MDL - User Guide

Authors: Trevor Cickovski, Chris Sweet and Jesús A. Izaguirre

June 1, 2009

Contents

1 Introduction **1**

2 Setting Things Up **3**
2.1 Prerequisites . . . 3
2.1.1 Required . . . 3
2.1.2 Optional . . . 4
2.2 Installation . . . 5

3 Simulation Protocols **6**
3.1 Importing Core Modules . . . 6
3.2 Physical System . . . 7
3.2.1 Requirements . . . 7
3.2.2 Other Attributes . . . 8
3.3 Forces and Force Fields . . . 9
3.4 Output . . . 11
3.4.1 Screen . . . 11
3.4.2 Files . . . 12
3.4.3 Plots . . . 12
3.5 Propagation . . . 14
3.5.1 Normal Mode Langevin . . . 15
3.6 Examples . . . 17
3.6.1 United Atom Butane in Vacuum for 1 ps, Leapfrog . . . 17
3.6.2 Solvated Bovine Pancreatic Tripsin Inhibitor (BPTI) running Langevin Impulse using cutoffs . . . 18
3.6.3 Alanine Dipeptide with OpenMM, Gromacs Topology and AMBER Force Fields . . . 19

4 Constructing New Propagators **20**
4.1 Propagator Factories . . . 20
4.2 Classes . . . 21
4.2.1 Modifiers . . . 22
4.3 Functions . . . 22

5 Constructing New Forces **24**

6 MDL Licensing **27**
6.1 Contact Information . . . 27

Abstract

Molecular Dynamics (MD) involves solving Newton's equations of motion for a system of atoms, using the resulting forces to propagate the system in time. Since the most computationally intensive calculations of MD simulations (i.e., pairwise force evaluations) yield great difficulty in modeling systems of reasonable size for a biologically relevant period of time, an important area of research is the development of efficient numerical methods for MD simulations. The Molecular Dynamics Lab (MDL) is designed for the prototyping, testing and debugging of these methods, built with the scripting language Python.

The user guide begins with a description of the features of MDL, followed by a guide to constructing MD simulations protocols for the purposes of testing numerical methods and interacting with external tools for data analysis. Following is a description of how to construct new propagators and force calculators. It concludes with some helpful examples, other necesssary background on MD, and software licensing.

Chapter 1

Introduction

MD simulations can be instrumental to understanding the short and long-term behavior of molecular systems, which holds implications for fields such as drug design, protein folding and the human genome project. Although researchers currently possess high-performance and optimized software running on parallel supercomputing clusters, it is extremely difficult to model systems of reasonable size for a biological time which exceeds the microsecond timescale, well short of desirable for protein folding which occurs on the second timescale. This is due mainly to two factors: (1) high-frequency motions (i.e., bond fluctuations) within a molecule which limit the propagation timestep, and (2) the quadratic algorithm for pairwise force computation.

Given these limitations which still exist with high performance computing, the continued development of efficient numerical methods for MD simulations is very important. We have seen for instance restraints such as SHAKE [34] and RATTLE [2] which afford larger timesteps by restraining fast frequency motions. Multiple timestepping propagators calculate different forces at different frequencies depending on how fast they vary. In Normal Mode Analysis [], motions are split into fast and slow frequencies, with fast frequencies oscillating around 'normal' values. Fast electrostatic methods such as Ewald [12] and PME [10] improve the quadratic complexity of evaluating coulombic interactions at a cost of some accuracy.

It thus makes sense to provide a tool which can above all facilitate construction of these methods, and the ability to test and debug with realistic biological systems and conduct parameter sweeps. But this can be difficult using a framework designed for performance because these tend to be written in languages such as FORTRAN or C which do not cater well to extensibility. Scripting languages make it easier to construct programs quicker and on the fly and are often interpreted, which avoids the need to reproduce machine code upon modifications. Python contains several features which cater well to testing and debugging such as dynamic typing and dynamic binding, and is also portable and readable. Moreoever there is an abundance of external tools built with Python which become interfacable to MDLab as a result.

This User Guide provides a summary for the operation of MDLab, including construction of simulation protocols, propagators and force calculators. MDL currently possesses the following capabilities:

1. **Compatibility with CHARMM force fields**. Construction of a physical system in MDL involves reading CHARMM [7] Protein Structure Files (PSFs) and Parameter (PAR) files. Files from the Protein Data Bank (PDB, [5]) can be used to populate atomic positions. Pairwise computations such as the Lennard-Jones potential which models both van der Waals and Pauli Exclusion force terms, and the electrostatic forces are customizable through evaluation algorithms, cutoffs and switching functions.

2. **Compatibility with GROMACS topology**. As an alternative to the CHARMM PSF, a user may set up the system *topology* (i.e., bonds, angles, etc.) using a topology file from Gromacs [4]. This topology file can be particularly useful if the user desires the Generalized Born [26] implicit solvent model for electrostatics, since parameters are provided.

3. **Compatibility with AMBER force fields**. Gromacs topology information is often coupled to AMBER force fields. MDL contains readers for these force fields, which can be used as an alternative to CHARMM.

4. **Construct a physical system**. MDL provides the ability for a user to control parameters such as simulation boundary conditions, Kelvin temperature, etc. along with easy access to observables such as the atomic position and velocity vectors, mass matrices, and cell basis vectors which are constructed as Numpy [25] arrays for optimized matrix-vector operations.

5. **Construct propagators and force calculators**. MDL provides several known MD propagators and evaluation of terms from CHARMM force fields. These exist as precompiled shared object binaries for maximum performance. Alternatively, propagators and force calculators can be built using constructs provided by Python modules within the MDL package along with mathematical operations on simulation data. MDL makes use of *factories* [1] which uniquely map Python strings to propagators and force calculators, enabling easy access from simulation protocols.

6. **Perform data analysis**. MDL provides several opportunities for data analysis. This can be done through file output, which includes binary DCD trajectory files, screen output, or Matlab [20]-compatible data charts. Through Gnuplot-py [15] or Matplotlib [21], MDL can plot observables such as temperature, pressure, total energy or potential terms, etc. which makes verification of propagation schemes easier; for example by ensuring conserved quantity fluctuations remain bounded. Finally, MDL can interact with the Java Molecular Viewer (JMV) of ProtoMol (see [22]), which can interactively visualize a system while running MDL commands in the background. MDL is actually part of a larger Problem Solving Environment (PSE, [14, 23]) ProtoMol which also contains a computational back end, for which MDL serves as a *scripting interface*. This is convenient since expansions to the computational back end become available in MDL.

7. **Compatibility with OpenMM**. OpenMM [13] provides a set of force calculators which can run on Graphical Processor Units (GPUs). These types of processors are specifically designed for processing large amounts of data quickly, and thus can improve performance by orders of magnitude for large-scale systems over a long period of time. Preliminary results indicate roughly a 500-fold speedup for these types of systems [9].

Chapter 2

Setting Things Up

MDL is distributed as a part of the ProtoMol framework, which is available open source on SourceForge (`http://www.sourceforge.net`). The latest version of the source code for MDL and the ProtoMol back end are available using Subversion [28], and can be checked out by running the following:

```
svn co https://protomol.svn.sourceforge.net/svnroot/protomol protomol
```

This will provide you with anonymous read access to the svn repository, from which you can obtain source code updates using `svn update`. The MDL framework is contained within the directory `protomol/mdl`, but is coupled to the ProtoMol computational back end in that the functionality of several C++ files has been wrapped for Python using SWIG [3], which also generates C function implementations as precompiled shared object libraries. To use MDL, you will need to install some external tools, which we outline below as prerequisites. Although there seems to be quite a few prerequisites, you may have some of them already and they are all relatively easy to install and do not themselves require other dependencies. So you should be able to have everything set up in a relatively short period of time.

