Pràctiques en Empresa
QuantumLabUB

Arnau Jurado Romero
Supervised by:
Bruno Juliá Díaz
Montserrat Guilleumas Morell

Departament de Física Quàntica i Astrofísica
Universitat de Barcelona
Grau de Física

Contents

1 Introduction

This report summarizes my work done from February 2019 to December 2019 at the Departament de Física Quàntica i Astrofísica of the Universitat de Barcelona under the supervision of Bruno Juliá Diaz and Montserrat Guilleumas Morell in collaboration with the QuantumLabUB project. The work included development of python applications with graphical interfaces and participation in weekly meetings as well as participation in the 'Festa de la Ciència'. A total of 253 hours have been dedicated to this project.

1.1 The QuantumLabUB Divulgation Project

For the last few years, Bruno Juliá and Muntsa Guilleurmas with the collaboration of Artur Polls have led undergraduate physics students from the Universitat de Barcelona in a divulgation project of quantum mechanics.
The project consists in the development of python codes to help illustrate some particular effects of quantum mechanics and build intuition on the matter. Although the main aim of the project is to showcase quantum mechanics, some classical mechanics programs have been developed to compare the phenomena.
It is remarkable that all of the programs solve the real physics equation and simulate the physical systems.

My initial task in this project was the development of a 2 dimensional classical simulator for non interacting particles to compare different situations with the 2D quantum simulator developed in parallel by Marina Orta and with previous programs (like Doubleslit by Daniel Allepuz).

2 Development Process

2.1 Weekly Meetings

Weekly meetings of one hour where a very important part of the work. In them Bruno Juliá and Muntsa Guilleurmas as supervisors and Marina Orta, Manu Canals and me met to share ideas and help each of our individuals projects advance. At the beginning these meetings helped decide the programs and their main features. Later these meetings helped with solving technical problems that arose during development. Overall they were a great tool to overcome obstacles through teamwork.

2.2 Main objectives

Among the different physical systems developed previously at QuantumLabUB were very restricted 1D classical simulations, 1D quantum simulations with lots of freedom and a very restricted 2D quantum simulation (double slit experiment). The next step was making a 2D quantum and classical simulator with freedom to choose the 'landscape' (potential field). It was decided that Marina Orta would develop the quantum version and I would develop the classical version.

From the start the main vision of my program was to allow the user the simulate multiple non interacting particles in a potential field. At first I wanted to make the potential be drawable on the screen, this proved to present lots of problems and ended with two potential fields with lots of parameters (Gauss and Woods-Saxon).

After all the features that we had planned were added to the the program the development of a second program was started. Based on the same interface, this program would include interactions between the particles to show phenomena related to interacting systems (like how they reach thermal equilibrium). This second program was not able to be finished before the internship ended, but the computational method and most interface features were finished so continued work could provide a very exhaustive program for divulgation of thermodynamics.

2.2.1 Other tasks

Shortly after finishing the development of the first program we were invited to present an exhibit at the 'Festa de la Ciència' at Barcelona in which we would present programs from QuantumLabUB. One disadvantage of python programs is that is not easy to run the code in other machines, a lot of setup has to be done before running the code. I explored the possibility of packaging the QuantumLabUB programs into an executable that did not need python installed to run. Using the `PyInstaller` tool I packaged each program into executable files that made the setup at the 'Festa de la Ciència' much easier. Each program needed its own tweaks and it was a time consuming process so I wrote some documentation explaining how to do it for future programs.

I also attended the 'Festa de la Ciència' exhibit in which I assisted in explaining the attendants to the exhibit about the physics involved in the programs and divulgating quantum mechanics. It was a very pleasing experience.

3 Two dimensional classical non-interacting particles simulator

3.1 Equations

The system under consideration is a particle under the influence of a scalar potential that is a function only of the position of the particle in the two dimensional plane. The lagrangian for this particle is:

$$L = \frac{1}{2}m\dot{x}^2 + \frac{1}{2}m\dot{y}^2 + \Phi(x, y) \tag{1}$$

Thus the movement of the particle can be solved through the Euler-Lagrange equations:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}}\right) - \frac{\partial L}{\partial q} = 0 \tag{2}$$

$$\ddot{x} = -\frac{1}{m}\frac{\partial \Phi(x, y)}{\partial x} \tag{3}$$

$$\ddot{y} = -\frac{1}{m}\frac{\partial \Phi(x, y)}{\partial y} \tag{4}$$

3.2 Numerical resolution

First we translate the set of equations found to a form such as:

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(\mathbf{y}; t) \tag{5}$$

Where **y** and **f**(**y**;t) are vectors for the dependent variables and their respective derivatives. In our problem we have 4 dependent variables: x, y, $\dot{x}$ and $\dot{y}$. And given the initial conditions, $\mathbf{y}_0$, their trajectory is determined by equations (3) and (4):

$$\frac{dx}{dt} = \dot{x} \tag{6}$$

$$\frac{dy}{dt} = \dot{y}$$

$$\frac{d\dot{x}}{dt} = \ddot{x} = -\frac{1}{m}\frac{\partial \Phi(x, y)}{\partial x}$$

$$\frac{d\dot{y}}{dt} = \ddot{y} = -\frac{1}{m}\frac{\partial\Phi(x,y)}{\partial y}$$

We can now solve this system of equations using any numerical method we prefer. Because the potential field, $\Phi(x, y)$, could vary in complexity from one point of the plane to another, I have decided that the Runge-Kutta-Fehlberg method is best suited to this problem due to its adaptive time step.

