

EMBO Practical Course on Biomolecular Simulation: Molecular simulations with MMTK

Konrad Hinsen
Centre de Biophysique Moléculaire, CNRS Orléans
and
Synchrotron Soleil, Saint Aubin
France

3 July 2008

1 Getting started

1.1 MMTK

The Molecular Modeling Toolkit (MMTK) is a Python library for molecular simulations. It provides a Python representation of (bio)molecular systems plus a large number of common operations (Molecular Dynamics, energy minimization, normal modes etc.). In contrast to traditional simulation programs, it can be used not only as a black-box code for running simulations, but also as the basis for the development of new simulation techniques and of user-friendly application programs.

For more information about MMTK, see the Web site

<http://dirac.cnrs-orleans.fr/MMTK/>

in particular the User's Guide at

<http://dirac.cnrs-orleans.fr/MMTK/Manual/MMTK.html>

and the examples at

<http://dirac.cnrs-orleans.fr/MMTK/using-mmtk/mmtk-example-scripts/>

There is also a scientific publication documenting MMTK:

K. Hinsen

The Molecular Modeling Toolkit: A New Approach to Molecular Simulations

J. Comp. Chem. **21**, 2000: 79–85

In one of the exercises, we will use a trajectory analysis tool that is based on MMTK. It is called nMOLDYN and its Web site is at

<http://dirac.cnrs-orleans.fr/nMOLDYN/>

nMOLDYN calculates dynamical quantities such as velocity autocorrelation functions and mean-square displacements, as well as various quantities of interest in neutron scattering experiments. For more information, see

T. Rog, K. Murzyn, K. Hinsén, and G.R. Kneller

nMoldyn : A Program Package for a Neutron Scattering Oriented Analysis of Molecular Dynamics Simulations

J. Comp. Chem. **24**, 2003: 657–667

1.2 Python

This practical will use the Python language, as well as some extensions to it that are useful for scientific applications. For more information, see

- Python home page: <http://www.python.org/>
- Python Tutorial: <http://www.python.org/doc/current/tut/tut.html>
- Python Library Reference: <http://www.python.org/doc/current/lib/lib.html>
- NumPy: <http://numpy.scipy.org/>
- Scientific Python: <http://dirac.cnrs-orleans.fr/ScientificPython/>
- A more complete introduction to Python for scientists can be found at <http://dirac.cnrs-orleans.fr/plone/Members/hinsen/courses-and-lecture-notes/python/a>
- An incomplete catalogue of available Python modules can be found at <http://www.python.org/pypi>

To run any of the examples in this practical, do one of the following:

- Type the example code into a file with an editor of your choice and then type “`python -i example.py`” (without the quotes), assuming that your file is called `example.py`. To exit from Python, type Ctrl-D. If you do not use the “-i”, Python will exit immediately after executing the program.
- Type “idle” to start a Python development environment.
- Those who know Emacs can use Emacs and its Python mode as a development environment.

When typing Python code, watch out for the number of spaces at the beginning of a line. Python uses this information for determining the structure of a program, i.e. to find out which lines belong to a loop or to a conditional block. The precise number of spaces is not important, but within one block all lines must have the same number of initial spaces, i.e. the first characters must line up properly.

2 First contact with MMTK

Run the following code:

```
from MMTK import *
molecule = Molecule('water')
molecule.view()
```

The first line tells Python to make the core parts of MMTK available; all examples will begin with that line. The second line creates a water molecule which is assigned to the variable `molecule`. Since no conformation is specified, a default conformation is used. The last line starts a visualization program (VMD in our installation) for viewing the water molecule.

A slightly more complicated example:

```
from MMTK import *
universe = InfiniteUniverse()
universe.addObject(Molecule('water', position=Vector(0., 0., 0.)))
universe.addObject(Molecule('water', position=Vector(0.5, 0., 0.)))
universe.view()
```

Here the second line creates a universe object. A universe is a complete system definition for a simulation; it consists not only of the molecules, but also a description of their environment: geometry (infinite or periodic), force field (none defined in the example), a thermostat (none present here) etc.

