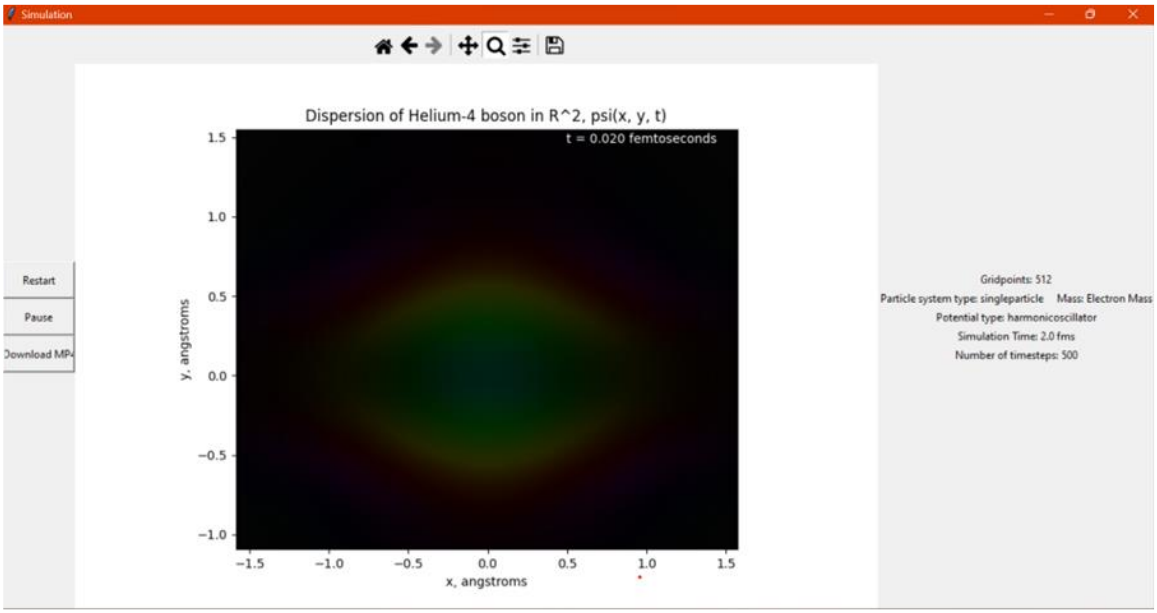
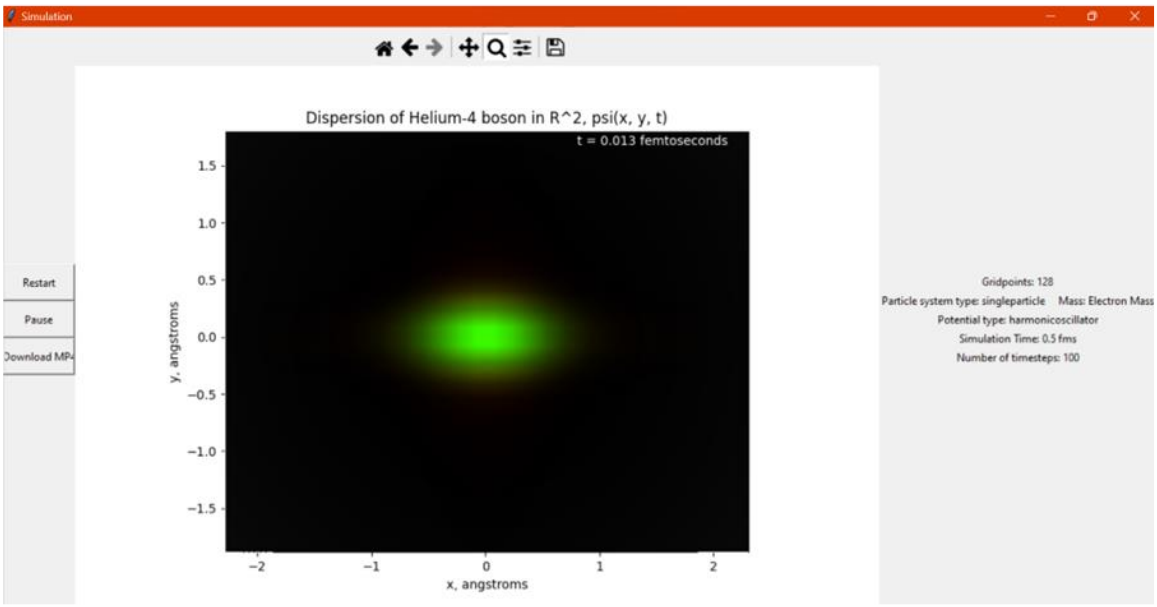




Simulating a superfluid

An exploration into the mechanics of Helium-4 and replicating these properties in a real-time simulation for use in education.
by Ismail Mehmood

Contents

Analysis

Problem Identification

The problem this project sets out to rectify is the lack of simulation or VFX representations of superfluids, more specifically educational tools which can demonstrate the simpler properties of such superfluids without the user needing to have a detailed understanding of the calculations and physics involved.

Currently, online superfluid information is mostly restricted to scientific research papers and graphs used for proving or deriving complex formulae. The amount of information regarding superfluids specific to a beginner is restricted only to the absolute fundamentals of their existence, with no demonstrations of some of their most interesting properties available.

Whilst it is true that superfluids are not closely linked to the scheme of work at A-Level in Physics, there are some concepts that can be seen in action such as the conservation of angular momentum and the properties of ideal gases. However, the main advantage of such a simulation being accessible to beginners is that it will encourage A-Level students to pursue Physics at a higher level and demonstrate some of the most interesting facets of the subject. Part of the aim of this project is to inspire students in the Sixth Form to pursue Physics at university level. It also allows students to access parts of Physics which they would otherwise not be able to outside of a university due to budget and practicality.

This program should be able to calculate the various physical properties and activities of a superfluid, the exact properties necessary for this simulation will be determined in this analysis and stated clearly at the end. This is due to the research required in order to determine which properties can be calculated mathematically (within my ability). Any reasoning behind this should become apparent before the end of this analysis.

User(s)

Such a simulation would have a wide user base, from post-GCSE students all the way to first-year undergrad students and would suit any level of understanding from basic curiosity up to a first-year university Physics student. However, the intended stakeholder, and thus the environment for which the design of this program will be optimised, is a Sixth Form Physics teacher demonstrating the basic concepts of a superfluid to an extra-curricular group of Year 12/13 Physics students.

Mr Powell is my chosen user for this project, a Physics teacher looking to demonstrate A-Level Physics concepts in action, using unfamiliar, intriguing contexts.

There are few computational solutions to this problem that already exist, most of which are incredibly limited or require lots of prior knowledge to use. This solution aims to take advantage of the existing research into computational solutions, combined with some recent developments in the Gross-Pitaevskii equation, for superfluid representation which is both accurate and accessible.

Initial Interview

Here is the transcript of my initial interview with Mr Powell discussing the initial expectations of this program:

1. What is this simulation to be used for?

Mr Powell: This simulation is to be used for teaching superfluids as an extracurricular topic to students in the Sixth Form currently studying Physics. Its main purpose is to show some of the properties of superfluids graphically given the impracticalities of an actual demonstration and also the lack of videos available due to the difficult conditions required to experiment with a superfluid.

2. What's wrong with existing solutions?

Mr Powell: Aside from the lack of existing solutions, most superfluid constructions are designed for scientific research purposes and tend to be in the form of detailed graphics in papers. These are not particularly suitable for those simply with a casual interest in superfluids as they tend to involve lots of complicated maths and are orientated towards researchers instead of students. As far as Ismail and I have been able to determine, there is only one superfluid simulation which can be easily used by somebody with absolutely no experience of superfluids and their peculiar dynamics.

[This simulation is presented as Solution 1 in the Research: Existing Solutions section]

3. Will this simulation require any specific/additional features to the original outlined brief at the beginning of this document?

Mr Powell: In addition to the outlined brief, the only feature which I believe that this simulation will require is the ability to show and hide each of the features on demand. Regardless of the interface window, it would be nice to simplify the output for demonstrations.

4. What features, if any, are required for this model to function correctly in your chosen environment?

Mr Powell: It would be useful if this model can:

a) Support direct intervention, more specifically vortices, heat sources can be dragged onto specific points on the simulation output. This is both simpler and more intuitive for students to see as opposed to typing in a command or coordinate.
b) Include a GUI with clearly visible parameters on an appropriate scale. This is mostly for the benefit of the students, so that they can clearly see adjustments being made but also so that I can manipulate the simulation without having to resort to a textbook.

5. Are there any specific parameters or variables you would like to control or experiment with in the superfluid simulation?

Mr Powell: The concepts I generally explain to students are friction-less motion and circulation, as these are generally the most interesting and simplest to explain. Representing the Landau two-fluid model would be a plus, but I understand that this is less feasible, especially simultaneously with some of the other variables involved. With that in mind, the specific parameters which I will need to modify are:

a) An obstacle, some item to introduce circulation to the fluid. Ideally, the size and velocity of this obstacle can be adjusted to suit requirements.

b) The temperature of the system. This can be devised and used in many different ways, but its presence is necessary in order to observe the evolution of the system at different stages in cooling/heating.
c) The condition that the superfluid is in. It would be useful to switch between simulating an open container and a trapped environment.
d) If a two-fluid model is chosen, a trapped and rotating environment would be a nice addition to show the reaction of each component to the circulation. This expands upon the angular momentum work already in the A-Level specification and would be a welcome addition.

6. How would you like to be able to analyse the results of the superfluid?

Mr Powell: In terms of output, the most essential consideration is the real time output of the program. However, an integrated recording mechanism would be very useful, as well as the ability to record the evolution of the superfluid over an extended period lapsed into a shorter clip without having to run the simulation for the entire duration. No mathematical output is required, but some accompanying explanations of the ongoing calculations when modifying parameters would be a useful feature. An ideal incorporation of this would be the ability to click on an artefact and view some basic properties and if feasible, where these properties were derived from.

7. Are there any integration requirements or compatibility considerations with other software tools or lesson plans and resources that you currently use?

Mr Powell: Currently, there are no formal resources for teaching this topic. I tend to use a mixture of videos and the online presentations of others in order to demonstrate this topic. There are therefore no integration requirements as such, compatibility considerations are limited to my laptop which I project onto a whiteboard for the class.

User Requirements

1.

i. Support direct intervention. Accommodate the insertion of vortices and heat sources to be dragged onto the simulation space.
ii. All adjustable parameters must be contained within a GUI. For example, I should be able to change the potential of the environment or the displayed energy state of the system using a slider or button.
iii. The simulation should allow for the localised insertion of circulation. Effectively, I want to demonstrate the conservation of angular momentum and how the vorticity of a superfluid is directly related to the vortex behaviour.
iv. It should be possible to raise the temperature or potential of the system to a high enough degree such that the system stops behaving like a superfluid. Hence, the simulation should be able to determine the lambda temperature of the condensate.
v. It is required that the environment of the superfluid can be manipulated between predefined environments including but not limited to: an open environment (such as a glass beaker), a harmonic potential (of adjustable frequency) and a rotating system (such as a spinning plate). **[5]**
vi. As a final requirement, the solution should be able to clearly represent the Landau two-fluid model and how a superfluid can be modelled as two distinct components.

Research: Existing Solutions

Whilst there is no existing comprehensive solution which covers every aspect of my intended simulation, there are several simulations covering individual aspects similar to those I wish to model,

and some which may partially cover properties that I intend to make use of. Superfluids are not generally a curricular topic at school level and as such, lots of the documentation and information surrounding them is aimed at technical researchers – meaning it assumes a certain base level of knowledge and often is focused at providing some new insight or technical detail as opposed to general, novice-friendly information. However, there are some aspects of these papers and projects which will be pivotal in designing the best solution for my problem and as such I have chosen to include them in this section.

Solution 1: WebGL Superfluid Simulation using dGPE (Stagg, George) [6]

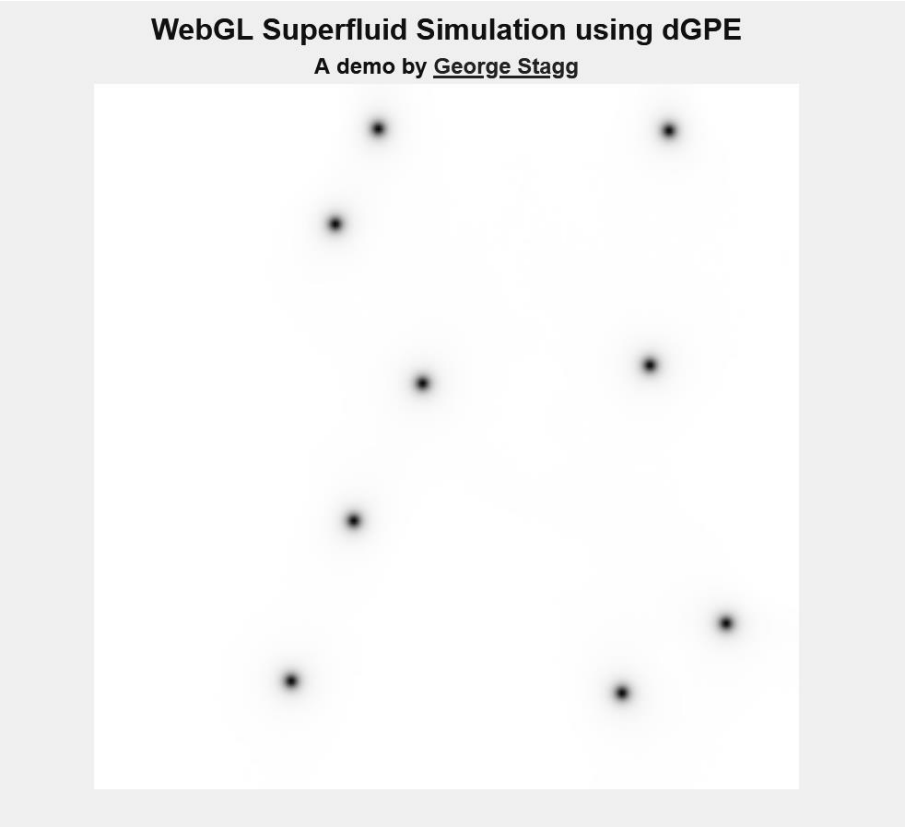


Figure 1 - The solution in its unmodified state.

The intended purpose of this simulation is to model the evolution of a boson-based superfluid whilst inserting vortices (quantized packets of circulation within a superfluid). Circulation in a superfluid is limited to these packets of a fixed size and shape. This simulation is mainly a proof-of-concept model, showcasing the formation of a vortex lattice in a trapped, rotating system. This effectively proves that the dissipative Gross-Pitaevskii equation being used to calculate the vortex polarity and energy dissipation of the system using the normalised wavefunction of the fluid is accurate in modelling superfluid phenomena (at least in the weakly interacting domain) as the vortex lattice generated effectively shows ideal flow within the superfluid (originally an engineering approximation of fluid flow with zero viscosity, but highly applicable here as superfluids actually have zero viscosity). This evidently is an application of a simulation far beyond my intended use case but the dGPE therefore suitable for modelling a superfluid.

One of the most important inclusions of this simulation is the dissipative function of the equation. This, for the purposes of those of us not researching Physics at the doctorate level, is a direct control of the energy leaving the system and thus an indicator of temperature relative to the lambda point of the chosen superfluid. For example, a temperature of absolute zero would be represented by a

dissipation of zero, meaning no energy leaves the system. This would clearly be shown as no waves will leave the system and there will be zero viscosity within the fluid. Whilst not a feature of this simulation, two-fluid models (where the superfluid is separated into a normal and a super component) would simply reduce to a super component. This, whilst a counter-intuitive and complex approach to one of the simplest physical properties to understand, has the benefits of significantly more precise imitations of actual observed and measured data.

Since this equation is a non-linear partial differential equation, there is some complexity involved in solving it. The creator of this simulation has chosen to use the RK4 time-stepping scheme and implement the visualisation of these solutions as WebGL shaders. The RK4 scheme is explained later.

One of the flaws of this mathematical approximation is the ability of the simulation to “blow up” under certain parameters. This occurs frequently when the dissipation parameter is set to 0 (i.e. the system is at absolute zero).

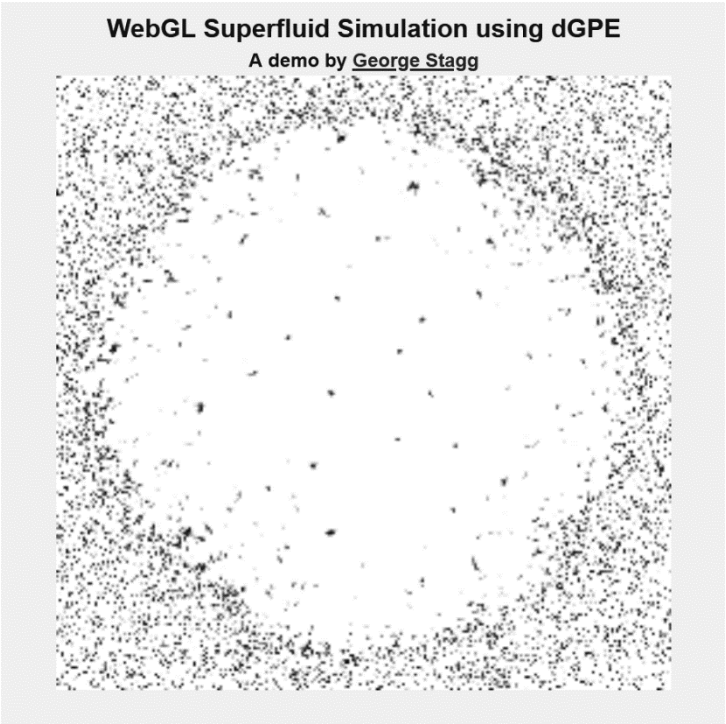


Figure 2 - The simulation after "blowing up". This occurred after leaving the simulation with a dissipation of 0 and adding in many vortices.

Overall, this simulation provides a useful insight into how the dGPE can be used to solve the possible states of a superfluid. The use of the iterative RK4 time-stepping scheme is a viable option for my simulation, as well as the use of WebGL shaders to visualise the vortices and phase. Unfortunately, I have little experience with WebGL and so creating such a visualisation package would be difficult. **[2]**

Solution 2: Simulation of dynamical properties of normal and superfluid helium (Makri, Nakayama)

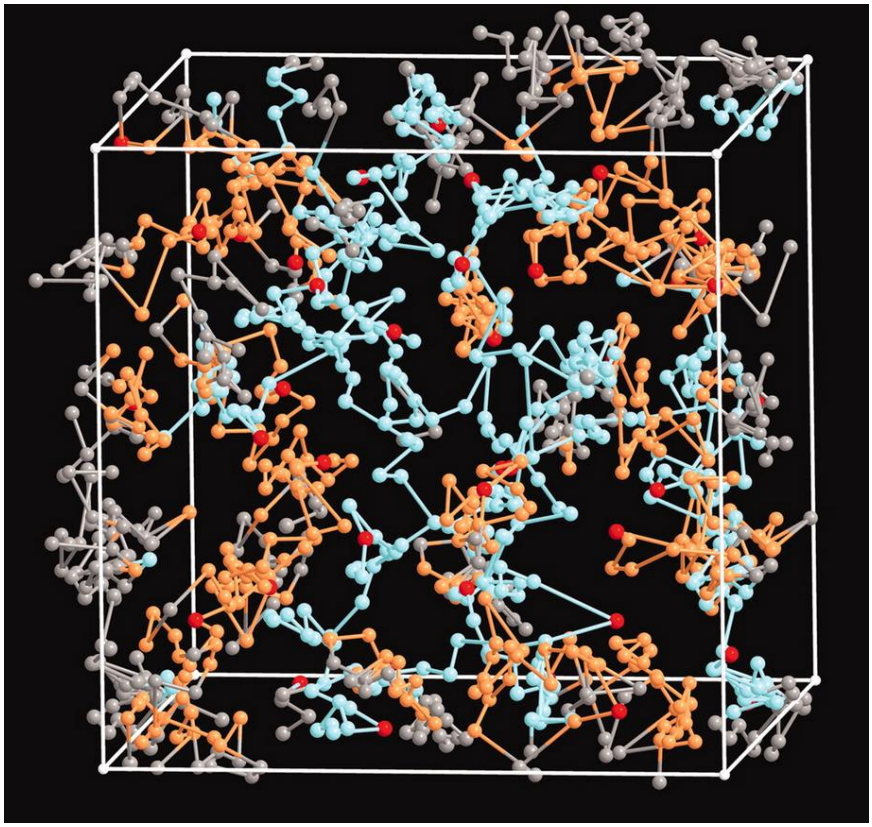


Figure 3 - A snapshot of the superfluid component of the Helium-4 system, where path integral beads are used to show each atom.

Solution 2 differs from Solution 1 in that it simply sets out to calculate the variables defining the superfluid in real time, neglecting to package this data into a visual representation. It also has the main objective of proving the accuracy of the statistical mechanics approach to the problem as opposed to creating a simulation. Nevertheless, it presents a possible alternative method.

This simulation uses a single particle velocity autocorrelation function, or in plain English, it attempts to relate a function of velocity to itself with a time gap. A general time autocorrelation function is used, developed through a forward backward semiclassical approximation and represented in phase space. The two new concepts here: forward backward semiclassical approximation and phase space, form the basis of this method. The FBSA is a complex series of functions developed to combine evolution operators and their adjoints in a quantum-mechanical expression within a time correlation function to be evaluated in a coherent state representation (similar to the momentum-velocity of a classical oscillator). [31] An iterative method (explained further on) is used to decompose a path integral which forms a discretised solution accurately evaluating the relevant phase space density. The path integral used here is a quantum representation of the basic action principle of classical physics, namely how the energy of a physical system changes over time. This is achieved by replacing the classical unique trajectory of a system with a functional integral (an integral over a region of functions instead of a space domain). This region of functions is defined by the set of possible trajectories (quantum-mechanically). This will give a complex number such that the square modulus of this number is the probability density of the possible trajectories.

This method carries numerous scientific advantages such as the inclusion of zero-point effects and the capture of imaginary components of trajectories. It can be sampled using the Monte Carlo method, and it also reduces the severity of the phase cancellation problem, where similar waveforms cancel and cause decoherence in the output. However, the complexity of the method used will

inevitably lead to a difficult program, and throughout the use of this simulation, the researchers involved have been forced to manually check the feasibility of a solution as the possibility of an iteration simply not converging onto a state or the method leading to some invalid possibility is high. As the objective of this program is to model some of the simpler properties of a superfluid, and properties such as soliton wave formation and the shape of the incoherent structure factor in neutron scattering are largely irrelevant for my purposes, this method does not present a viable solution; the complexity involved in the method is unwarranted for the necessary output.

Solution 3: QMSolve: A module for solving and visualising the Schrödinger equation (de la Fuente, Rafael) [29]

QMSolve is a Python module designed to solve the Schrödinger equation for one or two particles. This is achieved by discretising the Hamiltonian using an arbitrary energy from a function of the observed particle. A numerical technique known as exact diagonalisation is used to compute the eigenstates and energy eigenvalues of the Hamiltonian. The eigenstate of a quantum system is a state in which one of the defining variables has a determinate fixed value. As the allowed energies of the quantum system are discrete, not continuous, certain multiples, or eigenvectors, of an eigenstate represent the possible energy levels of the system. These eigenstates are plotted using a visualisation package known as mayavi in 3D, but 2D and 1D systems are plotted using an integrated visualisation system. As I intend to create a 2D simulation, I will not be trialling mayavi. I also noted that mayavi is incredibly temperamental on my chosen version of Python, 3.11, as it is dependent on VTK, TVTK and PyQt5, all of which have issues on the latest version of Python – 3.11 at the time of writing.

Running this simulation was a time-consuming and involved process as it required an older version of Python in order to use the VTK module (Visualisation Toolkit). I was forced to use PyEnv in order to run multiple versions of Python on my machine. It also includes a list of dependencies to install in a virtual environment, but I found the VTK version 9.0.1 recommended to be unstable at best, and so the simulations below were run on VTK version 8.2. I had the most success with Python 3.7.17, as my IDE (Visual Studio Code) no longer supports Python 3.6 and below by default. PyEnv is also highly temperamental on Windows machines and to save time, I used a Linux-based system to test the software. Before the examples, I've included a brief explanation of how this simulation works.

The Hamiltonian is the total sum of the potential and kinetic energies of a quantum system, as defined by its motion and position. The Hamiltonian is defined as a particle with a potential as a function of the particle's observables. The method *Hamiltonian.solve* is used to diagonalise the matrix created and outputs the energies and eigenstates of the system. This is a Hermitian matrix and so can be solved using the Lanczos algorithm, an iterative method designed to converge on the most useful eigenvectors of a matrix. This algorithm, whilst not too mathematically involved, has order $O(dmn)$ where the matrix it is applied to has dimensions $n \times m$ and d is the average number of non-zero elements in a row. This is an efficient method for low dimensional systems, but for 3D systems, QMSolve uses LOBPCG to converge to a solution quickly. LOBPCG, or Locally Optimal Block Preconditioned Conjugate Gradient Method, is another eigensolver which works on large Hermitian definite generalised eigenproblems. This is a function imported from *scipy* (mentioned in more detail later) which "preconditions" the matrix by reducing its condition number, achieved via inspection and utilising properties of matrix symmetry. The condition number of a matrix and the associated problem (in this case, the eigenvalue problem) is a measure of how much the output value of it can change with a small change in the input value. Hence, reducing the condition number of the matrix will increase the speed at which the iterative method converges onto the correct eigenstate.

Time dependence is implemented as an optional argument in the defined Hamiltonian, and the class *TimeSimulation* takes a parameter of either Split-Step Fourier or Cayley-Crank-Nicolson **[20]** (both methods are explained in more detail and prototyped later). The simulation is carried out using the initial wavefunction, time step, total time and number of steps to store in an array for later visualisation as parameters. This means that for potentials of the form $V(x, y, z)$, the Split-Step Fourier method can be utilised, which has a time step error of cubic error, an improvement over the quadratic error of the Cayley-Crank-Nicolson method, which has quadratic error in time. However, for systems where the potential is dependent on the momentum of the particle, Cayley-Crank-Nicolson is necessary.

In the documentation for this library, numerous examples are included of various quantum systems modelled using the package. I have included some of these below with a brief description of how they work, and what I can utilise from them.

