Molecular Dynamics and Monte Carlo Simulations

Exercises

Assistants:
Mathias Dankl (mathias.dankl@epfl.ch, BCH 4110)
François Mouvet (francois.mouvet@epfl.ch, BCH 4111)
Justin Villard (justin.villard@epfl.ch, BCH 4111)

The prediction of molecular properties at finite temperature - no matter whether it be a chemical reaction in solution or in an enzyme - requires more than the tools treated in the course 'Introduction to Electronic Structure Methods'. Finite temperature effects can give rise to substantial differences between an (idealised) isolated system at a hypothetical 0 K and the physical system at $T > 0$ K. Molecules have kinetic energy, and the system's behaviour is not governed by its potential energy alone, but by its *free energy*, with the effects of entropy taken into account. Consider the basic thermodynamical concept of the free energy of a reaction, $\Delta G = \Delta H - T\Delta S$. Although a reaction may be predicted to be endothermic based on quantum chemical calculations at 0 K, the entropic term of the transformation may still favour the reaction to be exergonic. The reaction will thus be spontaneous at finite temperature (*cf.* the solvation of certain salts in water, where the solute cools down due to an positive reaction enthalpy, but the process is spontaneous due to a considerable rise in entropy).

Both Molecular Dynamics (MD) and Monte Carlo (MC) simulations are techniques that allow to obtain information about the statistical distribution of a system, and thus on its thermodynamic properties at finite temperature, entropic effects included. The link between the microscopic system and the macroscopic thermodynamic observables is established in a branch of physics known as statistical mechanics. During the following exercise sessions, you will apply the techniques that have been treated in the lecture to both theoretical and practical problems; ranging from simple coding over statistical mechanics to actually performing both MD and MC simulations. Although the scope of this course is too small to give you a complete overview of the field, you should be able to gain some insight into the basic methodology and concepts.

Theses exercises are based on various textbooks and the Molecular Simulations tutorial provided by the University of Amsterdam [1].

[1] http://molsim.chem.uva.nl/course

Grading and attendance

Please note that your attendance to the exercises is *mandatory*, during the entire 2 hours of the sessions. All the exercises can be finished within these 2 hours, and the assistants will be with you to answer any questions that you may have. Every set of excercises will be accompanied by a written report.

During all except the first exercise session, each person will spend 5-10 minutes with an assistant where they will be asked questions about the past week's exercises and the respective report, which must have been handed in as a hard copy at the beginning of the session. The answers to these questions will be graded and, together with the written report, contribute 1/2 to your overall grade. Please note that the last session will be an exception, as the grading will be based on the written report only. All reports need to be handed in two weeks after the respective session.

Since the scope of this course is limited, each exercise session is accompanied by one or several more involved bonus questions treating theoretical problems of relevance. Solving these bonus questions will give you additional points at every exercise session and can thus substantially improve your final grade of the course. Although the first exercises include some coding in `C++`, you will not be tested on your knowledge of `C++`, but on the understanding of the general concepts instead.

Your final grade will be based on 5 out of a total of 6 grades.

Contents

This list gives an overview of the topics that will be covered during the next weeks.

- Statistics: Numerical estimation of π using Monte Carlo methods
- Statistics: Statistical Mechanics and the Boltzmann Distribution
- Monte Carlo: Detailed Balance in Monte Carlo
- Molecular Dynamics: Molecular Dynamics of a Water Box
- Molecular Dynamics: Force Fields
- Molecular Dynamics: Biological System

Questions

We are here to help - please do not hesitate to contact us outside the scheduled hours. You may contact us by mail or schedule an appointment to discuss with us in person. If you notice any typos or mistakes in the exercise script, please notify the assistants.

1 Basic Concepts of Molecular Dynamics and Monte Carlo Simulations

In this first set of exercises, you will encounter some basic concepts that are important in molecular dynamics and Monte Carlo simulations. Later in the course, you will be introduced to the underlying formal derivations and relations. This set of exercises considers some general concepts of practical significance, including the link between quantum mechanics and classical mechanics, and a statistical approach to solving certain mathematical problems.

