MMTK User's Guide

Konrad Hinsen
Centre de Biophysique Molculaire
Centre National de la Recherche Scientifique
Rue Charles Sadron
45071 Orlans Cedex 2
France
E-Mail: hinsen@cnrs-orleans.fr

2002-6-14

Contents

Chapter 1

Introduction

The Molecular Modelling Toolkit (MMTK) presents a new approach to molecular simulations. It is not a "simulation program" with a certain set of functions that can be used by writing more or less flexible "input files", but a collection of library modules written in an easy-to-learn high-level programming language, Python. This approach offers three important advantages:

- Application programs can use the full power of a general and well-designed programming language.
- Application programs can profit from the large set of other libraries that are available for Python. These may be scientific or non-scientific; for example, it is very easy to write simulation or analysis programs with graphical user interfaces (using the module `Tkinter` in the Python standard library), or couple scientific calculations with a Web server.
- Any user can provide useful additions in separate modules, whereas adding features to a monolithic program requires at least the cooperation of the original author.

To further encourage collaborative code development, MMTK uses a very unrestrictive licensing policy, just like Python. Although MMTK is copyrighted, anyone is allowed to use it for any purpose, including commercial ones, as well as to modify and redistribute it (a more precise description is given in the copyright statement that comes with the code).

This manual describes version 2.2 of MMTK. The 2.x versions contain some incompatible changes with respect to earlier versions (1.x), most importantly a package structure that reduces the risk of name conflicts with other Python packages, and facilitates future enhancements. There are also many new features and improvements to existing functions.
Using MMTK requires a basic knowledge of object-oriented programming and Python. Newcomers to this subject should have a look at the introductory section in this manual and at the Python tutorial (which also comes with the Python interpreter). There are also numerous books on Pythonthat are useful in getting started. Even without MMTK, Python is a very useful programming language for scientific use, allowing rapid development and testing and easy interfacing to code written in low-level languages such as Fortran or C.
This manual consists of several introductory chapters and a Module Reference. The introductory chapters explain how common tasks are handled with MMTK, but they do not describe all of its features, nor do they contain a full documentation of functions or classes. This information can be found in the Module Reference, which describes all classes and functions intended for end-user applications module by module, using documentation extracted directly from the source code. References from the introductory sections to the module reference facilitate finding the relevant documentation.

Chapter 2

Overview

This chapter explains the basic structure of MMTK and its view of molecular systems. Every MMTK user should read it at least once.

Using MMTK

MMTK applications are ordinary Python programs, and can be written using any standard text editor. For interactive use it is recommended to use either the special Python mode for the Emacs editor, or one of the Tk-based graphical user interfaces for Python, IDLE (comes with the Python interpreter from version 1.5.2) or PTUI.
MMTK tries to be as user-friendly as possible for interactive use. For example, lengthy calculations can usually be interrupted by typing Control-C. This will result in an error message ("Keyboard Interrupt"), but you can simply go on typing other commands. Interruption is particularly useful for energy minimization and molecular dynamics: you can interrupt the calculation at any time, look at the current state or do some analysis, and then continue.

Modules

MMTK is a package consisting of various modules, most of them written in Python, and some in C for efficiency. The individual modules are described in the Module Reference. The basic definitions that almost every application needs are collected in the top-level module, MMTK. The first line of most applications is therefore

```
from MMTK import *
```

The definitions that are specific to particular applications reside in submodules within the package MMTK. For example, force fields are defined in MMTK.ForceFields (page 98), and peptide chain and protein objects are defined in MMTK.Proteins (page 128).
Python provides two ways to access objects in modules and submodules. The first one is importing a module and referring to objects in it, e.g.:

```
import MMTK
import MMTK.ForceFields
universe = MMTK.InfiniteUniverse(MMTK.ForceFields.Amber94ForceField())
```

The second method is importing all or some objects *from*a module:

```
from MMTK import InfiniteUniverse
from MMTK.ForceFields import Amber94ForceField
universe = InfiniteUniverse(Amber94ForceField())
```

These two import styles can also be mixed according to convience. In order to prevent any confusion, all objects are referred to by their full names in this manual. The Amber force field object is thus called MMTK.ForceFields.Amber94ForceField (page 99). Of course the user is free to use selective imports in order to be able to use such objects with shorter names.

Objects

MMTK is an object-oriented system. Since objects are everywhere and everything is an object, it is useful to know the most important object types and what can be done with them. All object types in MMTK have meaningful names, so it is easy to identify them in practice. The following overview contains only those objects that a user will see directly. There are many more object types used by MMTK internally, and also some less common user objects that are not mentioned here.

Chemical objects

These are the objects that represent the parts of a molecular system:

- atoms
- groups
- molecules
- molecular complexes

These objects form a simple hierarchy: complexes consist of molecules, molecules consist of groups and atoms, groups consist of smaller groups and atoms. All of these, except for groups, can be used directly to construct a molecular system. Groups can only be used in the definitions of other groups and molecules in the chemical database.
A number of operations can be performed on chemical objects, which can roughly be classified into inquiry (constituent atoms, bonds, center of mass etc.) and modification (translate, rotate).
There are also specialized versions of some of these objects. For example, MMTK defines proteins as special complexes, consisting of peptide chains, which are special molecules. They offer a range of special operations (such as selecting residues or constructing the positions of missing hydrogen atoms) that do not make sense for molecules in general.

Collections

Collection objects represent arbitrary collections of chemical objects. They are used to be able to refer to collections as single entities. For example, you might want to call all water molecules collectively ”solvent”. Most of the operations on chemical objects are also available for collections.

Force fields

Force field objects represent a precise description of force fields, i.e. a complete recipe for calculating the potential energy (and its derivatives) for a given molecular system. In other words, they specify not only the functional form of the various interactions, but also all parameters and the prescriptions for applying these parameters to an actual molecular system.

Universes

Universes define complete molecular systems, i.e. they contain chemical objects. In addition, they describe interactions within the system (by a force field), boundary conditions, external fields, etc. Many of the operations that can be used on chemical objects can also be applied to complete universes.

Minimizers and integrators

A minimizer object is a special ”machine” that can find local minima in the potential energy surface of a universe. You may consider this a function, if you wish, but of course functions are just special objects. Similarly, an integrator is a special ”machine” that can determine a dynamical trajectory for a system on a given potential energy surface.

Trajectories

Minimizers and integrators can produce trajectories, which are special files containing a sequence of configurations and/or other related information. Of course trajectory objects can also be read for analysis.

Variables

Variable objects (not to be confused with standard Python variables) describe quantities that have a value for each atom in a system, for example positions, masses, or energy gradients. Their most common use is for storing various configurations of a system.

Normal modes

Normal mode objects contain normal mode frequencies and atomic displacements for a given universe.

Non-MMTK objects

An MMTK application program will typically also make use of objects provided by Python or Python library modules. A particularly useful library is the package Scientific, which is also used by MMTK itself. The most important objects are

- numbers (integers, real number, complex numbers), provided by Python
- vectors (in 3D coordinate space) provided by the module Scientific.Geometry.
- character strings, provided by Python
- files, provided by Python

Of course MMTK applications can make use of the Python standard library or any other Python modules. For example, it is possible to write a simulation program that provides status reports via an integrated Web server, using the Python standard module SimpleHTTPServer.

The chemical database

For defining the chemical objects described above, MMTK uses a database of descriptions. There is a database for atoms, one for groups, etc. When you ask MMTK to make a specific chemical object, for example a water molecule, MMTK looks for the definition of water in the molecule database. A database entry contains everything there is to know about the object it defines: its constituents and their names, configurations, other names used e.g. for I/O, and all information force fields might need about the objects. MMTK comes with database entries for many common objects (water, amino acids, etc.). For other objects you will have to write the definitions yourself, as described in the section on the database.

Force fields

MMTK contains everything necessary to use the Amber 94 force field on proteins, DNA, and water molecules. It uses the standard Amber parameter and modification file format. In addition to the Amber force field, there is a simple Lennard-Jones force field for noble gases, and a deformation force field for normal mode calculations on large proteins. MMTK was designed to make the addition of force field terms and the implementation of other force fields as easy as possible. Force field terms can be defined in Python (for ease of implementation) or in C or Fortran (for efficiency). This is described in the developer's guide.

Units

Since MMTK is not a black-box program, but a modular library, it is essential for it to use a consistent unit system in which, for example, the inverse of a frequency is a time, and the product of a mass and the square of a velocity is an energy, without additional conversion factors. Black-box programs can (and usually do) use a consistent unit system internally and convert to ”conventional” units for input and output.
The unit system of MMTK consists mostly of SI units of appropriate magnitude for molecular systems:

- nm for lengths
- ps for times
- atomic mass units (g/mol) for masses
- kJ/mol for energies
- THz (1/ps) for frequencies
- K for temperatures
- elementary charges

The module MMTK.Units (page 146) contains convenient conversion constants for the units commonly used in computational chemistry. For example, a length of 2 ngstrm can be written as `2*Units.Ang`, and a frequency can be printed in wavenumbers with `print frequency/Units.invcm`.

A simple example

The following simple example shows how a typical MMTK application might look like. It constructs a system consisting of a single water molecule and runs a short molecular dynamics trajectory. There are many alternative ways to do this; this particular one was chosen because it makes each step explicit and clear. The individual steps are explained in the remaining chapters of the manual.

```
# Import the necessary MMTK definitions.
from MMTK import *
from MMTK.ForceFields import Amber94ForceField
from MMTK.Trajectory import Trajectory, TrajectoryOutput, \
                            StandardLogOutput
from MMTK.Dynamics import VelocityVerletIntegrator
# Create an infinite universe (i.e. no boundaries, non-periodic).
universe = InfiniteUniverse(Amber94ForceField())
# Create a water molecule in the universe.
# Water is defined in the database.
universe.molecule = Molecule('water')
# Generate random velocities.
universe.initializeVelocitiesToTemperature(300*Units.K)
# Create an integrator.
integrator = VelocityVerletIntegrator(universe)
# Generate a trajectory
trajectory = Trajectory(universe, "water.nc", "w")
# Run the integrator for 50 steps of 1 fs, printing time and energy
# every fifth step and writing time, energy, temperature, and the positions
# of all atoms to the trajectory at each step.
integrator(delta_t = 1.*Units.fs, steps = 50,
           actions = [StandardLogOutput(5),
                      TrajectoryOutput(trajectory, ("time", "energy",
                                                    "thermodynamic",
                                                    "configuration"),
                                       0, None, 1)])
# Close the trajectory
trajectory.close()
```

Chapter 3

Constructing a molecular system

The construction of a complete system for simulation or analysis involves some or all of the following operations:

- Creating molecules and other chemical objects.
- Defining the configuration of all objects.
- Defining the ”surroundings” (e.g. boundary conditions).
- Choosing a force field.

MMTK offers a large range of functions to deal with these tasks.

Creating chemical objects

Chemical objects (atoms, molecules, complexes) are created from definitions in the database. Since these definitions contain most of the necessary information, the subsequent creation of the objects is a simple procedure. All objects are created by their class name (MMTK.Atom (page 65), MMTK.Molecule (page 67), and MMTK.Complex (page 69)) with the name of the definition file as first parameter. Additional optional parameters can be specified to modify the object being created. The following optional parameters can be used for all object types:

- `name=`string Specifies a name for the object. The default name is the one given in the definition file.
- `position=`vector Specifies the position of the center of mass. The default is the origin.
- `configuration=`string Indicates a configuration from the configuration dictionary in the definition file. The default is 'default' if such an entry exists in the configuration dictionary. Otherwise the object is created without atomic positions.

Some examples with additional explanations for specific types:

- `Atom('C')` creates a carbon atom.
- `Molecule('water', position=Vector(0.,0.,1.))`creates a water molecule using configuration 'default' and moves the center of mass to the indicated position.

Proteins, peptide chains, and nucleotide chains

MMTK contains special support for working with proteins, peptide chains, and nucleotide chains. As described in the chapter on the database, proteins can be described by a special database definition file. However, it is often simpler to create protein objects directly in an application program. The classes are MMTK.Proteins.PeptideChain (page 128), MMTK.Proteins.Protein (page 130), and MMTK.NucleicAcids.NucleotideChain (page 120).
Proteins can be created from definition files in the database, from previously constructed peptide chain objects, or directly from PDB files if no special manipulations are necessary.
Examples: `Protein('insulin')` creates a protein object for insulin from a database file. `Protein('1mbd.pdb')` creates a protein object for myoglobin directly from a PDB file, but leaving out the heme group, which is not a peptide chain.
Peptide chains are created from a sequence of residues, which can be a MMTK.PDB.PDBPeptideChain (page 123) object, a list of three-letter residue codes, or a string containing one-letter residue codes. In the last two cases the atomic positions are not defined. MMTK provides several

models for the residues which provide different levels of detail: an all-atom model, a model without hydrogen atoms, two models containing only polar hydrogens (using different definitions of polar hydrogens), and a model containing only the C-alpha atoms, with each C-alpha atom having the mass of the entire residue. The last model is useful for conformational analyses in which only the backbone conformations are important.
The construction of nucleotide chains is very similar. The residue list can be either a MMTK.PDB.PDBNucleotideChain (page 123) object or a list of two-letter residue names. The first letter of a residue name indicates the sugar type (`'R'` for ribose and `'D'` for desoxyribose), and the second letter defines the base (`'A'`, `'C'`, and `'G'`, plus `'T'` for DNA and `'U'` for RNA). The models are the same as for peptide chains, except that the C-alpha model does not exist.
Most frequently proteins and nucleotide chains are created from a PDB file. The PDB files often contain solvent (water) as well, and perhaps some other molecules. MMTK provides convenient functions for extracting information from PDB files and for building molecules from them in the module MMTK.PDB (page 123). The first step is the creation of a MMTK.PDB.PDBConfiguration (page 124)object from the PDB file:

```
from MMTK.PDB import PDBConfiguration
configuration = PDBConfiguration('some_file.pdb')
```

The easiest way to generate MMTK objects for all molecules in the PDB file is then

```
molecules = configuration.createAll()
```

The result is a collection of molecules, peptide chains, and nucleotide chains, depending on the contents of the PDB files. There are also methods for modifying the PDBConfiguration before creating MMTK objects from it, and for creating objects selectively. See the documentation for the modules MMTK.PDB (page 123) and Scientific.IO.PDB for details, as well as the protein and DNA examples.

Lattices

Sometimes it is necessary to generate objects (atoms or molecules) positioned on a lattice. To facilitate this task, MMTK defines lattice objects which are essentially sequence objects containing points or objects at points. Lattices can therefore be used like lists with indexing and `for`-loops. The lattice classes are MMTK.Geometry.RhombicLattice (page 107), MMTK.Geometry.BravaisLattice (page 108), and MMTK.Geometry.SCLattice (page 108).

Random numbers

The Python standard library and the Numerical Python package provide random number generators, and more are available in seperate packages. MMTK provides some convenience functions that return more specialized random quantities: random points in a universe, random velocities, random particle displacement vectors, random orientations. These functions are defined in module MMTK.Random (page 133).

Collections

Often it is useful to treat a collection of several objects as a single entity. Examples are a large number of solvent molecules surrounding a solute, or all sidechains of a protein. MMTK has special collection objects for this purpose, defined as class MMTK.Collection (page 66). Most of the methods available for molecules can also be used on collections.
A variant of a collection is the partitioned collection, implemented in class MMTK.PartitionedCollection (page 68). This class acts much like a standard collection, but groups its elements by geometrical position in small sub-boxes. As a consequence, some geometrical algorithms (e.g. pair search within a cutoff) are much faster, but other operations become somewhat slower.

Creating universes

A universe describes a complete molecular system consisting of any number of chemical objects and a specification of their interactions (i.e. a force field) and surroundings: boundary conditions, external fields, thermostats, etc. The universe classes are defined in module MMTK:

- MMTK.InfiniteUniverse (page 69) represents an infinite universe, without any boundary or periodic boundary conditions.

- MMTK.OrthorhombicPeriodicUniverse (page 70) represents a periodic universe with an orthorhombic elementary cell, whose size is defined by the three edge lengths.
- MMTK.CubicPeriodicUniverse (page 71) is a special case of MMTK.OrthorhombicPeriodicUniverse (page 70) in which the elementary cell is cubic.

Universes are created empty; the contents are then added to them. Three types of objects can be added to a universe: chemical objects (atoms, molecules, etc.), collections, and environment objects (thermostats etc.). It is also possible to remove objects from a universe.

Force fields

MMTK comes with several force fields, and permits the definition of additional force fields. Force fields are defined in module MMTK.ForceFields (page 98). The most import built-in force field is the Amber 94 force field, represented by the class MMTK.ForceFields.Amber94ForceField (page 99). It offers several strategies for electrostatic interactions, including Ewald summation, a fast multipole method [DPMTA], and cutoff with charge neutralization and optional screening [Wolf1999].
In addition to the Amber 94 force field, there is a Lennard-Jones force field for noble gases (Class MMTK.ForceFields.LennardJonesForceField (page 99)) and a deformation force field for protein normal mode calculations (Class MMTK.ForceFields.DeformationForceField (page 98)).

Referring to objects and parts of objects

Most MMTK objects (in fact all except for atoms) have a hierarchical structure of parts of which they consist. For many operations it is necessary to access specific parts in this hierarchy.
In most cases, parts are attributes with a specific name. For example, the oxygen atom in every water molecule is an attribute with the name "O". Therefore if `w` refers to a water molecule, then `w.O` refers to its oxygen atom. For a more complicated example, if `m`refers to a molecule that has a methyl group called "M1", then `m.M1.C`refers to the carbon atom of that methyl group. The names of attributes are defined in the database.
Some objects consist of parts that need not have unique names, for example the elements of a collection, the residues in a peptide chain, or the chains in a protein. Such parts are accessed by indices; the objects that contain them are Python sequence types. Some examples:

- Asking for the number of items: if `c`refers to a collection, then `len(c)` is the number of its elements.
- Extracting an item: if `p` refers to a protein, then `p[0]` is its first peptide chain.
- Iterating over items: if `p` refers to a peptide chain, then `for residue in p:  print residue.position()` will print the center of mass positions of all its residues.

Peptide and nucleotide chains also allow the operation of slicing: if `p`refers to a peptide chain, then `p[1:-1]`is a subchain extending from the second to the next-to-last residue.

The structure of peptide and nucleotide chains

Since peptide and nucleotide chains are not constructed from an explicit definition file in the database, it is not evident where their hierarchical structure comes from. But it is only the top-level structure that is treated in a special way. The constituents of peptide and nucleotide chains, residues, are normal group objects. The definition files for these group objects are in the MMTK standard database and can be freely inspected and even modified or overriden by an entry in a database that is listed earlier in MMTKDATABASE.

Peptide chains are made up of amino acid residues, each of which is a group consisting of two other groups, one being called "peptide" and the other "sidechain". The first group contains the peptide group and the C and H atoms; everything else is contained in the sidechain. The C atom of the fifth residue of peptide chain `p`is therefore referred to as `p[4].peptide.C_alpha`. Nucleotide chains are made up of nucleotide residues, each of which is a group consisting of two or three other groups. One group is called "sugar" and is either a ribose or a desoxyribose group, the second one is called "base" and is one the five standard bases. All but the first residue in a nucleotide chain also have a subgroup called "phosphate" describing the phosphate group that links neighbouring residues.

Analyzing and modifying atom properties

General operations

Many inquiry and modification operations act at the atom level and can equally well be applied to any object that is made up of atoms, i.e. atoms, molecules, collections, universes, etc. These operations are defined once in a mix-in classcalled MMTK.Collection.GroupOfAtoms (page 78), but are available for all objects for which they make sense. They include inquiry-type functions (total mass, center of mass, moment of inertia, bounding box, total kinetic energy etc.), coordinate modifications (translation, rotation, application of transformation objects) and coordinate comparisons (RMS difference, optimal fits).

Coordinate transformations

The most common coordinate manipulations involve translations and rotations of specific parts of a system. It is often useful to refer to such an operation by a special kind of object, which permits the combination and analysis of transformations as well as its application to atomic positions.

Transformation objects specify a general displacement consisting of a rotation around the origin of the coordinate system followed by a translation. They are defined in the module Scientific.Geometry, but for convenience the module MMTK contains a reference to them as well. Transformation objects corresponding to pure translations can be created with `Translation(displacement)`; transformation objects describing pure rotations with `Rotation(axis, angle)` or `Rotation(rotation_matrix)`. Multiplication of transformation objects returns a composite transformation.

The translational component of any transformation can be obtained by calling the method `translation()`; the rotational component is obtained analogously with `rotation()`. The displacement vector for a pure translation can be extracted with the method `displacement()`, a tuple of axis and angle can be extracted from a pure rotation by calling `axisAndAngle()`.

Atomic property objects

Many properties in a molecular system are defined for each individual atom: position, velocity, mass, etc. Such properties are represented in special objects, defined in module MMTK: MMTK.ParticleScalar (page 63)for scalar quantities, MMTK.ParticleVector (page 64) for vector quantities, and MMTK.ParticleTensor (page 65) for rank-2 tensors. All these objects can be indexed with an atom object to retrieve or change the corresponding value. Standard arithmetic operations are also defined, as well as some useful methods.

Configurations

A configuration object, represented by the class MMTK.Configuration (page 65) is a special variant of a MMTK.ParticleVector (page 64) object. In addition to the atomic coordinates of a universe, it stores geometric parameters of a universe that are subject to change, e.g. the edge lengths of the elementary cell of a periodic universe. Every universe has a current configuration, which is what all operations act on by default. It is also the configuration that is updated by minimizations, molecular dynamics, etc. The current configuration can be obtained by calling the method `configuration()`.
There are two ways to create configuration objects: by making a copy of the current configuration (with `copy(universe.configuration())`, or by reading a configuration from a trajectory file.