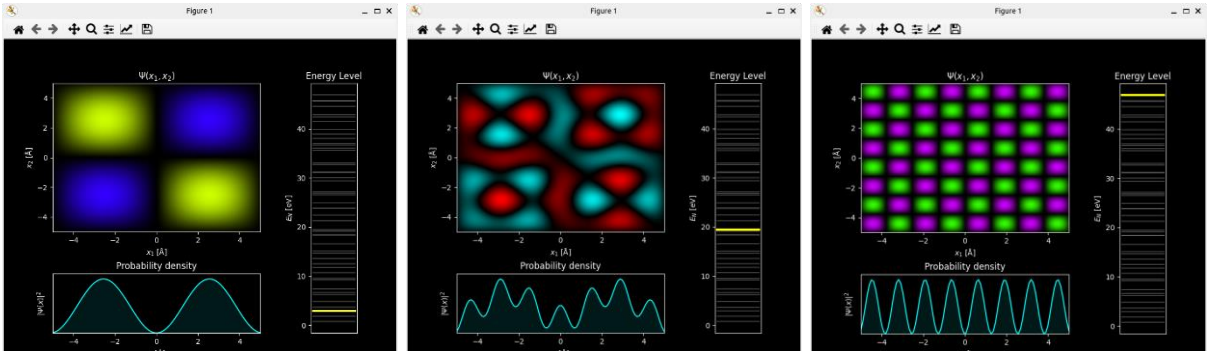


Figure 4 – Three of the eigenstate solutions to a system of 2 bosons – not considering the interaction parameter.

This representation of the eigenstates of a system of two 1D non-interacting bosons is an excellent example of the basic visualisation capabilities of the QMSolve package. The probability density of the position of the particles is shown in the graph, as well as a handy visualisation of the possible eigenstates shown on the energy state diagram. The actual graph shows the configuration space (or more accurately, the Hilbert space) where the probability density of the particle is shown by representing complex numbers with RGB colours. The real component of the complex number is demonstrated with the brightness of the pixel and the hue of the particle represents the imaginary (or phase) of the number throughout the grid. It is also useful to note the symmetry of the two bosons, clearly the symmetry condition for bosons, $\psi(x_1, x_2) = \psi(x_2, x_1)$, is displayed here. Also, observable is the random spacing of the energy state intervals, characteristic of bosons. A fermion-based example also exists which shows the equidistant nature of the eigenstates, but this does not apply to my chosen superfluid.

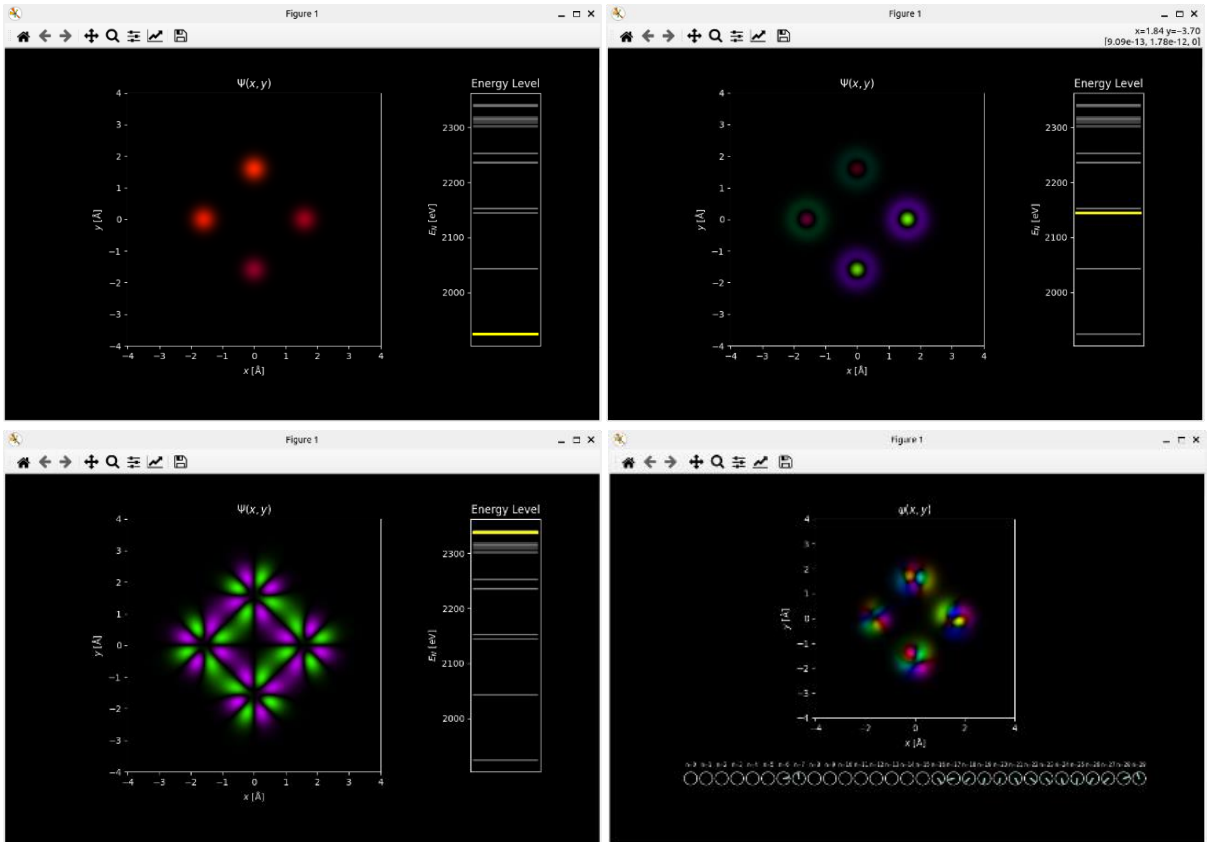


Figure 6 - 4 Gaussian wells – a theoretical system of 4 interacting atoms without singularities.

This example provided shows a different system of 4 Gaussian wells in 2D which interact similarly to atoms, with the absence of complex numbers which cannot be represented in the configuration space. More realistic examples are provided with hydrogen atoms and cations, but these sum the harmonic and Coulomb potentials in order to avoid singularities in the potential.

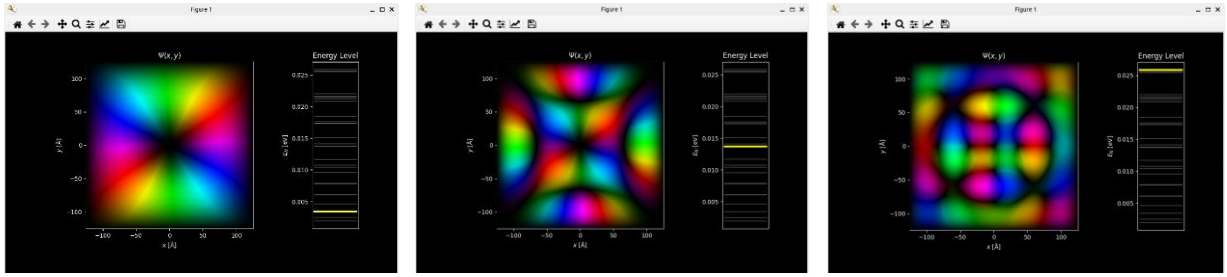


Figure 5 - An example of some of the eigenstate solutions of the location of a particle in a 2D box.

This picture shows 3 of the possible eigenstate solutions to the position of a particle in a 2D space. It is a relevant example because magnetic potential must be defined as a matrix in order to allow for momentum in the potential term. This complicates matters as matrix multiplication is required to apply the transformation potential matrix to the particle operator matrices. The magnetic field is pointing in the z-direction.

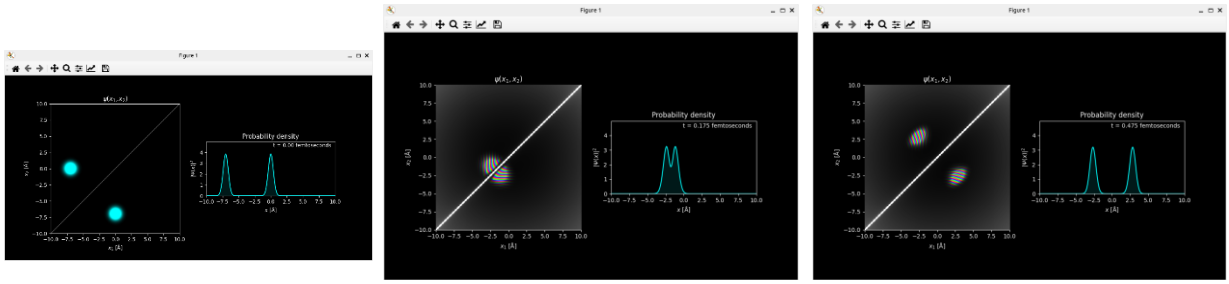


Figure 8 – Two 1D bosons, which now have an interaction potential and are evolved in time.

This example is highly relevant to my project, as it shows two interacting bosons evolved using a split-step Fourier method. This is achieved by performing a Fourier transform onto the potential term which can be propagated analytically in the frequency domain before an inverse Fourier transform brings it back to the spatial domain (the ordinary and symmetrised split step Fourier transforms are explained in the Method Comparison section). Shown are snapshots of the simulation as it progresses from $t = 0$, the probability density is shown using the same RGB scheme. This system retains the symmetry expected from a system of bosons, but its 1D limitation means that its usefulness for a qualitative understanding of a Bose-Einstein Condensate is very limited. **[3]** However, the use of the Fast Fourier transforms of $O(n\ log\ n)$ means that this method is highly time-efficient and should be relatively useful in scaling to higher quantities of particles and dimensions.

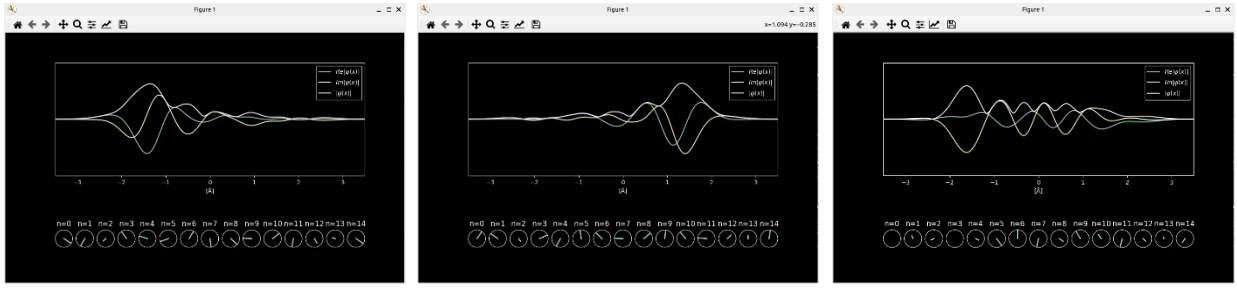


Figure 7 – A 2D charged particle accelerated in a constant magnetic field cyclotron using a varying electric field.

The cyclotron is an example of a system where the potential is dependent on the momentum of the particle as well as the position and energy of the particle. Hence, the total potential must be defined as a matrix which cannot be Fourier transformed. Here, the alternative Crank-Nicolson finite difference scheme is implemented (this method is visited in more detail in the Method Comparison). QMSolve also uses small circular widgets to control the coefficients of the system parameters, where the system can be modified as the simulation is live. I found these both fiddly and temperamental.

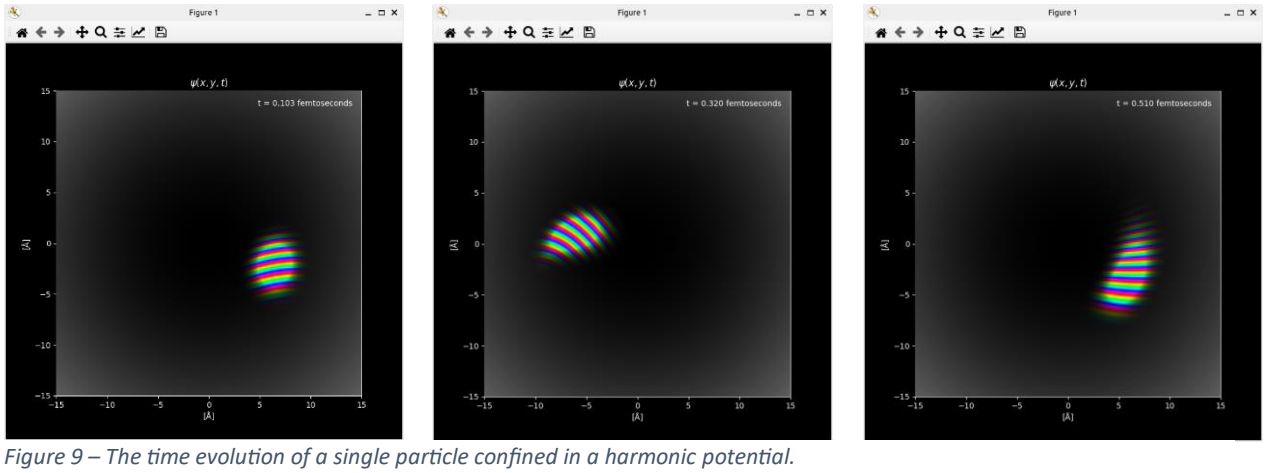


Figure 9 – The time evolution of a single particle confined in a harmonic potential.

This quantum analogue of the classical harmonic oscillator shares some properties with its classical equivalent. The potential energy of the system can be represented using Hooke's law. However, harmonic traps have relevance to superfluids as such systems can affect the formation of vortices or they can be used to approximate other potentials.

One significant drawback of this package is its limitation to the single or two particle linear Schrödinger equation. This is especially true for boson systems (such as the Helium-4 superfluid) as the coupling constant between the bosons is a significant factor in the evolution of the system (and also the reason for the non-linearity of the NLSE). The makers of the simulation overcome this by using the split-step Fourier transform method. Further investigation into the errors introduced by this are necessary in order to determine whether it is a suitable alternative. Another issue with QMSolve is its dependency on an old version of Python. I was only able to run all the examples successfully on Python 3.7.17 as lots of the visualisation methods are dependent on an outdated version of VTK (Visualisation Toolkit). **[1, 4]** Also, this simulation is only appropriate for those with a significant understanding of quantum mechanics and Python and is aimed thoroughly at scientific/researcher userbases.

The VTK package used in this simulation appears to be a viable option for representing the mathematical data, but the version and methods used in this project are outdated, unreliable and too complex for the end user. In my prototyping, I will investigate the possibility of using the newer version of VTK with my own set of finite difference methods **[27]** or the split-step Fourier transform.

From these solutions, it is clear that there are an abundance of methods to achieve various types of simulation of quantum systems and, specifically superfluids. However, before deciding on a method, it would be prudent to have a more thorough, scientific understanding of some of the options available. I have decided to split these into three sections: the mathematical understanding required, the scientific understanding required, and the computer science understanding required for both calculation and visualisation. Firstly, however, a second user interview will be conducted to discuss the feasibility of the initial idea and modify the user requirements as necessary.

Second User Interview

After having trialled each of these simulations for myself, I decided to speak to my user, Mr Powell, again and re-evaluate our initial set of expectations.

Ismail: Shall we begin with Solution 1? What aspects of it did you like?

Mr Powell: I'd like to start with talking about the interactivity of the simulation. I really liked that the user could apply changes as the simulation was live as opposed to having to change parameters in a program or recalculate or execute the program. The controls were simple and intuitive. I didn't necessarily like the styling of the control pane but the actual functionality was impressive.

Ismail: So to summarise, from Solution 1: the bits you liked were the controls and the easy functionality. What about the simulation was not to your liking?

Mr Powell: Firstly, it was not clear what exactly I was looking at. A brief explanation was included, but ultimately until it was explained to me that I was looking at a superfluid, introducing an obstacle and being shown the resulting image of a vortex in the system, I did not understand what was happening. My other issue with this simulation is that it only shows the quantum effects of vortices on the system, where it could also show the symmetric nature of bosons or allow for localised heating to show the sound wave nature of heat transport. It also has no representation of the two-fluid model.

Ismail: Okay. I'm going to briefly ask you about Solution 2, although I realise that it might be more relevant to my research and not so much to the end result.

Mr Powell: I don't really have a comment to make on this one. All I could see was lots of maths and not lots of graphics, which isn't really my objective.

Ismail: Understandable. Moving swiftly onto Solution 3, which had a lot more to look at, is there anything in particular you want to single out?

Mr Powell: Solution 3 has lots to speak about. I'd like to begin with the graphics. The visualisations are both impressive to look at and easy to understand logically. Some of the niches of the simulation are lost in translation, for example, what the colours represent is somewhat lost in translation, but the output is very close to what I'm hoping the outcome of this project will be. I like that a variety of environments can be shown using the same program, but I dislike needing an operator who has told me of the many hours it took to understand the package to do all this for me. The computation time is significantly higher that Solution 1, but it is understandable given the enhanced capabilities and still generally within reason. One adaptation that is absolutely necessary is a total overhaul of the system controls. Direct modification of the program to adjust parameters is not a workable solution and the controls that are sporadically provided are so complex that I couldn't begin to operate them, let alone explain them to a group. Having said that, I'd also like to question whether the same graphics can be applied to high-particle systems, at present this system can only handle one or two particles at a time.

Ismail: Perfect. Having seen the current options, I think a review of your requirements is in order. I'd like to take you through the original list and discuss how feasible each maybe, and determine some new requirements which may better suit your needs and the objectives of this project. Let's begin with User Requirement 1.i. Following my review of these solutions, it seems that some limited form of direct action is indeed possible, but as I believe it to be more feasible modelling this system on the order of a few particles as opposed to a macroscopic model, vortices are unlikely to be possible. What is achievable is the modification of coefficients of superposing eigenstates. This would indicate to the end user the various possibilities of eigenstate combinations. It may also be possible to add in a heat potential or similar whilst the simulation is paused or off, and then re-run the calculations to display the changed system.

Mr Powell: That seems like a reasonable compromise. I'm not sure of the benefit of the eigenstate probabilities for our purpose but pausing the simulation to create modifications is a totally reasonable solution. How about User Requirement 1.ii?

Ismail: I foresee no issues with User Requirement 1.ii., but 1.iii may be a struggle as the collapse of superfluid behaviour will simply cause the equation to lose meaning. Instead, what the system will be able to show instead is the behaviour at various potentials (in this case, temperature) and superpose these outputs to compare the predicted behaviours of the system. This should instead show the differences occurring in the system behaviour at different temperatures as opposed to simply the lambda temperature.

Mr Powell: That sounds preferable as we could instead focus on whether the behaviour at different temperatures will scale linearly. Going from your previous answer however, I assume User Requirement 1.iv is going to be a problem given that it is similar in nature to User Requirement 1.ii?

Ismail: Unfortunately, yes. What I propose instead is a potential with angular momentum and a system which behaves accordingly to show the movement (or lack thereof) of the bosons with

circular momentum. This could predict the location of a vortex but it's unlikely to be able to do this for a small number of particles. Having seen the compute time of the linear SE solvers, it's unlikely that the algorithm I choose will be able to compute the vortex locations on a large system (>2 particles).

Mr Powell: Okay. Here is my revised list of requirements which reflect some of the qualities I have seen in the other simulations.

I have deliberately left these points vague so that you may add your own interpretation, there's lots of different things that you could do to achieve these points and which you choose matters less than the output. Any one or few of the physical phenomena which are able to meet these criteria will be suitable. The function is more important than the idea, so to speak.

Ismail: Whilst I hope to meet every one of these user requirements, the actual project objectives and requirements determined by me are set out at the end of this analysis. Part of the purpose of the following research is to determine the feasibility of the user requirements.

Final User Requirements

a) Direct user control must be possible, I should be able to forward, rewind and freeze the simulation as well as zoom in and out.
b) I would like the ability to adjust the various parameters of the fluid during the course of the simulation. These should be displayed with an obvious scale and be adjustable as the simulation is live.
c) Interaction with the simulation must be possible. The user should be able to introduce some kind of obstacle into the simulation space.
d) Preferably, the environment should be the configuration space in which the system exists, as opposed to a complex space or single-property representation. This will allow scope to show how a 2D diagram can show many properties at once.
e) The program should output in words what it is calculating at any point, this is in order to show students what types of calculations might be required for such a model.
f) The program should be able to show interactivity between ideal identical particles and the symmetry relationship between bosons. This could be achieved using a two particle simulation.

Research: Maths

Formulating the problem as an equation:

The maths involved in creating this simulation can be grouped into three simple categories: solving some equation or finding some mathematical representation of the system, plotting the result of this calculation in some linear space and calculating the behaviour of the system over time. BECs are gases of bosons which all exist in the same quantum state and therefore are described by the same wavefunction (more on wavefunctions in the Research: Physics section). Wavefunctions of systems of bosons are typically determined by a differential equation, and in this case, the possible equations to use are all partial differential equations. The following is my investigation into finding solutions to these equations.

Finding Solutions to these equations:

One of the most popular methods for solving partial differential equation is the finite difference method. Finite difference methods are mathematical techniques for solving differential equations by approximating derivatives using finite differences. This is done by discretising continuous equations

into some approximation which can be solved numerically. The domain of the problem is divided into a grid of discrete points, and values of the function are approximated from the values of the function at neighbouring points. There are three basic types of finite difference method: forwards, backwards, and central differences. These are each derived from the Taylor series expansion of a function. This can be used to formulate a system of linear equations solvable using matrix or simplex methods. Each involves approximating the derivative of a function at a point by interpolating between the current point and a point ahead, the current point and a point previous or a point ahead and a point previous respectively. Assuming data is available in both the past and future the preferred method here is the central difference as it has a truncation error of $O(h^2)$, significantly less than the $O(h)$ of the forward and backwards methods. Central difference methods are common in solving partial differential equations as they tend to be numerically stable and convergent (where the numerical solution approaches the exact solution as h tends to 0). However, this comes at the disadvantage of being more numerically intensive, as numerical systems of equations must be solved on each time step.

The two methods most relevant to my project will be the implicit method and the Crank-Nicholson method **[20]**: a backward time, centred space method and a time-space centred method respectively, each with inherent advantages and disadvantages. This will be shown in the QMSolve trial where the time step is modified on each method, clearly demonstrating that the implicit method is favourable for larger time steps whilst the Crack-Nicolson method outperforms BCTS (standard implicit) when the time step is reduced.

Typically, a multistep method is employed to numerically solve an ordinary differential equation. Multistep methods are effectively higher-order extensions of a finite difference scheme, where solutions are dependent on more than a single previous point and its derivative. Multistep methods operate by referring to several previous points and their derivatives. It is important to note that multistep methods differ from half-step methods such as the RK4 method seen in Solution 1, where intermediate steps are taken between the current point and the next solution point, but then all information is discarded before the second step is taken. The advantage is that high-order approximations lead to high-order accuracy in spatial and temporal discretisation. The potential downfall of this method (aside from added complexity to the program) is that for some functions, a problem known as Runge's phenomenon can occur, where for some set of smooth functions, oscillations can occur in increasing intensity with progression along the interpolated points or as the order of the method is increased which are errors caused by using a higher order method. I do not anticipate this being a problem with my simulation however, as the function which I interpolate over are likely to be simple and demonstrably uniformly convergent (a mathematical property of a sequence of functions which must approach a limiting function at a uniform rate) over the interval I choose.

The main issue to prevent when forming a partial differential equation is ensuring that it is well-posed. This is the mathematical quality of the equation having a unique solution that continuously changes with adjustments in the data. There are algorithms to determine how well-posed a problem is, but the easiest and most relevant method to me is simply using the condition number of the Hamiltonian to determine how well-conditioned the energy matrix is.

Before attempting to implement a scheme in a program however, it is important to understand the difference between the two methods and the reasoning behind the different convergences and truncation errors. The following example is an adaptation of Bus 864 from the Simon Fraser University **[30]**. Imagine a partial differential equation $au_{xx} + bu_x + cu - u_t = 0$, and you wish to find a function $u(x,t)$ satisfied by it. First, a boundary condition is necessary, we shall use $u(x,0) =$

$f(x)$. The coefficients a, b and c may be functions of x, t. A solution shall be taken by considering the values that u takes on a grid of x, t values placed on a domain of interest. If we assume u to be smooth everywhere, we can use interpolation to determine arbitrary values for x and t. By calculating the gradient at a spacing higher and using this to approximate the value of the function at a "time-step" ahead, we can estimate the shape of the function.

Mathematical spaces:

This will return a series of values in an array which can be plotted onto a graph, but the function is being evolved in time and there must be a method of showing the evolution of the system across space.

Vector fields are functions of a space which have a vector quantity at any point in the region. In order to understand linear space mathematically, we generalise space to an N-dimensional vector space where basic rules of incidence geometry apply. Our use of this is in velocity fields. **[17]** As a field existing in the same region as the fluid occupies spatially, the direction vectors of the velocity of the fluid can be computed and used to evolve the fluid across space. It becomes necessary to understand two important concepts of vector fields: divergence and curl. Divergence is simple enough to understand logically, although computing divergence can be expensive computationally as it involves differentiating and integrating complex function of velocity.

Unfortunately, using vectors to evolve the system with velocity is not a practical solution due to the high expense of vector divergence and curl calculations at every single point in a discrete grid. However, it may be possible to incorporate vectors into the testing methodology by picking a few random points at the beginning of the simulation and then calculating the evolution using some other method and then working out the divergence vector of the particle at that point and comparing the values.

From this, I have learnt that I will need to define a wavefunction describing my system as a partial differential equation. This will be achieved in the Physics research. Also, I have explored vector fields as a possible way of evolving the simulation visually. However, I don't think that using vector fields will be necessary for reasons outlined in the Visualisation research section.

Research: Physics

Physical systems as equations: **[15, 21]**

In quantum physics, a wavefunction **[14]** is a mathematical description of the quantum states of a particle as a function of momentum, time, position and spin. It is typically denoted using the symbol psi, $\boldsymbol{\Psi}$. For Bose-Einstein condensates, the equation used is typically the Gross-Pitaevskii Equation. It is also known as the non-linear Schrödinger equation, as it is based on the Schrödinger equation with the addition of a coupling constant between the particles (setting this term to zero yields a Schrödinger equation describing a particle inside a trapping potential). It is a mean-field theory derived approximation. The coupling constant is determined by the following equation:

$$g = \frac{4\pi\hbar^2 a_s}{m}$$

Equation 1: The coupling constant (scattering) function

In this equation, g (the coupling constant) is proportional to a_s and inversely proportional to the mass of the boson. Assuming the boson mass remains a constant, g can be found from a_s, which requires some work to determine. The scattering length for a low energy system can be found using the low-energy limit:

$$\lim_{k\to 0} k \cot \delta(k) = \frac{-1}{a}$$

Equation 2: The scattering length of potentials

Here, k is the wavenumber, somewhat analogous to traditional frequency, applied to the spatial domain. This effectively means that for some k of the boson, a scattering length can be determined by the phase shift after interaction. Using the hard-sphere approximation, we can determine the s-wave scattering length of a low-energy hard sphere to be some function of -k as $O(k)$.