The following two lines add two water molecules to the universe. The first molecule has its center of mass at the coordinate origin, the second one is shifted by 0.5 nm along the x-axis.

A comment about units: for a programming library like MMTK, it is essential to use a single coherent unit system, in which the standard physical relations between quantities are numerically valid without any conversion factors. For example, the length unit divided by the time unit must be the velocity unit, and the velocity unit squared times the mass unit must be the energy unit. The SI units are such a coherent unit system, but they are not practical on the atomic scale. MMTK uses “atomic SI units”, i.e. SI units with convenient scale factors: nm for length, ps for time, and g/mol (i.e. amu) for mass. This leads to an energy unit of kJ/mol.

If you prefer to specify quantities in different units, you can use a large set of pre-defined conversion constants in the submodule `Units`. For example, to write the above example in Ångström, use `position=Vector(0.5, 0., 0.)*Units.Ang`.

We still cannot calculate energies, because there is no force field yet. MMTK provides several force fields, of which the most universal ones are the Amber 94

and Amber 99 force field. Force fields are not in the core part of MMTK, so they must be imported explicitly. And we must tell the universe which force field to use. This yields the following code:

```
from MMTK import *
from MMTK.ForceFields import Amber99ForceField
universe = InfiniteUniverse(Amber99ForceField())
universe.addObject(Molecule('water', position=Vector(0., 0., 0.)))
universe.addObject(Molecule('water', position=Vector(0.5, 0., 0.)))
print universe.energy()
```

If you want to know which force field terms contribute how much, type

```
print universe.energyTerms()
```

Note: The Amber force field has various parameters that influence the way energies are calculated. By default, MMTK uses very conservative parameters that guarantee that you get physically reasonable energy values. However, energy evaluation using the default parameters can be very slow. For simple applications such as those presented here, this is not a problem. Before using MMTK for bigger simulations, please refer to the manual to learn about force field parameters.

3 A simple application: normal modes of water

Now that we can calculate energies, we can have a first look at the dynamics of water by calculating the vibrational normal modes for a water molecule. The code is provided in the file `water_modes.py`, you don't need to type it in again. Vibrational normal modes are the fundamental vibrational motions of a molecule. Each mode consists of a pattern of motion for all atoms and an associated vibrational frequency. Note that vibrational normal modes are of interest mostly for small molecules. Other kinds of normal mode calculations are more appropriate for large molecules such as proteins. Vibrational normal modes are obtained by analyzing the potential energy surface of a system in the vicinity of a local minimum.

We start with a single water molecule, which we assign to a variable in order to be able to work on it later:

```
from MMTK import *
from MMTK.ForceFields import Amber99ForceField
universe = InfiniteUniverse(Amber99ForceField())
molecule = Molecule('water')
universe.addObject(molecule)
```

The first step is energy minimization. MMTK provides two “minimization engines” which use different algorithms. We use the conjugate gradient minimizer, which is more efficient at getting close to a minimum than the simpler steepest-descent minimizer:

```
from MMTK.Minimization import ConjugateGradientMinimizer
from MMTK.Trajectory import StandardLogOutput
minimizer = ConjugateGradientMinimizer(universe,
                                       actions=[StandardLogOutput(1)])
minimizer(convergence = 1.e-4, steps = 100)
```

The first two lines are imports, nothing new. Then a minimization engine is created for the universe. The second parameter (“actions=...”) is optional. It specifies additional actions that are performed periodically, usually for generating output. `StandardLogOutput(1)` prints some relevant information at every step as minimization goes on.

The last line starts the minimization engine. It is told to continue until it has converged to the minimum such that the average root-mean-square force on the atoms is 10^{-4} kJ/mol/nm or less, or to stop after 100 steps if convergence is not yet reached. The minimization takes only a few steps because the default configuration for water is very close to the energy minimum in the Amber 99 force field.