Chapter 4

Minimization and Molecular Dynamics

Trajectories

Minimization and dynamics algorithms produce sequences of configurations that are often stored for later analysis. In fact, they are often the most valuable result of a lengthy simulation run. To make sure that the use of trajectory files is not limited by machine compatibility, MMTK stores trajectories in netCDFfiles. These files contain binary data, minimizing disk space usage, but are freely interchangeable between different machines. In addition, there are a number of programs that can perform standard operations on arbitrary netCDF files, and which can therefore be used directly on MMTK trajectory files. Finally, netCDF files are self-describing, i.e. contain all the information needed to interpret their contents. An MMTK trajectory file can thus be inspected and processed without requiring any further information.

For illustrations of trajectory operations, see the trajectory examples.

Trajectory file objects are represented by the class MMTK.Trajectory.Trajectory (page 138). They can be opened for reading, writing, or modification. The data in trajectory files can be stored in single precision or double precision; single-precision is usually sufficient, but double-precision files are required to reproduce a given state of the system exactly.

A trajectory is closed by calling the method `close()`. If anything has been

written to a trajectory, closing it is required to guarantee that all data has been written to the file. Closing a trajectory after reading is recommended in order to prevent memory leakage, but is not strictly required.
Newly created trajectories can contain all objects in a universe or any subset; this is useful for limiting the amount of disk space occupied by the file by not storing uninteresting parts of the system, e.g. the solvent surrounding a protein. It is even possible to create a trajectory for a subset of the atoms in a molecule, e.g. for only the C-alpha atoms of a protein. The universe description that is stored in the trajectory file contains all chemical objects of which at least one atom is represented.
When a trajectory is opened for reading, no universe object needs to be specified. In that case, MMTK creates a universe from the description contained in the trajectory file. This universe will contain the same objects as the one for which the trajectory file was created, but not necessarily have all the properties of the original universe (the description contains only the names and types of the objects in the universe, but not, for example, the force field). The universe can be accessed via the attribute `universe`of the trajectory.
If the trajectory was created with partial data for some of the objects, reading data from it will set the data for the missing parts to "undefined". Analysis operations on such systems must be done very carefully. In most cases, the trajectory data will contain the atomic configurations, and in that case the "defined" atoms can be extracted with the method `atomsWithDefinedPositions()`.
MMTK trajectory files can store various data: atomic positions, velocities, energies, energy gradients etc. Each trajectory-producing algorithm offers a set of quantities from which the user can choose what to put into the trajectory. Since a detailed selection would be tedious, the data is divided into classes, e.g. the class "energy" stands for potential energy, kinetic energy, and whatever other energy-related quantities an algorithm produces.
For optimizing I/O efficiency, the data layout in a trajectory file can be modified by the `block_size` parameter. Small block sizes favour reading or writing all data for one time step, whereas large block sizes (up to the number of steps in the trajectory) favour accessing a few values for all time steps, e.g. scalar variables like energies or trajectories for individual atoms. The default value of the block size is one.
Every trajectory file contains a history of its creation. The creation of the file is logged with time and date, as well as each operation that adds data

to it with parameters and the time/date of start and end. This information, together with the comment and the number of atoms and steps contained in the file, can be obtained with the function
MMTK.Trajectory.trajectoryInfo.
It is possible to read data from a trajectory file that is being written to by another process. For efficiency, trajectory data is not written to the file at every time step, but only approximately every 15 minutes. Therefore the amount of data available for reading may be somewhat less than what has been produced already.

Options for minimization and dynamics

Minimizers and dynamics integrators accept various optional parameter specifications. All of them are selected by keywords, have reasonable default values, and can be specified when the minimizer or integrator is created or when it is called. In addition to parameters that are specific to each algorithm, there is a general parameter actions that specifies actions that are executed periodically, including trajectory and console output.

Periodic actions

Periodic actions are specified by the keyword parameter actions whose value is a list of periodic actions, which defaults to an empty list. Some of these actions are applicable to any trajectory-generating algorithm, especially the output actions. Others make sense only for specific algorithms or specific universes, e.g. the periodic rescaling of velocities during a Molecular Dynamics simulation.
Each action is described by an action object. The step numbers for which an action is executed are specified by three parameters. The parameter first indicates the number of the first step for which the action is executed, and defaults to 0. The parameter last indicates the last step for which the action is executed, and default to `None`, meaning that the action is executed indefinitely. The parameter skip speficies how many steps are skipped between two executions of the action. The default value of 1 means that the action is executed at each step. Of course an action object may have additional parameters that are specific to its action.
The output actions are defined in the module MMTK.Trajectory (page 138) and can be used with any trajectory-generating algorithm. They are:

- MMTK.Trajectory.TrajectoryOutput (page 142) for writing data to a trajectory. Note that it is possible to use several trajectory output actions simultaneously to write to multiple trajectories. It is thus possible, for example, to write a short dense trajectory during a dynamics run for analyzing short-time dynamics, and simultaneously a long-time trajectory with a larger step spacing, for analyzing long-time dynamics.

- MMTK.Trajectory.RestartTrajectoryOutput (page 143), which is a specialized version of MMTK.Trajectory.TrajectoryOutput (page

142). It writes the data that the algorithm needs in order to be restarted to a restart trajectory file. A restart trajectory is a trajectory that stores a fixed number of steps which are reused cyclically, such that it always contain the last few steps of a trajectory.

- MMTK.Trajectory.LogOutput (page 143) for text output of data to a file.
- MMTK.Trajectory.StandardLogOutput (page 144), a specialized version of MMTK.Trajectory.LogOutput (page 143) that writes the data classes ”time” and ”energy” during the whole simulation run to standard output.

The other periodic actions are meaningful only for Molecular Dynamics simulations:

- MMTK.Dynamics.VelocityScaler (page 91) is used for rescaling the velocities to force the kinetic energy to the value defined by some temperature. This is usually done during initial equilibration.
- MMTK.Dynamics.BarostatReset (page 92) resets the barostat coordinate to zero and is during initial equilibration of systems in the NPT ensemble.
- MMTK.Dynamics.Heater (page 91) rescales the velocities like MMTK.Dynamics.VelocityScaler (page 91), but increases the temperature step by step.
- MMTK.Dynamics.TranslationRemover (page 92) subtracts the global translational velocity of the system from all individual atomic velocities. This prevents a slow but systematic energy flow into the degrees of freedom of global translation, which occurs with most MD integrators due to non-perfect conservation of momentum.
- MMTK.Dynamics.RotationRemover (page 93) subtracts the global angular velocity of the system from all individual atomic velocities. This prevents a slow but systematic energy flow into the degrees of freedom of global rotation, which occurs with most MD integrators due to non-perfect conservation of angular momentum.

Fixed atoms

During the course of a minimization or molecular dynamics algorithm, the atoms move to different positions. It is possible to exclude specific atoms from this movement, i.e. fixing them at their initial positions. This has no influence whatsoever on energy or force calculations; the only effect is that the atoms' positions never change. Fixed atoms are specified by giving them an attribute `fixed`with a value of one. Atoms that do not have an attribute `fixed`, or one with a value of zero, move according to the selected algorithm.

Energy minimization

MMTK has two energy minimizers using different algorithms: steepest descent (MMTK.Minimization.SteepestDescentMinimizer (page 111)) and conjugate gradient (MMTK.Minimization.ConjugateGradientMinimizer (page 112)) . Steepest descent minimization is very inefficient if the goal is to find a local minimum of the potential energy. However, it has the advantage of always moving towards the minimum that is closest to the starting point and is therefore ideal for removing bad contacts in a unreasonably high energy configuration. For finding local minima, the conjugate gradient algorithm should be used.
Both minimizers accept three specific optional parameters:

- steps (an integer) to specify the maximum number of steps (default is 100)
- step_size (a number) to specify an initial step length used in the search for a minimum (default is 2 pm)
- convergence (a number) to specify the gradient norm (more precisely the root-mean-square length) at which the minimization should stop (default is 0.01 kJ/mol/nm)

There are three classes of trajectory data: ”energy” includes the potential energy and the norm of its gradient, ”configuration” stands for the atomic positions, and ”gradients” stands for the energy gradients at each atom position.
The following example performs 100 steps of steepest descent minimization without producing any trajectory or printed output:

```
from MMTK import *
from MMTK.ForceFields import Amber94ForceField
from MMTK.Minimization import SteepestDescentMinimizer
universe = InfiniteUniverse(Amber94ForceField())
universe.protein = Protein('insulin')
minimizer = SteepestDescentMinimizer(universe)
minimizer(steps = 100)
```

See also the example file NormalModes/modes.py.

Molecular dynamics

The techniques described in this section are illustrated by several Molecular Dynamics examples.

Velocities

The integration of the classical equations of motion for an atomic system requires not only positions, but also velocities for all atoms. Usually the velocities are initialized to random values drawn from a normal distribution with a variance corresponding to a certain temperature. This is done by calling the method `initializeVelocitiesToTemperature(temperature)`on a universe. Note that the velocities are assigned atom by atom; no attempt is made to remove global translation or rotation of the total system or any part of the system.

During equilibration of a system, it is common to multiply all velocities by a common factor to restore the intended temperature. This can done explicitly by calling the method `scaleVelocitiesToTemperature(temperature)`on a universe, or by using the action object MMTK.Dynamics.VelocityScaler (page 91).

Distance constraints

A common technique to eliminate the fastest (usually uninteresting) degrees of freedom, permitting a larger integration time step, is the use of distance constraints on some or all chemical bonds. MMTK allows the use of distance constraints on any pair of atoms, even though constraining anything but chemical bonds is not recommended due to considerable modifications of the dynamics of the system [vanGunsteren1982, Hinsen1995].

MMTK permits the definition of distance constraints on all atom pairs in an object that are connected by a chemical bond by calling the method `setBondConstraints`. Usually this is called for a complete universe, but it can also be called for a chemical object or a collection of chemical objects. The method `removeDistanceConstraints` removes all distance constraints from the object for which it is called.

Constraints defined as described above are automatically taken into account by Molecular Dynamics integrators. It is also possible to enforce the constraints explicitly by calling the method `enforceConstraints` for a universe. This has the effect of modifying the configuration and the velocities (if velocities exist) in order to make them compatible with the constraints.

Thermostats and barostats

A standard Molecular Dynamics integration allows time averages corresponding to the NVE ensemble, in which the number of molecules, the system volume, and the total energy are constant. This ensemble does not represent typical experimental conditions very well. Alternative ensembles are the NVT ensemble, in which the temperature is kept constant by a thermostat, and the NPT ensemble, in which temperature and pressure are kept constant by a thermostat and a barostat. To obtain these ensembles in MMTK, thermostat and barostat objects must be added to a universe. In the presence of these objects, the Molecular Dynamics integrator will use the extended-systems method for producing the correct ensemble. The classes to be used are MMTK.Environment.NoseThermostat (page 94) and MMTK.Environment.AndersenBarostat (page 94).

Integration

A Molecular Dynamics integrator based on the "Velocity Verlet" algorithm [Swope1982], which was extended to handle distance constraints as well as thermostats and barostats [Kneller1996], is implemented by the class MMTK.Dynamics.VelocityVerletIntegrator (page 90). It has two optional keyword parameters:

- steps (an integer) to specify the number of steps (default is 100)
- delta_t (a number) to specify the time step (default 1 fs)

There are three classes of trajectory data: "energy" includes the potential energy and the kinetic energy, as well as the energies of thermostat and barostat coordinates if they exist, "time" stands for the time, "thermodynamic" stand for temperature and pressure, "configuration"

stands for the atomic positions, ”velocities” stands for the atomic velocities, and ”gradients” stands for the energy gradients at each atom position.
The following example performs a 1000 step dynamics integration, storing every 10th step in a trajectory file and removing the total translation and rotation every 50th step:

```
from MMTK import *
from MMTK.ForceFields import Amber94ForceField
from MMTK.Dynamics import VelocityVerletIntegrator, TranslationRemover, \
                          RotationRemover
from MMTK.Trajectory import TrajectoryOutput
universe = InfiniteUniverse(Amber94ForceField())
universe.protein = Protein('insulin')
universe.initializeVelocitiesToTemperature(300.*Units.K)
actions = [TranslationRemover(0, None, 50),
           RotationRemover(0, None, 50),
           TrajectoryOutput("insulin.nc",
                            ("configuration", "energy", "time"),
                            0, None, 10)]
integrator = VelocityVerletIntegrator(universe, delta_t = 1.*Units.fs,
                                      actions = actions)
integrator(steps = 1000)
```

Snapshots

A snapshot generator allows writing the current system state to a trajectory. It works much like a zero-step minimization or dynamics run, i.e. it takes the same optional arguments for specifying the trajectory and protocol output. A snapshot generator is created using the class MMTK.Trajectory.SnapshotGenerator (page 144).

Chapter 5

Normal modes

Normal mode analysis provides an analytic description of the dynamics of a system near a minimum using an harmonic approximation to the potential. Before a normal mode analysis can be started, the system must be brought to a local minimum of the potential energy by energy minimization, except when special force fields designed only for normal mode analysis are used (e.g. MMTK.ForceFields.DeformationForceField (page 98)). See also the normal mode examples.

A standard normal mode analysis is performed by creating a normal mode object, implemented in class MMTK.NormalModes.NormalModes (page 114). A normal mode object behaves like a sequence of MMTK.NormalModes.Mode (page 114)objects which store the atomic displacement vectors corresponding to each mode and its vibrational frequency.

For short-ranged potentials, it is advantageous to store the second derivatives of the potential in a sparse-matrix form and to use an iterative method to determine some or all modes. This permits the treatments of larger systems that would normally require huge amounts of memory. A sparse-matrix method is implemented in class MMTK.NormalModes.SparseMatrixNormalModes (page 115).

Another approach to deal with large systems is the restriction to low-frequency modes which are supposed to be well representable by linear combinations of a given set of basis vectors. The basis vectors can be obtained from a basis for the full Cartesian space by elimination of known fast degrees of freedom (e.g. bonds); the module MMTK.Subspace (page 136) contains support classes for this approach. It is also possible to construct a suitable basis vector set from small-deformation vector fields (e.g. MMTK.FourierBasis.FourierBasis (page 104)). The normal mode analysis for a given set of basis vectors is performed by the class MMTK.NormalModes.SubspaceNormalModes (page 116). There is also a variant using finite difference differentiation (MMTK.NormalModes.FiniteDifferenceSubspaceNormalModes (page 117)) and another one using a sparse-matrix representation of the second derivatives (MMTK.NormalModes.SparseMatrixSubspaceNormalModes (page 118)).

Chapter 6

Analysis operations

Analysis is the most non-standard part of molecular simulations. The quantities that must be calculated depend strongly on the system and the problem under study. MMTK provides a wide range of elementary operations that inquire the state of the system, as well as several more complex analysis tools. Some of them are demonstrated in the examples section.

Properties of chemical objects and universes

Many operations access and modify various properties of an object. They are defined for the most general type of object: anything that can be broken down to atoms, i.e. atoms, molecules, collections, universes, etc., i.e. in the class MMTK.Collection.GroupOfAtoms (page 78).

The most elementary operations are inquiries about specific properties of an object: number of atoms, total mass, center of mass, total momentum, total charge, etc. There are also operations that compare two different conformations of a system. Finally, there are special operations for analyzing conformations of peptide chains and proteins.

Geometrical operations in periodic universes require special care. Whenever a distance vector between two points in a systems is evaluated, the minimum-image convention must be used in order to obtain consistent results. MMTK provides routines for finding these distance vectors as well as distances, angles, and dihedral angles between any points. Because these operations depend on the topology and geometry of the universe, they are

implemented as methods in class MMTK.Universe.Universe (page 147)and its subclasses. Of course they are available for non-periodic universes as well.
Universes also provide methods for obtaining atom propertyobjects that describe the state of the system (configurations, velocities, masses), and for restoring the system state from a trajectoryfile.

Energy evaluation

Energy evaluation requires a force field, and therefore all the methods in this section are defined only for universe objects, i.e. in class MMTK.Universe.Universe (page 147). However, they all take an optional arguments (anything that can be broken down into atoms) that indicates for which subset of the universe the energy is to be evaluated. In addition to the potential energy, energy gradients and second derivatives (force constants) can be obtained, if the force field implements them. There is also a method that returns a dictionary containing the values for all the individual force field terms, which is often useful for analysis.

Surfaces and volumes

Surfaces and volumes can be analyzed for anything consisting of atoms. Both quantities are defined by assigning a radius to each atom; the surface of the resulting conglomerate of overlapping spheres is taken to be the surface of the atom group. Atom radii for surface determination are usually called "van der Waals radii", but there is no unique method for determining them. MMTK uses the values from [Bondi1964]. However, users can change these values for each individual atom by assigning a new value to the attribute "vdW_radius".

Surface and volume calculations are implemented in the module MMTK.MolecularSurface (page 113) and make use of the NSC library by Frank Eisenhabes [Eisenhaber1993, Eisenhaber1995]. Because this library is subject to stricter copyright conditions than MMTK (it can be freely used only for non-commercial purposes), the whole molecular surface package (NSC and the module MolecularSurface) is distributed separately from the main MMTK distribution. If you get error messages when trying to evaluate surfaces or values, please verify that you have installed this separate package.

The operations provided in MMTK.MolecularSurface (page 113)include basic surface and volume calculation, determination of exposed atoms, and identification of contacts between two objects.

Chapter 7

Miscellaneous operations

Saving, loading, and copying objects

MMTK provides an easy way to store (almost) arbitrary objects in files and retrieve them later. All objects of interest to users can be stored, including chemical objects, collections, universes, normal modes, configurations, etc. It is also possible to store standard Python objects such as numbers, lists, dictionaries etc., as well as practically any user-defined objects. Storage is based on the standard Python module `pickle`.

Objects are saved with MMTK.save and restored with MMTK.load. If several objects are to be stored in a single file, use tuples: `save((object1, object2), filename)` and `object1, object2 = load(filename)` to retrieve the objects.

Note that storing an object in a file implies storing all objects referenced by it as well, such that the size of the file can become larger than expected. For example, a configuration object contains a reference to the universe for which it is defined. Therefore storing a configuration object means storing the whole universe as well. However, nothing is ever written twice to the same file. If you store a list or a tuple containing a universe and a configuration for it, the universe is written only once.

It should be noted that when saving an object, all objects that this object refers to are also saved in the same file (otherwise the restored object would be missing some references). In practice this means that saving any chemical object, even a single atom, involves saving the whole universe that this object is part of. However, when saving several objects in one file,

objects referenced several times are saved only once.
Frequently it is also useful to copy an object, such as a molecule or a configuration. There are two functions (which are actually taken from the Python standard library module `copy`) for this purpose, which have a somewhat different behaviour for container-type objects (lists, dictionaries, collections etc.). `MMTK.copy(object)` returns a copy of the given object. For a container object, it returns a new container object which contains the same objects as the original one. If the intention is to get a container object which contains copies of the original contents, then `MMTK.deepcopy(object)` should be used. For objects that are not container-type objects, there is no difference between the two functions.

Exporting to specific file formats and visualization

MMTK can write objects in specific file formats that can be used by other programs. Three file formats are supported: the PDB format, widely used in computational chemistry, the DCD format for trajectories, written by the programs CHARMM and X-Plor and read by many visualization programs, and the VRML format, understood by VRML browsers as a representation of a three-dimensional scene for visualization. MMTK also provides a more general interface that can generate graphics objects in any representation if a special module for that representation exists. In addition to facilitating the implementation of new graphics file formats, this approach also permits the addition of custom graphics elements (lines, arrows, spheres, etc.) to molecular representations.

PDB, VRML, and DCD files

Any chemical object, collection, or universe can be written to a PDB or VRML file by calling the method `writeToFile`, defined in class MMTK.Collection.GroupOfAtoms (page 78). PDB files are read via the class MMTK.PDB.PDBConfiguration (page 124). DCD files can be read by a MMTK.DCD.DCDReader (page 83) object. For writing DCD files, there is the function MMTK.DCD.writeDCDPDB, which also creates a compatible PDB file without which the DCD file could not be interpreted.
Special care must be taken to ensure a correct mapping of atom numbers when reading from a DCD file. In MMTK, each atom object has a unique identity and atom numbers, also used internally for efficiency, are not strictly necessary and are not used anywhere in MMTK's application programming interface. DCD file, however, simply list coordinates sorted by atom number. For interpreting DCD files, another file must be available which allows the identification of atoms from their number and vice versa; this can for example be a PDB file.
When reading DCD files, MMTK assumes that the atom order in the DCD file is identical to the internal atom numbering of the universe for which the DCD file is read. This assumption is in general valid only if the universe has been created from a PDB file that is compatible with the DCD file, without any additions or removals.