Alteratively if you are using Windows, you can run the self-installing executable on the MDLab home page. This executable has all necessary features to run the software, plus the ProtoMol JVM. For systems other than Windows, or if you want to run on GPUs, you must install the tools below and compile the software from scratch to ensure compatibility with your particular OS flavor.

2.1 Prerequisites

Here we provide a set of tools which are required to execute MDL. Some of them may be installed on your machine already. However, they may not be the correct versions, and you should check to make sure that the version of each tool corresponds to the version specified below. Please install these tools in the order they appear.

2.1.1 Required

Python 2.5

The Python interpreter should come standard on most versions of *nix and some versions of Mac OS. If Python has been installed, you can check the version by starting the Python interpeter and passing the `-V` flag, like so:

```
python -V
```

The version printed should be minimally Python 2.5. If it is not, you will need to install an updated version. Python is available open source at `http://www.python.org`. Follow instructions under 'Download'. The safest bet is to download Python 2.5 since this is the version used by developers; although future versions should be back-compatible.

SCons

SCons (`http://www.scons.org`) is a software construction tool built entirely in Python which offers an alternative to Makefiles for compiling source code and linking software applications. Follow the instructions for downloading version 0.98. You will use this to compile the ProtoMol back end into a shared object binary which will dynamically link to the shared libraries imported by MDL.

SWIG

SWIG (Simplified Wrapper Interface Generator, [3] is an open source tool which can generate wrapper code for C and C++ entities in many different languages (many of them scripting languages), Python being one of them. SWIG does come standard on many versions of *nix, but to use MDL you will minimially need version 1.3.31. You can check your version number by executing:

```
swig -version
```

If you do not have at least 1.3.31, you can download SWIG from `www.swig.org`. Follow the instructions, and make sure you check C++ as your input language and Python as you target, and select your appropriate platform.

Numpy

Numerical Python (NumPy, `http://numpy.scipy.org/`) is available open source and provides a set of libraries implemented in Python for scientific computing. Most importantly, NumPy matrix-vector operations are powerful and optimized, which are critical for MD simulations. Follow the directions for basic installation; there is no need to include any prefixes as the built NumPy libraries will be deposited in your Python installation.

2.1.2 Optional

Gnuplot-py or Matplotlib

Gnuplot-py is a plotting library built in Python which interfaces to the Gnuplot utility and can plot data interactively. On *nix, to make sure you have Gnuplot installed execute `which gnuplot` and make sure there is an application installed. Then follow the instructions from `http://gnuplot-py.sourceforge.net/` to complete the installation. Once again, install in the default location so that the installation will deposited in your Python installation.

Alternatively to Gnuplot-py, you can choose to install Matplotlib if you are certain that will most often be your plotting tool. Since however plotting with Matplotlib is not the default, we recommend also installing Gnuplot-py since it's fast and easy, but if you choose to only install Matplotlib you will need to indicate this in your simulation protocols. Matplotlib is available open source at `http://matplotlib.sourceforge.net`.

Java Molecular Viewer

The ProtoMol JVM is available open source on Chris Sweet's website `http://www.nd.edu/~csweet1`. Once you have installed and run this software, the default port will be 52753. MDL simulations submit data to this port automatically, so the molecule can be viewed interactively as a simulation runs.

OpenMM and CUDA

OpenMM provides a set of libraries for molecular modeling which can be invoked transparently. Specifically, the MDL environment uses OpenMM for the expensive force calculation, which generally consumes the highest percentage of execution time for a MD simulation. These libraries can be executed on graphical processing units (GPUs) which can yield extensive savings in execution time. These libraries are available at `https://simtk.org/home/openmm`. After downloading them, you can install MDL with the interface to OpenMM to achieve these improvements in speed. You also must have an NVidia (`https://www.nvidia.com`) graphics card installed.

CUDA (Compute Unified Device Architecture) provides a computing environment which processes all data on NVidia GPUs. This engine is available at `http://www.nvidia.com/cuda`. OpenMM submits computation to the CUDA engine, which in turn performs the computation on GPUs. With both OpenMM and CUDA installed, you can achieve the improvements in speed provided by the processing power of GPUs.

2.2 Installation

Once you have these tools, you are ready to install and run MDL. The first step will be to run (from the MDL root directory):

```
scons gui=1
```

You do not have to include the `gui=1` flag if you know for sure you will not use the ProtoMol JMV. However, it is beneficial to include this if you are not sure, since if you later change your mind you will have to recompile. Alternatively, if you have installed OpenMM and CUDA and would like to run on GPUs, type:

```
scons openmm=cuda
```

This will interface to the OpenMM libraries for force computation, and you can subsequently run these calculations on your NVidia GPUs.

Once the compilation is finished, you are ready to use MDL. The three basic types of Python scripts that MDL is built to handle are simulation protocols, propagator classes and functions, and force classes. We recommend reading the next section on simulation protocols and actually running MDL per the instructions in the next section, to get accustomed to the organization and procedures of the MDL framework.

Chapter 3

Simulation Protocols

The basic facility for testing numerical methods will be an MDL *simulation protocol*, which involves setting up a biological system and propagating it with time. Protocols can be run through a standard Python interpreter. MDL contains several example simulation protocols within the `simulations/` folder, as Python scripts runnable through:

```
./RunMDL simulations/<file>.py
```

You may also set up your own simulation protocol but the structure will be similar to the examples provided. We will now run through the 'typical' structure of an MDL simulation protocol.

3.1 Importing Core Modules

A Python module is implemented within its own Python script and contains classes and functions applicable to a specific task. MDL provides five 'core' Python modules which are each responsible for a separate portion of an MD simulation:

1. **Physical**: The physical system, consisting of atomic positions and velocities, mass and inverse mass matrices, boundary conditions and cell basis vectors (if applicable), Kelvin temperature, and biological time.
2. **IO**: Provides functionality for reading and writing data. This includes readers for file input when constructing the physical system, and output for data analysis including plots, screen, and files.
3. **Forces**: Contains the atomic force vector and each type of energy: kinetic and potential, including individual components (two-atom bond fluctuation potential, three atom angle, etc.).
4. **ForceField**: Contains a group of forces to evaluate. These forces can be CHARMM forces (bond, angle, dihedral, improper, Lennard-Jones and electrostatic), or Python-prototyped force calculators.
5. **Propagator**: Provides methods for propagating a system with time.

An MD simulation protocol will most likely need everything from these five modules, so the first step will typically be to import all of their functionality using the Python interpreter. This is done by importing the entire `MDL` module:

```
from MDL import *
```

3.2 Physical System

An MD simulation will always consist of some sort of biological system, that is being propagated with time. MDL provides several example systems in the `data/` directory. These include:

1. Alanine dipeptide in vacuum (22 atoms, includes both CHARMM and AMBER/Gromacs).
2. Solvated alanine (22 atom solute, 445 water molecules).
3. Block alanine dipeptide in vacuum (22 atom block structure).
4. Two argon systems (ideal gas, 280 and 400 molecules).
5. Two solvated bovine pancreatic tripsin inhibitor (882 and 898 atom solute, 73 and 4461 water molecules).
6. Decalanine in vacuum (66 atoms).
7. United atom butane in vacuum (4 atoms).
8. Two water boxes (72 and 216 water molecules).
9. The WW domain (551 atoms).

