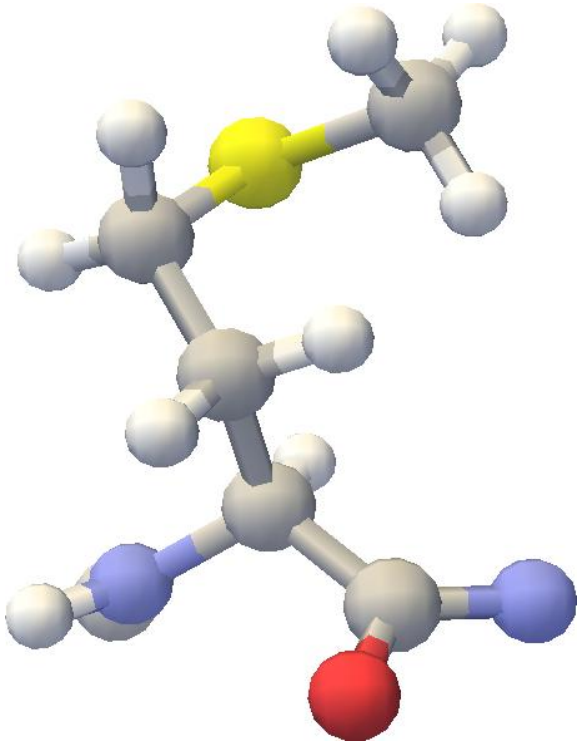
Molmodel



Molmodel User's Guide

Release 2.2

July, 2011

On the web at: https://simtk.org/home/molmodel

Copyright and Permission Notice

Acknowledgments

Molmodel, other SimTK software, and all related activities are funded by the Simbios National Center for Biomedical Computing (http://simbios.stanford.edu) through the National Institutes of Health Roadmap for Medical Research, Grant U54 GM072970. Information on the National Centers can be found at http://nihroadmap.nih.gov/bioinformatics.

Table of Contents

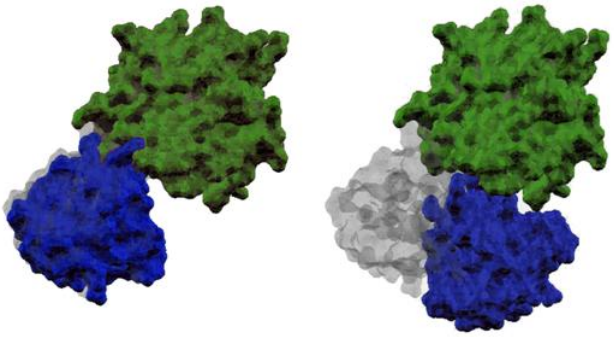
List of Figures

List of Examples

chapter

E

1 Introduction



Molmodel is a programmer's tool for building reduced-coordinate, yet all-atom, models of large biopolymers. You control the allowed mobility. By default, Molmodel builds torsion-coordinate models in which bond stretch and bend angles are rigid while bond torsion angles are mobile. You can rigidify or free any subsets of the atoms, for example building a molecule like the LAO binding protein shown here where the only relevant mobility is the relative motion of whole domains.

Molmodel can produce models with dramatically fewer degrees of freedom than a typical molecular model, yet the reduced set of coordinates is still a fully nonlinear basis for molecular motions of any size. Structural searches and optimizations benefit from a much-reduced search space, Monte Carlo moves can achieve much higher acceptance rates, and dynamics can proceed with much larger step sizes due to the lower natural frequencies produced by larger moving bodies. Because all the atoms are still present, conventional force fields and implicit solvent models can be used for energy and force computations. Alternatively, Molmodel is flexible enough to allow you to design your own force fields. Physics-based, knowledge-based, and special-purpose potentials can be designed and incorporated into your Molmodel model.

While reduced coordinate models have been used successfully for a variety of purposes, research is needed to determine the best way to model a given molecular system for the particular study at hand. Both the physical properties of a molecular system of interest, and the particular investigation being performed will influence the best choice of model. The point of Molmodel is to enable you or users of your software to perform those studies by making it easy to create molecular models with mobility only where you choose to allow it, and then to easily change those choices.

1.1 Scope of this Document

Molmodel is a molecule modeling API extending the Simbody API for simulation of articulated multibody systems. All of the Simbody API is available when using Molmodel and Simbody must be installed in order to use Molmodel. See https://simtk.org/home/simbody for more information. Read the Simbody User's Guide for background, installation instructions, and additional Molmodel examples. You must have Simbody 2.2 installed and functioning on your computer in order to use Molmodel 2.2.

Molmodel's molecular force field can optionally be accelerated using the OpenMM GPU-accelerated force field library as discussed in section 2.3.11. In that case OpenMM must be installed; see https://simtk.org/home/openmm for more information.

Molmodel, Simbody, and OpenMM are components of the open source biosimulation toolkit SimTK, developed and supported by the NIH-funded Center for Physics-Based Simulation of Biological Structures at Stanford (http://simbios.stanford.edu).

This document is not a technical specification, but is rather meant to serve as an introduction to the Molmodel API. The definitive API documentation is available in the installation's doc directory and on the web at https://simtk.org/home/molmodel, Documents page. See section 3.3 for more details on the API documentation.

1.2 Conventions Used in this Document

1.2.1 Warning icon

The icon shown to the left highlights warnings and common pitfalls.

1.2.2 Source code

Computer program source code is shown green, indented, and using a fixed-width font.

```
int example; // this demonstrates the appearance of code
```

Very short fragments of code, class names, and file names will be shown in a **`fixed-width font`**.

All Molmodel (and Simbody) symbols are in the SimTK namespace so must be preceded with "`SimTK::`"; typically we use the C++ `using` statement to avoid that.

1.2.3 New features in examples

Some of the example programs in this document share many features with earlier programs. The most important newer program elements will be highlighted in yellow.

1.3 What is Molmodel?

Molmodel is the SimTK molecular modeling library and API. Molmodel leverages Simbody, the SimTK order-n multibody dynamics library. Molmodel includes methods to construct molecular models for use in simulation with Simbody.

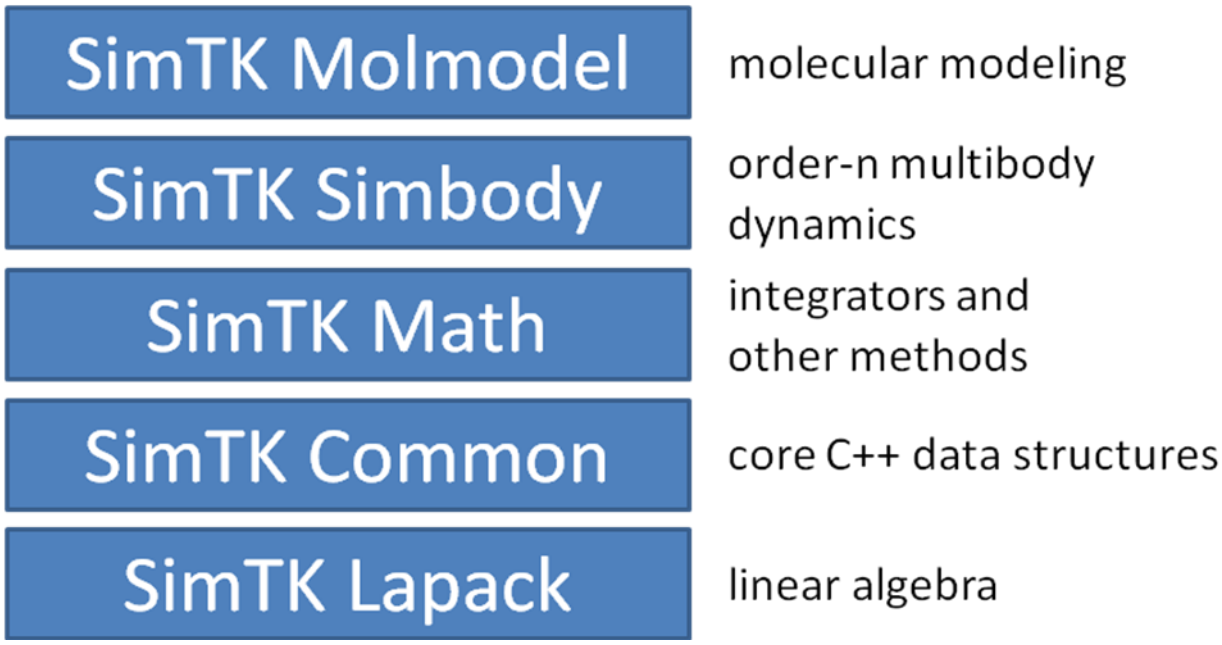


Figure 1.3-1: Principal libraries in SimTK. SimTK also includes the OpenMM package for GPU-accelerated molecular force fields, distributed separately.

Figure 1.3-1 shows some of the key libraries in the SimTK toolkit. Each of the libraries in the figure depends upon the libraries shown below it. For example, the Molmodel library depends upon the Simbody library. Simbody is a general order-n multibody dynamics tool kit. Molmodel, in turn, is an application-area-specific modeling tool kit, capable of creating multibody models of molecules that can be simulated in Simbody. SimTK also includes the OpenMM package for hardware-accelerated molecular mechanics calculations. Molmodel can use OpenMM to accelerate force field computations for the Simbody internal coordinate models it produces. OpenMM is distributed and installed separately. See https://simtk.org/home/openmm for information.

Those interested in the full source code of the Molmodel library can download the source distribution, or find the sources at https://simtk.org/websvn/wsvn/molmodel.

1.4 Prerequisites

1.4.1 Working knowledge of C++ programming language

The SimTK toolkit is written in the C++ programming language. The current target audience is programmers with some familiarity with C++.

1.4.2 Simbody, Molmodel and optionally OpenMM installed

You must have already installed Simbody in order to use Molmodel. Go to the Downloads page of the Simbody project https://simtk.org/home/simbody and follow the installation instructions. Then go to the Downloads page for Molmodel at https://simtk.org/home/molmodel and do the same. You must install Molmodel into the same installation directory as Simbody. If you plan to use OpenMM for force field acceleration you will need to install that separately from https://simtk.org/home/openmm.

When building Molmodel from source, you must already have Simbody and (optionally) OpenMM installed prior to starting the source build. The SimTK_INSTALL_DIR environment variable should have been set to the Simbody installation directory.

See below for a little more detail on installation and building from source.

1.4.3 Read the Simbody User's Guide

Molmodel is built on Simbody. Concepts that are covered in another important document, the Simbody User's Guide, are not be covered in detail in this document.

1.4.3.1 Supported platforms

Please check the Molmodel project at https://simtk.org/home/molmodel (Downloads tab) for the latest news on supported platforms. There will be precompiled binaries available for specific versions of Linux, Mac, and Windows.

1.5 Units are nanometers, atomic mass units, picoseconds, and kilojoules per mole

Simbody requires only consistent units. Molmodel, on the other hand, requires a particular set of units, sometimes called "MD units." Unless otherwise indicated, Molmodel API

methods require and return physical units expressed in MD units of nanometers (length), atomic mass units (mass), picoseconds (time), and kilojoules per mole (energy).

Be careful when including physical quantities in your programs. External data sources often express atom-scale lengths in Angstroms (10^{-10} meter). Molmodel assumes lengths are in nanometers (10^{-9} meter). You must convert quantities accordingly. Similarly, in Molmodel angles are in radians. Thus you must convert any angle values expressed in degrees.

The constants **`SimTK::Deg2Rad`** and **`SimTK::Rad2Deg`** are provided to help with these conversions.

In particular, PDB (Protein Data Bank) format molecular structure files express atomic locations in Angstroms (10^{-10}), not nanometers (10^{-9} m). Be careful when manually transcribing atomic coordinates. The built-in Molmodel methods for reading and writing PDB files already take this conversion into account.

One notable exception to the use of nanometers, atomic mass units, and kilojoules per mole is a set of alternate methods of the class DuMMForceFieldSubsystem, which takes units of Angstroms and kilocalories per mole, indicated with a suffix of "_KA" in the method name. For example, the method defineAtomClass(...) takes arguments expressed in nanometers and kilojoules per mole, while defineAtomClass_KA(...) takes arguments expressed in Angstroms and kilocalories per mole.

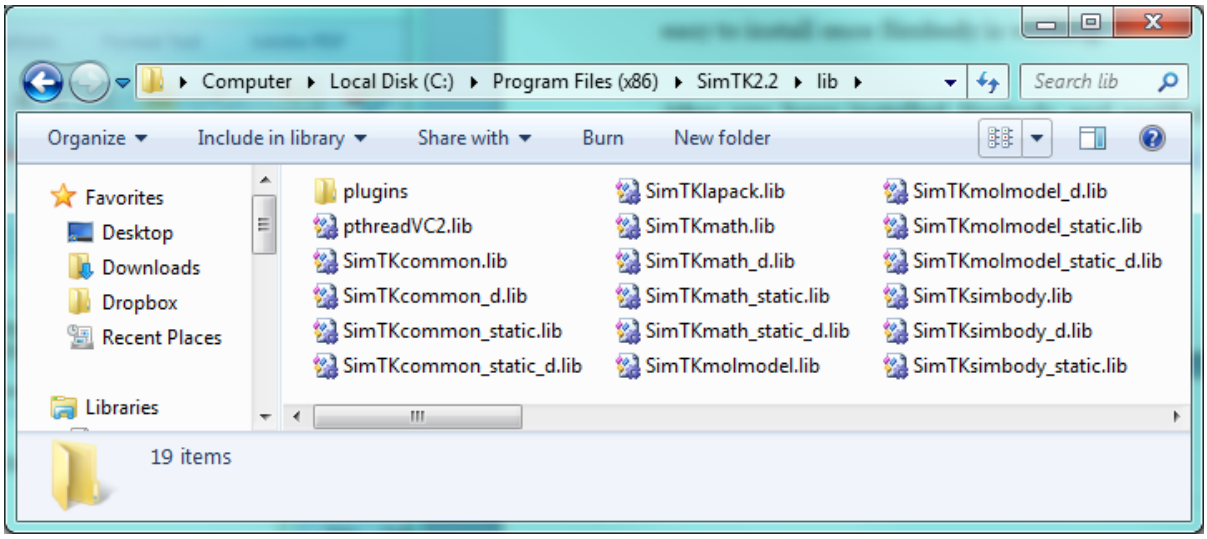
1.6 Installation instructions

Molmodel is an interface built on Simbody. Thus the first step for a Molmodel installation is to install Simbody. To do that, go to the Simbody home page https://simtk.org/home/simbody and follow the instructions there. The Simbody User's Guide provides installation instructions for Simbody and also for Molmodel which is very easy to install once Simbody is working.