Visualization and animation

The most common need for file export is visualization. All objects that can be visualized (chemical systems and subsets thereof, normal mode objects, trajectories) provide a method `view`which creates temporary export files, starts a visualization program, and deletes the temporary files. Depending on the object type there are various optional parameters.
MMTK also allows visualization of normal modes and trajectories using animation. Since not all visualization programs permit animation, and since there is no standard way to ask for it, animation is implemented only for the programs XMoland VMD. Animation is available for normal modes, trajectories, and arbitrary sequences of configurations (see function MMTK.Visualization.viewSequence).
For more specialized needs, MMTK permits the creation of graphical representations of most of its objects via general graphics modules that have to be provided externally. Suitable modules are provided in the package Scientific.Visualization and cover VRML (version 1), VRML2 (aka VRML97), and the molecular visualization program VMD. Modules for other representations (e.g. rendering programs) can be written easily; it is recommended to use the existing modules as an example. The generation of graphics objects is handled by the method `graphicsObjects`, defined in the class MMTK.Visualization.Viewable (page 156), which is a mix-in class that makes graphics objects generation available for all objects that define chemical systems or parts thereof, as well as for certain other objects that are viewable.
The explicit generation of graphics objects permits the mixture of different graphical representations for various parts of a system, as well as the combination of MMTK-generated graphics objects with arbitrary other graphics objects, such as lines, arrows, or spheres. All graphics objects are finally combined into a scene object (also defined in the various graphics modules) in order to be displayed. See also the visualization examples.

Fields

For analyzing or visualizing atomic properties that change little over short distances, it is often convenient to represent these properties as functions of position instead of one value per atom. Functions of position are also known as fields, and mathematical techniques for the analysis of fields have proven useful in many branches of physics. Such a field can be obtained by averaging over the values corresponding to the atoms in a small region of space. MMTK provides classes for scalar and vector field in module MMTK.Field (page 95). See also the example Miscellaneous/vector_field.py.

Charge fitting

A frequent problem in determining force field parameters is the determination of partial charges for the atoms of a molecule by fitting to the electrostatic potential around the molecule, which is obtained from quantum chemistry programs. Although this is essentially a straightforward linear least-squares problem, many procedures that are in common use do not use state-of-the-art techniques and may yield erroneous results. MMTK provides a charge fitting method that is numerically stable and allows the imposition of constraints on the charges. It is implemented in module MMTK.ChargeFit (page 73). See also the example Miscellaneous/charge_fit.py.

Chapter 8

Constructing the database

MMTK uses a database of chemical entities to define the properties of atoms, molecules, and related objects. This database consists of plain text files, more precisely short Python programs, whose names are the names of the object types. This chapter explains how to construct and manage these files. Note that the standard database already contains many definitions, in particular for proteins and nucleic acids. You do not need to read this chapter unless you want to add your own molecule definitions.

MMTK's database does not have to reside in a single place. It can consist of any number of subdatabases, each of which can be a directory or a URL. Typically the database consists of at least two parts: MMTK's standard definitions and a user's personal definitions. When looking up an object type in the database, MMTK checks the value of the environment variable `MMTKDATABASE`. The value of this variable must be a list of subdatabase locations seperated by white space. If the variable `MMTKDATABASE` is not defined, MMTK uses a default value that contains the path ".mmtk/Database" in the user's home directory followed by MMTK's standard database, which resides in the directory `Database` within the MMTK package directory (on many Unix systems this is /usr/local/lib/python2.2/site-packages/MMTK). MMTK checks the subdatabases in the order in which they are mentioned in `MMTKDATABASE`.

Each subdatabase contains directories corresponding to the object classes, i.e. Atoms (atom definitions), Groups (group definitions), Molecules (molecule definitions), Complexes (complex definitions), Proteins (protein definitions), and PDB (Protein Data Bank files). These directories contain the definition files, whose names may not contain any upper-case letters.

These file names correspond to the object types, e.g. the call `MMTK.Molecule('Water')`will cause MMTK to look for the file Molecules/water in the database (note that the names are converted to lower case).
The remaining sections of this chapter explain how the individual definition files are constructed. Keep in mind that each file is actually a Python program, so of course standard Python syntax rules apply.

Atom definitions

An atom definition in MMTK describes a chemical element, such as "hydrogen". This should not be confused with the "atom types" used in force field descriptions and in some modelling programs. As a consequence, it is rarely necessary to add atom definitions to MMTK.
Atom definition files are short and of essentially identical format. This is the definition for carbon:

```
name = 'carbon'
symbol = 'C'
mass = [(12, 98.90), (13.003354826, 1.10)]
color = 'black'
vdW_radius = 0.17
```

The name should be meaningful to users, but is not used by MMTK itself. The symbol, however, is used to identify chemical elements. It must be exactly equal to the symbol defined by IUPAC, including capitalization (e.g. 'Cl' for chlorine). The mass can be either a number or a list of tuples, as shown above. Each tuple defines an isotope by its mass and its percentage of occurrence; the percentages must add up to 100. The color is used for VRML output and must equal one of the color names defined in the module VRML. The van der Waals radius is used for the calculation of molecular volumes and surfaces; the values are taken from [Bondi1964].
An application program can create an isolated atom with `Atom('c')` or, specifying an initial position, with `Atom('c', position=Vector(0.,1.,0.))`. The element name can use any combination of upper and lower case letters, which are considered equivalent.

Group definitions

Group definitions in MMTK exist to facilitate the definition of molecules by avoiding the frequent repetition of common combinations. MMTK doesn't give any physical meaning to groups. Groups can contain atoms and other groups. Their definitions look exactly like molecule definitions; the only difference between groups and molecules is the way they are used.
This is the definition of a methyl group:

```
name = 'methyl group'
C  = Atom('C')
H1 = Atom('H')
H2 = Atom('H')
H3 = Atom('H')
bonds = [Bond(C, H1), Bond(C, H2), Bond(C, H3)]
pdbmap = [('MTH', {'C': C, 'H1': H1, 'H2': H2, 'H3': H3})]
amber_atom_type = {C: 'CT', H1: 'HC', H2: 'HC', H3: 'HC'}
amber_charge = {C: 0., H1: 0.1, H2: 0.1, H3: 0.1}
```

The name should be meaningful to users, but is not used by MMTK itself. The following lines create the atoms in the group and assign them to variables. These variables become attributes of whatever object uses this group; their names can be anything that is a legal Python name. The list of bonds, however, must be assigned to the variable "bonds". The bond list is used by force fields and for visualization.
The variable "pdbmap" is used for reading and writing PDB files. Its value must be a list of tuples, where each tuple defines one PDB residue. The first element of the tuple is the residue name, which is used only for output. The second element is a dictionary that maps PDB atom names to the actual atoms. The pdbmap entry of any object can be overridden by an entry in a higher-level object. Therefore the entry for a group is only used for atoms that do not occur in the entry for a molecule that contains this group.
The remaining lines in the definition file contain information specific to force fields, in this case the Amber force field. The dictionary "amber_atom_type" defines the atom type for each atom; the dictionary "amber_charge" defines the partial charges. As for pdbmap entries, these definitions can be overridden by higher-level definitions.

Molecule definitions

Molecules are typically used directly in application programs, but they can also be used in the definition of complexes. Molecule definitions can use atoms and groups.
This is the definition of a water molecule:

```
name = 'water'
structure = \
   "  O\n" + \
   " / \\\n" + \
   "H   H\n"
O  = Atom('O')
H1 = Atom('H')
H2 = Atom('H')
bonds = [Bond(O, H1), Bond(O, H2)]
pdbmap = [('HOH', {'O': O, 'H1': H1, 'H2': H2})]
pdb_alternative = {'OH2': 'O'}
amber_atom_type = {O: 'OW', H1: 'HW', H2: 'HW'}
amber_charge = {O: -0.83400, H1: 0.41700, H2: 0.41700}
configurations = {
    'default': ZMatrix([[H1],
[O,  H1,  0.9572*Ang],
[H2, O,   0.9572*Ang,  H1,  104.52*deg]])
    }
```

The name should be meaningful to users, but is not used by MMTK itself. The structure is optional and not used by MMTK either. The following lines create the atoms in the group and assign them to variables. These variables become attributes of the molecule, i.e. when a water molecule is created in an application program by `w = Molecule('water')`, then `w.H1`will refer to its first hydrogen atom. The names of these variables can be any legal Python names. The list of bonds, however, must be assigned to the variable "bonds". The bond list is used by force fields and for visualization.
The variable "pdbmap" is used for reading and writing PDB files. Its value must be a list of tuples, where each tuple defines one PDB residue. The first element of the tuple is the residue name, which is used only for output. The second element is a dictionary that maps PDB atom names to the actual

atoms. The pdbmap entry of any object can be overridden by an entry in a higher-level object, i.e. in the case of a molecule a complex containing it. The variable ”pdb_alternative” allows to read PDB files that use non-standard names. When a PDB atom name is not found in the pdbmap, an attempt is made to translate it to another name using pdb_alternative. The two following lines in the definition file contain information specific to force fields, in this case the Amber force field. The dictionary ”amber_atom_type” defines the atom type for each atom; the dictionary ”amber_charge” defines the partial charges. As for pdbmap entries, these definitions can be overridden by higher-level definitions.
The variable ”configurations” can be defined to be a dictionary of configurations for the molecule. During the construction of a molecule, a configuration can be specified via an optional parameter, e.g. `w = Molecule('water', configuration='default')`. The names of the configurations can be arbitrary; only the name ”default” has a special meaning; it is applied by default if no other configuration is specified when constructing the molecule. If there is no default configuration, and no other configuration is explicitly specified, then the molecule is created with undefined atomic positions.
There are three ways of describing configurations:

- By a Z-Matrix:

  ```
  ZMatrix([[H1],
           [O,  H1,  0.9572*Ang],
           [H2, O,   0.9572*Ang,  H1,  104.52*deg]])
  ```

- By Cartesian coordinates:

  ```
  Cartesian({O:  ( 0.004, -0.00518, 0.0),
             H1: (-0.092, -0.00518, 0.0),
             H2: ( 0.028,  0.0875,  0.0)})
  ```

- By a PDB file:

  ```
  PDBFile('water.pdb')
  ```

  The PDB file must be in the database subdirectory PDB, unless a full path name is specified for it.

Complex definitions

Complexes are defined much like molecules, except that they are composed of molecules and atoms; no groups are allowed, and neither are bonds.

Protein definitions

Protein definitions can take many different forms, depending on the source of input data and the type of information that is to be stored. For proteins it is particularly useful that database definition files are Python programs with all their flexibility.
The most common way of constructing a protein is from a PDB file. This is an example for a protein definition:

```
name = 'insulin'
# Read the PDB file.
conf = PDBConfiguration('insulin.pdb')
# Construct the peptide chains.
chains = conf.createPeptideChains()
# Clean up
del conf
```

The name should be meaningful to users, but is not used by MMTK itself. The second command reads the sequences of all peptide chains from a PDB file. Everything which is not a peptide chain is ignored. The following line constructs a PeptideChain object (a special molecule) for each chain from the PDB sequence. This involves constructing positions for any missing hydrogen atoms. Finally, the temporary data ("conf") is deleted, otherwise it would remain in memory forever.
The net result of a protein definition file is the assignment of a list of molecules (usually PeptideChain objects) to the variable "chains". MMTK then constructs a protein object from it. To use the above example, an application program would use the command `p = Protein('insulin')`.
The construction of the protein involves one nontrivial (but automatic) step: the construction of disulfide bridges for pairs of cystein residues whose sulfur atoms have a distance of less then 2.5 Angstrom.

Chapter 9

Threads and parallelization

This chapter explains the use of threads by MMTK and MMTK's parallelization support. This is an advanced topic, and not essential for the majority MMTK applications. You need to read this chapter only if you use multiprocessor computers, or if you want to implement multi-threaded programs that use MMTK.

Threads are different execution paths through a program that are executed in parallel, at least in principle; real parallel execution is possible only on multiprocessor systems. MMTK makes use of threads in two ways, which are conceptually unrelated: parallelization of energy evaluation on shared-memory multiprocessor computers, and support for multithreaded applications. Thread support is not available on all machines; you can check if yous system supports threads by starting a Python interpreter and typing `import threading`. If this produces an error message, then your system does not support threads, otherwise it is available in Python and also in MMTK. If you do not have thread support in Python although you know that your operating system supports threads, you might have compiled your Python interpreter without thread support; in that case, MMTK does not have thread support either.

Another approach to parallelization is message passing: several processors work on a program and communicate via a fast network to share results. A standard library, called MPI (Message Passing Interface), has been developped for sharing data by message passing, and implementations are available for all parallel computers currently on the market. MMTK contains elementary support for parallelization by message passing: only the energy evaluation has been paralellized, using a data-replication

strategy, which is simple but not the most efficient for large systems. MPI support is disabled by default. Enabling it involves modifying the file Src/Setup.template prior to compilation of MMTK. Furthermore, an MPI-enabled installation of ScientificPython is required, and the mpipython executable must be used instead of the standard Python interpreter.
Threads and message passing can be used together to use a cluster of shared-memory machines most efficiently. However, this requires that the thread and MPI implementations being used work together; sometimes there are conflicts, for example due to the use of the same signal in both libraries. Refer to your system documentation for details.
The use of threads for parallelization on shared-memory systems is very simple: Just set the environment variable `MMTK_ENERGY_THREADS` to the desired value. If this variable is not defined, the default value is 1, i.e. energy evaluations are performed serially. For choosing an appropriate value for this environment variable, the following points should be considered:

- The number of energy evaluation threads should not be larger than the number of processors that are fully dedicated to the MMTK application. A larger number of threads does not lead to wrong results, but it can increase the total execution time.
- MMTK assumes that all processors are equally fast. If you use a heteregenous multiprocessor machine, in which the processors have different speeds, you might find that the total execution time is larger than without threads.
- The use of threads incurs some computational overhead. For very small systems, it might be faster not to use threads.
- Not all energy terms necessarily support threads. Of the force field terms that part of MMTK, only the multipole algorithms for electrostatic interactions does not support threads, but additional force fields defined outside MMTK might also be affected. MMTK automatically evaluates such energy terms with a single thread, such that there is no risk of getting wrong results. However, you might not get the performance you expect.
- If second derivatives of the potential energy are requested, energy evaluation is handled by a single thread. An efficient implementation

> of multi-threaded energy evaluation would require a separate copy of the second-derivative matrix per thread. This approach needs too much memory for big systems to be feasible. Since second derivatives are almost exclusively used for normal mode calculations, which need only a single energy evaluation, multi-thread support is not particularly important anyway.

Parallelization via message passing is somewhat more complicated. In the current MMTK parallelization model, all processors execute the same program and replicate all tasks, with the important exception of energy evaluation. Energy terms are divided evenly between the processors, and at the end the energy and gradient values are shared by all machines. This is the only step involving network communication. Like thread-based parallelization, message-passing parallelization does not support the evaluation of second derivatives.

A special problem with message-passing systems is input and output. The MMTK application must ensure that output files are written by only one processor, and that all processors correctly access input files, especially in the case of each processor having its own disk space. See the example MPI/md.pyfor illustration.

Multithreaded applications are applications that use multiple threads in order to simplify the implementation of certain algorithms, i.e. not necessarily with the goal of profiting from multiple processors. If you plan to write a multithreaded application that uses MMTK, you should first make sure you understand threading support in Python. In particular, you should keep in mind that the global interpreter lock prevents the effective use of multiple processors by Python code; only one thread at a time can execute interpreted Python code. C code called from Python can permit other threads to execute simultaneously; MMTK does this for energy evaluation, molecular dynamics integration, energy minimization, and normal mode calculation.

A general problem in multithreaded applications is access to resources that are shared among the threads. In MMTK applications, the most important shared resource is the description of the chemical systems, i.e. universe objects and their contents. Chaos would result if two threads tried to modify the state of a universe simultaneously, or even if one thread uses information that is simultaneously being modified by another thread. Synchronization is therefore a critical part of multithreaded application.

MMTK provides two synchronization aids, both of which described in the documentation of the class MMTK.Universe.Universe (page 147): the configuration change lock (methods `acquireConfigurationChangeLock` and `releaseConfigurationChangeLock`), and the universe state lock (methods `acquireReadStateChangeLock`, `releaseReadStateChangeLock`, `acquireWriteStateChangeLock`, and `releaseWriteStateChangeLock`). Only a few common universe operations manipulate the universe state lock in order to avoid conflicts with other threads; these methods are marked as thread-safe in the description. All other operations should only be used inside a code section that is protected by the appropriate manipulation of the state lock. The configuration change lock is less critical; it is used only by the molecular dynamics and energy minimization algorithms in MMTK.

Chapter 10

Reference for Module MMTK

MMTK is the base module of the Molecular Modelling Toolkit. It contains the most common objects and all submodules. As a convenience to the user, it also imports some commonly used objects from other libraries:

- `Vector` from `Scientific.Geometry`
- `Translation` and `Rotation` from `Scientific.Geometry.Transformation`
- `copy` and `deepcopy` from `copy`
- `stdin`, `stdout`, and `stderr` from `sys`

Class ParticleScalar: Scalar property defined for each particle

A subclass of MMTK.ParticleProperties.ParticleProperty (page 127). ParticleScalar objects can be added to each other and multiplied with scalars.

Methods:

- maximum()
  Returns the highest value for any particle.

- minimum()
  Returns the smallest value for any particle.
- applyFunction(function)
  Applies function to each value and returns the result as a new ParticleScalar object.

Class ParticleVector: Vector property defined for each particle

A subclass of MMTK.ParticleProperties.ParticleProperty (page 127). ParticleVector objects can be added to each other and multiplied with scalars or MMTK.ParticleScalar (page 63) objects; all of these operations result in another ParticleVector object. Multiplication with a vector or another ParticleVector object yields a MMTK.ParticleScalar (page 63) object containing the dot products for each particle. Multiplications that treat ParticleVectors as vectors in a 3N-dimensional space are implemented as methods.

Methods:

- length()
  Returns a ParticleScalar containing the length of the vector for each particle.
- norm()
  Returns the norm of the ParticleVector seen as a 3N-dimensional vector.
- dotProduct(other)
  Returns the dot product with other (a ParticleVector) treating both operands as 3N-dimensional vectors.
- massWeightedDotProduct(other)
  Returns the mass-weighted dot product with other(a ParticleVector object) treating both operands as 3N-dimensional vectors.
- dyadicProduct(other)
  Returns a MMTK.ParticleTensor (page 65) object representing the dyadic product with other (a ParticleVector).

Class Configuration: Configuration of a universe

A subclass of MMTK.ParticleVector (page 64).
Its instances represent a configuration of a universe, consisting of positions for all atoms (like in a ParticleVector) plus the geometry of the universe itself, e.g. the cell shape for periodic universes.

Class ParticleTensor: Rank-2 tensor property defined for each particle

A subclass of MMTK.ParticleProperties.ParticleProperty (page 127).
ParticleTensor objects can be added to each other and multiplied with scalars or MMTK.ParticleScalar (page 63) objects; all of these operations result in another ParticleTensor object.

Class Atom: Atom

A subclass of MMTK.ChemicalObjects.ChemicalObject (page 75).
Constructor: Atom(element, **—properties—)

element a string (not case sensitive) specifying the chemical element

properties optional keyword properties:

- position: the atom position (a vector)
- name: the atom name (a string)

Methods:

- setPosition(position)
  Changes the position to position.
- position(conf=`None`)
  Returns the position in configuration conf. If conf is `None`, use the current configuration. If the atom has not been assigned a position, the return value is `None`.

- setMass(mass)
  Set the atom mass to mass.

- bondedTo()
  Returns a list of all atoms to which a chemical bond exists.

Class Collection: Collection of chemical objects

A subclass of MMTK.Collection.GroupOfAtoms (page 78)and MMTK.Visualization.Viewable (page 156).
Collections permit the grouping of arbitrary chemical objects (atoms, molecules, etc.) into one object for the purpose of analysis or manipulation.
Constructor: Collection(objects=None)

objects a chemical object or a sequence of chemical objects that define the initial content of the collection.

Collections permit length inquiry, item extraction by indexing, and iteration, like any Python sequence object. Two collections can be added to yield a collection that contains the combined elements.

Methods:

- addObject(object)
  Adds object to the collection. If object is another collection or a list, all of its elements are added.

- removeObject(object)
  Removes object from the collection. If object is a collection or a list, each of its elements is removed. The object to be removed must be an element of the collection.

- selectShell(point, r1, r2=0.0)
  Return a collection of all elements whose distance from point is between r1 and r2.

- selectBox(p1, p2)
  Return a collection of all elements that lie within a box whose corners are given by p1 and p2.

- objectList(klass=`None`)
  Returns a list of all objects in the collection. If klass is not None, only objects whose class is equal to klass are returned.

- atomList()
  Returns a list containing all atoms of all objects in the collection.

- numberOfAtoms()
  Returns the total number of atoms in the objects of the collection.

- universe()
  Returns the universe of which the objects in the collection are part. If no such universe exists, the return value is `None`.

- map(function)
  Applies function to all objects in the collection and returns the list of the results. If the results are chemical objects, a Collection object is returned instead of a list.

- distanceConstraintList()
  Returns the list of distance constraints.

- numberOfDistanceConstraints()
  Returns the number of distance constraints.

- setBondConstraints(universe=`None`)
  Sets distance constraints for all bonds.

- removeDistanceConstraints(universe=`None`)
  Removes all distance constraints.

Class Molecule: Molecule

A subclass of MMTK.ChemicalObjects.ChemicalObject (page 75).
Molecules consist of atoms and groups linked by bonds.
Constructor: Molecule(species, **—properties—)

species a string (not case sensitive) that specifies the molecule name in the chemical database properties optional keyword properties:

- position: the center-of-mass position (a vector)
- configuration: the name of a configuration listed in the database definition of the molecule, which is used to initialize the atom positions. If no configuration is specified, the configuration named "default" will be used, if it exists. Otherwise the atom positions are undefined.
- name: the atom name (a string)

Methods:

- findHydrogenPositions()
  Find reasonable positions for hydrogen atoms that have no position assigned.