Having established the non-linearising variable of this function, we can move onto establishing the energy density of the system. This can be shown as:

$$\varepsilon = \frac{\hbar^2}{2m}|\nabla\Psi(r)|^2 + V(r)|\psi(r)|^2 + \frac{1}{2}g|\Psi(r)|^4$$

Equation 3: Energy density of a system of identical bosons, **[32]**

This becomes a form of the NLSE, the dGPE.

$$\mu\psi(r) = (-\frac{\hbar^2}{2m}\nabla^2 + V(\boldsymbol{r}) + g|\psi(r)|^2)\psi(r)$$

Equation 4: The dissipative Gross-Pitaevskii Equation (NLSE)

Here, μ is the chemical potential of the system, found by solving the triple integral of the square of the wavefunction:

$$N = \int |\varphi(\boldsymbol{r})|^2 \, d^3r$$

Equation 5: Condition relating number of particles to the wavefunction. Note the power may change with the number of dimensions.

Whilst this would be complex to attempt by hand, many packages exist to perform complex integrations in Python. These will be explored further in the Prototyping section. However, in this case we shall simply use some borrowed maths from mean-field theory, namely the Hartree approximation (this only works in a stationary state, but this can be dealt with after, the Hamiltonian formed will solve for the energy eigenvalue μ independently). As we are working in Hartree atomic units, this can be set to 1 in the initial wavefunction.

When time is finally added as a parameter of the function, we are left with:

$$i\hbar\frac{\partial\psi(\boldsymbol{r},t)}{\partial t} = (-\frac{\hbar^2}{2m}\nabla^2 + V(\boldsymbol{r}) + g|\psi(\boldsymbol{r},t)|^2)\psi(\boldsymbol{r},t)$$

Equation 6: The time dependent GPE.

The exact derivations and mechanics behind how this equation works are not relevant to this project. My task is to simply find the best solution to this equation for my needs. Whilst many partial differential equation solvers exist, I want to create one tailored to this task as many are optimised for specific scenarios or equations and would be inefficient for my problem.

Method Comparison

The Crank-Nicolson method **[30]** is a second-order finite difference method for numerically solving partial differential equations. Whilst only a second-order method, the prototypes will demonstrate that the fourth-order methods such as Runge-Kutta 4 will be too time-expensive for the simulation to

run in real time. The second order approximation is accurate enough for the purposes of this project, and this can be ratified further with the use of adaptive mesh refinement. **[28]** I was unable to find an example of adaptive mesh refinement that suited my purposes in the time I had.

The Crank-Nicholson method is a second order finite-difference method which uses the central difference method on both the time and space domains in order to discretise and form systems of equations. These systems of equations are linear in certain cases (typically when being applied to a simpler ODE) but can also be non-linear. Another significant advantage of the Crank-Nicholson method is that if the equation is linearised before the discretisation, often the tridiagonal method can be used to solve the produced system of linear equations. As this algorithm is of order $O(N)$ compared to the standard $O(N^3)$ for solving a system of linear equations by matrix inversion, it massively reduces the compute time at each time step.

Runge-Kutta 4 **[16]** is a fourth-order multistep iterative method for approximating simultaneous non-linear equations. It works by utilising the basic Euler (forward difference) method explicitly, but making 4 approximations of the slope within the time-step interval. The implicit method for solving is a combination of the central and backward finite difference techniques.

The synchronised split-step Fourier transform **[18, 48]** is a numerical method used to solve partial differential equations. The method relies on separating the computation of the solution into two steps: linear and non-linear. The non-linear and linear steps are computed in different domains; the linear step occurs in the frequency domain, and the non-linear step occurs in the time domain. Using the outlined dissipative Gross-Pitaevskii equation, both the linear and non-linear components of the equation have analytical solutions. The following was the topic of a research paper in 2004 **[47]** which determined that the split-step Fourier method can be applied to the initial wavefunction using the following three operators.

$$\Psi_0 = \Psi(x,t);$$

Equation 7: Initial wavefunction

$$\Psi_1 = e^{\frac{i\hbar}{4}\frac{\partial^2}{\partial x^2}}\Psi_0$$

Equation 8: Applying an operator to apply a half time step to the initial wavefunction.

$$\Psi_2 = e^{-ih[V(x)+g|c_0\psi+c_1\psi_1|^2]}\Psi_1$$

Equation 9: Applying an operator to the initial wavefunction to obtain a second intermediate wavefunction.

The completed time evolution can be achieved by:

$$\Psi(x,t+h) = e^{\frac{i\hbar}{4}\frac{\partial^2}{\partial x^2}}\Psi_2$$

Equation 10: The final wavefunction evolved a time "h" ahead.

Treating these parts separately can be done provided the step taken is small (hence the earlier example evolving in femtoseconds). The error is generally of order $O(dt^2)$ but using some borrowed maths from the paper mentioned earlier, by using the latest variant of $\Psi(\boldsymbol{r},t)$ instead of an earlier variant whenever a $g|\psi(x,t)|$ is required alongside the potential energy, the error can be dropped to $O(dt^2)$.

One difficulty in this approximation is the presence of the second derivative term in the first operator. Taking the derivative of the wavefunction here in space is so difficult that it proves useless. Instead, a Laplace transform is used to find an approximation.

A Laplace transform is an integral transform mapping a function of some real variable onto a complex domain $s = \sigma + i\omega$. In this case, it is used to transform a partial differential equation into an ordinary linear differential equation. This can be done using a simple integral:

$$F(s) = \int_{-\infty}^{+\infty} f(t) \cdot e^{-st}\, dt$$

Equation 11: The Laplace transform of a function in t.

Luckily, the Laplace transform is a well-studied domain (no pun intended) and many efficient solutions exist already in Python. In the Prototyping section, I have elected to use the Laplace transform built into NumPy as NumPy is required for other aspects of my solution already and the differences between other Laplace calculation methods are minimal. Fortunately, the scope of my project does not include creating a complex integral calculator and I will be able to adopt this method. The SciPy package ndimage includes a Laplace routine which is capable of efficiently performing the transform and returning a result in the complex domain, which can be applied in momentum space.

From this, it is clear to see that of the finite difference methods – the Crank-Nicolson method is the most appropriate for my simulation. However, the SSFT looks equally appealing and has the added benefit of a theoretically faster runtime and a reduced truncation error.

To establish which technique is best, I have compared the simulations from QMSolve and quTARANG and modified the programs to include clear runtimes for the time stepping methods using the TQDM wrapper. Under various conditions, the runtime of each simulation will be compared. The caveat with each of these methods is that QMSolve is solving the much simpler LSE, but unfortunately, I could not find a similar NLSE solver when this testing was carried out. Later, I became aware of pytalises, but unfortunately this was long after my testing.

Research: Prototyping

Before creating the prototypes, a suitable environment and structure for the programs are needed. Here I outline how I created a shared environment for each simulation to run with the same resources and set out a basic structure for each of the program according to the basic steps involved in the simulations.

A virtual environment in Python is a separated environment built on top of the base installation of Python on a system, where the dependencies and resources utilised by the project are isolated. This is a common approach in data science and for developers as the issues caused by dependencies changing, updates to libraries or clashing projects can be catastrophic. For mathematically complex projects, the use of libraries for calculations is essential, yet each library is frequently updated and this could lead to future issues in the project should a change be made. Aside from this, virtual environments download all relevant information from a library, making it possible to run the program offline – another distinct advantage. In this project, I will be using a virtual environment, characteristically defined by requirements.txt. This file provides all necessary information to replicate the environment in which this project will function on any device.

The environment that I will be using to prototype examples in this section is described by requirementsforprototypes.txt. It is an environment created with virtualenv, a Python module which

can be used to create virtual environments. The creation process I underwent is outlined below, along with the dependencies which will be used throughout the project. The following process is specific to the Windows command line (cmd) but could be adapted to other systems with little effort (bash and other alternatives are highly documented in the Python documentation for virtualenv).

I used the python package manager pip to install the virtualenv module, using the command:

```
pip install virtualenv
```

From here, I navigated to a directory I'd already created for the environment and used virtualenv to create and environment:

```
virtualenv Test_Environment
```

The environment is automatically created, and you can see this by browsing the directory, where all the necessary prerequisites are downloaded for your convenience. The environment must be activated or deactivated in order to use it; this can be achieved via a command line.

```
Test_Environment/Scripts/activate.bat
```

This launches the environment in the terminal window, alternatively the environment can be integrated into an IDE and you can run programs as usual. The command:

```
deactivate
```

can be used to 'close' the environment at any point.

One of the main benefits of this approach outlined was the ability to store dependencies in a requirements file, which is the standalone necessity to recreate an identical environment should it become necessary. Once all the desired packages have been downloaded (paying close attention to the versions used), the requirements file can be created using the "freeze" command.

```
pip freeze > requirements.txt
```

The requirements list from the environment I used to test the simulations I found is as follows as an example.

contourpy==1.1.0
cycler==0.11.0
fonttools==4.42.1
kiwisolver==1.4.5
matplotlib==3.7.2
numpy==1.25.2
packaging==23.1
Pillow==10.0.0
pyparsing==3.0.9
python-dateutil==2.8.2
scipy==1.11.2
six==1.16.0

The environment used Python 3.7 as QMSolve uses a VTK dependency incompatible with newer versions of Python.

As you can see, there are a large number of dependencies involved in the prototyping of this project. Each is outlined below with a brief description of its purpose and use in the project.

NumPy: NumPy is a library for Python commonly used for mathematical operations. It is especially useful for arrays and matrices of high dimensions, as the data structures it introduces are highly efficient and powerful compared with Python's inbuilt offering. In my project, NumPy is used to create arrays and matrices storing complex numbers, carry out mathematical operations such as the Laplace transform and rearrange "shapes" of arrays.

Matplotlib: Matplotlib is a plotting library for Python, designed for creating plots of data and other visualisations for Python and NumPy. In the following prototypes, Matplotlib is used to create a graph figure, as well as func_animation() to animate the plots.

VTK: VTK, or Visualisation Toolkit is an open-source software package for Python specialising in manipulating scientific data. In my case, it is used in the final simulations to show the time evolving examples of the system.

SciPy: SciPy is a library in Python for scientific data manipulation and calculation. In this case, I am using SciPy for the LOBPCG method, the integrate method, the Laplace transform and the Fourier transform.

QMSolve Prototype - Comparing SSFT to Crank-Nicolson:

Test 1: 2D Double Slit Simulation

Parameters: Simulation runtime 0.7 femtoseconds, timesteps 5600

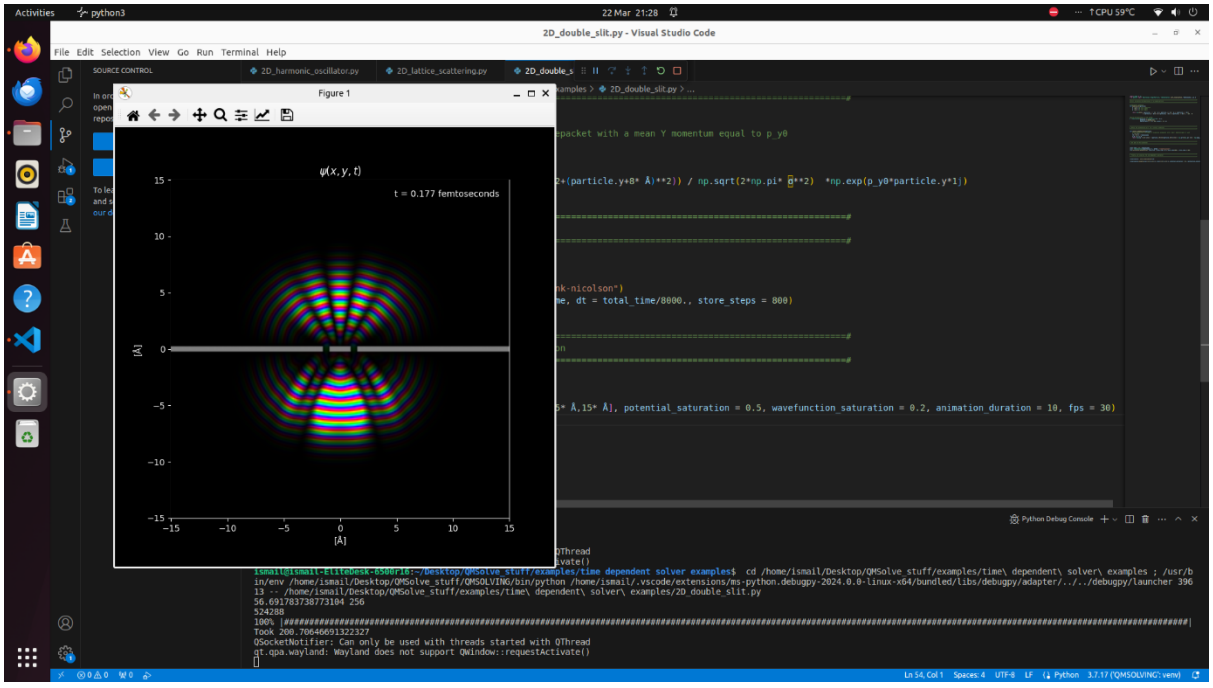


Figure 10: 2D Double Slit in QMSolve – Crank-Nicolson.

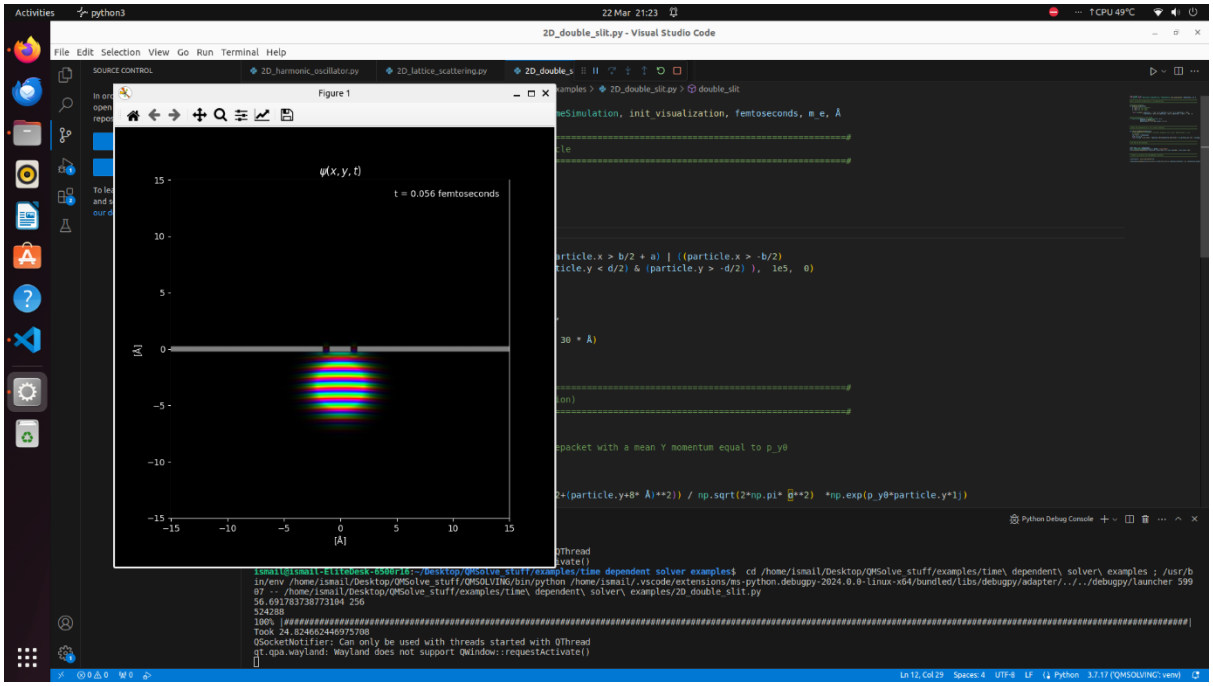


Figure 11: 2D Double Slit in QMSolve – SSFT.

Result: Processing time of Crank-Nicolson – 200 seconds to 3sf. Processing time of SSFT – 24.8 seconds to 3sf. The SSFT algorithm terminates over 8x faster.

Test 2: 2D Harmonic Oscillator Simulation

Parameters: Simulation runtime 1.5 femtoseconds, timesteps 1500

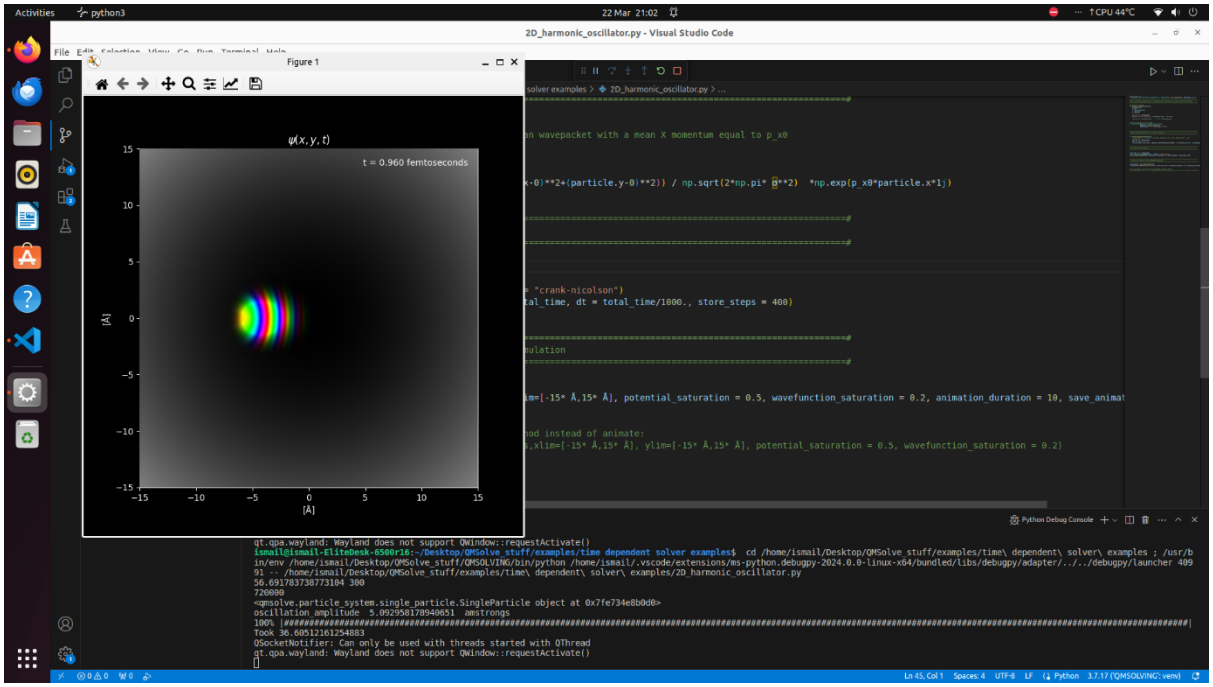


Figure 12: 2D Harmonic Oscillator in QMSolve - Crank-Nicolson.

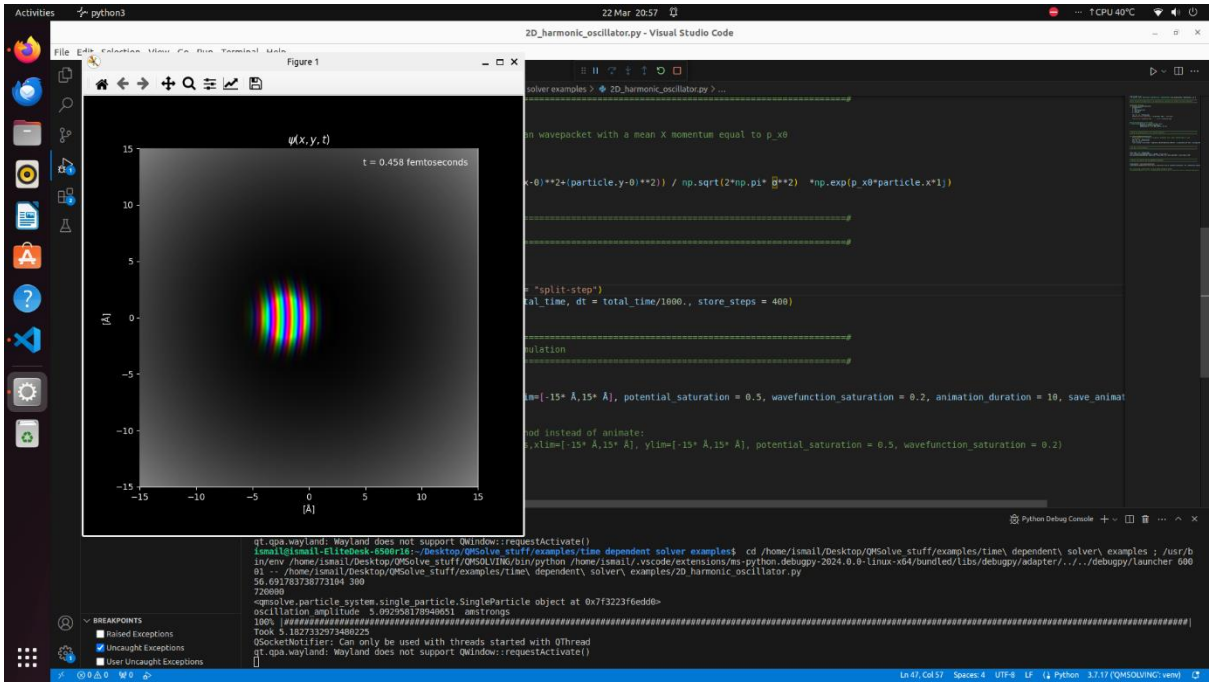


Figure 13: 2D Harmonic Oscillator in QMSolve - SSFT.

Result: Processing time of Crank-Nicolson – 36.6 seconds to 3sf. Processing time of SSFT – 5.18 seconds to 3sf. The SSFT algorithm terminates 7.1x faster.

Test 3: 2D Lattice Scattering Simulation

Parameters: Simulation runtime 0.4 femtoseconds, timesteps 800

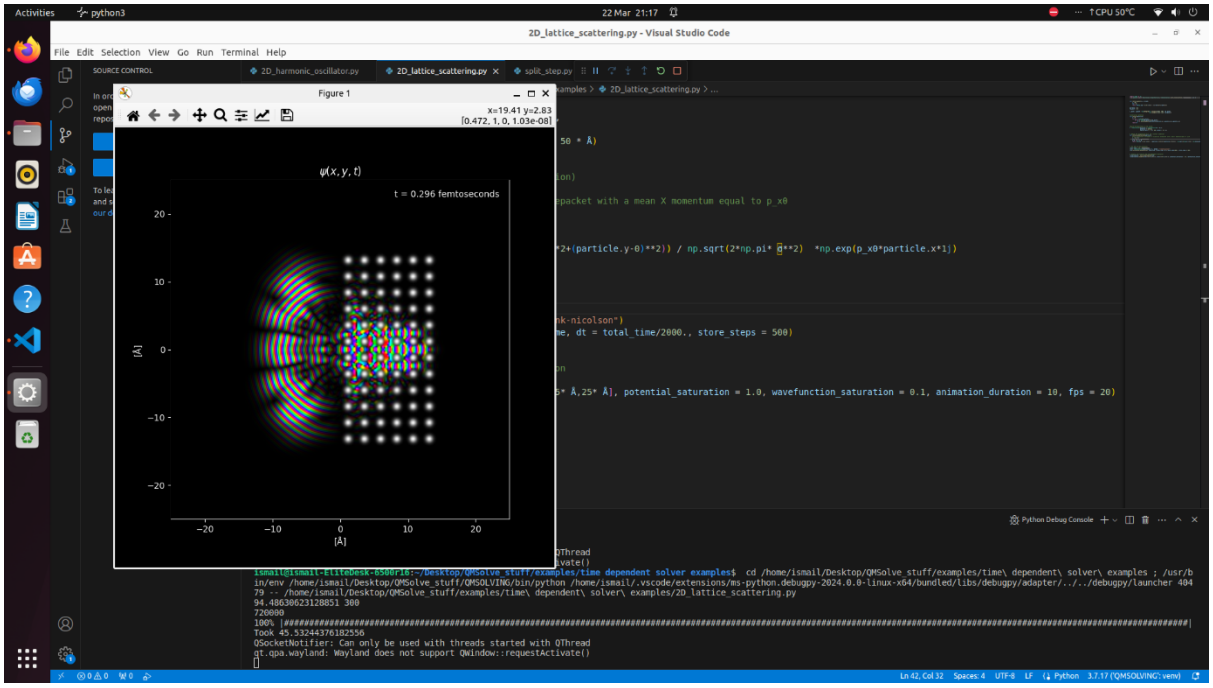


Figure 14: 2D Lattice Scattering in QMSolve - Crank-Nicolson.

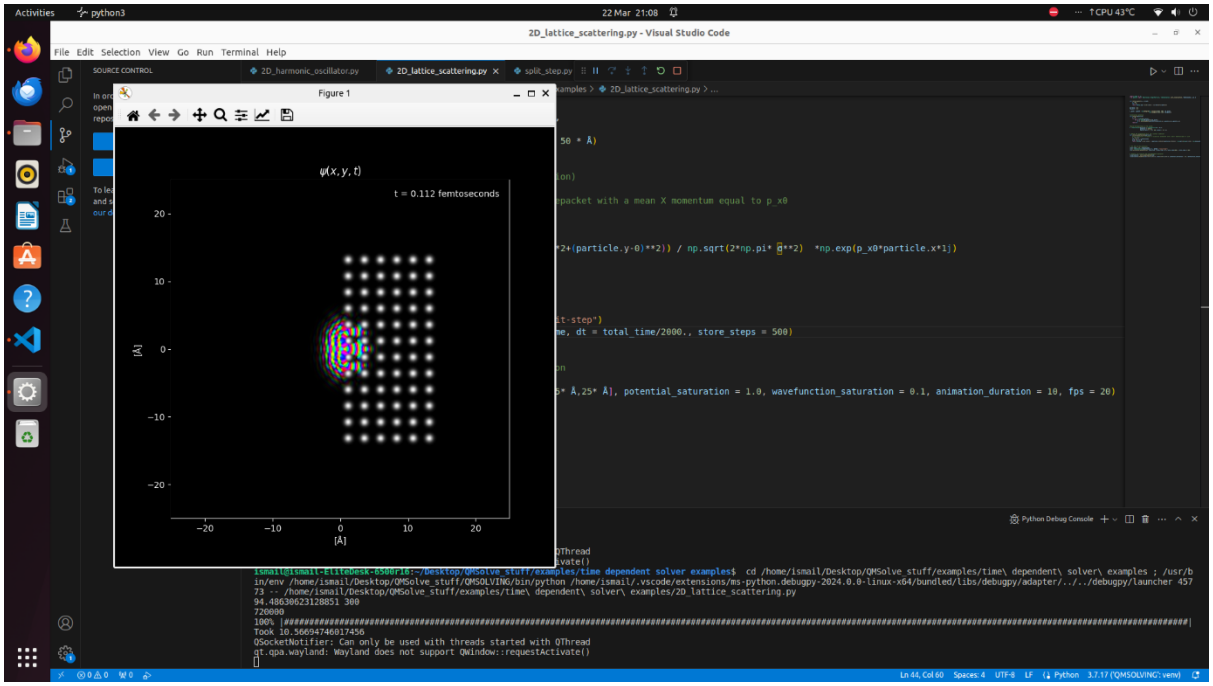


Figure 15: 2D Lattice Scattering in QMSolve - SSFT.

