Molecular Statistics

Exercise 1

Introduction

This is the first exercise in the course Molecular Statistics. The exercises in this course are split in two parts. The first part of each exercise is a general introduction to new concepts and programming techniques in the python programming language which covers and extends what is written in the curriculum. Here, small exercises labeled (a) to (z) are present to get you going. The second part of each exercise is the exercise itself with exercises labeled (1) to (n).

Python

The easiest way to get started with python (and to get comfortable with the language[1]) is to start its interactive version in what we will call the *interactive python shell* (or just the python shell). Alternatively, you can start the GUI named IDLE from the start menu. Here, you can enter commands that the python interpreter will interpret, i.e. simple calculations, iterating over lists and such.

Assuming you have started a terminal, you start the python shell by writing the following command

```
python
```

and hitting return. What you should see is the cursor blinking in front of a line of text saying “>>>”. This means everything is set up correctly and python is ready to obey your commands.

As was shown to you this morning, the interactive python shell can add, subtract, multiply and divide numbers.

(a) Try out the following statements. Does it behave like your regular calculator? If anything unusual happens, try and explain why. *Hint: Remember that each statement is executed by pressing return.*

```
5+9, 5-9, 5*9, 5/9, 5+2*9, 5.0 + 2.0*9.0, 5.0 / 9
```

Since manually entering numbers really does us no good, we might as well use a calculator. Instead, we want to utilize what programming languages can provide, namely storage of values in what is known as *variables*. Variables can store anything that you can think of, i.e. numbers, strings, lists of numbers and so forth. Variables are assigned values by specifying a name and a value, i.e.

```
myFirstVariable = 5.0
```

where a variable named `myFirstVariable` has been assigned the value of 5.0.

1 Even Harry Potter will think you are cool when you tell him that you too speak python(!).

(b) Try to make the expressions from exercise (a) using variables. Do the calculations give the same results as in (a)? *Hint: You can also mix numbers and variables – try it!*

We are not restricted in our naming of variables at all and I encourage you to give them useful names such xpos if you are saving the x-coordinates for instance. That way, you can more easily remember what values are stored.

Speaking of x-coordinates, lets try to store an array of numbers. In today's exercise, lists will do the job for us. Without further ado, jump back to the terminal and enter the following command

```
simpleList = []
```

which creates an empty list named simpleList. Even though it is empty, we can view its contents by writing `print simpleList`. The results [] indicates that it is a list with no contents. The type can be further verified by the `type(simpleList)` statement which states that simpleList is of the type list. An empty list, however, won't do us much good, so to add values to the list, we use the following command

```
simpleList.append(1.0)
```

to add the number 1.0 to the list.

(c) View the contents of the list as well as the type. What changed?

(d) Add the following numbers to the list: -1.0, 1.5, 2.0, -2.0, -3.0, 3.0 and view its contents again. A list in python has several methods which we can use to modify the list itself. For instance, by writing

```
simpleList.sort()
```

in the python terminal, we can sort the list. Try it and view its contents and comment on what changed. Compare it with the previously printed list before you sorted it.

Another way of adding items, is to do it directly when we create the list. This is done almost the same way as when we created the empty list, except we provide the initial content right away. This is accomplished using

```
qList = [1.0, 2.0, 3.0, 4.0, -2.0, -4.5, -1.0]
```

(e) Redo exercise (d) but this time using `qList` instead of `simpleList`.

We've now seen that python lists can be created in various ways and even sorted,[2] but it is quite tedious to enter the data manually, especially if there is a lot of it. Luckily, python provides us with the means to construct lists using various other approaches. The `range` command is probably the command you will spend most time with during this course, so I suggest you already now become friends with it.

2 For a complete list(!) of methods that you can use on the lists you create, I forward you to section 5.1 on the python homepage: http://docs.python.org/tutorial/datastructures.html

(f) Write the following commands in the python shell and explain the results before moving on. *Hint: use the* `type` *command to get the data type of the commands, i.e. type(range(10))*

```
range(10)
range(3,10)
range(-3,10,4)
```

List on a List

Another thing we want to be able to do, is use the numbers in one list to create new numbers in another list. This is smart if we want to tabulate the values y for a function $y=x^2$ provided that we have the points x in a list. Python allows the construction of the list of y-values in several different ways which we will now explore.

