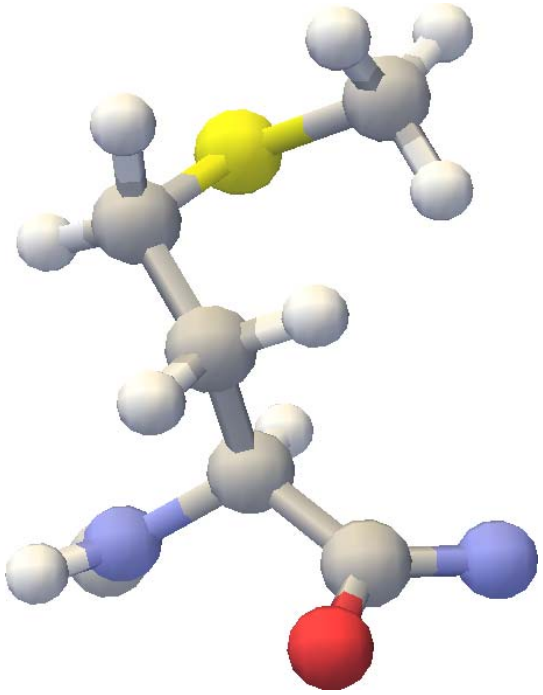
Documents



SimTK Molmodel Programmer's Guide

Release 1.0

March 19, 2008

Website: https://SimTK.org/, https://SimTK.org/docman/?group_id=97

Copyright and Permission Notice

Acknowledgments

SimTK software and all related activities are funded by the Simbios National Center for Biomedical Computing 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

1

Overview

1.1 Scope of this document

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 SimTKcore installation files and on the web. See section 5.3 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`**.

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 SimTK Simbody, the SimTK order-n multibody dynamics library. Molmodel includes methods to construct molecular models for use in simulation with Simbody. Both Molmodel and Simbody are part of the SimTK core tool kit, available at https://simtk.org/.

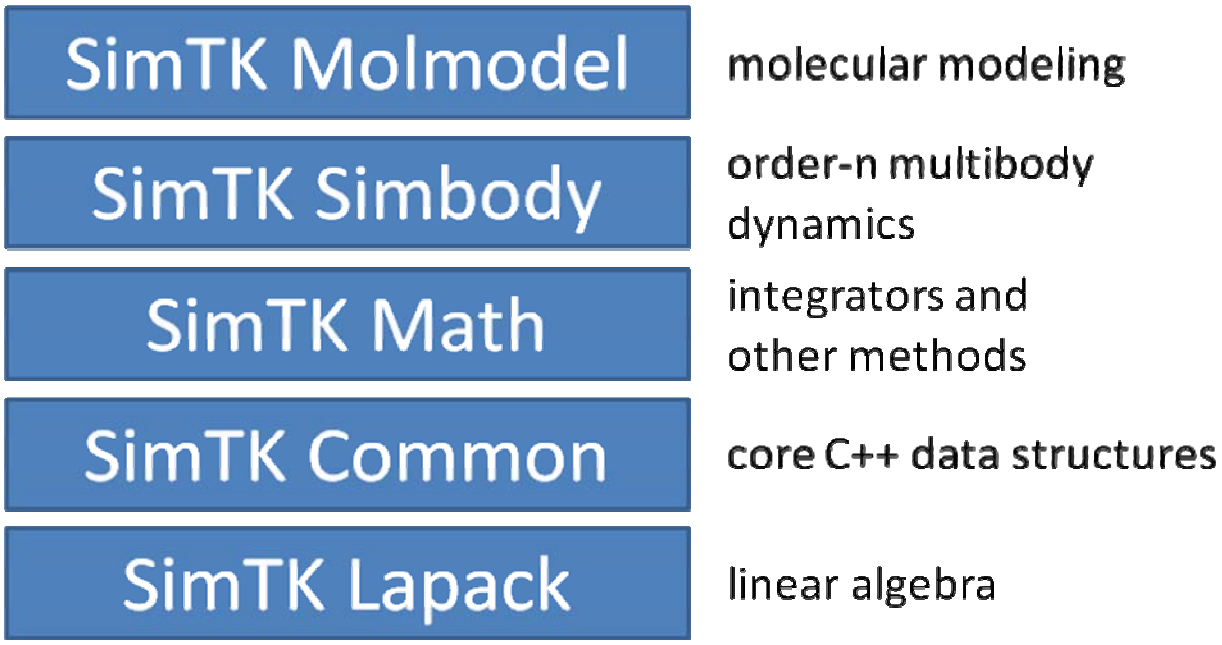


Figure 1-1: Principal libraries in the SimTK core tool kit

Figure 1-1 shows some of the key libraries in the SimTK core tool kit. 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. The CPODES library,

which represents the core methods for one of the integration methods, is not shown in the figure because it conceptually belongs in the same category as the SimTK Math library.

1.3.1 Current State of the Molmodel library

1.3.1.1 Slow

The slowest part of molecular dynamics simulations is almost always the force field calculation. The proof-of-concept force field implemented for this release has been subjected to no performance optimization, and is thus unlikely to achieve impressive speeds.

On the other hand, it is possible to evaluate the *relative* speed gains to be had by various coarse-grained modeling strategies, even if the *absolute* speeds may be (currently) discouraging.

1.3.1.2 Incomplete

There may need to be further modeling and force functionality added before truly useful molecular simulations can be accomplished with Molmodel.

1.4 Prerequisites

1.4.1 Working knowledge of C++ programming language

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

1.4.2 SimTK core simulation tool kit installed

To use the Molmodel library and the rest of the SimTK core tool kit, download the SimTK core tool kit from the “SimTK Core” project at https://simtk.org/.

1.4.3 Read the SimTK Tutorial

Molmodel is built on Simbody and the rest of the SimTK tool kit. Concepts that are covered in another important document, the SimTK tutorial, may not be covered in detail in this document.

You should become familiar with the Simbody library and API. Consult the SimTK tutorial available at the simtk.org web site. The SimTK tutorial is available in the documents section of the SimTKcore installation.

1.4.3.1 Supported platforms

At the time of this writing, the SimTK core tool kit is supported for use on three platforms:

- 32-bit Windows XP
- 32-bit Mac OS X 10.5 (Leopard) with Intel CPU
- 32-bit Linux

Please check the SimTKcore project at the SimTK.org website for the latest news on supported platforms.

1.5 Exercises

Exercise 1-1

Install the SimTK core toolkit from SimTK.org

Exercise 1-2

Run the test programs that came with the SimTK core tool kit. The test programs are called CoreInstallCheck and AuxInstallCheck.

chapter

2

Getting Started: Simulating Two Argon Atoms

Our first molecular simulation 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. If you read the SimTK tutorial, many elements of the following program should be familiar. Some of the varying program elements are highlighted in yellow.

Argon is one of a small number of molecule types that are predefined in the "Compound.h" header file in the SimTKcore distribution. That is how we are able to use it as a type in this example program.