After you have installed Simbody and verified that it is working correctly, go to the Molmodel home page https://simtk.org/home/molmodel. There you should select the

binary package that exactly matches the Simbody one you installed. All you need to do is unzip that package and merge it into the *same* installation directory into which you installed Simbody (see below). The path and environment variables that you set for Simbody will also work for Molmodel so you don't need to do any more of that.

Note that Molmodel 2.2 must be installed in the same directory as Simbody 2.2 for full functionality. We will remove that limitation in a future release. For now you should merge the contents of the Molmodel2.2 zip file into the directory where you installed Simbody. That is, you should end up with the installation subdirectories bin,lib,include,doc,examples containing the contents of the corresponding directories from the Molmodel and Simbody zip files. (Example: you'll end up with `include/Simbody.h` and `include/Molmodel.h`.) On Windows you can unzip anywhere and then drag and drop the bin,lib,include,doc,examples folders to the installation directory. You'll be prompted about whether to merge the folders; say yes. On Mac and Linux you can drag and drop or alternately use a command like "cp –rp *molmodelDir*/* *simbodyInstallDir*". Here is what the lib directory should look like on a Windows machine after the files are merged; Mac and Linux will look similar:



Post to the Molmodel forum if you have any trouble with the installation.

1.7 Build from source instructions

To build Molmodel from source, you must first build Simbody from source. The Simbody downloads page has a source zip file, and the documents page has build-from-source instruction. Follow those carefully and make sure you have successfully built and installed Simbody. Then go to the Molmodel downloads page and get the Molmodel source distribution zip file. The build procedure is nearly identical to the Simbody one, except the Molmodel build looks for an installed Simbody version. If it is unable to find Simbody, you must set the SimTK_INSTALL_PREFIX CMake variable to the Simbody installation directory and then select Configure. You may have to do this twice before you can configure without error. Then Generate the build system, making sure to specify the same compiler that you used for Simbody.

After that follow the same procedure as for Simbody to build Molmodel, create the Doxygen documentation and install Molmodel in the same directory as Simbody.

Post to the Molmodel forum if you have any trouble building from source.

1.8 Exercises

Exercise 1.8-1

Install Simbody and Molmodel from simtk.org.

Exercise 1.8-2

Run the test programs that came with Simbody and Molmodel. The test programs are called `SimbodyInstallTest` and `MolmodelInstallTest` and they are in the `examples/bin` and `examples/molmodel/bin` subdirectories, resp., of the installation directory.

chapter

2

2 Getting Started: Simulating a Protein Molecule

Simulating a protein molecule using Molmodel can be simpler than simulating other, simpler, molecule types. That is because the Molmodel API is oriented toward the simulation of protein and RNA molecules. For example, all of the force parameters for proteins are predefined in the AMBER99 force field, and the **`Protein`** class includes a constructor that takes a compact sequence string as an argument. Protein and RNA simulations are intended to be relatively easy. Other molecular simulations are intended to be merely possible, not necessarily easy.

The simulation example in this chapter will generate a protein molecule using a sequence of letters that represents the linear sequence of amino acid residues that comprise the protein we will simulate. The sequence is a very compact representation of a protein's topology.

2.1 Creating a Protein Model from a Sequence String

Example 2.1-1: A simple protein constructor

```
#include "Molmodel.h"
Protein("ACDEFGHIKLMNPQRSTVWY");
```

That string of characters "ACDEFGHIKLMNPQRSTVWY" represents a sequence of twenty different amino acid residues that comprise a protein. In fact, those twenty letters represent all of the canonical amino acid residues that can be represented using the one-letter protein code. For example, "A" stands for alanine, "C" stands for cysteine, etc.

When this Protein constructor is used, a protein is made by default in an "extended" conformation, which results in an elongated structure. One exception to this is the proline residue ("P"), which has a more restricted conformation and results in a kink in the molecule.

2.2 SimpleProtein Program

If you read the Simbody User's Guide (and you should do that first, at least the introductory material), many elements of the following program should be familiar. Some of the varying program elements are highlighted in yellow.

Example 2.2-1: Simulating a very small protein

```
#include "Molmodel.h"
#include <iostream>
#include <exception>
using namespace SimTK;

int main() {
try {
    // molecule-specialized simbody System
    CompoundSystem system;

    SimbodyMatterSubsystem matter(system);
    DecorationSubsystem decorations(system);

    // molecular force field
    DuMMForceFieldSubsystem forceField(system);
    forceField.loadAmber99Parameters();

    Protein protein("SIMTK");
```

```
    protein.assignBiotypes();
    system.adoptCompound(protein);

    // finalize mapping of atoms to bodies
    system.modelCompounds();

    // show me a movie
    system.addEventReporter(
           new Visualizer::Reporter(system, 0.020));

    // Maintain a constant temperature
    system.addEventHandler(new VelocityRescalingThermostat(
              system,  293.15, 0.1));

    // Instantiate simbody model and get default state
    State state = system.realizeTopology();

    // Relax the structure before dynamics run
    LocalEnergyMinimizer::minimizeEnergy(
              system, state, 15.0);

    // Simulate it.
    VerletIntegrator integ(system);
    TimeStepper ts(system, integ);
    ts.initialize(state);
    ts.stepTo(20.0);

    return 0;
}
catch(const std::exception& e) {
    std::cerr << "ERROR: " << e.what() << std::endl;
    return 1;
}
catch(...) {
    std::cerr << "ERROR: An unknown exception was raised"
              << std::endl;
    return 1;
}

}
```

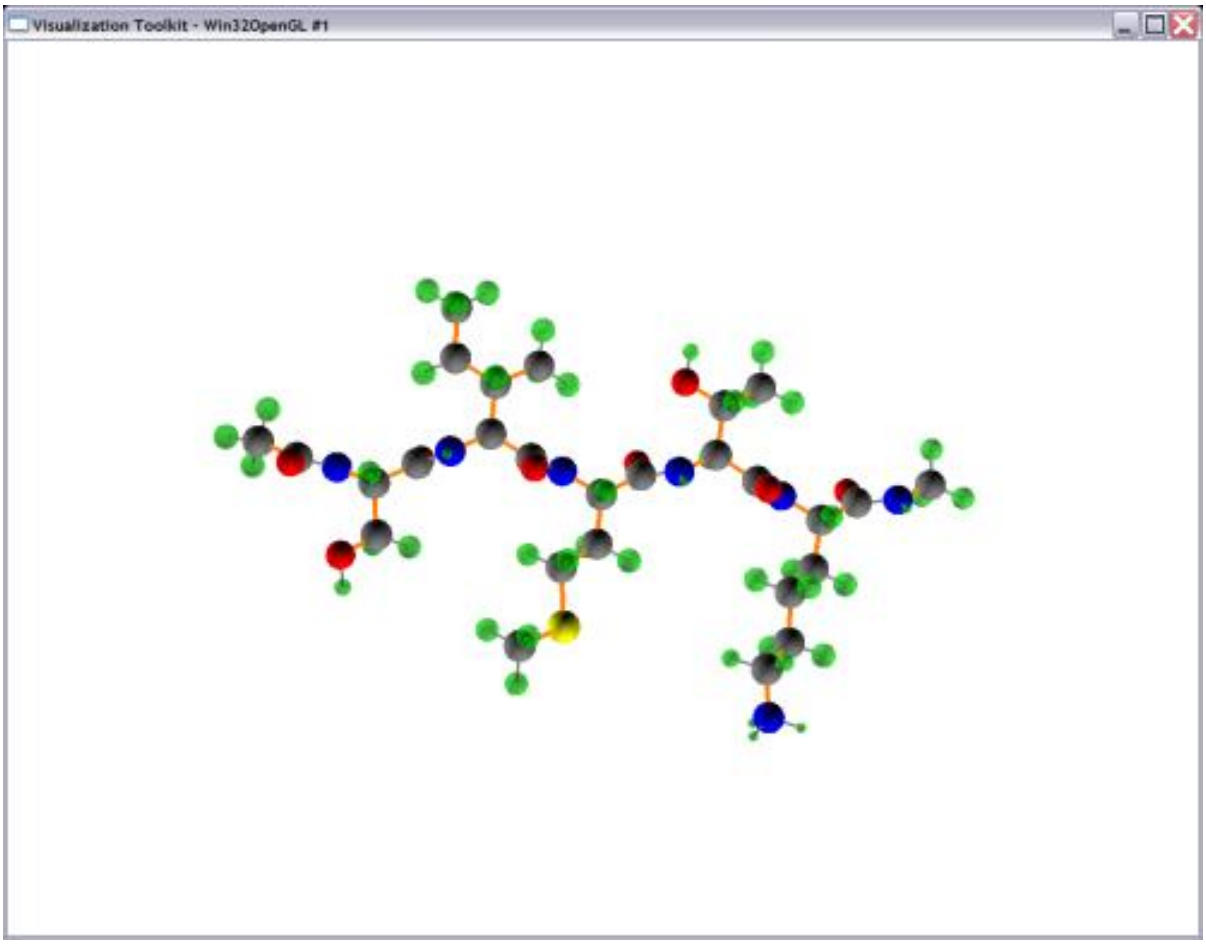


Figure 2.2-1: Frame from Small Protein Simulation

2.3 Analysis of SimpleProtein Program

The above program is available in the `examples/molmodel/src` subdirectory of the installation as `ExampleSimpleProtein.cpp`. If you just want to run it there is an already-compiled binary in `examples/molmodel/bin`.

2.3.1 try/catch block

The main program is wrapped in "**`try`**" and "**`catch`**" statements. These statements are used to handle exceptions that may arise during program execution. If you are unsure what this means, you may want to study exceptions in your **`C++`** language documentation. These are not optional – you must use them in order to get any meaningful error messages out of a Molmodel or Simbody program.

2.3.2 CompoundSystem

CompoundSystem is a specialized Simbody MultibodySystem.

```
// molecule-specialized simbody System
CompoundSystem system;
```

System is a core data type in Simbody simulations. The System concept occurs in almost every example in the Simbody User's Guide. The CompoundSystem class is derived from Simbody MultibodySystem. CompoundSystem includes additional methods and data for dealing with molecular simulations. It also provides a hint to the Simbody Visualizer that showing a ground and sky background would be a bad idea.

2.3.3 Amber99 force field

The Molmodel API provides a functional and flexible, but not well optimized, molecular mechanics force field subsystem called "DuMM".

```
// molecular force field
DuMMForceFieldSubsystem forceField(system);
```

Molecular forces are handled by the DuMMForceFieldSubsystem class, which derives from the Simbody class ForceSubsystem. DuMM can optionally be accelerated with the OpenMM GPU-accelerated force field library; we'll discuss how later.

In this example, we create an empty force subsystem, and explicitly define the few force parameters needed for argon.

We load AMBER99 force field parameters into the force field subsystem with this statement:

```
forceField.loadAmber99Parameters();
```

This ensures that we have available the various force field parameters necessary to simulate all of the atom types commonly found in protein molecules.

The DuMMForceFieldSubsystem class is capable of specifying "Amber-like" force fields. That is, force fields which are expressed in terms of forces including: bond-stretch, bond-

bend, bond-angle, dihedral angle, Lennard-Jones, and coulombic forces. Many popular molecular dynamics force fields are included in this category. Presently Molmodel includes hard-coded parameters for the AMBER99 (J. Wang, 2000) force field, via the loadAmber99Parameters() method of the DuMMForceFieldSubsystem class.

It is possible for Molmodel to load force field parameter definitions from parameter files used in the TINKER (J. W. Ponder, 1987) molecular mechanics program. But this functionality has only been tested for the Amber99 force field; no testing has been done for other TINKER force fields.

DuMM provides very flexible support for custom force fields, and you can selectively control which atoms are involved in force calculations. This has been used for large, coarse grained models employing custom knowledge-based potentials as well as limited physical ones. See the MMB application (formerly RNABuilder) at https://simtk.org/home/rnatoolbox.

2.3.4 Implicit solvent model

The use of an implicit solvent model is not shown in this program, because an implicit solvent model is automatically used unless it is explicitly turned off. This OBC solvent model (Onufriev et al. 2004) simulates some of the effects of water, so that we do not need to include a gazillion water molecules in the simulation, which would defeat the purpose of reduced-coordinate molecular modeling. A method for turning *off* the implicit solvent model will be shown in the argon simulation later (chapter 7). Thus the implicit solvent model is included in this protein simulation. Molmodel does not support the use of explicit solvent for bulk water effects, although you can have localized water molecules that are treated in the same manner as any other small molecule.