3.2.1 Requirements

The physical system describes attributes of the molecular system such as structure, coordinates, boundary conditions, pairwise exclusions, and temperature. Many of these attributes are defined within input files provided in these example folders, in addition you can download new example systems from the Protein Data Bank. To define a physical system in MDL, you will first want to declare a Python object to contain this information. In this case we use the variable name `phys`, but you can use any valid Python variable name to reference this object:

```
phys = Physical()
```

For initial coordinates, structure, and force field parameters you will want to use input files to populate the object. To access functionality for file I/O, you also must define an `IO` object:

```
io = IO()
```

Coordinates for positions and velocities can then be specified using PDB files (from the Protein Data Bank) or XYZ files (a simple format consisting of four columns - atom name, and (x, y, z) coordinates:

```
io.readPDBPos(phys, "examples/bpti_water_1101/bpti.pdb")
io.readXYZVel(phys, "examples/bpti_water_1101/bpti.vel.xyz")
```

Initializing atomic positions is a requirement. By invoking `readPDBPos` or `readPDBVel`, the `positions` data member of the passed `Physical` object will be populated. The `positions` data member is a NumPy array of floating point values, so it also can be populated manually if so desired. Explicitly initializing atomic velocities is not a requirement, if an initial Kelvin temperature is provided then velocities will be initialized using a Maxwell distribution with mean zero and standard deviation equal to $\frac{k_B T}{m}$. This can be specified by referencing the `temperature` attribute of `Physical`. In addition, the `seed` data member can be set to ensure random number consistency across simulations (it is by default 1234):

```
phys.temperature = 300
phys.seed = 1769
```

CHARMM parameter and PSF files specify structure and extra information such as spring constants, equilibrium lengths for bonds, etc. These are required and can only be populated through files:

```
io.readPSF(phys, "examples/bpti_water_1101/bpti.psf")
io.readPAR(phys, "examples/bpti_water_1101/bpti.par")
```

Alternatively, you can input a Gromacs topology file and AMBER force field, using the `readGromacs()` method of `IO`. This function accepts the `Physical` object, the name of the Gromacs topology file as a string, and the directory holding the AMBER force fields as a string. The example below uses AMBER96:

```
io.readGromacs(phys, "data/alanine_gromacs/alanine.top",
                     "data/alanine_gromacs/ffamber96/")
```

This is the minimum necessary to set up an MDL physical system, since default values are provided for other parameters. However, you can manually specify these as well by directly referencing member attributes of the `Physical` object.

3.2.2 Other Attributes

1. **Boundary Conditions**. The `bc` data member of `Physical` is a Python string which can either be set to `Periodic` (default) or `Vacuum`. Periodic boundary conditions implement a wraparound at the edges of the simulation box, so that when atomic pairwise distances are calculated the closer of either straight-up Euclidean or wraparound will be used. This is useful for modeling the effects of an encompassing bulk solvent. Vacuum simply computes Euclidean distance with no wraparound:

   ```
   phys.bc = "Periodic"         # Boundary conditions
   ```

2. **Pairwise Exclusions**. Certain nonbonded pairwise force calculations can be excluded if the same pairwise calculation is performed for a bonded potential. The `exclude` attribute specifies the level of exclusion, which is a Python string that can set to one of the following: `1-2` (exclude all directly covalently bonded atom pairs), `1-3` (1-2 plus exclude all atom pairs which share a covalently bonded atom), `1-4` (1-3 plus exclude all atom pairs within two covalent bonds of each other), `scaled1-4` (default, 1-4 but instead of completely excluding atom pairs within two covalent bonds, scale the nonbonded term down):

   ```
   phys.exclude = "scaled1-4"   # Pairwise exclusion
   ```

3. **Cell Size**. Internally, when computing pairwise forces MDL uses a routine from ProtoMol which divides simulation space into *cells*, and a *cell manager* keeps track of the atoms within each cell. Since it is inefficient to recompute all pairwise distances at every simulation step, only pairs of atoms within neighboring cells are considered. This will only make sense if a *cutoff* is used for pairwise calculations (explained later), and then maximum efficiency can be obtained by setting the `Physical` attribute to twice the cutoff:

   ```
   phys.cellsize = 4            # Periodic cell size
   ```

4. **Cell Basis Vectors**. When using periodic boundary conditions, the dimensions of the periodic box are by default obtained by using the minimum and maximum (x, y, z) coordinates of the initial configuration, but this can be specified manually by providing a set of three cell basis vectors which span the periodic box, plus an origin. These are contained within the `Physical` attributes `cB1`, `cB2`, `cB3`, and `cO` which are three-element numpy arrays of floating point values:

```
phys.cB1[0] = 29
phys.cB1[1] = 0
... etc ...
```

5. **Linear and Angular Momentum**. The effects of collective system linear and angular momentum can be removed at a given frequency using the `remcom` (actually references the center of mass motion) and `remang` attributes of `Physical`. These can hold values of -1 (never), 0 (once at the beginning), or an integer frequency in simulation steps. Biologically, for a single-timestepping propagation this time will be equal to the frequency times the timestep Δt:

```
phys.remcom = 2
phys.remang = 2
```

3.3 Forces and Force Fields

During any MDL simulation the atomic force vector and energy values are obtainable through an MDL `Forces` object, which can be constructed similarly:

```
forces = Forces()
```

Computed atomic forces are stored within the attribute `force`, which is a numpy array of floating point values. In addition, energies are stored within the attribute `energies`, which provides the following methods for accessing energy values:

bondEnergy(): Bond fluctuation potential.
angleEnergy(): Potential resulting from deviations of three-atom angles from equilibrium values.
dihedralEnergy(): Potential resulting from deviations of four-atom dihedrals from equilibrium values.
improperEnergy(): Potential resulting from deviations of four-atom impropers from equilibrium values.
ljEnergy(): Lennard-Jones potential.
coulombEnergy(): Electrostatic potential.
shadowEnergy(): Shadow potential, if calculated.
potentialEnergy(): Total potential energy, which is the sum of the above terms.
kineticEnergy(): Total kinetic energy.
totalEnergy(): Total energy (potential+kinetic).

The `energies` data member also provides methods to accumulate an energy quantity into one of the above terms, with **addBondEnergy(int)**, **addAngleEnergy(int)**, etc. These methods are useful when constructing Python forces, to designate which energy term is effected by the resulting potential.

The `Forces` object contains calculated force and energy data, and a `ForceField` object contains a specification of which forces to evaluate. A `ForceField` can be constructed to evaluate CHARMM forces, which include the following:

* Two-atom bond forces, resulting from deviations from equilibrium lengths.
* Three-atom angle forces.
* Four-atom dihedral and improper forces.
* Van der Waals forces (using LennardJones evaluation).
* Electrostatic or Coulombic forces.

If a `Forces` object has been defined, a `ForceField` can be constructed by invoking its member function `makeForceField()`, passing a `Physical` object and the Python string `charmm` to evaluate CHARMM forces. You can do this even if you used AMBER force field parameters; everything is converted transparently. To evaluate a subset (i.e. noble gas simulations will not have bonded forces) or a different set of forces, simply ignore the second argument:

```
forces = Forces()                              # Declare a Forces object
ff = forces.makeForceField(phys, "charmm")  # Return a ForceField
```

You can evaluate a subset of CHARMM forces by using `ForceField` member functions `bondedForces` and `nonbondedForces`, passing Python strings with uniquely identifying characters. For bonded forces, “b”=bond, “a”=angle, “d”=dihedral and “i”=improper. For nonbonded, “l”=Lennard Jones and “c”= electrostatic:

```
forces = Forces()                              # Declare a Forces object
ff = forces.makeForceField(phys)               # Return a ForceField
ff.bondedForces("bd")                          # Bond and dihedral
ff.nonbondedForces("l")                        # Lennard Jones
```

By default, all pairwise forces are evaluated using a quadratic *direct* algorithm, meaning all pairs are considered. For pairwise forces, MDL provides the ability to change the *algorithm* for force evaluation, to `Cutoff` for example. In the case of `Cutoff`, after a certain pairwise distance zero is assumed for the resulting potential, eliminating the need to evaluate pairwise force between atoms beyond the distance and saving performance. Parameters such as the algorithm and cutoff value are accessible through the `params` data member of `ForceField`, which is a Python dictionary:

```
# Cutoff the Lennard-Jones potential after 6.5 angstroms
ff.params = {'LennardJones':{'algorithm':'Cutoff',
                             'cutoff':6.5}}
```