1.1 From Quantum to Classical Mechanics: The Example of Forces

Later in the course, you will learn that in *Molecular Dynamics*, thermodynamic properties of a system are determined by propagating it in time. This propagation is done by evaluating the forces acting on the nuclei, which are usually considered to be classical particles. The nuclei are then moved according to the forces which act on them. This *clamped nuclei* approximation is also at the base of the geometry optimisation procedures in electronic structure theory, as discussed in the course 'Introduction to Electronic Structure Methods'. Analogously, in clamped-nuclei first-principles molecular dynamics, the information on the forces is obtained from the electronic structure of the system by solving the time-independent Schrödinger equation.

In classical mechanics, the forces acting on a system can be evaluated from:

$$F(\mathbf{q}) = -\nabla E(\mathbf{q}). \tag{1}$$

The link with the time-independent Schrödinger equation is based on the Hellmann-Feynman theorem:

$$\frac{\mathrm{d}E}{\mathrm{d}\lambda} = \left\langle \Psi_\lambda \left| \frac{\mathrm{d}\hat{\mathrm{H}}(\lambda)}{\mathrm{d}\lambda} \right| \Psi_\lambda \right\rangle, \tag{2}$$

where λ is some parameter on which the Hamiltonian, and thus the wavefunction, parametrically depends. By considering the Hamiltonian of a system with N electrons and M nuclei within the Born-Oppenheimer approximation, the forces are easily evaluated from the Hellmann-Feynman theorem. The Hamiltonian reads:

$$\hat{\mathrm{H}} = \hat{\mathrm{T}} + \hat{\mathrm{V}}_{\mathrm{ee}} + \sum_{I=1}^{M}\sum_{i=1}^{N} \frac{Z_I}{|\mathbf{r}_i - \mathbf{R}_I|} + \sum_{I=1}^{M}\sum_{J\neq I}^{M} \frac{Z_I Z_J}{|\mathbf{R}_I - \mathbf{R}_J|}, \tag{3}$$

where $\hat{\mathrm{T}}$ is the kinetic energy operator, $\hat{\mathrm{V}}_{\mathrm{ee}}$ is the operator that mediates electron-electron interactions, and capitals denote nuclear and lower case letters denote electronic indeces respectively. In cartesian coordinates, the forces acting on the x-component $\mathbf{X}_I$ of nucleus I are:

$$\mathbf{F}_{\mathbf{X}_I} = -\left\langle \Psi \left| \frac{\mathrm{d}\hat{\mathrm{H}}}{\mathrm{d}\mathbf{X}_I} \right| \Psi \right\rangle. \tag{4}$$

Insertion into eq. 2 yields:

$$\mathbf{F}_{\mathbf{X}_I} = -Z_I \int \frac{\mathbf{x} - \mathbf{X}_I}{|\mathbf{r} - \mathbf{R}_I|^3} \rho(\mathbf{r}) \mathrm{d}\mathbf{r} - \sum_{J \neq I}^{M} Z_I Z_J \frac{\mathbf{X}_J - \mathbf{X}_I}{|\mathbf{R}_J - \mathbf{R}_I|^3}. \tag{5}$$

The forces on the nuclei are thus easily derived from the electron density $\rho(\mathbf{r})$ using an analytical expression. In *classical molecular dynamics*, the forces are not evaluated from the electron density, but are fully parametrised instead.

1.2 Statistical Approaches to Numerical Estimation

Monte Carlo (MC) methods are a broad class of computational algorithms which rely on repeated random sampling to obtain the distribution of an unknown, often probabilistic entity. They are particularly useful for problems in which it is difficult (or impossible) to obtain a closed-form expression, or to apply a deterministic algorithm. A more detailed discussion of MC methods will follow in the lecture course, but here we intend to introduce the topic with a practical example.

One of the simplest yet intuitive examples of an MC method is to estimate the value of π through numerical integration. This exercise will focus on the importance of sampling and maintaining a uniform distribution in the choice of sampling points. Since MC methods rely heavily on uniform randomly distributed numbers, there is a detailed discussion of pseudo-random number generators (PRNGs) in section 1.4.