2.1 TwoArgons Example Program

Example 2-1: Complete program for simulating two argon atoms

```
#include "SimTKmolmodel.h"
#include "SimTKsimbody_aux.h" // for vtk visualization
```

```
using namespace SimTK;
using namespace std;

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

    // matter is required
    SimbodyMatterSubsystem  matter(system);

    // molecular force field
    TinkerDuMMForceFieldSubsystem dumm(system);

    // for drawing vtk visualization
    DecorationSubsystem     artwork(system);

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

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

    dumm.setBiotypeChargedAtomType(
DuMM::ChargedAtomTypeIndex(5000), Biotype::get("argon",
"argon").getIndex() );
    dumm.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.updDefaultSubsystem().addEventReporter(new
VTKEventReporter(system,
        0.500));

    system.modelCompounds(); // finalize multibody system

    State state = system.realizeTopology();

    // Simulate it.

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

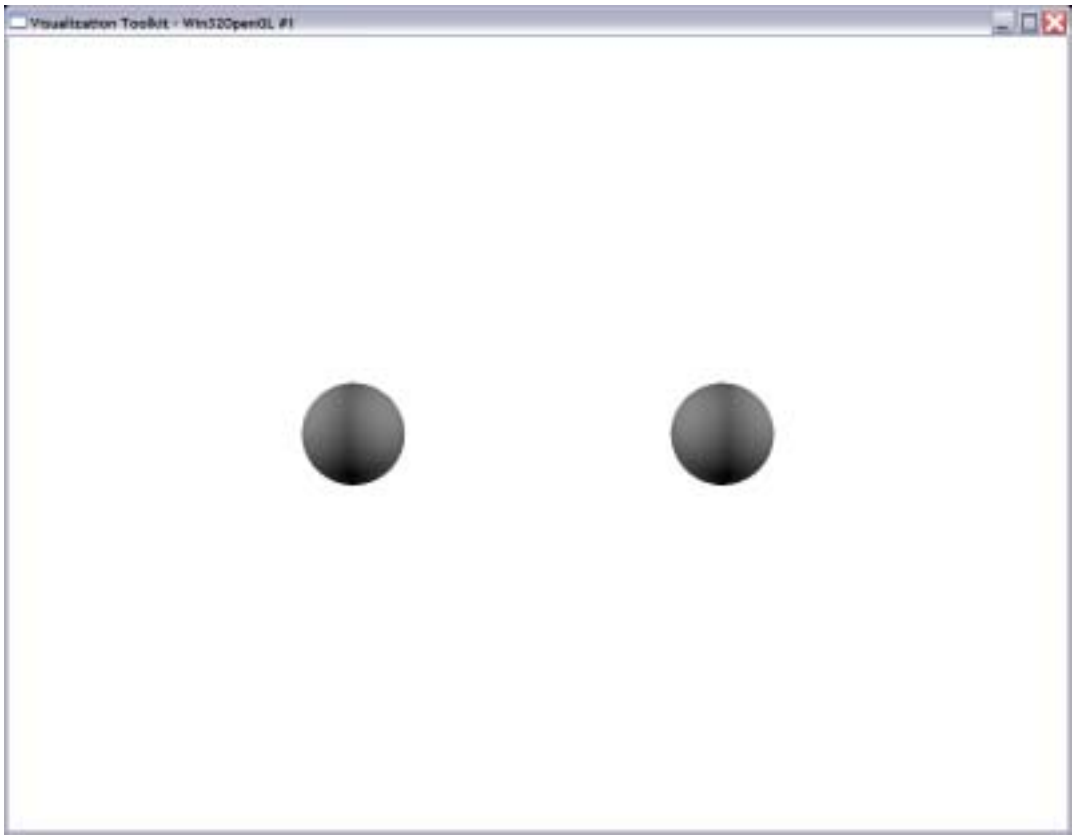


Figure 2-1: Frame from Argon Atom Animation

If all goes well, you should see an animation of two argon atoms repeatedly bumping into one another (Figure 2-1).

2.2 TwoArgons Program Discussion

2.2.1 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 SimTK tutorial. The CompoundSystem class is derived from simbody

MultibodySystem. CompoundSystem includes additional methods and data for dealing with molecular simulations.

2.2.2 Force Field

The Molmodel API implements an unoptimized force field definition for demonstrative purposes.

```
// molecular force field
TinkerDuMMForceFieldSubsystem dumm(system);
```

Molecular forces are handled by the TinkerDuMMForceFieldSubsystem class, which derives from DuMMForceFieldSubsystem, which in turn derives from ForceSubsystem.

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

2.2.3 Define Atom Class

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 2.2.4.

```
// Define an atom class for argon
dumm.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.

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.

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.

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

2.2.4 Define Atom Charged Type

The "charged type" for an atom further refines the atom "class" by assigning a particular atomic partial charge to the atom type.

```
dumm.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 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.

2.2.5 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.

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

Biotypes are managed by the Biotype class, and are independent of any particular force field, Compound, or even simulation. The Biotype::defineBiotype() method takes three 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.

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

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

2.2.6 Declaring the Argon Compounds

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

This line creates two argon atoms, using the constructor found in the header file Compound.h. These atoms are members of the class Argon, which is derived from Compound. No multibody model has yet been constructed at this point.

2.2.7 Attaching the Argon Atoms to the System

```
system.adoptCompound(argonAtom1, Vec3(-0.3, 0, 0));
```

This method transfers ownership of the argon atoms' internal data structures to the System. 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.

The second argument specifies the location and/or orientation of the molecule relative the ground frame.

2.2.8 Finalizing the multibody model

```
system.modelCompounds(); // finalize multibody system
```

The modelCompounds() step is a critical one. Modeling decisions are committed at this point. Also, the default configuration is transferred onto the dynamic configuration. Subsequent changed to the default configuration will have no affect on your simulation.

There is really only one way to model a single argon atom as a multibody model, so nothing particularly interesting is being finalized here. This situation will change in later examples.

2.2.9 Simulating

The simulation process is the same as that for non-molecular systems, as described in the SimTK tutorial. Here we have chosen the Verlet integrator, which performs well with 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 Units are nanometers, atomic mass units, and picoseconds

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.

2.4 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 8.2 for more details.

2.5 Exercises

Exercise 2-1

Compile and run the two argon example program.

Exercise 2-2

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 others atoms.

chapter

3

Simulating Two 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 SimTKcore 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 8 for more details.

3.1 TwoEthanes Example Program

Example 3-1: Complete program for simulating two ethane molecules

```
#include "SimTKmolmodel.h"
#include "SimTKsimbody_aux.h" // for vtk visualization

#include <iostream>

#include <fstream>

using namespace SimTK;

int main()
{
    CompoundSystem system;
    SimbodyMatterSubsystem  matter(system);
    TinkerDuMMForceFieldSubsystem dumm(system);
    DecorationSubsystem     artwork(system);

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

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

    dumm.defineChargedAtomType(
        DuMM::ChargedAtomTypeIndex(5001),
```

```
        "ethane H",
        DuMM::AtomClassIndex(34), // "HC" type in amber
        0.020 // made up, use net neutral charge
        );
    dumm.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.updDefaultSubsystem().addEventReporter(new
VTKEventReporter(system,
        0.050));

    system.modelCompounds(); // finalize multibody system

    State state = system.realizeTopology();

    // Simulate it.

