentation is open to the general public (Table 1.1).

Table 1.1: OpenSim Documentation on SimTK.org.
Documentation may be downloaded and viewed by the general public.

OpenSim_UsersGuide.pdf	This User's Guide. Description of the architecture and use of OpenSim.
OpenSim_ReferenceAPI.pdf	Low-level reference for the application programmer interface (API) of OpenSim. This reference is generated by Doxygen and contains hyperlinked entries for all classes in OpenSim.
GaitWorkflow_UsersGuide.pdf	Schematics describing how to run the executables in the OpenSim Gait Workflow to generate and analyze subject-specific, muscle-actuated simulations of gait.
GaitWorkflow_Tutorials.pdf	Tutorials for the OpenSim Gait Workflow. Tutorials are currently available for the scale and ik steps.

As of Release 0.6, a zipped binary distribution of OpenSim is available to project members (Table 1.2).

Table 1.2: OpenSim Downloads on SimTK.org.
Downloads are currently restricted to project members for a period of testing, debugging, and refinement of OpenSim.

OpenSim_051.zip	A zippled distribution of OpenSim 0.6. The distribution contains the libraries and executables compiled for the Microsoft Windows platform using Visual C++ .Net 2003 (v7.1). This is an alpha release not yet for public use.

Access to **OpenSim Source Code** is currentlyrestricted to project members for a period of testing and debugging. Project members may download the **OpenSim Source Code** from the **SimTK.org** repository using Subversion. Subversion is modern replacement for CVS (Concurrent Versions System). Documentation for Subversion is available at http://svnbook.red-bean.com/. It is software for managing concurrent development of source code by a potentially large number of programmers. For Microsoft Window's users, TortoiseSVN is a free graphical implementation of Subversion that integrates seamlessly into Windows Explorer.

To download the main **OpenSim** development trunk, use the following command:

```
svn checkout --username UserName https://simtk.org/svn/nmbltk/Trunk
```

To compile and link the libraries and executables in OpenSim, a cross-platform build system called CMake is used. OpenSim developers must also download and install CMake.

For further information and instruction on downloading, installing, and building OpenSim, refer to Chapter 7.

Chapter 2

Performing a Simulation

In this chapter, the steps typically involved in performing a forward dynamic simulation using OpenSim are illustrated. A sample main() routine is included for easy reference below. The source code and input xml files for this example are located in the source code repository:

`Trunk/OpenSim/Examples/FallingBlock/fallingBlock.cpp`

`Trunk/OpenSim/Examples/FallingBlock/fallingBlock_actuators.xml`

`Trunk/OpenSim/Examples/FallingBlock/fallingBlock_contacts.xml`

`Trunk/OpenSim/Examples/FallingBlock/fallingBlock_controls.xml`

The code can be compiled using Microsoft Visual C++. See Chapter 7 for more details on compiling in OpenSim.

This chapter is included early in the User's Guide as a brief introduction to some of the features and syntax of OpenSim. In the explanation that follows the sample code, the user is pointed to other chapters in the User's Guide where more detailed information can be found.