  This method uses a heuristic approach based on standard geometry data. It was developed for proteins and DNA and may not give good results for other molecules. It raises an exception if presented with a topology it cannot handle.

Class PartitionedCollection: Collection with cubic partitions

A subclass of MMTK.Collection (page 66).
A PartitionedCollection differs from a plain Collection by sorting its elements into small cubic cells. This makes adding objects slower, but geometrical operations like selectShell become much faster for a large number of objects.
Constructor: PartitionedCollection(partition_size, objects=None)

partition_size the edge length of the cubic cells

objects a chemical object or a sequence of chemical objects that define the initial content of the collection.

Methods:

- partitions()
  Returns a list of cubic partitions. Each partition is specified by a tuple containing two vectors (describing the diagonally opposite corners) and the list of objects in the partition.
- pairsWithinCutoff(cutoff)
  Returns a list containing all pairs of objects in the collection whose center-of-mass distance is less than cutoff.

Class PartitionedAtomCollection: Partitioned collection of atoms

A subclass of MMTK.PartitionedCollection (page 68).
PartitionedAtomCollection objects behave like PartitionedCollection atoms, except that they store only atoms. When a composite chemical object is added, its atoms are stored instead.
Constructor: PartitionedAtomCollection(partition_size, objects=None)

partition_size the edge length of the cubic cells

objects a chemical object or a sequence of chemical objects that define the initial content of the collection.

Class InfiniteUniverse: Infinite (unbounded and nonperiodic) universe.

A subclass of MMTK.Universe.Universe (page 147).
Constructor: InfiniteUniverse(forcefield=None)

forcefield a force field object, or `None` for no force field

Class Complex: Complex

A subclass of MMTK.ChemicalObjects.ChemicalObject (page 75).
A complex is an assembly of molecules that are not connected by chemical bonds.
Constructor: Complex(species, **—properties—)

species a string (not case sensitive) that specifies the complex name in the chemical database

properties optional keyword properties:

- position: the center-of-mass position (a vector)
- configuration: the name of a configuration listed in the database definition of the complex
- name: the atom name (a string)

Class AtomCluster: An agglomeration of atoms

A subclass of MMTK.ChemicalObjects.ChemicalObject (page 75).
An atom cluster acts like a molecule without any bonds or atom properties. It can be used to represent a group of atoms that are known to form a chemical unit but whose chemical properties are not sufficiently known to define a molecule.
Constructor: AtomCluster(atoms, **—properties—)

atoms a list of atom objects

properties optional keyword properties:

- position: the center-of-mass position (a vector)
- name: the atom name (a string)

Class OrthorhombicPeriodicUniverse: Periodic universe with orthorhombic elementary cell.

A subclass of MMTK.Universe.Universe (page 147).
Constructor: OrthorhombicPeriodicUniverse(shape, forcefield=None)

shape a sequence of length three specifying the edge lengths along the x, y, and z directions forcefield a force field object, or `None` for no force field

Methods:

- scaleSize(factor)
  Multiplies all edge lengths by factor.
- setVolume(volume)
  Multiplies all edge lengths by the same factor such that the cell volume becomes volume.

Class CubicPeriodicUniverse: Periodic universe with cubic elementary cell.

A subclass of MMTK.Universe.Universe (page 147).

shape a number specifying the edge length along the x, y, and z directions

forcefield a force field object, or `None` for no force field Constructor:
CubicPeriodicUniverse(shape, forcefield=None)

Functions

- save()
  Writes object to a newly created file with the name filename, for later retrieval by `load()`.
- load()
  Loads the file indicated by filename, which must have been produced by `save()`, and returns the object stored in that file.

Module MMTK.Biopolymers

Class ResidueChain: A chain of residues

A subclass of MMTK.Molecule (page 67).
This is an abstract base class that defines operations common to peptide chains and nucleic acid chains.

Methods:

- residuesOfType(*types)
  Returns a collection that contains all residues whose type (residue code) is contained in types.

- residues()
  Returns a collection containing all residues.

- sequence()
  Returns the sequence as a list of residue code.

Functions

- defineAminoAcidResidue()
  Adds a non-standard amino acid residue to the residue table. The definition of the residue must be accesible by full_namein the chemical database. The three-letter code is specified by code3, and an optional one-letter code can be specified by code1.

  Once added to the residue table, the new residue can be used like any of the standard residues in the creation of peptide chains.

- defineNucleicAcidResidue()
  Adds a non-standard nucleic acid residue to the residue table. The definition of the residue must be accesible by full_namein the chemical database. The residue code is specified by code.

  Once added to the residue table, the new residue can be used like any of the standard residues in the creation of nucleotide chains.

Module MMTK.ChargeFit

This module implements a numerically stable method (based on Singular Value Decomposition) to fit point charges to values of an electrostatic potential surface. Two types of constraints are avaiable: a constraint on the total charge of the system or a subset of the system, and constraints that force the charges of several atoms to be equal. There is also a utility function that selects suitable evaluation points for the electrostatic potential surface. For the potential evaluation itself, some quantum chemistry program is needed.
The charge fitting method is described in [Hinsen1997]. See also Miscellaneous/charge_fit.py.

Class ChargeFit: Fit of point charges to an electrostatic potential surface

Constructor: ChargeFit(system, points, constraints=None)

system any chemical object, usually a molecule

points a list of point/potential pairs (a vector for the evaluation point, a number for the potential), or a dictionary whose keys are Configuration objects and whose values are lists of point/potential pairs. The latter case permits combined fits for several conformations of the system.

constraints a list of constraint objects (TotalChargeConstraint and/or EqualityConstraint objects). If the constraints are inconsistent, a warning is printed and the result will satisfy the constraints only in a least-squares sense.

A ChargeFit object acts like a dictionary that stores the fitted charge value for each atom in the system.

Class TotalChargeConstraint: Constraint on the total system charge

To be used with MMTK.ChargeFit.ChargeFit (page 73).

Constructor: TotalChargeConstraint(object, charge)

object any object whose total charge is to be constrained

charge the total charge value

Class EqualityConstraint: Constraint forcing two charges to be equal

To be used with MMTK.ChargeFit.ChargeFit (page 73).
Constructor: EqualityConstraint(atom1, atom2), where atom1 and atom2 are the two atoms whose charges should be equal.
Any atom may occur in more than one EqualityConstraint object, in order to keep the charges of more than two atoms equal.

Functions

- evaluationPoints()
  Returns a list of n points suitable for the evaluation of the electrostatic potential around object. The points are chosen at random and uniformly in a shell around the object such that no point has a distance larger than largest from any atom or smaller than smallest from any non-hydrogen atom.

Module MMTK.ChemicalObjects

Class ChemicalObject: General chemical object

A subclass of MMTK.Collection.GroupOfAtoms (page 78)and
MMTK.Visualization.Viewable (page 156).
This is an Glossary:abstract-base-class that implements methods which are applicable to any chemical object (atom, molecule, etc.).

Methods:

- topLevelChemicalObject()
  Returns the highest-level chemical object of which the current object is a part.

- universe()
  Returns the universe to which the object belongs.

- bondedUnits()
  Returns a list containing the subobjects which can contain bonds. There are no bonds between any of the subobjects in the list.

- fullName()
  Returns the full name of the object. The full name consists of the proper name of the object preceded by the full name of its parent separated by a dot.

- distanceConstraintList()
  Returns the list of distance constraints.

- numberOfDistanceConstraints()
  Returns the number of distance constraints.

- setBondConstraints(universe=`None`)
  Sets distance constraints for all bonds.

- removeDistanceConstraints(universe=`None`)
  Removes all distance constraints.

- setRigidBodyConstraints(universe=None)
  Sets distance constraints that make the object fully rigid.

- getAtomProperty(atom, property)
  Returns the value of the specified property for the given atom from the chemical database.

  Note: the property is first looked up in the database entry for the object on which the method is called. If the lookup fails, the complete hierarchy from the atom to the top-level object is constructed and traversed starting from the top-level object until the property is found. This permits database entries for higher-level objects to override property definitions in its constituents.

  At the atom level, the property is retrieved from an attribute with the same name. This means that properties at the atom level can be defined both in the chemical database and for each atom individually by assignment to the attribute.

Class CompositeChemicalObject: Chemical object with subobjects

This is an Glossary:abstract-base-class that implements methods which can be used with any composite chemical object, i.e. any chemical object that is not an atom.

Methods:

- atomList()
  Returns a list containing all atoms in the object.

Class Group: Group of bonded atoms

A subclass of MMTK.ChemicalObjects.ChemicalObject (page 75).
Groups can contain atoms and other groups, and link them by chemical bonds. They are used to represent functional groups or any other part of a molecule that has a well-defined identity.
Groups cannot be created in application programs, but only in database definitions for molecules.
Constructor: Group(species, **—properties—)

species a string (not case sensitive) that specifies the group name in the chemical database

properties optional keyword properties:

- position: the center-of-mass position (a vector)
- name: the atom name (a string)

Functions

- isChemicalObject()
  Returns 1 if object is a chemical object.

Module MMTK.Collection

Class GroupOfAtoms: Anything that consists of atoms

This class is a mix-in class that defines a large set of operations which are common to all objects that consist of atoms, i.e. any subset of a chemical system. Examples are atoms, molecules, collections, or universes.

Methods:

- numberOfAtoms()
  Returns the number of atoms.
- numberOfPoints()
  Returns the number of geometrical points that define the object. It is currently always equal to the number of atoms, but could be different e.g. for quantum systems, in which each atom is described by a wave function or a path integral.
- numberOfFixedAtoms()
  Returns the number of atoms that are fixed, i.e. cannot move.
- degreesOfFreedom()
  Returns the number of mechanical degrees of freedom.
- atomCollection()
  Returns a collection containing all atoms in the object.
- atomsWithDefinedPositions(conf=`None`)
  Returns a collection of all atoms that have a definite position.
- mass()
  Returns the total mass.
- centerOfMass(conf=`None`)
  Returns the center of mass.
- centerAndMomentOfInertia(conf=`None`)
  Returns the center of mass and the moment of inertia tensor.

- rotationalConstants(conf=`None`)
  Returns a sorted array of rotational constants A, B, C in internal units.

- boundingBox(conf=`None`)
  Returns two opposite corners of a bounding box around the object. The bounding box is the smallest rectangular bounding box with edges parallel to the coordinate axes.

- boundingSphere(conf=`None`)
  Returns a sphere that contains all atoms in the object. This is *not* the minimal bounding sphere, just *some*bounding sphere.

- rmsDifference(conf1, conf2=`None`)
  Returns the RMS (root-mean-square) difference between the conformations of the object in two universe configurations, conf1and conf2 (the latter defaults to the current configuration).

- findTransformation(conf1, conf2=`None`)
  Returns the linear transformation that, when applied to the object in configuration conf1, minimizes the RMS distance to the conformation in conf2, and the minimal RMS distance. If conf2 is `None`, returns the transformation from the current configuration to conf1 and the associated RMS distance. The algorithm is described in [Kneller1990].

- translateBy(vector)
  Translates the object by the displacement vector.

- translateTo(position)
  Translates the object such that its center of mass is at position.

- normalizeConfiguration(repr=`None`)
  Applies a linear transformation such that the coordinate origin becomes the center of mass of the object and its principal axes of inertia are parallel to the three coordinate axes.

  A specific representation can be chosen by setting repr to Ir : x y z ¡–¿ b c a IIr : x y z ¡–¿ c a b IIIr : x y z ¡–¿ a b c Il : x y z ¡–¿ c b a IIl : x y z ¡–¿ a c b IIIl : x y z ¡–¿ b a c

- applyTransformation(`t`)
  Applies the transformation `t` to the object.

- displacementUnderTransformation(`t`)
  Returns the displacement vectors (in a ParticleVector) for the atoms in the object that correspond to the transformation `t`.

- rotateAroundCenter(axis_direction, angle)
  Rotates the object by the given angle around an axis that passes through its center of mass and has the given direction.

- rotateAroundOrigin(axis, angle)
  Rotates the object by the given angle around an axis that passes through the coordinate origin and has the given direction.

- rotateAroundAxis(point1, point2, angle)
  Rotates the object by the given angle around the axis that passes through point1 and point2

- writeToFile(filename, configuration=`None`, format=`None`)
  Writes a representation of the object in the given configuration to the file identified by filename. The format can be either "pdb" or "vrml"; if no format is specified, it is deduced from the filename. An optional subformat specification can be added to the format name, separated by a dot. The subformats of "pdb" are defined by the module `Scientific.IO.PDB`, the subformats of "vrml" are "wireframe" (the default, yielding a wireframe representation), "ball_and_stick" (yielding a ball-and-stick representation), "highlight" (like wireframe, but with a small sphere for all atoms that have an attribute "highlight" with a non-zero value), and "charge" (wireframe plus small spheres for the atoms with colors from a red-to-green color scale to indicate the charge).

- view(configuration=`None`, format=`'pdb'`)
  Starts an external viewer for the object in the given configuration. The optional parameter format indicates which format (and hence which viewer) should be used; the formats are "pdb" and "vrml". An optional subformat specification can be added to the format name, separated by a dot. The subformats of "pdb" are defined by the module `Scientific.IO.PDB`, the subformats of "vrml" are

"wireframe" (the default, yielding a wireframe representation), "ball_and_stick" (yielding a ball-and-stick representation), "highlight" (like wireframe, but with a small sphere for all atoms that have an attribute "highlight" with a non-zero value), and "charge" (wireframe plus small spheres for the atoms with colors from a red-to-green color scale to indicate the charge).

- kineticEnergy(velocities=`None`)
  Returns the kinetic energy.

- temperature(velocities=`None`)
  Returns the temperature.

- momentum(velocities=`None`)
  Returns the momentum.

- angularMomentum(velocities=`None`, conf=`None`)
  Returns the angular momentum.

- angularVelocity(velocities=`None`, conf=`None`)
  Returns the angular velocity.

- universe()
  Returns the universe of which the object is part. For an object that is not part of a universe, the result is `None`.

- charge()
  Returns the total charge of the object. This is defined only for objects that are part of a universe with a force field that defines charges.

- dipole(reference=`None`)
  Returns the total dipole moment of the object. This is defined only for objects that are part of a universe with a force field that defines charges.

- booleanMask()
  Returns a ParticleScalar object that contains a value of 1 for each atom that is in the object and a value of 0 for all other atoms in the universe.

Functions

- isCollection()
  Return 1 if object is a Collection.

Module MMTK.DCD

Class DCDReader: Reader for DCD trajectories (CHARMM/X-Plor)

A DCDReader reads a DCD trajectory and "plays back" the data as if it were generated directly by an integrator. The universe for which the DCD file is read must be perfectly compatible with the data in the file, including an identical internal atom numbering. This can be guaranteed only if the universe was created from a PDB file that is compatible with the DCD file without leaving out any part of the system.
Constructor: DCDReader(universe, **options)

universe the universe for which the information from the trajectory file is read

options keyword options:

- dcd_file: the name of the DCD trajecory file to be read
- actions: a list of actions to be executed periodically (default is none)

Reading is started by calling the reader object. All the keyword options listed above can be specified either when creating the reader or when calling it.
The following data categories and variables are available for output:

- category "time": time
- category "configuration": configuration

Functions

- writeDCDPDB()
  Write the configurations in conf_list (any sequence of Configuration objects) to a newly created DCD trajectory file with the name dcd_file_name. Also write the first configuration to a PDB file with the

name pdb_file_name; this PDB file has the same atom order as the DCD file. The time step between configurations can be specified by delta_t.

- writeVelocityDCDPDB()
  Write the velocities in vel_list (any sequence of ParticleVector objects) to a newly created DCD trajectory file with the name dcd_file_name. Also write the first configuration to a PDB file with the name pdb_file_name; this PDB file has the same atom order as the DCD file. The time step between configurations can be specified by delta_t.

Module MMTK.Deformation

This module implements deformational energies for use in the analysis of motions and conformational changes in macromolecules. A description of the techniques can be found in [Hinsen1998] and [Hinsen1999].

Class DeformationFunction: Infinite-displacement deformation function

Constructor: DeformationFunction(universe, range=0.7, cutoff=1.2, factor=46402.)

universe the universe for which the deformation function should be defined

range the range parameter r_0 in the pair interaction term

cutoff the cutoff used in the deformation calculation

factor a global scaling factor

The default values are appropriate for a C_alpha model of a protein with the global scaling described in the reference cited above.
A DeformationFunction object must be called with a single parameter, which is a ParticleVector object containing the infinitesimal displacements of the atoms for which the deformation is to be evaluated. The return value is a ParticleScalar object containing the deformation value for each atom.

Class NormalizedDeformationFunction: Normalized infinite-displacement deformation function

Constructor: NormalizedDeformationFunction(universe, range=0.7, cutoff=1.2, factor=46402.)

universe the universe for which the deformation function should be defined

range the range parameter r_0 in the pair interaction term

cutoff the cutoff used in the deformation calculation factor a global scaling factor

The default values are appropriate for a C_alpha model of a protein with the global scaling described in the reference cited above. The normalization is defined by equation 10 of reference 1.
A NormalizedDeformationFunction object must be called with a single parameter, which is a ParticleVector object containing the infinitesimal displacements of the atoms for which the deformation is to be evaluated. The return value is a ParticleScalar object containing the deformation value for each atom.

Class FiniteDeformationFunction: Finite-displacement deformation function

Constructor: FiniteDeformationFunction(universe, range=0.7, cutoff=1.2, factor=46402.)

universe the universe for which the deformation function should be defined

range the range parameter r_0 in the pair interaction term

cutoff the cutoff used in the deformation calculation

factor a global scaling factor

The default values are appropriate for a C_alpha model of a protein with the global scaling described in the reference cited above.
A FiniteDeformationFunction object must be called with a single parameter, which is a Configuration or a ParticleVector object containing the alternate configuration of the universe for which the deformation is to be evaluated. The return value is a ParticleScalar object containing the deformation value for each atom.

Class DeformationEnergyFunction: Infinite-displacement deformation energy function

The deformation energy is the sum of the deformation values over all atoms of a system.
Constructor: DeformationEnergyFunction(universe, range=0.7, cutoff=1.2, factor=46402.)

universe the universe for which the deformation energy should be defined

range the range parameter r_0 in the pair interaction term

cutoff the cutoff used in the deformation energy calculation

factor a global scaling factor

The default values are appropriate for a C_alpha model of a protein with the global scaling described in the reference cited above.
A DeformationEnergyFunction is called with one or two parameters. The first parameter is a ParticleVector object containing the infinitesimal displacements of the atoms for which the deformation energy is to be evaluated. The optional second argument can be set to a non-zero value to request the gradients of the energy in addition to the energy itself. In that case there are two return values (energy and the gradients in a ParticleVector object), otherwise only the energy is returned.

Class NormalizedDeformationEnergyFunction: Normalized infinite-displacement deformation energy function

The normalized deformation energy is the sum of the normalized deformation values over all atoms of a system.
Constructor: NormalizedDeformationEnergyFunction(universe, range=0.7, cutoff=1.2, factor=46402.)

universe the universe for which the deformation energy should be defined

range the range parameter r_0 in the pair interaction term

cutoff the cutoff used in the deformation energy calculation

factor a global scaling factor

The default values are appropriate for a C_alpha model of a protein with the global scaling described in the reference cited above. The normalization is defined by equation 10 of reference 1.
A NormalizedDeformationEnergyFunction is called with one or two parameters. The first parameter is a ParticleVector object containing the

infinitesimal displacements of the atoms for which the deformation energy is to be evaluated. The optional second argument can be set to a non-zero value to request the gradients of the energy in addition to the energy itself. In that case there are two return values (energy and the gradients in a ParticleVector object), otherwise only the energy is returned.

Class FiniteDeformationEnergyFunction: Finite-displacement deformation energy function

The deformation energy is the sum of the deformation values over all atoms of a system.
Constructor: FiniteDeformationEnergyFunction(universe, range=0.7, cutoff=1.2, factor=46402.)

universe the universe for which the deformation energy should be defined

range the range parameter r_0 in the pair interaction term

cutoff the cutoff used in the deformation energy calculation

factor a global scaling factor

The default values are appropriate for a C_alpha model of a protein with the global scaling described in the reference cited above.
A FiniteDeformationEnergyFunction is called with one or two parameters. The first parameter is a ParticleVector object containing the alternate configuration of the universe for which the deformation energy is to be evaluated. The optional second argument can be set to a non-zero value to request the gradients of the energy in addition to the energy itself. In that case there are two return values (energy and the gradients in a ParticleVector object), otherwise only the energy is returned.

Class DeformationReducer: Iterative reduction of the deformation energy

Constructor: DeformationReducer(universe, range=0.7, cutoff=1.2, factor=46402.)

universe the universe for which the deformation function should be defined range the range parameter r_0 in the pair interaction term

cutoff the cutoff used in the deformation calculation

factor a global scaling factor

The default values are appropriate for a C_alpha model of a protein with the global scaling described in the reference cited above.
A DeformationReducer is called with two arguments. The first is a ParticleVector containing the initial infinitesimal displacements for all atoms. The second is an integer indicating the number of iterations. The result is a modification of the displacements by steepest-descent minimization of the deformation energy.