Result: Processing time of Crank-Nicolson – 45.5 seconds to 3sf. Processing time of SSFT – 10.6 seconds to 3sf. The SSFT algorithm terminates 4.3x faster.

The QMSolve prototyping clearly shows that for the same calculation, the split step Fourier method terminates several times faster than the Crank-Nicolson method. This proves that the SSFT is more time-efficient than the Crank-Nicolson method for the types of calculations that I will be performing.

quTARANG Prototype – Stability Testing:

One notable disadvantage of the Crank-Nicolson method is the instability in dynamical calculations. The quTARANG model uses the SSFT time evolution scheme to avoid this. Note that the outputs from this program are not animated, the numerical results are returned into a hdf5 file (a data file used for holding significant quantities of numerical data for manipulation). In order to view these results, I used the online site myhdf5 **[49]** to view the plot of the evolved wavefunction. The uniform position of the plot will prove the stability of the algorithm.

Test 1:

Simulation runtime: 3 fms

Time-step: 0.01

SSFT Duration: 00:15

Stable: PASS

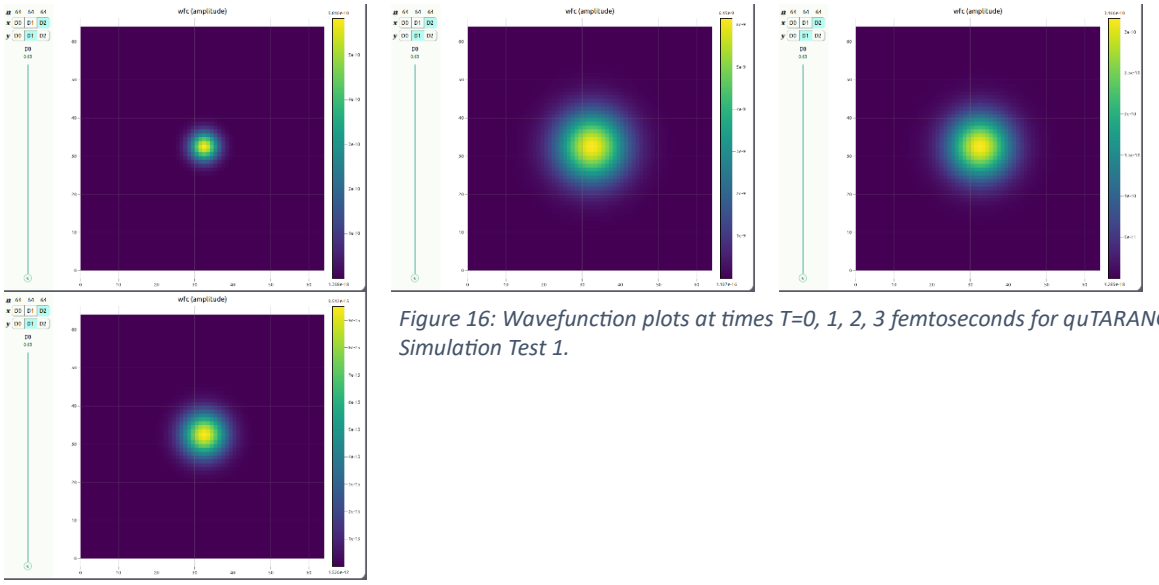


Figure 16: Wavefunction plots at times T=0, 1, 2, 3 femtoseconds for quTARANG Simulation Test 1.

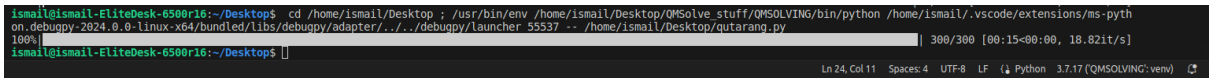


Figure 17: Evidence for quTARANG Test 1.

Test 2:

Simulation runtime: 6 fms

Time-step: 0.001

SSFT Duration: 05:12

Stable: PASS

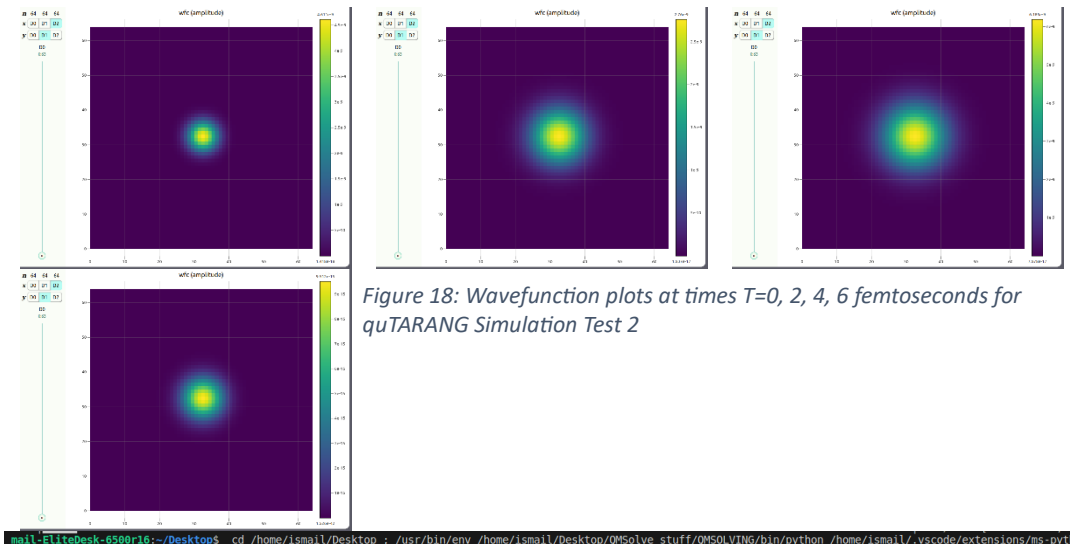


Figure 18: Wavefunction plots at times T=0, 2, 4, 6 femtoseconds for quTARANG Simulation Test 2

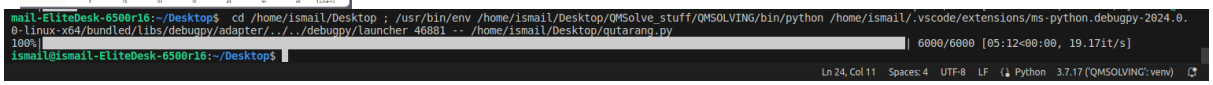


Figure 19: Evidence for quTARANG Test 2.

Test 3:

Simulation runtime: 4 fms

Time-step: 0.002

SSFT Duration: 01.48

Stable: PASS

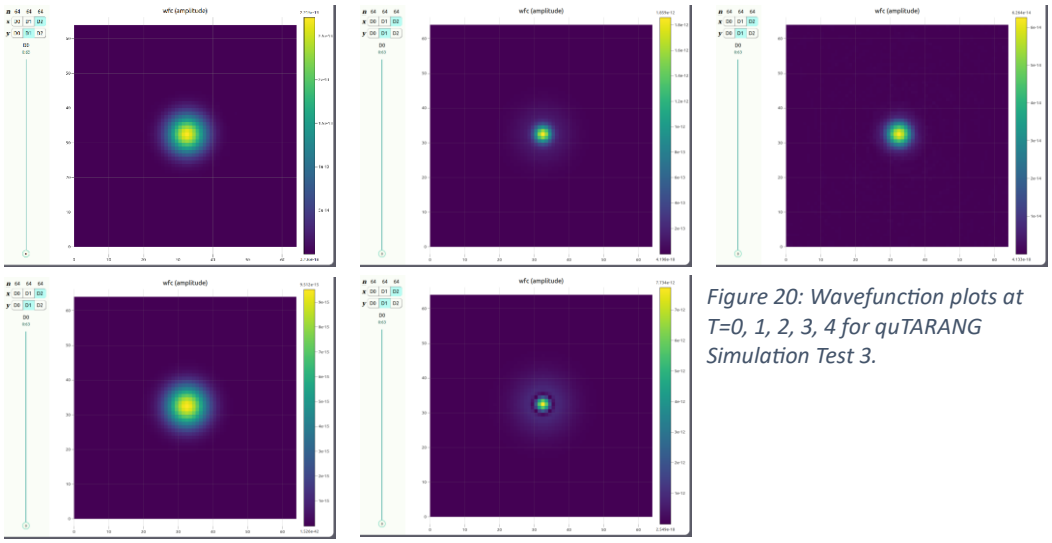


Figure 20: Wavefunction plots at T=0, 1, 2, 3, 4 for quTARANG Simulation Test 3.

Figure 21: Evidence for quTARANG Test 3.

From these tests, it is clear that the SSFT method is far faster than the Crank-Nicolson finite difference scheme, and the SSFT method has the advantage of numerical stability for the range of values I expect to be using in my simulation with a non-linear Gross-Pitaevskii equation. Hence, I will be proceeding with the SSFT as my time evolution algorithm.

I also decided to create my own example of a basic Gaussian wave packet (the code below was partly created using ChatGPT and has not formed any part of my actual program) which creates a linear space and animates a basic electron based wavepacket. This simulation aims to form a linear Schrödinger equation solver using a finite difference scheme (forward Euler method).

```
# 1. Import Libraries
import numpy as np
from scipy import integrate
from scipy.ndimage import laplace
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

# 2. Define your spatial grid and parameters

# Spatial grid
Nx, Ny = 100, 100
Lx, Ly = 10.0, 10.0
x = np.linspace(0, Lx, Nx)
y = np.linspace(0, Ly, Ny)
X, Y = np.meshgrid(x, y)

# Time parameters
T = 5.0  # Total simulation time
dt = 0.01  # Time step
Nt = int(T / dt)

# Constants
hbar = 1.0
m = 1.0
g = 1.0
V = 0.0  # Potential (can be a function of x and y)

# 3. Define the initial state δψδψ and the ground state ψ0ψ0. You can use
appropriate initial conditions for your problem.

# Set initial conditions for psi_0 and delta_psi here
# You can set psi_0 and delta_psi to represent your desired initial state
# For example, if you want a Gaussian wave packet as the initial state:
sigma = 0.2
kx = 2.0
ky = 2.0
x0 = Lx / 2
y0 = Ly / 2

psi_0 = np.exp(-((X - x0)**2 + (Y - y0)**2) / (2 * sigma**2)) * np.exp(1j * (kx * X
+ ky * Y))
delta_psi = np.random.rand(Nx, Ny) * 0.01  # Small random perturbation

# 4. Set up the numerical solver for the linearized equation using finite
differences or other methods. Here, we'll use a simple finite-difference scheme for
time evolution:
def evolve_linearized_PDE(delta_psi, dt):
    # Implement your finite-difference scheme here to update delta_psi in time
    # Example:
    delta_psi_new = delta_psi + dt * (-(hbar**2 / (2 * m)) * laplace(delta_psi) + V
* delta_psi + 2 * g * np.abs(psi_0)**2 * delta_psi)
    return delta_psi_new

# 5. Initialize arrays to store the time evolution of δψδψ:
delta_psi = np.zeros((Nx, Ny), dtype=complex)  # Initialize delta_psi with
appropriate initial conditions
delta_psi_history = np.zeros((Nt, Nx, Ny), dtype=complex)
```

Figure 22 - Simulating a basic Gaussian wave packet with the linear SE and a basic finite difference scheme.

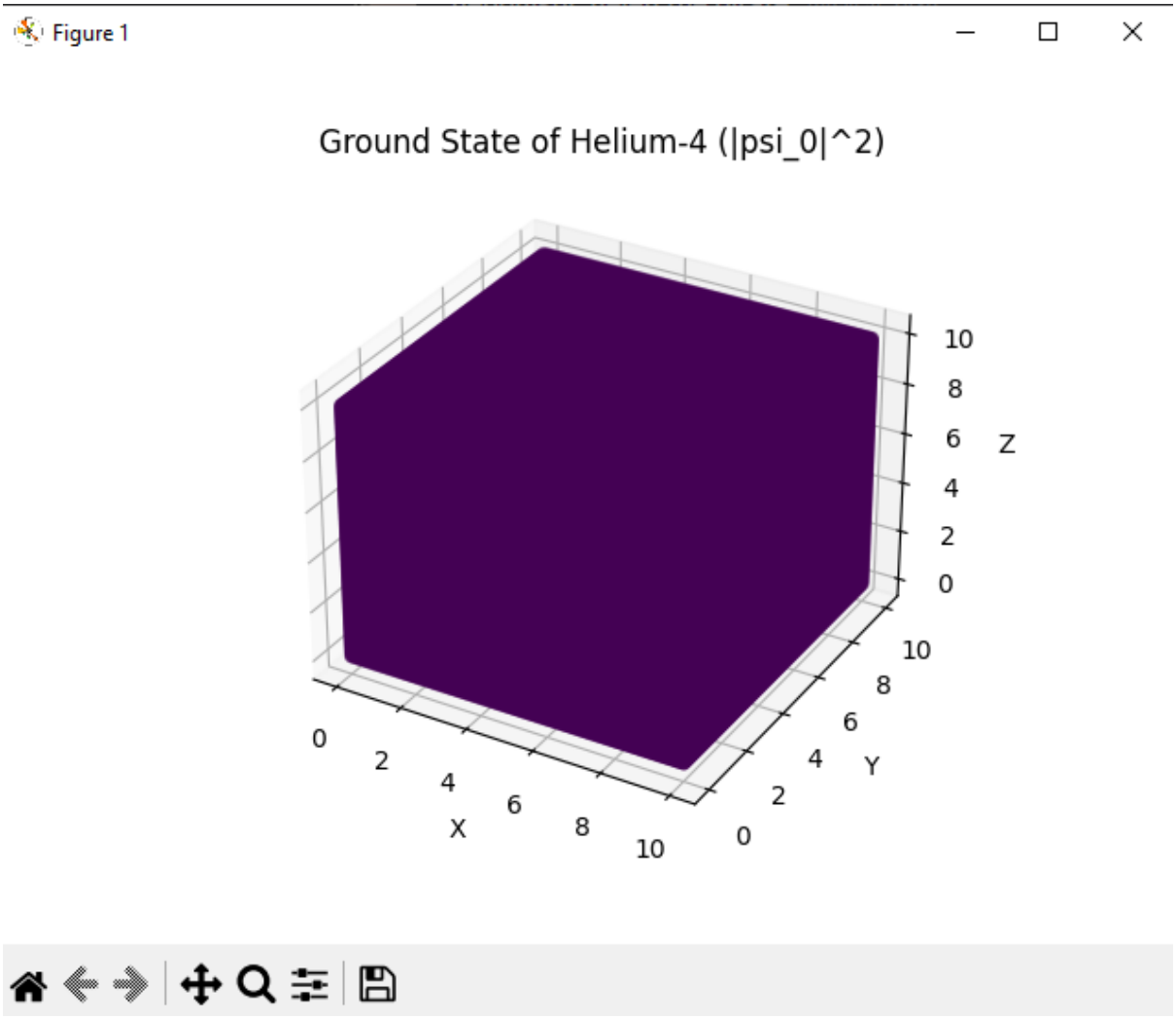


Figure 23: The 3D Linear Space with a (not visible) wave packet.

Whilst the output looks unremarkable, the steps followed are a good indication of what I will need to undergo to create my simulation. I found this program of no further use, however.

Objectives and Requirements

First, I have outlined the objectives of the program, which state in literal, concise steps what I expect it to do. The results of the program should correspond with the requirements outlined afterwards.

Objectives:
Part A: The equation

1. The program should create a two-particle system object for a Helium-4 system.
   a) Using the constant m_h (mass of helium atom), the object should be instantiated.
2. The program should formulate a Hamiltonian for a Helium-4 system as an object and use this information to create a grid and kinetic energy matrix.
   a) The user should select whether they would like to use the harmonic oscillator potential offered, and if so, input a time period and maximum amplitude of the potential. This method will require the mass of the particle from the particle system and also whether there is a two-particle system in use.
   b) The number of grid points desired should be selected by the user (128, 256, or 512). The significance of these is explained later.

c) The Hamiltonian object should be constructed using the above particle system and parameters. It should use 2 spatial dimensions with a spatial extent of 20 Angstroms.
d) The Hamiltonian constructor should store the properties of N (number of grid points), spextent (spatial extent of the helium atom), dx (spextent / N), dy (spextent / N (as grid is square and the particle has the same spatial extent in x and y)), dV (dx * dy), the type of particle system uses, the number of spatial dimensions and the number of dimensions.
e) The constructor should obtain the kinetic energy matrix by calculating the Kronecker product of the sparse diagonal matrix of the system. This requires the use of the kron function from SciPy and the Hamiltonian attribute N. This tensor product must have an absolute value of >0.
f) The constructor must now call a method to create the 1D vector arrays that will create the grid. It should create 2 1D vector arrays and then create a meshgrid using these two arrays.
g) The Hamiltonian constructor must formulate the potential as a matrix, which entails passing the particle system into the potential definition and then reshaping it into a NxN array (where N is the number of gridpoints).

3. The program should formulate an initial dissipative Gross-Pitaevskii equation for a Helium-4 system using a wavefunction Ψ_0:
   a) The program should take in a desired time step and runtime. The number of steps to make (runtime/time step) must be an integer. If not, the value of steps_to_make should be rounded.
   b) If the environment is selected as a harmonic trap, a potential ωx^2 should be added to the $V(x)$ term which will be determined by the harmonic in use. This can be stored as an array instead of the wavefunction provided it is a potential energy of the grid and not of the particle system (a pre-requisite of the SSFT method used).
   c) Using the exponential arrangement of the GPE, the initial wavefunction should be set as:
   $$k * e\text{^}(-(x1 ** 2 + 2 * y1 ** 2/(2 * c)))$$
   Where k, c are constants based on the energy the atom starts with. (This can be simplified to a function of temperature for the user. Coulomb force, magnetic fields and spin related functions are beyond the scope of my simulation).
4. The wavefunction should be established and the pre-calculation checks should be implemented (Hamiltonian constructor):
   a) Using Ψ_0, the time parameters and the Hamiltonian, define a simulation object.
   b) Using the LOBPCG module, the condition number of the determined Hamiltonian should be evaluated. If the result is over 13, the L1-NORM condition number of the matrix determines that the calculation required for the eigenstates is inefficient and so the LOBPCG method should be used to simplify this into a sparse matrix.
5. A synchronised split-step Fourier method should be implemented to determine a set of solutions:
   a) A temporary array for storing the time evolution of $\delta\Psi$ should be initialised. These must be able to store complex numbers, which can be achieved using the NumPy dtype (datatype) of complex128. The arrays from the half-steps of the program should not be stored in order to conserve memory, these have no significance to the output of the program.
   b) The Fourier transform into momentum space should be applied to the wavefunction. This is achieved using the library SciPy, function scipy.fft.fftn. This Fourier transform is 2 dimensional, but the n-dimensional operator can be used to the same effect. Store

c) Apply the propagator function to the wavefunction Ψ_0 in momentum space. This function should carry out half a timestep and yields Ψ_1.

$$\Psi_1 = e\text{^}(-0.5i * dt * V + (g|\Psi_0|^2)$$

d) Take half a timestep (kinetic propagation) using the operator for Ψ_2.

$$\Psi_2 = e\text{^}(-0.5i * dt * (\frac{d^2}{dx^2}(\Psi_0))$$

e) Return to momentum space using the SciPy fft.ifftn inverse Fourier transform, and finalise the evolution of the wavefunction by applying the final operator to obtain Ψ_3.

$$\Psi_3 = e\text{^}(-0.5i * dt * (V + g(|c_0\Psi_0 + c_1\Psi_1 + c_2\Psi_2|^2))$$

f) Using the probability distribution formula for the integral of the wavefunction, written above, normalise the wavefunction by calculating the absolute square of the wavefunction and integrating, then multiplying by the achieved normalisation constant.

6. Determine the ground state of the system by running the system in imaginary time. (optional, for eigenstate solver)

a) Define $\tau = it$ as imaginary time.
b) Begin the wavefunction iteration as outlined in step 3, substituting time for τ.
c) Keep iterating until all of the higher-energy states decay into the ground state energy function.
d) Renormalise the wavefunction between steps in order to retain the wavefunction density. Use the integral $\int_{-\infty}^{\infty}|\psi|^2\,dx = 1$ as a normalisation condition to ensure density isn't lost as higher energy states decay.

Part B: The animation

1. In order to animate the time evolution of the wavefunctions, a visualisation object must be instantiated.
   a) Create a visualisation object using the simulation object from the wavefunction evolution.
   b) The constructor of this object must have the attributes of the simulation object, the Hamiltonian object and the time step width (dt).
2. The visualisation system must plot the results of the wavefunction evolution on a grid using a suitable animation method:
   a) The parameters required in order to animate the wavefunction are: a figure size, a duration for the animation, fps (frames per second), wavefunction saturation and potential saturation. The frame should be assigned a size of (10, 10) in arbitrary units. The default animation runtime is 20 seconds. The default fps is 30 fps. The saturations are each set to 0.8 on a unitless scale of 0 – 1. These parameters do not need to be modifiable by the user as there is no feasible particle system which would require modification of these values.
   b) Calculate the number of frames required. This is the number of frames per second multiplied by the animation duration. This must be an integer, which can be an issue for float values of animation runtime. This requires rounding to an integer.
   c) Create a matplotlib figure (using matplotlib.plt) of the specified figure size and define axes using the subplot of 1, 1, 1 on the grid (puts axis at the positions of 1 from the edge of the figure).

d) The length of the plot within the graph should be the spatial extent of the simulation in metres/another SI unit. This is the spatial extent of the particle system divided by the angstrom unit.
e) Plot the potential on the grid by plotting the potential as a grid divided by the range of the potential. The maximum potential at any point in the grid is 1/the saturation.
f) From the complex results of the wavefunction, convert the complex numbers into a suitable format to equate them to HSV space. This means iteratively converting the complex values into a modulus argument form for conversion.
g) Convert all numbers iteratively into HSV colours. For a complex number of modulus MOD and argument θ, H will be equal to $\frac{\theta+\pi}{2\pi}$, and s, v are determined by MOD/psi_max.
h) Using matplotlib.hsv_rgb, convert the colours into usable RGB values. This requires placing the respective h, s and v values into an array and passing it into a function.
i) The absolute value of each complex number must be concatenated into the output array with the RGB value. This value must be capped at 1, as it has a range of 0 < abs(z) < 1 where z is the complex number being compiled.
j) Plot the wavefunction colour arrays onto the quantum configuration space using the .imshow() method, within the spatial extent of the length defined in objective 2.d.
k) Continue to iteratively plot the simulation using a method which plots frames using the method outlined in objectives A.2efghij, over the duration of the time simulation, whilst also outputting the time onto each frame.

3. The solution filled grid should be exported and displayed in the GUI.
   a) Using a suitable TKinter backend aggregator, embed the matplotlib figure into a TKinter root window.
   b) Embed the matplotlib toolbar into the TKinter window. Use a suitable frame or grid to structure this in a user operatable way.
   c) Clearly display the parameters under which the simulation is currently operating, including all user inputted parameters from objectives A.1a, A.2abcd.
   d) Include simulation controls: pause/play, save (as image and as mp4) and update simulation. The change parameters function should execute in the current window. These functions should activate in a reasonable amount of time – within 1 second of being clicked.

Part C: Interactivity (optional)

1. a) Using an adjusted potential method – insert an obstacle into the grid. Unclear how this could be achieved, to be detailed after Design.

Objectives C were scrapped as it was not clear how to go about inserting a potential into a pre-determined wavefunction. It later transpired that this was not possible in the manner I was envisioning. The possibility of methods with obstacles has been discussed with the user in the Evaluation.

My requirements:

This project must be able to do the following:

1. The simulation must be able to calculate the position of 1+ helium-4 bosons in real space and how the probability density of the position changes over time.

2. The simulation must be able to show how the bosons interact with each other, using some interaction parameter (in my method, this is achieved using the coupling constant g).

3. The simulation must be able to show on a visualisation (plot) a visual demonstration of the bosons as a density represented by a colour, the behaviour (motion and direction) of the boson.

4. The plot in this simulation must be able to pause, play, zoom, scroll and pan. The plot canvas should also be resizable.

5. The interface for this simulation should display the parameters which determine the condition of the simulation window including the runtime, simulation total time, oscillator time period, maximum gid potential amplitude, grid size, number of timesteps and particle mass.

Design

Programming Model

Each program that has been reviewed in the analysis was programmed in an object-oriented style. Initially, I considered that a functional programming technique might have been more appropriate as the SSFT method selected is an algorithm entirely composable of mathematical functions. However, immutable variables come with a significant memory trade-off, and object-oriented programming is capable of a similar modular function approach to creating the algorithm, using methods – and carries the significant advantage of encapsulation. The data that my program will need to operate on (Hamiltonians and wavefunctions) are each associated with a set of particular methods, as opposed to being raw data onto which any function can be applied. Aside from this, the reusability of these classes is critical to the simulation design, as the method is iteratively applied. This also allows the same instance of the program to run several simulations without storing the data in memory.

GUI design

The GUI requirements for this project are basic. It must incorporate the simulation window and a toolbar for controls, as well as be simple enough for a layman to operate and be able to run standalone without need for a terminal, shell or Python window.

I began by creating a mock-up of what I intended the end result to look like.