Sample Code

```
// fallingBlock.cpp
#include <iostream>
#include <OpenSim/Tools/rdIO.h>
#include <OpenSim/Simulation/Model/rdModel.h>
#include <OpenSim/Simulation/Model/rdActuatorSet.h>
#include <OpenSim/Simulation/Model/rdContactForceSet.h>
#include <OpenSim/Simulation/Model/rdAnalysisSet.h>
#include <OpenSim/Simulation/Control/rdControlLinear.h>
#include <OpenSim/Simulation/Control/rdControlSet.h>
#include <OpenSim/Simulation/Manager/rdManager.h>
#include <OpenSim/Models/Block/rdBlock.h>
#include <OpenSim/Analyses/suActuation.h>
#include <OpenSim/Analyses/suContact.h>
#include <OpenSim/Analyses/suKinematics.h>
```

```
using namespace std;


//_____________________________________________________________________________
/**
 * Run a simulation of a falling block acted on by contact
 * forces and actuators.
 */
void main()
{
   // STEP 1
   // Set output precision
   rdIO::SetPrecision(8);
   rdIO::SetDigitsPad(-1);

   // STEP 2
   // Construct the actuator set, contact set, and control set
   // for the model.
   rdActuatorSet actuatorSet("fallingBlock_actuators.xml");
   rdContactForceSet contactSet("fallingBlock_contacts.xml");
   rdControlSet controlSet("fallingBlock_controls.xml");

   // STEP 3
   // Construct the model and print out some information
   // about the model.
   rdBlock model(&actuatorSet,&contactSet);
   model.printDetailedInfo(cout);

   // STEP 4
   // Alter the initial states if desired.
   int ny = model.getNY();     // Number of states.
   rdArray<double> yi(0.0,ny); // Array of doubles set to 0.0
   model.getInitialStates(&yi[0]); // Get initial states
   yi[1] = 0.25;  // Y Position of block center of mass (com)
   yi[7] = 1.0;   // X Velocity of block com
   yi[9] = 0.0;   // Z Velocity of block com
   model.setInitialStates(&yi[0]); // Set new initial states

   // STEP 5
   // Specify the acceleration due to gravity.
   double g[] = { 0.0, -9.81, 0.0 };
   model.setGravity(g);

   // STEP 6
   // Add analyses to the model.
   int stepInterval = 4;
   // Kinematics
   suKinematics *kin = new suKinematics(&model);
   kin->setStepInterval(stepInterval);
   model.addAnalysis(kin);
   // Actuation
   suActuation *actuation = new suActuation(&model);
   actuation->setStepInterval(stepInterval);
   model.addAnalysis(actuation);
```

```
    // Contact
    suContact *contact = new suContact(&model);
    contact->setStepInterval(stepInterval);
    model.addAnalysis(contact);

    // STEP 7
    // Construct the integrand and the manager.
    rdModelIntegrand *integrand = new rdModelIntegrand(&model);
    integrand->setControlSet(controlSet);
    rdManager manager(integrand);

    // STEP 8
    // Specify the initial and final times of the simulation.
    double ti=0.0,tf=10.0;
    manager.setInitialTime(ti);
    manager.setFinalTime(tf);

    // STEP 9
    // Set up the numerical integrator.
    int maxSteps = 20000;
    rdIntegRKF *integ = manager.getIntegrator();
    integ->setMaximumNumberOfSteps(maxSteps);
    integ->setMaxDT(1.0e-2);
    integ->setTolerance(1.0e-7);
    integ->setFineTolerance(5.0e-9);

    // STEP 10
    // Integrate
    cout<<"\n\nIntegrating from "<<ti<<" to "<<tf<<endl;
    manager.integrate();

    // STEP 11
    // Print the analysis results.
    model.getAnalysisSet()->printResults("fallingBlock","./");
}
```

Explanation

Performing a forward dynamic simulation in OpenSim generally involves a dozen or so steps. Each of these steps is high-level and usually requires only a few lines of code. The following explanation refers directly to the steps labeled in the sample code above.

Include files are organized in directories below the top-level OpenSim directory. This allows the OpenSim libraries and header files to coexist easily with other software distributions.

Step 1. The first step of a simulation is usually to specify the output precision for a simulation. Class rdIO contains several utilities for controlling how data is written to file (e.g., scientific vs. decimal notation, how many digits of precision, etc.). **For more information on class `rdIO`, see Chapter 3.**

Step 2. The actuators and contact forces that apply loads to a model are usually constructed from file. Contact forces are distinguished from actuators in that they are by definition passive. Actuators, on the other hand, can have controls the modulate the loads applied to the model. The controls for a simulation (e.g., the time histories of muscle excitations) are similarly constructed from file. All objects in OpenSim can be written to files written in xml format. **For more information on writing and reading objects to file, see Chapter 3. For more information on actuators, contact forces, and controls, see Chapter 4. To learn how to develop actuators of your own, see Chapter 6.**

Step 3. A model is typically constructed by specifying an actuator set and a contact set. In this example, a model representing a block is constructed. Information about the a model can be obtained by calling the method `printDetailedInfo()`. Models are loaded from dynamically linked libraries (e.g., rdBlock.dll). This is required because SDFast, the dynamics engine currently used by OpenSim, requires the equations of motion for a model to be complied. Once a model is constructed, it can be used in a simulation. Model `rdBlock` is a pre-existing model. **For additional information on models and building models, see Chapter 4.**

Step 4. When a model is constructed, it starts off with a valid set of initial states that can be obtained by calling `model.getInitialStates()`. If desired, the initial states can be altered. In the example above, the initial states are altered manually. In most situations, however, the initial states would be read in from file. **For more information on the methods available on a model, see Chapter 4.**