Class FiniteDeformationReducer: Iterative reduction of the finite-displacement deformation energy

Constructor: FiniteDeformationReducer(universe, range=0.7, cutoff=1.2, factor=46402.)

universe the universe for which the deformation function should be defined

range the range parameter r_0 in the pair interaction term

cutoff the cutoff used in the deformation calculation

factor a global scaling factor

The default values are appropriate for a C_alpha model of a protein with the global scaling described in the reference cited above.
A FiniteDeformationReducer is called with two arguments. The first is a ParticleVector or Configuration containing the alternate configuration for which the deformation energy is evaluated. The second is the RMS distance that defines the termination condition. The return value a configuration that differs from the input configuration by approximately the specified RMS distance, and which is obtained by iterative steepest-descent minimization of the finite-displacement deformation energy.

Module MMTK.Dynamics

See also the Molecular Dynamics example applications.

Class VelocityVerletIntegrator: Velocity-Verlet molecular dynamics integrator

The integrator can handle fixed atoms, distance constraints, a thermostat, and a barostat, as well as any combination. It is fully thread-safe.
Constructor: VelocityVerletIntegrator(universe, **options)

universe the universe on which the integrator acts

options keyword options:

- steps: the number of integration steps (default is 100)
- delta_t: the time step (default is 1 fs)
- actions: a list of actions to be executed periodically (default is none)
- threads: the number of threads to use in energy evaluation (default set by MMTK_ENERGY_THREADS)
- background: if true, the integration is executed as a separate thread (default: 0)
- mpi_communicator: an MPI communicator object, or None, meaning no parallelization (default: None)

The integration is started by calling the integrator object. All the keyword options listed above can be specified either when creating the integrator or when calling it.
The following data categories and variables are available for output:

- category ”time”: time
- category ”configuration”: configuration and box size (for periodic universes)
- category ”velocities”: atomic velocities

- category "gradients": energy gradients for each atom
- category "energy": potential and kinetic energy, plus extended-system energy terms if a thermostat and/or barostat are used
- category "thermodynamic": temperature, volume (if a barostat is used) and pressure
- category "auxiliary": extended-system coordinates if a thermostat and/or barostat are used

Class VelocityScaler: Periodic velocity scaling action

A VelocityScaler object is used in the action list of a VelocityVerletIntegrator. It rescales all atomic velocities by a common factor to make the temperature of the system equal to a predefined value.
Constructor: VelocityScaler(temperature, temperature_window=0., first=0, last=None, skip=1)

temperature the temperature value to which the velocities should be scaled

temperature_window the deviation from the ideal temperature that is tolerated in either direction before rescaling takes place

first the number of the first step at which the action is executed

last the number of the last step at which the action is executed. A value of `None` indicates that the action should be executed indefinitely.

skip the number of steps to skip between two applications of the action

Class Heater: Periodic heating action

A Heater object us used in the action list of a VelocityVerletIntegrator. It scales the velocities to a temperature that increases with time.
Constructor: Heater(temperature1, temperature2, gradient, first=0, last=None, skip=1)

temperature1 the temperature value to which the velocities should be scaled initially temperature2 the final temperature value to which the velocities should be scaled

gradient the temperature gradient (in K/ps)

first the number of the first step at which the action is executed

last the number of the last step at which the action is executed. A value of `None` indicates that the action should be executed indefinitely.

skip the number of steps to skip between two applications of the action

Class BarostatReset: Barostat reset action

A BarostatReset object is used in the action list of a VelocityVerletIntegrator. It resets the barostat coordinate to zero.
Constructor: BarostatReset(first=0, last=None, skip=1)

first the number of the first step at which the action is executed

last the number of the last step at which the action is executed. A value of `None` indicates that the action should be executed indefinitely.

skip the number of steps to skip between two applications of the action

Class TranslationRemover: Action that eliminates global translation

A TranslationRemover object is used in the action list of a VelocityVerletIntegrator. It subtracts the total velocity from the system from each atomic velocity.
Constructor: TranslationRemover(first=0, last=None, skip=1)

first the number of the first step at which the action is executed

last the number of the last step at which the action is executed. A value of `None` indicates that the action should be executed indefinitely.

skip the number of steps to skip between two applications of the action

Class RotationRemover: Action that eliminates global rotation

A RotationRemover object is used in the action list of a VelocityVerletIntegrator. It adjusts the atomic velocities such that the total angular momentum is zero.
Constructor: RotationRemover(first=0, last=None, skip=1)

first the number of the first step at which the action is executed

last the number of the last step at which the action is executed. A value of `None` indicates that the action should be executed indefinitely.

skip the number of steps to skip between two applications of the action

Module MMTK.Environment

Class NoseThermostat: Nose thermostat for Molecular Dynamics

A thermostat object can be added to a universe and will then modify the integration algorithm to a simulation of an NVT ensemble.
Constructor: NoseThermostat(temperature, relaxation_time=0.2)

temperature the temperature set by the thermostat

relaxation_time the relaxation time of the thermostat coordinate

Class AndersenBarostat: Andersen barostat for Molecular Dynamics

A barostat object can be added to a universe and will then together with a thermostat object modify the integration algorithm to a simulation of an NPT ensemble.
Constructor: AndersenBarostat(pressure, relaxation_time=1.5)

pressure the pressure set by the barostat

relaxation_time the relaxation time of the barostat coordinate

Module MMTK.Field

This module defines field objects that are useful in the analysis and visualization of collective motions in molecular systems. Atomic quantities characterizing collective motions vary slowly in space, and can be considered functions of position instead of values per atom. Functions of position are called fields, and mathematical techniques for the analysis of fields have proven useful in many branches of physics. Fields can be described numerically by values on a regular grid. In addition to permitting the application of vector analysis methods to atomic quantities, the introduction of fields is a valuable visualization aid, because information defined on a coarse regular grid can be added to a picture of a molecular system without overloading it. See also the example Miscellaneous/vector_field.py.

Class AtomicField: A field whose values are determined by atomic quantities

This is an Glossary:abstract-base-class. To create field objects, use one of its subclasses.

Methods:

- particleValues()
  Returns the values of the field at the positions of the atoms in an appropriate subclass of MMTK.ParticleProperties.ParticleProperty (page 127).

- writeToFile(filename, scale=`1.0`, color=`None`)
  Writes a graphical representation of the field to the VRML file named by filename, multiplying all values by scale. color permits the choice of a color for the graphics objects.

- view(scale=`1.0`, color=`None`)
  Shows a graphical representation of the field using a VRML viewer. All values are multiplied by scale. colorpermits the choice of a color for the graphics objects.

Class AtomicScalarField: Scalar field defined by atomic quantities

A subclass of MMTK.Field.AtomicField (page 95) and MMTK.Visualization.Viewable (page 156).
Constructor: AtomicScalarField(system, grid_size, values)

system any subset of a molecular system

grid_size the spacing of a cubic grid on which the field values are stored

values an object of class MMTK.ParticleScalar (page 63) containing the atomic values that define the field

The field values are obtained by averaging the atomic quantities over all atoms in a cube of edge length grid_size surrounding each grid point.
The method graphicsObjects, defined in class MMTK.Visualization.Viewable (page 156), returns a small cube for each grid point, whose color indicates the field's value on a symmetric red-to-green color scale defined by the range of the field values. Additional keyword options are:
– scale=factor, to multiply all field values by a factor
– range=(min, max), to eliminate graphics objects for values that are smaller than min or larger than max

Methods:

- gradient()
  Returns an MMTK.Field.AtomicVectorField (page 96) object representing the gradient of the field.
- laplacian()
  Returns an MMTK.Field.AtomicScalarField (page 96) object representing the laplacian of the field.

Class AtomicVectorField: Vector field defined by atomic quantities

A subclass of MMTK.Field.AtomicField (page 95) and MMTK.Visualization.Viewable (page 156).
Constructor: AtomicVectorField(system, grid_size, values)

system any subset of a molecular system

grid_size the spacing of a cubic grid on which the field values are stored

values an object of class MMTK.ParticleVector (page 64) containing the atomic values that define the field

The field values are obtained by averaging the atomic quantities over all atoms in a cube of edge length grid_size surrounding each grid point.
The method graphicsObjects, defined in class MMTK.Visualization.Viewable (page 156), returns a small arrow for each grid point. The arrow starts at the grid point and represents the vector value at that point. Additional keyword options are:
– scale=factor, to multiply all field values by a factor
– diameter=number, to define the diameter of the arrow objects (default: 1.)
– range=(min, max), to eliminate graphics objects for values whose lengths are smaller than min or larger than max
– color=string, to define the color of the arrows

Methods:

- length()
  Returns an MMTK.Field.AtomicScalarField (page 96) object representing the length of the field vectors.

- divergence()
  Returns an MMTK.Field.AtomicScalarField (page 96) object representing the divergence of the field.

- curl()
  Returns an MMTK.Field.AtomicVectorField (page 96) object representing the curl of the field.

- laplacian()
  Returns an MMTK.Field.AtomicVectorField (page 96) object representing the laplacian of the field.

Module MMTK.ForceFields

Class CalphaForceField: Effective harmonic force field for a C-alpha protein model

Constructor: CalphaForceField(cutoff=None, scale_factor=1.)

cutoff the cutoff for pair interactions, should be at least 2.5 nm

scale_factor a global scaling factor.

Pair interactions in periodic systems are calculated using the minimum-image convention; the cutoff should therefore never be larger than half the smallest edge length of the elementary cell.
See [Hinsen2000] for a description of this force field.

Class DeformationForceField: Deformation force field for protein normal mode calculations

Constructor: DeformationForceField(range=0.7, cutoff=1.2, factor=46402.)

range the range parameter

cutoff the cutoff for pair interactions, should be significantly larger than range.

factor a global scaling factor.

Pair interactions in periodic systems are calculated using the minimum-image convention; the cutoff should therefore never be larger than half the smallest edge length of the elementary cell.
The pair interaction energy has the form U(r)=—factor—*exp(-(r-0.01)**2/—range—**2). The default value for range is appropriate for a C-alpha model of a protein. See [Hinsen1998] for details.

Class LennardJonesForceField: Lennard-Jones force field for noble gases

Constructor: LennardJonesForceField(cutoff)

cutoff a cutoff value or `None`, meaning no cutoff

Pair interactions in periodic systems are calculated using the minimum-image convention; the cutoff should therefore never be larger than half the smallest edge length of the elementary cell.
The Lennard-Jones parameters are taken from the atom attributes LJ_radius and LJ_energy. The pair interaction energy has the form U(r)=4*LJ_energy*((LJ_radius/r)**12-(LJ_radius/r)**6).

Class Amber94ForceField: Amber 94 force field

Constructor: Amber94ForceField(lennard_jones_options, electrostatic_options)

lennard_jones_options parameters for Lennard-Jones interactions; one of:

- a number, specifying the cutoff
- `None`, meaning the default method (no cutoff; inclusion of all pairs, using the minimum-image conventions for periodic universes)
- a dictionary with an entry "method" which specifies the calculation method as either "direct" (all pair terms) or "cutoff", with the cutoff specified by the dictionary entry "cutoff".

electrostatic_options parameters for electrostatic interactions; one of:

- a number, specifying the cutoff
- `None`, meaning the default method (all pairs without cutoff for non-periodic system, Ewald summation for periodic systems)
- a dictionary with an entry "method" which specifies the calculation method as either "direct" (all pair terms), "cutoff" (with the cutoff specified by the dictionary entry "cutoff"), "ewald" (Ewald summation, only for periodic universes), "screened" (see below), or "multipole" (fast-multipole method).

Pair interactions in periodic systems are calculated using the minimum-image convention; the cutoff should therefore never be larger than half the smallest edge length of the elementary cell.
For Lennard-Jones interactions, all terms for pairs whose distance exceeds the cutoff are set to zero, without any form of correction. For electrostatic interactions, a charge-neutralizing surface charge density is added around the cutoff sphere in order to reduce cutoff effects [Wolf1999].
For Ewald summation, there are some additional parameters that can be specified by dictionary entries:

- "beta" specifies the Ewald screening parameter
- "real_cutoff" specifies the cutoff for the real-space sum. It should be significantly larger than 1/beta to ensure that the neglected terms are small.
- "reciprocal_cutoff" specifies the cutoff for the reciprocal-space sum. Note that, like the real-space cutoff, this is a distance; it describes the smallest wavelength of plane waves to take into account. Consequently, a smaller value means a more precise (and more expensive) calculation.

MMTK provides default values for these parameter which are calculated as a function of the system size. However, these parameters are exaggerated in most cases of practical interest and can lead to excessive calculation times for large systems. It is preferable to determine suitable values empirically for the specific system to be simulated.
The method "screened" uses the real-space part of the Ewald sum with a charge-neutralizing surface charge density around the cutoff sphere, and no reciprocal sum [Wolf1999]. It requires the specification of the dictionary entries "cutoff" and "beta".
The fast-multipole method uses the DPMTA library [DPMTA]. Note that this method provides only energy and forces, but no second-derivative matrix. There are several optional dictionary entries for this method, all of which are set to reasonable default values. The entries are "spatial_decomposition_levels", "multipole_expansion_terms", "use_fft", "fft_blocking_factor", "macroscopic_expansion_terms", and "multipole_acceptance". For an explanation of these options, refer to the DPMTA manual [DPMTA].

Module MMTK.ForceFields.BondFF

Class HarmonicForceField: Simplified harmonic force field for normal mode calculations

Constructor: HarmonicForceField()
This force field is made up of the bonded terms from the Amber 94 force field with the equilibrium positions of all terms changed to the corresponding values in the input configuration, such that the input configuration becomes an energy minimum by construction. The nonbonded terms are replaced by a generic short-ranged deformation term. See [Hinsen1999b] for a description of this force field, and [Viduna2000] for an application to DNA.

Module MMTK.ForceFields.ForceFieldTest

Force field consistency tests
To be documented later!

Module MMTK.ForceFields.Restraints

This module contains harmonic restraint terms that can be added to any force field.
Example:
from MMTK import * from MMTK.ForceFields import Amber94ForceField from MMTK.ForceFields.Restraints import HarmonicDistanceRestraint universe = InfiniteUniverse() universe.protein = Protein(bala1) force_field = Amber94ForceField() + HarmonicDistanceRestraint(universe.protein[0][1].peptide.N, universe.protein[0][1].peptide.O, 0.5, 10.) universe.setForceField(force_field)

Class HarmonicDistanceRestraint: Harmonic distance restraint between two atoms

Constructor: HarmonicDistanceRestraint(atom1, atom2, distance, force_constant)

atom1, atom2 the two atoms whose distance is restrained distance the distance at which the restraint is zero

force_constant the force constant of the restraint term

The functional form of the restraint is
—force_constant—*((r1-r2).length()-—distance—)**2, where r1 and r2 are the positions of the two atoms.

Class HarmonicAngleRestraint: Harmonic angle restraint between three atoms

Constructor: HarmonicAngleRestraint(atom1, atom2, atom3, angle, force_constant)

atom1, atom2, atom3 the three atoms whose angle is restrained; atom2 is the central atom

angle the angle at which the restraint is zero

force_constant the force constant of the restraint term

The functional form of the restraint is
—force_constant—*(phi-—angle—)**2, where phi is the angle
—atom1—-—atom2—-—atom3—.

Class HarmonicDihedralRestraint: Harmonic dihedral angle restraint between three atoms

Constructor: HarmonicDihedralRestraint(atom1, atom2, atom3, atom4, angle, force_constant)

atom1, atom2, atom3, atom4 the four atoms whose dihedral angle is restrained; atom2 and atom3 are on the common axis

angle the dihedral angle at which the restraint is zero

force_constant the force constant of the restraint term

The functional form of the restraint is
—force_constant—*(phi-—distance—)**2, where phi is the dihedral angle
—atom1—-—atom2—-—atom3—-—atom4—.

Module MMTK.ForceFields.SPCEFF

Class SPCEForceField: Force field for water simulations with the SPC/E model

Constructor: SPCEForceField(lennard_jones_options, electrostatic_options)
The meaning of the arguments is the same as for the class
[MMTK.ForceFields.Amber94ForceField (page 99)]

Module MMTK.FourierBasis

This module provides a basis that is suitable for the calculation of low-frequency normal modes. The basis is derived from vector fields whose components are stationary waves in a box surrounding the system. For a description see [Hinsen1998].

Class FourierBasis: Collective-motion basis for low-frequency normal mode calculations

To be used with MMTK.NormalModes.SubspaceNormalModes (page 116).
Constructor: FourierBasis(universe, cutoff)

universe the universe for which the basis will be used

cutoff the wavelength cutoff. A smaller value means a larger basis.

A FourierBasis object behaves like a sequence of MMTK.ParticleVector (page 64) objects. The vectors are *not*orthonormal, because orthonormalization is handled automatically by the class MMTK.NormalModes.SubspaceNormalModes (page 116).

Functions

- countBasisVectors()
  Returns the number of basis vectors in a FourierBasis for the given universe and cutoff.

- estimateCutoff()
  Returns an estimate for the cutoff that will yield a basis of nmodes vectors for the given universe. The two return values are the cutoff and the precise number of basis vectors for this cutoff.

Module MMTK.Geometry

This module defines several elementary geometrical objects, which can be useful in the construction and analysis of molecular systems. There are essentially two kinds of geometrical objects: shape objects (spheres, planes, etc.), from which intersections can be calculated, and lattice objects, which define a regular arrangements of points.

Class GeometricalObject3D: 3D shape object

This is an Glossary:abstract-base-class. To create 3D objects, use one of its subclasses.

Methods:

- intersectWith(other)
  Return a 3D object that represents the intersection with other(another 3D object).

Class Box: Box

A subclass of MMTK.Geometry.GeometricalObject3D (page 105).
Constructor: Box(corner1, corner2)

corner1, corner2 diagonally opposite corner points

Class Sphere: Sphere

A subclass of MMTK.Geometry.GeometricalObject3D (page 105).
Constructor: Sphere(center, radius)

center the center of the sphere (a vector)

radius the radius of the sphere (a number)

Class Cylinder: Cylinder

A subclass of MMTK.Geometry.GeometricalObject3D (page 105).
Constructor: Cylinder(center1, center2, radius)

center1 the center of the bottom circle (a vector)

center2 the center of the top circle (a vector)

radius the radius (a number)

Class Plane: 2D plane in 3D space

A subclass of MMTK.Geometry.GeometricalObject3D (page 105).
Constructor: Plane(point, normal) or Plane(point1, point2, point3)

point any point in the plane

normal the normal vector of the plane

point1, point2, point3 three points in the plane that are not collinear.

Class Cone: Cone

A subclass of MMTK.Geometry.GeometricalObject3D (page 105).
Constructor: Cone(center, axis, angle)

center the center (tip) of the cone (a vector)

axis the direction of the axis of rotational symmetry (a vector)

angle the angle between any straight line on the cone surface and the axis of symmetry (a number)

Class Circle: 2D circle in 3D space

A subclass of MMTK.Geometry.GeometricalObject3D (page 105).
Constructor: Circle(center, normal, radius)

center the center of the circle (a vector)

normal the normal vector of the plane of the sphere (vector)

radius the radius of the circle (a number)

Class Line: Line

A subclass of MMTK.Geometry.GeometricalObject3D (page 105).
Constructor: Line(point, direction)

point any point on the line (a vector)

direction the direction of the line (a vector)

Methods:

- distanceFrom(point)
  Returns the smallest distance of point from the line.

- projectionOf(point)
  Returns the orthogonal projection of point onto the line.

Class Lattice: General lattice

Lattices are special sequence objects that contain vectors (points on the lattice) or objects that are constructed as functions of these vectors. Lattice objects behave like lists, i.e. they permit indexing, length inquiry, and iteration by 'for'-loops. See also the example Miscellaneous/lattice.py. This is an Glossary:abstract-base-class. To create lattice objects, use one of its subclasses.

Class RhombicLattice: Rhombic lattice

A subclass of MMTK.Geometry.Lattice (page 107).
Constructor: RhombicLattice(elementary_cell, lattice_vectors, cells, function=`None`)

elementary_cell a list of points (vectors) in the elementary cell

lattice_vectors a list of lattice vectors. Each lattice vector defines a lattice dimension (only values from one to three make sense) and indicates the displacement along this dimension from one cell to the next.

cells a list of integers, whose length must equal the number of dimensions. Each entry specifies how often a cell is repeated along this dimension.

function a function that is called for every lattice point with the vector describing the point as argument. The return value of this function is stored in the lattice object. If the function is `None`, the vector is directly stored in the lattice object.

The number of objects in the lattice is equal to the product of the values in cells times the number of points in elementary_cell.

Class BravaisLattice: Bravais lattice

A subclass of MMTK.Geometry.Lattice (page 107).
A Bravais lattice is a special case of a general rhombic lattice in which the elementary cell contains only one point.
Constructor: BravaisLattice(lattice_vectors, cells, function=`None`)

lattice_vectors a list of lattice vectors. Each lattice vector defines a lattice dimension (only values from one to three make sense) and indicates the displacement along this dimension from one cell to the next.

cells a list of integers, whose length must equal the number of dimensions. Each entry specifies how often a cell is repeated along this dimension.

function a function that is called for every lattice point with the vector describing the point as argument. The return value of this function is stored in the lattice object. If the function is `None`, the vector is directly stored in the lattice object.

The number of objects in the lattice is equal to the product of the values in cells.

Class SCLattice: Simple Cubic lattice

A subclass of MMTK.Geometry.Lattice (page 107).
A Simple Cubic lattice is a special case of a Bravais lattice in which the elementary cell is a cube.
Constructor: SCLattice(cell_size, cells, function=`None`)

cell_size the edge length of the elementary cell cells a list of integers, whose length must equal the number of dimensions. Each entry specifies how often a cell is repeated along this dimension.

function a function that is called for every lattice point with the vector describing the point as argument. The return value of this function is stored in the lattice object. If the function is `None`, the vector is directly stored in the lattice object.