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

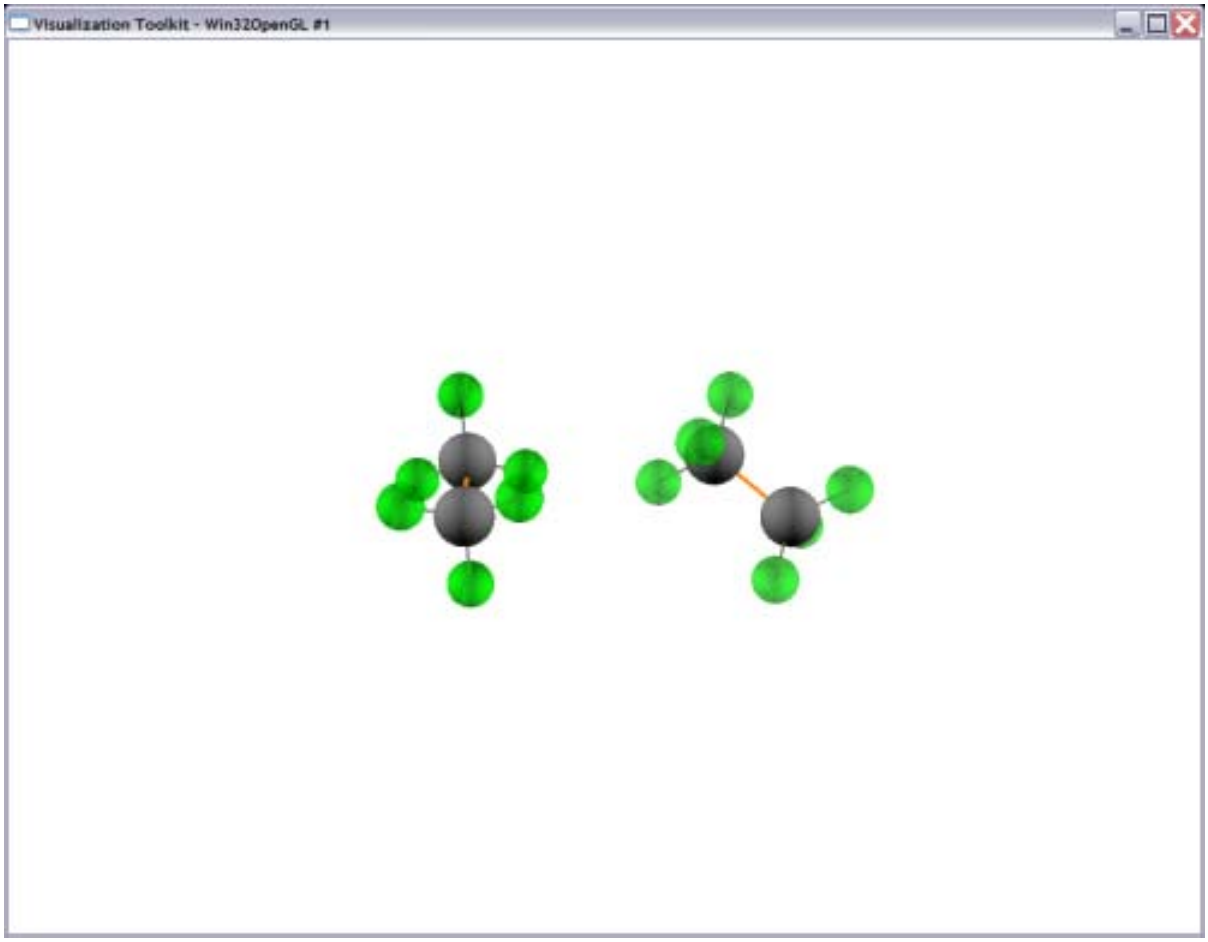


Figure 3-1: Frame from Two Ethanes Simulation

3.2 Discussion of TwoEthanes Program

3.2.1 Force Field

```
TinkerDuMMForceFieldSubsystem dumm(system);
```

As in the TwoArgons program, we create an empty TinkerDuMMForceFieldSubsystem object to manage the forces.

```
dumm.loadAmber99Parameters();
```

Unlike the case with the TwoArgons program, we can leverage some predefined force field parameters for our ethane simulation. The AMBER99 force field does not include

parameters for the ethane molecule itself, but this force field 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.

3.2.2 Define Atom Charged 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 is equivalence is implied by the symmetry of the ethane molecule.

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

dumm.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 guestimated 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.

3.2.3 Declare the Ethane Compounds and Attach 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.

3.2.3.1 Initial Orientation/Reference Frame of Each Molecule

In the TwoArgon 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 location by a Transform. I won't explain in detail here exactly what that does, 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 the TwoEthanes example follows the same concepts as the TwoArgons example.

3.2.4 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.

3.3 Do I have to do an internal coordinate simulation?

No! You can perform a full atomic Cartesian simulation using Molmodel and Simbody. The default behavior is to automatically construct internal coordinate models, but this can be changed. See chapter 7 for more details.

3.4 Exercises

Exercise 3-1

Compile and run the ExampleTwoEthanes program.

Exercise 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 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 7 of this guide before attempting this.

chapter

4

Simulating a Protein Molecule

Next we move to a much more complicated molecule type: protein. Proteins are defined in the **"Protein.h"** header file. Although protein molecules are much more complicated that argon or ethane, the following example program is actually the shortest one so far! That is because all of the force parameters for proteins are predefined in the AMBER99 force field, and because the **Protein** class includes a constructor that takes a compact sequence string as an argument. The sequence is a very compact representation of a protein's topology.

4.1 Creating a Protein Model from a Sequence String

Example 4-1: Vision for simple protein constructor

```
#include "SimTKmolmodel.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.

4.2 SimpleProtein Program

Example 4-2: Complete program for simulating a very small protein

```
#include "SimTKmolmodel.h"
#include "SimTKsimbody_aux.h" // for vtk visualization

using namespace SimTK;
using namespace std;

int main()
{
    CompoundSystem system; // molecule-specialized simbody
System
    SimbodyMatterSubsystem  matter(system); // matter is
required
    TinkerDuMMForceFieldSubsystem dumm(system); // molecular
force field
    DecorationSubsystem     artwork(system); // for drawing
vtk visualization

    dumm.loadAmber99Parameters();

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

    system.updDefaultSubsystem().addEventReporter(new
VTKEventReporter(system,
        0.020));

    system.modelCompounds(); // finalize multibody system

    State state = system.realizeTopology();

    // Simulate it.

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

```
        ts.stepTo(20.0);
    }
```