Step 5. The gravitational acceleration constant can be set for a model. **For more information on the methods available for a model, see Chapter 4.**

Step 6. Analyses gather information during a simulation without altering the simulation. They are run during a simulation by constructing them and adding them to the model. The frequency with which an analysis records information during a simulation can be specified by setting the step interval (ever 4 integration steps in the above example). **To learn more about analyses, see Chapters 4 and 5. To learn how to develop your own analyses, see Chapter 6.**

Step 7. The simulation manager takes care of a variety of low-level initialization necessary for performing a dynamic simulation. It is constructed by specifying the model integrand for the simulation. The model integrand is what is numerically integrated during the simulation. It provided the integrator with the time derivatives of the states. **To learn more about model integrands, see Chapters 4.**

Step 8. The initial and final times for a simulation are specified using the simulation manager. In this case, the values for the initial and final times are specified manually. They are more often specified by examining the time range over which the controls for the simulation are valid. **To learn more about the simulation manager and controls, see Chapters 4.**

Step 9. The numerical integrator included in Release 0.6 is a Runge-Kutta-Feldberg 5-6 variable-step explicit integrator. It has a variety of tunable parameters that control accuracy and step size. **To learn more about the integrator, see Chapters 4.**

Step 10. Once a simulation has been set up, one simply calls the `integrate()` method on the manager to perform the simulation.

Step 11. When a simulation terminates, the results gathered by the analyses are printed to file. The printed results files are text files and can be read by applications such as Microsoft Excell or Matlab for further inspection and plotting. **To learn more about analyses and printing results, see Chapters 4.**

Chapter 3

Tools Library

The OpenSim Tools Library contains tools for input/output, basic math, vector, and matrix operations, curve fitting, and storage of simulation results.

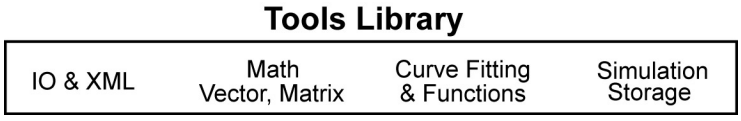

Figure 3.1: Functionality in the OpenSim Tools Library

To gain access to this functionality, link your programs with the library appropriate for your computer platform (Table 3.1). The OpenSim Tools Library is compiled as a dynamically linked library (*.lib and *.dll) on Microsoft Windows and as a shared library (*.so) on Unix and Linux platforms.

Table 3.1: Names of the OpenSim Tools Library on different platforms. Release libraries are optimized for speed. Debug libraries contain symbolic information that allows them to run in a debugger. Debug libraries are suffixed by "`_d`". On Windows systems, debug and release libraries must not be mixed when linking. The Windows libraries are complied and linked using the Microsoft ***multi-threaded dll*** runtime libraries.

Platform	Release	Debug
Windows	`osimTools.lib`, `osimTools.dll`	`osimTools_d.lib`, `osimTools_d.dll`
Linux or Unix	`osimTools.so`	`osimTools_d.so`

In the rest of this Chapter, the functionality available in the Tools library will be introduced. For details concerning any class in the Tools library, consult the OpenSim API Reference.

The Object Class

Most classes in the OpenSim modeling and simulation framework are derived from a base class called Object (Fig. 3.2). Classes derived from class Object inherit its functionality. This functionality includes the ability to get an object's type (i.e., it's class name), set and get an object's name, and write objects to file. In addition, class Object also contains an Observer/Observable mechanism that allows an object to keep track of what other objects are interested in it.

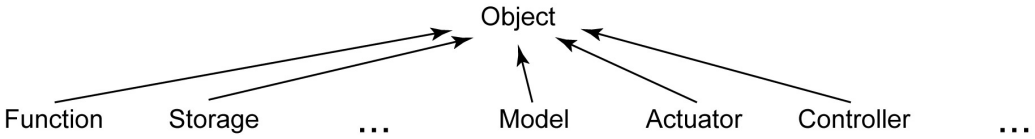

Figure 3.2: Schematic of classes derived from class Object.

Serialization, XML, & Properties

An important aspect of OpenSim is being able to read and write objects to file. Writing to file is referred to as "serialization." Reading from file is referred to as "deserialization." OpenSim uses Extensible Markup Language (XML) to serialize its objects (Fig. 3.3). XML is a widely-used, standardized syntax for storing information. It is text-based, so users can read and edit serialized objects if desired. In addition, it allows for hierarchical representations of objects. This means that an object can contain other objects. In XML, all objects (or elements in XML jargon) begin with an opening tag and end with a closing tag of the same name.