1.2.1 Numerical Estimation of π

Consider a circle of diameter d, sitting at the centre of a square of length l as depicted below in Figure 1. If points (x,y) are randomly distributed within the square and those which fall within the circle versus the square are counted, numerical integration is effectively being performed *via* the MC method. The area ratio of the circle to the square thus provides a route to estimate π explicitly. It is worthwhile to note that in order to obtain an accurate approximation of π, the randomly generated coordinates must be uniformly

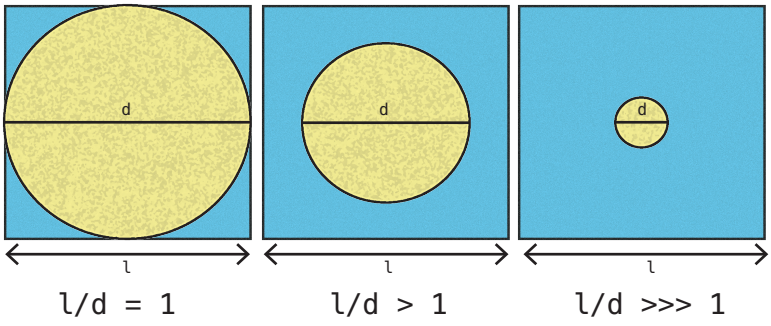

Figure 1: Schematic representation of modifying the l/d ratio. This ratio in part determines the accuracy of the π estimation.

distributed across the entire square to prevent bias. In addition, a large enough number of sample points must be used to appropriately approximate the areas (and their ratio).

1.2.2 Designing the C++ Program

This section will serve two purposes: to (re-)familiarise you with the C++ programming language, and also to guide you through the implementation of a π-estimation program using the MC method. In a directory of your choosing, create a `main.cpp` file and open it in a text-editor. An entry-point for where your program should execute must now be created. In your `main.cpp` file add the following:

```
1  #include <iostream>
2  #include <time.h>
3  #include <cstdlib>
4
5  double random(double lower, double upper);
6
7  unsigned int globalSeed;
8  unsigned int successfulHits = 0;
9  unsigned int numberOfTrials;
10 double circleRadius;
11 double squareLength;
12 unsigned int answerPrecision = 15;
13
14 using namespace std;
15
16 int main(int argc, char ** argv) {
17     return 0;
18 }
```

This code will form the base of your MC method. The variables `numberOfTrials`, `circleRadius` and `squareLength` are necessary for the π estimation. The variable `globalSeed` is a requirement for our pseudo-random number generator while the variable `answerPrecision` is simply to ensure you print out an appropriate number of decimals for the estimation. You can find a detailed description of libraries (iostream, time, cstdlib), forward declaration, main functions, command-line arguments, namespaces, global variables and variable scope in section 1.5.1. Below the main function, insert the following code.

```
1  double random(double lower, double upper) {
2      double zero_to_one = (double) rand() / RAND_MAX;
3      return (lower + (upper * zero_to_one));
4  }
```

This function returns a uniformly distributed random number. It makes use of the PRNG `rand()`. The seed for this PRNG must also be initialised so insert the following code within your main function:

```
1 srand(time(NULL));
```

For further information about seeds, using PRNGs and the functions `srand()` and `time()`, see section 1.5.2. Your goal is to estimate the value of π using the circle and square method, thus the dimensions of the circle and square must be parsed into your program, as well as the number of trials you wish to perform this estimation with. This is achieved using the functions `atoi()` and `atof()`. Place the following early within your main function:

```
1 numberOfTrials = atoi(argv[1]);
2 circleRadius = atof(argv[2]);
3 squareLength = atof(argv[3]);
```

For more information about the functions `atoi` and `atof`, see section 1.5.3. You may also wish at this point to print out these variables, to ensure that the the correct values are being passed into the program. This can be achieved using the following example:

```
1 cout << "Number of Trials" <<  numberOfTrials << endl;
```

The cout object is used to print a character stream to the terminal, while the endl function flushes the stream. For further information about using standard output within C++, see section 1.5.4. You can now begin with the MC method. Enter the following into your main.cpp file in an appropriate place (i.e, once all variables are parsed, and the seed is set):

```
1     for (int i = 0; i < numberOfTrials; i++) {
2         // 1) Generate a new point (x,y) within the square.
3         // 2) Test whether this new point resides within the circle.
4         //       2.1) If above is true, increase the hit counter.
5     }
6
7     cout << "Successful hits: " << successfulHits << endl;
8      double piEstimation = 0;
9     //piEstimation = ??
10    cout.precision(answerPrecision);
11    cout << "Pi estimate: " << fixed << piEstimation << endl;
12    return 0;
```

You may notice that the code is commented. This is intentional, as you must now modify this section of the code to produce a functioning π estimation program. Follow the steps in the comments and ask for help if you are stuck.