2.3.5 Protein(sequence) constructor

The sequence constructor for the **`Protein`** class is used, as shown below:

```
Protein protein("SIMTK");
```

The string of one-letter codes “SIMTK” results in a protein with sequence of five amino acids: serine (S), isoleucine (I), methionine (M), threonine (T), lysine (K, don’t ask). In

addition, small neutral end-cap residues are placed at both ends of the protein chain by default, to create a chemically complete molecule. (You can override that.)

The assignBiotypes() method is then called:

```
protein.assignBiotypes();
```

This method automatically resolves the atom Biotype mapping using residue types and atom names, and matching them to the residue types and atom names defined in the TINKER version of the AMBER99 force field, which we loaded earlier. The concept of Biotypes will be discussed in more detail in the chapters about creating custom molecules.

2.3.6 Attaching the protein to the system

The **`adoptCompound()`** method transfers ownership of the argon atoms' internal data structures to the System.

```
system.adoptCompound(protein);
```

This method is specific to the CompoundSystem class. No modeling decisions have been made yet at this point. A Compound can only belong to one CompoundSystem.

2.3.7 Finalizing the multibody model

The call to modelCompounds() is a critical step:

```
// finalize mapping of atoms to bodies
system.modelCompounds();
```

Modeling decisions are committed at this point. Groups of atoms are placed on Simbody mobilized bodies, with appropriate mobilizers connecting them. Also, the default configuration (the initial configuration of the molecule) is transferred into the System's default configuration (stored in a Simbody **`State`** object). Subsequent changes to the default (initial) configuration will have no effect on your simulation.

In this example, we are relying on a set of default modeling decisions. Those decisions create a model in which bond lengths and bond angles remain fixed, while torsion angles are

permitted to vary. This is known as a "torsion model", or "internal coordinate model." Other modeling choices will be explored in chapter 4.

2.3.8 Maintaining temperature

A method for maintaining the physical system at a constant temperature is included in this example:

```
system.addEventHandler(new
  VelocityRescalingThermostat(
    system, SimTK_BOLTZMANN_CONSTANT_MD, 293.15, 0.1));
```

A Nosé-Hoover thermostat is also available and is a better choice in most cases. See the API documentation for more details.

2.3.9 Relaxing the structure

Before running a dynamic simulation, it is a good idea to relax the structure so that your trajectory does not begin with a high energy configuration. The call to do that is shown below:

```
LocalEnergyMinimizer::minimizeEnergy(system, state, 15.0);
```

See the API documentation for more details.

2.3.10 Simulating

The simulation process is the same as that for non-molecular systems, as described in the Simbody User's Guide. Here we have chosen the Verlet integrator, a popular choice for molecular systems, in which the computation of the forces tends to be vastly more expensive than the computation of the motions. See the API documentation to find what other integrators are available.

2.3.11 Accelerating Molmodel with OpenMM

If you have OpenMM installed and operating on a hardware-accelerated platform (e.g. CUDA) on your computer, you can easily use it to accelerate the protein simulation we did above. Simply insert the lines:

```
forceField.setUseOpenMMAcceleration(true);
```

```
forceField.setTraceOpenMM(true);
```

before the call to realizeTopology(). Only the first line is required; the second is very useful though to verify that OpenMM is actually being used and to debug the problem if not—it sends helpful trace output to stdout (cout). Note that if you did not install SimTK in its standard location, the environment variable SimTK_INSTALL_DIR must be set so that the OpenMM plugin can be located (it is in $(SimTK_INSTALL_DIR)/lib/plugins).

For problems with OpenMM installation, go to https://simtk.org/home/openmm. Note that acceleration requires supported hardware, appropriate drivers, and an OpenMM 3.0 installation.

2.4 Exercises

Exercise 2.4-1

Compile and run the protein example program.

Exercise 2.4-2

Try a different protein sequence.

chapter

3

3 Simulating an RNA Molecule

Simulating RNA molecules is very similar to simulating proteins.

3.1 SimpleRNA Program

RNA and DNA structural topology, like that of proteins, can be represented by a sequence of letters in a one-letter-per-residue code. The size of the alphabet for RNA and DNA is much smaller than that for proteins. RNA has letters A, C, G, U, while DNA has letters A, C, G, T.

This example runs a bit slower than the previous one because atomic coordinates are being written periodically.

The program below is available in the `examples/molmodel/src` subdirectory of the installation as `ExampleSimpleRNA.cpp`. If you just want to run it there is an already-compiled binary in `examples/molmodel/bin`.

Example 3.1-1: Simulating a small RNA molecule.

```
#include "Molmodel.h"
#include <iostream>
```

```
#include <fstream>
#include <exception>

using namespace SimTK;
const char* PdbFileName = "simpleRNA.pdb";

int main() {
try {
    // molecule-specialized simbody System
    CompoundSystem system;
    SimbodyMatterSubsystem matter(system);
    GeneralForceSubsystem forces(system);
    DecorationSubsystem decorations(system);

    // molecular force field
    DuMMForceFieldSubsystem forceField(system);
    forceField.loadAmber99Parameters();

    RNA rna("AUG");
    rna.assignBiotypes();
    system.adoptCompound(rna);

    // Maintain a constant temperature
    NoseHooverThermostat thermo(forces, matter, 293.15);

    system.addEventReporter(
        new Visualizer::Reporter(system, 0.020));

    std::ofstream pdbFile; pdbFile.open(PdbFileName);
    system.addEventReporter(
        new PeriodicPdbWriter(system, pdbFile, 0.100));

    // finalize mapping of atoms to bodies
    system.modelCompounds();

    // Instantiate simbody model and get default state
    State state = system.realizeTopology();

    // Relax the structure before dynamics run
    LocalEnergyMinimizer::minimizeEnergy(system, state, 15.0);

    // Simulate it.
    VerletIntegrator integ(system);
```

```
    TimeStepper ts(system, integ);
    ts.initialize(state);
    ts.stepTo(10.0);

    pdbFile.close();
    std::cout << "\nWrote pdb file "
              << Pathname::getCurrentWorkingDirectory()
              << PdbFileName << "\n";
    return 0;
}
catch(const std::exception& e) {
    std::cerr << "ERROR: " << e.what() << std::endl;
    return 1;
}
catch(...) {
    std::cerr << "ERROR: An unknown exception was raised"
              << std::endl;
    return 1;
}

}
```

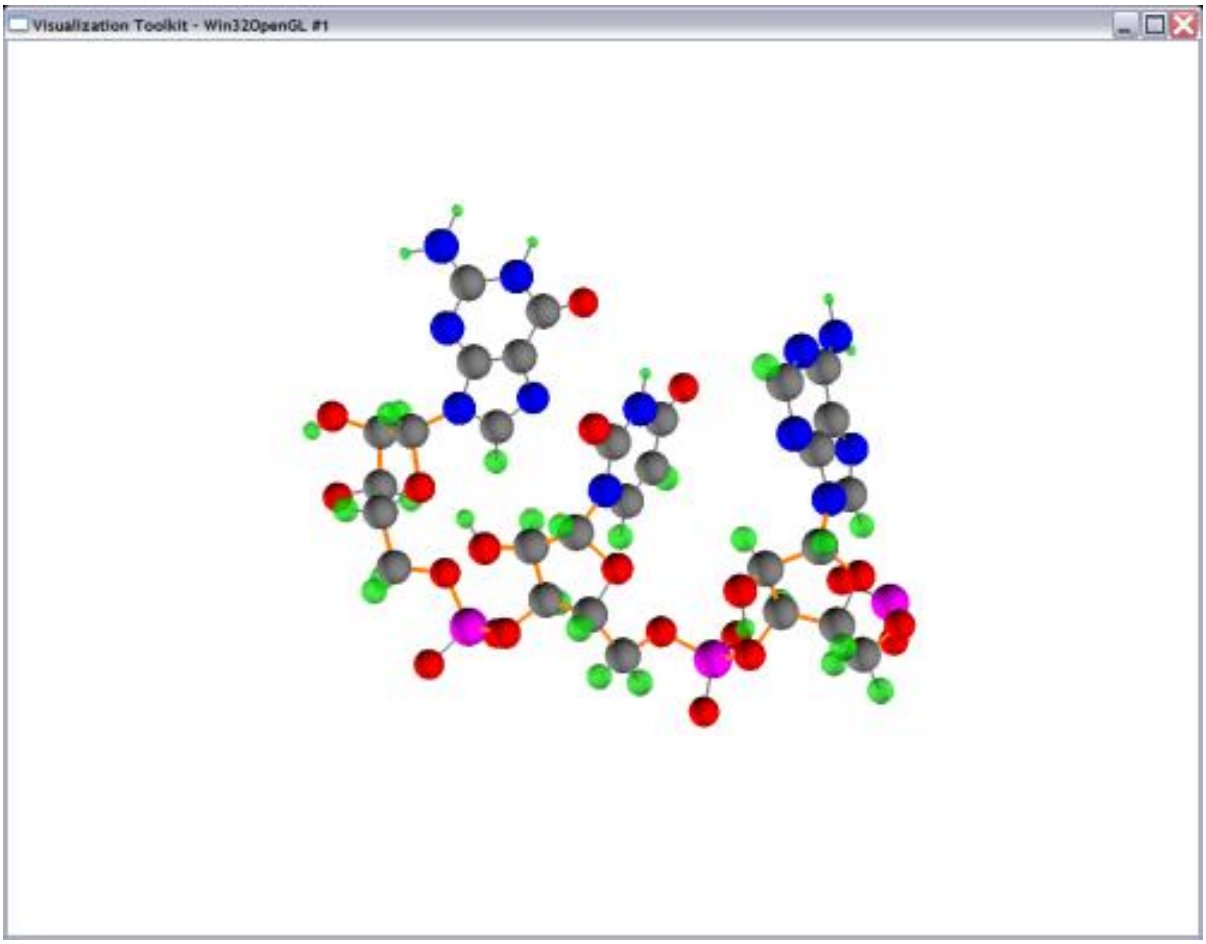


Figure 3.1-1: Small RNA model

3.2 Writing PDB Coordinates

This example writes a set of PDB coordinates to a file periodically. The atomic coordinates in PDB files can be used by many molecular computer programs to share structural data. The example program uses the class **`PeriodicPdbWriter`**, which is derived from **`PeriodicEventReporter`**, which is described in the Simbody User's Guide and the API documentation. **`PeriodicPdbWriter`** uses the **`writePdb()`** method of **`Compound`**. The Compound::writePdb() methods can also be used directly.

If you are curious about how the class **`PeriodicPdbWriter`** is implemented, you can see the full implementation in the header file $(SimTK_INSTALL_DIR)**`/include/molmodel/internal/PeriodicPdbWriter.h`**.

3.3 Finding API Documentation

Using the reference documentation of the Molmodel and Simbody APIs is essential to mastering Molmodel programming. Doxygen-generated HTML API documentation is available in the Molmodel installation in a directory on your computer:

$(SimTK_INSTALL_DIR)**/doc/MolmodelAPI.html**

and the Simbody API material may be accessed through **SimbodyAPI.html**.

The above assumes you set the SimTK_INSTALL_DIR variable to where you put Simbody and Molmodel on your computer; see the Simbody User's Guide installation instructions.

You can also access the API documentation at the Simtk.org website.

1. Browse to the Molmodel project site at http://simtk.org/home/molmodel.
2. Select the "Documents" section on the left navigation bar.
3. Select the "Molmodel 2.2 API Reference" link.

The Simbody 2.2 API Reference link is also available online.

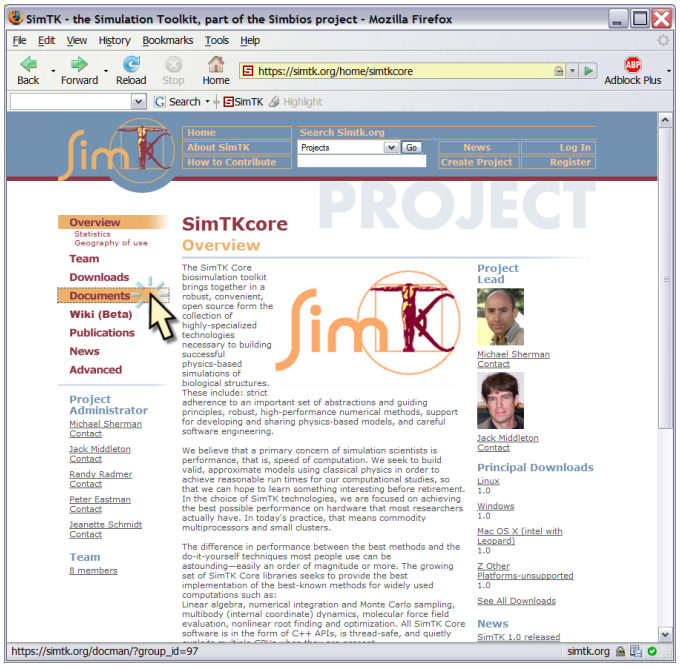


Figure 3.3-1: Selecting the Documents section of the SimTK Core project.

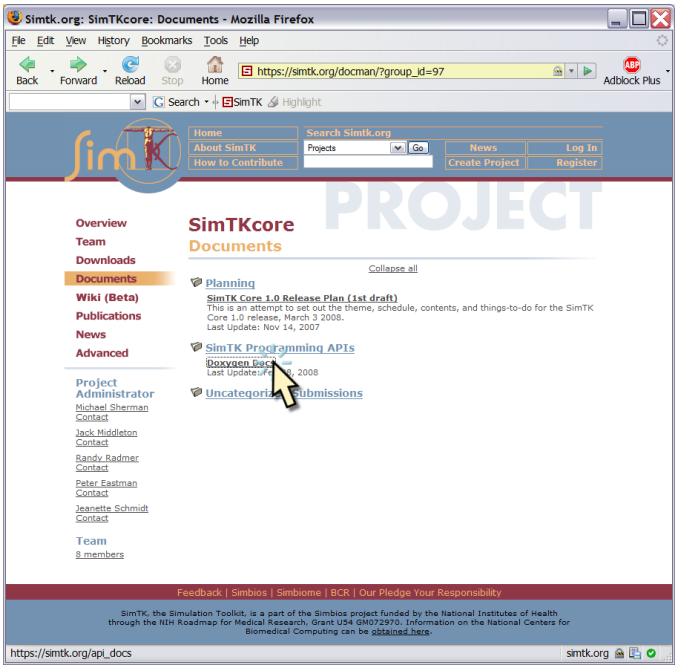


Figure 3.3-2: Selecting the "Doxygen Docs" link.