When an object is serialized in OpenSim, the tag name used is the name of the C++ class of that object. In Fig. 3.3, a hypothetical Muscle has been serialized. The corresponding C++ class for Muscle is shown in Fig. 3.4.

```
<!-- Hypothetical muscle serialized to file. -->
<Muscle name="gluteus_maximus">
      <optimal_strength> 3000.0 </optimal_strength>
      <optimal_fiber_length> 0.012 </optimal_fiber_length>
      <tendon_slack_length> 0.030 </tendon_slack_length>
      <WrapSphere name="gmax_wrap1">
            <diameter> 0.500 </diameter>
            ...
      </WrapShere>
      ...
</Muscle>
```

Figure 3.3: OpenSim XML file representation of a class object.

The opening and closing tags are `<Muscle>`, and the name of the muscle is `"gluteus_maximus"`. Variable members of the class are contained within the

opening and closing tags. When the variable members are fundamental data types, such as `int`, `double`, or `string`, there is no restriction on the tag name. An informative tag name is chosen by the programmer. By convention in OpenSim, lowercase words separated by underscores are used. Examples of such variables in the Muscle class include optimal_force, optimal_fiber_length, and tendon_slack_length (Fig. 3.3). When a member variable is another object, its opening and closing tags are required, again, to be that object's class name. For example, the Muscle contains a `<WrapShere>`, which might be a class for modeling how a muscle maps around underlying tissue and bone (Fig. 3.3).

```
class Muscle : public Object
{
      // SERIALIZED MEMBER VARIABLES
      /** Optimal strength of the muscle. */
      PropertyDbl optimalStrengthProp;

      /** Optimal fiber length. */
      PropertyDbl optimalFiberLengthProp;

      /** Tendon slack length. */
      PropertyDbl tendonSlackLengthProp;

      /** Spherical wrap obstacle. */
      PropertyObj wrapSphereProp;

      // NON-SERIALIZED MEMBER VARIABLES
      /** Calculated total muscle length. */
      double lmt;

      // METHODS BELOW HERE ...
}
```

Figure 3.4: Hypothetical C++ Muscle Class.

It is unnecessary for each class in OpenSim to concern itself with the mechanics of serialization. This is taken care of by class Object. By deriving from class Object and implementing several required methods, serialization and deserialization are taken care of automatically for the developer.

The low-level serialization that class Object performs relies on the key member variables of a class being represented by properties. If a member variable is not a property, such as `lmt` in Fig. 3.4, it will not be serialized. A property is a paring of a name and a value, such as `optimal_strength` and `3000.0` as shown in Fig. 3.3. The available property types are shown in Fig. 3.5. They are all derived from a base class called Property. The property classes include types for representing the fundamental data types of `bool`, `int`, `double`, and `string`, as well as arrays of these types. In addition, there is a property type that can be used to hold any object derived from class Object. `PropertyObj` is used in the Muscle class to hold the WrapSphere.

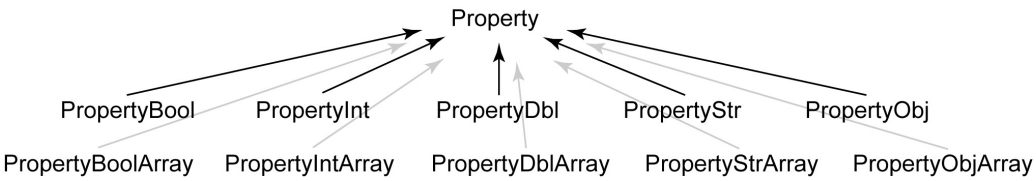

Figure 3.5: Schematic of the OpenSim property classes.

Each property class contains methods for setting and getting its name and value. For more on serialization and properties, see Chapter 6 on extending OpenSim.

Additional Input / Output Functionality

In addition to the low-level serialization capabilities that class Object has, there is another class in OpenSim that provides the developer with some additional control. Class `OpenSim::IO` contains methods for setting the precision of numbers written to file, whether or not numbers are written in scientific notation, and if the numbers are written out with a pad (blank spaces) in front (Table 3.3).

Table 3.2: Some key methods of class OpenSim::IO

Method	Description
`SetScientific(…)`	Sets whether or not scientific notation is used to write numbers.
`SetPrecision(…)`	Sets the number of decimal places written out for numbers to n.
`SetDigitsPad(…)`	Pads (or precedes) a number with n white spaces.