3.2.1 Runge-Kutta-Fehlberg method

The Runge-Kutta Felbherg (RKF) method uses the results from two Runge-Kutta methods with order n and $n+1$ and estimates the error of the computation by taking the difference between the results. I will be using the RKF method with order 4 and 5 (also known as RKF45) which optimizes the coefficients so only one extra calculation has to be made to estimate the error[1] :

$$k_0 = \mathbf{f}\left(t_0, \mathbf{y}_0\right) \tag{7}$$

$$k_1 = \mathbf{f}\left(t_0 + \frac{h}{4}, \mathbf{y}_0 + \frac{h}{4}k_0\right)$$

$$k_2 = \mathbf{f}\left(t_0 + \frac{3h}{8}, \mathbf{y}_0 + \frac{3h}{32}k_0 + \frac{9h}{32}k_1\right)$$

$$k_3 = \mathbf{f}\left(t_0 + \frac{12h}{13}, \mathbf{y}_0 + \frac{1932h}{2197}k_0 - \frac{7200h}{2197}k_1 + \frac{7296h}{2197}k_2\right)$$

$$k_4 = \mathbf{f}\left(t_0 + h, \mathbf{y}_0 + \frac{439h}{216}k_0 - 8hk_1 + \frac{3680h}{513}k_2 - \frac{845h}{4104}k_3\right)$$

$$k_5 = \mathbf{f}\left(t_0 + \frac{h}{2}, \mathbf{y}_0 - \frac{8h}{27}k_0 + 2hk_1 - \frac{3544h}{2565}k_2 + \frac{1859h}{4104}k_3 - \frac{11h}{40}k_4\right)$$

$$\mathbf{y} = \mathbf{y}_0 + h\left(\frac{25}{216}k_0 + \frac{1408}{2565}k_2 + \frac{2197}{4104}k_3 - \frac{1}{5}k_4\right)$$

$$\hat{\mathbf{y}} = \mathbf{y}_0 + h\left(\frac{16}{35}k_0 + \frac{6656}{12825}k_2 + \frac{28561}{56430}k_3 - \frac{9}{50}k_4 + \frac{2}{55}k_5\right)$$

$$\hat{\mathbf{y}} - \mathbf{y} = h\left(\frac{1}{360}k_0 - \frac{128}{4275}k_2 - \frac{2197}{75240}k_3 + \frac{1}{50}k_4 + \frac{2}{55}k_5\right)$$

Where h is the current step and $\hat{\mathbf{y}}$ and $\mathbf{y}$ are the values of the dependent variables after a step in the independent variable (t) of h for the fifth and fourth order methods respectively. A new estimation for the step size can be determined following:

$$h_{new} = h\left(\frac{h\epsilon}{|\mathbf{y} - \hat{\mathbf{y}}|}\right)^{\frac{1}{n}} \tag{8}$$

Where n is the lowest order of our methods, in this case, $n = 4$ and ϵ is the error we desire in our new values. After computing the new step size we should check if it is smaller than the previous one, if it is then we compute again all the values of (7), reevaluate (8) and check again.

It is worth noting that it is not necessary to evaluate $\mathbf{y}$ nor $\hat{\mathbf{y}}$ to update the step size, the difference between the two is enough. Once the appropriate step size for the desired precision is determined we can compute $\hat{\mathbf{y}}$, which will be the next value for the dependent variables.

3.3 Implementation

A particle with mass m (chosen by the user) and will be moving in a LxL box with a potential field defined by the user from different pre-made potential functions. The program will compute the trajectory of the particle until a time T has past since the initial instant ($t = 0$). For simplicity I have choose the box to be 200 arbitrary units (so, for example $x = +50$ is halfway to the right from the center to the edge horizontally, $y = -33$ is a third of the way downwards from the center to the edge vertically, etc.).
The physical units will be determined from the units chosen for m, L and T:

$$[p] = [m]\tfrac{[L]}{[T]} \quad [E] = [m]\tfrac{[L]^2}{[T]^2}$$

So for the SI ($[m] = kg$, $[L] = m$, $[T] = s$) the energy would be in Joules and the box would be 200 meters by 200 meters while in c.g.s. system ($[m] = g$, $[L] = cm$, $[T] = s$) it would be erg and the box would be 2 meters by 2 meters.

Like previously discussed, the numerical resolution of the trajectory will be performed by the RKF45 method. While RKF45 allows for very large step sizes I have limited the biggest step size possible to make the later representation more intuitive and fluid. The (current) relevant parameters are presented next:

$$L = 200 \quad T = 60$$
$$h_{max} = 0.1[T] \quad n_{max} = 100 \quad \epsilon = 0.01 = 10^{-2}$$

n_{max} is a parameter that limit the number of iterations that RKF45 does before fixing the final step size, it ensures that the program does not try to make the step size infinitely small in zones with discontinuities in the potential. This parameter can prevent the algorithm from achieving the desired precision, ϵ, but if the potentials and their derivatives are continuous (which in our case, they will be) this should not be a problem.
Also, because the potential field will not be changing with time, I can ignore all the time dependence on the set of equations (7).

3.3.1 Potentials

For defining the potential field, $\Phi(x, y)$, the users has a series of defined potentials with some parameters. The sum of the potentials at every point will make up the potential field.
For stability of the RK45 method, the functions representing these potentials should be continuous and derivable up to the first derivative.
Following is a list of the implemented potentials with their parameters and special considerations, if any:

1. **Woods-Saxon potential**:

   - Formula:

   $$\Phi(x,y) = -V_0 \frac{1}{1+e^{\frac{|x'|-R_x}{a}}} \frac{1}{1+e^{\frac{|y'|-R_y}{a}}} \quad \begin{array}{l} x' = (x-x_0)\cos(\theta) - (y-y_0)\sin(\theta) \\ y' = (x-x_0)\sin(\theta) + (y-y_0)\cos(\theta) \end{array}$$

   - Parameters:
     - x_0: center x position
     - y_0: center y position
     - V_0: depth (if positive) or height (if negative) of the potential well/mountain
     - R_x and R_y: length and width of the potential well/mountain, for $x > R_x$ or $y > R_y$ the potential rises/drops rapidly to 0
     - a: slope of the transition zone around $r = R$. For stability, $0.1 < a < 1$. It is fixed at $a = 1$.
     - θ: sets the angle of rotation for the rectangle.
   - Considerations: has an exception for $r = 0$ for the derivative. The derivative tends to 0 on $r \to 0$.