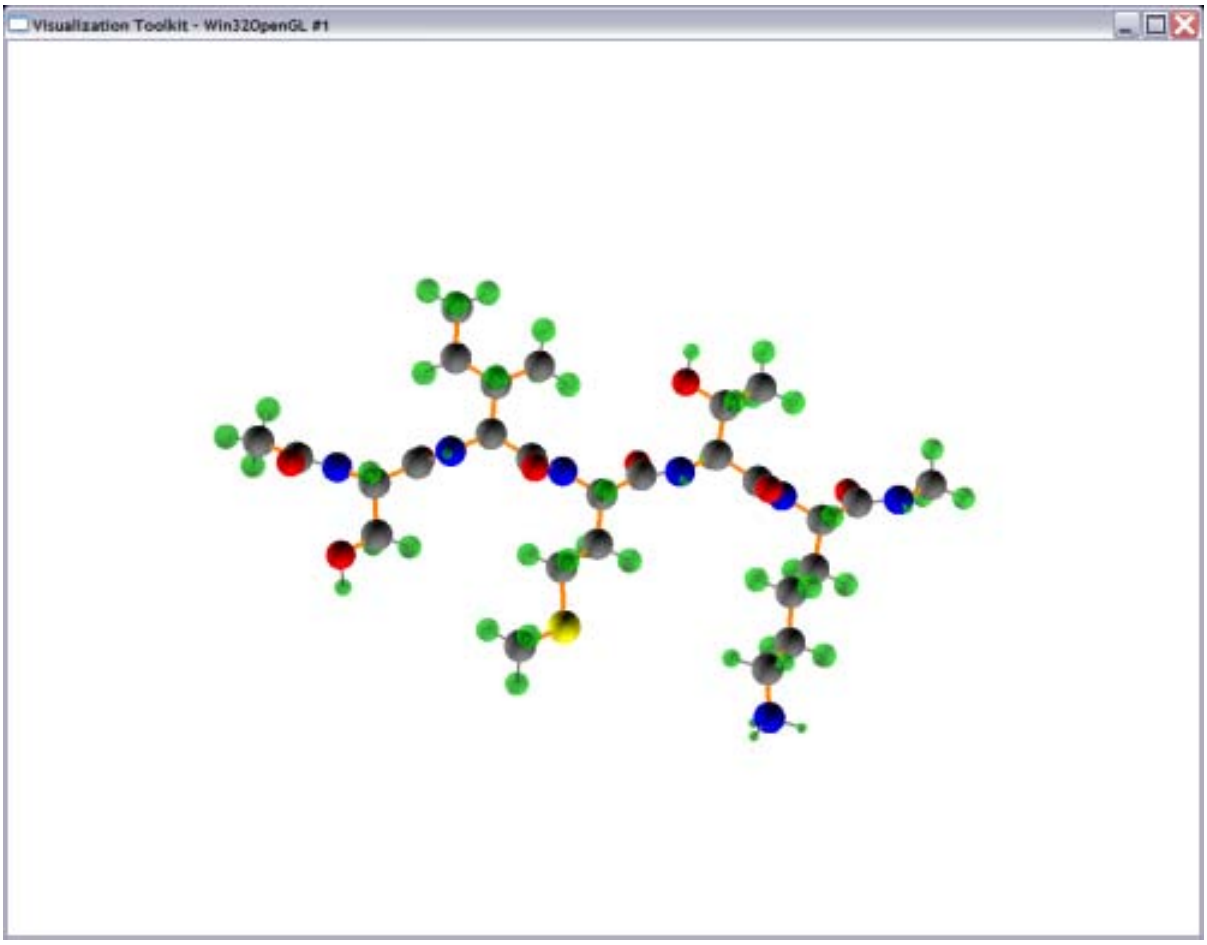


Figure 4-1: Frame from Small Protein Simulation

4.3 Analysis of SimpleProtein Program

As in the TwoEthanes example, we load AMBER99 force field parameters into the force field subsystem.

```
        dumm.loadAmber99Parameters();
```

This time, because all of the parameters for protein atoms are specified in the AMBER99 standard force field, we have much less bookkeeping to do. All of the atom partial charges come for free, for example. And they might not even be wrong.

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

The sequence constructor is used. The one letter code “SIMTK” results in a protein with sequence of five amino acids serine-isoleucine-methionine-threonine-lysine. In addition, small neutral end-cap residues are placed at both ends of the protein chain by default.

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

The assignBiotypes() method is new here. It 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.

The rest of the simulation repeat concepts raised in the TwoArgon and TwoEthane programs.

4.4 Amber99 Force Field

The TinkerDuMMForceField 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 TinkerDuMMForceFieldSubsystem 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, and thus will probably fail for other force fields pending further debugging.

4.5 Exercises

Exercise 4-1

Compile and run the protein example program.

Exercise 4-2

Try a different protein sequence.

chapter

5

Simulating an RNA Molecule

Simulating a simple RNA molecule is very similar to simulating a protein.

5.1 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 ones because atomic coordinates are being written periodically.