For details concerning class OpenSim::IO, see the OpenSim API Reference.

Math, Vector, & Matrix Operations

Math, vector, and matrix operations are frequently needed when performing simulations. OpenSim contains two classes, `OpenSim::Math` and `OpenSim::Mtx`, that provide basic functionality in this area. Note that these classes, although solid, are not optimized for speed and are only appropriate for

vector an matrix operations involving relatively few elements (i.e., fewer than 1000 for most vector operations and fewer than 10 for most matrix operations). For matrix operations that involve a large number of elements, a cache-optimized multithreaded library like LAPACK should be used. As part of the SimTK core, platform-optimized libraries for LAPACK are available for Linux, Max OS X, and Windows.

Class OpenSim::Math contains a number of useful constants, such as π (3.1415…), as well as a number of methods for simple computational geometry (Table 3.3). Class OpenSim::Mtx contains a number of methods for performing vector and matrix operations (Table 3.4).

Table 3.3: Some key methods of class OpenSim::Math

Method or Constant	Description
`PI`	π computed to the number of significant decimal places supported by the machine precision for a `double` (usually 16).
RTD	Constant for converting radians to degrees. The conversion is accomplished by multiplication.
DTR	Constant for converting degrees to radians. The conversion is accomplished by multiplication.
`IsZero(…)`	Returns true if a number x is so small as to be indistinguishable from zero.
`SigmaUp(…)`	A sigmoidal function that transitions smoothly from 0.0 up to 1.0.
`SigmaDn(…)`	A sigmoidal function that transitions smoothly from 1.0 down to 0.0.
`FitParabola(…)`	Fit a parabola to three points.
`ComputeIntersection(…)`	Compute the intersection of a line with a plane.
`ComputeNormal(…)`	Compte the normal of a triangle.

Table 3.4: Some key methods of class OpenSim::Mtx

Method	Description
`Angle(…)`	Compute the angle between two vectors.
`Normalize(…)`	Normalize a vector.
`Magnitude(…)`	Compute the magnitude of a vector.
`DotProduct(…)`	Compute the dot product of two vectors.
`CrossProduct(…)`	Compute the cross product of two vectors.
`Interpolate(…)`	Interpolate along the line between two points in n-space.
`FitParabola(…)`	Fit a parabola to three points.
`Interpolate(…)`	Interpolate a value along a line.
`Translate(…)`	Translate a point in 3D space.
`Rotate(…)`	Rotate a point about an axis in 3D space.
`Identity(…)`	Assign a matrix to the identity matrix.
`Add(…)`	Add two matrices or vectors.
`Subtract(…)`	Subtract two matrices or vectors.
`Multiply(…)`	Multiply two matrices or vectors.
`Invert(…)`	Invert a matrix.
`Transpose(…)`	Transpose a matrix.
`Print(…)`	Print a matrix.
`FindIndexLess(…)`	Find the first position (index) in an array that is less than a specified value.
`FindIndexGreater(…)`	Find the first position (index) in an array that is greater than a specified value.
`ComputeIndex(…)`	Compute the equivalent single dimension index for a multi-dimensional array.

For details concerning class OpenSim::Math, see the OpenSim API Reference.

Functions & Curve Fitting

It is often desirable to fit a time series of data points with a curve so that the value of the data can be estimated as thought he data were continuous, so that the data can be smoothed, or so that the data can be differentiated. OpenSim currently contains a two classes for just this purpose (Fig. 3.6). Class OpenSim::GCVSpline uses a generalized cross-validated splines to fit smooth data. These spline can be linear, cubic, quintic, or heptic. Class OpenSim::NaturalCubicSpline uses natural cubic splines to fit smooth data.

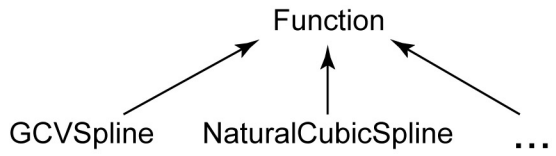

Figure 3.6: Schematic of the OpenSim function classes.

Both of these classes are derived from class OpenSim::Function. OpenSim::Function is used in the argument list of many methods in other classes in OpenSim. By doing so, there is not a constraint on which type of curve representation to use. Rather, one can choose to fit data with any class, as long as the curve fitting class is derived from OpenSim::Function and implements several required methods. At the current time only classes GCVSpline and NaturalCubicSpline are available in OpenSim. In the future, however, more implementations will likely be added that might include polynomial or Fourier series fitting functions.