2. **Gaussian Potential**:

   - Formula:

   $$\Phi(x,y) = V_0 e^{-\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2}}$$

   - Parameters:
     - x_0: center x position
     - y_0: center y position
     - V_0: depth (if positive) or height (if negative) of the potential well/mountain
     - σ: controls the size of the potential well/mountain. If this potential was a normalized gaussian distribution then about 68.2% of the values represented by the distribution would be located from 0 to σ. However, this has little to no meaning in our problem outside how wide the potential is.
   - Considerations: none

3.3.2 Testing

A series of different tests were done on the method to see its stability and accuracy. Theoretically, the method should be accurate in all of the dependent variables up to ϵ precision.
To analyze the results, 7 different plots were used:

- A color map of the potential field.
- The kinetic, potential and mechanical energy of the particle vs. time.
- The trajectory of the particle without time (x vs. y).
- The trajectory of the particle for x (x vs. t).
- The trajectory of the particle for y (y vs. t)
- The phase space of the particle for x ($\dot{x}$ vs. x)
- The phase space of the particle for y ($\dot{y}$ vs. y)

Because the lagrangian (1) of the particle is independent of time then on all tests the mechanical energy should be conserved (up to what ϵ allows).

Also, for initial conditions and potential fields where we expect the particle to oscillate or to make a periodic motion on either the x or y coordinates, a closed line should be seen on the phase space plots.

Using this plots it was determined that the method worked well in simulating the trajectories of the particles. Energy was conserved, particles oscillated in harmonic oscillators with the appropriate frequency, etc. These plots also helped in diagnosing the an error in the implementations of the RKF45 method.

3.3.3 Error in the implementation of RKF45

Although multiple errors and bugs were found and fixed during the development of the numerical method, this one stands out from the rest because it was difficult to detect and it was probably present from the very beginning of development.
Due to an error in the code the algorithm performed the adaptation of the step properly (and the trajectories were, in fact, correct) but then registered as if the step was always the same (the maximum step, 0.1).

Because all of the particles were computed independently (and with their own steps) the steps that each particle took were different but for the algorithm (and the subsequent interpolation) that was not the case. It is notable that if a particle did not need to adapt the step (because with 0.01 was enough) then this bug did not manifest. This consistently happened with gaussian potentials, however Woods-Saxon potentials were more demanding and forced the particles to adapt the step so we first detected the effects of this bug with WS potentials.

When WS potentials were implemented the particles got 'stuck' for a long time when contacting the walls of the WS. At first we thought this was normal due to the huge inclination of theses potentials that slowed down the particles so much that maybe this was the result. Of course, before seeing this we thought that the particles would bounce in a short time so this behavior seemed weird. After the implementation of the cone particle addition mode we saw that sending a 'wave front' of particles to a WS wall caused the front to distort weirdly, the expected behavior was for the front to reflect and maintain its spherical (or rather, circular) shape. This last behavior was the one that led to the investigation of the code and finding the error.
After finding the error the strange behavior was easily explained, because the particles were moving as if a lesser step equaled a 0.01 step then the particles got 'slowed down' when they needed to adapt the step size, this explained the first behavior. The second behavior occurred because different particles took different step sizes (because they encountered different potentials along their respective trajectory), and so some particles were, at some point, 'slowed down' more or less respect to others, which caused the distortion in the wave fronts.

3.4 Program

Because the final objective was to add as many particles as the user wanted, the process of defining and computing the particle and its trajectory had to be generalized. That is why I decided to create the `particle` python class that allowed me to define a completely generic particle so adding more particles to the simulation would not be a problem. In a similar manner, the `Phi` class allows multiple potentials with an analytic expression to be superposed, although different potential configurations were not necessary in this case.

3.4.1 `Phi` class

`Phi` is a python object that stores the analytic expressions of the potentials and its partial derivatives and the means to compute the values at any point in a 2D plane. The potentials have to be set as python functions with the following arguments: `(r,param)`. The first argument is a `numpy.array` or list of two elements that represent the coordinates of the point to be evaluated. The second argument is a list of parameters of the potential (like, for example, the potential height or slope). The `param` list has to be common for all three of the functions, even if some parameters are not used. After setting the three separate functions the following methods can be used:

- `add_function(fun,dfunx,dfuny,param)`: with this method the new potentials are added to the list of potentials stored by the object. The method `clear()` can also be used to delete all the added potentials.
- `val(x,y)`, `dvalx(x,y)` and `dvaly(x,y)`: evaluates and returns the potential, its x partial derivative or its y partial derivative, respectively, at a point (x,y) in the plane. Note that this methods can accept and return `numpy.array` objects if the functions that define them are compatible with `numpy`.

3.4.2 `Particle` class

`Particle` is a python object contains all the information regarding the particle and its trajectory: mass, charge, positions, velocities, energy, etc. It also contains the RKF45 method programmed. For the computation the particle class requires a potential field set with `Phi`. The main methods of the `Particle` class are:

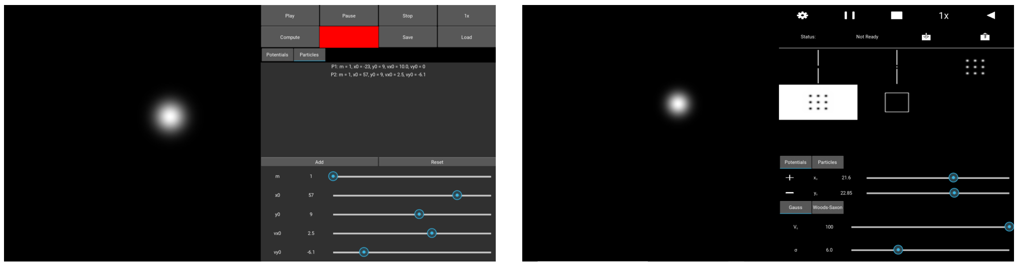
- `ComputeTrajectoryF(r0,T,pot)`: this method updates the `trajectory` array using the RKF45 method, which is a (4,steps.size) shaped array containing x, y, v_x and v_y in that order. The initial conditions are specified by r0, which is a list or `numpy.array` of shape 4. T specifies when to stop the computation and pot the potential field (which is a `Phi` object). This method also registers the steps taken in the `particle.steps` array and creates an interpolation of the trajectory to be used in the animation.

- `KEnergy()`, `PEnergy()` and `Energy()`: returns the kinetic, potential and total energy at the same times as `particle.steps`, this is useful for checking conservation of energy.

After performing a succesful computation, the interpolated trajectory of the particle is available and easy to acces for the animation.