Now we are ready for the normal mode calculation:

```
from MMTK.NormalModes import VibrationalModes
modes = VibrationalModes(universe)
```

3.1 Analysis

First of all we look at the frequencies:

```
for mode in modes:
    print mode
```

The frequency unit is $1 \text{ THz} = 1 \text{ ps}^{-1}$. The first six modes (numbered 0 to 5) have frequencies very close to zero (those with a `j` in the end are imaginary). These modes represent the rigid-body motions of the molecule: three modes for translation, and three modes for rotation.

Exercise: Compare the time scales of the vibrations to the typical time steps used in Molecular Dynamics simulations (1 fs).

Advanced exercise: Calculate the distance between the quantum energy levels of a harmonic oscillator ($h\nu$, or `Units.h*modes[6].frequency` for the computer) and compare

it to the thermal energy at 300 K ($300 \cdot \text{Units.k}_B$). Is classical dynamics a reasonable approximation for a water molecule?

To get a first impression of what the modes look like, we can look at an animation:

```
modes[6].view()
```

By default, the amplitudes of the modes are calculated to correspond to the amplitude of thermal fluctuations at a temperature of 300 K.

Exercise: Give an approximate description of the three modes with non-zero frequency from the animations.

Next we do a quantitative analysis, using the two bond lengths (O-H₁ and O-H₂) and the bond angle (H₁-O-H₂) as convenient coordinates. First we must calculate their values in the minimized configuration:

```
d_O_H1 = universe.distance(molecule.O, molecule.H1)
d_O_H2 = universe.distance(molecule.O, molecule.H2)
a_H1_O_H2 = universe.angle(molecule.H1, molecule.O, molecule.H2)
```

You might wonder why the universe is mentioned in the calculation. The reasons is that distances and angles have to be calculated differently in periodic systems, and this information is stored in the universe.

Next we construct a configuration in which all atoms are displaced along the directions that correspond to the first normal mode:

```
minimum_configuration = copy(universe.configuration())
displaced_configuration = minimum_configuration + modes[6]
```

Then we calculate the internal coordinates for the displaced configuration:

```
universe.setConfiguration(displaced_configuration)
displaced_d_O_H1 = universe.distance(molecule.O, molecule.H1)
displaced_d_O_H2 = universe.distance(molecule.O, molecule.H2)
displaced_a_H1_O_H2 = universe.angle(molecule.H1, molecule.O,
                                     molecule.H2)
universe.setConfiguration(minimum_configuration)
```

Finally, we calculate the changes in the three internal coordinates:

```
print "Change in distance O-H1:", displaced_d_O_H1 - d_O_H1
print "Change in distance O-H2:", displaced_d_O_H2 - d_O_H2
print "Change in angle H1-O-H2:", displaced_a_H1_O_H2 - a_H1_O_H2
```

Exercise: Repeat this calculation for the three non-zero modes and compare the results with your visual analysis.

Exercise: The “CRC Handbook of Physics and Chemistry” gives the following spectroscopical values:

Mode	Frequency [cm^{-1}]
symmetric stretch	3657
bend	1595
asymmetric stretch	3756

Compare with the results of the calculation.

Note: Spectroscopists measure frequencies in inverse centimeters. Multiply by the speed of light to obtain real frequency units. With MMTK, the conversion can be written as `3657*Units.invc`.

3.2 From water to proteins

What if we want to do calculations on a protein instead of a water molecule? In principle, this is simple: replace the line

```
molecule = Molecule('water')
```

by

```
from MMTK.Proteins import Protein
molecule = Protein('some_protein.pdb')
```

However, keep in mind that bigger systems require more minimization steps, therefore increase the number of steps substantially. The convergence criterion is independent of the system size, so you can keep it at the same value, unless you want more precise calculations.

As the calculations take more time, it is a good idea not to have to repeat them too often. When doing a normal mode calculation, it is thus advisable to save the result to a file. We can achieve this by adding

```
save(modes, 'helix_modes.modes')
```

right after the calculation of the modes.

Exercise: Do a normal mode analysis of the alpha helix that is provided (`helix.pdb`). Save the resulting modes for use in a later exercise. Describe the first two modes that have non-zero frequency.