See Chapter 6 on extending OpenSim for more information on writing your own implementations of an OpenSim::Function's. For details on how to use classes OpenSim::GCVSpline and OpenSim::NaturalCubicSpline, consult the OpenSim API Reference.

Storage

Running simulations can generate vast amounts of data. Class OpenSim::Storage is a utility class that is quite handy for storing information gathered from a simulation. Such data might include, for example, the time history of the joint angles in a model that occurred during a simulation, among many other possibilities.

A storage object is simply a collection of ordered, time-stamped arrays. Like all other OpenSim::Object's has a print() method that can be used to serialize (write) itself to file. In contrast to other object, however, class OpenSim::Storage does

note serialize itself using an XML file format. Instead, it writes itself to file as a text file that contains a table of data. Each row in the file is one of the time-stamped arrays of data. The first column is assumed to be time, although this is not required, and the other columns are whatever data has been added to the storage object. So, if the time histories of joint angles have been stored, the first column might be *ankle_angle*, the second *knee_angle*, and so on.

Chapter 4

Simulation Library

In this chapter, …

Overview

The OpenSim Simulation Library is comprised of…

The Model Class

Class Model is the central class of the Simulation Library. It defines the functionality that a model is expected to provides…

A model is an articulated linkage of rigid bodies….

States, Pseudostates, & Controls

There are three fundamental variables types involved in a simulation: states (y), pseudostates (yp), and controls (x).

Control Representations

Simulation controls may range from the voltage on a torque motor to neural excitations sent to muscles. OpenSim requires a continuous representation for each control in a simulation. However, the specific control representation is flexible.

Actuators & Contact Forces

The rigid bodies in a model are acted on by actuators and contact forces. Constants, step functions, linear interpolation of control nodes.

Integrators & Integrands

The rigid bodies in a model are acted on by actuators and contact forces.

Constants, step functions, linear interpolation of control nodes.

Simulation Manager

The rigid bodies in a model are acted on by actuators and contact forces.

Constants, step functions, linear interpolation of control nodes.

Analyses & Investigations

The rigid bodies in a model are acted on by actuators and contact forces.

Constants, step functions, linear interpolation of control nodes.

Chapter 5

Additional Libraries

In this chapter, …

Analysis

The OpenSim Analysis Library contains about a dozen libraries that gather information from a forward dynamic simulation. Most of the analyses in this library were developed and authored by graduate students at Stanford University.

Actuators

Modeling actuators is one of the critical tasks of the biomechanical researcher.

Optimization – SQP

Optimization is a frequently used tool for solving problems in biomechanics.

Chapter 6

Extending OpenSim

In this chapter, …

Inheritance, Plugins, & Libraries

Modern object-oriented programming languages like C++ enable software to be extended.

Models

It will be common practice for users to build and compile their own models. Because dynamic models are build using SD/Fast, models must be compiled and build as dynamically-linked libraries (dll).

Analyses

There are three fundamental variables types involved in a simulation: states (y), pseudostates (yp), and controls (x).

Investigations

Simulation controls may range from the voltage on a torque motor to neural excitations sent to muscles. OpenSim requires a continuous representation for each control in a simulation. However, the specific control representation is flexible.

Actuators & Contact Forces

Researchers will need to be able to develop and refine their own models of muscles, tendons, and ligaments.

Controllers

One of the more challenging problems in biomechanics is generating forward dynamic simulations that accurately replicate a particular activity like jumping, walking, or running. Advances are occurring at a rapid pace in controller technology. Examples are Computed Muscle Control (Thelen & Anderson, 2006) and Sliding Mode Control (ref Neptune et al., 20??).

Extending OpenSim

Chapter 7

Installation & Developer Setup

In this chapter, we will explain the steps necessary to get yourself going as an OpenSim user. We begin by describing the different uses of OpenSim and how to get started in each case.

User Groups

There are multiple user groups for OpenSim, based on their different needs. One group does not have to develop code based on the OpenSim platform but rather runs OpenSim and experiment with the different executables included in the OpenSim binary distribution. These users should go directly to the "Downloading & Installing OpenSim section".