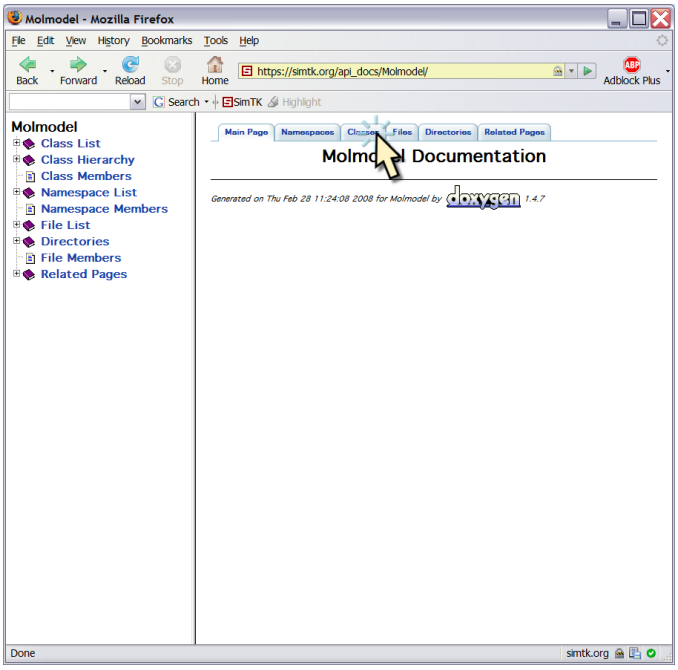


Figure 3.3-3: Selecting the "Classes" tab in API documentation.

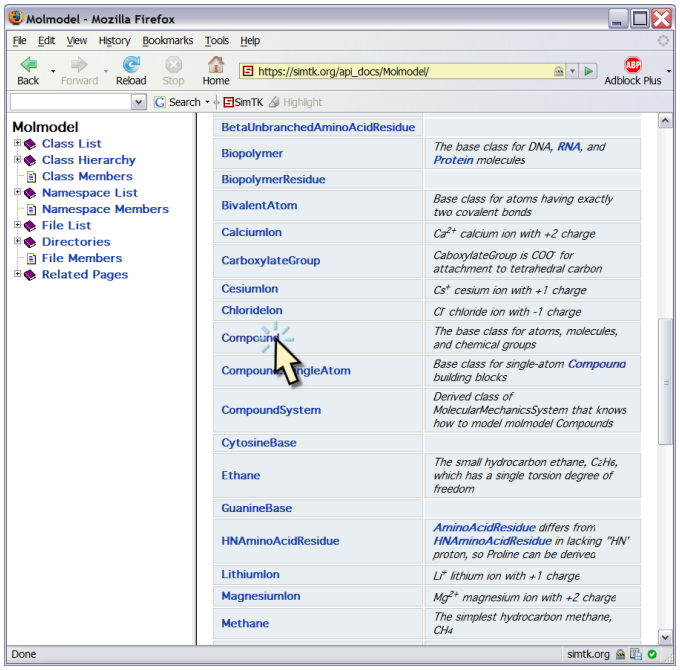


Figure 3.3-4: Selecting the Compound class in the API documentation.

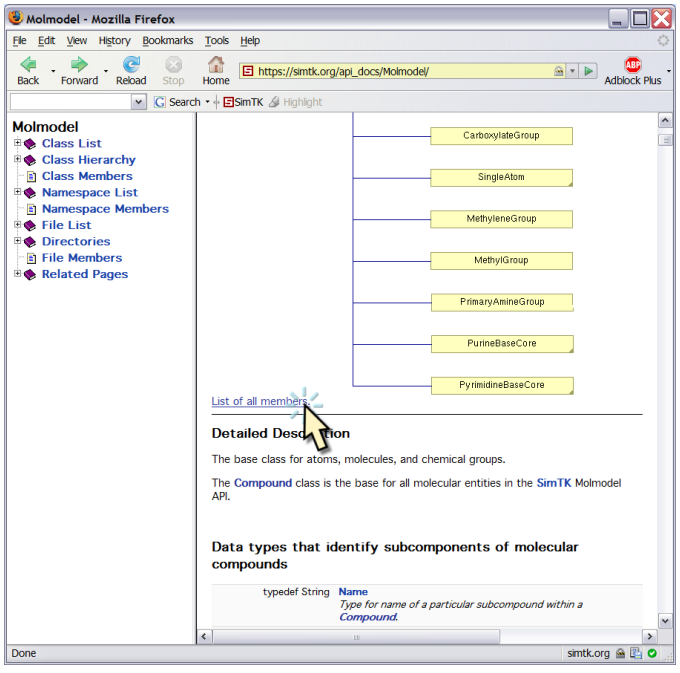


Figure 3.3-5: Getting an alphabetical list of Compound methods.

3.4 Exercises

Exercise 3.4-1

Compile and run "RNA example" program.

Exercise 3.4-2

See how much faster it runs if you don't write PDB coordinates.

Exercise 3.4-3

Simulate with electrostatic forces turned off. Use the methods **`setCoulombGlobalScaleFactor()`** and **`setGbsaGlobalScaleFactor()`** of **`DuMMForceFieldSubsystem`**.

Exercise 3.4-4

If you run this simulation for a long time you may find that the molecule drifts out of the view. This is a common problem with molecular simulations and the common solution is to remove the overall rigid body motion of the molecule as a whole periodically. Add that feature to the above simulation and verify that the molecule does not drift over a long time. Hint: look at the API documentation for `MassCenterMotionRemover`, which is a periodic event handler. You may also want to look at the example program `ExampleFoldPolyalanine` which makes use of this handler.

chapter

4

4 Tuning Molecular Mobility

4.1 Modeling and Coarse-grained Representations

The default mobilities of Compounds defined in the **`Compound.h`**, **`Protein.h`**, and **`RNA.h`** header files are internal coordinate mobilities. In other words, bond lengths and bond angles are constrained to be fixed, while dihedral (torsion) angles are permitted to move during simulation. Further, planar groups, such as peptide bonds and aromatic ring systems, are held rigid by default. These default mobilities can be changed. Such changes must be made before the multibody model is finalized with the CompoundSystem::modelCompounds() method.

The example below simulates a rigid protein, where all mobilities, including those for the dihedral angles, are fixed. It is not very interesting, because the protein is not capable of moving. If there were *two* rigid proteins on the other hand, they would be able to move relative to one another. The code is also useful as a template for experimenting with different mobility combinations.

Example 4.1-1: Simulating a Rigid Protein

```
#include "Molmodel.h"
#include <iostream>
#include <exception>
using namespace SimTK;

int main() {
try {
    CompoundSystem system;
    SimbodyMatterSubsystem matter(system);
    DecorationSubsystem decorations(system);
    DuMMForceFieldSubsystem forceField(system);

    forceField.loadAmber99Parameters();

    Protein protein("SIMTK");
    protein.assignBiotypes();
    system.adoptCompound(protein);

    for ( Compound::BondIndex bondIx(0);
          bondIx < protein.getNumBonds();
          ++bondIx)
    {
        // set all bonds rigid
        protein.setBondMobility(
            BondMobility::Rigid,
            bondIx);
    }

    system.addEventReporter(new Visualizer::Reporter(system,
        0.020));

    // finalize assignment of atoms to bodies
    system.modelCompounds();

    // Maintain a constant temperature
    system.addEventHandler(new VelocityRescalingThermostat(
              system,  293.15, 0.1));

    // Instantiate simbody model and get default state
    State state = system.realizeTopology();

    // Relax the structure before dynamics run
```

```
        LocalEnergyMinimizer::minimizeEnergy(system, state, 15.0);

        // Simulate it.

        VerletIntegrator integ(system);
        TimeStepper ts(system, integ);
        ts.initialize(state);
        ts.stepTo(20.0);

        return 0;
    }
    catch(const std::exception& e) {
        std::cerr << "ERROR: " << e.what() << std::endl;
        return 1;
    }
    catch(...) {
        std::cerr << "ERROR: An unknown exception was raised"
                  << std::endl;
        return 1;
    }

    }
```

4.2 Specifying the Degrees of Freedom of a Molecule

The dynamic mobility of a molecule is controlled by setting the mobilities of each of its bonds. You can set the mobility of a particular bond using the **`setBondMobility()`** method.

```
Compound::setBondMobility(
            BondMobility::Mobility mobility,
            Compound::BondIndex bondIndex);
```

This approach works well for setting every bond in a Compound to a particular mobility, in which case you loop over every index from zero to getNumBonds()-1 as shown in Example 4.1-1 above. To set the mobility for a particular bond, use the version of setBondMobility that takes atom names as arguments:

```
Compound::setBondMobility(
```

```
BondMobility::Mobility mobility,
Compound::AtomName atomName1,
Compound::AtomName atomName2);
```

See the API reference documentation for more information.

4.3 Defining Dihedral Angles

To specify particular atoms and bonds in a protein or RNA structure, you need to know the names of the residues that comprise the molecule. Protein and RNA residue names are numeric and begin with the name "1" at the beginning of the chain, and proceed to "2", "3", etc. although they do not have to be consecutive. So you could identify a particular atom as "3/O5*" for example.

You can set a dihedral angle at the time of bond creation, at which point several assumptions are made about the meaning of that dihedral angle in the interest of concise syntax.

To set a dihedral angle later, you create a name for that angle, and then use that name to access the dihedral later in your program. A dihedral angle is properly defined by a sequence of four bonded atoms. You can define a dihedral like so:

```
defineDihedralAngle("angleName", "atom1Name", "atom2Name",
"atom3Name", "atom4Name")
```

You can also define a dihedral angle in terms of TWO bond centers: a) the BondCenter linking atom2 to atom 1, and b) the BondCenter linking atom 3 to atom 4.

4.4 An Index is Not an ID

Avoid the error of using an AtomIndex defined in one Compound to identify the same atom in a subcompound or parent Compound. For example, the atom with AtomIndex 13, for example, in a particular AminoAcidResidue Compound within a Protein Compound, will, in general, have a different AtomIndex at the Protein level.

Every atom is however assigned a unique index by the DuMM force field subsystem, of type DuMM::AtomIndex, rather than Compound::AtomIndex.

4.5 Combining Fine-grain and Coarse-grain Mobilities in a Single Model

The following example creates a small RNA molecule containing three nucleotide residues. Each of the three residues has a different set of mobilities.

Example 4.5-1: Multi-grain RNA simulation

```
#include "Molmodel.h"
#include <iostream>
#include <fstream>
using namespace SimTK;

int main() {
try {
   // Initialize simbody objects
   CompoundSystem system;
   SimbodyMatterSubsystem matter(system);
   DecorationSubsystem decoration(system);
   DuMMForceFieldSubsystem forces(system);
   forces.loadAmber99Parameters();

   // Initialize molmodel objects
   RNA rna("AAA");
   rna.assignBiotypes();
   system.adoptCompound(rna);

   // Set bond mobilities
   Compound& residue1 = rna.updResidue(Compound::Index(0));
   Compound& residue2 = rna.updResidue(Compound::Index(1));
   Compound& residue3 = rna.updResidue(Compound::Index(2));

   // Set first residue to Euclidean mobilities
   for (Compound::BondIndex bond(0);
        bond < residue1.getNumBonds(); ++bond)
      residue1.setBondMobility(BondMobility::Free, bond);
```

```
    // Leave second residue at default combination of
    // Torsion and Rigid mobilities

    // Set third residue to Rigid
    for (Compound::BondIndex bond(0);
         bond < residue3.getNumBonds(); ++bond)
        residue3.setBondMobility(BondMobility::Rigid, bond);

    // Assign atoms to bodies
    system.modelCompounds();

    // Maintain temperature
    system.addEventHandler(new VelocityRescalingThermostat(
            system,  293.15, 0.1));

    // Show me a movie
    system.addEventReporter( new VTKEventReporter(system,
0.015) );

    // Instantiate simbody model and get default state
    State state = system.realizeTopology();

    // Relax the structure before dynamics run
    LocalEnergyMinimizer::minimizeEnergy(system, state, 15.0);

    // Prepare for molecular dynamics
    VerletIntegrator integrator(system);
    integrator.setAccuracy(0.001);
    TimeStepper timeStepper(system, integrator);
    timeStepper.initialize(state);

    // Start simulation
    timeStepper.stepTo(500.0);

    return 0;
}
catch(const std::exception& e) {
    std::cerr << "ERROR: " << e.what() << std::endl;
    return 1;
}
catch(...) {
```

```
        std::cerr << "ERROR: An unknown exception was raised"
                  << std::endl;
        return 1;
    }

    }
```

4.6 Exercises

Exercise 4.6-1

Compile and run "rigid protein" example. Why is this simulation a bit boring?

Exercise 4.6-2

Add a second rigid protein to interact with the first one. You will need to place the second protein at a different location than the first protein. Use the optional second argument of the adoptCompound() method to choose this location (see API documentation).