Example 5-1: Complete program for simulating a small RNA molecule.

```
#include "SimTKmolmodel.h"
#include "SimTKsimbody_aux.h" // for vtk visualization

using namespace SimTK;
using namespace std;

class WritePdbReporter : public PeriodicEventReporter {
public:
    WritePdbReporter(
        const MultibodySystem& system,
```

```
        const Compound& compound,
        std::ostream& outputStream,
        Real interval)
        : PeriodicEventReporter(interval),
          system(system),
          compound(compound),
          outputStream(outputStream)
    {}
    void handleEvent(const State& state) const {
        system.realize(state, Stage::Position);
        compound.writePdb(state, outputStream);
    }
private:
    const MultibodySystem& system;
    const Compound& compound;
    std::ostream& outputStream;
};

int main()
{

    CompoundSystem system; // molecule-specialized simbody
System
    SimbodyMatterSubsystem  matter(system); // matter is
required
    TinkerDuMMForceFieldSubsystem dumm(system); // molecular
force field
    DecorationSubsystem     artwork(system); // for drawing
vtk visualization

    dumm.loadAmber99Parameters();

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

    system.updDefaultSubsystem().addEventReporter(
        new VTKEventReporter(system, 0.020));

    system.updDefaultSubsystem().addEventReporter(
        new WritePdbReporter(system, rna, cout, 0.100));

    system.modelCompounds(); // finalize multibody system
```

```
        State state = system.realizeTopology();

        // Simulate it.

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

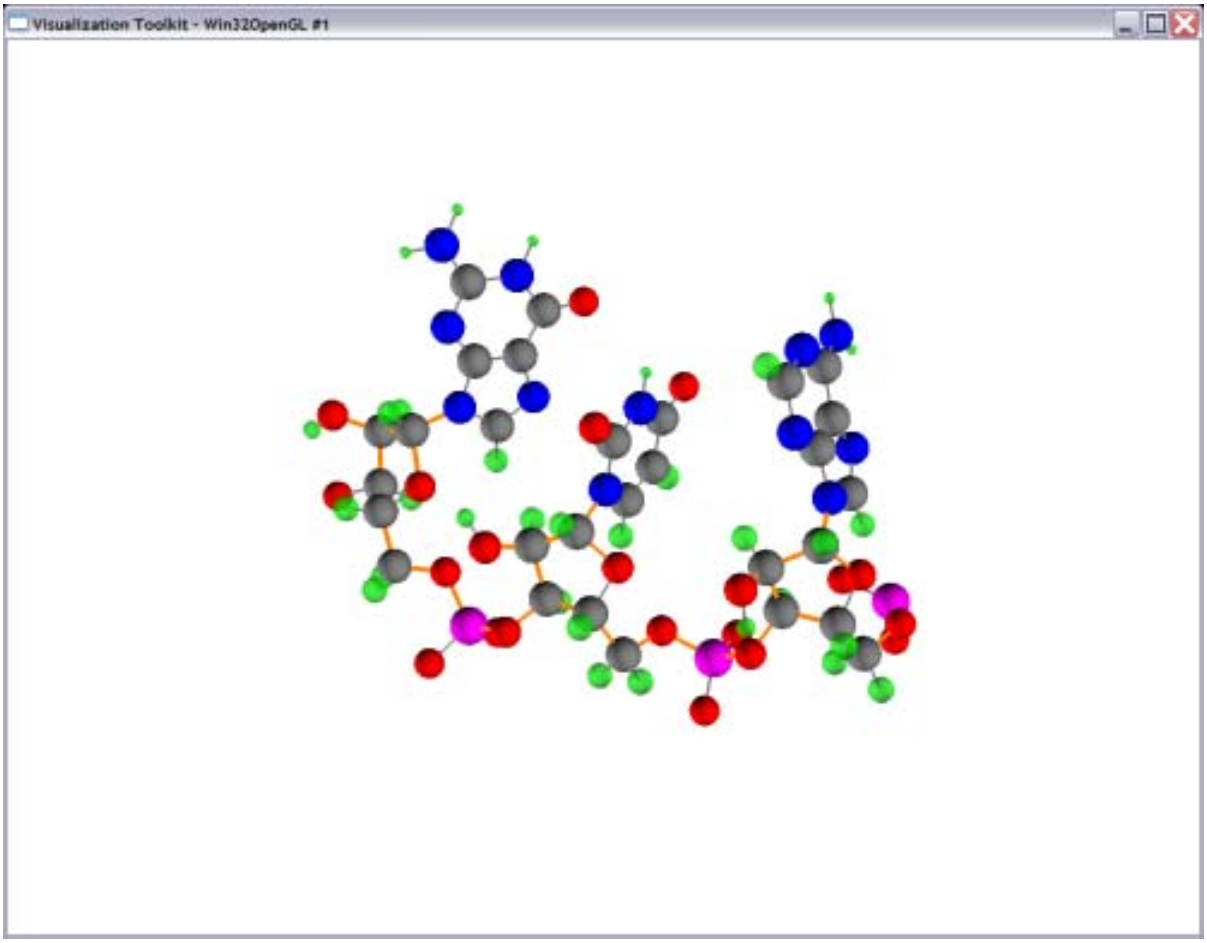


Figure 5-1: Small RNA model

WARNING: At present, the RNA molecules produced are not chemically complete. The initial phosphorus atom and final 3' oxygen atom are each

missing a bonding partner. This is a defect in the current Molmodel definition of RNA molecules.

5.2 Writing PDB coordinates

This example writes a set of PDB coordinates to the screen periodically. The atomic coordinates in PDB files can be used by many molecular computer programs to share structural data. The example program creates a new class **`WritePdbReporter`**, which is derived from **`PeriodicEventReporter`**, which is described in the SimTK tutorial. **`WritePdbReporter`** uses the **`writePdb()`** method of **`Compound`**. The Compound::writePdb() methods can also be used directly.

5.3 Finding API documentation

Using documentation of the Molmodel API is essential to mastering Molmodel programming. HTML API documentation is available in the SimTK core distribution in a directory on your computer under <SIMTK_HOME>/core/doc/api/Molmodel/index.html. (Assuming you installed the SimTK core distribution on your computer.

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

1. Browse to simtk.org and search for the “SimTKcore” project.
2. Select “SimTKcore” from the project search results list.
3. Select the “Documentation” section on the left navigation bar.
4. Select the “Doxygen Docs” link.
5. Select the “Molmodel API” link. Now you are at the page corresponding to the core/doc/api/Molmodel/index.html file in the SimTKcore distribution.

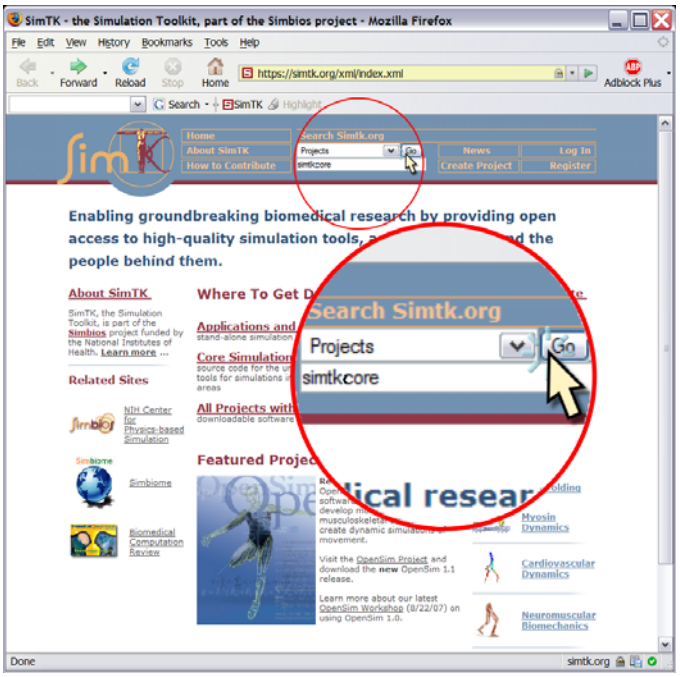


Figure 5-2: Searching for the "SimTKcore" project at SimTK.org

Figure 5-3: Selecting "SimTKcore" from project search results list.

Figure 5-4: Selecting the Documents section of the SimTKcore project.

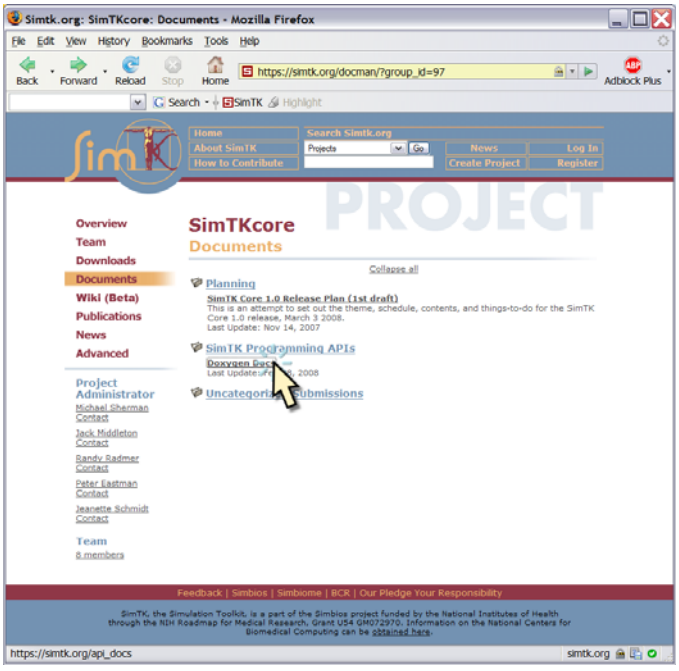


Figure 5-5: Selecting the "Doxygen Docs" link.

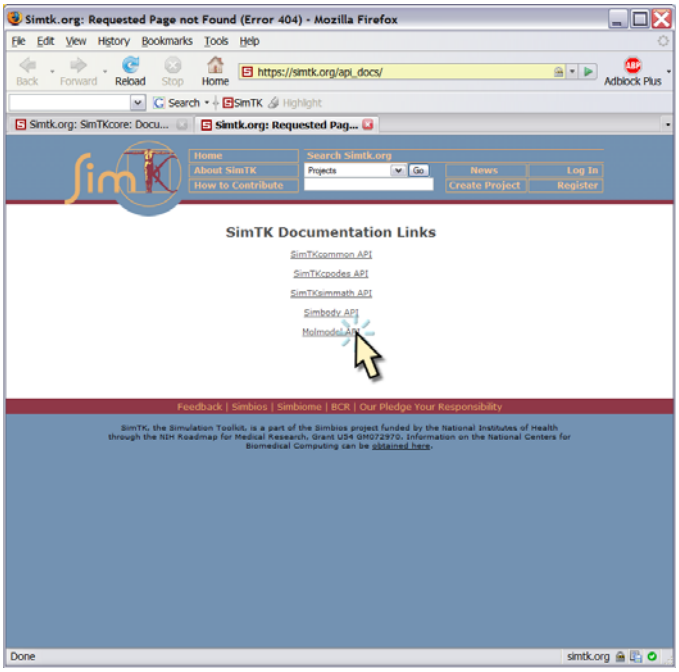


Figure 5-6: Selecting the "Molmodel API" documents.

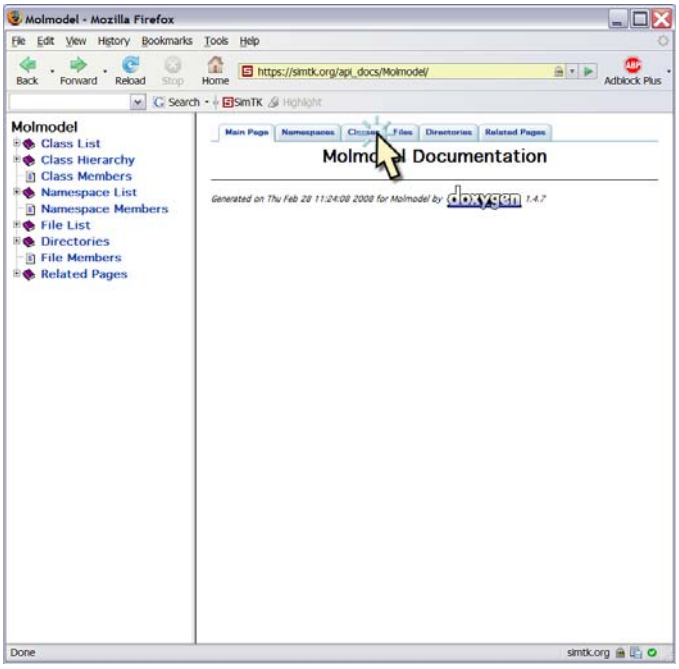


Figure 5-7: Selecting the "Classes" tab in API documentation.

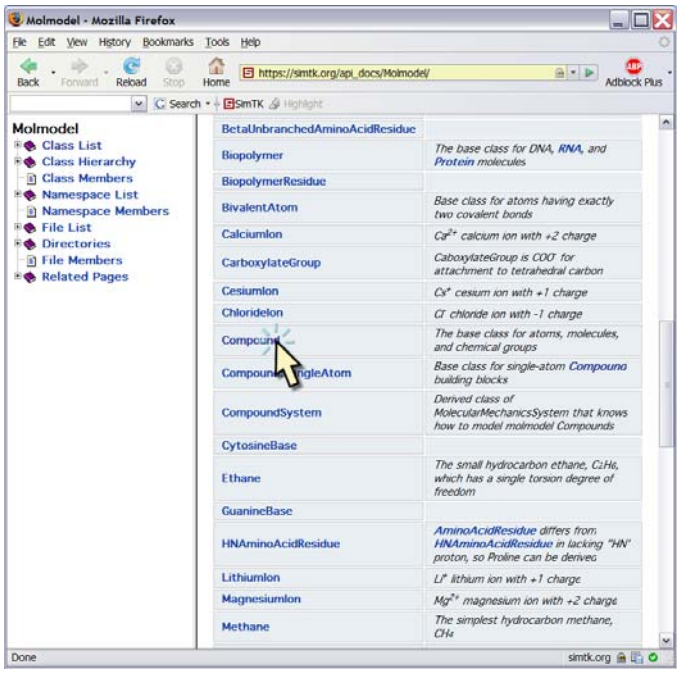


Figure 5-8: Selecting the Compound class in the API documentation.

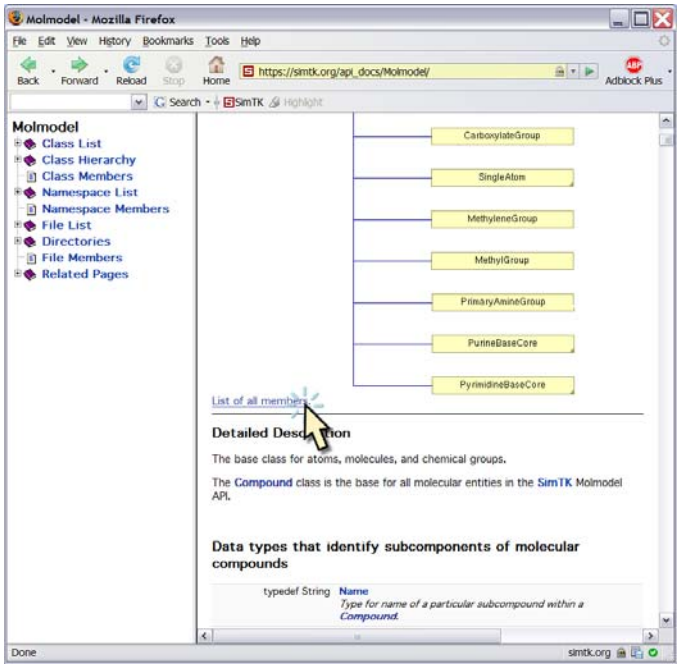


Figure 5-9: Getting an alphabetical list of Compound methods.

5.4 Exercises

Exercise 5-1

Compile and run “RNA example” program.

Exercise 5-2

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

Exercise 5-3

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

chapter

6

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. This example demonstrates a technique for matching the configuration of a protein structure in a PDB file. There are some limitations to this approach that we hope to address in a future release.

6.1 LoadPDB Program

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 SimTK tutorial by Peter Eastman.

Example 6-1: Complete program for simulating a protein from PDB coordinates.