The number of objects in the lattice is equal to the product of the values in cells.

Class BCCLattice: Body-Centered Cubic lattice

A subclass of MMTK.Geometry.Lattice (page 107).
A Body-Centered Cubic lattice has two points per elementary cell.
Constructor: BCCLattice(cell_size, cells, function=`None`)

cell_size the edge length of the elementary cell

cells a list of integers, whose length must equal the number of dimensions. Each entry specifies how often a cell is repeated along this dimension.

function a function that is called for every lattice point with the vector describing the point as argument. The return value of this function is stored in the lattice object. If the function is `None`, the vector is directly stored in the lattice object.

The number of objects in the lattice is equal to the product of the values in cells.

Class FCCLattice: Face-Centered Cubic lattice

A subclass of MMTK.Geometry.Lattice (page 107).
A Face-Centered Cubic lattice has four points per elementary cell.
Constructor: FCCLattice(cell_size, cells, function=`None`)

cell_size the edge length of the elementary cell

cells a list of integers, whose length must equal the number of dimensions. Each entry specifies how often a cell is repeated along this dimension.

function a function that is called for every lattice point with the vector describing the point as argument. The return value of this function is stored in the lattice object. If the function is `None`, the vector is directly stored in the lattice object.

The number of objects in the lattice is equal to the product of the values in cells.

Module MMTK.Minimization

Class SteepestDescentMinimizer: Steepest-descent minimizer

The minimizer can handle fixed atoms, but no distance constraints. It is fully thread-safe.
Constructor: SteepestDescentMinimizer(universe, **options)

universe the universe on which the minimizer acts

options keyword options:

- steps: the number of minimization steps (default is 100)
- step_size: the initial size of a minimization step (default is 0.002 nm)
- convergence: the root-mean-square gradient length at which minimization stops (default is 0.01 kJ/mol/nm)
- actions: a list of actions to be executed periodically (default is none)
- threads: the number of threads to use in energy evaluation (default set by MMTK_ENERGY_THREADS)
- background: if true, the minimization is executed as a separate thread (default: 0)
- mpi_communicator: an MPI communicator object, or None, meaning no parallelization (default: None)

The minimization is started by calling the minimizer object. All the keyword options listed above can be specified either when creating the minimizer or when calling it.
The following data categories and variables are available for output:

- category "configuration": configuration and box size (for periodic universes)

- category "gradients": energy gradients for each atom
- category "energy": potential energy and norm of the potential energy gradient

Class ConjugateGradientMinimizer: Conjugate gradient minimizer

The minimizer can handle fixed atoms, but no distance constraints. It is fully thread-safe.
Constructor: ConjugateGradientMinimizer(universe, **options)

universe the universe on which the minimizer acts

options keyword options:

- steps: the number of minimization steps (default is 100)
- step_size: the initial size of a minimization step (default is 0.002 nm)
- convergence: the root-mean-square gradient length at which minimization stops (default is 0.01 kJ/mol/nm)
- actions: a list of actions to be executed periodically (default is none)
- threads: the number of threads to use in energy evaluation (default set by MMTK_ENERGY_THREADS)
- background: if true, the minimization is executed as a separate thread (default: 0)

The minimization is started by calling the minimizer object. All the keyword options listed above can be specified either when creating the minimizer or when calling it.
The following data categories and variables are available for output:

- category "configuration": configuration and box size (for periodic universes)
- category "gradients": energy gradients for each atom
- category "energy": potential energy and norm of the potential energy gradient

Module MMTK.MolecularSurface

This module provides functions that calculate molecular surfaces and volumes.

Functions

- surfaceAndVolume()
  Returns the molecular surface and volume of object, defining the surface at a distance of probe_radius from the van-der-Waals surfaces of the atoms.

- surfaceAtoms()
  Returns a dictionary that maps the surface atoms to their exposed surface areas.

- surfacePointsAndGradients()
  Returns a dictionary that maps the surface atoms to a tuple containing three surface-related quantities: the exposed surface are, a list of points in the exposed surface, and a gradient vector pointing outward from the surface.

- findContacts()
  Returns a list of MMTK.MolecularSurface.Contact objects that describe atomic contacts between object1 and object2. A contact is defined as a pair of atoms whose distance is less than —contact_factor—*(r1+r2+—cutoff—) where r1 and r2 are the atomic van-der-Waals radii.

Module MMTK.NormalModes

See also the normal mode example applications.

Class Mode: Single normal mode

A subclass of MMTK.ParticleVector (page 64).
Mode objects are created by indexing a NormalModes object. They contain the atomic displacements corresponding to a single mode.
Mode objects are specializations of MMTK.ParticleVector (page 64)objects and support all their operations. In addition, the frequency corresponding to the mode is stored in the attribute "frequency".
Note: the normal mode vectors are *not* mass weighted, and therefore not orthogonal to each other.

Methods:

- view(factor=`1.0`, subset=`None`)
  Start an animation of the mode. The displacements can be scaled by a factor to make them better visible, and a subset of the total system can be specified as well. This function requires an external viewer, see module MMTK.Visualization (page 156) for details.

Class NormalModes: Normal modes

Constructor: NormalModes(universe, temperature=300)

universe the system for which the normal modes are calculated; it must have a force field which provides the second derivatives of the potential energy

temperature the temperature for which the amplitudes of the atomic displacement vectors are calculated. A value of `None` can be specified to have no scaling at all. In that case the mass-weighted norm of each normal mode is one.

In order to obtain physically reasonable normal modes, the configuration of the universe must correspond to a local minimum of the potential energy.

A NormalModes object behaves like a sequence of modes. Individual modes (see class MMTK.NormalModes.Mode (page 114)) can be extracted by indexing with an integer. Looping over the modes is possible as well.

Methods:

- reduceToRange(first, last)
  Discards all modes except for those whose numbers are between first (inclusive) and last (exclusive). This is done to reduce memory requirements, especially before saving the modes to a file.

- fluctuations()
  Returns a MMTK.ParticleScalar (page 63) containing the thermal fluctuations for each atom in the universe.

Class SparseMatrixNormalModes: Normal modes using a sparse matrix

A subclass of MMTK.NormalModes.NormalModes (page 114).
This class differs from the class NormalModes in that it obtains the Cartesian force constant matrix in a sparse-matrix format and uses a sparse-matrix eigenvalue solver from the ARPACK library. This is advantageous if the Cartesian force constant matrix is sparse (as it is for force fields without long-range terms), but for non-sparse matrices the memory requirements are higher than for NormalModes. Note that the calculation time depends not only on the size of the system, but also on its frequency spectrum, because an iterative algorithm is used.
Constructor: SparseMatrixNormalModes(universe, nmodes, temperature=300)

universe the system for which the normal modes are calculated; it must have a force field which provides the second derivatives of the potential energy

nmodes the number of modes that is calculated. The calculation time can grow significantly with an increasing number of modes.

temperature the temperature for which the amplitudes of the atomic displacement vectors are calculated. A value of `None` can be specified to have no scaling at all. In that case the mass-weighted norm of each normal mode is one.

In order to obtain physically reasonable normal modes, the configuration of the universe must correspond to a local minimum of the potential energy.
A SparseMatrixNormalModes object behaves like a sequence of modes. Individual modes (see class MMTK.NormalModes.Mode (page 114)) can be extracted by indexing with an integer. Looping over the modes is possible as well.

Class SubspaceNormalModes: Normal modes in a subspace

A subclass of MMTK.NormalModes.NormalModes (page 114).
Constructor: SubspaceNormalModes(universe, basis, temperature=300)

universe the system for which the normal modes are calculated; it must have a force field which provides the second derivatives of the potential energy

basis the basis for the subspace in which the normal modes are calculated (or, more precisely, a set of vectors spanning the subspace; it does not have to be orthogonal). This can either be a sequence of MMTK.ParticleVector (page 64) objects or a tuple of two such sequences. In the second case, the subspace is defined by the space spanned by the first set of vectors projected on the complement of the space spanned by the second set of vectors. The second set thus defines directions that are excluded from the subspace.

temperature the temperature for which the amplitudes of the atomic displacement vectors are calculated. A value of `None` can be specified to have no scaling at all. In that case the mass-weighted norm of each normal mode is one.

In order to obtain physically reasonable normal modes, the configuration of the universe must correspond to a local minimum of the potential energy.
A SubspaceNormalModes object behaves like a sequence of modes. Individual modes (see class MMTK.NormalModes.Mode (page 114)) can be extracted by indexing with an integer. Looping over the modes is possible as well.

Class FiniteDifferenceSubspaceNormalModes: Normal modes in a subspace with numerical differentiation

A subclass of MMTK.NormalModes.SubspaceNormalModes (page 116).
This class differs from SubspaceNormalModes in the way it obtains the force constant matrix. Instead of obtaining the full Cartesian force constant matrix from the force field and projecting it on the subspace, it performs a numerical differentiation of the gradients along the basis vectors of the subspace. This is useful in two cases:

- for small subspaces this approach uses less memory, because the full Cartesian force constant matrix is not needed
- it can be used even if the force field does not provide second derivatives

Constructor: FiniteDifferenceSubspaceNormalModes(universe, basis, delta=0.0001, temperature=300)

universe the system for which the normal modes are calculated

basis the basis for the subspace in which the normal modes are calculated (or, more precisely, a set of vectors spanning the subspace; it does not have to be orthogonal). This can either be a sequence of ParticleVector objects or a tuple of two such sequences. In the second case, the subspace is defined by the space spanned by the first set of vectors projected on the complement of the space spanned by the second set of vectors. The second set thus defines directions that are excluded from the subspace.

delta the length of the displacement used for numerical differentiation

temperature the temperature for which the amplitudes of the atomic displacement vectors are calculated. A value of `None` can be specified to have no scaling at all. In that case the mass-weighted norm of each normal mode is one.

In order to obtain physically reasonable normal modes, the configuration of the universe must correspond to a local minimum of the potential energy.

A FiniteDifferenceSubspaceNormalModes object behaves like a sequence of modes. Individual modes (see class MMTK.NormalModes.Mode (page 114)) can be extracted by indexing with an integer. Looping over the modes is possible as well.

Class SparseMatrixSubspaceNormalModes: Normal modes in a subspace using a sparse matrix

A subclass of MMTK.NormalModes.SubspaceNormalModes (page 116). This class differs from SubspaceNormalModes in that it obtains the Cartesian force constant matrix in a sparse-matrix format. This is advantageous if the Cartesian force constant matrix is sparse (as it is for force fields without long-range terms), but for non-sparse matrices the memory requirements are higher than for SubspaceNormalModes. Constructor: SparseMatrixSubspaceNormalModes(universe, basis, temperature=300)

universe the system for which the normal modes are calculated; it must have a force field which provides the second derivatives of the potential energy

basis the basis for the subspace in which the normal modes are calculated (or, more precisely, a set of vectors spanning the subspace; it does not have to be orthogonal). This can either be a sequence of ParticleVector objects or a tuple of two such sequences. In the second case, the subspace is defined by the space spanned by the first set of vectors projected on the complement of the space spanned by the second set of vectors. The second set thus defines directions that are excluded from the subspace.

temperature the temperature for which the amplitudes of the atomic displacement vectors are calculated. A value of `None` can be specified to have no scaling at all. In that case the mass-weighted norm of each normal mode is one.

In order to obtain physically reasonable normal modes, the configuration of the universe must correspond to a local minimum of the potential energy. A SparseMatrixSubspaceNormalModes object behaves like a sequence of modes. Individual modes (see class MMTK.NormalModes.Mode (page

114)) can be extracted by indexing with an integer. Looping over the modes is possible as well.

Module MMTK.NucleicAcids

Class Nucleotide: Nucleic acid residue

A subclass of MMTK.ChemicalObjects.Group (page 76).
Nucleotides are a special kind of group. Like any other group, they are defined in the chemical database. Each residue has two or three subgroups (`sugar` and `base`, plus `phosphate`except for 5'-terminal residues) and is usually connected to other residues to form a nucleotide chain. The database contains three variants of each residue (5'-terminal, 3'-terminal, non-terminal).
Constructor: Nucleotide(kind, model="all")

kind the name of the nucleotide in the chemical database. This is the full name of the residue plus the suffix "_5ter" or "_3ter" for the terminal variants.

model one of "all" (all-atom), "none" (no hydrogens), "polar" (united-atom with only polar hydrogens), "polar_charmm" (like "polar", but defining polar hydrogens like in the CHARMM force field). Currently the database has definitions only for "all".

Methods:

- backbone()
  Returns the sugar and phosphate groups.
- bases()
  Returns the base group.

Class NucleotideChain: Nucleotide chain

A subclass of MMTK.Biopolymers.ResidueChain (page 72).
Nucleotide chains consist of nucleotides that are linked together. They are a special kind of molecule, i.e. all molecule operations are available.
Constructor: NucleotideChain(sequence, **—properties—)

sequence the nucleotide sequence. This can be a list of two-letter codes (a "d" or "r" for the type of sugar, and the one-letter base code), or a PDBNucleotideChain object. If a PDBNucleotideChain object is supplied, the atomic positions it contains are assigned to the atoms of the newly generated nucleotide chain, otherwise the positions of all atoms are undefined.

properties optional keyword properties:

- model: one of "all" (all-atom), "no_hydrogens" or "none" (no hydrogens), "polar_hydrogens" or "polar" (united-atom with only polar hydrogens), "polar_charmm" (like "polar", but defining polar hydrogens like in the CHARMM force field). Default is "all". Currently the database contains definitions only for "all".
- terminus_5: 1 if the first nucleotide should be constructed using the 5'-terminal variant, 0 if the non-terminal version should be used. Default is 1.
- terminus_3: 1 if the last residue should be constructed using the 3'-terminal variant, 0 if the non-terminal version should be used. Default is 1.
- circular: 1 if a bond should be constructed between the first and the last residue. Default is 0.
- name: a name for the chain (a string)

Nucleotide chains act as sequences of residues. If `n` is a NucleotideChain object, then

- `len(n)` yields the number of nucleotides
- `n[i]` yields nucleotide number `i` (counting from zero)
- `n[i:j]` yields the subchain from nucleotide number `i` up to but excluding nucleotide number `j`

Methods:

- backbone()
  Returns a collection containing the sugar and phosphate groups of all nucleotides.

- bases()
  Returns a collection containing the base groups of all nucleotides.

Class NucleotideSubChain: A contiguous part of a nucleotide chain

NucleotideSubChain objects are the result of slicing operations on NucleotideChain objects. They cannot be created directly. NucleotideSubChain objects permit all operations of NucleotideChain objects, but cannot be added to a universe.

Functions

- isNucleotideChain()
  Returns 1 if x is a NucleotideChain.

Module MMTK.PDB

This module provides classes that represent molecules in PDB file. They permit various manipulations and the creation of MMTK objects. Note that the classes defined in this module are specializations of classed defined in Scientific.IO.PDB; the methods defined in that module are also available.

Class PDBPeptideChain: Peptide chain in a PDB file

A subclass of Scientific.IO.PDB.PeptideChain. See the description of that class for the constructor and additional methods. In MMTK, PDBPeptideChain objects are usually obtained from a PDBConfiguration object via its attribute peptide_chains (see the documentation of Scientific.IO.PDB.Structure).

Methods:

- createPeptideChain(model=`'all'`, n_terminus=`None`, c_terminus=`None`)
  Returns a PeptideChain object corresponding to the peptide chain in the PDB file. The parameter modelhas the same meaning as for the PeptideChain constructor.

Class PDBNucleotideChain: Nucleotide chain in a PDB file

A subclass of Scientific.IO.PDB.NucleotideChain. See the description of that class for the constructor and additional methods. In MMTK, PDBNucleotideChain objects are usually obtained from a PDBConfiguration object via its attribute nucleotide_chains (see the documentation of Scientific.IO.PDB.Structure).

Methods:

- createNucleotideChain(model=`'all'`)
  Returns a NucleotideChain object corresponding to the nucleotide chain in the PDB file. The parameter modelhas the same meaning as for the NucleotideChain constructor.

Class PDBMolecule: Molecule in a PDB file

A subclass of Scientific.IO.PDB.Molecule. See the description of that class for the constructor and additional methods. In MMTK, PDBMolecule objects are usually obtained from a PDBConfiguration object via its attribute molecules (see the documentation of Scientific.IO.PDB.Structure). A molecule is by definition any residue in a PDB file that is not an amino acid or nucleotide residue.

Methods:

- createMolecule(name=`None`)
  Returns a Molecule object corresponding to the molecule in the PDB file. The parameter name specifies the molecule name as defined in the chemical database. It can be left out for known molecules (currently only water).

Class PDBConfiguration: Everything in a PDB file

A PDBConfiguration object represents the full contents of a PDB file. It can be used to create MMTK objects for all or part of the molecules, or to change the configuration of an existing system.
Constructor: PDBConfiguration(filename)

filename the name of a PDB file

Methods:

- createPeptideChains(model=`'all'`)
  Returns a list of PeptideChain objects, one for each peptide chain in the PDB file. The parameter modelhas the same meaning as for the PeptideChain constructor.

- createNucleotideChains(model=`'all'`)
  Returns a list of NucleotideChain objects, one for each nucleotide chain in the PDB file. The parameter modelhas the same meaning as for the NucleotideChain constructor.

- createMolecules(names=`None`, permit_undefined=`1`)

Returns a collection of Molecule objects, one for each molecule in the PDB file. Each PDB residue not describing an amino acid or nucleotide residue is considered a molecule.

The parameter names allows the selective construction of certain molecule types and the construction of unknown molecules. If its value is a list of molecule names (as defined in the chemical database) and/or PDB residue names, only molecules mentioned in this list will be constructed. If its value is a dictionary, it is used to map PDB residue names to molecule names. By default only water molecules are recognizes (under various common PDB residue names); for all other molecules a molecule name must be provided by the user.

The parameter permit_undefined determines how PDB residues without a corresponding molecule name are handled. A value of 0 causes an exception to be raised. A value of 1 causes an AtomCluster object to be created for each unknown residue.

- createAll(molecule_names=`None`, permit_undefined=`1`)
  Returns a collection containing all objects contained in the PDB file, i.e. the combination of the objects returned by createPeptideChains(), createNucleotideChains(), and createMolecules(). The parameters have the same meaning as for createMolecules().

- applyTo(object)
  Sets the configuration of object from the coordinates in the PDB file. The object must be compatible with the PDB file, i.e. contain the same subobjects and in the same order. This is usually only guaranteed if the object was created by the method createAll() from a PDB file with the same layout.

Class PDBOutputFile: PDB file for output

Constructor: PDBOutputFile(filename, subformat=`None`)

filename the name of the PDB file that is created

subformat a variant of the PDB format; these subformats are defined in module Scientific.IO.PDB. The default is the standard PDB format.

Methods:

- write(object, configuration=`None`, tag=`None`)
  Writes object to the file, using coordinates from configuration (defaults to the current configuration). The parameter tag is reserved for internal use.

- close()
  Closes the file. Must be called in order to prevent data loss.

Module MMTK.ParticleProperties

Class ParticleProperty: Property defined for each particle

This is an abstract base class; for creating instances, use one of its subclasses: MMTK.ParticleScalar (page 63), MMTK.ParticleVector (page 64), MMTK.ParticleTensor (page 65).
ParticleProperty objects store properties that are defined per particle, such as mass, position, velocity, etc. The value corresponding to a particular atom can be retrieved or changed by indexing with the atom object.

Methods:

- zero()
  Returns an object of the element type (scalar, vector, etc.) with the value 0.
- sumOverParticles()
  Returns the sum of the values for all particles.
- assign(other)
  Copy all values from other, which must be a compatible ParticleProperty object.
- scaleBy(factor)
  Multiply all values by factor (a number).

Functions

- isParticleProperty()
  Returns 1 if object is a ParticleProperty.
- isConfiguration()
  Returns 1 if object is a Configuration.

Module MMTK.Proteins

Class Residue: Amino acid residue

A subclass of MMTK.ChemicalObjects.Group (page 76).
Amino acid residues are a special kind of group. Like any other group, they are defined in the chemical database. Each residue has two subgroups (`peptide` and `sidechain`) and is usually connected to other residues to form a peptide chain. The database contains three variants of each residue (N-terminal, C-terminal, non-terminal) and various models (all-atom, united-atom, C_alpha).
Constructor: Residue(kind, model="all")

kind the name of the residue in the chemical database. This is the full name of the residue plus the suffix "_nt" or "_ct" for the terminal variants.

model one of "all" (all-atom), "none" (no hydrogens), "polar" (united-atom with only polar hydrogens), "polar_charmm" (like "polar", but defining polar hydrogens like in the CHARMM force field), "polar_opls" (like "polar", but defining polar hydrogens like in the latest OPLS force field), "calpha" (only the C_alpha atom)

Methods:

- backbone()
  Returns the peptide group.
- sidechains()
  Returns the sidechain group.
- phiPsi(conf=`None`)
  Returns the values of the backbone dihedral angles phi and psi.

Class PeptideChain: Peptide chain

A subclass of MMTK.Biopolymers.ResidueChain (page 72).