Exercise 4.6-3

Modify the RigidProtein example to make every atom independent. We won't write out the source code for this one. Set the bond mobilities to BondMobility::Free instead of BondMobility::Rigid in Example 4.1-1.

Exercise 4.6-4

Compile and run the ExampleAdenylateMobilitiesViz program

Exercise 4.6-5

Install the program VMD (http://www.ks.uiuc.edu/Research/vmd/), and view the ExampleAdenylateMobilitiesVMD example program using that viewer.

chapter

5

5 Loading a Molecule from PDB Coordinates

The previous protein example created a protein in a fully extended configuration. Functional proteins in the real world are folded into compact shapes. It would be extremely tedious to set all of the internal coordinates manually to match a known structure. Fortunately, the coordinates for a folded protein are captured in PDB (Protein Data Bank) format molecular structure files which can be read into Molmodel. The Molmodel API includes a few approaches for loading molecular structures from PDB files.

5.1 Using the PDBReader Class

The LoadPDB program reads a protein configuration from a PDB (Protein Data Bank) format file. In addition, there are a few more nice features that bring this simulation a bit closer to being physically reasonable than the other examples in this guide.

This program is based on an example from the Simbody User's Guide by Peter Eastman.

Example 5.1-1: Simulating a protein from PDB coordinates.

```
#include "Molmodel.h"
```

```
#include <iostream>
#include <exception>

using namespace SimTK;

int main() {
try {
    // Load the PDB file and construct the system.
    CompoundSystem system;
    SimbodyMatterSubsystem matter(system);
    DecorationSubsystem decorations(system);
    DuMMForceFieldSubsystem forceField(system);
    forceField.loadAmber99Parameters();

    PDBReader pdb("1AKG.pdb");
    pdb.createCompounds(system);
    system.modelCompounds();

    system.addEventHandler(new VelocityRescalingThermostat(
        system, 293.15, 0.1));
    system. addEventReporter(new Visualizer::Reporter(system,
        0.025));

    system.realizeTopology();

    // Create an initial state for the simulation.
    State state = system.getDefaultState();
    pdb.createState(system, state);
    LocalEnergyMinimizer::minimizeEnergy(system, state, 15.0);

    // Simulate it.

    VerletIntegrator integ(system);
    integ.setAccuracy(1e-2);
    TimeStepper ts(system, integ);
    ts.initialize(state);
    ts.stepTo(10.0);

    return 0;
}
catch(const std::exception& e) {
    std::cerr << "ERROR: " << e.what() << std::endl;
    return 1;
}
```

```
catch(...) {
    std::cerr << "ERROR: An unknown exception was raised"
              << std::endl;
    return 1;
}

}
```

5.2 PDBReader Methods

One way to load in the PDB file information is to create a PDBReader object, as shown below:

```
PDBReader pdb("1AKG.pdb");
pdb.createCompounds(system);
…
pdb.createState(system, state);
```

The PDBReader object manages the reading of a PDB file, in this case "1AKG.pdb", and directly populates the CompoundSystem object. Thus the PDBReader methods obviate the need for the Protein() constructor, the assignBiotypes() method, and the adoptCompound() method.

5.2.1 Advantages of the PDBReader class

- PDBReader can read Protein and RNA molecules from PDB files

5.2.2 Disadvantages of PDBReader class

- cannot read molecules other than Protein and RNA
- cannot model structures with large rigid segments

5.3 Internal Coordinates Differ from Cartesian Coordinates

The atomic coordinates in a PDB file specify Cartesian (x, y, z) coordinates in orthogonal Angstrom units for the location of each atom in a molecule.

Structures in Molmodel use internal coordinates to specify atomic locations. After the first three atoms of a molecule are placed, the location of each atom is specified relative to three other atoms. Three values are used to specify the atom's position:

1. Bond length to the previous atom
2. Bond angle formed by atom and the two previous atoms
3. Dihedral angle formed by the atom and the three previous atoms

Molmodel uses internal coordinates to specify atomic locations for two reasons. First, this representation is closely related to the internal coordinate dynamics model that is created by default. Second, internal coordinates can be more convenient for defining localized structural groups.

5.4 Default (initial) Configuration Differs from Dynamic Configuration

Although `Compound` structures are defined using internal coordinates in Molmodel, this does not imply that internal coordinate dynamics must be used in your simulation. The choice of the number of degrees of freedom to use in dynamic simulation (e.g., internal coordinate, full Cartesian, or rigid bodies) is made after the initial (default) configuration of the `Compound` has been specified. That initial default configuration is always specified in internal coordinates.

The program in Example 5.1-1 demonstrates a technique for setting the degrees of freedom in a dynamic model to match the structure in a PDB file. This approach might not work well for models that have large sections of rigid bonds. An approach that incorporates the configuration from a PDB into the initial default configuration is under development.

5.5 Another Way to Load PDB Structures: Specialized Protein Constructor

Example 5.5-1: Load from PDB File Using Specialized Protein() constructor

```
#include "Molmodel.h"
#include <iostream>
#include <exception>
#include <fstream>

using namespace SimTK;

int main() {
try {
    // Load the PDB file and construct the system.
    CompoundSystem system;
    SimbodyMatterSubsystem matter(system);
    DecorationSubsystem decorations(system);
    DuMMForceFieldSubsystem forceField(system);
    forceField.loadAmber99Parameters();

    std::ifstream pdbFile("1AKGtrim.pdb");
    Protein protein(pdbFile);

    protein.assignBiotypes();
    system.adoptCompound(protein);
    system.modelCompounds();

    system.addEventHandler(new VelocityRescalingThermostat(
        system, 293.15, 0.1));
    system.addEventReporter(new VTKEventReporter(system,
        0.025));

    system.realizeTopology();

    // Create an initial state for the simulation.
    State state = system.getDefaultState();
    LocalEnergyMinimizer::minimizeEnergy(system, state, 15.0);

    // Simulate it.

    VerletIntegrator integ(system);
    integ.setAccuracy(1e-2);
    TimeStepper ts(system, integ);
    ts.initialize(state);
    ts.stepTo(10.0);
```

```
        return 0;
    }
    catch(const std::exception& e) {
        std::cerr << "ERROR: " << e.what() << std::endl;
        return 1;
    }
    catch(...) {
        std::cerr << "ERROR: An unknown exception was raised"
                  << std::endl;
        return 1;
    }

    }
```

5.5.1 Advantages of the Protein(istream) constructor

- can read from any istream, not just named files
- can model rigid segments after structure is loaded

5.5.2 Disadvantages of Protein(istream) constructor

- has trouble with unrecognized residue types in the PDB file
- only exists for Protein, not RNA (for now)

5.6 Exercises

Exercise 5.6-1

Download PDB structure 1AKG from http://www.rcsb.org/

Exercise 5.6-2

Load and simulate 1AKG structure.

chapter

6

6 Constructing a Custom Molecule

6.1 Introduction to Custom Molecule Construction

The current Molmodel API is focused on easy construction of RNA and protein molecules. This API also makes it possible to construct other molecule types. Construction of a Compound from scratch in Molmodel is a complex subject. This chapter gives a light overview of the process.

The examples here do not go to the very deepest level of Compound construction, because it uses instances of the Compound::SingleAtom subclass, which are themselves built upon a lower API. It is recommended to use classes derived from Compound::SingleAtom, including UnivalentAtom, BivalentAtom, TrivalentAtom, etc., and their higher level descendants AliphaticCarbon and AliphaticHydrogen, as is done in the example in this chapter.

Careful examination of the example programs in this and subsequent chapters, combined with examination of the various molecule definitions in the header file Compound.h, may provide enough information for a motivated programmer to design new molecule types.

6.2 Compound Parts List

To get started constructing custom molecules, it is important to understand the fundamental building blocks that are used to construct **`Compound`**s. Figure 6.2-1 shows a pictorial representation of these parts in a partially constructed molecule. The numbers in parentheses show the number of each part in the figure: there are nine BondCenters, three Atoms, two Bonds, and one top-level Compound.

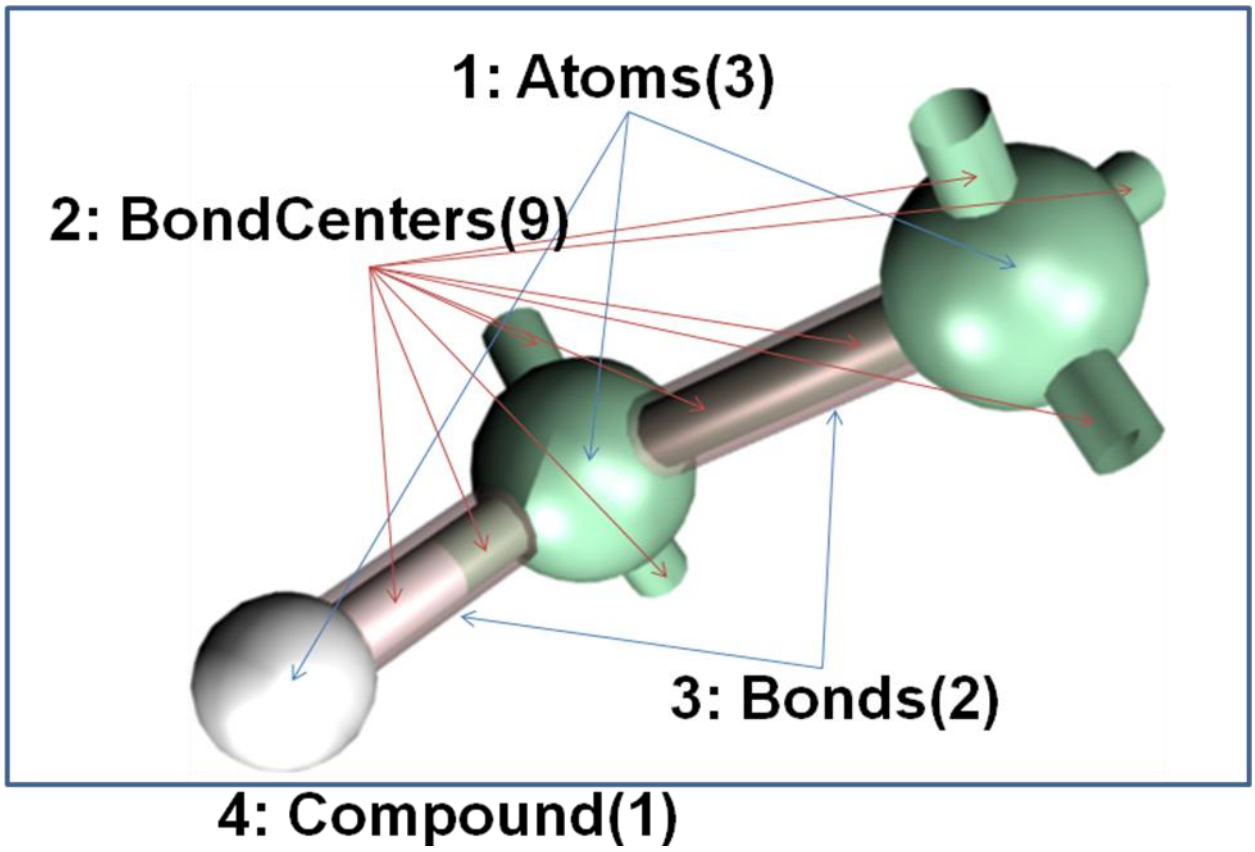


Figure 6.2-1: Parts of a Compound.

6.2.1 Atoms and Bonds

Atoms and Bonds correspond directly to atoms and covalent bonds in the real world. There is no explicit Atom class in the public Molmodel API. Atoms are specified using atom names or atom indices within a **`Compound`**.

A Bond is formed by connecting BondCenters on two Atoms. There is no explicit Bond class in the public Molmodel API. Bonds are specified by Bond names or Bond indices within a Compound.

6.2.2 BondCenters

A BondCenter represents one half-bond, or a location on an Atom where a Bond can be formed. Thus it is possible to specify, via its BondCenters, how many Bonds an Atom can make, even before any other Atoms have been introduced. There is no explicit BondCenter class in the public Molmodel API. BondCenters are specified by BondCenter name or BondCenter index within a **`Compound`**.

6.2.3 Compounds

A Compound is composed of Atoms, Bonds, BondCenters, other sub-Compounds, and need not represent a complete molecule. **`Compound`** is the central base class for molecular structures in the Molmodel API. For example, **`Protein`**, **`RNA`**, **`AminoAcidResidue`**, **`Argon`**, and **`Ethane`** are all derived classes of **`Compound`**.

chapter

7

7 Simulating Two Custom Argon Atoms

Our first foray into custom molecule construction will represent the interaction of two argon atoms. Argon is an inert noble gas, meaning that is does not have chemical bonds. This fact simplifies the force field considerations. The only important force affecting the interaction between argon atoms is the van der Waals interaction, which is mildly attractive at large distances, and highly repulsive at short distances.

Argon is one of a small number of molecule types that are predefined in the "Compound.h" header file in the Molmodel installation directory. That is how we are able to use it as a type in this example program. So this is not a completely custom molecule.