```
#include "SimTKmolmodel.h"
```

```
#include "SimTKsimbody_aux.h"

using namespace SimTK;

int main() {

    // Load the PDB file and construct the system.
    CompoundSystem system;
    SimbodyMatterSubsystem matter(system);
    DecorationSubsystem decoration(system);
    TinkerDuMMForceFieldSubsystem forces(system);
    forces.loadAmber99Parameters();

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

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

    // Create an initial state for the simulation.

    State& state = system.updDefaultState();
    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);
}
```

6.2 Discussion of LoadPDB Program

6.2.1 PDBReader Object

```
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.

6.2.2 Maintaining Temperature

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

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

See the SimTK tutorial for more details.

6.2.3 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. See the SimTK tutorial for more details.

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

6.3 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.

6.4 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 6-1 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.

6.5 Exercises

Exercise 6-1

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

Exercise 6-2

Load and simulate 1AKG structure

chapter

7

Making an Entire Protein a Single Rigid Body

7.1 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.

Example 7-1: Complete Program for Simulating a Rigid Protein

```
#include "SimTKmolmodel.h"
#include "SimTKsimbody_aux.h"

using namespace SimTK;
using namespace std;

int main()
{
```

```
    CompoundSystem system;
    SimbodyMatterSubsystem  matter(system);
    TinkerDuMMForceFieldSubsystem dumm(system);
    DecorationSubsystem     artwork(system);

    dumm.loadAmber99Parameters();

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

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

    system.updDefaultSubsystem().addEventReporter(new
VTKEventReporter(system,
        0.020));

    // finalize multibody system
    system.modelCompounds();

    State state = system.realizeTopology();

    // Simulate it.

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

This simulation 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.

7.2 Specifying the Degrees of Freedom of a Molecule

The dynamic mobility of a molecule is set 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 getNBonds(). 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);
```

7.3 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 begin with the name “0” (zero) at the beginning of the chain, and proceed “1”, “2”, “3”, etc.
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 must be more precise about the definition of that dihedral angle. 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.