4 Molecular Dynamics

We will now turn to Molecular Dynamics simulations, using the alpha helix from the last section as a convenient toy example. First we have to define the system, which is done exactly as before:

```
from MMTK import *
from MMTK.Proteins import Protein
from MMTK.ForceFields import Amber99ForceField

universe = InfiniteUniverse(Amber99ForceField())
universe.addObject(Protein('helix.pdb'))
```

At this point, the system has a well-defined configuration, but no velocities at all. The following step assigns velocities from a random distribution corresponding to a temperature of 50 Kelvin:

```
universe.initializeVelocitiesToTemperature(50.*Units.K)
```

The rather low temperature is chosen intentionally. Since the system is not equilibrated yet, large initial velocities could have devastating effects. It is better to start at a low temperature and then heat up the system:

```
from MMTK.Dynamics import VelocityVerletIntegrator, Heater, \
    TranslationRemover, RotationRemover
from MMTK.Trajectory import StandardLogOutput

integrator = VelocityVerletIntegrator(universe, delta_t=1.*Units.fs)

integrator(steps=1000,
           # Heat from 50 K to 300 K applying a temperature
           # change of 0.5 K/fs; scale velocities at every step.
           actions=[Heater(50.*Units.K, 300.*Units.K, 0.5*Units.K/Units.fs,
                           0, None, 1),
                   # Remove global translation every 50 steps.
                   TranslationRemover(0, None, 50),
                   # Remove global rotation every 50 steps.
                   RotationRemover(0, None, 50),
                   # Log output to screen every 100 steps.
                   StandardLogOutput(100)])
```

After the usual imports, an integration engine with a time step of 1 fs is created. The choice of time step obviously depends on the system that is simulated; 1 fs is the most common choice for all-atom models of biomolecules. In the last

(long) statement, the integration engine is started for 1000 steps. The parameter `actions` specifies a list of actions that are executed periodically during the integration. The last three parameters of each action determine when the action is launched: first step, upper limit, and interval. The heater action is thus executed at every step (from 0 with no upper limit at interval 1), and the translation removal is executed every 50th step.

The heater action multiplies the velocities of all particles with a common factor such that the kinetic energy corresponds to a specific temperature. This temperature increases as the simulation progresses until the final temperature is reached. In our case, this will happen after 500 steps (250 K at 0.5 K/fs). For the second half of our simulation run, the velocities will be rescaled to 300 K at each step, which is useful to equilibrate the system.

The translation and rotation removal actions are less evident. In classical dynamics, which is the basis of Molecular Dynamics simulations, the total energy of the system, its momentum, and its angular momentum are conserved quantities. Our helix should thus conserve its initial momentum and angular momentum forever. However, due to the discretization of the equations of motion, this is strictly true in a simulation only for vanishingly small time steps. For the time step commonly used for biological systems (1 fs), the energy is well conserved, but momentum and angular momentum are not. A molecule initially at rest will thus start to move and rotate around its axis. The translation and rotation removers suppress this artefact.

The remaining action, `StandardLogOutput`, prints the time and the potential and kinetic energies to the screen. It is useful for monitoring the progress of the simulation.

Now that we have an equilibrated system (note however that for a real protein the equilibration time is much longer), we can start the Molecular Dynamics simulation that we wish to analyze later on. For analysis, we need to store the generated trajectory in a file, so the first step is to create this file:

```
from MMTK.Trajectory import Trajectory

trajectory = Trajectory(universe, "helix.nc", "w",
                       "MD run for an alpha helix")
```

The first argument is the object for which we wish to create a trajectory. We use the full system here, but for larger systems it often makes sense to store only a part, e.g. a protein without its solvent, in order to limit the file size. It is possible to create multiple trajectories in a single Molecular Dynamics run, so one could have one trajectory without solvent at every time step and another one with solvent but sampled less frequently.

The second argument is the file name, the third one specifies write mode (a new file is created, any existing file with the same name is overwritten), and the last

argument is an optional comment string that is written to the file.