Peptide chains consist of amino acid residues that are linked by peptide bonds. They are a special kind of molecule, i.e. all molecule operations are available.
Constructor: PeptideChain(sequence, **—properties—)

sequence the amino acid sequence. This can be a string containing the one-letter codes, or a list of three-letter codes, or a PDBPeptideChain object. If a PDBPeptideChain object is supplied, the atomic positions it contains are assigned to the atoms of the newly generated peptide chain, otherwise the positions of all atoms are undefined.

properties optional keyword properties:

- model: one of ”all” (all-atom), ”no_hydrogens” or ”none” (no hydrogens), ”polar_hydrogens” or ”polar” (united-atom with only polar hydrogens), ”polar_charmm” (like ”polar”, but defining polar hydrogens like in the CHARMM force field), ”polar_opls” (like ”polar”, but defining polar hydrogens like in the latest OPLS force field), ”calpha” (only the C_alpha atom of each residue). Default is ”all”.

- n_terminus: 1 if the first residue should be constructed using the N-terminal variant, 0 if the non-terminal version should be used. Default is 1.

- c_terminus: 1 if the last residue should be constructed using the C-terminal variant, 0 if the non-terminal version should be used. Default is 1.

- circular: 1 if a peptide bond should be constructed between the first and the last residue. Default is 0.

- name: a name for the chain (a string)

Peptide chains act as sequences of residues. If `p` is a PeptideChain object, then

- `len(p)` yields the number of residues

- `p[i]` yields residue number `i` (counting from zero)

- `p[i:j]` yields the subchain from residue number `i` up to but excluding residue number `j`

Methods:

- sequence()
  Returns the primary sequence as a list of three-letter residue codes.
- backbone()
  Returns a collection containing the peptide groups of all residues.
- sidechains()
  Returns a collection containing the sidechain groups of all residues.
- phiPsi(conf=`None`)
  Returns a list of the (phi, psi) backbone angle pairs for each residue.
- replaceResidue(r_old, r_new)
  Replaces residue r_old, which must be a residue object that is part of the chain, by the residue object r_new.

Class SubChain: A contiguous part of a peptide chain

SubChain objects are the result of slicing operations on PeptideChain objects. They cannot be created directly. SubChain objects permit all operations of PeptideChain objects, but cannot be added to a universe.

Class Protein: Protein

A subclass of MMTK.Complex (page 69).
A Protein object is a special kind of a Complex object which is made up of peptide chains.
Constructor: Protein(specification, **—properties—)

specification one of:

- a list of peptide chain objects

- a string, which is interpreted as the name of a database definition for a protein. If that definition does not exist, the string is taken to be the name of a PDB file, from which all peptide chains are constructed and assembled into a protein.

properties optional keyword properties:

- model: one of "all" (all-atom), "no_hydrogens" or "none" (no hydrogens), "polar_hydrogens" or "polar" (united-atom with only polar hydrogens), "polar_charmm" (like "polar", but defining polar hydrogens like in the CHARMM force field), "polar_opls" (like "polar", but defining polar hydrogens like in the latest OPLS force field), "calpha" (only the C_alpha atom of each residue). Default is "all".
- position: the center-of-mass position of the protein (a vector)
- name: a name for the protein (a string)

If the atoms in the peptide chains that make up a protein have defined positions, sulfur bridges within chains and between chains will be constructed automatically during protein generation based on a distance criterion between cystein sidechains.
Proteins act as sequences of chains. If `p` is a Protein object, then

- `len(p)` yields the number of chains
- `p[i]` yields chain number `i` (counting from zero)

Methods:

- residuesOfType(*types)
  Returns a collection that contains all residues whose type (one- or three-letter code) is contained in types.
- backbone()
  Returns a collection containing the peptide groups of all residues in all chains.

- sidechains()
  Returns a collection containing the sidechain groups of all residues in all chains.
- residues()
  Returns a collection containing all residues in all chains.
- phiPsi(conf=`None`)
  Returns a list containing the phi/psi backbone dihedrals for all chains.

Functions

- isPeptideChain()
  Returns 1 f x is a peptide chain.
- isProtein()
  Returns 1 f x is a protein.

Module MMTK.Random

This module defines various random quantities that are useful in molecular simulations. For obtaining random numbers, it tries to use the RNG module, which is part of the LLNL package distribution, which also contains Numerical Python. If RNG is not available, it uses the random number generators in modules RandomArray (part of Numerical Python) and whrandom (in the Python standard library).

Functions

- randomPointInBox()
  Returns a vector drawn from a uniform distribution within a rectangular box with edge lengths a, b, c. If b and/or care omitted, they are taken to be equal to a.

- randomPointInSphere()
  Returns a vector drawn from a uniform distribution within a sphere of radius r.

- randomDirection()
  Returns a vector drawn from a uniform distribution on the surface of a unit sphere.

- randomDirections()
  Returns a list of n vectors drawn from a uniform distribution on the surface of a unit sphere. If n is negative, return a deterministic list of not more than -—n— vectors of unit length (useful for testing purposes).

- randomRotation()
  Returns a Rotation object describing a random rotation with a uniform axis distribution and angles drawn from a uniform distribution between -—max_angle— and max_angle.

- randomVelocity()
  Returns a random velocity vector for a particle of a given mass, drawn from a Boltzmann distribution for the given temperature.

- randomParticleVector()
  Returns a ParticleVector object in which each vector is drawn from a Gaussian distribution with a given width centered around zero.

Module MMTK.Solvation

See also the example MolecularDynamics/solvation.py.

Functions

- numberOfSolventMolecules()
  Returns the number of solvent molecules of type solventthat must be added to universe, in addition to whatever it contains already, to obtain the specified solvent density.

- addSolvent()
  Scales up the universe by scale_factor and adds as many molecules of type solvent (a molecul object or a string) as are necessary to obtain the specified solvent density, taking account of the solute molecules that are already present in the universe. The molecules are placed at random positions in the scaled-up universe, but without overlaps between any two molecules.

- shrinkUniverse()
  Shrinks universe, which must have been scaled up by Function:MMTK.Solvation.addSolvent, back to its original size. The compression is performed in small steps, in between which some energy minimization and molecular dynamics steps are executed. The molecular dynamics is run at the given temperature, and an optional trajectory (a MMTK.Trajectory.Trajectory object or a string, interpreted as a file name) can be specified in which intermediate configurations are stored.

Module MMTK.Subspace

This module implements subspaces for infinitesimal (or finite small-amplitude) atomic motions. They can be used in normal mode calculations (see example NormalModes/constrained_modes.py) or for analyzing complex motions [Hinsen1999a].

Class Subspace: Subspace of infinitesimal atomic motions

Constructor: Subspace(universe, vectors)

universe the universe for which the subspace is created

vectors a list of MMTK.ParticleVector (page 64) objects that define the subspace. They need not be orthogonal or linearly independent.

Methods:

- getBasis()
  Returns a basis for the subspace, which is obtained by orthonormalization of the input vectors using Singular Value Decomposition. The basis consists of a sequence of MMTK.ParticleVector (page 64) objects that are orthonormal in configuration space.

- projectionOf(vector)
  Returns the projection of vector (a MMTK.ParticleVector (page 64)object) onto the subspace.

- projectionComplementOf(vector)
  Returns the projection of vector (a MMTK.ParticleVector (page 64)object) onto the orthogonal complement of the subspace.

Class RigidMotionSubspace: Subspace of rigid-body motions

A subclass of MMTK.Subspace.Subspace (page 136).

A rigid-body motion subspace is the subspace which contains the rigid-body motions of any number of chemical objects.
Constructor: RigidMotionSubspace(universe, objects)

universe the universe for which the subspace is created

objects a sequence of objects whose rigid-body motion is included in the subspace

Class PairDistanceSubspace: Subspace of pair-distance motions

A subclass of MMTK.Subspace.Subspace (page 136).
A pair-distance motion subspace is the subspace which contains the relative motions of any number of atom pairs along their distance vector, e.g. bond elongation between two bonded atoms.
Constructor: PairDistanceSubspace(universe, atom_pairs)

universe the universe for which the subspace is created

atom_pairs a sequence of atom pairs whose distance-vector motion is included in the subspace

Module MMTK.Trajectory

Class Trajectory: Trajectory file

Constructor: Trajectory(object, filename, mode="r", comment=None, double_precision=0, cycle=0, block_size=1)

object the object whose data is stored in the trajectory file. This can be `None` when opening a file for reading; in that case, a universe object is constructed from the description stored in the trajectory file. This universe object can be accessed via the attribute `universe` of the trajectory object.

filename the name of the trajectory file

mode one of "r" (read-only), "w" (create new file for writing), or "a" (append to existing file or create if the file does not exist)

comment optional comment that is stored in the file; allowed only with mode="r"

double_precision if non-zero, data in the file is stored using double precision; default is single precision. Note that all I/O via trajectory objects is double precision; conversion from and to single precision file variables is handled automatically.

cycle if non-zero, a trajectory is created for a fixed number of steps equal to the value of cycle, and these steps are used cyclically. This is meant for restart trajectories.

block_size an optimization parameter that influences the file structure and the I/O performance for very large files. A block size of 1 is optimal for sequential access to configurations etc., whereas a block size equal to the number of steps is optimal for reading coordinates or scalar variables along the time axis. The default value is 1. Note that older MMTK releases always used a block size of 1 and cannot handle trajectories with different block sizes.

The data in a trajectory file can be accessed by step or by variable. If `t` is a Trajectory object, then:

- `len(t)` is the number of steps
- `t[i]` is the data for step i, in the form of a dictionary that maps variable names to data
- `t[i:j]` and `t[i:j:n]` return a SubTrajectory object that refers to a subset of the total number of steps (no data is copied)
- `t.variable` returns the value of the named variable at all time steps. If the variable is a simple scalar, it is read completely and returned as an array. If the variable contains data for each atom, a TrajectoryVariable object is returned from which data at specific steps can be obtained by further indexing operations.

The routines that generate trajectories decide what variables are used and what they contain. The most frequently used variable is "configuration", which stores the positions of all atoms. Other common variables are "time", "velocities", "temperature", "pressure", and various energy terms whose name end with "_energy".

Methods:

- close()
  Close the trajectory file. Must be called after writing to ensure that all buffered data is written to the file. No data access is possible after closing a file.

- readParticleTrajectory(atom, first=`0`, last=`None`, skip=`1`, variable=`'configuration'`)
  Read the values of the specified variable for the specified atom at all time steps from first to last with an increment of skip. The result is a ParticleTrajectory object. If the variable is "configuration", the resulting trajectory is made continuous by eliminating all jumps caused by periodic boundary conditions. The pseudo-variable "box_coordinates" can be read to obtain the values of the variable "configuration" scaled to box coordinates. For non-periodic universes there is no difference between box coordinates and real coordinates.

- readRigidBodyTrajectory(object, first=`0`, last=`None`, skip=`1`, reference=`None`)
  Read the positions for the specified object at all time steps from first to last with an increment of skip and extract the rigid-body motion (center-of-mass position plus orientation as a quaternion) by an optimal-transformation fit. The result is a RigidBodyTrajectory object.

- variables()
  Returns a list of the names of all variables that are stored in the trajectory.

- view(first=`0`, last=`None`, step=`1`, object=`None`)
  Show an animation of object using the positions in the trajectory at all time steps from first to last with an increment of skip. object defaults to the entire universe.

Class SubTrajectory: Reference to a subset of a trajectory

A SubTrajectory object is created by slicing a Trajectory object or another SubTrajectory object. It provides all the operations defined on Trajectory objects.

Class TrajectoryVariable: Variable in a trajectory

A TrajectoryVariable object is created by extracting a variable from a Trajectory object if that variable contains data for each atom and is thus potentially large. No data is read from the trajectory file when a TrajectoryVariable object is created; the read operation takes place when the TrajectoryVariable is indexed with a specific step number.
If `t` is a TrajectoryVariable object, then:

- `len(t)` is the number of steps

- `t[i]` is the data for step i, in the form of a ParticleScalar, a ParticleVector, or a Configuration object, depending on the variable

- `t[i:j]` and `t[i:j:n]` return a SubVariable object that refers to a subset of the total number of steps

Class SubVariable: Reference to a subset of a TrajectoryVariable

A subclass of MMTK.Trajectory.TrajectoryVariable (page 140).
A SubVariable object is created by slicing a TrajectoryVariable object or another SubVariable object. It provides all the operations defined on TrajectoryVariable objects.

Class TrajectorySet: Trajectory file set

A trajectory set permits to treat a sequence of trajectory files like a single trajectory for reading data. It behaves like an object of the class MMTK.Trajectory.Trajectory (page 138). The trajectory files must all contain data for the same system. The variables stored in the individual files need not be the same, but only variables common to all files can be accessed.
Constructor: TrajectorySet(object, filename_list)

object the object whose data is stored in the trajectory files. This can be (and usually is) `None`; in that case, a universe object is constructed from the description stored in the first trajectory file. This universe object can be accessed via the attribute `universe` of the trajectory set object.

filename_list a list of trajectory file names or (filename, first_step, last_step, increment) tuples.

Note: depending on how the sequence of trajectories was constructed, the first configuration of each trajectory might be the same as the last one in the preceding trajectory. To avoid counting it twice, specify (filename, 1, None, 1) for all but the first trajectory in the set.

Class TrajectorySetVariable: Variable in a trajectory set

A TrajectorySetVariable object is created by extracting a variable from a TrajectorySet object if that variable contains data for each atom and is thus potentially large. It behaves exactly like a TrajectoryVariable object.

Class ParticleTrajectory: Trajectory data for a single particle

A ParticleTrajectory object is created by calling the method `readParticleTrajectory` on a Trajectory object. If `pt` is a ParticleTrajectory object, then

- `len(pt)` is the number of steps stored in it
- `pt[i]` is the value at step `i` (a vector)

Methods:

- translateBy(vector)
  Adds vector to the values at all steps. This does *not*change the data in the trajectory file.

Class RigidBodyTrajectory: Rigid-body trajectory data

A RigidBodyTrajectory object is created by calling the method `readRigidBodyTrajectory` on a Trajectory object. If `rbt` is a RigidBodyTrajectory object, then

- `len(rbt)` is the number of steps stored in it
- `rbt[i]` is the value at step `i` (a vector for the center of mass and a quaternion for the orientation)

Class TrajectoryOutput: Trajectory output action

A TrajectoryOutput object is used in the action list of any trajectory-generating operation. It writes any of the available data to a trajectory file. It is possible to use several TrajectoryOutput objects at the same time in order to produce multiple trajectories from a single run.
Constructor: TrajectoryOutput(trajectory, data=None, first=0, last=None, skip=1)

trajectory a trajectory object or a string, which is interpreted as the name of a file that is opened as a trajectory in append mode data a list of data categories. All variables provided by the trajectory generator that fall in any of the listed categories are written to the trajectory file. See the descriptions of the trajectory generators for a list of variables and categories. By default (data = `None`) the categories ”configuration”, ”energy”, ”thermodynamic”, and ”time” are written.

first the number of the first step at which the action is executed

last the number of the last step at which the action is executed. A value of `None` indicates that the action should be executed indefinitely.

skip the number of steps to skip between two applications of the action

Class RestartTrajectoryOutput: Restart trajectory output action

A RestartTrajectoryOutput object is used in the action list of any trajectory-generating operation. It writes those variables to a trajectory that the trajectory generator declares as necessary for restarting.
Constructor: RestartTrajectoryOutput(trajectory, skip=100, length=3)

trajectory a trajectory object or a string, which is interpreted as the name of a file that is opened as a trajectory in append mode with a cycle length of length and double-precision variables

skip the number of steps between two write operations to the restart trajectory

length the number of steps stored in the restart trajectory; used only if trajectory is a string

Class LogOutput: Protocol file output action

A LogOutput object is used in the action list of any trajectory-generating operation. It writes any of the available data to a text file.
Constructor: LogOutput(file, data, first=0, last=None, skip=1)

file a file object or a string, which is interpreted as the name of a file that is opened in write mode data a list of data categories. All variables provided by the trajectory generator that fall in any of the listed categories are written to the trajectory file. See the descriptions of the trajectory generators for a list of variables and categories. By default (`data` = `None`) the categories "energy" and "time" are written.

first the number of the first step at which the action is executed

last the number of the last step at which the action is executed. A value of `None` indicates that the action should be executed indefinitely.

skip the number of steps to skip between two applications of the action

Class StandardLogOutput: Standard protocol output action

A StandardLogOutput object is used in the action list of any trajectory-generating operation. It is a specialization of LogOutput to the most common case and writes data in the categories "time" and "energy" to the standard output stream.
Constructor: StandardLogOutput(skip=50)

skip the number of steps to skip between two applications of the action

Class SnapshotGenerator: Trajectory generator for single steps

A SnapshotGenerator is used for manual assembly of trajectory files. At each call it writes one step to the trajectory, using the current state of the universe (configuration, velocities, etc.) and data provided explicitly with the call.
Constructor: SnapshotGenerator(universe, **options)

universe the universe on which the integrator acts

options keyword options:

- data: a dictionary that supplies values for variables that are not part of the universe state (e.g. potential energy)

  - actions: a list of actions to be executed periodically (default is none)

Each call to the SnapshotGenerator object produces one step. All the keyword options listed above can be specified either when creating the generator or when calling it.

Functions

- isTrajectory()
  Returns 1 if object is a trajectory.

- trajectoryInfo()
  Return a string with summarial information about the trajectory file identified by filename.

Module MMTK.Units

This module defines constants and prefactors that convert between MMTK's internal unit system and other units. There are also some common physical constants.
SI Prefixes: ato, femto, pico, nano, micro, milli, centi, deci, deca, hecto, kilo, mega, giga, tera, peta
Length units: m, cm, mm, nm, pm, fm, Ang, Bohr
Angle units: rad, deg
Volume units: l
Time units: s, ns, ps, fs
Frequency units: Hz, invcm (wavenumbers)
Mass units: amu, g, kg
Quantity-of-matter units: mol
Energy units: J, kJ, cal, kcal, Hartree
Temperature units: K
Pressure units: Pa, bar, atm
Electrostatic units: C, A, V, D, eV, e
Physical constants: c (speed of light), Nav (Avogadro number), h = (Planck constant), hbar = (Planck constant divided by 2*Pi), k_B = (Boltzmann constant), eps0 = (permittivity of vacuum), me = (electron mass)

Module MMTK.Universse

Class Universe: Complete model of chemical system

A subclass of MMTK.Collection.GroupOfAtoms (page 78)and
MMTK.Visualization.Viewable (page 156).
A universe represents a complete model of a chemical system, i.e. the molecules, their environment (topology, boundary conditions, thermostats, etc.), and optionally a force field.
The class Universe is an Glossary:abstract-base-class that defines properties common to all kinds of universes. To create universe objects, use one of its subclasses.
In addition to the methods listed below, universe objects support the following operations (`u` is any universe object, `o` is any chemical object):

- `len(u)` yields the number of chemical objects in the universe
- `u[i]` returns object number `i`
- `u.name = o` adds `o` to the universe and also makes it accessible as an attribute
- `del u.name` removes the object that was assigned to `u.name` from the universe

Methods:

- objectList(klass=`None`)
  Returns a list of all chemical objects in the universe. If klass is not None, only objects whose class is equal to klass are returned.
- environmentObjectList(klass=`None`)
  Returns a list of all environment objects in the universe. If klass is not None, only objects whose class is equal to klass are returned.
- atomList()
  Returns a list of all atoms in the universe. This includes atoms that make up the compound chemical objects (molecules etc.).

- universe()
  Returns the universe itself.

- addObject(object)
  Adds object to the universe. If object is a Collection, all elements of the Collection are added to the universe. An object can only be added to a universe if it is not already part of another universe.

- removeObject(object)
  Removes object from the universe. If object is a Collection, each of its elements is removed. The object to be removed must be in the universe.

- selectShell(point, r1, r2=0.0)
  Return a Collection of all objects in the universe whose distance from point is between r1 and r2.

- selectBox(p1, p2)
  Return a Collection of all objects in the universe that lie within a box whose corners are given by p1 and p2.

- acquireReadStateLock()
  Acquire the universe read state lock. Any application that uses threading must acquire this lock prior to accessing the current state of the universe, in particular its configuration (particle positions). This guarantees the consistency of the data; while any thread holds the read state lock, no other thread can obtain the write state lock that permits modifying the state. The read state lock should be released as soon as possible.

  The read state lock can be acquired only if no thread holds the write state lock. If the read state lock cannot be acquired immediately, the thread will be blocked until it becomes available. Any number of threads can acquire the read state lock simultaneously.

- acquireWriteStateLock()
  Acquire the universe write state lock. Any application that uses threading must acquire this lock prior to modifying the current state of the universe, in particular its configuration (particle positions). This guarantees the consistency of the data; while any thread holds

the write state lock, no other thread can obtain the read state lock that permits accessing the state. The write state lock should be released as soon as possible.

The write state lock can be acquired only if no other thread holds either the read state lock or the write state lock. If the write state lock cannot be acquired immediately, the thread will be blocked until it becomes available.

- releaseReadStateLock(write=0)
  Release the universe read state lock.

- releaseWriteStateLock(write=0)
  Release the universe write state lock.

- acquireConfigurationChangeLock(waitflag=1)
  Acquire the configuration change lock. This lock should be acquired before starting an algorithm that changes the configuration continuously, e.g. minimization or molecular dynamics algorithms. This guarantees the proper order of execution when several such operations are started in succession. For example, when a minimization should be followed by a dynamics run, the use of this flag permits both operations to be started as background tasks which will be executed one after the other, permitting other threads to run in parallel.

  The configuration change lock should not be confused with the universe state lock. The former guarantees the proper sequence of long-running algorithms, whereas the latter guarantees the consistency of the data. A dynamics algorithm, for example, keeps the configuration change lock from the beginning to the end, but acquires the universe state lock only immediately before modifying configuration and velocities, and releases it immediately afterwards.

