CHM 579 LAB 1: BASIC MONTE CARLO ALGORITHM

The goal of this lab is to get familiar with a simple Monte Carlo program and to be able to compile and run it on a Linux server.

Lab Procedure:

Before starting with the procedure below, make sure that you know basic things about the Linux command line environment.

You have 15 files that contain an implementation of the Monte Carlo algorithm in *FORTRAN 90*. If you are not familiar with *FORTRAN*, examine these files one after another following the directions and explanations below. You may go to step 15 right away if you are a *FORTRAN* guru.

<u>Step 1</u>

This is our first *FORTRAN* program. It does not do much, only prints a line on the terminal in which you are running the program.

Note how comments are made:
• An exclamation sign (`!`) means that anything to its right is a comment.

Compile and run the file mc-001.f90

To compile: `gfortran mc-001.f90 -o mc-001.out`

This command will tell the computer to make the *gfortran* compiler to take the code written in the file `mc-001.f90` and compile it into an executable output file called: `mc-001.out`

Once the file is produced it may be executed with the command `./` which will execute the program on the screen.

To run the code: `./mc-001.out`

However, in this lab it will be convenient to “stream” whatever the program prints on the screen to a log file, which can be reviewed again at any other time.

To run the code and send the output to a file: `./mc-001.out >& mc-001.log &`

Compare the `mc-001.log` file with what you saw on the screen when you executed the program without streaming the output.

<u>Step 2</u>

In any program that does anything there are a number of variables.

The statement `implicit none` forces the explicit definition of all the variables.

There are several variable types: `integer`, `real` and `double precision` are just three examples. `real` and `double precision` variables are to store floating point numbers.

The "variable" `np` is defined as a `parameter`, which means that its value will never change. Therefore it is a constant!

The variables `x`, `y` and `z` are actually arrays. Each one of them stores `np` (20, in this case) real numbers. We use `x`, `y` and `z` to keep the Cartesian coordinates of the particles in the simulation.

Note the structure of a `do` loop. It will assign the value 1 to the variable `i`, and execute all the statements between the `do` and `enddo` keywords. Then, the value stored in variable `i` will be *incremented* in 1, and again execute everything between the `do` and `enddo`.

This process will be repeated until `i` is larger than `np`.

Compile and run the file `mc-002.f90`

We have not assigned any value to the coordinates, so the printed numbers have no meaning.

Step 3

We introduce the parameters for the Lennard-Jones interaction potential.

The selected values correspond to the Oxygen atom in the SPCE water model, but this is irrelevant at this point.

Mathematics with *FORTRAN*:

- `a*b` means `a` times `b`
- `a/b` means `a` divided by `b`
- `a**b` means `a` to the power `b`

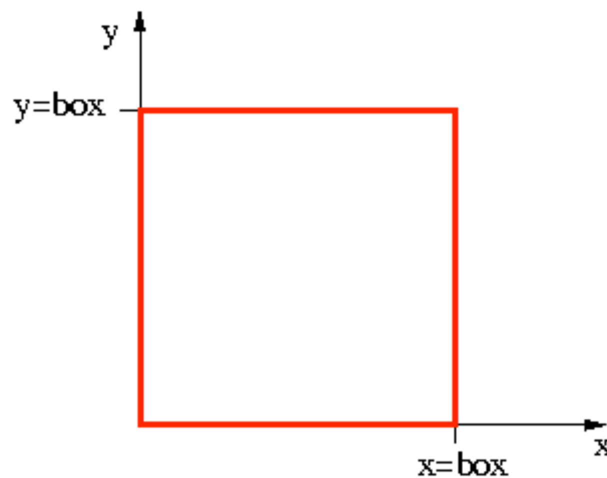
Compile and run the file `mc-003.f90`

Now we have assigned values to the coordinates!

Step 4

We assign values to the initial coordinates of the particles using a *(pseudo) random number generator*. This is not the most common way to initialize the system, but is easy to understand, so for our simple program it is a good alternative.

Imagine the simulation box as a cube with one corner in the origin of the Cartesian coordinates system:



Then, the coordinates of the particles should be a number between `0` and `box`.

The function `ran()` returns a real number between `0` and `1`.

Then, the output of `ran()` is multiplied by `box` to get a coordinate in the proper range.

Compile and run the file `mc-004.f90`

IMPORTANT: The function `ran()` is not standard *FORTRAN* and may not be available in compilers other than *gfortran*.

Step 5

To start the simulation we need not only the initial coordinates, but also the initial energy.

At this point we introduce the variable `u`, which will be used to keep the value of the potential energy of the system throughout the simulation. Also we define some auxiliary variables (`ddx`, `ddy`, `ddz` and `dd`) to calculate the distance between particles.