MMTK trajectories use the netCDF file format, which is a standardized binary file format that is compact, platform independent, and self-documenting in the sense that a single file can contain not only the trajectory data, but also all the metadata necessary for its interpretation. An MMTK trajectory thus contains a description of the system that is simulated, the names of the variables that are stored, and even the units for all quantities.

The Molecular Dynamics run itself contains many elements we have already discussed:

```
from MMTK.Trajectory import TrajectoryOutput, RestartTrajectoryOutput

integrator(steps=1000,
           # Remove global translation every 50 steps.
           actions = [TranslationRemover(0, None, 50),
                     # Remove global rotation every 50 steps.
                     RotationRemover(0, None, 50),
                     # Write every second step to the trajectory file.
                     TrajectoryOutput(trajectory, ("time", "energy",
                                                    "thermodynamic",
                                                    "configuration"),
                                       0, None, 2),
                     # Log output to screen every 10 steps.
                     StandardLogOutput(100)])

trajectory.close()
```

The only new aspect is another periodic action. `TrajectoryOutput` makes sure that the state of the system is recorded in a trajectory regularly. Its arguments are the trajectory file object and a list of the variable categories that should be written. Some of these categories correspond to a single variable (e.g. "time"), others to several. The category "energy" includes all energy contributions: potential energy, kinetic energy, and possibly other terms such as a thermostat energy. The category "thermodynamic" includes temperature and pressure (if available).

Exercise: Run the example (the complete Python script is provided as `md.py`) and look at the resulting trajectory by running

```
tvviewer helix.nc
```

Look at the energies over time and click on "view" to obtain an animation. Re-run the example with a longer equilibration time (2000 steps). Does this make a difference?

Exercise: Considering that we wrote every second step to the trajectory, what is the frequency of the fastest motions that we can expect to find in the trajectory?

5 Analyzing a trajectory

5.1 Calculating averages

While Molecular Dynamics is a fairly standardized procedure, the analysis of the resulting trajectories depends very much on the system being studied and on the questions that one is trying to answer. Python and MMTK are an ideal combination for such tasks that invariably require some programming. As an illustration, we will calculate the average length of our helix during the simulation, as well as its fluctuation.

First of all, we need to decide what exactly we mean by length of the helix. For convenience, we will use the distance from the first to the last C_α atom, but other definitions are of course possible.

A protein object such as `Protein('helix.pdb')` consists in general of multiple chains, even though our example has only one chain. We can extract that chain by indexing:

```
protein = Protein('helix.pdb')
chain = protein[0]
```

A chain can also be decomposed by indexing, yielding the individual residues. The first residue is `chain[0]`, the last one is `chain[-1]`. A residue consists of a peptide group called `peptide` and a sidechain called `sidechain`. Each of these groups then contains its respective atoms. We can thus calculate the length of our helix as

```
helix_length = (chain[-1].peptide.C_alpha.position()
                - chain[0].peptide.C_alpha.position()).length()
```

Exercise: Calculate the length of the helix in the configuration given by `helix.pdb`.

For our trajectory analysis project, we need to repeat this calculation once for each time step stored in the trajectory. First of all we need to open the trajectory file for reading:

```
from MMTK import *
from MMTK.Trajectory import Trajectory

trajectory = Trajectory(None, "helix.nc")
universe = trajectory.universe
protein = universe[0]
chain = protein[0]
```

When we pass `None` as the universe argument, MMTK retrieves the definition of the universe from the trajectory itself. We can then access it as the attribute `universe` of the trajectory. Next we obtain the protein as the first (and only) object in the universe and the chain as the first (and only) object inside the protein.