  If waitflag is true, the method waits until the lock becomes available; this is the most common operation. If waitflag is false, the method returns immediately even if another thread holds the lock. The return value indicates if the lock could be acquired (1) or not (0).

- releaseConfigurationChangeLock()
  Releases the configuration change lock.

- setForceField(forcefield)
  Assign a new forcefield to the universe.

- configuration()
  Return the configuration object describing the current configuration of the universe. Note that this is not a copy of the current state; the positions in the configuration object will change when coordinate changes are applied to the universe in whatever way.

- copyConfiguration()
  Returns a copy of the current configuration.

  This operation is thread-safe; it won't return inconsistent data even when another thread is modifying the configuration.

- setConfiguration(configuration, block=1)
  Copy all positions are from configuration (which must be a Configuration object) to the current universe configuration.

  This operation is thread-safe; it blocks other threads that want to access the configuration while the data is being updated. If this is not desired (e.g. when calling from a routine that handles locking itself), the optional parameter block should be set to 0.

- addToConfiguration(displacement, block=1)
  Add displacement (a ParticleVector object) to the current configuration of the universe.

  This operation is thread-safe; it blocks other threads that want to access the configuration while the data is being updated. If this is not desired (e.g. when calling from a routine that handles locking itself), the optional parameter block should be set to 0.

- getParticleScalar(name, datatype=`'d'`)
  Return a ParticleScalar object containing the values of the attribute name for each atom in the universe.

- getParticleBoolean(name)
  Return a ParticleScalar object containing the boolean values (0 or 1) of the attribute name for each atom in the universe. An atom that does not have the attribute name is assigned a value of zero.

- masses()
  Return a ParticleScalar object containing the atom masses.

- charges()
  Return a ParticleScalar object containing the atom charges. Since charges are parameters defined by a force field, this method will raise an exception if no force field is defined or if the current force field defines no charges.

- velocities()
  Returns ParticleVector object containing the current velocities of all atoms. If no velocities are defined, the return value is `None`. Note that the return value is not a copy of the current state but a reference to it; its data will change when any changes are made to the current velocities.

- setVelocities(velocities, block=1)
  Set the current atom velocities to the values contained in the ParticleVector object velocities. If velocities is None, the velocity information is removed from the universe.

  This operation is thread-safe; it blocks other threads that want to access the velocities while the data is being updated. If this is not desired (e.g. when calling from a routine that handles locking itself), the optional parameter block should be set to 0.

- initializeVelocitiesToTemperature(temperature)
  Generate random velocities for all atoms from a Boltzmann distribution at the given temperature.

- scaleVelocitiesToTemperature(temperature, block=1)
  Scale all velocities by a common factor in order to obtain the specified temperature.

  This operation is thread-safe; it blocks other threads that want to access the velocities while the data is being updated. If this is not desired (e.g. when calling from a routine that handles locking itself), the optional parameter block should be set to 0.

- distanceConstraintList()
  Returns the list of distance constraints.

- numberOfDistanceConstraints()
  Returns the number of distance constraints.

- setBondConstraints()
  Sets distance constraints for all bonds.

- removeDistanceConstraints()
  Removes all distance constraints.

- enforceConstraints(configuration=None, velocities=None)
  Enforces the previously defined distance constraints by modifying the configuration and velocities.

- adjustVelocitiesToConstraints(velocities=None, block=1)
  Modifies the velocities to be compatible with the distance constraints, i.e. projects out the velocity components along the constrained distances.

  This operation is thread-safe; it blocks other threads that want to access the velocities while the data is being updated. If this is not desired (e.g. when calling from a routine that handles locking itself), the optional parameter block should be set to 0.

- forcefield()
  Returns the force field.

- energy(subset1=None, subset2=None, small_change=0)
  Returns the energy. Without any parameters, the energy is calculated for the whole universe. If subset1 is given, only the energy terms within the atoms in subset1 are calculated. If subset1 and subset2 are given, only the energy terms between atoms of the two subsets are evaluated. The parameter small_change can be set to one in order to obtain a faster energy evaluation when the current configuration differs from the one during the last energy evaluation only by small displacements.

- energyAndGradients(subset1=None, subset2=None, small_change=0)
  Returns the energy and the energy gradients (a ParticleVector).

- energyAndForceConstants(subset1=None, subset2=None, small_change=0)

Returns the energy and the force constants (a SymmetricParticleTensor).

- energyGradientsAndForceConstants(subset1=`None`, subset2=`None`, small_change=`0`)
  Returns the energy, the energy gradients (a ParticleVector), and the force constants (a SymmetricParticleTensor).

- energyTerms(subset1=`None`, subset2=`None`, small_change=`0`)
  Returns a dictionary containing the energy values for each energy term separately. The energy terms are defined by the force field.

- distanceVector(p1, p2, conf=`None`)
  Returns the distance vector between p1 and p2 (i.e. the vector from p1 to p2) in the configuration conf. p1 and p2 can be vectors or subsets of the universe, in which case their center-of-mass positions are used. If conf is `None`, the current configuration of the universe is used. The result takes the universe topology (periodic boundary conditions etc.) into account.

- distance(p1, p2, conf=`None`)
  Returns the distance between p1 and p2, i.e. the length of the distance vector.

- angle(p1, p2, p3, conf=`None`)
  Returns the angle between the distance vectors —p1—-—p2— and —p3—-—p2—.

- dihedral(p1, p2, p3, p4, conf=`None`)
  Returns the dihedral angle between the plane containing the distance vectors —p1—-—p2— and —p3—-—p2— and the plane containing the distance vectors —p2—-—p3— and —p4—-—p3—.

- basisVectors()
  Returns the basis vectors of the elementary cell of a periodic universe. For a non-periodic universe the return value is `None`.

- reciprocalBasisVectors()
  Returns the reciprocal basis vectors of the elementary cell of a periodic universe. For a non-periodic universe the return value is `None`.

- cellVolume()
  Returns the volume of the elementary cell of a periodic universe. For a non-periodic universe the return value is `None`.

- largestDistance()
  Returns the largest possible distance that any two points can have in the universe. Returns `None` if no such upper limit exists.

- contiguousObjectOffset(objects=`None`, conf=`None`, box_coordinates=`0`)
  Returns a ParticleVector with displacements relative to the configuration conf which when added to the configuration create a configuration in which none of the objects is split across the edge of the elementary cell. For nonperiodic universes the return value is `None`. If no object list is specified, the list of elements of the universe is used. The configuration defaults to the current configuration of the universe.

- contiguousObjectConfiguration(objects=`None`, conf=`None`)
  Returns configuration conf (default: current configuration) corrected by the contiguous object offsets for that configuration.

- realToBoxCoordinates(vector)
  Returns the box coordinate equivalent of vector. Box coordinates are defined only for periodic universes; their components have values between -0.5 and 0.5; these extreme values correspond to the walls of the simulation box. For a nonperiodic universe, vector is returned unchanged.

- boxToRealCoordinates(vector)
  Returns the real-space equivalent of the box coordinate vector.

- randomPoint()
  Returns a random point from a uniform distribution within the universe. This operation is defined only for finite-volume universes, e.g. periodic universes.

- map(function)
  Applies function to all objects in the universe and returns the list of the results. If the results are chemical objects, a Collection is returned instead of a list.

- setFromTrajectory(trajectory, step=`None`)
  Set the state of the universe to the one stored in the given step of the given trajectory. If no step number is given, the most recently written step is used for a restart trajectory, and the first step (number zero) for a normal trajectory.

  This operation is thread-safe; it blocks other threads that want to access the configuration or velocities while the data is being updated.

Functions

- isUniverse()
  Returns 1 if object is a Universe.

Module MMTK.Visualization

This module provides visualization of chemical objects and animated visualization of normal modes and sequences of configurations, including trajectories. Visualization depends on external visualization programs. On Unix systems, these programs are defined by environment variables. Under Windows NT, the system definitions for files with extension "pdb" and "wrl" are used.
A viewer for PDB files can be defined by the environment variable `PDBVIEWER`. For showing a PDB file, MMTK will execute a command consisting of the value of this variable followed by a space and the name of the PDB file.
A viewer for VRML files can be defined by the environment variable `VRMLVIEWER`. For showing a VRML file, MMTK will execute a command consisting of the value of this variable followed by a space and the name of the VRML file.
Since there is no standard for launching viewers for animation, MMTK supports only two programs: VMD and XMol. MMTK detects these programs by inspecting the value of the environment variable `PDBVIEWER`. This value must be the file name of the executable, and must give "vmd" or "xmol" after stripping off an optional directory specification.

Class Viewable: Any viewable chemical object

This class is a mix-in class that defines a general visualization method for all viewable objects, i.e. chemical objects (atoms, molecules, etc.), collections, and universes.

Methods:

- graphicsObjects(**options)
  Returns a list of graphics objects that represent the object for which the method is called. All options are specified as keyword arguments:

  configuration the configuration in which the objects are drawn (default: the current configuration)

  model a string specifying one of several graphical representations ("wireframe", "tube", "ball_and_stick"). Default is "wireframe".

ball_radius the radius of the balls representing the atoms in a ball-and-stick model, default: 0.03

stick_radius the radius of the sticks representing the bonds in a ball-and-stick or tube model, default: 0.02 for the tube model, 0.01 for the ball-and-stick model

graphics_module the module in which the elementary graphics objects are defined (default: Scientific.Visualization.VRML)

color_values a MMTK.ParticleScalar (page 63) object that defines a value for each atom which defines that atom's color via the color scale object specified by the option color_scale. If no value is given for color_values, the atoms' colors are taken from the attribute `color` of each atom object (default values for each chemical element are provided in the chemical database).

color_scale an object that returns a color object (as defined in the module Scientific.Visualization.Color) when called with a number argument. Suitable objects are defined by Scientific.Visualization.Color.ColorScale and Scientific.Visualization.Color.SymmetricColorScale. The object is used only when the option color_values is specified as well. The default is a blue-to-red color scale that covers the range of the values given in color_values.

color a color name predefined in the module Scientific.Visualization.Color. The corresponding color is applied to all graphics objects that are returned.

Functions

- view()
  Equivalent to object.view(parameters).

- viewTrajectory()
  Launches an external viewer with animation capabilities to display the configurations from first to last in increments of step in trajectory. The trajectory can be specified by a MMTK.Trajectory.Trajectory (page 138) object or by a string which is interpreted as the file name of a trajectory file. An optional parameter subset can specify an

object which is a subset of the universe contained in the trajectory, in order to restrict visualization to this subset.

- viewSequence()
  Launches an external viewer with animation capabilities to display object in the configurations given in conf_list, which can be any sequence of configurations, including the variable "configuration" from a MMTK.Trajectory.Trajectory (page 138) object. If periodicis 1, the configurations will be repeated periodically, provided that the external viewers supports this operation.

Module MMTK.Visualization_win32

This module provides visualization of chemical objects and animated visualization of normal modes and sequences of configurations, including trajectories. Visualization depends on external visualization programs. On Unix systems, these programs are defined by environment variables. Under Windows NT, the system definitions for files with extension "pdb" and "wrl" are used.
A viewer for PDB files can be defined by the environment variable `PDBVIEWER`. For showing a PDB file, MMTK will execute a command consisting of the value of this variable followed by a space and the name of the PDB file.
A viewer for VRML files can be defined by the environment variable `VRMLVIEWER`. For showing a VRML file, MMTK will execute a command consisting of the value of this variable followed by a space and the name of the VRML file.
Since there is no standard for launching viewers for animation, MMTK supports only two programs: VMD and XMol. MMTK detects these programs by inspecting the value of the environment `PDBVIEWER`. This value must be the file name of the executable, and must give "vmd" or "xmol" after stripping off an optional directory specification.

Class Viewable: Any viewable chemical object

This class is a mix-in class that defines a general visualization method for all viewable objects, i.e. chemical objects (atoms, molecules, etc.), collections, and universes.

Methods:

- graphicsObjects(**options)
  Returns a list of graphics objects that represent the object for which the method is called. All options are specified as keyword arguments:

  configuration the configuration in which the objects are drawn (default: the current configuration)

  model a string specifying one of several graphical representations ("wireframe", "tube", "ball_and_stick"). Default is "wireframe".

ball_radius the radius of the balls representing the atoms in a ball-and-stick model, default: 0.03

stick_radius the radius of the sticks representing the bonds in a ball-and-stick or tube model, default: 0.02 for the tube model, 0.01 for the ball-and-stick model

graphics_module the module in which the elementary graphics objects are defined (default: Scientific.Visualization.VRML)

color_values a MMTK.ParticleScalar (page 63) object that defines a value for each atom which defines that atom's color via the color scale object specified by the option color_scale. If no value is given for color_values, the atoms' colors are taken from the attribute `color` of each atom object (default values for each chemical element are provided in the chemical database).

color_scale an object that returns a color object (as defined in the module Scientific.Visualization.Color) when called with a number argument. Suitable objects are defined by Scientific.Visualization.Color.ColorScale and Scientific.Visualization.Color.SymmetricColorScale. The object is used only when the option color_values is specified as well. The default is a blue-to-red color scale that covers the range of the values given in color_values.

color a color name predefined in the module Scientific.Visualization.Color. The corresponding color is applied to all graphics objects that are returned.

Functions

- view()
  Equivalent to object.view(parameters).

- viewTrajectory()
  Launches an external viewer with animation capabilities to display the configurations from first to last in increments of step in trajectory. The trajectory can be specified by a MMTK.Trajectory.Trajectory (page 138) object or by a string which is interpreted as the file name of a trajectory file. An optional parameter subset can specify an

object which is a subset of the universe contained in the trajectory, in order to restrict visualization to this subset.

- viewSequence()
  Launches an external viewer with animation capabilities to display object in the configurations given in conf_list, which can be any sequence of configurations, including the variable "configuration" from a MMTK.Trajectory.Trajectory (page 138) object. If periodicis 1, the configurations will be repeated periodically, provided that the external viewers supports this operation.

Chapter 11

Examples

One of the best ways to learn how to use a new tool is to look at examples. The examples given in this manual were adapted from real-life MMTK applications. They are also contained in the MMTK distribution (directory "Examples") for direct use and modification.
The example molecules, system sizes, parameters, etc., were chosen to reduce execution time as much as possible, in order to enable you to run the examples interactively step by step to see how they work. If you plan to modify an example program for your own use, don't forget to check all parameters carefully to make sure that you obtain reasonable results.

- Molecular Dynamics examples
  - The file argon.py contains a simulation of liquid argon at constant temperature and pressure.
  - The file protein.py contains a simulation of a small (very small) protein in vacuum.
  - The file restart.py shows how the simulation started in protein.py can be continued.
  - The file solvation.py contains the solvation of a protein by water molecules.
- Monte-Carlo examples
  - The program backbone.py generates an ensemble of backbone configuration (C-alpha atoms only) for a protein.

- Trajectory examples
  - The file snapshot.py shows how a trajectory can be built up step by step from arbitrary data.
  - The file dcd_import.py converts a trajectory in DCD format (used by the programs CHARMM and X-Plor) to MMTK's format.
  - The file dcd_export.py converts an MMTK trajectory to DCD format (used by the programs CHARMM and X-Plor).
  - The file trajectory_average.py calculates an average structure from a trajectory.
  - The file trajectory_extraction.py reads a trajectory and writes a new one containing only a subset of the original universe.
  - The file view_trajectory.py shows an animation of a trajectory, provided that an external molecule viewer with animation is available.
  - The file calpha_trajectory.py shows how a much smaller C_alpha-only trajectory can be extracted from a trajectory containing one or more proteins.
- Normal mode examples
  - The file modes.py contains a standard normal mode calculation for a small protein.
  - The file constrained_modes.py contains a normal mode calculation for a small protein using a model in which each amino acid residue is rigid.
  - The file deformation_modes.py contains a normal mode calculation for a mid-size protein using a simplified model and a deformation force field.
  - The file harmonic_force_field.py contains a normal mode calculation for a protein using a detailed but still simple harmonic force field.
- Protein examples

  - The file construction.py shows some more complex examples of protein construction from PDB files.
  - The file analysis.py demonstrates a few analysis techniques for comparing protein conformations.

- DNA examples
  - The file construction.py contains the construction of a DNA strand with a ligand.

- MPI examples (parallelization)
  - The file md.py contains a parallelized version of MolecularDynamics/solvation.py.

- Langevin dynamics integrator

  The files LangevinDynamics.py and MMTK_langevinmodule.c implement a simple integrator for Langevin dynamics. It is meant as an example of how to write integrators etc. in C, but of course it can also be used directly.

- Visualization examples
  - The file additional_objects.py describes the addition of custom graphics objects to the representation of a molecular system.

- Micellaneous examples
  - The example charge_fit.py demonstrates fitting point charges to an electrostatic potential energy surface.
  - The file construct_from_pdb.py shows how a universe can be built from a PDB file in such a way that the internal atom ordering is compatible. This is important for exchanging data with other programs.
  - The file lattice.py constructs molecules placed on a lattice.
  - The file vector_field.py shows how vector fields can be used in the analysis and visualization of collective motions.

Chapter 12

Glossary

Abstract base class
A base class that is not directly usable by itself, but which defines the common properties of several subclasses. Example: the class MMTK.ChemicalObjects.ChemicalObject (page 75) is an abstract base class which defines the common properties of its subclasses MMTK.Atom (page 65), MMTK.ChemicalObjects.Group (page 76), MMTK.Molecule (page 67), MMTK.Complex (page 69), and MMTK.AtomCluster (page 70). A mix-in class is a special kind of abstract base class.
Base class
A class from which another class inherits. In most cases, the inheriting class is a specialization of the base class. For example, the class MMTK.Molecule (page 67) is a base class of MMTK.Proteins.PeptideChain (page 128), because peptide chains are special molecules. Another common application is the abstract base class.
Mix-in class
A class that is used as a base class in other classes with the sole intention of providing methods that are common to these classes. Mix-in classes cannot be used to create instances. They are a special kind of abstract base class. Example: class MMTK.Collection.GroupOfAtoms (page 78).
Subclass
A class that has another class as its base class. The subclass is usually a specialization of the base class, and can use all of the methods defined in the base class. Example: class MMTK.Proteins.Residue (page 128) is a subclass of MMTK.ChemicalObjects.Group (page 76).

Chapter 13

References

[Bondi1964] A. Bondi
van der Waals Volumes and Radii
J. Phys. Chem. **68**, 441-451 (1964)

[DPMTA] William T. Rankin
DPMTA - A Distributed Implementation of the Parallel Multipole Tree Algorithm - Version 3.0
http://www.ee.duke.edu/Research/SciComp/Docs/Dpmta/users_guide/dpmta.html

[Eisenhaber1993] F. Eisenhaber, P. Argos
Improved Strategy in Analytic Surface Calculation for Molecular Systems: Handling of Singularities and Computational Efficiency
J. Comp. Chem. **14**(11), 1272-1280 (1993)

[Eisenhaber1995] F. Eisenhaber, P. Lijnzaad, P. Argos, M. Scharf
The Double Cubic Lattice Method: Efficient Approaches to Numerical Integration of Surface Area and Volume and to Dot Surface Contouring of Molecular Assemblies
J. Comp. Chem. **16**(3), 273-284 (1995)

[Hinsen1995] Konrad Hinsen, Gerald R. Kneller
Influence of constraints on the dynamics of polypeptide chains
Phys. Rev. E **52**, 6868 (1995)

[Hinsen1997] Konrad Hinsen, Benoit Roux
An accurate potential for simulating proton transfer in acetylacetone
J. Comp. Chem. **18**, 368 (1997)

[Hinsen1998] Konrad Hinsen
Analysis of domain motions by approximate normal mode calculations
Proteins **33**(3), 417-429 (1998)

[Hinsen1999] Konrad Hinsen, Aline Thomas, Martin J. Field
Analysis of domain motions in large proteins
Proteins **34**(3), 369-382 (1999)

[Hinsen1999a] Konrad Hinsen, Gerald R. Kneller
Projection methods for the analysis of complex motions in macromolecules
Molecular Simulations **23**(3), 203-241 (1999)

[Hinsen1999b] Konrad Hinsen, Gerald R. Kneller
A simplified force field for describing vibrational protein dynamics over the whole frequency range
J. Chem. Phys. **111**(24), 10766-10769 (1999)

[Hinsen2000] Konrad Hinsen, Andrei J. Petrescu, Serge Dellerue, Marie-Claire Bellissent-Funel, Gerald R. Kneller
Harmonicity in slow protein dynamics
submitted

[Kneller1990] Gerald R. Kneller
Superposition of molecular structures using quaternions
Mol. Sim. **7**, 113-119 (1990)

[Kneller1996] Gerald R. Kneller, Thomas Mülders
Nosé-Andersen dynamics of partially rigid molecules: Coupling of all degrees of freedom to heat and pressure baths
Phys. Rev. E **54**, 6825-6837 (1996)

[Swope1982] W.C. Swope, H.C. Andersen, P.H. Berens, 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)

[vanGunsteren1982] Wilfred F. van Gunsteren, Martin Karplus
Effect of Constraints on the Dynamics of Macromolecules
Macromolecules **15**, 1528-1544 (1982)

[Viduna2000] David Viduna, Konrad Hinsen, Gerald R. Kneller
The influence of molecular flexibility on DNA radiosensitivity: A simulation study
Phys. Rev. E, in print

[Wolf1999] D. Wolf, P. Keblinski, S.R. Philpot, J. Eggebrecht
Exact method for the simulation of Coulombic systems by spherically truncated, pairwise r^{-1} summation
J. Chem. Phys. **110**(17), 8254-8282 (1999)