7.4 An Index is Not an ID

Avoid the error of using an AtomIndex used 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.

7.5 Exercises

Exercise 7-1

Compile and run “rigid protein” example.

7.6 OK, Now Make Every Atom Independent

Exercise 7-2

We won’t write out the source code for this one. Set the bond mobilities to `BondMobility::Free` instead of `BondMobility::Rigid` in Example 7-1.

chapter

8

Constructing a Custom Molecule

8.1 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, though not necessarily easy, 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 example program here does 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 program in this chapter, 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.

8.2 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 8-1 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.

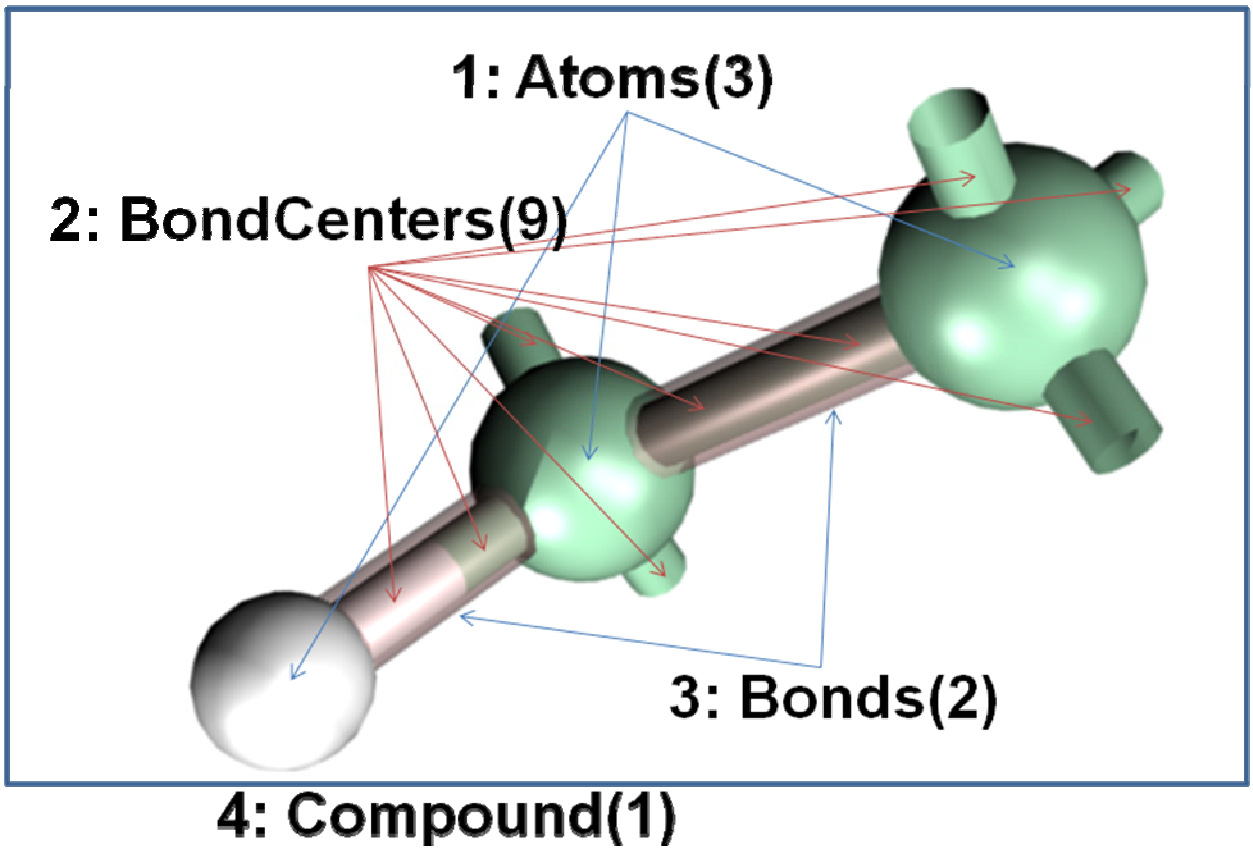


Figure 8-1: Parts of a Compound.

8.2.1 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.

8.2.2 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**.

8.2.3 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**.

8.3 Defining a New Molecule: Propane

Example 8-1: Complete Program for Defining and Simulating Propane.

```
#include "SimTKmolmodel.h"
#include "SimTKsimbody_aux.h" // for vtk visualization

#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(TinkerDuMMForceFieldSubsystem& dumm)
    {
        instantiateBiotypes();

        DuMM::ChargedAtomTypeIndex chargedAtomIndex(5000);

        dumm.defineChargedAtomType(
            chargedAtomIndex,
            "propane C1_or_C3",
            DuMM::AtomClassIndex(1), // "CT" type in amber
            -0.060 // made up
            );
```

```
        dumm.setBiotypeChargedAtomType( chargedAtomIndex,
Biotype::get("propane", "C1_or_C3").getIndex() );
        ++chargedAtomIndex;

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

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

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

int main()
{
    CompoundSystem system;
    SimbodyMatterSubsystem  matter(system);
    TinkerDuMMForceFieldSubsystem dumm(system);
    DecorationSubsystem     artwork(system);
```

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

    Propane::setAmberLikeParameters(dumm);

    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.updDefaultSubsystem().addEventReporter(new
VTKEventReporter(system,
        0.100));

    system.modelCompounds(); // finalize multibody system

    State state = system.realizeTopology();

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

```
inboard bond center
Biotypes
DuMMForceFieldSubsystem
```

8.4 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.

8.5 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.

8.5.1 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.

8.5.2 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.

8.6 Ring-closing Bonds

Because bonded structures are built up in tree-like fashion, which 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.

8.7 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.

8.8 Exercises

Exercise 8-1

Compile and run propane example.

Exercise 8-2

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

9

Getting More Information

9.1 The SimTK.org Website

The SimTK.org website, at https://simtk.org/, has the latest downloads, documentation, and source code for the entire SimTK core, including Molmodel.

9.2 Help Us Help You: Submitting Feature Requests and Bug Reports Online

9.2.1 How to submit Bug Reports and Feature Requests

Bug reports and feature requests can be submitted online at https://simtk.org/.

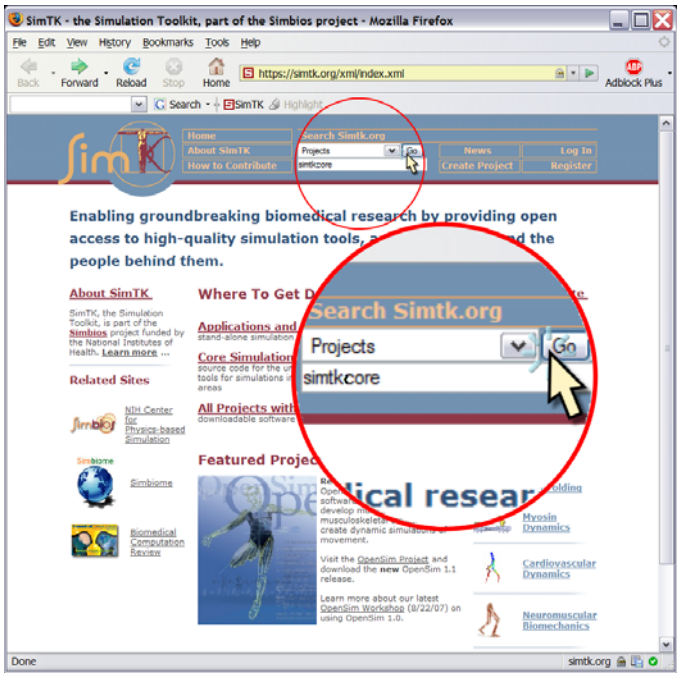


Figure 9-1: Searching for the "SimTKcore" project at SimTK.org

Figure 9-2: Selecting "SimTKcore" from project search results list.