If both Lennard-Jones and Coulomb forces are being evaluated and the same parameters are used for both, you can save performance by setting both sets of parameters at the same time. In this case, atomic pairs will be evaluated once for both, resulting in considerable performance savings. This is also the default:

```
# Cutoff the Lennard-Jones and electrostatic potential after 6.5 angstroms
ff.params = {'LennardJonesCoulomb':{'algorithm':'Cutoff',
                                    'cutoff':6.5}}
```

An abrupt dropoff in the potential at the cutoff values can cause numerical instabilities. This can be avoided by *smoothing* the force to zero at the cutoff value. MDL thus provides *switching functions* which can be applied to the potential. The following switching functions are available for pairwise forces:

1. `Universal` - no switching, only possiblity for direct calculation
2. `Cutoff` - abruptly cutoff the value, default for cutoff
3. `C1` - continuous first derivative
4. `C2` - continuous second derivative
5. `Cn` - flexible continuity

In addition, there are a few extra parameters that you can specify to customize your pairwise force evaluation:

1. `blocksize` - for `SimpleFull` only, provides the number of atoms to evaluate in each cycle of a nested loop. This is just for optimization and does not affect the result at all, but when used properly can reduce page faults and cache misses.
2. `cutoff` - for `Cutoff` only, provides the cutoff value in angstroms.
3. `switchon` - distance in angstroms to turn on switching. The default is zero.
4. `switchoff` - distance in angstroms to turn off switching. Defaults the cutoff.
5. `order` - for `Cn` switching function only, the continuity.

3.4 Output

MD numerical methods can often be tested by viewing the behavior of an observable with time. To supply this ability for analysis purposes, MDL provides three types of formats for outputting observables: files, plots, and the screen. Files generated by MDL are all formatted as charts and are Matlab compatible, so they can be supplied as input to one of Matlab's many commands for graphing information.

3.4.1 Screen

MDL screen output consists of the step number, biological time, total energy, Kelvin temperature, and volume in cubic angstroms. The frequency at which this takes place is controllable through the data member `screen` of the `IO` class. So assuming we have an instance of class `IO`, we can set this frequency:

```
io.screen = 2   # Screen output every two steps
```

To produce output similar to the following:

```
Step: 0,Time: 0.000[ps],TE:256.3166[kcal/mol],T:308.2064[K],V:21117.72[AA^3]
Step: 2,Time: 0.040[ps],TE:198.0234[kcal/mol],T:302.3148[K],V:21117.72[AA^3]
Step: 4,Time: 0.080[ps],TE:198.0834[kcal/mol],T:284.0183[K],V:21117.72[AA^3]
Step: 6,Time: 0.120[ps],TE:198.2843[kcal/mol],T:263.5827[K],V:21117.72[AA^3]
Step: 8,Time: 0.160[ps],TE:198.2702[kcal/mol],T:256.1271[K],V:21117.72[AA^3]
...
```

3.4.2 Files

In addition, the `IO` class contains a data member dictionary `files` which maps uniquely identifying Python strings to Python tuples containing file names and frequencies. The default frequency for all types of file output is -1 (never), but you can override this for all desired file outputs. Usable string names include:

1. **energies**: All system energies as a chart including step number.
2. **dcdtrajpos**: A DCD trajectory file of atomic positions.
3. **dcdtrajvel**: A DCD trajectory file of atomic velocities.
4. **xyztrajpos**: An XYZ trajectory file of atomic positions.
5. **xyztrajvel**: An XYZ trajectory file of atomic velocities.
6. **xyztrajforce**: An XYZ trajectory file of atomic forces.
7. **gui**: Set this frequency if using the ProtoMol Java Molecular Viewer (JMV). This provides the step frequency at which to send positional information to this GUI. In this case the 'filename' is a name for the simulation which appears in the GUI. The default port for submitting information is 52753, the same default for the GUI when running on the localhost.

Thus for example, if you wanted to output energies every 2 steps and an XYZ position trajectory file every 5, you can modify the dictionary as follows:

```
io.files = {'energies':('alanine.energies', 2),
            'xyztrajpos':('alanine.pos.xyz', 5)}
```

Finally, instantaneous file I/O can be performed for atomic positions or velocities by invoking one of the following methods of class `IO` in your simulation protocol, passing the desired filename as the parameter:

1. **writePDBPos** - PDB file of atomic positions
2. **writeXYZPos** - XYZ file of atomic positions
3. **writePDBVel** - PDB file of atomic velocities
4. **writeXYZVel** - XYZ file of atomic velocities

For example, if you wanted to write the final atomic positions and velocities, you could specify at the end of your Python simulation protocol script:

```
io.writePDBPos('alanine.pos.fin.pdb')
io.writePDBVel('alanine.vel.fin.pdb')
```

3.4.3 Plots

MDL plots make use of one of two possible external libraries: Gnuplot-py or Matplotlib. Each are available open source and Gnuplot-py is the default as Gnuplot itself comes standard on many *nix platforms. To use Matplotlib instead, you can set the `useMPL` data member of `IO` to True:

```
io.useMPL = True
```

Several observables can be plotted in MDL, once again using uniquely identifying strings and a member Python dictionary `plots` of class `IO`. The following observables can be plotted (these strings are used as

keys in the dictionary):

1. **temperature**: Kelvin temperature.
2. **pressure**: System pressure.
3. **volume**: System volume in cubic angstroms.
4. **bondenergy**: Bond fluctuation potential.
5. **angleenergy**: Potential resulting from deviations of three-atom angles from equilibrium values.
6. **dihedralenergy**: Potential resulting from deviations of four-atom dihedrals from equilibrium values.
7. **improperenergy**: Similar, for four-atom impropers.
8. **ljenergy**: Lennard-Jones potential.
9. **coulombenergy**: Electrostatic potential.
10. **shadowenergy**: Shadow potential, if calculated.
11. **potentialenergy**: Total potential energy, which is the sum of the above terms.
12. **kineticenergy**: Total kinetic energy.
13. **totalenergy**: Total energy (potential+kinetic).

So for example, if you wanted to plot temperature every 10 steps and total energy every 5, you could set the `plots` member dictionary of `IO` as follows:

```
io.plots = {'temperature':5,
            'totalenergy':10}
```

Subsequently, data will be plotted with either Gnuplot-py or Matplotlib as designated by the user. The plots will occur interactively, as the simulation proceeds all plots will be updated with time at their desired frequencies. Figure 3.1 shows total energy plotted through Matplotlib while running a simulation of united-atom butane in vacuum.

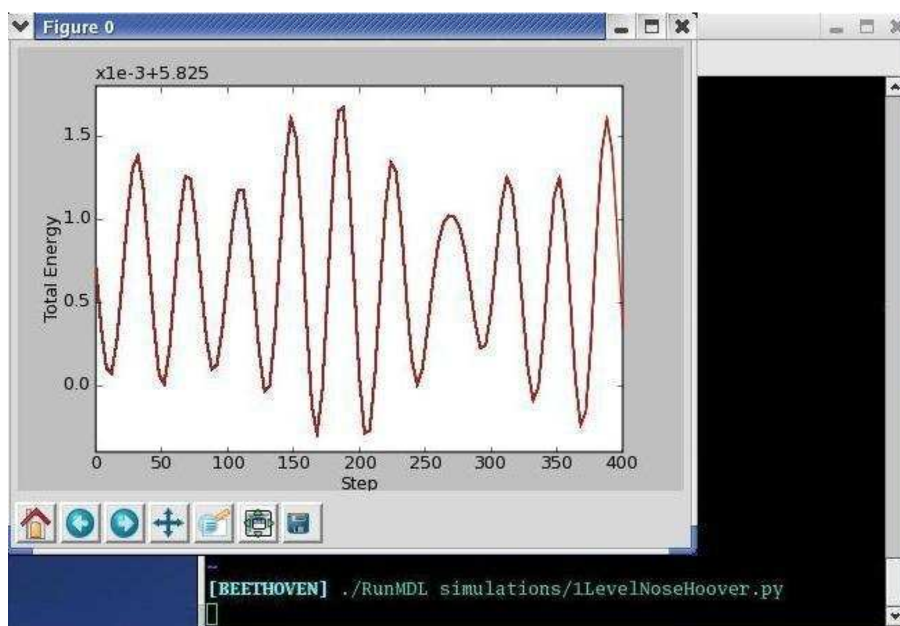

Figure 3.1: Interactive plot of total energy while running an MDL simulation of united-atom butane.

