Practiques en empresa

Arnau Jurado Romero

arnau.jurado.romero@gmail.com

Contents

1 Introduction

This report summarizes my work done from February 2019 to July 2019 at the Departament de Fisica Quantica i Astrofisica of the Universitat de Barcelona under the supervision of Bruno Julia Diaz in collaboration with the QuantumLabUB project. The work included development of python applications with graphical interfaces and participation in weekly meetings. A total of ### 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 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 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 each of 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 use teamwork to overcome obstacles and they were a very satisfactory experience.

2.2 Choosing the Simulation

Among the different physical systems developed previously at Quantum-LabUB 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 as many 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).

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 $\mathbf{y}$ and $\mathbf{f}$(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 at t_0, $\mathbf{y}_0$, their is determined by equations (3) and (4):

$$\begin{aligned} \frac{dx}{dt} &= \dot{x} \\ \frac{dy}{dt} &= \dot{y} \\ \frac{d\dot{x}}{dt} = \ddot{x} &= -\frac{1}{m}\frac{\partial \Phi(x,y)}{\partial x} \\ \frac{d\dot{y}}{dt} = \ddot{y} &= -\frac{1}{m}\frac{\partial \Phi(x,y)}{\partial y} \end{aligned} \tag{6}$$

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 adapaptative 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 for 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:

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

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

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

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

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

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

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

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

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

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 and will be moving in 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 it leaves the box or a time T has past since the initial instant ($t = 0$). For simplicity I have choose the box to be 200 arbitrary units.
The physical units will be determined from the units chosen for m, L and T:

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

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 in erg and the box would be 2 meter by 2 meter.

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:

$$L = 200 \quad m = 1$$
$$T = 60 \quad h_{max} = 0.1[T] \quad n_{max} = 10 \quad \epsilon = 0.01 = 10^{-2}$$

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 algorythm from acheiving the desired precision, ϵ.
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 list of the implemented potentials to date with their parameters and special considerations.

1. **Woods-Saxon potential**:
   - Formula:
     - $\Phi(x,y) = -\frac{V_0}{1+e^{\frac{r-R}{a}}}, r^2 \equiv (x-x_0)^2 + (y-y_0)^2$
     - $\frac{\partial\Phi(x,y)}{\partial x} = -\frac{V_0(x-x_0)e^{\frac{r}{a}}}{ar(1+e^{\frac{r-R}{a}})}$
     - $\frac{\partial\Phi(x,y)}{\partial y} = -\frac{V_0(y-y_0)e^{\frac{r}{a}}}{ar(1+e^{\frac{r-R}{a}})}$
   - 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: size of the potential well/mountain, for $r > R$ the potential rises/drops rapidly to 0
     - a: slope of the transition zone around $r = R$. For stability, $0.1 < a < 1$
   - Considerations: has an exception for $r = 0$ for the derivative. The derivative tends to 0 on $r \to 0$.

2. **Gaussian Potential**:
   - Formula:
     - $\Phi(x,y) = V_0 e^{-\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2}}$
     - $\frac{\partial\Phi(x,y)}{\partial x} = -\frac{V_0(x-x_0)}{\sigma^2}e^{-\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2}}$
     - $\frac{\partial\Phi(x,y)}{\partial y} = -\frac{V_0(y-y_0)}{\sigma^2}e^{-\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2}}$
   - 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 have been done on the method to see its stability and accuracy. Theoretically, the method should be accurate in all of the dependent variables according to ϵ.
To analyze the results, 7 different plots have been 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 of the particle is independent of time (see eq. 1) 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.

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

lets me 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 where not necessary in this case.

3.4.1 `Phi` class

`Phi` is a python object that stores the analytical expressions of the potentials and its partial derivatives and 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.

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.

1. 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. 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 and important error in the implementation of the RKF45 method (see section ...)

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. The compute and play button was 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 analytical function far more complex.

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:

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

4.2 Periodic boundary conditions and interaction truncation

...

4.3 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 (15) and (16):

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

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.

4.4 Force computation

...

5 Usage of the Program

After so many features had been added. It was pointed out by some colleagues and users that the program was complicated and overwhelming when it was first seen by someone. To help explain the program a button was added to serve as a tutorial of sorts. In this section a detailed explanation of the intended usage of the program is presented:

5.1 Interface

...

5.2 Adding Potentials

My 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 ...) 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.

Like with the potentials, the '-' button will erase all the 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 '1x' button. The pause button will pause the animation and can be resumed by pressing play. The stop button will reset the time of the animation to the start and stop it.

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

...

6 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.