Another user group consists of users who are more programming oriented and as such will mostly check out the source code, compile it into libraries/executables and add their own classes, methods and or models. For this group we recommend proceeding with the "downloading the source code with subversion" section below that contains instructions on how to check out the source code (OpenSim is an open source platform) and modify it to fit their needs. In addition to allowing users to run the OpenSim Gait Workflow, this allows users to run their own custom simulations, write their own investigations, and extend OpenSim by writing plugins for analyses, actuators, and controllers.

The third user-group consists of researchers somewhere in the spectrum between the first two in terms of their C++ programming skills, but still need to create their own models for different subjects to derive their research. The needs of this group are addressed under the "Toolkit-user installation" section.

Downloading & Installing OpenSim

The OpenSim distribution consists of header files, libraries, and executables that can be downloaded from SimTK.org. To run the OpenSim executables, follow the following steps:

1. Download the binary distribution from SimTK.org at URL (https://simtk.org/project/xml/downloads.xml?group_id=15). You should get one zip file (OpenSim_0501.zip). Only binaries are for DevStudio 7.1 (.Net 2003) on Windows. Support for other platforms will be made available soon.

2. Unzip the distribution to a directory of your choice. Recommended location is (C:/Program Files/ SimTK/OpenSim). We'll call this directory hereafter ***OpenSim_InstallDir***.

3. In order to run the gait workflow executables you need to add the directory ***OpenSim_InstallDir/bin*** to your path environment variable (available at My Computer->Properties->Advanced->Environment Variables->"Path" where (***OpenSim_InstallDir)*** is substituted with the actual directory selected in step 2.

4. Now you're ready to run the executables in the binary distribution of OpenSim. It is recommended that you go through the documentation on simtk.org (under "Documentation Links" on the download page) for a description of the capabilities as well as the limitations of the current version of OpenSim.

Downloading the Source Code Using Subversion

The source code for OpenSim resides in a Subversion repository on SimTK.org. Subversion is a source control tool employed on simtk.org to keep revision history of the code maintained there (Subversion, in short SVN is an open source version control system that offers many advantages over CVS). For the time being the code builds on Windows platform using DevStudio so it is assumed that you have access to a DevStudio installation on the machine you're going to use for development in the OpenSim framework.

To download the OpenSim source code, following these steps:

1. Install the latest version of SVN client. The cygwin distribution comes with a subversion client (svn) but there's a more interactive client for windows that's also available at (http://tortoisesvn.tigris.org/).

2. Choose a directory for the source code.

3. Checkout the source code at URL `https://simtk.org/svn/nmbltk/trunk` into the directory you chose, we'll call it ***OpenSim_SourceDir***..

4. Proceed to the Developer Setup section below for instructions on how to build the source into libraries/executables.

Developer Setup Using CMake

CMake (www.cmake.org) is the tool used to build the OpenSim source into executables and libraries and its use is recommended. To get CMake (also comes as part of the cygwin distribution) go to the web page at (http://www.cmake.org/HTML/Download.html) and download and install the latest version for your platform (2.2 or higher is recommended). Now to build OpenSim libraries and executables:

1. Launch the CMake GUI and browse for the directory ***OpenSim_SourceDir*** when prompted "Where is the source code".

2. You are free to select any directory when prompted "Where to build the binaries". It is recommended that you build the binaries into a directory outside the source tree, we'll call it ***OpenSim_BinDir***

3. CMake generates a native build file to use on your platform. On windows, CMake generates Microsoft DevStudio workspace/solution files based on the configuration you choose. This process is called "Configuration" and it may take more than iteration. Hit the "Configure" button after you select your platform, after a couple iterations the "OK" button on the CMake GUI will become available and when you hit CMake generates the DevStudio solution files.

4. Open the top level solution file (in ***OpenSim_BinDir)*** and build the project ALL_BUILD. This should build all the libraries, utilities and workflow executables in the OpenSim distribution.

5. You may need to modify your Path environment variable (by pre-pending ***OpenSim_BinDir*** /VC71/Debug and/or ***OpenSim_BinDir*** /VC71/Release) to it to make the executables accessible.

Gait Workflow Specific comment

For building individual subject DLLs to use with the gait workflow, a "perl" script has been provided that generates DevStudio solution files. "Perl" comes as part of the cygwin distribution but can also be downloaded and installed from (http://www.perl.org/). You must have a valid license of SD/Fast from Parametric Technology Corporation to generate the equations of motion for individual subjects, and it will also be helpful to have valid license for "SIMM" from Musculographics for visualization purposes.

References

Index