3.5 Propagation

After setting up a physical system and a set of forces to evaluate, the system can be propagate with time by applying a *propagator* to the positions and velocities. A propagator updates positions and velocities using a *timestep* Δt. Each application of the propagator updates a system over biological time Δt, so *n* applications propagates over time $n\Delta t$. Δt must of course be chosen carefully, a very low value results in longer simulations but high values may result in instabilities or unacceptable losses in accuracy by failing to account for fast frequency motions. Certain propagators may make acceptable approximations and as a result allow for longer timesteps.

Your first step will be to construct an instance of the MDL `Propagator` class, passing three objects as parameters: a `Physical`, `Forces`, and `IO`:

```
prop = Propagator(phys, forces, io)
```

Subsequently, the system can be propagated by invoking the `propagate` member function of class `Propagator`. For single-timestepping (STS) propagation, the `propagate` method accepts as parameters a uniquely identifying Python string for the propagator name, an integer for the number of steps to run, a floating point value for the timestep, and a `ForceField` object for evaluation. For example, to propagate for 1 ps using a timestep of 0.5 fs using the Leapfrog [18] method (conserving number of atoms, volume, and total energy):

```
prop.propagate(scheme="Leapfrog", steps=2000, dt=0.5, forcefield=ff)
```

This will update the `Physical` and `Forces` objects that were passed to the constructor of this `Propagator`. `propagate` will also invoke any output routines specified in the `IO` object passed to this constructor. There is one more optional parameter for the `propagate` method, a Python dictionary `params`. Some propagators can accept extra parameters, and some do not. The Leapfrog method for example does not need any other parameters, but Langevin dynamics (conserving number of atoms, volume, and temperature) requires a damping coefficient γ for its frictional and random collision terms, and also a Kelvin temperature for the collision term. MDL provides an integrator `LangevinImpulse` [27] for this purpose:

```
prop.propagate(scheme="LangevinImpulse",
               steps=2000,
               dt=0.5,
               forcefield=ff,
               params={'gamma':3.0,
                       'temp':300})
```

In either case defaults are always provided, so you can always invoke a propagation scheme without providing extra parameters and use default values. MDL provides the following STS propagation schemes, with corresponding parameters, data types and default values:

1. **BBK**: The Brooks-Brunger-Karplus method [8]. *Parameters:* **temp** *(Kelvin temperature, floating point, 300),* **gamma** *(damping coefficient, floating point, 91),* **seed** *(random number seed, integer, 1234).*

2. **Leapfrog**: The Leapfrog (or Velocity Verlet) method [18]. *Parameters: NONE*

3. **PLeapfrog**: The Position Verlet method [31]. *Parameters: NONE*

4. **LeapfrogTruncatedShadow**: The Leapfrog method, which uses the solution of a nearby *Shadow Hamiltonian* [11]. *Parameters: NONE*

5. **LangevinImpulse**: The Langevin Impulse method [27]. *Parameters:* **temp** *(Kelvin temperature, floating point, 300),* **gamma** *(damping coefficient, floating point, 91),* **seed** *(random number seed, integer, 1234).*

6. **DMDLeapfrog**: Extension of the Leapfrog method to Langevin dynamics. *Parameters: Same as LangevinImpuse, plus* **iter** *(number of velocity updates before calculating the dissipative term, integer, 5).*

7. **NoseNVTLeapfrog**: The Nose Hoover method [24, 19]. *Parameters:* **temp** *(Kelvin temperature, floating point, 300),* **inertia** *(thermal inertia, floating point, 0.5),* **bathpos** *(heat bath center of mass location, floating point, 1.0).*

8. **CGMinimizer**: Non-linear conjugate gradient minimizer.*Parameters:* **alpha** *(Point where decrease is significant, floating point, 0.001),* **beta** *(Point where directional derivative decrease is significant, floating point, 0.05),* **restart** *(restart interval, integer, 0).*

9. **NumericalDifferentiation**: Used to check force implementations by making a second approximation to forces by perturbing positions by some small value, recomputing potential energy and then using a Taylor series to estimate the force value. Outputs the force error and Hessian. *Parameters:* **epsilon** *(Amount of perturbation, floating point, 1.0)*

10. **OpenMM**: Runs Langevin dynamics, computing forces using OpenMM. *Parameters: Same as Langevin-Impulse, plus* **HarmonicBondForce** *(specifies whether or not to compute bond potential, boolean, True),* **HarmonicAngleForce** *(specifies whether or not to compute angle potential, boolean, True),* **RBDihedralForce** *(specifies whether or not to compute dihedral potential, boolean, True),* **Periodic-Torsion** *(specifies whether or not to compute improper potential, boolean, True),* **NonbondedForce** *(specifies whether or not to compute pairwise forces, boolean, True),* **GBSAEpsilon** *(epsilon parameter for Generalized Born, floating point, 1.0),* **GBSASolvent** *(solvent dielectric constant for Generalized Born, floating point, 78.3),* **commotion** *(frequency to remove center of mass motion, integer, 0)*

Each of the above STS integrators evaluates one set of forces per step. In some cases it may be desirable to evaluate two different sets of forces at different frequencies, for example if one set is more slowly varying than the other. This can be done by using a chain of one or more *multiple timestepping* (MTS) propagators, sequentially evaluating faster varying forces at larger frequencies and the fastest using an STS propagator, terminating the chain.

3.5.1 Normal Mode Langevin

MDL contains several built-in propagators to perform *normal mode analysis*. The frequencies of the normal modes of a system are determined by the eigenvalues of the mass-reweighted Hessian matrix (contains second derivatives of the potential). The idea of normal mode analysis is to divide a system into slow and fast frequency modes, propagating the former with a propagator such as Langevin Impulse and the

latter using a more efficient Brownian dynamics, relaxing their values around local energy minima. The Normal Mode Langevin [29] method does this using a three-level MTS propagation: with the outermost MTS propagator rediagonalizing the mass-reweighted Hessian, an inner MTS propagator running Langevin dynamics on the slow modes, and the innermost STS propagator performing energy minimization using mass-reweighted steepest descent. In this case you will need three `ForceField` objects, assume they are bound to variable names `ff`, `ff2` and `ff3`. For MTS propagation, the member function `propagate` of class `Propagator` accepts a `cyclelength` parameter, which will just be an integer for a two-level scheme, or a Python list for three or more levels:

```
prop.propagate(scheme=['NormalModeDiagonalize',
                        'NormalModeLangevin',
                        'NormalModeMinimizer'],
               steps=20,
               cyclelength=[1,1],
               dt=4.0,
               forcefield=[ff, ff2, ff3],
               params={'NormalModeDiagonalize':{'reDiag':100},
                        'NormalModeLangevin':{'numbermodes':22,
                                              'gamma':80,
                                              'seed':1234,
                                              'temperature':300},
                        'NormalModeMinimizer':{'minimlim':0.5}})
```

The outermost `NormalModeDiagonalize` accepts a frequency to rediagonalize the mass-reweighted Hessian, the number of minimization steps. `NormalModeLangevin` accepts parameters similar to `LangevinImpulse` (the Kelvin temperature, damping coefficient and random seed), and in addition accepts a number of fast frequency modes (in this case, 22). For the `NormalModeMinimizer`, a threshold is specified for the potential energy deviation. Energy minimization will stop after this point, and the resulting conformation will be used as the most probable for fast modes.