7.1 TwoArgons Example Program

Example 7.1-1: Simulating two argon atoms

```
#include "Molmodel.h"
using namespace SimTK;

int main() {
    // molecule-specialized simbody System
    CompoundSystem system;
```

```
    // matter is required
    SimbodyMatterSubsystem  matter(system);

    // molecular force field
    DuMMForceFieldSubsystem forceField(system);

    // required for visualization
    DecorationSubsystem     artwork(system);

    // Define an atom class for argon
    forceField.defineAtomClass_KA(
        DuMM::AtomClassIndex(100),
        "argon",
        18,
        0,
        1.88,
        0.0003832
        );
    forceField.defineChargedAtomType(
        DuMM::ChargedAtomTypeIndex(5000),
        "argon",
        DuMM::AtomClassIndex(100),
        0.0
        );

    if (! Biotype::exists("argon", "argon"))
        Biotype::defineBiotype(Element::Argon(), 0, "argon",
"argon");

    forceField.setBiotypeChargedAtomType(
DuMM::ChargedAtomTypeIndex(5000), Biotype::get("argon",
"argon").getIndex() );
    forceField.setGbsaGlobalScaleFactor(0);

    Argon argonAtom1, argonAtom2; // two argon atoms

    // place first argon atom, units are nanometers
    system.adoptCompound(argonAtom1, Vec3(-0.3, 0, 0));

    // place second argon atom, units are nanometers
    system.adoptCompound(argonAtom2, Vec3( 0.3, 0, 0));

    system.addEventReporter(new Visualizer::Reporter(system,
```

```
            0.500));

        system.modelCompounds(); // assign atoms to bodies

        State state = system.realizeTopology();

        // Simulate it.

        VerletIntegrator integ(system);
        TimeStepper ts(system, integ);
        ts.initialize(state);
        ts.stepTo(500.0);
    }
```

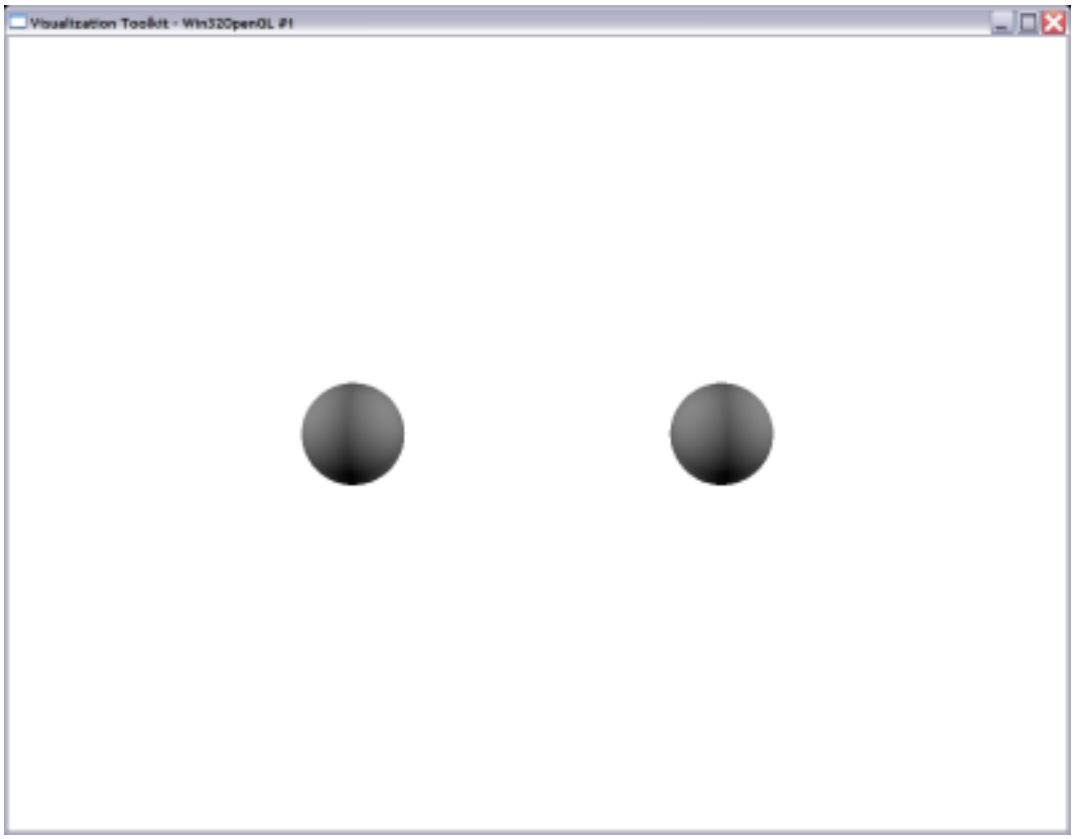


Figure 7.1-1: Frame from Argon Atom Animation

If all goes well, you should see an animation of two argon atoms repeatedly bumping into one another (Figure 7.1-1).

7.2 TwoArgons Program Discussion

7.2.1 Define Atom Class

In this example, we create an empty force subsystem, and explicitly define the few force parameters needed for argon.

DuMMForceFieldSubsystem has two levels of hierarchy when defining atom types. The first, more general, level is the atom "class" which roughly corresponds to a particular element in a particular bonding environment. The second, more detailed, level is the atom "charged type" which includes a partial charge on the atom and is discussed in section 7.2.2. Let's begin by looking at the code defining the first, more general level:

```
// Define an atom class for argon
forceField.defineAtomClass_KA(
    DuMM::AtomClassIndex(100),
    "argon",
    18,
    0,
    1.88,
    0.0003832
    );
```

The "**_KA**" part of **defineAtomClass_KA()** denotes that the units used are based on kilocalories-per-mole and Angstroms, rather than on kilojoules-per-mole and nanometers, which is the default set of units.

The first argument, 100, is the index within the force field subsystem for the new atom class that is being defined. The number 100 is meant to be large enough to probably not collide with other atom class indices that have been defined. If another class already has index 100, an error will occur. Yes, this is not particularly elegant – it makes more sense when you're defining a whole force field that has a defined set of atom classes.

The second argument, "argon," is a name for the atom class, and has no practical use.

The third argument, 18, is the atomic number of the chemical element represented by the atom class; in this case argon, which is element number 18 on the periodic table of the elements.

The fourth argument, 0 (zero), is the number of bonding partners for this atom type. Since argon is an inert gas, it never forms bonds.

The fifth argument, 1.88, is the van der Waals radius of the atom class, here defined as half the distance between two argon atoms that are separated by a distance that minimizes the energy of their interaction. This value may not be quite right and is difficult to get a perfect value for. That is why we will be so relieved when we start using predefined force field parameters in later examples. The value is in Angstroms because of the _KA suffix on the method name. Otherwise it would have to be in nanometers. In any case Molmodel will translate to nanometers immediately for internal use.

The sixth and final argument, 0.0003832, is the energy well-depth of the van der Waals interaction at its minimum. The units are kilocalories-per-mole. If the method did not have the _KA suffix, the units would have to be in kilojoules-per-mole. In any case Molmodel will translate to kJ/mole immediately for internal use.

WARNING: The numbers for van der Waals radius and well-depth in this example are made up. Do not treat them as physically accurate!

7.2.2 Define Charged Atom Type

The "charged atom type" for an atom further refines the atom "class" by assigning a particular atomic partial charge to the atom type. The code for doing this is provided below:

```
forceField.defineChargedAtomType(
    DuMM::ChargedAtomTypeIndex(5000),
    "argon",
    DuMM::AtomClassIndex(100),
```

```
        0.0
        );
```

The first argument, 5000, is the index in the force subsystem for the new charged type that is being defined. Like the atom class index, it is chosen to not collide with other values. It is much larger because there are potentially many more different charged types.

The second argument is a name for the charged type. It is not used.

The third argument refers to the atom class of which this charged atom type is a sub-type. That 100 must match the 100 in the call to **`defineAtomClass_KA()`**.

The final argument, 0.0 (zero), is the total partial charge on the charged atom type. In the case of argon, the net charge is zero, which is part of why the force situation in this case is particularly simple.

7.2.3 Biotypes

The previous section discussed atom "class" and atom "charged type," which are both atom classifications related to specific force field parameters. Biotype is another atom classification. But Biotype is not associated with a specific force field.

The purpose of the Biotype is to decouple the chemical concept of the atom from any particular force field that models the atom. In other words, the Biotype for a particular atom can be defined before any force field has been chosen. The Biotype then acts as a link between the chemical atom type and the atom types used in a particular force field. The concept of the Biotype is borrowed from the molecular mechanics package TINKER (J. W. Ponder, 1987).

First, we link a particular atom to a Biotype, which has approximately the same granularity as the atom charged type, but can be assigned before a force field is chosen. Second, once a force field is chosen, the Biotypes are linked to atom charged types of the force field. The Biotype has no charge associated with it.

Biotypes are managed by the Biotype class, and are independent of any particular force field, Compound, or even simulation. To define a biotype for argon, we use the Biotype::defineBiotype() method:

```
if (! Biotype::exists("argon", "argon"))
    Biotype::defineBiotype(Element::Argon(),
                           0,        // number of bonds
                           "argon",  // compound name
                           "argon"); // atom name
```

The Biotype::defineBiotype() method takes four arguments. The first argument is the chemical element of that Biotype. The second argument is the number of bonds to the atom in the Biotype category. Argon does not form bonds, so this value is zero. The third argument is the Compound name of the Biotype, and fourth, the atom name. In the case of argon, we have chosen "argon" for both the Compound name and the atom name.

This defines a global argon biotype, but does not attach it to anything. The biotype for the argon atoms could be set using the Compound::setAtomBiotype() method. In this case, the biotype has already been assigned in the constructor for Argon() in Compound.h, so we do not need to do it in the example program.

Once the force field parameters are defined, the argon biotype can be associated with a particular atom charged type:

```
    forceField.setBiotypeChargedAtomType(
DuMM::ChargedAtomTypeIndex(5000), Biotype::get("argon",
"argon").getIndex() );
```

The DuMMForceFieldSubsystem::setBiotypeChargedAtomType() method takes two arguments: the index of an existing atom charged type, and the index of a Biotype.

The numbering of Biotype indices is arbitrary, and is managed by the Biotype class.

7.2.4 Declaring the Argon Compounds

The following line creates two argon atoms, using the constructor found in the header file Compound.h.

```
Argon argonAtom1, argonAtom2; // two argon atoms
```

These atoms are members of the class Argon, which is derived from Compound. No multibody model has yet been constructed at this point.

7.3 Where is the "Atom" type?

There is no explicit Atom type exposed in the public Molmodel API. Atoms are managed within Compounds using atom names and atom indices. Compound is the central parent data type in Molmodel from which Molecules, Residues, and other molecular assemblies are derived. Atoms, Bonds, and BondCenters are identified within a Compound using names or indices. See section 6.2 for more details.

7.4 Exercises

Exercise 7.4-1

Compile and run the two argon example program.

Exercise 7.4-2

Add a third argon atom. Be careful to place it neither too close to, nor too far away from, the other atoms. Try to keep the initial locations of each atom at least 0.3 nanometers away from each of the other atoms.

chapter

8

8 Simulating Two Custom Ethane Molecules

We will now move from the simplest molecular simulation, argon, to one that includes chemical bonds. The inclusion of bonds (and charges) increases the complexity of the molecular force field. In this example, we will ignore most of that complexity, and make use of predefined force field parameters. We will still need to define the atomic charges, however.

The ethane molecule is the second simplest hydrocarbon, and consists of two carbon atoms and six hydrogen atoms. It is the simplest hydrocarbon that possesses a torsion angle, which requires a series of four atoms to be bonded together sequentially. Torsion angles are a central feature of internal coordinate simulation.

Ethane is one of a small number of molecule types that are predefined in the `"Compound.h"` header file in the Molmodel distribution. That is how we are able to use it as a type in the example program. Examining `"Compound.h"` can be useful when trying to design new molecules. See chapter 6 for more details.

8.1 ExampleTwoEthanes Program

Example 8.1-1: Simulating two ethane molecules

```
#include "Molmodel.h"
#include <iostream>
#include <fstream>

using namespace SimTK;

int main()
{
    CompoundSystem system;
    SimbodyMatterSubsystem  matter(system);
    DuMMForceFieldSubsystem forceField(system);
    DecorationSubsystem     artwork(system);

    // Atom classes are available, but not charged atom types
for ethane
    // in standard Amber force field
    forceField.loadAmber99Parameters();

    if (! Biotype::exists("ethane", "C"))
        Biotype::defineBiotype(Element::Carbon(), 4,
                               "ethane", "C");
    if (! Biotype::exists("ethane", "H"))
        Biotype::defineBiotype(Element::Hydrogen(), 1,
                               "ethane", "H");

    forceField.defineChargedAtomType(
        DuMM::ChargedAtomTypeIndex(5000),
        "ethane C",
        DuMM::AtomClassIndex(1), // "CT" type in amber
        -0.060 // made up
        );
    forceField.setBiotypeChargedAtomType(
DuMM::ChargedAtomTypeIndex(5000), Biotype::get("ethane",
"C").getIndex() );

    forceField.defineChargedAtomType(
        DuMM::ChargedAtomTypeIndex(5001),
        "ethane H",
        DuMM::AtomClassIndex(34), // "HC" type in amber
        0.020 // made up, use net neutral charge
        );
```

```
    forceField.setBiotypeChargedAtomType(
DuMM::ChargedAtomTypeIndex(5001), Biotype::get("ethane",
"H").getIndex() );

    Ethane ethane1, ethane2;

    // place first ethane, units are nanometers
    // skew it a little to break strict symmetry
    system.adoptCompound(ethane1,
        Transform(Vec3(-0.5, 0, 0))
        * Transform(Rotation(0.1, YAxis)) );

    // place second ethane, units are nanometers
    system.adoptCompound(ethane2, Vec3( 0.5, 0, 0));

    system.addEventReporter(new Visualizer::Reporter(system,
        0.050));

    // assign atoms to bodies, finalize multibody system,
    // get default state
    system.modelCompounds();
    State state = system.realizeTopology();

    // Simulate it.
    VerletIntegrator integ(system);
    TimeStepper ts(system, integ);
    ts.initialize(state);
    ts.stepTo(200.0);
}
```