The total potential energy is defined as the sum of the Lennard-Jones interaction of all the pair of particles in the system:

$$U_{LJ} = 4\epsilon \sum_{i=1}^{np} \sum_{j=1}^{i-1} \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right]$$

The distance between particles "`i`" and "`j`" is calculated by first finding the distance along each Cartesian component. Then, the sum of the squares of the components is the square of the distance.

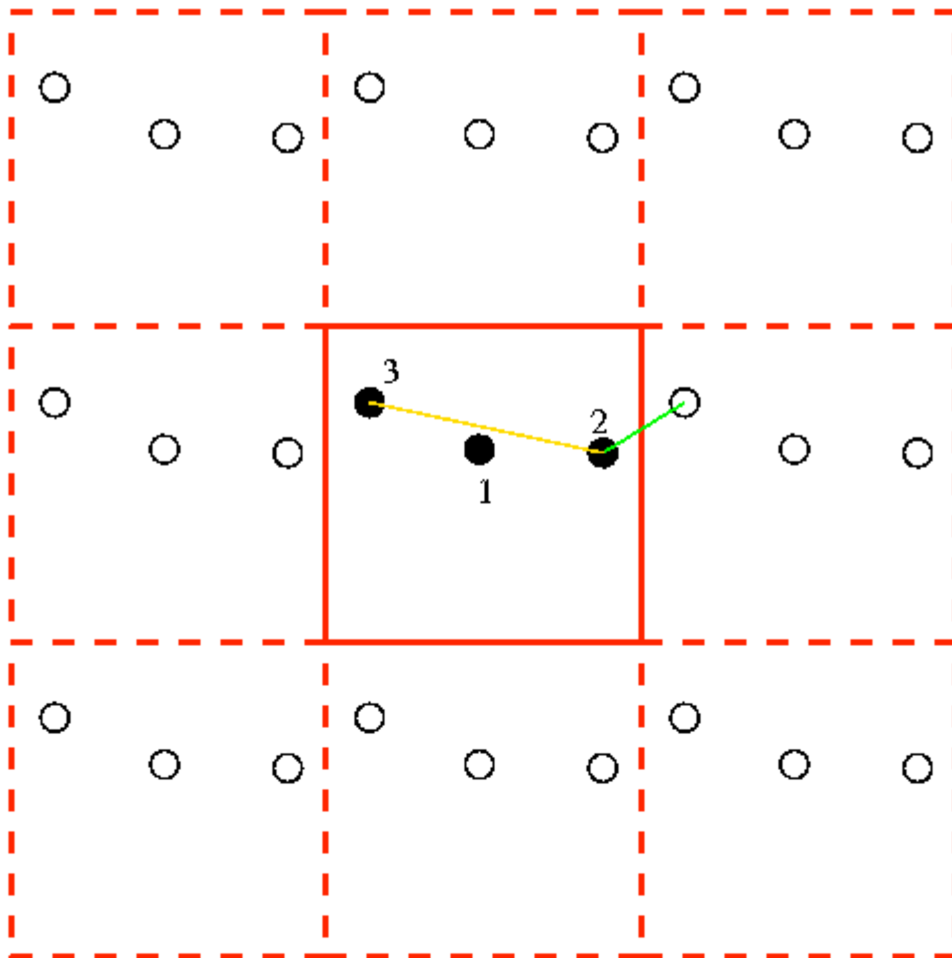
Note the function `sqrt()` which calculates the square root of its argument.

IMPORTANT: So far we have not applied periodic boundary conditions, necessary to mimic bulk conditions.

Compile and run the file `mc-005.f90`

Step 6

In order to mimic a bulk system we imagine that our simulation cell is replicated and shifted in all six Cartesian directions.



With this picture in mind, we should decide the proper way to calculate the energy. We will apply here the simplest criterion, which states that the distance between particles "`i`" and "`j`" is the minimum possible distance between these two particles when both of them are in the simulation box, or one of them is in any of the neighboring replica boxes.

Then, the 1-2 and 1-3 distances are taken within the central box, but the distance 2-3 is shorter if we consider particle 2 in the central box, and particle 3 in the replica to the right of the central box.

Using this *minimum image criterion*, the maximum distance (parallel to a Cartesian axis) is `box/2`. Note in the code how we have applied this criterion.

Note the structure of the `if` conditional operator.

Compile and run the file `mc-006.f90`

Step 7

A spherical cut-off is necessary to have an isotropic system. As we said previously, the maximum distance (along a Cartesian axis) between particles is `box/2`. In the XY plane, for example, the maximum distance between two particles is `sqrt(2)*box/2`. Therefore, without imposing a cut-off to the interaction will have a longer range along the diagonals than along the Cartesian axis.