(g) Define a list named `xvalues` to be a range of integers from -5 to 5 (both included). *Hint: Look in exercise (f) for information about what numbers you will need to supply to the range function.*

(h) Generate the y-values by writing the following command

```
yvalues = [x**2 for x in xvalues]
```

and comment on the results. This technique is known as list comprehension.[3]

This is by far the simplest way to generate a list of values based on another list. There are two other properties of lists we need to consider before moving on to today's exercise. The first is to get the length of a list, i.e. get the number of elements that a list contains. It is a simple function called `len` which takes the list from which you want to get the number of elements. You can invoke it by

```
len(xvalues)
```

(i) you should try and guess what the length of yvalues is before your write len(yvalues) – did your guess match pythons answer?

The last thing we need to investigate today is a way to get hold of the i'th element in a list provided that $0 \leqslant i < N$ and N is the number of elements in the list.

(j) To get the i'th element of the list `qlist`, write the following statements and describe the results

```
qlist[3]
qlist[0]
qlist[-1]
qlist[len(qlist)]
qlist[len(qlist)-1]
```

(k) what is the index of the first item in a list? Also, what is the index of the last item?

We are now ready to construct the y-values again, this time using a slightly different approach

3 I suggest you look at http://docs.python.org/tutorial/datastructures.html#list-comprehensions for more information since this is an important point.

which is more general than the method you saw before. It is however, also a little bit more complicated to understand, but I urge you to really try and understand the next sections of code and text as they are integral to solving the final exam projects as well as the exercises.

(l) We shall use the x-values array as a basis again for determining the y-values in the equation $y=x^2$. We will construct an empty list `yvalues2` and then for each element x_i in the x-values array `xvalues`, we will calculate the corresponding y-value and append it to `yvalues2`.

```
yvalues2=[]
for i in range(len(xvalues)):
  sqr = xvalues[i]**2
  yvalues2.append(sqr)
```

NB! You'll notice the ":" at the end of line 2 above, this means that you have to indent[4] your code accordingly. You'll also notice that the python terminal changes from ">>>" to "...". When you are done, hit return twice to execute it.

(m) View the contents of the `yvalues2` and compare them to the yvalues. Are there any differences?

Plotting

Lastly in this introduction we want to plot the function $y=x^2$ using python. To do so, we will need to import a library which allows us to plot numbers in a regular coordinate system. Such a library is already present on the fys.ku.dk servers and it is called matplotlib. To load it, you write:

```
import pylab
```

which tells python to use the library `pylab` and to access its methods, use `pylab` as the prefix for the methods and classes of that library – the examples below will show you how to invoke some of the methods, but just remember that an import statement means that we load extra modules which extend the basic python functionality. Another example is the math library from which you can import math-related functions such as `cos`, `sin` and `exp`. It is done by invoking

```
from math import cos, sin, exp
```

If all goes well with the import statement of `pylab`, nothing should happen(!). To plot your lists of numbers (`xvalues` and `yvalues2`) we can use a built-in function of that library called `plot` which takes the x-coordinates and y-coordinates you want to plot. Invoke it as

```
pylab.plot(x,y)
```

and to show the result, use

```
pylab.show()
```

which tells us that matplotlib should show the result on the screen. This is useful if you want to

4 Indentation usually means hit [SPACE] twice and continue writing your code, that is at least how I do it – other people prefer using [TAB] but it is entirely up to you, just be consistent.

debug something and see intermediate results, but since it pauses execution of the program, it is not very useful for a final program. Instead you can save the image directly to disk by the following command

```
pylab.savefig('myfirstplot.png')
pylab.clf()
```

which saves the plot as a .png file. A final command that is useful is the `pylab.clf()` which clears the current plot for all its data – this is useful if you want to iterate over particle positions and store snapshots of those positions.[5]

Final Comments

Before the actual exercise, the recommended work flow will be presented here briefly.

The idea is that you use your favorite text editor[6] to edit python scripts. These scripts are then executed on the command line (or by pressing F5 on Windows). I also recommend that you keep the exercises from the different weeks separate by using folders, this will make it much easier to find files later on.