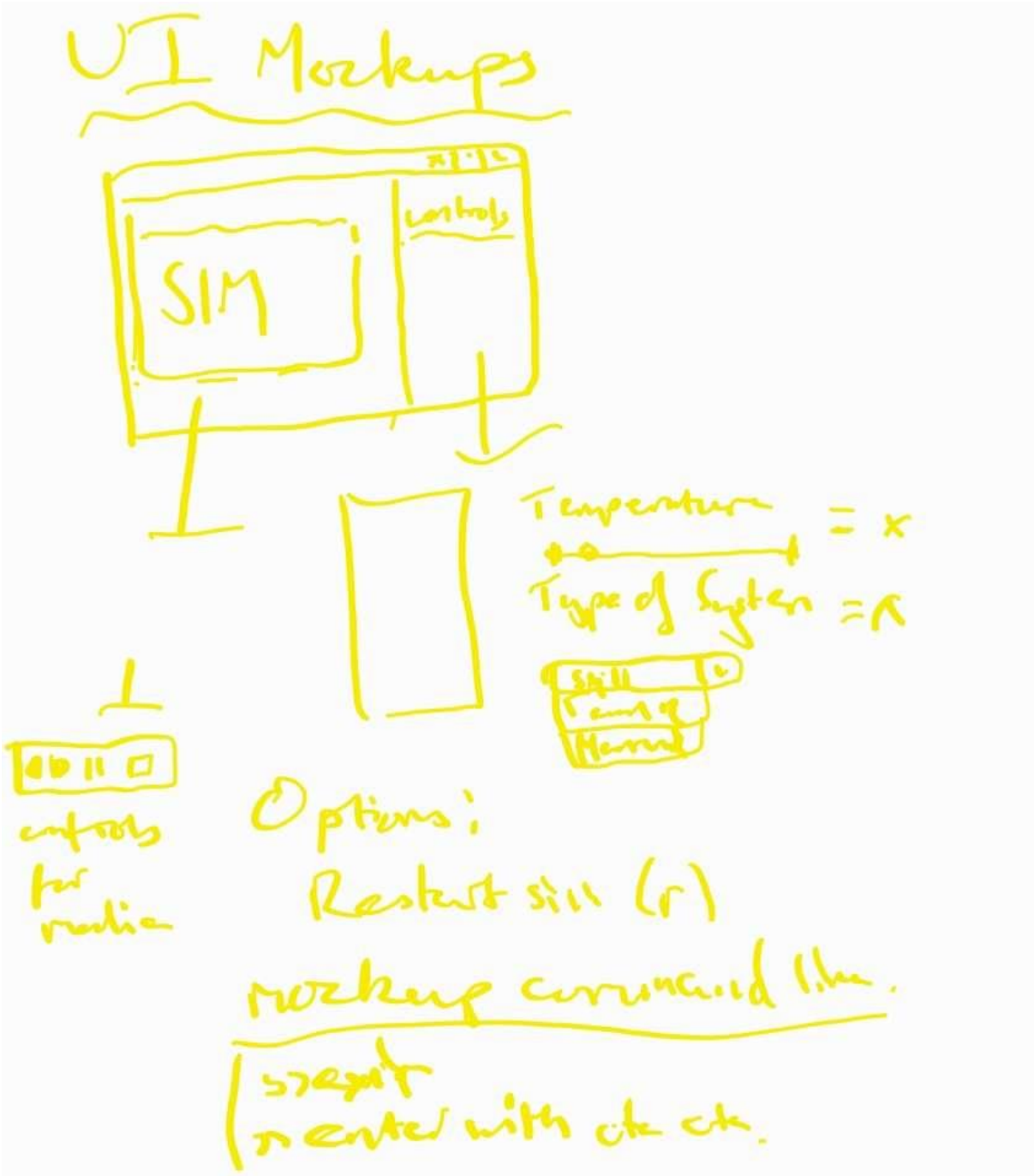


Figure 24 - A hand electronic drawing of what I expect my GUI to look like.

The first iteration of my GUI:

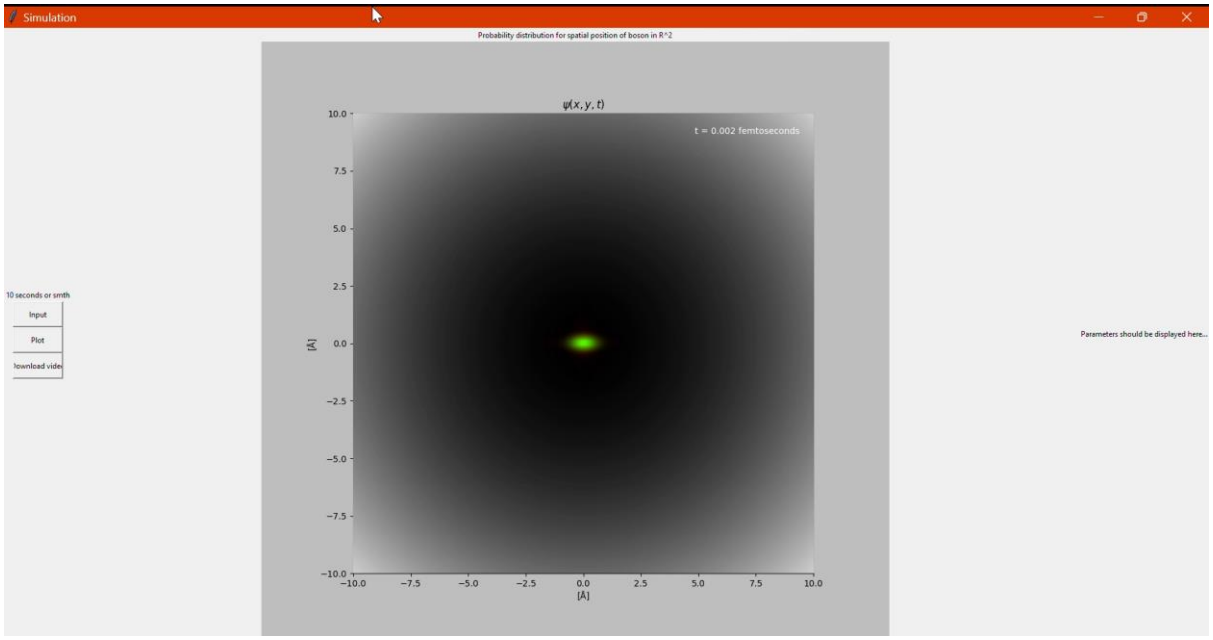


Figure 25 - The first attempt at a GUI window using TKinter.

This window is a simple TKinter window with a matplotlib figure added using a premade TKinter backend aggregator. It displays options to the left of the figure for controlling the animation and on the right, displays parameters provided to the simulation (at time of making, the simulation routine was not complete and so placeholders have been used instead). As can be seen, the text within the buttons is cut off due to the size, a significant amount of the screen is blank space, and the figure is not centred within the window. However, the figure integration works will – the window can be resized without fault and the canvas adjusts size automatically, and each button executes the method provided. Also placed on the right frame in a later version was the input dialogue menu for running another simulation, but unfortunately, I forgot to take a screenshot of this window. This version was also reviewed by the user.

Having consulted the user on the GUI, we decided that having both the input dialogues for a new simulation and the controls for the existing simulation on one window was confusing and liable to lead to errors. Mr Powell stated that it was difficult to understand which parameters corresponded to the current simulation and which parameters were inputs for the next simulation.

Below is the final version of my GUI. This includes a “Simulation” window which includes the animation and the controls to modify it, as well as a separated “Input Dialogue” menu.

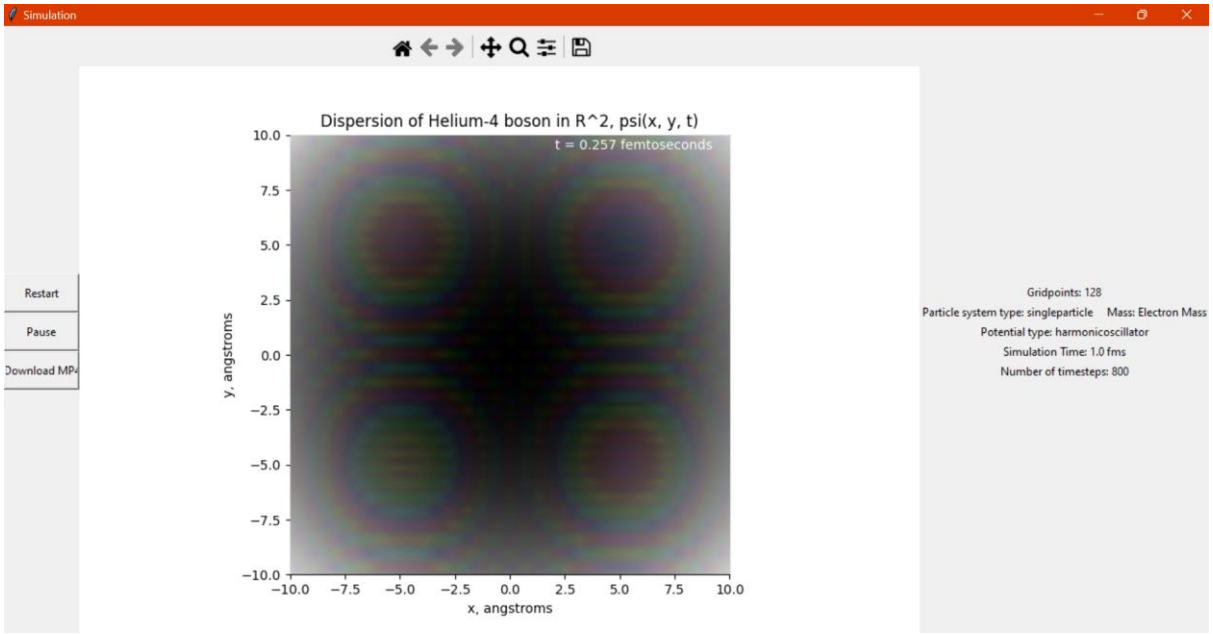


Figure 26 - The final version of my GUI, "Simulation" window.

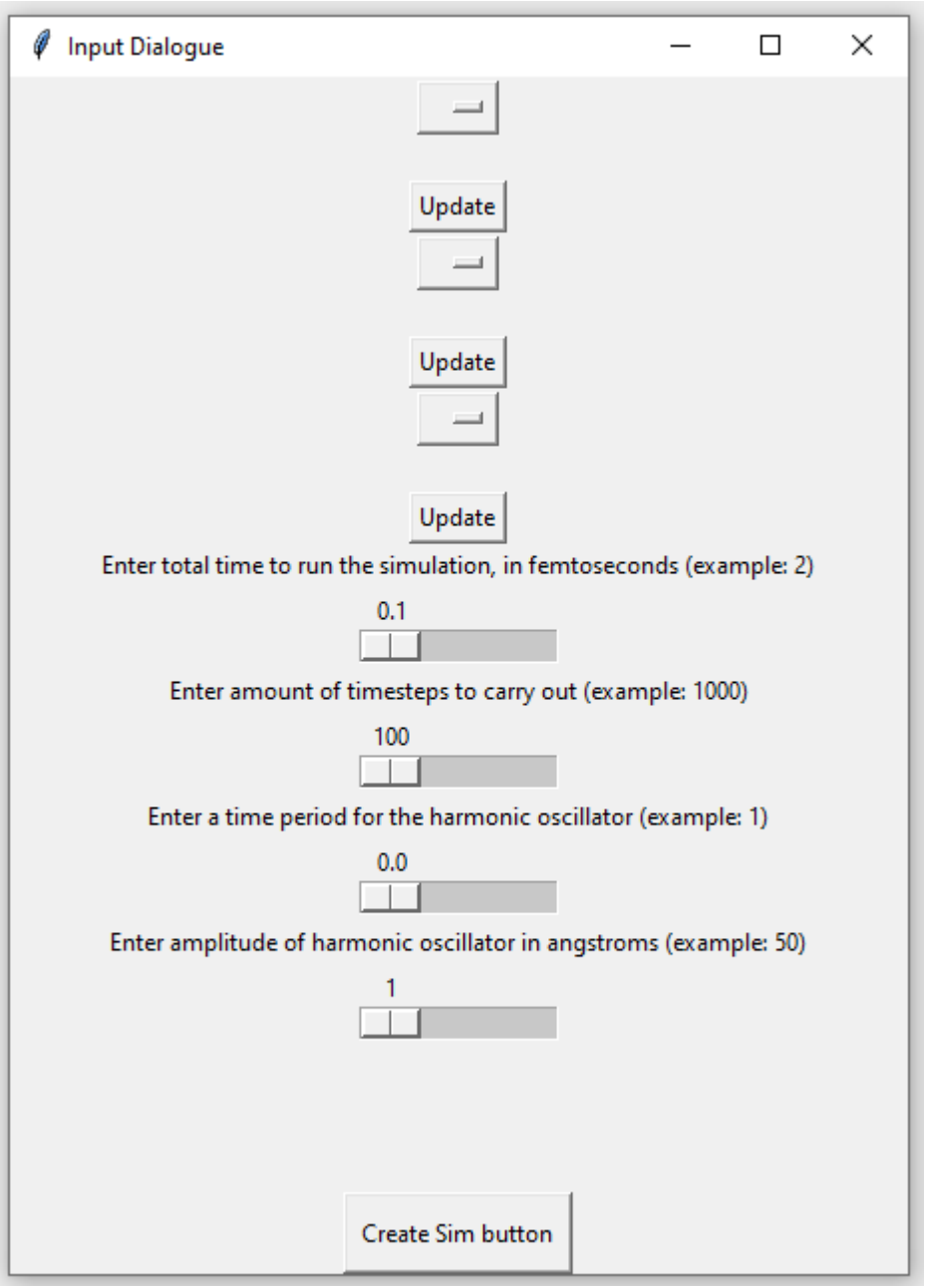


Figure 27 - Final version of the GUI "Input Dialogue".

User review

Mr Powell: I like that the input dialogue can be separated from the main window, as the amount of parameter inputs clutter the input window. This approach means that I can minimise the dialogue when not in use. I did have an issue with the scale of the sliders making it difficult to input specific values (especially for the timesteps sliders), but Ismail managed to sort this on the spot. The main window, whilst not an aesthetic masterpiece, does the job and once all the controls are working, I would have no issues with using it.

UML Class Diagram

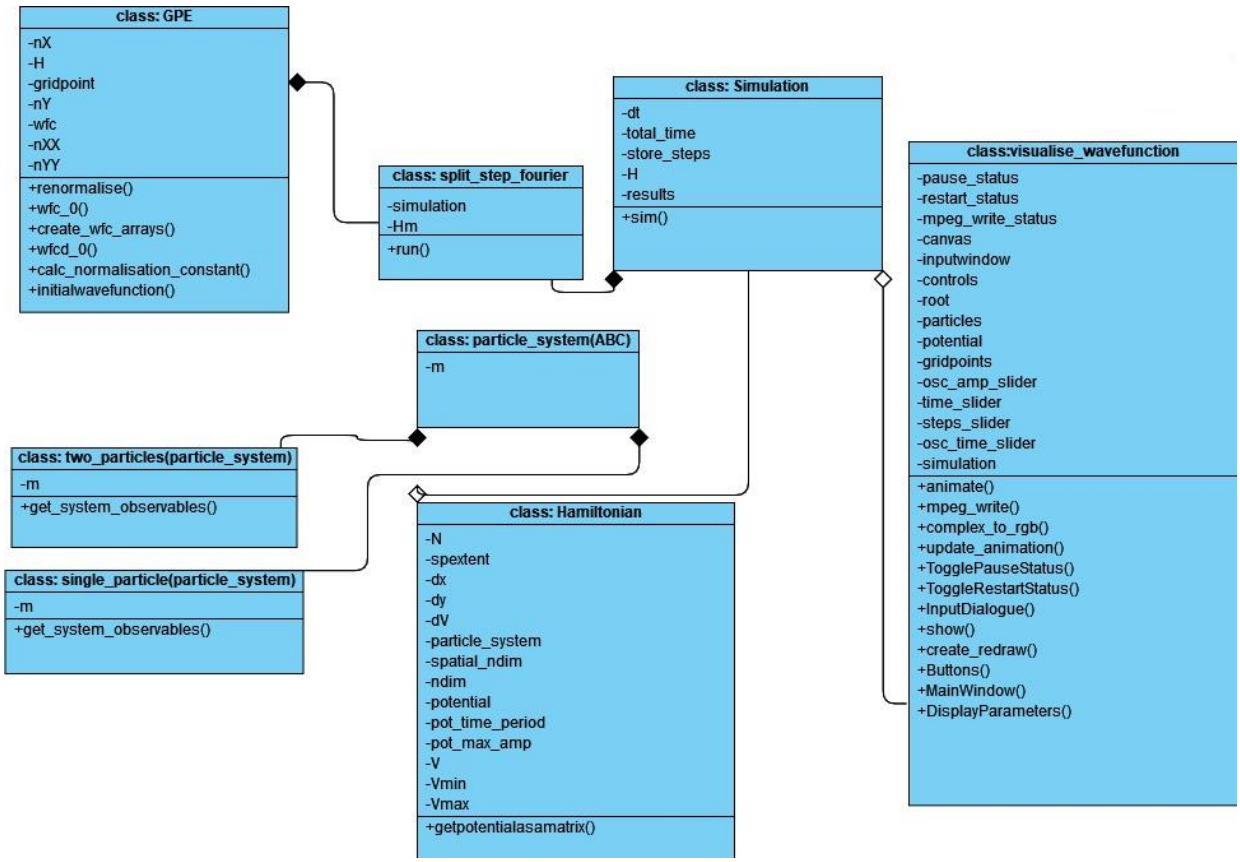


Figure 28 - UML Class Diagram for Final Program.

The above UML class diagram shows how the classes in my final project are related. I decided to separate the split_step_fourier and GPE classes despite the programming redundancy as the methods for the GPE are specific to individual wavefunctions and should not be carried forward to new wavefunctions when new simulations are created, and this limits the information provided to the system to the Simulation object, which limits the data available to the visualisation object.

Flowchart

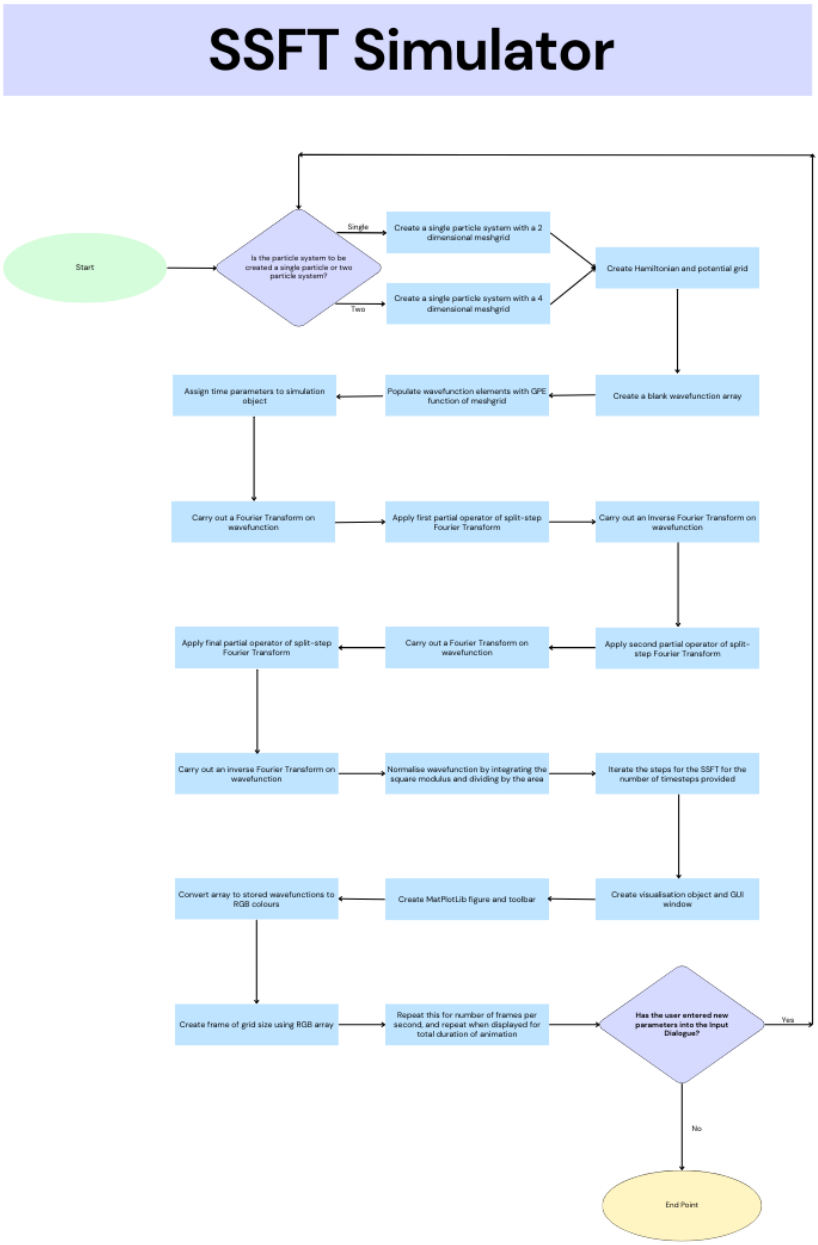


Figure 29 - Flowchart of processes requrired to generate an animation of a particle system.

Memory

The raw data that will be generated by the program for each simulation will include wavefunctions, time stepping information, energy states, a Hamiltonian, arrays of colours and particle system objects. Storing all of this data will take a significant amount of memory. As an example, I have calculated the amount of memory taken up by a single example Simulation – the parameters are taken from Black-Box Testing, Test A.

These are the only relevant variables which take up a non-negligible amount of storage space, as can be seen from the flowchart. The UML diagram also includes variables such as derivative operators and button value storage, but it can be reasonably assumed that all of these variables take up less than the space that is accounted for by the following variables. Therefore I will assume that the program will require up to double the RAM and storage availability of the below calculations.

Creating a 2D meshgrid of 512 point linear arrays for a two particle system:

A linear array of spacing N is N elements long, which with elements of NumPy datatype complex128, requires N*128 bits for each linear array. If a meshgrid combines arrays of elements N, it creates N to the power of the number of arrays being combined elements. Hence a meshgrid taking in 4 linear arrays of size 512 elements, each of size 128 bits will be:

$$512^4 \times 128 = 8.796093022 \times 10^{12}\ bits$$

which is 1099.5 GB (to 5sf). Obviously, this is beyond the capacity of the RAM of the target device. The issue here is due to the need for 4 arrays increasing the capacity required by 1024x. There is no simple rectification for this, so in order to compute two particle systems, the gridpoint size of two particle systems will be limited to 128 gridpoints, yielding a total size requirement of

$$128^4 \times 128 = 128^5 = 3.435973837 \times 10^{10}\ bits$$

which is 4.2950 GB (to 5sf). As the RAM of the target device is 8GB, this represents a suitable proportion of the RAM likely to be available.

Clearly, this is an unreasonable amount of data to feasibly retain, especially if the user wants to compare multiple simulations. To avoid this need, and to benefit from the reusability of the OOP design chosen, I have included a file saving method which retains all the information relevant to the user whilst vastly reducing the amount of space needed.

Python:

```
    def mpeg_write(self): # writes video to mped
        self.mpeg_write_status = not self.mpeg_write_status # set mpeg write status
        self.ToggleRestartStatus()
        if self.pause_status:
            self.TogglePauseStatus()
        Writer = animation.writers['ffmpeg'] # use ffmpeg as the writer
        writer = Writer(fps=30, bitrate=1800) # write file using given parameters
        self.a.save(r"SAVE_PATH"+str(self.download_count)+".mp4", writer=writer) # write animation to file
using defined writer
        self.download_count += 1
        self.mpeg_write_status = not self.mpeg_write_status # unset mpeg write status
```

Typical RAM usage of a likely simulation input (parameters from Black-Box Test B)

A linear array of spacing N is N elements long, which with elements of NumPy datatype complex128, requires N*128 bits for each linear array. If a meshgrid combines arrays of elements N, it creates N to the power of the number of arrays being combined elements. Hence a meshgrid taking in 2 linear arrays of size 256 elements, each of size 128 bits will be:

$$256^2 \times 128 = 8.388608 \times 10^6\ bits$$

which is 1.04 MB (to 3sf). Clearly this is a far smaller swap requirement and as this array is constantly worked on, this will remain in swap space for the majority of the program calculation. The wavefunctions will occupy space in RAM individually, but stored steps will require secondary storage space.

The size of a wavefunction (for a single particle system) is N^2, and each element is also 128 bits, so another 1.04 MB is required in RAM, and the number of steps (in this case 1000) multiplied by this figure is required, 1.04 GB of secondary storage is required for the outputted evolved wavefunctions.

The file storage required by the RGB arrays is limited as the animation function reuses the RGB colour conversion method as opposed to storing frames. As these calculations are limited to relatively simple numerical operations (see Design – complex-to-rgb), these calculations are computed instantly which is more efficient than saving frames. The space occupied by a single array of RGB colours for a single frame of the animation is:

$$bit\ size\ of\ value \times 3\ (number\ of\ values\ in\ each\ array) \times\ N^2$$

$$16 \times 3 \times\ 256^2 = 393\ \text{kB}$$

Each calculation proceeds in a similar way for each use cae, and so I've summarised the requirements for the program for an ideal case and for a worst case (in terms of performance).

Ideal Case (Test B scenario):

Variable	RAM	Storage
Wavefunction	negligible	1.04 MB
Meshgrid/Potential	1.04 MB	1.04 MB
RGB arrays	negligible	393kB
Total	1.04 MB	2.4 MB

Worst Case (2 particle scenario):

Variable	RAM	Storage
Wavefunction	negligible	2.08 MB
Meshgrid/Potential	4.23 GB	2.08 MB
RGB arrays	negligible	1.3 MB
Total	4.23 GB	5.5 MB

As can be seen, the requirements for the program greatly increase, especially in terms of RAM requirements for two particle systems, and the increase in storage and RAM for grid size increases is negligible. However, increasing the grid size will greatly increase the processing time as there is an exponential relationship between increasing the grid size and the resulting amount of element calculations to perform. However, provided the 2 particle simulations are limited to 128 gridpoints, the simulation uses far less storage and RAM than the target device (school-issue laptop with 8GB RAM, 10th generation Intel i5 processor) is capable of providing. Processing time does increase greatly with 512 gridpoint simulations and simulations with higher numbers of timesteps, but testing has shown that this is not an issue up to the allowed values (allowed values were determined by the testing).

Algorithms

Linear space

The first of the algorithms necessary will be creating the grid necessary to describe the space the particle system will be in. This must consist of an N point, 2D space where a co-ordinate system can be used to describe points in the system. The following is a representation of how this might look using two NumPy functions which create evenly spaced arrays and combine arrays into co-ordinate systems respectively.

Pseudocode:

 # linspace is a numpy function which returns an evenly spaced array of *spacing* points between x, y
x1 <- linspace(-spatial_extent/2, spatial_extent/2, spacing = N)
x2 <- linspace(-spatial_extent/2, spatial_extent/2, spacing = N)
y1 <- linspace(-spatial_extent/2, spatial_extent/2, spacing = N)
y2 <- linspace(-spatial_extent/2, spatial_extent/2, spacing = N)
 # meshgrid is a numpy function which combines array spaces into co-ordinate spaces.
gridspace <- meshgrid(x1, x2, y1, y2)
gridpace = [x1,y1,x2,y2]
This pseudocode is very similar to the real Python method to achieve the same objective. This method requires access to the Hamiltonian object as the number of grid points of the system is a Hamiltonian attribute.

