EMBO Practical Course on Biomolecular Simulations: Normal mode analysis

Konrad Hinsen
Centre de Biophysique Moléculaire (CNRS)
Orléans, France

7 July 2000

1 Getting started

1.1 Python and MMTK

For the normal mode exercises we will use the Molecular Modelling Toolkit (MMTK). This is a library that contains most standard techniques (and some non-standard ones) for molecular simulations, and is written mostly in a high-level language called Python. The main goal of the exercises is to become familiar with normal mode techniques, not with MMTK or Python, and therefore the technical details are kept to a minimum. Python will appear again in the practical on programming, and will be treated in more detail there.
For more information, consult the following references:

- Python: `http://www.python.org/`
- MMTK: `http://dirac.cnrs-orleans.fr/MMTK/`

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

- Type "`python`" (without the quotes) and then type in the example code line by line. To exit from Python, type Ctrl-D.
- 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.

More convenient methods of using Python exist, but these two one have the advantage of working on all computers without additional programs.

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.
Some of the material and exercises are not essential or require special knowledge. They are indicated by this text style. You are invited to look at these parts if you have time, but it is not required to go on with the rest.

2 Preliminary exercises

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.
The initially somewhat unusual syntax in the last line occurs very frequently in Python. It executes an operation named “`view`” which was defined for the kind of objects to which “`molecule`” belongs. Anything that can be visualized supports an operation called “`view`”, although the details of what it does may vary. For example, we will see later that for normal modes this operation involves showing an animation.

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 define 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. The length unit in MMTK is the nanometer; if you prefer to specify Ångstrom values, replace `0.5` by `5*Units.Ang`.

We still cannot calculate energies, because there is no force field yet. MMTK provides several force fields, but the only one suitable for our purposes is the Amber 94 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 Amber94ForceField
universe = InfiniteUniverse(Amber94ForceField())
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()
```

3 A simple application: water

Now we are ready to tackle normal mode calculations. 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 Amber94ForceField
universe = InfiniteUniverse(Amber94ForceField())
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 generation 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 step because the default configuration for water is very close to the energy minimum in the Amber 94 force field.

Now we are ready for the normal mode calculation:

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

3.1 Analysis

First of all we look at the frequencies:

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

As expected, the first six modes (numbered 0 to 5) have frequencies very close to zero; those with a `j` in the end are imaginary. The frequency unit is 1 THz = 1 ps^{-1}. The modes with zero frequency represent the rigid-body motions of the molecule.

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

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.*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-H_1 and O-H_2) and the bond angle (H_1-O-H_2) 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: For historical reasons, 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 e.g. `3657*Units.invcm`.

4 A more interesting application: an α-helix

Since a normal mode study on a complete protein takes too much time, we limit ourselves to a piece of a protein: the 20 last residues (i.e. the C-terminal portion) of myoglobin (PDB entry 1MBD), which forms an α-helix. The original extract can be found in the file `helix_original.pdb`. A prepared version (with hydrogens added and energy minimized up to a gradient RMS of 10^{-3} kJ/mol/nm) can be found in the file `helix_minimized.pdb`.

Exercise: Compare the original and the minimized configuration visually.

The following Python program calculates the RMS distance between the original and the minimized configurations:

```
# Construct system
from MMTK import *
from MMTK.Proteins import Protein

universe = InfiniteUniverse()
molecule = Protein('helix_original.pdb')
universe.addObject(molecule)

original_conf = copy(universe.configuration())

# Load minimized configuration
from MMTK.PDB import PDBConfiguration
pdb = PDBConfiguration('helix_minimized.pdb')
pdb.applyTo(molecule)

minimized_conf = copy(universe.configuration())

# Calculate RMS distance
print "RMS distance:", molecule.rmsDifference(original_conf,
                                              minimized_conf)
```

Whereas a water molecule is constructed simply by specifying its name (which MMTK finds in its database), proteins are usually constructed from PDB files. This happens in line 6. The class `Protein` used here has to be imported explicitly, because it is not in the core part of MMTK. It handles a lot of operations: load the PDB file, generate a model for the protein, add missing hydrogen positions, and find disulphur bridges.
The file for the minimized configuration must be handled differently, because we do not want to create a second protein, but assign a new configuration to an existing one. This is handled by another class, which is also used internally when

a protein is created. The details are explained in the MMTK manual [2] for details.

Note: To get a more detailed description of the major changes, we can ask for the atoms that moved by the largest distance. This code will not be explained here, but it serves to illustrate what can be done with a few lines of Python code:

```
# Find the 10 atoms that moved most
distances = []
for atom in molecule.atomList():
    dist = (original_conf[atom]-minimized_conf[atom]).length()
    distances.append([atom, dist])
distances.sort(lambda x, y: cmp(x[1], y[1]))
for atom, dist in distances[-10:]:
    print atom, "moved by", dist, "nm"
```

The minimization and normal mode calculation proceeds just like for water, only one line must be changed:

```
molecule = Protein('helix_minimized.pdb')
```

However, to save time we will minimize only up to a gradient RMS of 10^{-2} kJ/mol/nm, and allow some more steps:

```
minimizer(convergence = 1.e-2, steps = 500)
```

After the normal mode calculation, it is advisable to save the result in a file, then it does not have to be repeated later:

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

Later, the modes can be retrieved by executing

```
modes = load('helix.modes')
```

Now run the minimization and the normal mode calculation. This will take some time.

Exercise: The input structure had been minimized to 10^{-3} kJ/mol/nm. Yet we minimize it again, and only to 10^{-2} kJ/mol/nm, and it still takes a considerable number of steps. Why? (Hint: looking at the PDB file might give you an idea.)

Exercise: Look at animations for some modes at the lower and upper end of the spectrum, as well as in the middle. Try to characterize the size of the units that move in each part of the spectrum.

4.1 Analysis

With so many densely spaced modes, it is useful to plot a frequency spectrum, which is nothing but a histogram of the frequency values. Since we have about 1000 modes, we can make a nice histogram with 100 bins:

```
from Scientific.Statistics.Histogram import Histogram
from Gnuplot import plot
frequencies = []
for i in range(6, len(modes)):
    frequencies.append(modes[i].frequency)
histogram = Histogram(frequencies[6:], 100)
plot(histogram)
```

Note that we can't just make a histogram of `modes.frequencies` because it contains imaginary numbers. The first six modes have to be excluded.

Exercise: The frequency spectrum of the helix looks much like the frequency spectrum of a complete protein, except at the very lower end. Can you explain why, based on your characterization of the modes in different frequency intervals?

Exercise: Which kinds of motions can safely be treated classically, and which ought to be treated quantum-mechanically?

As an example for calculating thermal averages from normal modes, we will calculate the thermal fluctuations of the atoms in the helix. These fluctuations are often compared to crystallographic temperature factors, although temperature factors also depend on other effects. The fluctuation of atom i can be expressed in terms of normal modes as

$$f_i = \sum_{\text{modes } j} \frac{k_B T}{m_i \omega_j^2} \left| \mathbf{u}_i^{(j)} \right|^2 ,$$

where $\omega_j = 2\pi\nu_j$ is the angular frequency of mode j and $\mathbf{u}_i^{(j)}$ is the displacement of atom i in mode j.
The following code calculates the fluctuations and plots those of the C_α atoms as a function of the residue number:

```
fluctuations = 0.
for i in range(6, len(modes)):
    fluctuations = fluctuations + 0.5*modes[i]*modes[i]
calpha = []
for chain in molecule:
    for residue in chain:
        calpha.append(fluctuations[residue.peptide.C_alpha])
plot(calpha)
```

Note that again the first six modes are excluded from the sum. Note also that the atomic displacements are already scaled by the amplitudes of thermal vibrations, such that only a factor 1/2 remains to be applied to obtain the fluctuations.

5 Domain analysis of proteins

Now we will turn to full proteins, even big ones, and try to analyze their slowest collective motions in terms of *dynamical domains*, i.e. regions that behave approximately like rigid bodies and which are seperated by more flexible interdomain regions. This could be done with short Python scripts as well, but we will use an interactive program called DomainFinder, which is written in Python and uses MMTK internally. To start DomainFinder, type “DomainFinder” (without the quotes). The following paragraphs give a short description of DomainFinder, for a complete description see the DomainFinder manual [3]

5.1 Normal mode calculation

After starting DomainFinder, load your protein structure using “Load reference structure...” in the File menu. DomainFinder will then propose reasonable values for the number of calculated modes and the number of modes kept for analysis. The number of calculated modes determines the accuracy of the normal mode calculations; the more modes you calculate, the better the description of the motions. The value proposed by DomainFinder should be taken as a minimum value. The number of modes kept for analysis has no influence on the quality of the results; it exists purely for efficiency reasons. The proposed value should in general not be changed.
To start the normal mode calculation, press the button labeled “Calculate modes”. The calculation takes from a few seconds to a few minutes. When the calculation is finished, DomainFinder displays a list of the modes with non-zero frequencies and the actual number of calculated modes. This number is in general different from the one you entered, because not all values are possible for technical reasons (see Ref. [4] for an explanation).
Now select the modes that you want to use for the deformation and domain analysis. To help with this choice, the normal mode list indicates the average deformation energy per residue for each mode. A deformation energy is associated with every atom; low values characterize rigid regions, whereas high values indicate flexible regions. A low average deformation energy thus indicates a mode with large rigid regions, which has a good chance of describing domain motions. There is no simple recipe for selecting an optimal set of modes (otherwise DomainFinder would apply it automatically!). As a general guideline, look for jumps in the average deformation energy from one mode to the next, and choose all modes before such a jump; there is no justification for selecting only some out of a larger set of modes with very similar deformation energies. Start with few modes, and add more modes only if you are not satisfied with the amount of detail in the domain analysis.
After selecting the modes, you must choose the deformation threshold that defines which regions are sufficiently rigid to be candidates for domains. This choice

is related to the mode selection; to have a reasonable number of rigid regions, the deformation threshold should be of the same order of magnitude as the average deformation energy of the modes you have chosen. A higher deformation threshold leads to larger rigid regions; if you find later that your domains cover too small a part of the protein, you should increase the deformation threshold.

5.2 Deformation analysis

The entry "Show deformation" in the Deformation menu shows the reference conformation with a color code representing the deformation energies. Blue atoms correspond to small deformation energies, green atoms are close to the deformation threshold, and red atoms are in particularly flexible regions of the protein. Blue and green atoms are thus below the deformation threshold and candidates for domains in a subsequent domain analysis; if you find that most of your atoms are yellow or red, you should increase the deformation threshold before starting the domain analysis. However, you should also consider the possibility that your protein has no domains, either because it is too flexible to allow a description of its slow motions by quasi-rigid substructures, or because the deformation associated with these motions is uniformly distributed over the whole protein.

5.3 Domain analysis

The techniques implemented in DomainFinder allow an identification of three types of regions in a protein [5]:

- Flexible regions, for which a description of the motion by rigid bodies is not useful. These regions are recognized during deformation analysis and not used at all during domain analysis.
- Rigid regions with uniform motion. These regions are recognized as rigid during deformation analysis; domain analysis then groups them together according to the similarity of their overall motion. However, these regions are not rigid bodies in any strict sense; they do show internal deformations, but these deformations do not destroy the uniformity of the overall motion. The term "dynamical domain" is best interpreted to describe these rigid regions only.
- Intermediate regions, whose internal deformation is sufficiently small everywhere, but systematic enough that over the size of the region it adds up to produce sufficiently different overall motion between extremal parts of the region. Such intermediate regions often occur in between dynamical domains.

The central parameter which allows a distinction between rigid and intermediate regions is the “domain coarseness” parameter. It specifies how similar the global motions in a region must be to be considered similar enough to form a domain (see Ref. [5] for details). However, this parameter does not have one specific “best” value which one should find in order to obtain the “right” domain decomposition. It is a parameter which should be varied, and the ensemble of results for several values of this parameter provides the information for identifying domains and intermediate regions.
In order to obtain the motion parameters necessary for the domain analysis, DomainFinder first divides the protein into small cubic regions containing on average six residues. For each cube, six motion parameters are calculated, three for translation and three for rotation. In case of a normal mode based analysis, there are six parameters per mode; this is one reason why a normal mode based analysis usually gives better results. The cubes are then grouped into domains according to the similarity of their motion parameters. For a single value of the domain coarseness, it is not possible to distinguish between rigid and intermediate regions; the word “domain” therefore refers to both type of regions in the program.
After choosing a value for the domain coarseness, select “Show domains” from the Domains menu. This causes a window to be opened which shows the domain decomposition for this coarseness level. In the top left, the protein structure is drawn with various regions indicated by colors. The list to the right of the structure display contains all these regions with their color and size. The order in the list is significant; the best-defined domains are listed first, and the last item(s) frequently contain cubes that do not really belong to any recognizable domain. The bottom picture shows a parallel-axis plot of the motion parameters for all cubes, color-coded by domain. In this plot, each line represents one cube, and each vertical axis one motion parameter. For well-defined domains, the lines belonging to the same domain (i.e. same color) should be very close, whereas lines belonging to different domains should be clearly separated. A wide band of lines indicates an intermediate region. The plot provides both a verification of the domain decomposition and a first impression of the nature of the domains. However, it should be interpreted with caution; the eye tends to consider two lines with small differences in all axes more similar than two lines which coincide in some axes but differ significantly in others, although from a mathematical point of view both situations are equivalent.
It should be noted that the residues shown in black do not belong to any domain, for one of the following reasons:

- they are part of a cube with less than three points, which is too small to permit the calculation of the motion parameters

- they are in a cube whose average deformation energy is higher than the deformation threshold

For more detailed information on a particular domain, click on that domain's entry in the domain list. This will open another window with information for this domain only. In the top left there is again the protein structure with just one domain highlighted. To its right, a list of all residues in the domain is shown. Below there is a parallel-axis plot showing only the cubes in this domain.
When you vary the domain coarseness limit, you will observe that some domains remain essentially the same, growing or shrinking only by small amounts and in response to significant coarseness variations, whereas others grow and shrink rapidly, or tend to break up into smaller parts as the coarseness limit is decreased. The first kind represents stable rigid regions, i.e. dynamical domains. The second kind represents intermediate regions. The parallel-axis plot at the bottom of the window helps in this classification by showing the variation of motion parameters within the domains at one glance.

Exercise: Perform a dynamical domain analysis for lactoferrin (PDB entry 1LFH).

References

[1] G. van Rossum, "Python Tutorial",
`http://www.python.org/doc/current/tut/tut.html`

[2] K. Hinsen, "MMTK User's Guide",
`http://dirac.cnrs-orleans.fr/MMTK/Manual/MMTK.html`

[3] K. Hinsen, "DomainFinder User's Manual",
`http://dirac.cnrs-orleans.fr/DomainFinder/DomainFinder.html`

[4] K. Hinsen, "Analysis of domain motions by approximate normal mode calculations", Proteins, **33**, 417–429 (1998)

[5] K. Hinsen, A. Thomas, M.J. Field, "Analysis of domain motions in large proteins", Proteins, **34**, 369–382 (1999)