3.6 Examples

Here we provide two examples of complete MDL simulation protocol scripts. I n addition, several more are available in the `simulations/` directory in the MDL framework.

3.6.1 United Atom Butane in Vacuum for 1 ps, Leapfrog

```
from MDL import *

# PHYSICAL SYSTEM
phys = Physical()
io = IO()
io.readPDBPos(phys, "data/UA_butane/UA_butane.pdb")
io.readPSF(phys, "data/UA_butane/UA_butane.psf")
io.readPAR(phys, "data/UA_butane/UA_butane.par")
phys.bc = "Periodic"
phys.cellsize = 6.5
phys.temperature = 300

# FORCES
forces = Forces()
ff = forces.makeForceField(phys, "charmm")


# OUTPUT
# PLOT KINETIC ENERGY EVERY 4 STEPS
# GENERATE SCREEN OUTPUT EVERY 2 STEPS
io.plots = {'kineticenergy':4}
io.screen = 2


# PROPAGATION
prop = Propagator(phys, forces, io)
prop.propagate(scheme="Leapfrog", steps=200, dt=0.5, forcefield=ff)
```

3.6.2 Solvated Bovine Pancreatic Tripsin Inhibitor (BPTI) running Langevin Impulse using cutoffs

```
from MDL import *

# PHYSICAL
phys = Physical()
io = IO()
io.readPDBPos(phys, "data/bpti_water_1101/bpti.pdb")
io.readPSF(phys, "data/bpti_water_1101/bpti.psf")
io.readPAR(phys, "data/bpti_water_1101/bpti.par")
phys.bc = "Periodic"
phys.cellsize = 5
phys.exclude = "scaled1-4"
phys.temperature = 0
phys.seed = 7536031

# FORCES
forces = Forces()
ff = forces.makeForceField(phys, "charmm")
ff.params['LennardJonesCoulomb'] = {'algorithm':'Cutoff',
                                    'switching':['C2', 'C1'],
                                    'cutoff':8.0}

# OUTPUT
# WRITE FILE bpti.energies EVERY 4 STEPS
# GENERATE SCREEN OUTPUT EVERY 2 STEPS
io.files = {'energies':('bpti.energies',4)}
io.screen = 2

# EXECUTE
prop = Propagator(phys, forces, io)
gamma = prop.propagate(scheme="LangevinImpulse", steps=2000, dt=0.1,
                       forcefield=ff,
                       params={'LangevinImpulse':{'gamma':0.5}})
```

3.6.3 Alanine Dipeptide with OpenMM, Gromacs Topology and AMBER Force Fields

```
from MDL import *

# PHYSICAL
phys = Physical()
io = IO()
io.readPDBPos(phys, "data/alanine_gromacs/alanine.pdb")
io.readGromacs(phys, "data/alanine_gromacs/alanine.top",
                     "data/alanine_gromacs/ffamber96/")
phys.bc = "Vacuum"
phys.temperature = 10


# FORCES
forces = Forces()
ff = forces.makeForceField(phys)
ff.gbsa = True

# OUTPUT
io.screen = 2

# EXECUTE
prop = Propagator(phys, forces, io)
gamma = prop.propagate("OpenMM", steps=2000, dt=0.5, forcefield=ff)
```

Up to this point, all simulation protocols have used predefined propagation schemes. Each of these propagation schemes was referenced using a uniqely identifying Python string, and is implemented as pre-compiled machine code for maximum efficiency. However, the central purpose of MDL is to test *new* numerical methods, and we now discuss (1) how to construct new propagators and force calculators using pure Python and (2) how to equate a uniquely identifying string with the new numerical method and couple this as a key within the internal MDL propagator and force factories.

Chapter 4

Constructing New Propagators

MDL provides a great deal of freedom in terms of acceptable implementations of new propagation schemes. In particular, you can choose the programming style with which you are most familiar; if you prefer procedural programming you can construct a propagator as a Python function, and if you like object-oriented design you can also construct a propagator as a Python class. Each are recognizable by the MDL propagator factories, as long as the new propagator has a uniquely defined Python string which does not serve as the key for any other propagator.

4.1 Propagator Factories

Internally, MDL propagator factories map Python strings (keys) to *initialization methods*; that is, a method for constructing the new propagator. In the case of Python functions, this is just a Python function handle; in the case of Python classes, this is the constructor of the class. In the `src/propagators` directory of the MDL framework, there are two subdirectories, `classes` and `functions`. The propagator factory automatically searches these directories for Python-prototyped propagation schemes, and loads all available modules. Thus if you are constructing a new propagator in Python, you will want to include its module in either `classes` or `functions`, depending on which implementation method you choose. There are several example Python-prototyped propagators in these directories. In the `classes` folder, there are implementations of Leapfrog [18], Brünger-Brooks-Karplus (BBK, [8]), the Hybrid Monte Carlo (HMC, []) sampling method, Nose-Poincaré [6], and Recursive Multiple Thermostatting [30], and the MTS propagator Verlet-I/r-RESPA [17, 16, 33] (also known as *Impulse*). For `functions`, we have included BBK, Leapfrog, Impulse, a simplified version of Takahashi-Imada [32], and a velocity scaler which runs Velocity Verlet, but includes one final scaling of velocities to keep the average kinetic energy over all atoms constant. These provide a wide range of examples which you can use as a guide when constructing your own propagators. Most importantly however, there are two Python variables which you must define someplace in your new module: `name` and `parameters`, in order for the factory to recognize your propagator. Set `name` equal to the uniquely identifying Python string, and set `parameters` equal to a Python tuple which alternates Python strings for the name of the variable, and default values (see one of the examples for help). For propagators implemented as Python classes, these variable names will become bound as member variables of the class. For Python functions, you will still need to pass them as formal parameters. For instance, if you look at `BBK.py`, which runs Langevin dynamics and thus needs the same parameters as Langevin Impulse (a temperature, frictional coefficient and random number seed):

```
name="BBK"
parameters=("temp", 300,
            "gamma", 2,
            "seed", 1234)
```

parameter names and defaults alternate in the Python tuple. Note also that in the implementation of class BBK, each of these variable names are used at some point in the calculation, as data members of class BBK (thus bound as attributes of self.

4.2 Classes

Constructing a propagator class first requires inheriting from either STS (for single-timestepping propagators) or MTS for (multiple-timestepping). As an example, the Python implementation of Leapfrog inherits from STS:

```
class LeapfrogPy(STS):
```

while Impulse inherits from MTS:

```
class Impulse(MTS):
```

Once the class has been declared, you can define any of three member functions which are recognized internally by MDL. Recall that upon invoking propagate, a user supplies a number of steps to execute the propagator. These member functions are called automatically at various stages of propagation:

1. init: Invoked once at the beginning of propagation.
2. run: Invoked at every step of propagation.
3. finish: Invoked once at the end of propagation.