1.2.3 Compiling Your C++ Program

To compile the program, navigate your terminal focus to the directory which contains `main.cpp`, and type the following:

```
g++ -std=c++11 main.cpp -o pi.x
```

This will produce an executable file called `pi.x` in your current directory. You can execute this program using the following command-line arguments:

```
./pi.x NumberOfTrials SizeOfCircle SizeOfSquare
```

1.3 Exercises

1.3.1 Numerical Estimation of π

a) How do you calculate π using the ratio of points that fall within the circle and square? Complete the Monte Carlo program to calculate π using this integration technique. Include the entire code within your report and comment upon the lines of code that you wrote.

b) Perform the π estimation for 1000, 1,000,000 and 100,000,000 trials. Take a screen-shot of these estimations and include them in your report. What happens to the accuracy of the π estimation when going from 1000 to 100,000,000 trials and why?

c) What happens if you use the same seed for the PRNG?

d) What happens to the estimation of π when the circle origin is changed? Why?

e) What happens to the accuracy of the estimation when you increase the square size, or decrease the circle size? Is there an optimal ratio?

1.3.2 From Quantum to Classical Mechanics

a) Prove the Hellmann-Feynman theorem, equation (2).

b) **Bonus:** Explain the Born-Oppenheimer approximation in your own words. You do not have to use any equations (but you may if you wish).

1.4 Random Sampling: Random and Pseudo-random Numbers

Monte Carlo methods require a source of randomness. It is desirable that these random numbers are delivered as a stream of independent $U[0,1]$ random variables. It is a necessity to generate random numbers uniformly, such that bias is not introduced into any physical property we wish to predict or estimate. There are two options for generating random numbers: using a physical or pseudo-random number generator. It would be satisfying to generate random numbers from a process that, according to a well established understanding of physics, is truly random. From this, our mathematical model would then match our computational method. Devices have been built that generate random numbers from physical processes such as radioactive particle emission, that are thought to be truly random. Unfortunately, physical random number generators are awkward to use in practice: simulations cannot be rerun so generated numbers have to be non-compressively stored, and random numbers cannot be supplied particularly fast. In contrast, a pseudo-random number generator (PRNG) uses simple recursions and modular arithmetic and thus are much faster. The pseudo term refers to the fact that it is possible to observe the sequence produced by the PRNG, infer the inner state and then predict future values. Pseudo-random number sequences are not truly random, however they can still pass the necessary tests for randomness. For some applications such as cryptography it is necessary to have pseudo-random number generators for which prediction is computationally infeasible, but Monte Carlo sampling does not require this caveat.

Designing pseudo-random number generators is outside the scope of this course, however, some basic examples are discussed below for your interest. A well-known example of a PRNG is the multiple recursive congruential generator (MRG):

$$x_i \equiv a_1 x_{i-1} + a_2 x_{i-2} + \ldots + a_k x_{i-k} \pmod{M}, \tag{6}$$

where $k \geq 1$ and $a_k \neq 0$. Another PRNG worthy of note is the lagged Fibonacci generator (LFG) which takes the form:

$$x_i \equiv x_{i-r} + x_{i-s} \pmod{M}, \tag{7}$$

with carefully chosen r, s and M. The LFG is a special case because it is rather fast. Interestingly, the optimal ratios of $i - r$ and $i - s$ have been found to closely match the golden ratio.

There are a number of very good and thoroughly tested generators. Among these high-quality generators, the Mersenne twister algorithm (MT19937) of Matsumoto and Mishimura (1998) has become the most prominent. Sometimes, however, very bad number generators are embedded in general or specific purpose software. L'Ecuyer and Simard (2007) published very extensive results that found many operating systems, programming languages and computing environments to have random number generators that failed many tests of randomness. To conclude, it is best to check documentation to be sure that your environment or programming language of choice implements a suitable PRNG by default.