Figure 9-3: Opening the Advanced options on the navigation bar.

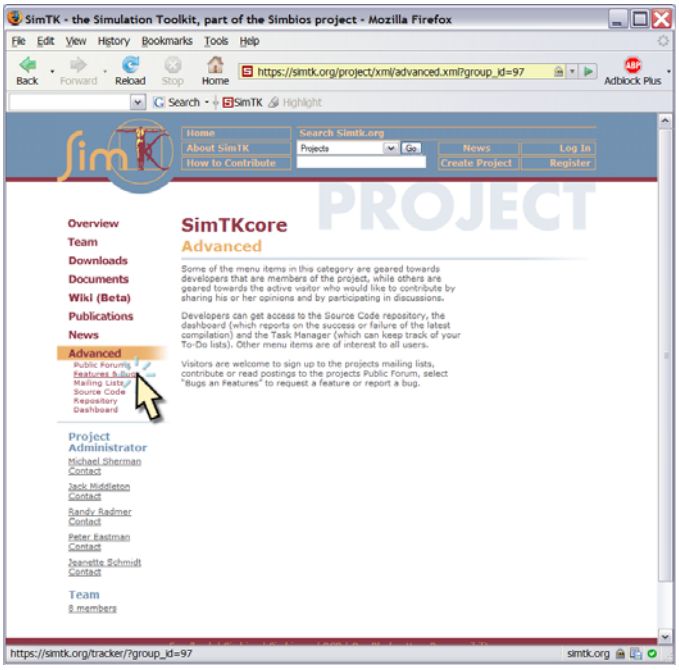


Figure 9-4: Selecting "Features & Bugs" page

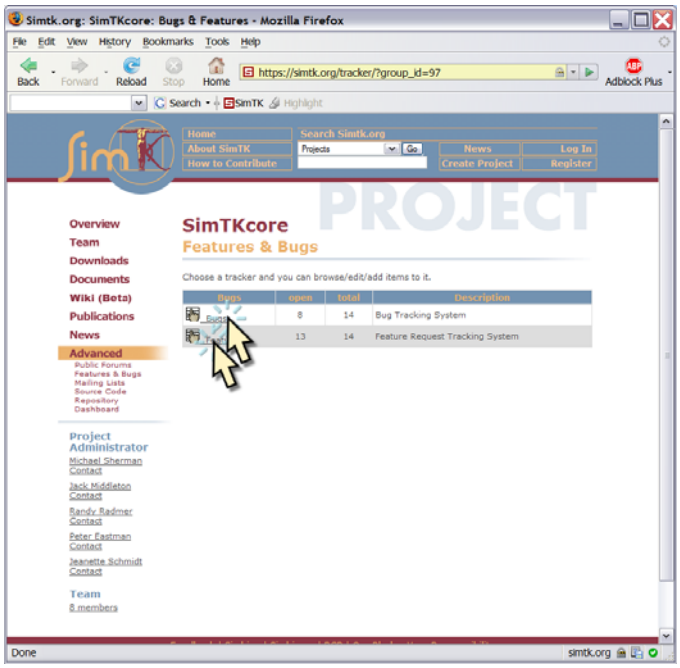


Figure 9-5: Choosing between Bugs or Features

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

A “Bug Report” is supposed to identify problems where the software is not working the way it is supposed to. A “Feature Request” is meant to suggest 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. So please try to remember which one you clicked.

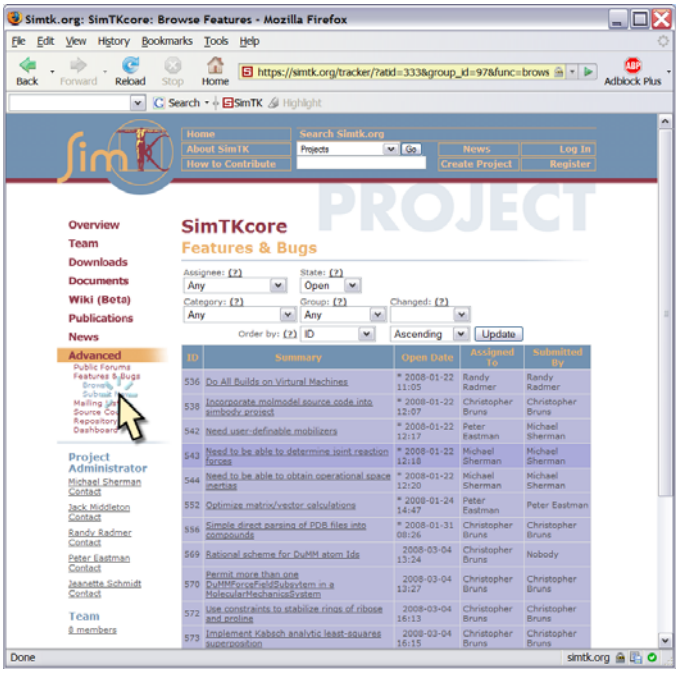


Figure 9-6: Choosing to submit a new Feature Request or Bug Report

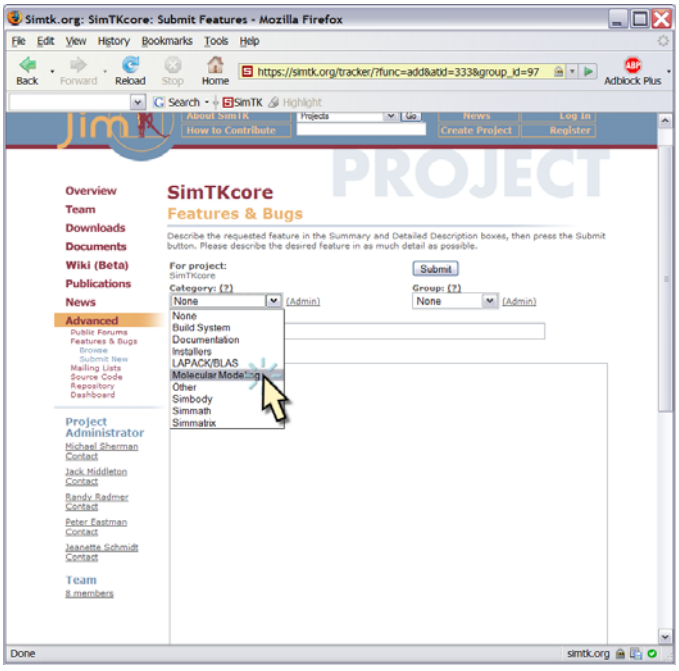


Figure 9-7: Selecting a Bug/Feature category.

Don't spend too much time worrying about the fields on the form you don't understand.

9.2.3 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. 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 use 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/

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

J. Wang, P. C. (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.

W. Schroeder, e. a. (2007). *The Visualization Tool Kit, version 5.* Retrieved from http://www.vtk.org/