Each of these member functions returns nothing and accepts three parameters: a Physical object, a Forces object and a Propagator object. In their definitions, they can access any attributes of these formal parameter objects, along with any member attributes that were defined as parameters for this propagator. At some point, these functions will likely update the positions and velocities data members of the Physical object, but of course they may need other attributes like the force vector to accomplish this. Note that matrix-vector operations can be performed to facilitate this process. For example in the case of Leapfrog (or Velocity Verlet), recall that two half-timestep updates of velocities (half-kicks) sandwich a full timestep update of positions (kicks). We can implement this functionality within a run member function. Note that the calculateForces member function will update the atomic force vector:

```
def run(self, phys, forces, prop):
   phys.velocities += forces.force*0.5*self.dt*phys.invmasses
   phys.positions += phys.velocities*self.dt
   prop.calculateForces(forces)
   phys.velocities += forces.force*0.5*self.dt*phys.invmasses
```

MTS propagators will be implemented similar except that you will want to invoke some of the following member functions of Propagator: initNext, runNext, and finishNext. This is because MDL only actually invokes the outermost integrator on a call to propagate. The inner integrators are invoked at the proper times in your implementation through calls to one of these functions. For example, in the Python implementation of Impulse, we run the inner integrator (faster frequency forces) and then calculate our own slower frequency forces. Note that cyclelength is automatically bound as a member variable for MTS propagators:

```
prop.runNext(phys, forces, self.cyclelength)
prop.calculateForces(forces)
```

4.2.1 Modifiers

Functionality of propagator objects can be slightly modified for special cases using *modifiers*, which are Python functions designed to be invoked at specific points in the propagation. For example, we could modify a Leapfrog propagator to sample the NVT ensemble by doing *velocity scaling*, which multiplies the velocities by $\sqrt{T_0/T}$ where T_0 is our target temperature, keeping the average kinetic energy $< KE >= \frac{3}{2}NkT$ constant. We can define a modifier for this:

```
def scaleVelocities(phys, forces, prop, obj):
   phys.velocities *= numpy.sqrt(obj.T0 / phys.temperature())
```

The location at which a modifier is invoked depends on its *type*, which is specified at the point it is coupled to a propagator. The type of a modifier can be: PreInit, PostInit, PreRun, PostRun, PreForce or PostForce. In this case we would scale velocities at the end of propagation to prevent further updates after the scaling and ensure proper Kelvin temperature, so we would classify this as a PostRun modifier. We then could construct a VelocityScale class which inherits from Leapfrog but uses this modifier, by defining a variable modifiers in the VelocityScale module and assigning it a Python array of two-element tuples which contain modifier names and types. T0 in this case is our target temperature, and this becomes bound as an attribute of class VelocityScale:

```
class VelocityScale(Leapfrog):
   pass

name="VelocityScale",
parameters=('T0', 300)
modifiers = [("scaleVelocities", "PostRun")]
```

4.3 Functions

If procedural programming is more favorable for you than object-oriented, you may prefer to implement a new propagator as a Python function instead of a Python class. In the case of a Python function, functionality is not divided into initialization, execution and finishing routines like classes, since everything is implemented as one function which is called within propagate. Propagator functions require a minimum of six formal parameters in the following order: a Physical object, a Forces object, an IO object (note that you will now need to manually invoke the member function run of IO whenever you want output

produced), number of steps, timestep (or cyclelength for MTS, and `ForceField` object. Following these are any extra parameter names that you specified in the variable `parameters`, to be recognized by the propagator factory. Finally, if this propagator is MTS, you must include a parameter for the next propagator in the chain (a Python function handle) and a formal parameter `*args` for its parameters. Thus for a BBK implementation, the header might look something like this, if we had defined propagator parameters `temp` (temperature), `gamma` and `seed`:

```
def bbk(phys, forces, prop, steps, dt, ff, temp, gamma, seed):
```

Or for Impulse since it is an MTS propagator, we must add formal parameters for the next propagator in the chain:

```
def impulse(phys, forces, io, steps, cyclelength, ff, nextprop, *args):
```

We then can update system structures as with classes. The only real differences are that calculating forces is now done through the `ForceField` object:

```
ff.calculateForces(phys, forces)
```

And for MTS, the next propagator in the chain can simply be invoked at the appropriate time. Note that the same `Physical`, `Forces` and `IO` objects must be passed. Propagator functions cannot use modifiers.

```
nextinteg(phys, forces, io, *args)
```

Chapter 5

Constructing New Forces

Unlike propagators, new force calculators must be constructed as Python classes. We plan to add force functions eventually. Any Python-prototyped forces should be contained in the `src/forces` directory. MDL provides an example implementation of a *harmonic dihedral* force as a Python class, which we now discuss. This force effectively implements a biasing potential which can allow sampling of traditionally unfavorable areas of conformational space by restraining a system around a particular angle value of one specific dihedral. Of course multiple dihedrals may be restrained by including this force more than once with different target values. A Python force class is declared in the usual way:

```
class HDForce:
```

Python-prototyped forces must define two member functions: a constructor and an `eval` method which takes no arguments and returns nothing. The harmonic dihedral restraint can be represented by the following potential energy term:

$$V(\vec{x}) = k(\phi_i(\vec{x}) - \phi_0)^2, \tag{5.1}$$

where *i* is the dihedral number which we are restraining and $\phi_i(\vec{x})$ is its dihedral angle value, ϕ_0 is the equilibrium or 'target' dihedral angle value and *k* is a scaling factor. To use this restraint, you would need to know the values of *k*, *i*, and ϕ_0. Thus when defining the constructor we would want the user to supply values for each of these variables by passing them as formal parameters to the constructor, and binding their names as data members of the new class `HDForce`. In addition we would want to pass any system structures which are necessary for computing this force, i.e. the MDL `Physical` (for computing the result) and `Forces` object (to populate the atomic force vector):

```
def __init__(self, phys, forces, phi0, index, k):
    self.phys = phys
    self.forces = forces
    self.phi0 = phi0
    self.index = index
    self.k = k
```

The job of the `eval` method is to compute both energies and forces and populate the `energies` data member of `Forces` and the atomic force vector `force`. The `eval` method does not accept any extra parameters, other than the `self` pointer which all member functions must accept. Thus all computation

must be performed by data members of self (and thus all data member bindings should be performed in the constructor):

```
def eval(self):
```

In the above case, we first compute the potential energy term according to Eq. (5.1). We first compute the difference between the current value of dihedral *i* and its target value ϕ_0 and accumulate the resulting potential energy value into the dihedral energy term of the `energies` data member of `Forces`:

```
diff = self.phys.angle(self.index) - self.phi0;
self.forces.energies.addDihedralEnergy(self.k * diff * diff)
```

Next, the force on each atom must be computed as:

$$F_i = \frac{\partial U}{\partial \phi} \frac{\partial \phi}{\partial x_i}. \tag{5.2}$$

Since there are only four atoms involved in the dihedral being restrained, the atomic force vector will only be modified in four locations. Computing $\frac{\partial U}{\partial \phi}$ is trivial:

```
dVdPhi = 2 * self.k * diff
```

Computing $\frac{\partial \phi}{\partial x_i}$ is a bit more complicated:

$$\nabla \phi = \frac{1}{\cos(\phi)} \nabla \left(\frac{\vec{C} \cdot \vec{B}}{|\vec{C}||\vec{B}|} \right), \tag{5.3}$$

where:

$$\begin{aligned} \vec{A} &= \vec{r}_{ij} \times \vec{r}_{jk} \\ \vec{B} &= \vec{r}_{jk} \times \vec{r}_{kl} \\ \vec{C} &= \vec{r}_{jk} \times \vec{A}. \end{aligned}$$

We thus must first compute the distance vectors for applicable pairs of atoms in the dihedral, then set the values of $\vec{A}$, $\vec{B}$. We make use of the `cross` function provided by numpy to compute the cross product of two three-element vectors:

```
i = self.phys.angle(self.i-1).atom1 - 1
j = self.phys.angle(self.i-1).atom2 - 1
k = self.phys.angle(self.i-1).atom3 - 1
l = self.phys.angle(self.i-1).atom4 - 1
rij = self.phys.positions[j*3:j*3+3] - self.phys.positions[i*3:i*3+3]
rkj = self.phys.positions[j*3:j*3+3] - self.phys.positions[k*3:k*3+3]
rkl = self.phys.positions[l*3:l*3+3] - self.phys.positions[k*3:k*3+3]
A = numpy.cross(rij, rkj)
B = numpy.cross(rkj, rkl)
```

Finally we compute the force on each atom `fi`, `fj`, etc. and accumulate these into the atomic force vector, data member `force` of `Forces`:

```
fi = A * (-dVdPhi * norm(rkj) / norm2(A));
fl = B * (dVdPhi * norm(rkj) / norm2(B));
fj =   fi * (-1 + numpy.dot(rij, rkj)/norm2(rkj))
     - fl * (numpy.dot(rkl, rkj)/norm2(rkj));
fk = - (fi + fj + fl);

self.forces.force[atomI*3:atomI*3+3] += fi
self.forces.force[atomJ*3:atomJ*3+3] += fj
self.forces.force[atomK*3:atomK*3+3] += fk
self.forces.force[atomL*3:atomL*3+3] += fl
```

Since Python-prototyped forces are not by default recognized as CHARMM forces, instances of their classes must be explicitly constructed in the simulation protocol, passing all necessary parameters to the constructor, i.e.:

```
hd = HDForce(phys, forces, 3.1, 11, 5.0)
```

To add the object to a `ForceField` for evaluation, you can pass an instance as a formal parameter to the `ForceField` member function `addPythonForce`:

```
ff.addPythonForce(hd)
```

Chapter 6

MDL Licensing

MDL is distributed as a tier of the ProtoMol framework, and all licensing, conditions and regulations of ProtoMol apply to MDL as well.