3.4.3 User Interface

I decided to use the open source python library Kivy for the UI. This decision was made because of its simplistic design, the ability to port the interface to Android and touchscreen devices and the fact that some code in the QuantumLabUB repository used Kivy so a it could be used as a reference.



```
P1: m = 1, x0 = -23, y0 = 9, vx0 = 10.0, vy0 = 0
P2: m = 1, x0 = 57, y0 = 9, vx0 = 2.5, vy0 = -6.1
```

m	1
x0	57
y0	9
vx0	2.5
vy0	-6.1

x_0	21.6
y_0	22.85
V_0	100
σ	6.0

Figure 1: Comparison between a first version of the interface (left) and the final version (right).

Once the `Particle` object was implemented, a way to tell what to compute and to show it was necessary, that is the objective of the UI. I decided to divide the screen of the UI in two areas: the left area would show the potentials and the trajectories of the particles and the right side would show all the interactive buttons to communicate with the program.

A summary of the development process of the UI in chronological order is presented next (from February 2019 to April 2019 all work was dedicated to developing the numerical method):

1. (05/2019) At first the screen was divided in two halves. Kivy works by adding building blocks named widgets which range from labels and buttons to file browsers and drop-down menus, so at first I simply divided in two `BoxLayout` which allows to put more widgets in an orderly manner. The left half was left empty and the right half was filled with buttons that I thought were needed (play, compute, pause, speed of the animation, a zone to add and reset potentials and particles, etc.). All of the buttons would not do anything when pressed, they needed to be 'wired'.

2. For adding the particles and potentials I decided to use a tabbed panel, which is a widget already implemented in the Kivy library. I started by adding the necessary elements to add a Gaussian potential and a single particle (you could, at this moment, add multiple particles but only one by one). A list of all the added potentials and particles was also added but this feature is not present in the final program. The parameters were set by using sliders.

3. At this moment the animation was ready to be shown. At first I decided to use the implementation of `MatPlotLib` in Kivy but ended using the drawing tools of Kivy itself for the animation. This posed a problem immediately: most computer screen have a 16:9 aspect ration which meant that the halves were not square but rather rectangular while the space of the physical simulation was 200 by 200 units. This was solved by forcing the left half to be square.

4. After the implementation of the animation more features were added: a speed modifier for the reproduction of the animation, an status label that informed if a computation is needed and clicking a point on the screen modified the x_0 and y_0 (the central positions for either a potential or a particle) so the slider were no longer needed to set these parameters.

5. A save/load feature was added. At first it simply saved the configuration of potentials and particles in a `save.dat` file and loaded the same

file. Although not useful for using multiple saves, it sped up the testing process for the next potential to be added.

6. (06/2019) After lots of trial and errors with lots of functions, the Woods-Saxon form was found to be useful for making rectangles in two dimensions. The function itself needed a lot of fine tuning to make it work without overflowing or causing problems in the computation (especially its derivatives). However, the Woods-Saxon potential helped in finding an important error in the implementation of the RKF45 method (see section 3.3.3)

7. Once the Woods-Saxon potential was integrated into the UI I started adding additional modes for adding particle. The spread mode for adding multiple particles in a cone was the first to be added. At this point I also started working on the preview of the added particles, for a single particle it would show in blue where the particle would be added with the current selection of parameters, the velocity was also represented with a line. Once the particle was added it would stay in place in orange until the animation started.

8. (07/2019) The compute and play button were joined, now it was impossible to press play without first computing (which was causing crashes and problems). Also started the implementation of a rotation angle for the WS potential which made the analytic function far more complex.

9. I added the line particle addition mode which added multiple particles with parallel velocities. Also added a preview for this mode. Started working in particles addition mode that could be used to compare with quantum phenomena, after some discussion at the meetings we reached the conclusion that sampling the free-particle eigenstate could fulfill this role (in conjunction with a double slit demo). This mode was also added with a preview.

10. Once all the previews were ready I removed the list of particles that was present from the start, as I thought it provided redundant information. This space was later used to add the demos

11. A this point the major error in the RKF45 was fixed. This fixed the behavior with the WS potential and the numerical resolution didn't seem to have any more errors.

12. The time inversion button was added. This button re-added all the particles with their velocities at the time of the press reversed. After this, the computation had to be re-done. This served both as a check of the numerical method as a tool to show the time symmetry, which sometimes produced interesting effects like the ordination of seemingly randomly distributed particles.

13. (08/2019) Demos of interesting potential landscapes were added for quick usage, this demos were meant to be loaded and then add particles. The demos added were: single slit and double slit (for comparison with the quantum equivalent); lattice and alternative lattice (which featured 9 positive/negative Gauss potentials distributed in a square lattice); and square (which featured a closed surface with WS potentials).

14. Finally, all of the buttons of the UI were given icons to make the aesthetic of the program more appealing and easier to read.

From September 2019 until December 2019 all work was dedicated towards the interacting particles simulator.

4 Two dimensional classical interacting particles simulator

4.1 Equations and interaction

The system is now N particles that interact with each other. The lagrangian of the system now has to account for all the particles:

$$L = \sum_{i=1}^{N} \frac{1}{2} m_i |\dot{\vec{r}_i}|^2 - \sum_{i<j}^{N} V(\vec{r}_i, \vec{r}_j) \tag{9}$$

Where $V(r_i, r_j)$ is some potential that describes the interaction between the particles. This again can be solved trough the Euler-Lagrange equations:

$$\ddot{\vec{r}_i} = -\frac{1}{m} \sum_{j \neq i}^{N} \vec{f}_{i,j} = -\frac{1}{m} \vec{f}_i \quad with \quad \vec{f}_{ij} \equiv \vec{\nabla}_i \cdot V(\vec{r}_i, \vec{r}_j) \tag{10}$$

Where $\vec{f}_i$ are all of the forces applied to the i-th particle by the rest.

We will consider that all the particles have the same mass and that the interaction potential is the so-called Lennard-Jones potential:

$$V(r_i, r_j) = 4\epsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right] \quad r_{ij} \equiv |\vec{r}_i - \vec{r}_j| \tag{11}$$

Which has a strong repulsion at distances shorter than σ (hard sphere) and an attractive well of depth ϵ. For simplicity, we will work in *reduced units* which consists in taking ϵ for the unit of energy, σ for the unit of length and the mass of the particles m as the unit of mass. All the other units are derived from these. The values of these units have been taken from experimental values of Argon gas:

$$\epsilon = 0.0103\ eV \quad \sigma = 3.405\ \text{Å} \quad m = 6.6323 \cdot 10^{-26}\ kg$$
$$t^* = \sigma\sqrt{m/\epsilon} = 0.109\ fs \quad T^* = \epsilon/k_B = 119.8\ K$$

Where an asterisk has been used to denote reduced units and k_B is the Boltzmann constant.