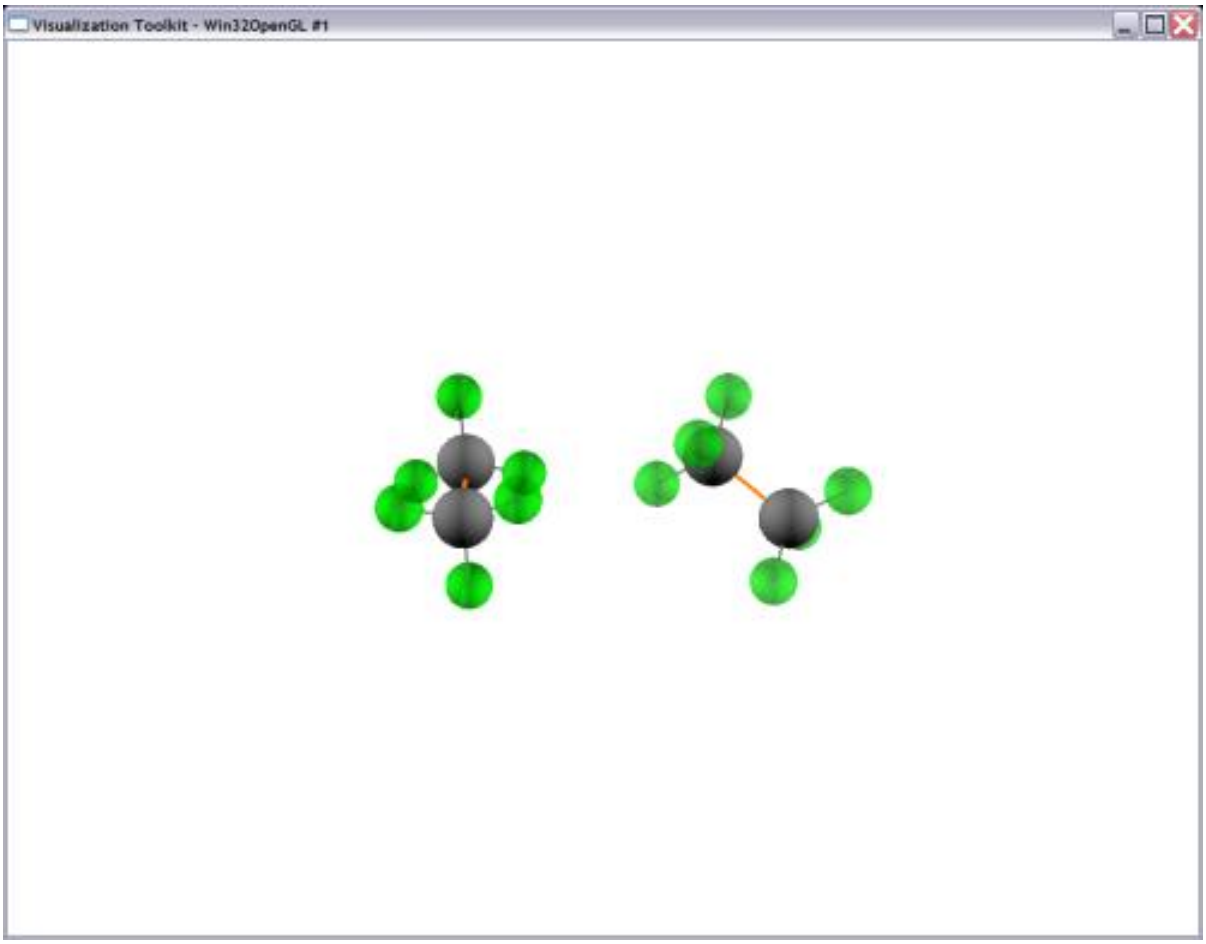


Figure 8.1-1: Frame from Two Ethanes Simulation. Note that bonds that can move are displayed in orange.

8.2 Discussion of TwoEthanes Program

8.2.1 Force Field

As in the TwoArgons program, we create an empty DuMMForceFieldSubsystem object to manage the forces:

```
DuMMForceFieldSubsystem forceField(system);
```

Unlike the case with the TwoArgons program, we can leverage some predefined force field parameters for our ethane simulation, in this case those of the AMBER99 force field:

```
forceField.loadAmber99Parameters();
```

The AMBER99 force field does not include parameters for the ethane molecule itself, but it does include atom classes that can be used for ethane. So by using the loadAmber99Parameters() method, we avoid the need to call defineAtomClass() methods, as we did in the TwoArgons example.

8.2.2 Define Charged Atom Type

The AMBER99 force field gives us parameters for the atom classes in ethane, but not for the atom charged types. Again, we will make up some parameters for these.

There are two kinds of atoms in ethane: carbon and hydrogen. There are two carbon atoms and six hydrogen atoms, but only two atom types, because each hydrogen atom is chemically equivalent to all of the others, and each carbon atom is chemically equivalent to the other one. This equivalence is implied by the symmetry of the ethane molecule.

So we need to define the atom charged types for the carbon and hydrogen atoms:

```
forceField.defineChargedAtomType(
    DuMM::ChargedAtomTypeIndex(5000),
    "ethane C",
    DuMM::AtomClassIndex(1), // "CT" type in amber
    -0.060 // made up
    );

forceField.defineChargedAtomType(
    DuMM::ChargedAtomTypeIndex(5001),
    "ethane H",
    DuMM::AtomClassIndex(34), // "HC" type in amber
    0.020 // made up, use net neutral charge
    );
```

The charges on the atom are made up and probably wrong, but I did make sure that the total charge on the whole ethane molecule will be zero, because ethane does not have a net charge. I guesstimated the charges themselves based on the parameters for similar groups

that actually are found in the standard AMBER99 force field. Again, do not take these atomic charges as correct. This is just for expository purposes.

The bad part about using the predefined atom classes is that I needed to know the AMBER99 atom class indices for tetrahedral carbon (1) and for aliphatic hydrogen (34), as used by the program TINKER. There is no good way right now to look those indices up in the Molmodel API. Sorry.

8.2.3 Declare the Ethane Compounds and attach them to the system

The next part of the code declares the ethane compounds and attaches them to the system:

```
Ethane ethane1, ethane2;

system.adoptCompound(ethane1,
    Transform(Vec3(-0.5, 0, 0))
    * Transform(Rotation(0.1, YAxis)) );
```

Like Argon, Ethane is one of the few Compounds defined in the header file Compound.h.

8.2.3.1 Initial orientation/reference frame of each molecule

In the ExampleTwoArgon program, the center of each argon atom was placed at the location given in the adoptCompound() method. What part of the ethane molecule goes there?

The specified location is where the first atom of the Compound will be located. In the case of ethane, that atom is the first carbon, atom "C1." In the Compound reference frame, the first atom center is at the origin, the first BondCenter of that atom is along the y-axis, and the second BondCenter of the first atom lies in the x-y plane. These rules define the internal reference frame of a Compound.

In one of the adoptCompound() statements, I have multiplied the starting pose by another Transform. I won't explain in detail here exactly what that does (see the Simbody User's Guide for more information), but its purpose is to skew the orientation of one of the ethane molecules a bit to break perfect symmetry, so the simulation will look more interesting.

The rest of ExampleTwoEthanes follows the same concepts as the ExampleTwoArgonAtoms example.

8.2.4 Why are some bonds gray and others orange?

If you look carefully at the ethane animation, you will see that the bond connecting the carbons is orange, while the carbon-hydrogen bonds are gray. Gray bonds connect members of the same rigid body. So each methyl group is a single rigid body. The only internal motion permitted is a rotation about the carbon-carbon bond. This is an internal coordinate simulation. Internal coordinate simulation, in which bond lengths and bond angles remain fixed, while dihedral angle are permitted to vary, is the default modeling behavior of the Molmodel API.

8.3 Exercises

Exercise 8.3-1

Compile and run the ExampleTwoEthanes program.

Exercise 8.3-2

Add a third ethane molecule. Keep in mind that each ethane molecule is about 0.4 nanometers wide, and is centered on the "C1" atom. The long direction of each molecule is initially along the y-axis.

Exercise 8.3-3

Adjust the mobilities of the three ethane molecules so that one moves using internal coordinates (the default), one is completely rigid, and the third is a full Cartesian model. You may want to examine Chapter 4 of this guide before attempting this.

chapter

9

9 Defining a New Molecule: Propane

The example program below defines and simulates a new molecule, propane. Here for the first time we're creating a molecule that has not already been defined for us in Compound.h. The training wheels are off!

Example 8.3-1: Complete Program for Defining and Simulating Propane.

```
#include "Molmodel.h"
#include <iostream>
#include <fstream>

using namespace SimTK;

// Propane is a three carbon linear alkane
// C(3)H(8), or CH3-CH2-CH3
class Propane : public Molecule
{
public:
    // constructor
    Propane()
    {
        setCompoundName("Propane");
```

```
        instantiateBiotypes();

        // First atom
        setBaseAtom( AliphaticCarbon("C1") );
        setBiotypeIndex( "C1", Biotype::get("propane",
"C1_or_C3").getIndex() );
        convertInboardBondCenterToOutboard(); // this is the
root of the parent compound

        // Second atom
        bondAtom( AliphaticCarbon("C2"), "C1/bond1" );
        setBiotypeIndex( "C2", Biotype::get("propane",
"C2").getIndex() );

        // Third atom
        bondAtom( AliphaticCarbon("C3"), "C2/bond2" );
        setBiotypeIndex( "C3", Biotype::get("propane",
"C1_or_C3").getIndex() );

        // First methyl hydrogens
        bondAtom( AliphaticHydrogen("H11"), "C1/bond2" );
        bondAtom( AliphaticHydrogen("H12"), "C1/bond3" );
        bondAtom( AliphaticHydrogen("H13"), "C1/bond4" );
        setBiotypeIndex("H11", Biotype::get("propane",
"H1_or_H3").getIndex() );
        setBiotypeIndex("H12", Biotype::get("propane",
"H1_or_H3").getIndex() );
        setBiotypeIndex("H13", Biotype::get("propane",
"H1_or_H3").getIndex() );

        // Second methylene hydrogens
        bondAtom( AliphaticHydrogen("H21"), "C2/bond3" );
        bondAtom( AliphaticHydrogen("H22"), "C2/bond4" );
        setBiotypeIndex("H21", Biotype::get("propane",
"H2").getIndex() );
        setBiotypeIndex("H22", Biotype::get("propane",
"H2").getIndex() );

        // Third methyl hydrogens
        bondAtom( AliphaticHydrogen("H31"), "C3/bond2" );
        bondAtom( AliphaticHydrogen("H32"), "C3/bond3" );
        bondAtom( AliphaticHydrogen("H33"), "C3/bond4" );
        setBiotypeIndex("H31", Biotype::get("propane",
"H1_or_H3").getIndex() );
```

```
        setBiotypeIndex("H32", Biotype::get("propane",
"H1_or_H3").getIndex() );
        setBiotypeIndex("H33", Biotype::get("propane",
"H1_or_H3").getIndex() );
    }

    static void instantiateBiotypes() {
        // Create biotypes if they do not exist yet
        // four chemically distinct atom types
        if (! Biotype::exists("propane", "C1_or_C3"))
            Biotype::defineBiotype(Element::Carbon(), 4,
"propane", "C1_or_C3");
        if (! Biotype::exists("propane", "C2"))
            Biotype::defineBiotype(Element::Carbon(), 4,
"propane", "C2");
        if (! Biotype::exists("propane", "H1_or_H3"))
            Biotype::defineBiotype(Element::Hydrogen(), 1,
"propane", "H1_or_H3");
        if (! Biotype::exists("propane", "H2"))
            Biotype::defineBiotype(Element::Hydrogen(), 1,
"propane", "H2");
    }

    // create charged atom types
    // ensure that charges sum to zero, unless molecule has a
formal charge
    static void
setAmberLikeParameters(DuMMForceFieldSubsystem& forceField)
    {
        instantiateBiotypes();

        DuMM::ChargedAtomTypeIndex chargedAtomIndex(5000);

        forceField.defineChargedAtomType(
            chargedAtomIndex,
            "propane C1_or_C3",
            DuMM::AtomClassIndex(1), // "CT" type in amber
            -0.060 // made up
            );
        forceField.setBiotypeChargedAtomType(
chargedAtomIndex, Biotype::get("propane",
"C1_or_C3").getIndex() );
        ++chargedAtomIndex;
```

```
        forceField.defineChargedAtomType(
            chargedAtomIndex,
            "propane C2",
            DuMM::AtomClassIndex(1), // "CT" type in amber
            -0.040 // made up
            );
        forceField.setBiotypeChargedAtomType(
chargedAtomIndex, Biotype::get("propane", "C2").getIndex() );
        ++chargedAtomIndex;

        forceField.defineChargedAtomType(
            chargedAtomIndex,
            "propane H1_or_H3",
            DuMM::AtomClassIndex(34), // "HC" type in amber
            0.020 // made up, use net neutral charge
            );
        forceField.setBiotypeChargedAtomType(
chargedAtomIndex, Biotype::get("propane",
"H1_or_H3").getIndex() );
        ++chargedAtomIndex;

        forceField.defineChargedAtomType(
            chargedAtomIndex,
            "propane H2",
            DuMM::AtomClassIndex(34), // "HC" type in amber
            0.020 // made up, use net neutral charge
            );
        forceField.setBiotypeChargedAtomType(
chargedAtomIndex, Biotype::get("propane", "H2").getIndex() );
         ++chargedAtomIndex;
    }
};

int main() {
    CompoundSystem system;
    SimbodyMatterSubsystem  matter(system);
    DuMMForceFieldSubsystem forceField(system);
    DecorationSubsystem     artwork(system);

    // Atom classes are available, but not charged atom
    // types for propane in standard Amber force field
    forceField.loadAmber99Parameters();

    Propane::setAmberLikeParameters(forceField);
```

```
    Propane propane1, propane2;

    // place first propane, units are nanometers
    // skew it a little to break strict symmetry
    system.adoptCompound(propane1,
        Transform(Vec3(-0.5, 0, 0))
        * Transform(Rotation(0.1, YAxis)) );

    // place second propane, units are nanometers
    system.adoptCompound(propane2, Vec3( 0.5, 0, 0));

    system.addEventReporter(new Visualizer::Reporter(system,
        0.100));


    // assign atoms to bodies, finalize multibody system,
    // get default state
    system.modelCompounds();
    State state = system.realizeTopology();

    VerletIntegrator integ(system);
    TimeStepper ts(system, integ);
    ts.initialize(state);
    ts.stepTo(100.0);
}
```

9.1 The Inboard Bond Center

Every atom and every Compound has (at most) exactly one BondCenter that is known as the *inboard bond center*. For an atom, the inboard bond center is ordinarily the first BondCenter for that atom. For a Compound, the inboard bond center is ordinarily the inboard bond center of its first atom.