6.1 Contact Information

The best contact path for licensing issues is by e-mail to *protomol@cse.nd.edu* or send correspondence to:

PROTOMOL Team
c/o Prof. Jesús A. Izaguirre
Laboratory for Computational Life Sciences
Department of Computer Science and Engineering
University of Notre Dame
384 Fitzpatrick Hall of Engineering
Notre Dame, Indiana 46556 USA

Or lead developer Trevor Cickovski:

Trevor Cickovski
Natural Sciences Collegium
Eckerd College
115 Sheen Science C
St. Petersburg, Florida 33517 USA
cickovtm@eckerd.edu

Bibliography

[1] A. Alexandrescu. *Modern C++ design: Generic programming and design patterns applied.* Addison-Wesley, Reading, Massachusetts, 2001.

[2] H. C. Andersen. Rattle: A 'velocity' version of the Shake algorithm for molecular dynamics calculations. *J. Comput. Phys.*, 52:24–34, 1983.

[3] D. M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceeding of The Forth Annual Tcl/Tk Workshop '96*, pages 129–139. USENIX Association, July 1996.

[4] H. J. C. Berendsen, D. van der Spoel, and R. van Drunen. GROMACS: A message-passing parallel molecular dynamics implementation. *Comput. Phys. Commun.*, 91:43–56, 1995.

[5] F. C. Bernstein, T. F. Koetzle, G. J. Williams, E. F. Meyer, M. D. Brice, J. R. Rogers, O. Kennard, T. Shimanouchi, and M. Tasumi. The Protein Data Bank: A computer-based archival file for macromolecular structures. *J. Mol. Biol.*, 112:535–542, 1977.

[6] Stephen D. Bond, Benedict J. Leimkuhler, and Brian B. Laird. The Nosé–Poincaré method for constant temperature molecular dynamics. *J. Comput. Phys*, 151(1):114–134, 1999.

[7] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. CHARMM: A program for macromolecular energy, minimization, and dynamics calculations. *J. Comp. Chem.*, 4(2):187–217, 1983.

[8] A. Brünger, C. B. Brooks, and M. Karplus. Stochastic boundary conditions for molecular dynamics simulations of ST2 water. *Chem. Phys. Lett.*, 105:495–500, 1982.

[9] T. Cickovski, S. Chatterjee, C. Sweet, and J. A. Izaguirre. Mdlab: A molecular dynamics simulation prototyping environment. *J. Comp. Chem.*, 2009. submitted.

[10] T. Darden, D. York, and L. Pedersen. Particle mesh Ewald: An N log(N) method for Ewald sums in large systems. *J. Chem. Phys.*, 98(12):10089–10092, 1993.

[11] R. D. Engle, R. D. Skeel, and M Drees. Monitoring energy drift with shadow Hamiltonians. *J. Comput. Phys.*, 206(2):432–452, 2005.

[12] P. Ewald. Die Berechnung optischer und elektrostatischer Gitterpotentiale. *Ann. Phys.*, 64:253–287, 1921.

[13] M. S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. LeGrand, A. L. Beberg, D. L. Ensign, C. M. Bruns, and V. S. Pande. Accelerating molecular dynamic simulation on graphics processing units. *J. Comp. Chem.*, 30(6):864–872, 2009.

[14] E. Gallopoulos, E. Houstis, and J. R. Rice. Computer as thinker/doer: Problem-solving environments for computational science. *Computing in Science and Engineering*, 1(2):11–23, 1994.

[15] Gnuplot.py. Gnuplot.py on sourceforge, 2005. `http://gnuplot-py.sourceforge.net`.

[16] H. Grubmüller, H. Heller, A. Windemuth, and K. Schulten. Generalized Verlet algorithm for efficient molecular dynamics simulations with long-range interactions. *Molecular Simulation*, 6:121–142, 1991.

[17] Helmut Grubmüller. Dynamiksimulation sehr großer makromoleküle auf einem parallelrechner. Diplomarbeit, Technische Universität München, Physik-Department, T 30, James-Franck-Straße, 8046 Garching, 1989.

[18] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles*. McGraw-Hill, New York, 1981.

[19] William G. Hoover. Canonical dynamics: Equilibrium phase-space distribution. *Phys. Rev. A*, 31(3):1695–1697, 1985.

[20] Mathematica. The way the world calculates. `http://www.wolfram.com/products/mathematica`, 2005.

[21] Matplotlib. Matlab style python plotting. `http://matplotlib.sourceforge.net/`, 2005.

[22] Thierry Matthey, Trevor Cickovski, Scott S. Hampton, Alice Ko, Qun Ma, Matthew Nyerges, Troy Raeder, Thomas Slabach, and Jesús A. Izaguirre. PROTOMOL: An object-oriented framework for prototyping novel algorithms for molecular dynamics. *ACM Trans. Math. Softw.*, 30(3):237–265, 2004.

[23] R. M. H. Merks, A. G. Hoekstra, J. A. Kaandorp, P. M. A. Sloot, and P. Hogeweg. Problem-solving environments for biological morphogenesis. *Computing in Science and Engineering*, 8(1): 61–72, 2006.

[24] Shuichi Nosé. A unified formulation of the constant temperature molecular dynamics methods. *J. Chem. Phys.*, 81(1):511–519, 1984.

[25] NumPy. Numerical python libraries. `http://numeric.scipy.org`, 2005.

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

[27] R. D. Skeel and J. A. Izaguirre. An impulse integrator for Langevin dynamics. *Mol. Phys.*, 100(24):3885–3891, 2002.

[28] Subversion. Version control with Subversion. `http://subversion.tigris.org`, 2007.

[29] C. Sweet, P. Petrone, V. J. Pande, and J. A. Izaguirre. Normal mode partitioning of langevin dynamics for biomolecules. *J. Chem. Phys.*, 2008. in press.

[30] C. R. Sweet. *Hamiltonian Thermostatting Techniques for Molecular Dynamics Simulation*. PhD thesis, University of Leicester, 2004.

[31] W.C. Swope, H.C. Andersen, P.H. Berens, and K.R. Wilson. A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: application to small water clusters. *J. Chem. Phys.*, 76:637–649, 1982.

[32] M. Takahashi and M. Imada. Monte carlo calculation of quantum systems. *J. Phys. Soc. Jpn*, 53:3765–3769, 1984.

[33] M. Tuckerman, B. J. Berne, and G. J. Martyna. Reversible multiple time scale molecular dynamics. *J. Chem. Phys.*, 97(3):1990–2001, 1992.

[34] W. F. van Gunsteren and H. J. C. Berendsen. Algorithms for macromolecular dynamics and constraint dynamics. *Mol. Phys.*, 34(5):1311–1327, 1977.