Usage of reduced units is very useful from a theoretical perspective. Two different systems at the same density and temperature (or same pressure and temperature, etc.) in reduced units present the same properties. This is the law of corresponding states and is a consequence of the universality of thermodynamic systems. This means that a simulation can be translated to any (hydrsotatic) system by using the corresponding ϵ, σ and m.
It is also useful from a practical standpoint: numerical values in reduced units are more manageable than in S.I units (which are usually much larger or much smaller than 1). This helps prevention of overflows in computation.

So, in reduced units the Lennard-Jones potential acquires the form:

$$V(r_i, r_j) = 4\left[\left(\frac{1}{r^*_{ij}}\right)^{12} - \left(\frac{1}{r^*_{ij}}\right)^{6}\right] \tag{12}$$

The spatial derivatives of this potential will determine the force applied to a particle i by another j:

$$\vec{f}_{i,j} = \frac{48}{r^2_{ij}}\vec{r}_{ij}\left[\left(\frac{1}{r^*_{ij}}\right)^{12} - \frac{1}{2}\left(\frac{1}{r^*_{ij}}\right)^{6}\right] \quad with \quad \vec{r}_{ij} = \begin{pmatrix} x_i - x_j \\ y_i - y_j \end{pmatrix} \tag{13}$$

In addition to the interaction between the particles, the system will be enclosed in impenetrable walls. The usual way in computational physics to simulate such a wall is to 'flip' the normal component of the velocity to that wall when a particle gets close enough. This simulates an elastic collision between the wall and the particle. Because the algorithm that will be used does not use velocities we cannot use this approach. Instead we will place Woods-Saxon potentials at the edges of the box which have proved to be a good reproduction of walls in the simulations of non-interacting particles.

4.2 Numerical resolution

A popular algorithm in molecular dynamics is the Verlet algorithm, this algorithm uses only positions to integrate the equations of motion. We start by expanding by Taylor the position (which I will label r but remember that it has two components) of a particle at a time $t + \Delta t$ around t:

$$r(t+\Delta t) = r(t) + \dot{r}(t)(t+\Delta t - t) + \frac{1}{2}\ddot{r}(t)(t+\Delta t-t)^2 + \frac{1}{6}\dddot{r}(t)(t+\Delta t-t)^3 + \theta(\Delta t^4) \tag{14}$$

By equation (10) $\ddot{r}$ is exactly $-\frac{1}{m}f(r)$ (where I have dropped the sub-index i in favor of r to symbolize the specific particle). So (14) becomes:

$$r(t+\Delta t) = r(t) + \dot{r}(t)\Delta t - \frac{f(r(t))}{2m}\Delta t^2 + \frac{1}{6}\dddot{r}(t)\Delta t^3 + \theta(\Delta t^4) \tag{15}$$

Similarly we can expand the position at a time $t - \Delta t$ around t:

$$r(t-\Delta t) = r(t) - \dot{r}(t)\Delta t - \frac{f(r(t))}{2m}\Delta t^2 - \frac{1}{6}\dddot{r}(t)\Delta t^3 + \theta(\Delta t^4) \tag{16}$$

And summing the two expansions we obtain:

$$r(t+\Delta t) + r(t-\Delta t) = 2r(t) - \frac{f(r(t))}{m}\Delta t^2 + \theta(\Delta t^4) \tag{17}$$

So, approximately we obtain that the position at $r(t + \Delta t)$ is:

$$r(t+\Delta t) \approx 2r(t) - r(t-\Delta t) - \frac{f(r(t))}{m}\Delta t^2 \tag{18}$$

So knowing the position of the of a particle at a certain time we can know its next position with its previous position and the forces acting on the particle at that time. If we wish to compute the velocities we can take the difference of (16) and (17):

$$v(t) \approx \frac{r(t+\Delta t) - r(t-\Delta t)}{2\Delta t} \tag{19}$$

If we know the initial conditions of the particles we need the previous positions to compute the next step and iterate further. We can generate an approximate first position by using a different algorithm (say, an RK4 step) or we can simply generate a previous position (instead of the next) by computing:

$$r(t_0 - \Delta t) = r(t_0) - v(t_0)\Delta t \tag{20}$$

Which is, in fact, an Euler method step performed backwards.

4.3 Force computation

To evaluate (18) it is needed to compute the sum of all forces (13) which requires the distances of all the particles with each other. Making this process efficient requires the usage of the Python library `NumPy` to the fullest. `NumPy` is based on the `array` object which behaves like vectors or matrices and allows python to perform computations between these vectors and matrices, which is not possible with lists, the common form of python vectors. Let d^x and d^y be the NxN matrices of the differences of the x and y coordinates such that:

$$d^\alpha_{ij} = \alpha_i - \alpha_j \quad with \quad i,j \in [0,N] \quad and \quad \alpha = \{x,y\} \tag{21}$$

Both matrices are antisimmetrical and their diagonal is 0, but we will not take advantage of this information, as it would make the program more difficult to implement. To compute these matrices we can use the `numpy.meshgrid` function which takes two uniaxial arrays (vectors) returns two arrays with each vector 'extended' along the other vectors' dimension. This function has many uses (it isn't even restricted to vectors) but we will focus on the result when the two vectors are the same one. Suppose that we have the vectors with the α coordinate of every particle:

$$\mathcal{A} = \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \cdots \\ \alpha_N \end{pmatrix} \tag{22}$$

Then if we apply `meshgrid(`$\mathcal{A},\mathcal{A}$`)` we obtain two NxN arrays of the form:

$$M_\alpha = \begin{pmatrix} \alpha_1 & \alpha_2 & \cdots & \alpha_N \\ \alpha_1 & \alpha_2 & \cdots & \alpha_N \\ \cdots & \cdots & \cdots & \cdots \\ \alpha_1 & \alpha_2 & \cdots & \alpha_N \end{pmatrix} \quad M_\alpha^T = \begin{pmatrix} \alpha_1 & \alpha_1 & \cdots & \alpha_1 \\ \alpha_2 & \alpha_2 & \cdots & \alpha_2 \\ \cdots & \cdots & \cdots & \cdots \\ \alpha_N & \alpha_N & \cdots & \alpha_N \end{pmatrix} \tag{23}$$