For treating step after step, we clearly need a loop. There are a couple of ways to iterate over a trajectory. Here we choose a loop over the integers from 0 to the length of the trajectory:

```
for i in range(len(trajectory)):
    universe.setFromTrajectory(trajectory, i)
```

The statement inside the loop retrieves the configuration for step `i` (it would also retrieve velocities if there were any) and makes it the current configuration of the universe. We can then calculate the length exactly as shown above. But... what do we do with the result? We need to store all helix lengths for all steps to do statistics. The simplest approach is to accumulate them in a list. The complete loop thus becomes

```
lengths = []
for i in range(len(trajectory)):
    universe.setFromTrajectory(trajectory, i)
    helix_length = (chain[-1].peptide.C_alpha.position()
                   - chain[0].peptide.C_alpha.position()).length()
    lengths.append(helix_length)
```

The statistics calculations are then straightforward:

```
from Scientific.Statistics import average, standardDeviation

print "Average length:", average(lengths)
print "Fluctuation:", standardDeviation(lengths)
```

Exercise: Run the complete analysis script (`helix_lengths.py`). Redo the calculations for the first and the second half of the trajectory. Are the results similar?

5.2 Dynamics analysis

In our first analysis, we have used our trajectory merely as a sampling tool for calculating averages. We have not used the time scales of the motions at all. In the second analysis, we will look at the density of states, which describes the distribution of kinetic energy over the frequency scale. Experimentally, the

density of states can be obtained by spectroscopic techniques. However, any given technique will yield only a part of the total frequency range.

Exercise: Re-run the molecular dynamics script after modifying it to obtain a 10 times longer trajectory called `helix_long.nc`.

For this analysis, we will use a readily available analysis tool called nMOLDYN. In a shell window, type

```
xMoldyn helix_long.nc
```

In the menu “Dynamics”, select “Density of states ...” and then “from coordinates”. nMOLDYN does not allow to select “from velocities” because we did not store the velocities in the trajectory. A dialog box will pop up in which you don’t need to change anything - just press “OK”. In the second dialog box, click on “Run”. This will launch the calculation as a background job.

You can follow the progress of your job by clicking on “Show running calculations” in the menu “File”. Once it says that there are no more jobs running, you can quite xMoldyn. Your density of states is stored in the file `DOS_helix_long.plot`. Use Gnuplot, Grace, or any other plotting program to inspect it.

Exercise: Look at the density of states. What are the time scales of the slowest and of the fastest motions? Can you guess why there is such a large gap from 60 to 80 ps⁻¹?

The density of states can also be obtained approximately from a normal mode calculation. In fact, if one approximates the potential energy surface of the protein by a single harmonic well, then the density of states is proportional to the number of normal modes in a given small frequency interval. We can thus obtain the density of states from a normal mode calculation as a histogram of the vibrational frequencies. The Python script is provided as `spectrum.py`. The first step is to retrieve the normal modes of the helix that we calculated earlier and saved to a file:

```
from MMTK import *
modes = load('helix.modes')
```

Next we prepare a list of the non-zero frequencies:

```
frequencies = []
for i in range(6, len(modes)):
    frequencies.append(modes[i].frequency)
```

A convenient histogram generator is provided by ScientificPython:

```
from Scientific.Statistics.Histogram import Histogram
spectrum = Histogram(frequencies, 100)
```

We have chosen to use 100 bins for the calculation of the histogram. Given that we have about 1000 modes, this is a reasonable choice. We can now write the spectrum to a file:

```
from Scientific.IO.ArrayIO import writeArray
from Scientific import N
writeArray(N.array(spectrum), 'frequency_spectrum.plot')
```

Exercise: Look at the frequency spectrum. Does it look similar to the density of states?

It would be nice to compare both calculations on the same plot, but this is not straightforward because they are scaled differently. We will rescale our spectrum such that its area is identical to the area under the density of states curve. To this end we read in the density of states:

```
from Scientific.IO.ArrayIO import readArray
frequency_axis, dos = N.transpose(readArray('DOS_helix_long.plot'))
```

This yields two arrays, one for the frequency values and one for the density of states values. We can obtain a simple approximation to the area under the curve by

```
area = N.sum(dos)*(frequency_axis[1]-frequency_axis[0])
```

All that remains to be done is to scale the spectrum and write it again to a file:

```
spectrum.normalizeArea(area)
writeArray(N.array(spectrum), 'frequency_spectrum.plot')
```

Exercise: Compare the two curves by plotting them together. In which frequency range are the largest differences? Can you guess why?