Every time a covalent bond is formed using a bondAtom() or bondCompound() method (but NOT those created with the addRingClosingBond() method), a bond is formed between the inboard bond center of the child compound, and an explicitly specified BondCenter of the parent compound. The inboard bond center of the parent Compound remains the inboard bond center of the resulting combined Compound.

The tree-structure of parent-child relationships that is built up using these bonding methods is directly related to the topology of the multibody system that will be created when the CompoundSystem::modelCompounds() method is called.

9.2 The First Few Atoms

Because three previous atoms are required, in general, to specify an atom location in internal coordinates, the first three atoms placed in a molecule are special. However, because the Compound::SingleAtom derived classes come preloaded with BondCenters, only the first atom is special. The relative locations of the BondCenters specify all of the bond angles.

9.2.1 The first atom

The first atom of a Compound is placed using the **`setBaseAtom()`** method. You can specify a Cartesian (x,y,z) location for the atom; otherwise it defaults to (0,0,0). When you later place an entire molecule, the location of that molecule (i.e., its reference frame) is the location of the first atom of the molecule.

9.2.2 Subsequent atoms

Additional atoms are placed relative to previous ones using the **`bondAtom()`** method.

The bond length needs to be specified, but may have a default value already built into one of the BondCenters. If a default bond length is already set on exactly one of the BondCenters (as is the case for AliphaticHydrogen), the bond length can be omitted. If both BondCenters have differing built-in default values, an error will occur.

The name of the bond center on the previous atom must be specified. As with all of the bondWhatever() methods, the *inboard bond center* in the new atom will be used to form the second half of the bond.

The bond angles are already specified by the relative arrangements of BondCenters on the atoms.

Dihedral angles are also specified in the BondAtom() and BondCompound() methods, to complete the internal coordinate representation of the default molecular configuration.

9.3 Ring-closing Bonds

Because bonded structures are built up in tree-like fashion, with child atoms and Compounds attaching to parent Compounds via their *inboard bond centers*, ring and loop closures require a special process (not shown in the propane example; look in the header files Protein.h and RNA.h for examples). One bond in each ring or cycle must be specified using the addRingClosingBond() method. This method takes two BondCenters as arguments, and has no effect upon the implicit tree structure of the Compound. Although you can specify a default bond length and dihedral angle with the addRingClosingBond() method, these may have no effect upon the default configuration, which is completely specified by the internal coordinates defined using non-ring-closing bonds.

The addition of ring-closing bonds is necessary for the force field to know where all of the bonds are.

9.4 Setting Default Geometry

You can set default geometry at construction time using arguments to the setBaseAtom(), bondAtom(), and bondCompound() methods. You can change the default geometry later using setDefaultBondLength(), setDefaultBondAngle(), and setDefaultDihedralAngle() methods. Be aware that setting the default geometry will have no effect on your dynamic simulation after you have already realized a dynamic model with the CompoundSystem::modelCompounds() method.

9.5 Exercises

Exercise 9.5-1

Compile and run propane example.

Exercise 9.5-2

Create a molecule of your own. Doing this properly involves understanding the Amber atom types for each atom in your molecule, plus knowing the partial charges on each atom. It is beyond the scope of this document to explain how to determine those parameters.

chapter

10

10 Getting More Information

10.1 The Simtk.org Website

The Molmodel and Simbody project websites, hosted on the Simtk.org website (https://simtk.org/home/molmodel, https://simtk.org/home/simbody), has the latest downloads, documentation, and source code for Molmodel and Simbody. The Doxygen-generated HTML API reference, and other documents, are posted on the Documents pages of those projects, and can also be found in the "doc" subdirectory in the installation directory on your computer.

10.2 Help Us Help You: Submitting Feature Requests and Bug Reports

10.2.1 How to submit Bug Reports and Feature Requests

Bug reports and feature requests for Molmodel can be submitted online at https://simtk.org/home/molmodel, Advanced->Features & Bugs.

Figure 10.2-1: Opening the Advanced options on the navigation bar.

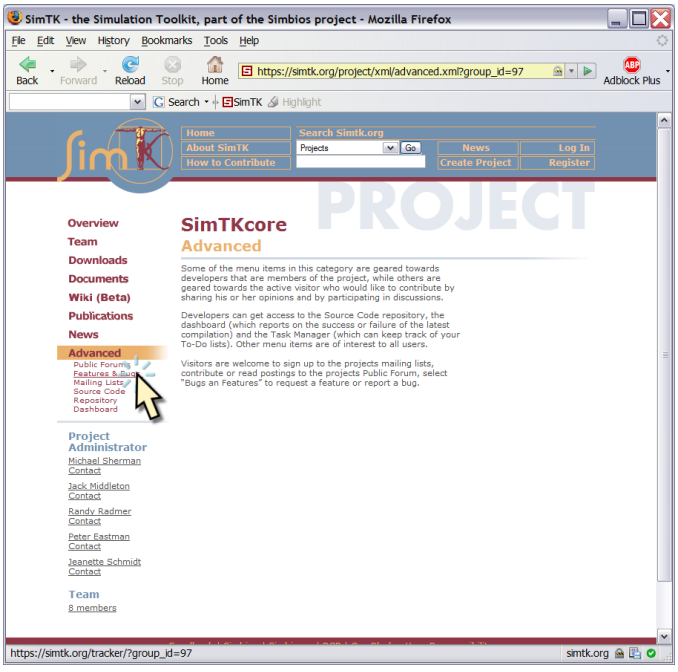


Figure 10.2-2: Selecting "Features & Bugs" page

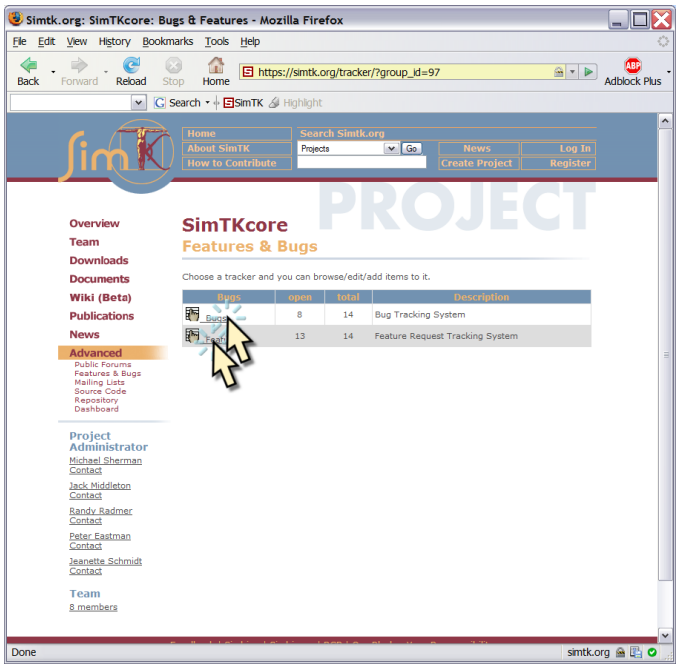


Figure 10.2-3: Choosing between Bugs or Features

10.2.2 What is the difference between a Bug Report and a Feature Request?

A "Bug Report" identifies problems where the software is not working the way it is supposed to. A "Feature Request" is suggests new functionality that does not yet exist in the software. Sometimes it is not obvious which category your issue falls into. In this case, just use your judgment. Once you have selected "Bug Reports" or "Feature Requests" from the left navigation menu, it will no longer be obvious which of the two categories you are submitting.

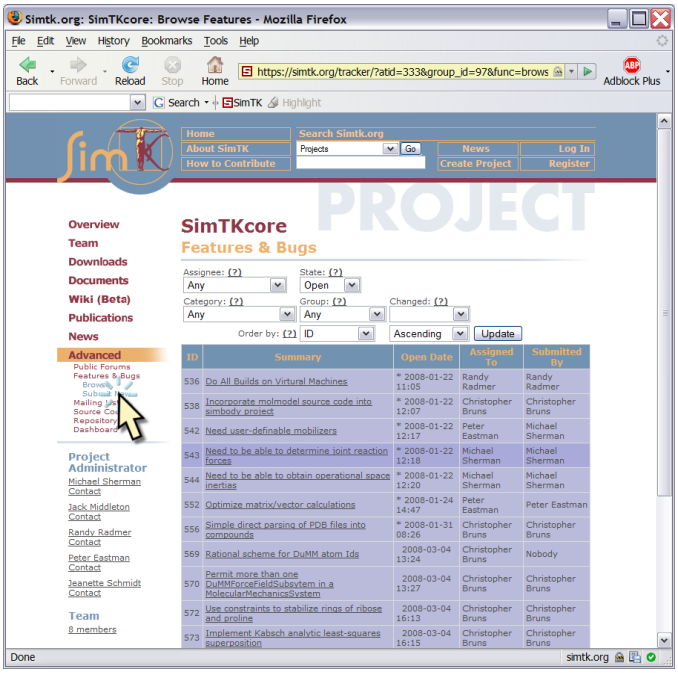


Figure 10.2-4: Choosing to submit a new Feature Request or Bug Report

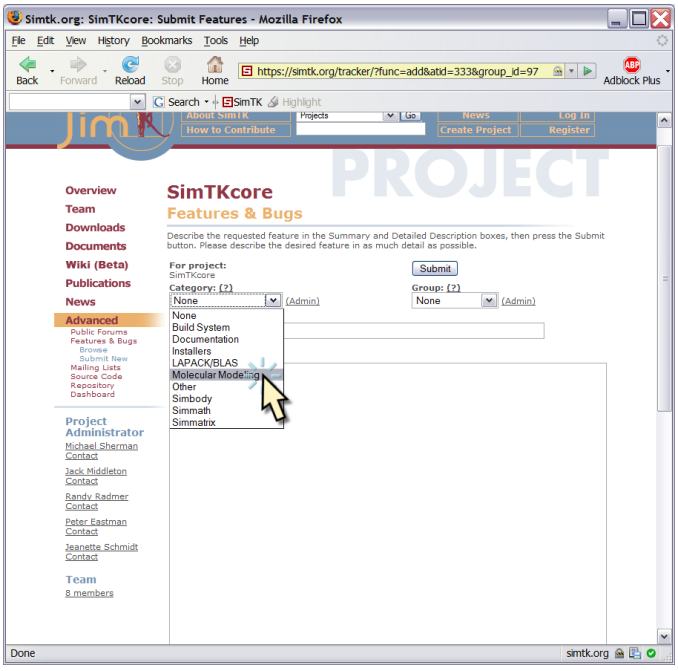


Figure 10.2-5: Selecting a Bug/Feature category.

Don't spend too much time worrying about the fields on the form you don't understand. Just provide information we can use to fix the bug or add the feature.

10.2.3 How can I ensure that I am submitting a truly excellent bug report?

Submitting high quality bug reports is an art that may require some practice. Practice the following steps to become a bug reporter who is beloved by the software developers.

When submitting a software problem report, please try to give as complete a report as possible.

1. Include a description of what you expected would happen, had there been no problem.
2. Include a detailed description of what actually did happen, including error messages and other output.
3. Create a short program that demonstrates the problem (if possible). Submit the complete program text, plus any input files, in the Bug Report. Use the "Check to Upload and Attach File" field to include these files. If that doesn't work, just paste your whole program into the comments section. Please do not be shy about pasting all of this information and files into the form.

Please include complete programs that really compile, if possible, with your Bug Reports. Do not just paste in the six or seven lines of code that you think are causing the problem. Please send us a complete program to demonstrate the error whenever you can. It is also important to include your input files, if any, with the example program. This will make it much easier for us to reproduce the bug, and therefore much more likely that we can fix it.

References

The RCSB Protein Data Bank: http://www.rcsb.org/pdb/ H.M.Berman, J.Westbrook, Z.Feng, G.Gilliland, T.N.Bhat, H.Weissig, I.N.Shindyalov, P.E.Bourne (2000) The Protein Data Bank. *Nucleic Acids Research*, **28**: 235-242

A. Onufriev , D. Bashford , D.A. Case (2004) Exploring protein native states and large-scale conformational changes with a modified generalized born model. *Proteins: Structure, Function, and Bioinformatics* **55**(2): 383-394

J. W. Ponder, F. M. Richards (1987). An Efficient Newton-like Method for Molecular Mechanics Energy Minimization of Large Molecules. *J. Comput. Chem. , 8*, 1016-1024.

J. Wang, P. Cieplak, P.A. Kollman (2000). How Well Does a Restrained Electrostatic Potential (RESP) Model Perform in Calculating Conformational Energies of Organic and Biological Molecules? *J. Comput. Chem. , 212*, 1049-1074.