From here we can obtain d^α by simply taking the difference:

$$d^\alpha = \begin{pmatrix} \alpha_1 - \alpha_1 & \alpha_1 - \alpha_2 & \cdots & \alpha_1 - \alpha_N \\ \alpha_2 - \alpha_1 & \alpha_2 - \alpha_2 & \cdots & \alpha_2 - \alpha_N \\ \cdots & \cdots & \cdots & \cdots \\ \alpha_N - \alpha_1 & \alpha_N - \alpha_2 & \cdots & \alpha_N - \alpha_N \end{pmatrix} \tag{24}$$

This array follows the definition (21) so if we take $d^\alpha[i,:]$ we have the vector of distances of the i-th particle with the rest, so we can compute (13) easily by also computing $R^2 = (d^x)^2 + (d^y)^2$ which is the matrix of square distances.

4.4 Temperatures and Maxwell-Boltzmann distributions

Seeing particles collide and attract each other can be interesting on its own, but showing how a system reaches thermal equilibrium is is also an interesting demonstration. The temperature of our system (and any system similar to ours) is given in reduced units by:

$$T = \frac{1}{N_{df}} \sum_{i=1}^{N} V_i^2 \tag{25}$$

Which is the mean kinetic energy of the system in reduced units. N_{df} is the number of degrees of freedom of out system, which is: $N_{df} = 2N - 2$ because we are in 2 dimensions and we have fixed the velocity of the center of mass.

This temperature can be computed even when the system is not in thermal equilibrium. It is known from statistical mechanics that once a system like this reaches thermal equilibrium the magnitude of the velocities of the particles will follow a Maxwell-Boltzmann distribution. In 2D and in reduced units the Maxwell-Boltzmann distribution curve has the following expression:

$$f(V,T) = \frac{V}{T} e^{-\frac{V^2}{2T}} \tag{26}$$

So if we plot an histogram of the magnitude of the velocities it should have the same shape as this distribution. The more particles we have in our system (i.e. the more samples we have) the better we will be able to represent the distribution.

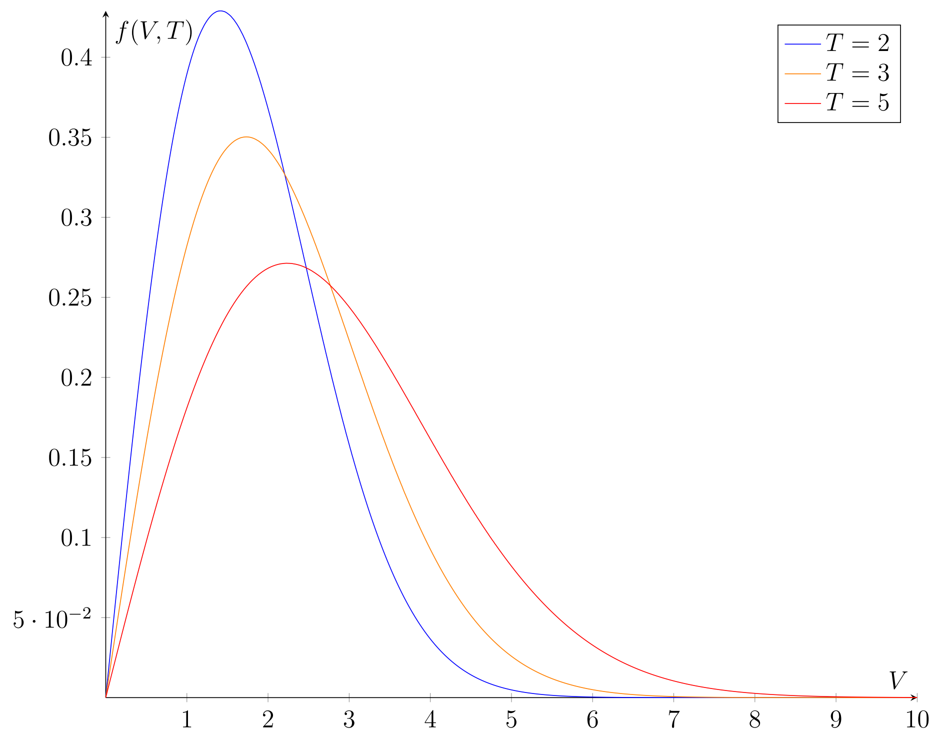


Figure 2: Examples of Maxwell-Boltzmann distributions at different temperatures

The only problem is that computations with a high number of particles take lots of time. So another strategy is to bring the system to equilibrium and start sampling the velocities of the system after a certain number of steps (ideally after some collisions so the system 'forgets' the previous configuration). This way after some time has passed we have a collection of sample significant enough to make the histogram.

4.5 Initialization of the system

Before applying the Verlet algorithm to evolve the system we first need to assign initial positions and velocities to all of our N particles. The initial configuration is irrelevant once the system has reached equilibrium (or at least it is for the properties we want to study) but we need to take into account that:

1. There should not be any overlap between two particles at the initial configuration.

2. Some initial configurations do not evolve towards thermal equilibrium.

The first consideration is easily avoided by setting the initial positions in a square lattice. The second consideration can be tricky. If we set all the initial velocities in a square lattice to, for example, towards positive x, all the particles will start performing one-dimensional periodic movements if the interaction is very weak at long distances (and the density is low enough, both of which happen in our case).

In general, if there is any kind of symmetry in the initial configuration the system will take more time to reach thermal equilibrium (or not reach it at all like in the previous example). One way to solve this is to assign a random velocity to each particle. There are two main ways to do this: we can assign a random x and y component or we can assign a fixed magnitude and a random direction. Keeping in mind that the main objective is to make an histogram of the magnitude of the velocities and see how it approaches the Maxwell-Boltzmann distribution it is more interesting the use the second approach because the initial distribution of a random uniform-uniform for both the x and y components is very similar to a MB distribution. But if we fix the magnitude of the velocity the histogram would show a spike at the chosen velocity, which is very different from a MB distribution.