Python:

```
    def get_system_observables(self, H): # returns a meshgrid which functions as a composition of the linear spaces needed to represent the system
        #create meshgrid of space equivalent to hamiltonian based on 2D - 4 1D arrays for 2D, each.
        x1 = np.linspace(-H.spextent/2, H.spextent/2, H.N)
        y1 = np.linspace(-H.spextent/2, H.spextent/2, H.N)
        x2 = np.linspace(-H.spextent/2, H.spextent/2, H.N)
        y2 = np.linspace(-H.spextent/2, H.spextent/2, H.N)
        self.x1, self.y1, self.x2, self.y2 = np.meshgrid(x1, y1, x2, y2) # The meshgrid is a co-ordinate matrix of the numpy arrays provided.
```

The outcome of this algorithm is a linear space of x and y, which either one or two particles can be described by. One issue posed by the 4-dimensional linear space is the memory occupied. The size of the space in memory grows by a factor of $O(n^4)$ which means that it is unreasonable to create a large grid for a two-particle system. An example of how this file size would grow rapidly is included in the Further Testing section.

An alternative to this would have been to use smaller values for the grid sizes, but at any power of 2 below 128, the grid squares would be visible and the animation quality unsatisfactory (as the resolution reduces with factor $O(n^2)$ for each reduction in grid length).

Fourier Transform

A Fourier Transform transforms a vector in the real domain into the frequency domain using a root of unity. The particular Fourier Transform that I will be using is the Cooley-Tukey FFT algorithm which performs recursive DFTs on a composite $N = N_1N_2$. This has the effect of reinterpreting a 1D DFT of

size N as a 2D DFT of size N_1N_2, meaning that small DFTs can be performed before multiplying by a phase factor (complex roots of unity) before performing DFTs in the other direction. The result is the transpose of the DFT desired. Fortunately, this can all be achieved in a single statement using NumPy.

Pseudocode not provided, not necessary as using a library method.

In order to achieve this in Python, I am using the NumPy fft.fftn method.

Python:

```
np.fft.fftn(psi)
```

Time Evolution

The time evolution of the wavefunction is accomplished using a split operator method which requires forming initial operators using the wavefunction, completing half a timestep in real space, applying a full time step in momentum space before transferring the wavefunction back to the real domain and applying the final half-timestep operator.

Pseudocode:

```
for stepnumber <- (0, steps_to_make)
   temp_psi_store <- []
   temp_psi_store[0] = psi[stepnumber]
   operatorone <- e^(-0.5i*dt*(potential+(g*MOD(temp_psi_store[0]^2))))
   operatortwo <- e^(0.5i*dt)*(laplace(temp_psi_store[0]))
   temp_psi_store[1] <- (operatorone*(fftn(temp_psi_store[0])))
   temp_psi_store[2] <- (operatortwo*(inv_fftn(temp_psi_store[0])))
   operatorthree <- e^((-
0.5i*dt*(potential+g*MOD(c_0*temp_psi_store[0]+c_1*temp_psi_store[1]+c_2*temp_psi_store[2
])^2)))
   temp_psi_store[3] <- (operatorthree*(fftn(temp_psi_store[1])))
   psi[stepnumber] <- temp_psi_store[3]
```

Python:

```
    def run(self, wfc, Hm): # carries out the time evolution
        self.Hm = Hm # assigns the Hamiltonian attribute
        steps_to_make = self.simulation.store_steps # takes in the number of steps to carry out from the
simulation
        dt = self.simulation.dt # takes in the step width to use from the simulation
        total_time = self.simulation.total_time # takes in the total time to evolve over from the simulation
        dt_store = total_time/dt # the number of steps to take by width
        dt = dt_store # time step width
        psi = [wfc] # import initial wavefunction
        c_0 = 1 # constant for split-operator approximation to GPE
        c_1 = -c_0 # constant for split-operator approximation to GPE
        c_2 = 1 # constant for split-operator approximation to GPE
        g = 4*np.pi*hbar**2*a_s/self.Hm.particle_system.m # coupling constant
        potential = self.Hm.V # takes the potential matrix from the Hamiltonian

        for i in range(0, steps_to_make): # iteration for number of steps to carry out
```

```
        TPSI = [] # temporary array to store wavefunctions being manipulated
        TPSI.append(psi[i]) # add the current array psi into TPSI

        operatorone = np.exp(-0.5j*dt*(potential+(g*np.abs(TPSI[0]**2)))) # first operator, for position space
        operatortwo = np.exp(0.5j*dt)*(ndimage.laplace(TPSI[0])) # second operator, for momentum space

        TPSI.append(operatorone*(np.fft.fftn(TPSI[0]))) # TPSI[1] is psi_1, the first wavefunction from the partial operator application, here psi is stored in momentum space using the fft.fftn n-dimensional fourier transform
        TPSI.append(operatortwo*(np.fft.ifftn(TPSI[1]))) # TPSI[2] is psi_2, the second wavefunction from the partial operator application, here psi is stored in real space using the fft.ifftn n-dimensional inverse fourier transform

        operatorthree = np.exp((-0.5j*dt*(potential+g*np.abs(c_0*TPSI[0]+c_1*TPSI[1]+c_2*TPSI[2])**2))) # the third operator is now built - as it uses psi_1 and psi_2 as part of its definition, the first two operations had to be carried out first.

        TPSI.append(np.fft.ifftn(operatorthree*(TPSI[1]))# the third operator can now be applied to the psi_1 wavefunction
        psi.append(TPSI[3]) # appends the final wavefunction to the psi array
    return psi # returns psi - the array full of stored steps
```

Commonly, time-stepping algorithms include the ability to store only a fraction of the steps taken, but my algorithm does not do this as not storing the step will result in it not being animated, and for the purpose of this simulation, animation quality is more relevant than scientific precision (to an extent). As there is therefore no scenario in which a step would be taken but not animated in order to save time, my method does not have this ability.

Normalising wavefunction

The wavefunction will lose density as it is evolved, in order to retain the entire system, the normalisation routine will ensure that the area under the square modulus of the wavefunction is equivalent to 1 (explained in Analysis – Research: Physics, wavefunctions). Within my analysis, the use of a different normalisation constant such as an L2-NORM was explored. This has the benefit of conditioning the Hamiltonian matrix in order to reduce the compute time of the evolution algorithm. However, I found this to be unnecessary as the Hamiltonians constructed by the program are all Hermitian in nature due to the use of a square grid and system, and an entirely symmetrical system (no spin or momentum to consider). Hence, any Hamiltonian matrix created would already be well conditioned enough to proceed directly to calculation.
Pseudocode:

```
Subroutine renormalise(normconstant, wfc):
  wfc /= normconstant
  return wfc

Subroutine calc_normalisation_constant(Hm, psi):
  n_c = Hm.dV * total(MOD(psi))^2 # The probability of the particle being in the real space is 1.
  return n_c

# The following should be in the previous time evolution loop.
norm_const <- calc_normalisation_constant(H, temp_psi_store[i])
psi[i] <- renormalise(norm_const, temp_psi_store[i])
```

Python:

```
        norm = GPE.calc_normalisation_constant(self.Hm, TPSI[3]) # uses the GPE method to calculate a normalisation constant to the wavefunction.
        final_psi = GPE.renormalise(norm, TPSI[3]) # uses the GPE method to normalise the final wavefunction

# GPE methods
    def calc_normalisation_constant(Hm, psi): # normalisation constant of wavefunction
        n_c = Hm.dV * (np.sum(np.abs(psi)))**2 # The probability of the particle being in the real space is 1.
        return n_c

    def renormalise(normconstant, wfc): # normalisation routine - tales in the calculated normalisation constant and divides wfc by it
        wfc /= normconstant
        return wfc
```

Complex to HSV colour

The time evolution scheme outputs a series of arrays which are populated with complex numbers. These complex numbers represent the density of the system occupying the grid co-ordinate. In order to represent these arrays on the graph, these values must be converted to colours. As discussed in **[50]**, complex numbers map directly to the HSV colour space, and so in order to get to the RGB colour space, the complex values can be first converted to RGB using the following algorithm,

The pseudocode algorithm for converting a complex number of modulus MOD and argument θ into HSV colour is:

Pseudocode:

```
Z <- x + iy
mod <- sqrt(x**2+y**2)
arg <- arctan(y/x)
h <- (arg + pi) / (2 * pi)
s <- (mod/self.simulation.psi_max), type array[N,N]
v <- (mod/self.simulation.psi_max), type array[N,N]
# matplotlib hsv_to_rgb routine applied to array [h,s,v]
abs_z <- mod / wfc_saturation # Takes the wavefunction saturation from the function call and sets the maximum value of the magnitude of the complex value to the
if abs_z > 1:
   abs_z <- 1
return [rgb, abs_z]
```

Python:

```
def complex_to_rgb(self, Z: np.ndarray, max_val: float = 1.0) -> np.ndarray:
    mod = np.abs(Z) # takes modulus of complex value
    arg = np.angle(Z) # takes argument of complex value
    h = (arg + np.pi) / (2 * np.pi) # hue value determined by polar mapping of argument
    s = np.reshape((mod/self.simulation.psi_max), (N, N))
    v = np.reshape((mod/self.simulation.psi_max), (N, N))
```

```
    rgb = hsv_to_rgb(np.moveaxis(np.array([h,s,v]) , 0, -1)) # moves h, s, v into an array, rearranges the axis and uses the hsv_to_rgb routine from matplotlib to convert the HSV value obtained into an RGB value.

    abs_z = mod / max_val # Takes the wavefunction saturation from the function call and sets the maximum value of the magnitude of the complex value to the
    abs_z = np.where(abs_z> 1., 1. ,abs_z) # where the magnitude of the complex number is greater than 1, replace abs_z with 1
    return np.concatenate((rgb, abs_z.reshape((*abs_z.shape,1))), axis= (abs_z.ndim))
```

This method was later adjusted as despite the colour accuracy benefits, the "washed-out" appearance of the colours was an issue for the user, as projecting the simulation causes the colours to become difficult to distinguish between. Hence the saturation was set to 1, its maximum value.

Harmonic Oscillator Potential

The split-operator method for solving a time-dependent non-linear Schrödinger equation only supports potentials defined in terms of the grid, as it isn't possible to apply a matrix potential term into the wavefunction. Therefore, a harmonic oscillator is a good, simple alternative which can oscillate the grid potential constantly throughout the simulation.

Pseudocode:

```
boson_mass = 4.002602 * au
mass <- boson_mass
T <- 1 fms
w = 2pi/T
A = 44 * angstroms
for grid <- (0, length(grid))
    PE[grid] = 0.5 * (A*x1)^2 + 0.5 * (A*y1)^2
```

Python:

```
def harmonicoscillator(particle_system):
    m = particle_system.m # mass of particle
    T = 1 * femtoseconds # time period, Enter a suitable time period for the harmonic oscillator in femtoseconds
    w = np.pi*2/T # omega
    A = 44 * Å # max magnitude of the oscillation
    PE = 0.5 * A**2 * particle_system.x1**2 + 0.5 * A**2 * particle_system.y1**2 # total potential energy from position in harmonic oscillator
    return PE
```

The amplitude and time period of the oscillator here are examples, these are user determined adjustable parameters. If the system is a two-particle system, it must also assign a potential to the grid points of x2 and y2, where x2 and y2 are the vectors of the second particle system linear space.

The potential is a method, which is useful as other types of grid potential could be added in the future.

Visualisation

In order to visualise arrays (which have now been obtained as colours), a function to animate the colours is required. Matplotlib natively includes a method called FuncAnimation which repeatedly calls a function to draw a frame of the animation. [51]

class matplotlib.animation.FuncAnimation(*fig*, *func*, *frames=None*, *init_func=None*, *fargs=None*, *save_count=None*, ***, *cache_frame_data=True*, ***kwargs*)

The arguments this function requires are: a figure to plot onto, a function to draw the frames to plot, a number of frames to calculate and optional arguments for passing in multiple different function, save data or metadata (which I do not require). My func_animation will be very similar to the one previously seen in QMSolve, as I am plotting a very similar type of data onto the same type of figure.

Pseudocode: Not included, the algorithm is simply limited to applying a series of other functions, pseudocode was not necessary.

Python:

```
    def func_animation(*arg): # increments the time, and calculates the next frame to display by calculating a frame for each time step, sim_dt. Used for the FuncAnimation call
        if self.restart == True:
            self.restart = False
            animation_data['t'] = 0.0 # reset the animation time to 0
        if not self.pause_status: # stops next frame being calculated when pasued
            time_ax.set_text(u"t = {} femtoseconds".format("%.3f"  % (animation_data['t']/femtoseconds))) # update time axis value

            animation_data['t'] = animation_data['t'] + sim_dt # increment time
            if animation_data['t'] > self.simulation.total_time:
                animation_data['t'] = 0.0 # replays simulation when toal time reached

            animation_data['frame'] +=1 # increments frame number
            index = int((self.simulation.store_steps)/self.simulation.total_time * animation_data['t']) # t/dt

            plot_wavefunction.set_data(self.complex_to_rgb(self.simulation.psi_plotarea[index], max_val=wavefunction_saturation)) # the potential of the wavefunction is plotted over the plot area
            return plot_potential, plot_wavefunction, time_ax
        else:
            return plot_potential, plot_wavefunction, time_ax
```

Validation

As this project is centred around a simulation, there are several inputs to take in for parameters, each of which must be validated. Each input must be checked for its data type, and that it is within a suitable range for the parameter it describes.

Numerical inputs:

For the number of grid points to split the simulation into, using a power of 2 is both convention and convenient for the scale of the simulation space. The usual powers used are 2^7 up to 2^10 (128, 256, 512 or 1024). The larger the value picked, the more calculations the simulation will have to perform by a factor of $O(n^2)$ for 2D simulation. Whilst it is theoretically possible to include other values both higher and lower, or to allow for unconventional grid sizes – these values will improve calculation times as the size of each array will be a power of 2 meaning that every value in the program will be stored in a size directly computable. If the values were in a power of 3, for example, each operand would have to be stored in a similarly sized matrix which would require an extra half-filled array, therefore increasing the compute time.

Non-numerical inputs:

The non-numerical inputs for this program are the type of potential, the mass of the particle(s) of the system (as the mass is one of two options) and the type of particle system. These options can be validated by taking in an input based on a text input and then using a loop to indefinitely iterate the prompt until a valid input is received.

I decided to remove the ability to select the particle mass as it does not affect the output, except for scale and too many input options were making the input window intimidating to the user.

Python:

```
# Particle System START

# Validated input for particle system type - s is a single particle system, d is a dual particle, non-interacting system and ds is an interacting two particle system.
validparticleinput = False
while not validparticleinput:
    validparticleinput = True
    psysteminput = input("Please select the type of particle system desired: single particle (s), dual particle (d) or two interacting particles (ds): ")
    # Selects type of particle system based on user input
    if psysteminput == "s":
        particlesystem = singleparticle
    elif psysteminput == "d":
        particlesystem = twoparticles
    elif psysteminput == "ds":
        particlesystem = twoparticles
    else:
        print("Invalid input, enter only s, d or ds in lowercase.")
        validparticleinput = False

# returns only s, d or ds into particlesystem

# Validated input for mass of particle system - m_e is the mass of an electron, for wavepacket simulatons and m_h is the mass of a helium atom for superfluid simulations.
validmassinput = False
while not validmassinput:
    validmassinput = True
    mass = input("Enter either m_h (mass of helium) or m_e (mass of electron) for the particle system: ")
    if mass == "m_e":
       mass = m_e
    elif mass == "m_h":
        mass = m_p #NEEDS TO BE M_H
    else:
        print("Invalid input, enter only m_e or m_h in lowercase.")
        validmassinput = False
 # returns m_e or m_h only


# Instantiation of particle system
```

```
sys = particlesystem(mass)

print("Done")

# Particle System END
# Validated input for grid potential
validpotentialinput = False
while not validpotentialinput:
    validpotentialinput = True
    potentialtype = input("Enter either hm (harmonic oscillator) or none for the potential type: ")
    if potentialtype == "hm":
        potentialtype = harmonicoscillator
    elif potentialtype == "none":
        potentialtype = nopotential
    else:
        print("Invalid type of potential, please type either hm or none.")
        validpotentialinput = False


#Hamiltonian START

# Validated input for grid potential and parameters
validpotentialinput = False
while not validpotentialinput:
    validpotentialinput = True
    potentialtype = input("Enter either hm (harmonic oscillator) or none for the potential type: ")
    if potentialtype == "hm":
        potentialtype = harmonicoscillator
    elif potentialtype == "none":
        potentialtype = nopotential
    else:
        print("Invalid type of potential, please type either hm or none.")
        validpotentialinput = False

if potentialtype == harmonicoscillator:
    # time period of oscillator
    validtimeperiodinput = False
    while not validtimeperiodinput:
        validtimeperiodinput = True
        potentialtime = input("Enter a time period for the harmonic oscillator, the unit is fms. Alternatively,
type default. :")
        if potentialtime == "default":
            potentialtime = 2
        else:
            try:
                int(potentialtime)
            except ValueError:
                print("Please enter a numeric value with no letters or symbols.")
                validtimeperiodinput = False
            else:
                potentialtime = int(potentialtime)
                if potentialtime <= 0:
                    print("Enter a positive value for the time period.")
                    validtimeperiodinput = False
    # amplitude of oscillator
```

```
    validamplitudeinput = False
    while not validamplitudeinput:
        validamplitudeinput = True
        potentialamp = input("Enter a maximum amplitude for the harmonic oscillator, the unit is angstroms.
Alternatively, type default. :")
        if potentialamp == "default":
            potentialamp = 50
        else:
            try:
                int(potentialamp)
            except ValueError:
                print("Please enter a numeric value with no letters or symbols.")
                validamplitudeinput = False
            else:
                potentialamp = int(potentialamp)
                if potentialamp <= 0:
                    print("Enter a positive value for the maximum amplitude.")
                    validamplitudeinput = False

# sets harmonic oscillator variables to 0 if not in use.
if potentialtype == nopotential:
    potentialtime = 0
    potentialamp = 0

# Validated gridpoints input
validgridpoints = False
while not validgridpoints:
    validgridpoints = True
    gridpoints = input("Enter either 128, 256 or 512 for the number of gridpoints to use for the quantum
configuration space. :")
    try:
        int(gridpoints)
    except ValueError:
        print("Enter one of the three values as a number, eg 256.")
        validgridpoints = False
    else:
        gridpoints = int(gridpoints)
        validoptions = [128,256,512]
        if gridpoints not in validoptions:
            print("Enter one of the three options given.")
            validgridpoints = False

# Validated spatial extent input
validspaceext = False
while not validspaceext:
    validspaceext = True
    spaceext = input("Enter an integer number between 5 and 40 for the spateial extent of the particle
system in angstroms, the default for the spatial extent of a 2 particle helium-4 composite boson system is
20. : :")
    try:
        int(spaceext)
    except ValueError:
        print("Enter a numeric integer value only.")
        validspaceext = False
    else:
```

```
        spaceext = int(spaceext)
        if spaceext < 5:
            print("Enter an integer of at least 5. ")
            validspaceext = False
        if spaceext > 40:
            print("Enter an integer below 40. ")
            validspaceext = False

print("Creating Hamiltonian...")
# Instantiate Hamiltonian object with input parameters, in 2 dimensions.
H = Hamiltonian(sys, potentialtype, potentialtime, potentialamp, gridpoints, spaceext*Å, 2)
print("Done")

# Hamiltonian END

# wavefunction START

print("Formulating initial wavefunction...")
# Instantiate GPE wavefunction based on grid size of Hamiltonian.
wfc = GPE([H.N, H.N], H.N)
psi = wfc.initialwavefunction(H) # formulate psi_0
print("Done")

# wavefunction END

# simulation START



# Validated time input
validduration = False
while not validduration:
    validduration = True
    total_time = input("Enter a number between 0 and 25 femtoseconds for teh duration of the simulation.
:")
    try:
        float(total_time)
    except ValueError:
        print("Enter a numeric value only.")
        validduration = False
    else:
        total_time = float(total_time)
        if total_time <= 0:
            print("Enter a value greater than 0.")
            validduration = False
        if total_time > 25:
            print("Enter a value below 25.")
            validduration = False


# Validated store steps input
validsteps = False
while not validsteps:
    validsteps = True
```

```
    store_steps = input("Enter an integer number between 100 and 10000 for the number of timesteps to
make. :")
    try:
        int(store_steps)
    except ValueError:
        print("Enter an integer numeric value only.")
        validsteps = False
    else:
        store_steps = int(store_steps)
        if store_steps <= 100:
            print("Enter a value greater than 100.")
            validsteps = False
        if store_steps > 10000:
            print("Enter a value below 10000.")
            validsteps = False

print("Beginning time evolution...")
simulationone = Simulation(H, total_time, store_steps)
evolvedwfcs = simulationone.sim(psi, H)
print("Done")

# simulation END

# visualisation START

graph = visualise_wavefunction(simulationone)
print("Animating...")

graph.animate((10, 10), 10, 30, 0.8, 0.8)
```

Clearly type and range validation is possible for each input, but it is bulky, time-consuming and not suitable for the end user. For my final simulation, I will use an input window to take in parameters.

In order to circumvent the need for this validation, a dropdown menu is utilised. This menu will physically limit the possible input values. As I will be using the TKinter GUI, I will be using the dropdown utility OptionMenu from TKinter to create a menu. Using the TKinter Scale widget, the need to validate numerical inputs can also be avoided, provided a suitable start, end and resolution is passed to the widget.

Interactivity/Controls

One issue with TKinter buttons is that they process the command they are assigned to on packing, and do not support arguments passes into the function. In order to avoid this, I have used a method which packs a new button into the Input Dialogue which has the data from all the inputs packed into a partial function.

The MatPlotLib toolbar is packed at the top of the window, and allows for: back/forward, zoom, pan, save snapshot and adjusting subplots.

Testing

The testing of this program is split into three objectives – operation, efficiency and reliability. The functionality of the program in relation to its objectives is primarily conducted through Black-Box

Testing which specifically and sequentially references objectives, with tests designed to test whether the objective has been accomplished to a functional level. The efficiency and reliability are tested mainly through White-Box Testing, where the TEX acronym (Typical, Erroneous, eXtreme) is used to identify and trial data inputs under which the behaviour of the program is observed for a correct response (under typical and extreme data the program will be expected to proceed as normal, and for erroneous data, the program is expected to have an appropriate exception handling routine).

In White-Box Testing, the program is tested on a modular level. Each module is taken individually and probed using data representative of all input data and the output data is reviewed for accuracy and reliability. The efficiency is also measured by including an arbitrary acceptable runtime for each test, which the task must complete within in order to pass the test. These runtimes are established by the nature of the calculation/process involved – for example, processing 1000 SSFT timesteps will require a longer runtime as opposed to simple variable assignment or the method adopted for complex number conversion.

Where relevant, I've included the basis for the runtime assumption below.

Based on the average termination time for the QMSolve method in the Analysis section for a similar system (5.18 seconds), this method should terminate in under 2x this time. The extra time is to allow for the fact that my algorithm stores every step of the wavefunction as opposed to storing only a fraction of the steps, explained in Design under SSFT.

Scenarios B and C are not directly comparable to any other simulation, but a reasonable execution time limit of 3 minutes is included for the main program, and up to 15 seconds for any manipulation operation (excluding playback operations, which are expected to be instant).

Validation Testing

Function	Objective reference	Purpose of Test	TEX (Typical, Erroneous, Extreme)	Expected Outcome	Pass/Fail	Timestamp
Particle System	1a	Correct type of particle system instantiated given a particle system time and a mass.	T1 s, m_h T2 s, m_e T3 d, m_h T4 d, m_e T5 ds, m_h T6 ds, m_e	Correct particle system selected.	**PASS**	[00:14] to [00.35] of Test Video **A**
			E – @:.;./13dA invalid type	Error message, prompt repeated.	**PASS**	
Hamiltonian - Potential	2a	Type of grid potential selected.	T1 hm T2 none	Program proceeds.	**PASS**	[00:35] to [00.50] of

			E – neither ?@>@?32As	Error message, prompt repeated.	**PASS**	Test Video **A**
Hamiltonian - Potential	2a	The time period and the maximum amplitude of the grid potential are assigned.	T1 1, 12 T2 default, default	Program proceeds.	**PASS**	[00:50] to [01.20] of Test Video **A**
			E – {}/324 -3 -12313 /'.S	Error message, prompt repeated.	**PASS**	
Hamiltonian – Grid points attribute	2b	Select number of grid points to use from 128, 256, 512.	T1 128 T2 256 T3 512	Program proceeds to time parameters.	**PASS**	[01:20] to [01.36] of Test Video **A**
			E – 12666 /@	Error message, prompt repeated.	**PASS**	
Simulation – Time Parameters	3a	Take in a simulation runtime and the amount of timesteps to perform.	T1 1, 100	Program proceeds.	**PASS**	[01:55] to [02.18] of Test Video **A**
			X – 10001 (invalid) 10000 (valid)	Input 1 -> Output 1		
			E – -2 @	Input 1 -> Output 1		

Black-box Testing Method

The Black-Box Testing is designed to be a comprehensive assessment of the program performance in relation to the objectives. However, as the program has multiple operation modes (due to the adaptable grid size and two particle system types – it was not possible to test every objective thoroughly within a single test scenario. Hence, I have split the Black-Box Testing into 3 tests, each of which is run sequentially in order to assess every objective completely. Each test can be conducted in

the same instance of the program, or separately and this has no bearing on the results. I have chosen to conduct Tests A and B in one instance of the program in order to prove that the program is able to successfully reload with new data inputs (not an objective) but conduct Test C separately (in order to reduce the length of the Test Videos).

Test A:

Using the input dialogue, create a single particle system with a particle of mass m_e, and a spatial extent of 10 angstroms. The potential used should be a harmonic oscillator with a time period of 0.4 femtoseconds and a maximum amplitude of 2 angstroms. The grid should be 128 points, and the simulation should evolve the system over 2 femtoseconds, using 100 timesteps.

Test B:

Using the input dialogue, create a single particle system with a particle of mass m_e, and a spatial extent of 20 angstroms. The potential used should be a harmonic oscillator with a time period of 1.0 femtoseconds and a maximum amplitude of 3 angstroms. The grid should be 256 points, and the simulation should evolve the system over 2 femtoseconds, using 1000 timesteps.