Linux (and Mac OS X)
(n) When you open a new terminal it always starts in your home folder where all your files are stored. To create a directory called molstat and enter it, use the following commands

```
mkdir molstat
cd molstat
```

The `mkdir` command creates a folder with the name `molstat` and the `cd` command enters the folder named molstat. Make a new directory called `ex1` and enter it as well. This will be the starting point for exercise 1. To create a new file, launch `kedit` from the programs menu and save the empty file as example.py in the `molstat/ex1` folder.

Windows
Make a new document in `IDLE`, save it as example.py in a place where you can find it later. Remember that on Windows, you just have to press F5 to run your code when it is open. In the exercises later on, when it says `kedit`, you can easily replace this with `IDLE`.

5 Which coincidentally is what we want to do today.
6 For most people, the `kedit` text editor on the fys.ku.dk servers is more than enough. For windows, the IDLE environment serves our purpose nicely.

Generally
Write the following code and understand it before you proceed:

```
x = range(-5,5)
print x
```

Once you've gotten the grasp of what you think this little program does, save the file and in the terminal you can execute the script[7] by writing

```
python example.py
```

(o) What happens? How is it different from executing stuff on the command line?

(p) Extend the example.py script to calculate $y=x^2$. When you have calculated the yvalues array, print it to verify it, plot the results and save the figure using `savefig` command.

From now on, everything we do will be done like this. You create a file in `kedit`, save it and execute it using python.

7 Remember that we are now in the regular shell, and *not* in python.

Getting into coding – non-interacting particles

Goal of today

- Initialize particles with random coordinates and random velocities.
- Propagate the particle positions in time.
- Confine the particles to be in a box.[8]

Implementation

The code below is your starting point for today's exercise.

```
# import pylab and a function to create random numbers
import random
import pylab

# initialize some variables
npart = 100
nsteps = 1
dt = 0.001

# create the x- and y-coordinates
X = [random.random() for I in range(npart)]
Y = [random.random() for I in range(npart)]

# plot the x- and y-coordinates in a figure.
pylab.plot(X,Y,'ro')
pylab.axis((-1,1,-1,1))
pylab.savefig('startcoord.png')
```

As you can see, a lot of variables have been declared such as `npart` which is the number of particles that we want to generate and simulate. `nsteps` is the number of steps we want to take in a simulation and `dt` controls how much of the velocity is scaled. Lastly we have defined the x-coordinates and y-coordinates for `npart` random particles using list-comprehensions.

(1) Inspect and understand the above code. You should be able to tell what the different parts of the program does before you continue. To be sure, try to print the values of the variables as well. You should also try to print the random() function, what happens when you run your script twice after each other?

The first programming task for today will be to correct the y-coordinates of the particle positions. On Figure 1a you can see what happens if you run the above code and what it is supposed to look like in Figure 1b, given that we would like the particles to start in the right hand side of the box.

(2) Correct the y-coordinates of the particle positions by making them initialize in the range $y \in [-1,1]$ instead of the standard $y \in [0,1]$.

When you have corrected the particle positions, it is time to give the particles *random* velocities to allow the particles to move around.

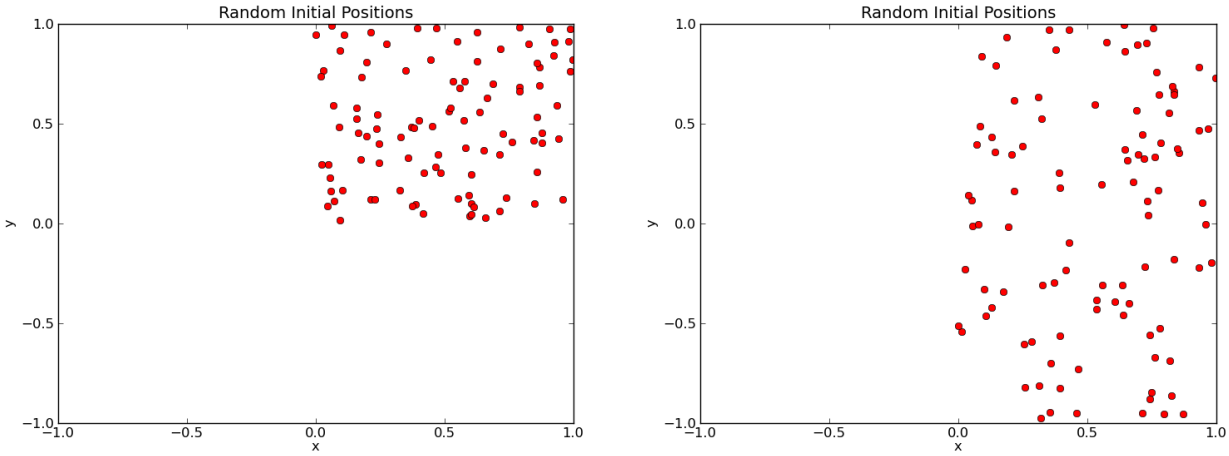
8 Although it is tempting to think of quantum mechanics when particles are confined to a box, it is, however, not the case here.