To choose this velocity we will set our system at a chosen temperature and rescale all the kinetic energies to match this temperature. If we take (25) with $N_{df} = 2N$, because we the center of mass velocity is not fixed yet, we can see that the scale factor for a a given velocity is:

$$T = \frac{1}{2N} N V^2 = \frac{1}{2} V^2 \rightarrow \frac{T_{set}}{T} = \frac{V_{set}^2}{V^2} \rightarrow V_{set} = V \sqrt{2 \frac{T_{set}}{V^2}} \qquad (27)$$

Where T_{set} denotes our chosen temperature (which in general will be high $T = 3$ or $\sim 360K$). it is not important what the value of V is.

The last operation we will perform on the initial state of our system is set the center of mass velocity to 0. This will ensure that the system does not drift out of the box (although we have enclosed the box on walls, which makes this not as necessary). Because all the particles have the same mass the center of mass velocity is simply the mean velocity of the system. Because we were initially working in polar coordinates we have to transform to Cartesian coordinates first and then compute:

$$V^{\alpha}_{CM} = \frac{1}{N}\sum_{i=1}^{N} V^{\alpha} \tag{28}$$

And then subtract this velocity from every particle. In practice this operation will be done at the same time as operation as (27) like so:

$$\vec{V}_{set} = (\vec{V} - \vec{V}_{CM})\sqrt{2\frac{T_{set}}{V^2}} \tag{29}$$

Note that this last operation would not have any effect for $N \to \infty$ because the center of mass velocity would be 0 due to the fact that there is not a privileged direction in the plane. But when dealing with a finite number of particles it is necessary to consider it. This operation will make the final magnitude of the velocity vectors of the particles different from each other (more or less, depending on the 'luck' we have when generating the numbers).

4.6 Program

The two dimensional simulator with interactions was developed after the non-interacting, which allowed me to use the knowledge I learned while developing it to design it in a more streamlined manner. The base of the interface is the same as in the previous program with slight modifications.

4.6.1 `particle` class

The particle class for the interacting program was designed differently from the non-interacting one. Without interactions, the trajectory of the particle only depends on its initial conditions and the potential field but with interactions it also depends on the other particles that from the physical system. Because of this, the `particle` class for the interacting program only stores its mass, charge, its initial position and velocities in a r_0 and v_0 `numpy.array`s and its current position and velocities in a r and v `numpy.array`.
The only method of this class is the `reset` method which sets r and v to r_0 and v_0.

Even though each particle can have different mass, in practice all of them will have the same one. Charge is also not used but its added so different interactions could be easily added in the future.

4.6.2 `PhySystem` class

`PhySystem` contains the Verlet algorithm and computes the trajectories of all the particles, among other information such as temperature of the system and the distribution of velocities.

The class is initialized by providing a `numpy.array` full of `particle` objects which will form the system and a python list with the units of the system in the following order: [Unit of energy, Unit of length, Size of the box (in units of length)]. Remember that the unit of energy is the minimum of the associated Lennard-Jones and the units of length is the radius of the particles (See section 4.1). Note that this radius is not stored in the `particle` object.

Once initialized you can use the `solververlet(T,dt)` method to compute the trajectory of the system until a time T units of time have passed (in reduced units) and at a time step of dt. After the computation (which may take some time depending on the number of particles, T and dt) the following information will be available:

- X,Y,VX,VY,V: the x,y coordinates and the velocities (x and y component and magnitude) of all particles at all time steps. They all are `numpy.array`s of shape ($\frac{T}{dt}$,N) so the first index accesses the time and the second the particle.
- T: associated temperature of the system. `numpy.array` of one index to access time.
- MB: associated Maxwell-Boltzmann curve of the system. `numpy.array` of one index which corresponds to the x-axis of velocity histogram plot (sampled with 100 points in the [0,V.max()] range)
- Vacu,Tacu,MBacu: lists with the cumulative velocities (magnitude), temperatures and Maxwell-Boltzmann distributions for N,2N,3N... each element of the lists is an object of the same form of V, T and MB but with increasingly more particles (except for MB which is sampled with 100 points always).
- KE: total kinetic energy of the system. `numpy.array` with one index that accesses time .
- U: total potential energy of the system. `numpy.array` with one index that accesses time.

4.7 Future improvements

The interacting particles program was not able to be finished before the internship ended. The program has lots of potential to show interesting physical systems like for example:

- Two gasses at different temperatures reaching thermal equilbrium through collisions.
- Other interacting systems like star/planetary systems (changing the interacting potential to $-\frac{1}{r}$ is easy to implement)
- Crystal lattices or fluids reacting to external potential fields.

I hope that in the future this program can be continued by other students of the Universitat de Barcelona.

5 Usage of the Program

After so many features had been added to the non-interacting particles program. It was pointed out by some colleagues and users that the program was complicated and overwhelming when it was first seen by someone. In this section a detailed explanation of the intended usage of the program is presented.

5.1 Interface

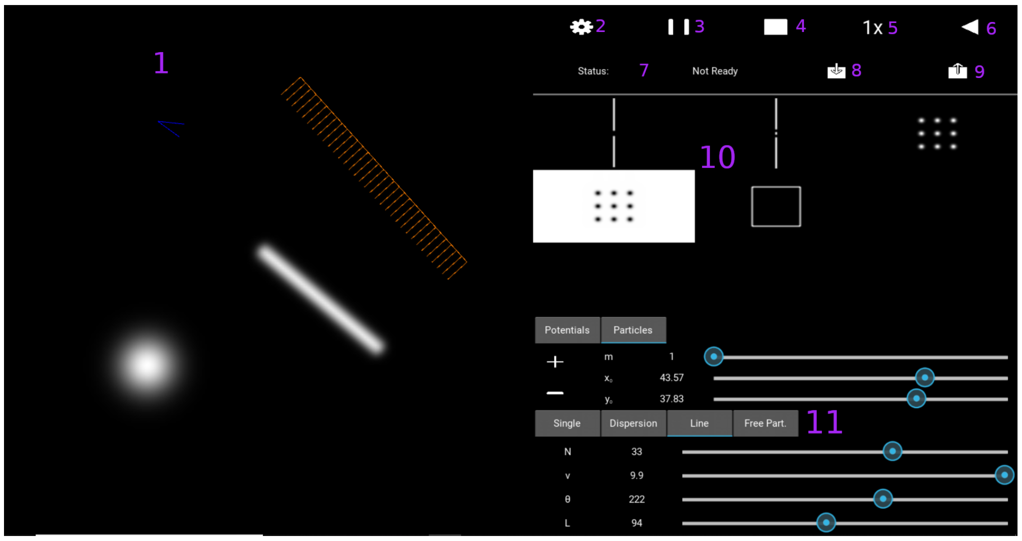


Figure 3: Elements of the interface

1. Main screen. The previews and animation will show here. An example of a gaussian potential, a Woods-Saxon potential, a line mode preview and a dispersion mode are shown.
2. Compute/Play button.
3. Pause button.
4. Stop button.
5. Speed selection button.