The spherical cut-off distance has to be at most equals to half the box length.

Now the initial energy is calculated now.

Compile and run the file `mc-007.f90`

Step 8

The energy will be calculated many times during the simulation. Therefore, it is convenient to have a separate function, or subroutine, that performs the energy calculation. Note that we pass all the necessary information to the subroutine in the argument section, which is indicated by the parenthesis. The last argument is an integer number which should be equal to "`0`" in order to calculate the total potential energy. Other options will be necessary in the following steps.

Except for the introduction of the subroutine, there is no difference with program `mc-007.f90`

Compile and run the file `mc-008.f90`

Step 9

In a Monte Carlo simulation we need to select a particle and displace it from its current position. The move will be accepted or rejected based on a criterion that depends with the change in energy of the system.

In this program, we select at random the particle to move.
The line: `it=int(ran()*np)+1` assigns an integer between `0` and `np` to the variable `it`, which is the particle that we will attempt to displace.

In the subroutine `energy`, we have added the possibility of calculating the interaction energy of any single particle with the rest of the system.

Compile and run the file `mc-009.f90`

Step 10

The selected particle is displaced from its position by a random distance. It is important to store the particle's coordinates, in case we decide not to accept the move. Also, once the particle has been moved we check if it is inside or outside the central box. In the latter case, we apply the periodic boundary conditions to put the particle back into the central box.

Compile and run the file `mc-010.f90`

Step 11

All the moves that lower the energy are accepted.

Compile and run the file `mc-011.f90`

Step 12

In a Monte Carlo program, we attempt to move the particles many times, therefore we use a do loop to repeat the procedure.

Compile and run the file `mc-012.f90`

Step 13

As we said, every time the energy is lowered by a particle move, the move is accepted. When the energy rises, we accept the move with a probability `P=Exp[-DeltaU/kT]`. In practical terms, this means to compare `P` with a random number `RN` between `0` and `1`. If `P>RN`, we accept the move.

Compile and run the file `mc-013.f90`

Step 14

By using a subroutine to perform the `move` procedure, we get a more compact and readable main program.

We have introduced a new variable that will count the number of accepted moves.

Compile and run the file `mc-014.f90`

Step 15

To complete this first stage in the development of the Monte Carlo program, we define the variable `mcsteps`, which represents the number of attempted moves. Also, we monitor the potential energy of the system, writing its value every 100 steps. This energy is recorded in the file `energy.dat`. Once the simulation is completed, examine the energy file. Plot the energy vs the Monte Carlo step!

Compile and run the file `mc-015.f90`

Step 16: YOUR TURN!

(A) Random Numbers

Run the program from step 15 twice and compare the results, geometries and energies.

1. Are they any different?

You may wonder why it is so. The reason is in the way the random number generator is initialized, *i.e.*, what is its *seed* number.

Find the first call to `ran()` in the program. You see that there is no explicit initialization. It means that a default integer, which may be different between different compilers and different operating systems, is used as *seed* each time, and a sequence of random numbers is the same in all runs. In order to make the random number sequence *really* random, do the following trick: before the first time `ran()` is called, add a line:

```
call srand(seed)
```

Here seed should be any odd integer. Remember to initialize it in the beginning of the program. Providing different seed numbers, you will get different random sequences and different results. To make the randomization procedure really automatic, you can use current time as an argument. For example, like this:

```
seed1=2*int(secnds(0.0))
seed=7654321+seed1
```

```
call srand(seed)
```

Here the current seconds (counted from midnight) are added to an integer which will result in different seed numbers unless you run the program too often.

To make sure that this procedure works properly, compile the program and run it twice, saving and comparing both results.

(B) Different Step Size

2. Plot the energy as a function of the step number. Note what the acceptance rate is.

Decrease the step size in the program (10 and 100 times smaller).

You can do that by changing the following lines in the subroutine `move` within the program:

```
x(it)=x(it) + (ran()-0.5)          →          x(it)=x(it) + 0.1*(ran()-0.5)
y(it)=y(it) + (ran()-0.5)          →          y(it)=y(it) + 0.1*(ran()-0.5)
z(it)=z(it) + (ran()-0.5)          →          z(it)=z(it) + 0.1*(ran()-0.5)
```

Compile and run the program again; observe the acceptance rate and energy changes.

3. Do you see any interesting differences?

Hint: you may look in more detail at a shorter segment of the energy plot in order to see interesting changes.

(C) Different Temperature

Change the temperature. Again, recompile and rerun.

4. How are the energy and the acceptance rate different?

5. In the lab report, provide acceptance rates and energy plots corresponding to different step sizes and temperatures. Explain observed differences.