(3) Create two new lists named `Vx` and `Vy` which we shall use to store the velocities for the particles they should be random, as the positions were. The particle velocities should have an equal possibility to go in all directions.

You can plot the particle velocities by using the quiver function of the matplotlib library. It plots arrows which we normally use to represent vector quantities. You can use it by writing

```
pylab.quiver(X,Y,Vx,Vy)
```

before the `pylab.axis` command is issued.



a) Initial coordinates from code b) Correct coordinates to start with

Figure 1: Generating random coordinates in python can result in many different solutions. We want to generate the particles in the right side of a container which spans $x \in [0,1]$ and $y \in [-1,1]$.

We are now ready to loop over each particle in our system, but before we do this, you should make it really clear to yourself how we can obtain the coordinates and velocities for the i'th particle. *Hint: Check out exercise (j) again.*

The idea with this exercise is that since no forces act on the particles, only the velocity of the particle influences its position. Hence the location of the i'th particle $X_i^{(n+1)}$ at a time-step $n+1$ is given as

$$X_i^{(n+1)} = X_i^n + dt \cdot Vx_i^n$$

which tells us that the i'th particle will be at its previous position + its velocity at the previous position times a time step.

(4) Modify your script to loop over each particle and update the x- and y-coordinates with the respective particle velocities (`Vx` and `Vy`). Plot the particle positions, after you have changed them, to a file called stepcoords.png to verify that your particles indeed have moved.

It turns out, however, that simulations that only propagate time in one step are rather boring in the long run(!). We therefore wish to make *several* steps to view the behavior of the particles over a

long time-span before the simulation ends.
(5) Modify your code to repeat the displacement of the particles `nstep` times, and in each step, update the positions of the particles as you did in exercise (5). Where are the particles after 10 steps? 100 steps? 1000 steps? 10000 steps? Make a plot of the region {-1,1,-1,1} as well as {-10,10,-10,10} to help answer the question.

What you simulate is how particles in vacuum would behave if they cannot feel each other and have kinetic energy. What remains is to keep the particles inside a box, i.e. they should make an elastic reflection on the walls and change direction. To change the direction on the i'th particle, we must change the sign on the velocity for that particle. For simplicity, we shall add a box which corresponds to the region we are plotting, that is $x \in [-1,1]$ and $y \in [-1,1]$.

For simplicity we start out with the x-coordinates first and then, when we have confirmed that it is working, we move onto the y-coordinates.

(6) Modify your code such that after you updated a particles position, you test if the particle is outside the boundary of the box. If it is outside the box, then change the sign on the velocity of that particle. *Hint: You will need if-statements to solve this problem. Also, it is a good idea to draw your plan for what happens with a pen and paper(!).*

There is one unsatisfactory thing about the solution you just made. Can you guess what it is? Since we are not doing quantum mechanics on particles in a well, the particles should never be allowed to be outside the box in a simulation step and the above solution permits that.

(7) Make the appropriate changes such that no particles are left outside the box when we have modified all the particle positions in a single step. When you are done, plot the final particle positions after 10000 steps and comment on the result.

Congratulations. You've completed your first programming task (ever?). Learning programming is all about trying and failing, and then trying some more.

(8) Until next week, try and start over with the exercise from today (assignment 1 to 7) and program it again.