Test C: Using the input dialogue, create a two particle system with a particle of mass m_e, and a spatial extent of 20 angstroms. The potential used should be a harmonic oscillator with a time period of 1.0 femtoseconds and a maximum amplitude of 3 angstroms. The grid should be 512 points, and the simulation should evolve the system over 2 femtoseconds, using 500 timesteps.

Black-Box Testing of Prototype 1

These tests were conducted on the program Prototyping 1.2 – Superfluid SSFT Simulator, and unless stated otherwise, the video evidence of this testing is from the file “SSFT Simulator Testing Prototype A Black-box.mp4” for Test A and “SSFT Simulator Testing Prototype B Black-box.mp4" for Test B. Test C was not conducted on Prototype 1.

Test Number	Objective reference	Purpose of Test	Input	Expected Outcome	Pass/Fail	Timestamp
A1	A.1a	Menu Functionality + Input Handling (particle system type)	Particle System Dialog: s	Accepted input, next prompt	**PASS**	[00:21]
A2	A.1a	Menu Functionality + Input Handling (particle system mass)	Particle System Dialog: m_e	Accepted input, message to confirm particle system created, next prompt.	**PASS**	[00:22]
A3	A.2a	Menu Functionality + Input Handling (grid potential type, amplitude, oscillator time period)	Hamiltonian Dialog: hm, 2, 1	Accepted input, output of default oscillator parameters,	**PASS**	[00:27]

				next prompt.		
A4	A.2b	Menu Functionality + Input Handling (grid point parameter)	Hamiltonian Dialog: 128	Accepted input, next prompt.	**PASS**	[00:35]
A5	A.2c	Menu Functionality + Input Handling (spatial extent)	Hamiltonian Dialog: 20	Accepted input, next prompt.	**PASS**	[00:37]
A6	A.3a	Menu Functionality + Input Handling (simulation time parameters)	Time Evolution Dialog: 0.4, 100	Accepted input, simulation begins.	**PASS**	[00:39]
A7	B.3be	Simulation Controls (Matplotlib toolbar)	Zoom function	Graph zooms to cropped area.	**FAIL**	[01:17]
A8	B.3be	Simulation Controls (Matplotlib toolbar)	Home function	Graph resets to original view.	**FAIL**	[01:17]
A9	B.3be	Simulation Controls (Matplotlib toolbar)	Pan function	Graph pans about selected area.	**FAIL**	[01:17]
A10	B.3be	Simulation Controls (Matplotlib toolbar)	Back function	Graph reverts to before last change made using toolbar.	**FAIL**	[01:17]

A11	B.3be	Simulation Controls (Matplotlib toolbar)	Forward function	Graph redoes previously undone operation.	**FAIL**	[01:17]
A12	B.3be	Simulation Controls (Matplotlib toolbar)	Subplots function – test by changing graph to 25% of canvas.	Matplotlib menu opens with subplot dialog	**FAIL**	[01:17]
A13	B.3be	Simulation Controls (Matplotlib toolbar)	Save snapshot image of canvas.	Save location dialog.	**FAIL**	[01:17]
A14	B.3d	Simulation controls (TKinter GUI)	Pause animation (TKinter button)	Animation pauses.	**FAIL**	[01:01]
A15	B.3d	Simulation controls (TKinter GUI)	Restart simulation (TKinter button)	Animation replays from t = 0.	**FAIL**	[01:02]
A16	B.2d	Simulation controls (TKinter GUI)	Export simulation (TKinter button)	Animation downloads as .mp4.	**FAIL**	[01:03]
A17	B.2b	Simulation (TKinter GUI)	Previous input parameters (A1-A6)	Simulation window displays the inputs used.	**FAIL**	[01:03]
A18	B.2,3d (not directly stated)	Simulation (verification)	Previous input parameter (A6)	Simulation replays after reaching the runtime.	**FAIL**	[01:23]

B1	A.1a	Menu Functionality + Input Handling (particle system type)	Particle System Dialog: s	Accepted input, next prompt	**PASS**	[00:17]
B2	A.1a	Menu Functionality + Input Handling (particle system mass)	Particle System Dialog: m_h	Accepted input, message to confirm particle system created, next prompt.	**PASS**	[00:19]
B3	A.2a	Menu Functionality + Input Handling (grid potential type, amplitude, oscillator time period)	Hamiltonian Dialog: hm, 1, 3	Accepted input, output of default oscillator parameters, next prompt.	**PASS**	[00:20]
B4	A.2b	Menu Functionality + Input Handling (grid point parameter)	Hamiltonian Dialog: 256	Accepted input, next prompt.	**PASS**	[00:36]
B5	A.2c	Menu Functionality + Input Handling (spatial extent)	Hamiltonian Dialog: 20	Accepted input, next prompt.	**PASS**	[00:40]
B6	A.3a	Menu Functionality + Input Handling (simulation time parameters)	Time Evolution Dialog: 1, 1000	Accepted input, simulation begins.	**PASS**	[00:42]
B7	B.3be	Simulation Controls (Matplotlib toolbar)	Zoom function	Graph zooms to cropped area.	**FAIL**	[01:28]

B8	B.3be	Simulation Controls (Matplotlib toolbar)	Home function	Graph resets to original view.	**FAIL**	[01:28]
B9	B.3be	Simulation Controls (Matplotlib toolbar)	Pan function	Graph pans about selected area.	**FAIL**	[01:28]
B10	B.3be	Simulation Controls (Matplotlib toolbar)	Back function	Graph reverts to before last change made using toolbar.	**FAIL**	[01:28]
B11	B.3be	Simulation Controls (Matplotlib toolbar)	Forward function	Graph redoes previously undone operation.	**FAIL**	[01:28]
B12	B.3be	Simulation Controls (Matplotlib toolbar)	Subplots function – test by changing graph to 25% of canvas.	Matplotlib menu opens with subplot dialog	**FAIL**	[01:28]
B13	B.3be	Simulation Controls (Matplotlib toolbar)	Save snapshot image of canvas.	Save location dialog.	**FAIL**	[01:28]
B14	B.3d	Simulation controls (TKinter GUI)	Pause animation (TKinter button)	Animation pauses.	**FAIL**	[01:30]
B15	B.3d	Simulation controls (TKinter GUI)	Restart simulation (TKinter button)	Animation replays from t = 0.	**FAIL**	[01:30]

B16	B.2d	Simulation controls (TKinter GUI)	Export simulation (TKinter button)	Animation downloads as .mp4.	**FAIL**	[01:30]
B17	B.2b	Simulation (TKinter GUI)	Previous input parameters (A1-A6)	Simulation window displays the inputs used.	**FAIL**	[01:17]
B19	B.2,3d (not directly stated)	Simulation (verification)	Previous input parameter (A6)	Simulation replays after reaching the runtime.	**FAIL**	[01:49]

At this point testing was halted as clearly the prototype model did not meet the objectives required. This is because the TKinter controls were not operating correctly and the matplotlib toolbar failed to appear in the simulation window, a small error causing the prototype to fail an entire section. In the next section issues in the program are identified and rectified by the tests failed. The rows corresponding to the fixes are pasted above for convenience.

Resolving Issues

After consulting with the user, the command line style user input interface was not appealing and the time it took to enter the inputs was unacceptable. Mr Powell also specified that there was too much choice in the input parameters which was overwhelming and confusing. In order to simplify this, I created a GUI input dialogue window.

Script:

```
    def InputDialogue(self):
        self.inputwindow = Tk()
        self.inputwindow.geometry("300x500")
        self.inputwindow.title("Input Dialogue")
        grd_options = [
            "128",
            "256",
            "512",
        ]

        # datatype of menu text
        grd_clicked = StringVar()

        # initial menu text
        grd_clicked.set("128")

        # Create Dropdown menu
        grd_drop = OptionMenu(self.inputwindow, grd_clicked, *grd_options )
        grd_drop.pack()

        # Create Label
```

```
        self.gridpoints = Label(self.inputwindow, text = " " )
        self.gridpoints.pack()

        # Create button, it will change label text
        grd_button = Button(self.inputwindow, text = "Update", command = partial(self.show, self.gridpoints,
grd_clicked)).pack()

        ### NUMBER OF PARTICLES

        # Dropdown menu options
        par_options = [
            "1",
            "2 - Separated",
            "2 - Superposing",
        ]

        # datatype of menu text
        par_clicked = StringVar()

        # initial menu text
        par_clicked.set("1")

        # Create Dropdown menu
        par_drop = OptionMenu(self.inputwindow, par_clicked, *par_options )
        par_drop.pack()

        # Create Label
        self.particles = Label(self.inputwindow, text = " " )
        self.particles.pack()

        # Create button, it will change label text
        par_button = Button(self.inputwindow, text = "Update", command = partial(self.show, self.particles,
par_clicked)).pack()


        ### POTENTIAL

        # Dropdown menu options
        pot_options = [
          "Harmonic Oscillator",
          "No potential",
        ]
        # datatype of menu text
        pot_clicked = StringVar()

        # initial menu text
        pot_clicked.set("Harmonic Oscillator")

        # Create Dropdown menu
        pot_drop = OptionMenu(self.inputwindow, pot_clicked, *pot_options )
        pot_drop.pack()

        # Create Label
        self.potential = Label(self.inputwindow, text = " " )
        self.potential.pack()
```

```
        # Create button, it will change label text
        pot_button = Button(self.inputwindow, text = "Update", command = partial(self.show, self.potential,
pot_clicked)).pack()

        ### TIME_PARAMETERS

        time_label = Label(self.inputwindow, text = "Enter total time to run the simulation, in femtoseconds
(example: 2)")
        time_label.pack()
        self.time_slider = Scale(self.inputwindow, from_=0, to=10, orient = HORIZONTAL)
        self.time_slider.pack()

        steps_label = Label(self.inputwindow, text = "Enter amount of timesteps to carry out (example: 1000)")
        steps_label.pack()
        self.timesteps_slider = Scale(self.inputwindow, from_=100, to=10000, orient = HORIZONTAL)
        self.timesteps_slider.pack()

        osc_time_label = Label(self.inputwindow, text = "Enter a time period for the harmonic oscillator
(example: 1)")
        osc_time_label.pack()
        self.osc_time_slider = Scale(self.inputwindow, from_=0, to=5, orient = HORIZONTAL)
        self.osc_time_slider.pack()

        osc_amp_label= Label(self.inputwindow, text = "Enter amplitude of harmonic oscillator in angstroms
(example: 50)")
        osc_amp_label.pack()
        self.osc_amp_slider = Scale(self.inputwindow, from_=1, to=80, orient = HORIZONTAL)
        self.osc_amp_slider.pack()

        self.simnumber += 1

        create_button = Button(self.inputwindow,
                    command = partial(self.create_redraw, self.inputwindow),
                    height = 2,
                    width = 15,
                    text = "Create Sim button")
        create_button.pack(side = BOTTOM)
        self.inputwindow.mainloop()
```

Screenshot:

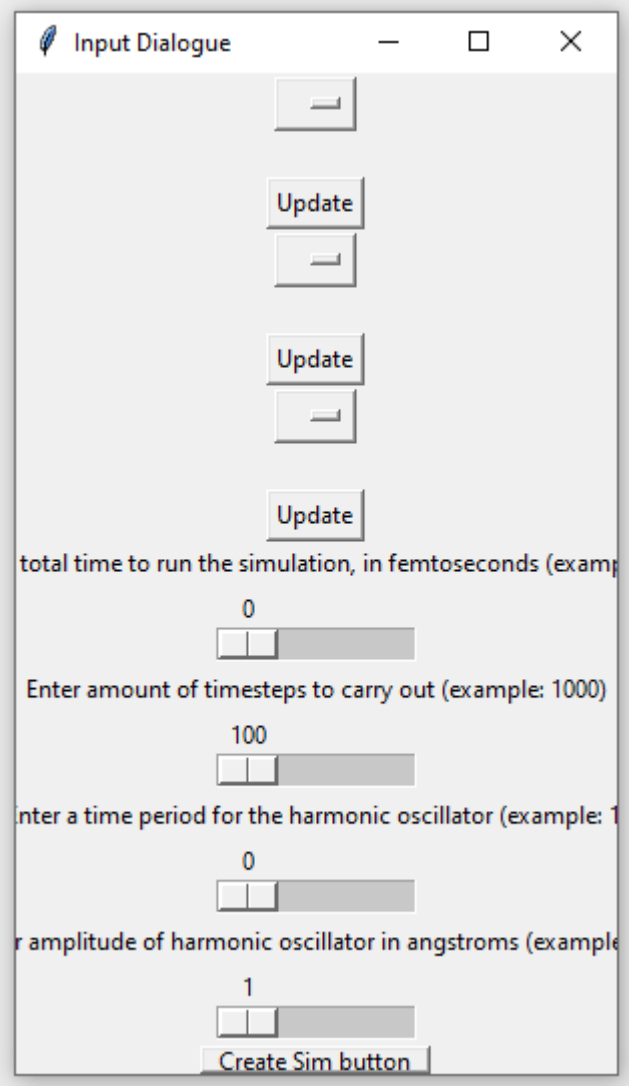


Figure 30 - New input dialogue.

The user was pleased with the addition of an input dialogue but suggested that the box could be wider to accommodate the text (an oversight on my part) and the sliders were still difficult to select values. I solved this issue by changing the resolution of the sliders to more suitable intervals.

Final Script:

```
    def InputDialogue(self):
        self.inputwindow = Tk()
        self.inputwindow.geometry("450x600")
        self.inputwindow.title("Input Dialogue")
        grd_options = [
            "128",
            "256",
            "512",
        ]

        # datatype of menu text
        grd_clicked = StringVar()

        # initial menu text
```

```
        grd_clicked.set("128")

        # Create Dropdown menu
        grd_drop = OptionMenu(self.inputwindow, grd_clicked, *grd_options )
        grd_drop.pack()

        # Create Label
        self.gridpoints = Label(self.inputwindow, text = " " )
        self.gridpoints.pack()

        # Create button, it will change label text
        grd_button = Button(self.inputwindow, text = "Update", command = partial(self.show, self.gridpoints,
grd_clicked)).pack()

        ### NUMBER OF PARTICLES

        # Dropdown menu options
        par_options = [
            "1",
            "2 - Superposing",
        ]

        # datatype of menu text
        par_clicked = StringVar()

        # initial menu text
        par_clicked.set("1")

        # Create Dropdown menu
        par_drop = OptionMenu(self.inputwindow, par_clicked, *par_options )
        par_drop.pack()

        # Create Label
        self.particles = Label(self.inputwindow, text = " " )
        self.particles.pack()

        # Create button, it will change label text
        par_button = Button(self.inputwindow, text = "Update", command = partial(self.show, self.particles,
par_clicked)).pack()


        ### POTENTIAL

        # Dropdown menu options
        pot_options = [
          "Harmonic Oscillator",
          "No potential",
        ]
        # datatype of menu text
        pot_clicked = StringVar()

        # initial menu text
        pot_clicked.set("Harmonic Oscillator")

        # Create Dropdown menu
```

```
        pot_drop = OptionMenu(self.inputwindow, pot_clicked, *pot_options )
        pot_drop.pack()

        # Create Label
        self.potential = Label(self.inputwindow, text = " " )
        self.potential.pack()

        # Create button, it will change label text
        pot_button = Button(self.inputwindow, text = "Update", command = partial(self.show, self.potential,
pot_clicked)).pack()

        ### TIME_PARAMETERS

        time_label = Label(self.inputwindow, text = "Enter total time to run the simulation, in femtoseconds
(example: 2)")
        time_label.pack()
        self.time_slider = Scale(self.inputwindow, from_=0.1, to=10, resolution=0.1, orient = HORIZONTAL)
        self.time_slider.pack()

        steps_label = Label(self.inputwindow, text = "Enter amount of timesteps to carry out (example: 1000)")
        steps_label.pack()
        self.timesteps_slider = Scale(self.inputwindow, from_=100, to=10000, resolution = 100, orient =
HORIZONTAL)
        self.timesteps_slider.pack()

        osc_time_label = Label(self.inputwindow, text = "Enter a time period for the harmonic oscillator
(example: 1)")
        osc_time_label.pack()
        self.osc_time_slider = Scale(self.inputwindow, from_=0.0, to=5, resolution=0.1, orient = HORIZONTAL)
        self.osc_time_slider.pack()

        osc_amp_label= Label(self.inputwindow, text = "Enter amplitude of harmonic oscillator in angstroms
(example: 50)")
        osc_amp_label.pack()
        self.osc_amp_slider = Scale(self.inputwindow, from_=1, to=80, orient = HORIZONTAL)
        self.osc_amp_slider.pack()


        create_button = Button(self.inputwindow,
                    command = partial(self.create_redraw, self.inputwindow),
                    height = 2,
                    width = 15,
                    text = "Create Sim button")
        create_button.pack(side = BOTTOM)
        self.inputwindow.attributes('-topmost',True)
            self.inputwindow.mainloop()
```

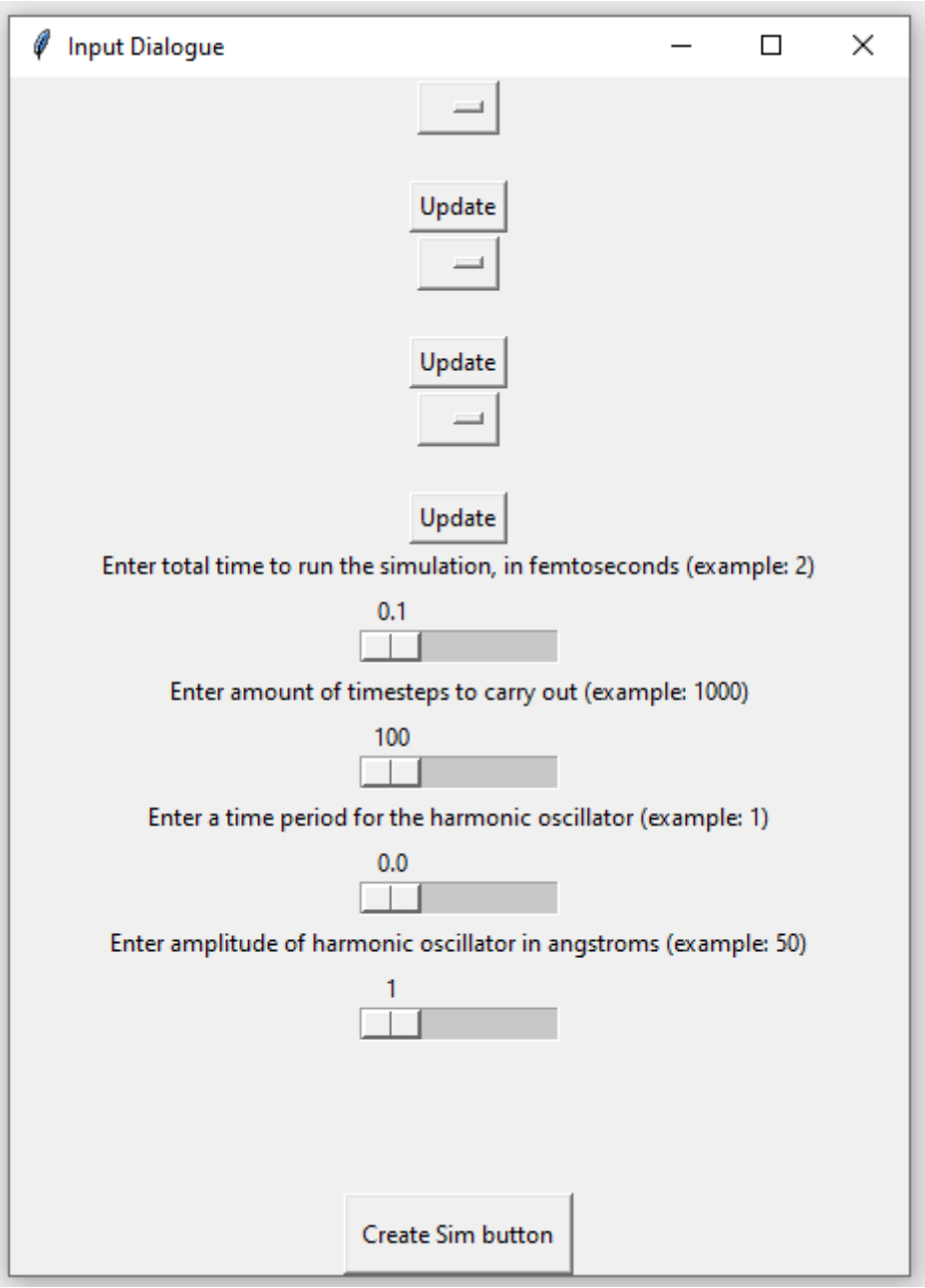


Figure 31 - Input Dialogue with corrected slider and window resolution.

Another error identified was in the run method of the split_step_fourier object:

Operator two of the SSFT method I have used requires adding the potential as a function of x to the square of the wavefunction (with some constants, irrelevant to the problem here). Hence the potential must be stored in the same datatype as the wavefunction. Initially I stored the potential as a NumPy array shaped into (N ** dims) elements. However, getting the non-linear wavefunction into this same format proved to be difficult leading to the error “potential not in form of wavefunction”. I rectified this by adjusting my wavefunction to be in a similar form. I used an alternative representation of the NLSE which I found on the quTARANG simulator **[48]**.

Further Testing

Redrawing a simulation with new parameters using the input dialogue.

When the new animation is plotted on top of the existing canvas, the simulation freezes. This is because the old contents of the canvas need to be cleared first. However, the canvas is not currently an accessible attribute from the update_animation() method. This is simply rectified, and the canvas can be cleared by checking for the attribute canvas (as it will not be present on the first iteration of update_animation()), and if present, clearing the canvas by deleting each widget inside. The current method for clearing the canvas is as follows:

```
if hasattr(self, "canvas"):
    self.canvas.get_tk_widget().destroy()
```

This works with the caveat of resetting the background colour of the canvas. Unfortunately, this error is not one I have time to rectify, but it neither affects the function nor the requirements of the program. Testing determined that the visual errors are limited to the background colour.

The second failure during testing was the toolbar controls of Matplotlib not working for redrawn simulations. This appears to be caused by the redrawn toolbar being associated with the previous simulation window. This was due to a missing return statement on the animation routine, adding this resolved the issue.

White-box Testing

At this point I chose to conduct the white box testing to discover any final errors before the Black-box retest. Additionally, some aspects of the program were impossible to test without reviewing outputs from within the program. For each method in the program, a comprehensive test ensuring the reliability of the output generated will be conducted. As most of the methods take in arguments from within the program, I will not be testing most methods with invalid data types or values as provided the function calling the method is operating correctly (which will also be tested here) then the function cannot receive an invalid input.

Some tests did not fit in the grid and are detailed below. Where possible, the tests are conducted from a run of the simulation, though some tests have required modifications to the program in order to display data within methods at specific points in time.

These modifications to the program are marked with comments # WB x.x.x and do not form part of the program except for this test.

The program is to be executed with the Black-Box Test A parameters.