1.5 C++ Glossary

1.5.1 Libraries, Forward Declaration, Main Function, Command-line Arguments, Namespaces, Global Variables and Variable Scope

Libraries Libraries are pre-compiled code that contain useful functions for a program to use. The iostream, time and C standard general utility (cstdlib) libraries are included through the `#include` pre-process directive. The `#include` pre-process directive tells the compiler that you wish to include the contents of <library> into your source file before it is converted to machine code, thus enabling you to use the aforementioned functions in your program.

Forward Declaration The `double random(double lower, double upper)` declaration is what is known as a forward-declaration of a function and serves to inform the compiler that this function is defined somewhere within your code.

Namespaces The namespace `std` is the namespace in which the C++ standard library functions are declared (standard input, standard output, random, string, regex... etc). Namespaces are simply structures to prevent function name conflicts.

Main Function The main function is a special function which defines the point at which your program begins execution.

Command-line Arguments The parameters `argc` and `argv` are special variables which are passed to the program from the command-line as arguments. The variable `argc` is an integer, while the variable `argv` is a pointer, which is an address to some useful data within your computers memory. In this case `argc` is simply a count of the number of arguments passed to the program, while `argv` points to an 2-dimensional array of characters, i.e the passed arguments themselves. It is worth noting that arrays in C/C++ are represented in their true form: a pointer to the beginning of the memory block at which the array starts. Thus, individual characters passed to the program as command-line arguments can be accessed by `argv[i][j]`, where `i` is the index of the string being passed, and `j` is the index of an individual character within the aforementioned string.

Global Variables and Variable Scope Variables can optionally be defined with access modifiers through the keywords public, protected and private. These access modifiers change the scope of a variable, i.e from where this variable can be acccessed within the program. If no access specifier is present, as is the case for the variables `numberOfTrials`, `circleRadius` and `squareLength`, then the access specifier `private` is inferred. The public specifier infers that the associated variable is accessible throughout the entire program. The private specifier limits the scope of the variable to that of the class in which it is defined, while the protected modifier performs the same role except that it is also accessible from derived classes. In this example you have only defined one class/file

and hence it is perfectly fine for you to not label the variables with an access modifier. In contrast, variables defined within functions are only accessible within that function.

1.5.2 Generating Random Numbers and PRNG Seeds

Generating Random Numbers The `<random>` library provides many ways to generate pseudo-random numbers but are overly complicated for your purpose. Instead, you use the function `rand()` defined within the cstdlib library. The function `rand()` is a pseudo-random number generator (PRNG) and returns a uniformly distributed value between 0 and `RAND_MAX`. In this implementation, the function `rand()` is wrapped with another function `random(double lower, double upper)`. Within this new function `rand()` is first modified slightly to create a uniformly distributed random number within the range [0, 1], and then this is applied to the lower and upper parameters to create the desired domain distribution within the bounds [`lower`, `upper`].

PRNG Seeds The seed is an integer which `rand()` uses to generate a sequence of pseudo-random numbers. Using the same seed will generate the same pseudo-random number sequence. Since you are building a stochastic model, it is most useful that the seed is unique every time your program is run. The function `srand(int)` sets the seed for the `rand()` PRNG and is defined in the `<cstdlib>` library. Here you pass to `srand(int)` another function `time(time_t *)` which is defined in the ctime library `<ctime>`. The argument of `time(time_t *)` allows you to calculate the time passed between now and the time passed in to the function, through the `time_t type`. However, you envoke the special case of `time(timer_t *)`, such that if the argument `timer_t *` is `NULL`, then this function returns the number of seconds since the Epoch (defined as 00:00 hours, Jan 1, 1970 UTC). This number is a suitable seed for your purposes, since your program should execute in a time period longer than one second, hence your sequence of pseudo-random numbers should be unique upon each calculation.

1.5.3 Library Functions

The function `atoi(const char *)` retrieves an integer from an input string, while the function `atof(const char *)` performs the same but for a floating point. These functions are defined in the `<cstdlib>` library you import at the beginning of your code.

1.5.4 Standard Output

The `cout` variable is an object of class ostream. This represents the standard output stream for narrow characters (and corresponds to the C stream `stdout`). The standard output stream is the default destination of characters determined by the enviroment, in this case, the terminal. The operator `«` is an overloaded operator to provide the functionality of pushing characters and strings to the stream buffer. Finally, the `endl` function simply inserts a new line character and flushes the stream. In other words, the line `cout « "Hello" « endl;` prints the message "Hello" to your terminal.