6. Reverse time button.
7. Status of the computation/animation.
8. Save button.
9. Load button.
10. Demos.
11. Selection of potentials and particles.

5.2 Adding Potentials

The intended way to use the program is to first add the potential fields (or choose from an already prepared demo) and then add the particles. To add potentials you must be situated in the 'Potentials' tab on the bottom right of the screen. Then the kind of potential has to be chosen by selecting its corresponding tab (this defaults to the Gaussian potential). After deciding the parameters (see the parameters in section 3.3.1) then the location of the potential can be chosen be using the sliders above or by clicking any point of the left screen. Finally the potential can be added by pressing the '+' button next to the position sliders. The left screen will automatically update to show the newly added potential.

If you wish to remove the potentials the '-' button has to be pressed. This will erase all the present potentials (without affecting the particles). Sadly all the potentials are deleted by this action and you cannot delete one potential at a time.

Note that you can choose a demo and modify it by adding more potential field.

5.3 Adding Particles

To add particles you must be situated in the 'Particles' tab and select which mode of addition you wish to use (by default 'Single', see section ... for other modes). Like in potentials, the position of the particle/cone/center of dispersion of the particles can be chosen with the sliders or clicking on the screen. Above the position sliders you can choose the mass of the particles as well. For the modes 'Dispersion' and 'Line' the direction of the cone/line can be also specified by dragging the mouse after selecting the position on the left screen.

While the parameters are been adjusted, you will see orange indicators on the left screen showing a preview of the particle to be added. The indicator is different for each mode. When the parameters are the ones desired the '+' button next to the positions sliders will add the particle(s) and the preview indicator will change from orange to blue, indicating the particles that have been added. You can continue to add particles after this.

In the same way as with potentials, the '-' button will erase all particles.

5.4 Computation and Reproduction

When the combination of potentials and particles are ready. You can press the compute button indicated by a cog on the upper left corner of the right screen, the cog will turn blue and the computation will start. The process can take more or less time depending on the number of particles added and complexity of the potential. The progress can be checked in the console. After the computation is complete the cog will turn into a play button.

To start the animation you can press the play button. The speed of the animation can be increased by pressing the speed select button. The pause button will pause the animation and can be resumed by pressing play. The stop button will stop the animation and reset it to the start.

If you wish to modify the potentials/particles you can do so by adding them directly (which will stop the animation) or stopping beforehand. After a modification, a computation has to be done.

At any point during the animation the reverse time button can be pressed and all the particles will be re-added with their speed flipped. After another computation the system will start to evolve like time has been reversed.

5.5 Saving and Loading

Before or after the computation. The save button can be pressed to open a pop-up that will allow to save the current configuration potentials in a file.

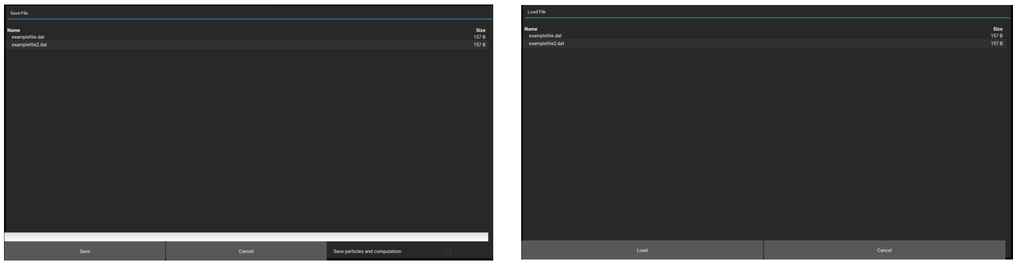


Figure 4: Save (left) and load (right) screens

After pressing the save button the pop-up shown on the left image of figure 4 will appear. After typing the name of the file to be saved in the white field the button labeled "Save" can be pressed to confirm. If the check-box 'Save particles and computation' is checked the file will also contain the particles and the computation (which should be performed before saving). The 'Cancel' button an also be pressed to close the saving window.

A saved file can be loaded by pressing the load button, which will make the load pop-up shown on the right image of figure 4 appear. Selecting a file and pressing the button labeled 'Load' will automatically close the window and load the file. If the 'Save particles and computation' check-box was checked in the saving process then the play button can be pressed immediately to start running the animation, otherwise the compute button will show. Like with the saving window, the 'Cancel' button will close the loading window.

6 Conclusions

This project has allowed me to develop and learn skills both as a physicist as a programmer. In every step of the development I learned something new and I have expanded my array of tools to tackle any future problems in my scientific and professional career.
I also have learned how to organize and prepare for long term projects, which is a invaluable skill for any programming related work.

I wholeheartedly recommend this experience to other students of physics at the UB as this has been growing and gratifying to work in such a good environment.

References

[1] Erwin Fehlberg, *Low-Oder classical Runge-Kutta formulas with stepsize control and their application to some heat transfer problems.* NASA Technical Report 315.
https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19690021375.pdf (page 18)

[2] Daan Frenkel, Berend Smit, *Understanding Molecular Simulation.* Academic Press, San Diego CA, 2002.

[3] Dusry Philips, *Creating Apps in Kivy.* O'Reilly, Sebastopol CA, 2014.

[4] QuantumLabUB and ClassicalLabUB github repositories:
`github.com/brunojulia/quantumlabUB and`
`github.com/brunojulia/classicallabUB`

[5] Smashicons, *Essential set icons*, `www.flaticon.com/packs/essential-set-2`.