Test Number	System being tested	Purpose of Test	Input(s) received	Expected Outcome	Pass/Fail	Timestamp
1.1	Particle System - Constructor	1.1.1 - To ensure that the correct type of particle system is established, along with the correct mass.	Electron mass, and either a single or two particle system.	Program runs when executed.	**PASS**	[00:45]
				Parameter pane displays the correct mass.	**PASS**	[00:49]

				Particle is of correct dimensions in the animation window (within a 20-angstrom range on the animation plot at t = 0)	**PASS**	[00:56]
2.1	Hamiltonian - Constructor	2.1.1 - Ensuring that the constructor correctly assigns the particle system to the Hamiltonian	Previously established particle system	Successful completion of WB tests 3.2.1ab	**FAIL**	[03:12]
		2.1.2 - '' assigns the correct dimension particle system and calculates associated derivatives.	Number of grid points (N) entered by user	Grid is composed of N points on each axis, verify using parameter pane.	**PASS**	[00:50]
		2.1.3 - '' creates a potential using the assigned dimensions and system potential parameters.	Time parameters of the oscillator, grid points from Test	Check using WB Test 2.1.3 Method (below).	**PASS**	see below
1.2	Particle System – Methods	1.2.1 - To create a series of linear arrays proportional to N and of the number of dimensions required.	N, number of grid points, number of system dimensions (2 for single particle, 4 for two particles), and the spatial extent of the system.	Successful completion of WB Test 1.2.2 with arrays equally spaced over a range of -20 to 20 with N points.	**PASS**	[02:40]
		1.2.2 – To ensure that these arrays are merged into a mesh grid of N dimensions each way (for storing the system energies/densitie	Linear arrays from WB Test 1.2.1	When a print statement is added to the routine, the program should output a matrix storing symmetrically	**PASS**	[02:40]

		s/potentials on the grid).		(ignoring sign), spaced values across the interval. Only one array printed as all are unpacked from the same meshgrid. Note: print method hides some middle values to save space.		
2.2	Hamiltonian - Methods	2.2.1 – To ensure the correct definition of the harmonic potential as a function of the grid created. (getpotentialasa matrix)	Hamiltonian attributes of spatial dimensions, N, particle system type and time parameters for oscillator.	When a print statement is added to the variable V, an array with periodic varying energies should be displayed.	**PASS**	[02:40]
				Check using WB Test 2.1.3 Method (below, carried out previously).	**PASS**	see below
3.1	GPE – constructor + initial wavefunctio n	3.1.1 - To confirm the assignment of the referenced Hamiltonian as a wavefunction attribute.	Previously created Hamiltonian object, N (this is also accessible through H and could be used without the import), grid point tuples.	The successful execution of 3.1.3 (as this is dependent on the use of Hamiltonian attributes).	**PASS**	[02:47]
		3.1.2 - To ensure the unpacking of the imported tuple into the correct number of axes (2 for a single particle, 4 for a two particle system).		Printed wavefunction from Test 3.1.3 contains a single blank complex number in each element.	**PASS**	[02:47]
		3.1.3 - Ensures that a call to the GPE method create_wfc_array s is made and the correct		Printing the wavefunction with each element displays a 2D array of length N with an	**PASS**	[02:47]

		formulation of the initial wavefunction is stored.		array within each element of length N		
				containing the initial NLSE conditions.	**PASS**	[02:47]
3.2	GPE – Methods, Normalisation (calc_normalisation_constant, renormalise)	3.2.1 - To ensure the correct normalisation of the array by integrating the square of the wavefunction and finding the reciprocal of the result as the normalisation constant.	Wavefunction, Hamiltonian attribute of derivative approximation.	When printed, a normalisation constant which is equal to 1/area found, using the integral A = $\int \lvert\psi\rvert^2 \, dV$ (as my method divides by the constant found ahead, the program value n_c is actually the area found and not 1/area.).	**PASS**	[03:11]
				Using testing comment in method renormalise, the area under the normalised wavefunction (squared modulus) should be 1.	**FAIL**	[03:12]
				No fuzzy rainbow pixels in animation, shown in Figure **x** below test. This would indicate a density higher than allowed in the area.	**PASS**	[01:05]

4.1	Simulation - constructor	4.1.1 – To ensure the correct calculation of all the time parameters of the simulation (dt, total_time, store_steps)	Total time, steps to store.	Using test statement, prints correct values of all three constants based on user input. (Note, time value is normalised into Hartree atomic units)	**PASS**	[03:04]
4.2	Simulation – Methods (sim)	4.2.1 – To ensure the return of an array of evolved wavefunctions (testing is for reliability only, accuracy is tested in 5.2, split_step_fourier.run)	Initial wavefunction, associated Hamiltonian and time parameters from WB 4.1.1.	Returns an array of wavefunctions to plot, which are 2D and of length N (so there are N*N points in each wavefunction).	**PASS**	[03:20]
5.1	Split Step Fourier - Constructor	5.1.1 - Ensures the correct assignment of the minimum and maximum of the potential grid (for use in visualisation)	Simulation object (WB 4)	Successful completion of WB Test 6.2.2.	**PASS**	[01:01]
5.2	Split Step Fourier – Methods (run)	5.2.1 - To ensure the accuracy of the evolution of the wavefunction using an SSFT method.	A wavefunction and a Hamiltonian object.	Using test print statement, prints a wavefunction in momentum space with the first half-step carried out	**PASS**	[03:08]
				Using test print statement, prints a wavefunction in real space with the second kinetic operator applied.	**PASS**	[03:10]
				Using test print statement, prints a completely evolved wavefunction with the third operator applied.	**PASS**	[03:11]

6.1	Visualise Wavefunction - Constructor	6.1.1 - Ensure the correct definition of attributes for pause status, restart status, video writing status and download count.	None	Using test print statement, output of False, False, False, 0 at beginning of simulation.	**PASS**	[00:21]
				When writing a video, test command indicates in the console that a video is being written.	**PASS**	[01:37]
				When simulation is paused, test statement writes a message in the console.	**PASS**	[01:37]
				When simulation is restarted, a test command logs this in the console.	**PASS**	[01:37]
6.2	Visualise Wavefunction – Animation Methods (complex_to_rgb, animate, func_animation)	6.2.1 - To ensure that the wavefunction arrays from the simulation object are correctly converted into RGB colours.	Wavefunction array from simulation object.	With test command, prints a N*N array of RGB values in lists of 3.	**PASS**	[03:30]
		6.2.2 - To ensure the creation and animation of frames of the calculated RGB colours across the grid.	Maximum and minimum potentials, (assigned to the simulation object from the SSFT object),	Animation does not exceed figure boundary over the course of the animation.	**PASS**	[01:01]
		6.2.3 - To ensure the integration of the matplotlib figure into the TKinter window (using the FigureCanvasTkAgg and	Matplotlib figure from WB 6.2.2, a TKinter window	Figure is embedded in window.	**PASS**	[00:50]
				Figure adapts with window when window is manipulated.	**PASS**	[01:35]

		NavigationToolbar2Tk).		Toolbar controls the current figure in the canvas.	**PASS**	[01:14]
6.3	Visualise Wavefunction – Methods (mpeg_write, TogglePauseStatus, ToggleRestartStatus, InputDialogue, show, create_redraw, update_animation, Buttons, DisplayParameters, MainWindow)	6.3.1 - Ensuring the creation of a blank GUI window with an integrated Matplotlib figure and associated toolbar for controlling a figure.	Visualisation object.	TKinter window with Matplotlib figure embedded.	**PASS**	[00:50]
				Window has parameters of simulation displayed.	**PASS**	[00:50]
				Toolbar embedded in suitable position on the window.	**PASS**	[00:50]
		6.3.2 - To ensure the creation and functionality of controls for the simulation allowing for downloading video, pausing and restarting the simulation.	Attributes from 6.1.1.	Records an mp4 video of the animation beginning at t = 0 and lasting 10 seconds of the animation, saving to the path in the program (User Video Folder).	**PASS**	[01:23]
				Pauses the simulation when pause button clicked (check time and animation pause, and play again when re-clicked).	**PASS**	[01:00]
				Animation restarts from t = 0 when restart button clicked (test for effects of pause button and time display in figure).	**PASS**	[00:55]
				Selecting any combination of the controls does not break the program (downloading whilst paused,	**PASS**	[01:00]

				restarting whilst paused).		
		6.3.3 - To ensure the reliability of the input dialogue window (specifically related to GUI related interaction, not validation or accuracy of values in program).	All inputs from InputDialogue as attributes of the visualisation object. (Note: use of global variables in InputDialogue explained later).	Separate input window	**PASS**	[00:18]
				with buttons and sliders.	**PASS**	[00:25]
				Buttons have appropriate dropdowns	**PASS**	[00:26]
				and sliders have appropriate scales.	**PASS**	[00:31]
				All interfaces work correctly and pass an input to the simulation.	**PASS**	[00:40]

One White-Box test did not fit in the grid, details of the testing method are included below.

WB Test 2.1.3 – In order to test for the grid potential, the system must be visually tested and it must be possible to see the oscillations of the wavefunction on the grid. This should start with the centred particle system and progress outwards with a time period of the input (in this case 0.5 femtoseconds) causing the function to progress with approximately the amplitudes of the oscillations entered. The best way to assess this is to play back the downloaded animation window, pause the video at elapsed times 0 and 0.5 fms, and compare the position of the particle animation. It should have moved approximately 2 angstroms and this can be checked visually using the graph axes. I adjusted the runtime of the simulation as previously the duration was not long enough to use this method. I took screenshots of the simulation window as the downloaded video does not include the axes.

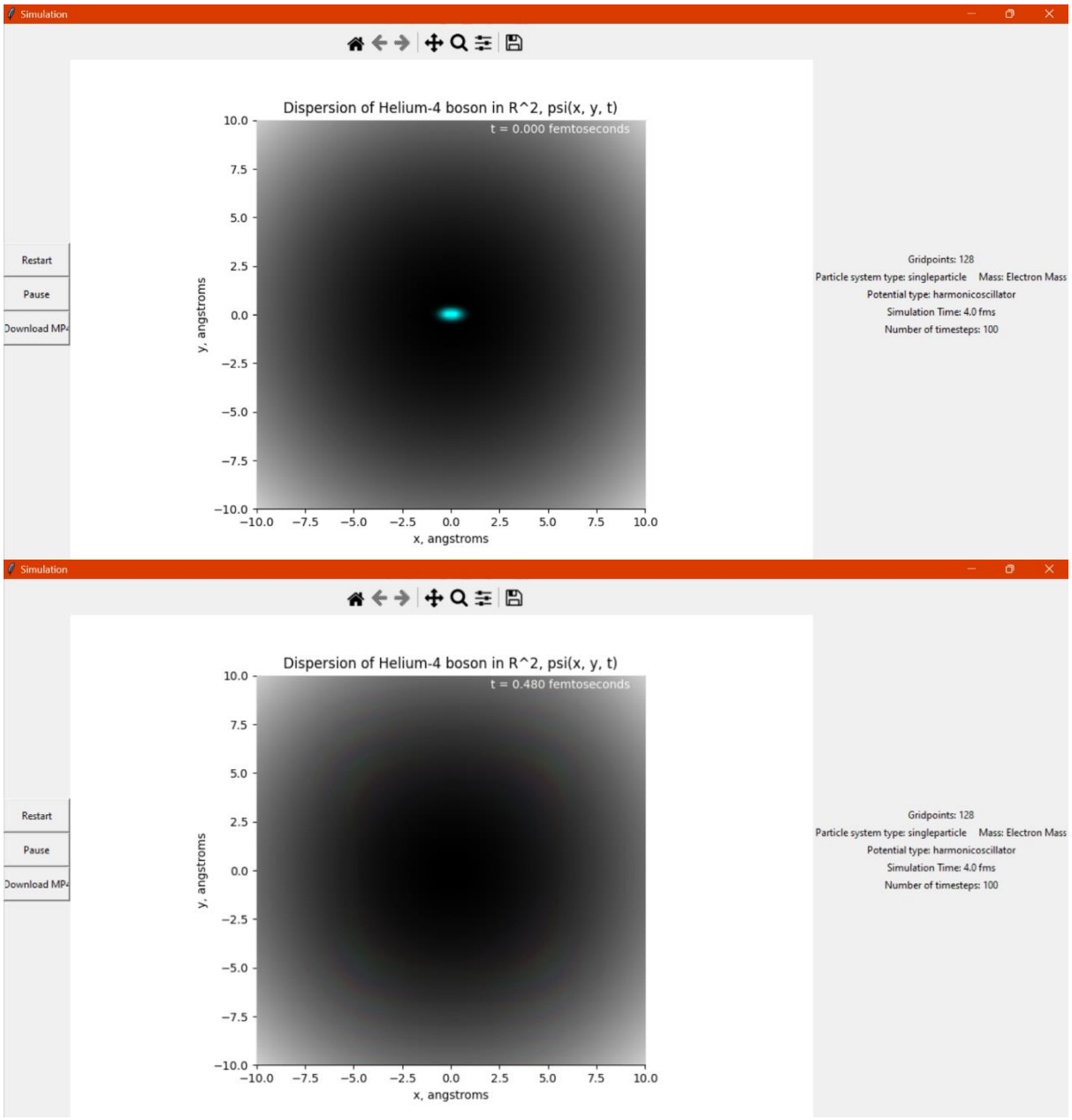


Figure 32 - WB Test 2.1.3 showing accuracy of harmonic oscillator.

As can be seen, the system has moved at least 2 angstroms away from the centre. The system has moved further than the amplitude of the oscillator, but it has also lost energy density which can be assumed to have transferred to kinetic and thus propagating the system slightly further. Whilst not

the most scientific method for testing this function of the program, it is reliable and the best method I have been able to identify for testing the potential grid.

Test 1.2.1 was successful, but from the console log of the linear arrays, it was not possible to identify the number of elements in the array – due to limitations of Python print method.. However, as the potential is defined using the meshgrid and then later added to a wavefunction as part of an SSFT operator, it must be dimensionally similar to the wavefunction, which is proven to be N elements by WB Test 3.1.3a.

WB Test 3.2.1b was a surprising fail, as it should theoretically be impossible for this step to go wrong, but the method fails to account for the subsitution of hbar in the Hartree atomic system used. Therefore, the wavefunction is normalised by a factor of hbar * omega, and so the area under the wavefunction will not be 1. Hence, the method passes the adjusted test.

2.1	Hamiltonian - Constructor	2.1.1 - Ensuring that the constructor correctly assigns the particle system to the Hamiltonian	Previously established particle system	Successful completion of WB tests 3.2.1ab	**PASS**	[03:12]
3.2	GPE – Methods, Normalisation (calc_normalisation_constant, renormalise)	3.2.1 - To ensure the correct normalisation of the array by integrating the square of the wavefunction and finding the reciprocal of the result as the normalisation constant.	Wavefunction, Hamiltonian attribute of derivative approximation.	Using testing comment in method renormalise, the area under the normalised wavefunction (squared modulus) should be proportional to hbar*omega, where omega is 2*pi/duration in Hartree fms time.	**PASS**	[03:12]

Test 3.2.1c is testing to ensure that results like below do not appear (where I have manually caused an overload in the RBG conversion method leading to a fuzzy rainbow-like image.

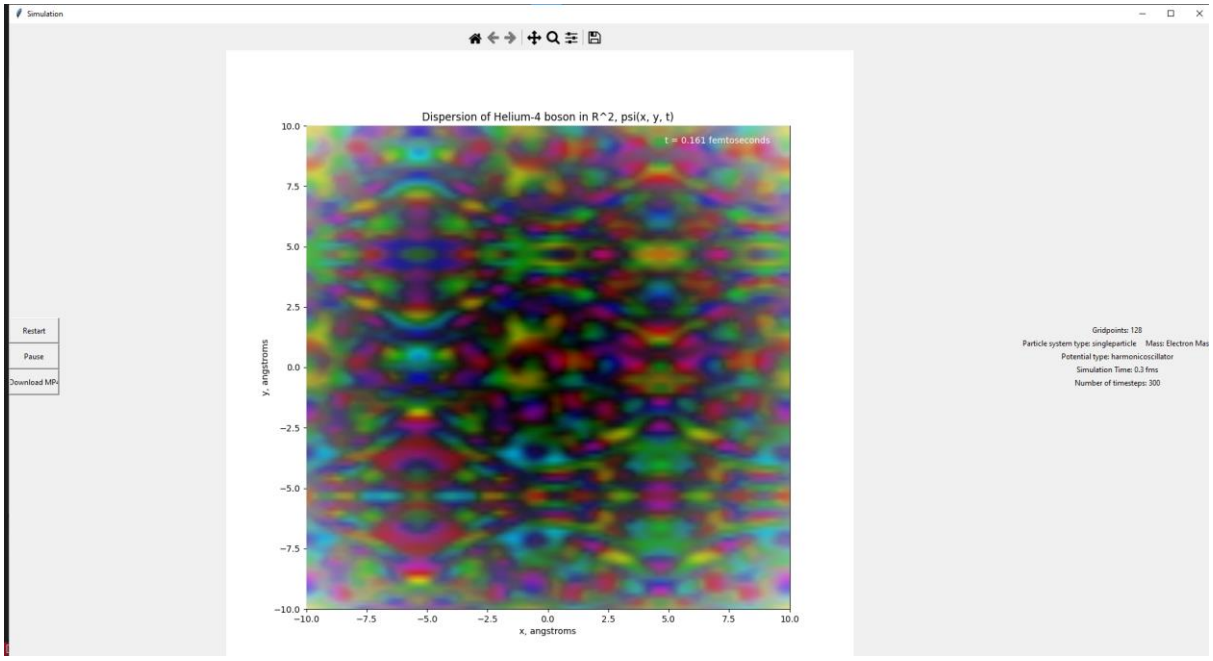


Figure 33 - Manual overriding of the wavefunction to destroy the RGB conversion method.

In WB Test 6.3.3, it was noticed that InputDialogue uses global variables for the buttons and sliders on the input window. Whilst this may seem incorrect for an OOP program, the InputDialogue associated widgets only exist for as long as the user is inputting data, before being batch deleted in the update_animation method. Hence the use of these variables does not affect the reliability or the design of this program. Below are two screenshots of the video, with the elapsed time shown in the corner.

Note that White Box Testing will break the functionality of the progress bar due to the print statements interfering with TQDM in the console. When in use, this will

All White Box Tests have now been passed successfully.

Adjusting Black-box Tests

As the input method to the program has been significantly revised, tests 1-6 have been modified to accommodate the new GUI. Other modifications include the removal of tests 2 and 5, the expansion of test 18 into tests 18a, 18b, 18c and 18d and the addition of test 20. This is to account for the removed input of mass and spatial extent, the introduction of an improved methodology for comparing the simulation to a similar counterpart, as explained in the section “Resolving Issues”

Revised tests are in bold.

Testing Revised Program (Black-Box)

These tests were conducted on the program CS NEA Final – Superfluid SSFT Simulator, and unless stated otherwise, the video evidence of this testing is from the file “SSFT Simulator Testing A Black-box Final.mp4” for tests A and B. Test C was conducted in a separate video file, “”.

Test Number	Objective reference	Purpose of Test	Input	Expected Outcome	Pass/Fail	Timestamp

A1	A.1a	Menu Functionality + Input Handling (particle system type)	Particle System Dialog: **1, update**	Accepted input**, output 1**	**PASS**	[00:38]
A2	A.1a	Menu Functionality + Input Handling (particle system mass)	**NO LONGER REQUIRED**	-	**N/A**	-
A3	A.2a	Menu Functionality + Input Handling (grid potential type, amplitude, oscillator time period)	Hamiltonian Dialog: hm, 2, 1	Accepted input, output of default oscillator parameters, next prompt.	**PASS**	[00:39]
A5	A.2b	Menu Functionality + Input Handling (grid point parameter)	Hamiltonian Dialog: 128	Accepted input, **output 128**	**PASS**	[00:36]
A4	A.2c	Menu Functionality + Input Handling (spatial extent)	**NO LONGER REQUIRED**	Accepted input, next prompt.	**N/A**	-
A6	A.3a	Menu Functionality + Input Handling (simulation time parameters)	Time Evolution Dialog: 0.4, 100	Accepted input, simulation begins.	**PASS**	[00:42]
A7	B.3be	Simulation Controls (Matplotlib toolbar)	Zoom function	Graph zooms to cropped area.	**PASS**	[01:31]
A8	B.3be	Simulation Controls (Matplotlib toolbar)	Home function	Graph resets to original view.	**PASS**	[01:48]

A9	B.3be	Simulation Controls (Matplotlib toolbar)	Pan function	Graph pans about selected area.	**PASS**	[01:38]
A10	B.3be	Simulation Controls (Matplotlib toolbar)	Back function	Graph reverts to before last change made using toolbar.	**PASS**	**Screenshot evidence as test missed.**
A11	B.3be	Simulation Controls (Matplotlib toolbar)	Forward function	Graph redoes previously undone operation.	**PASS**	**Screenshot evidence as test missed.**
A12	B.3be	Simulation Controls (Matplotlib toolbar)	Subplots function – test by changing graph to 25% of canvas.	Matplotlib menu opens with subplot dialog	**PASS**	**Screenshot evidence as test missed.**
A13	B.3be	Simulation Controls (Matplotlib toolbar)	Save snapshot image of canvas.	Save location dialog.	**PASS**	**Screenshot evidence as test missed.**
A14	B.3d	Simulation controls (TKinter GUI)	Pause animation (TKinter button)	Animation pauses.	**PASS**	[01:07]
A15	B.3d	Simulation controls (TKinter GUI)	Restart simulation (TKinter button)	Animation replays from t = 0.	**PASS**	[01:09]
A16	B.2d	Simulation controls (TKinter GUI)	Export simulation (TKinter button)	Animation downloads as .mp4.	**PASS** (delayed save time)	[01:53]

A17	B.2b	Simulation (TKinter GUI)	Previous input parameters (A1-A6)	Simulation window displays the inputs used.	**PASS**	[01:00]
A18	B.2,3d (not directly stated)	Simulation (verification)	Previous input parameter (A6)	Simulation replays after reaching the runtime.	**PASS**	[01:14]
B1	A.1a	Menu Functionality + Input Handling (particle system type)	Particle System Dialog: **1, update**	Accepted input**, output 1**	**PASS**	[02:38]
B2	A.1a	Menu Functionality + Input Handling (particle system mass)	**NO LONGER REQUIRED**	-	**N/A**	-
B3	A.2a	Menu Functionality + Input Handling (grid potential type, amplitude, oscillator time period)	Hamiltonian Dialog: hm, 2, 3	Accepted input, output of default oscillator parameters, next prompt.	**PASS**	[02:39]
B4	A.2b	Menu Functionality + Input Handling (grid point parameter)	Hamiltonian Dialog: 256	Accepted input, **output 256**	**PASS**	[02:36]
B5	A.2c	Menu Functionality + Input Handling (spatial extent)	**NO LONGER REQUIRED**	Accepted input, next prompt.	**N/A**	-
B6	A.3a	Menu Functionality + Input Handling (simulation time parameters)	Time Evolution Dialog: 1, 1000	Accepted input, simulation begins.	**PASS**	[02:43]

B7	B.3be	Simulation Controls (Matplotlib toolbar)	Zoom function	Graph zooms to cropped area.	**PASS**	[04:16]
B8	B.3be	Simulation Controls (Matplotlib toolbar)	Home function	Graph resets to original view.	**PASS**	[01:48]
B9	B.3be	Simulation Controls (Matplotlib toolbar)	Pan function	Graph pans about selected area.	**PASS**	[04:23]
B10	B.3be	Simulation Controls (Matplotlib toolbar)	Back function	Graph reverts to before last change made using toolbar.	**PASS**	**Screenshot evidence as test missed.**
B11	B.3be	Simulation Controls (Matplotlib toolbar)	Forward function	Graph redoes previously undone operation.	**PASS**	**Screenshot evidence as test missed.**
B12	B.3be	Simulation Controls (Matplotlib toolbar)	Subplots function – test by changing graph to 25% of canvas.	Matplotlib menu opens with subplot dialog	**PASS**	**Screenshot evidence as test missed.**
B13	B.3be	Simulation Controls (Matplotlib toolbar)	Save snapshot image of canvas.	Save location dialog.	**PASS**	**Screenshot evidence as test missed.**
B14	B.3d	Simulation controls (TKinter GUI)	Pause animation (TKinter button)	Animation pauses.	**PASS**	[03:45]

B15	B.3d	Simulation controls (TKinter GUI)	Restart simulation (TKinter button)	Animation replays from t = 0.	**PASS**	[04:01]
B16	B.2d	Simulation controls (TKinter GUI)	Export simulation (TKinter button)	Animation downloads as .mp4.	**PASS** (delayed save time)	**Screenshot evidence as test missed.**
B17	B.2b	Simulation (TKinter GUI)	Previous input parameters (A1-A6)	Simulation window displays the inputs used.	**PASS**	[01:00]
B18	B.2,3d (not directly stated)	Simulation (verification)	Previous input parameter (A6)	Simulation replays after reaching the runtime.	**PASS**	[01:14]
C1	A.1a	Menu Functionality + Input Handling (particle system type)	Particle System Dialog: **1, update**	Accepted input**, output 1**	**PASS**	[02:38]
C2	A.1a	Menu Functionality + Input Handling (particle system mass)	**NO LONGER REQUIRED**	-	**N/A**	-
C3	A.2a	Menu Functionality + Input Handling (grid potential type, amplitude, oscillator time period)	Hamiltonian Dialog: hm, 2	Accepted input, output of default oscillator parameters, next prompt.	**PASS**	[00:20]

C4	A.2b	Menu Functionality + Input Handling (grid point parameter)	Hamiltonian Dialog: 512	Accepted input, **output 512**	**PASS**	[00:16]
C5	A.2c	Menu Functionality + Input Handling (spatial extent)	**NO LONGER REQUIRED**	Accepted input, next prompt.	**N/A**	-
C6	A.3a	Menu Functionality + Input Handling (simulation time parameters)	Time Evolution Dialog: 2.0, 500	Accepted input, simulation begins.	**PASS**	[00:23]
C7	B.3be	Simulation Controls (Matplotlib toolbar)	Zoom function	Graph zooms to cropped area.	**PASS**	[02:39]
C8	B.3be	Simulation Controls (Matplotlib toolbar)	Home function	Graph resets to original view.	**PASS**	[03:10]
C9	B.3be	Simulation Controls (Matplotlib toolbar)	Pan function	Graph pans about selected area.	**PASS**	[02:57]
C10	B.3be	Simulation Controls (Matplotlib toolbar)	Back function	Graph reverts to before last change made using toolbar.	**PASS**	[03:12]
C11	B.3be	Simulation Controls (Matplotlib toolbar)	Forward function	Graph redoes previously undone operation.	**PASS**	[03:15]

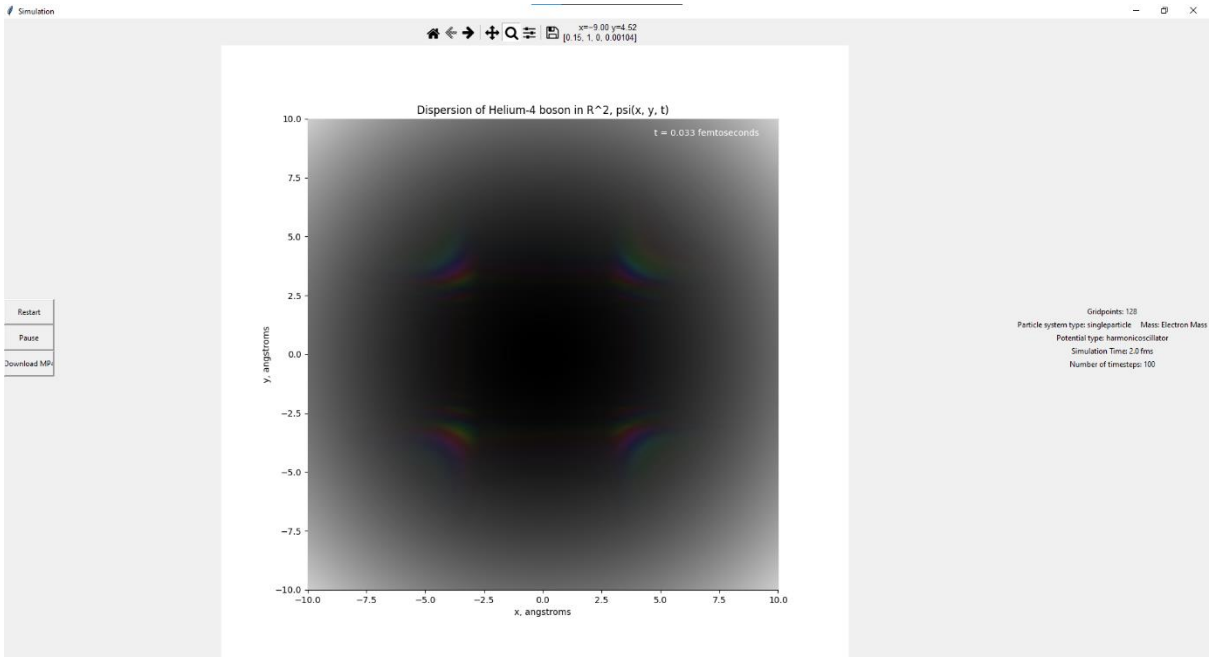
C12	B.3be	Simulation Controls (Matplotlib toolbar)	Subplots function – test by changing graph to 25% of canvas.	Matplotlib menu opens with subplot dialog	**PASS**	[03.20]
C13	B.3be	Simulation Controls (Matplotlib toolbar)	Save snapshot image of canvas.	Save location dialog.	**PASS**	[03:27]
C14	B.3d	Simulation controls (TKinter GUI)	Pause animation (TKinter button)	Animation pauses.	**PASS**	[02:25]
C15	B.3d	Simulation controls (TKinter GUI)	Restart simulation (TKinter button)	Animation replays from t = 0.	**PASS**	[02:29]
C16	B.2d	Simulation controls (TKinter GUI)	Export simulation (TKinter button)	Animation downloads as .mp4.	**PASS** (delayed save time)	[04:08]
C17	B.2b	Simulation (TKinter GUI)	Previous input parameters (A1-A6)	Simulation window displays the inputs used.	**PASS**	[02:17]
C18	B.2,3d (not directly stated)	Simulation (verification)	Previous input parameter (A6)	Simulation replays after reaching the runtime.	**PASS**	[03:35]

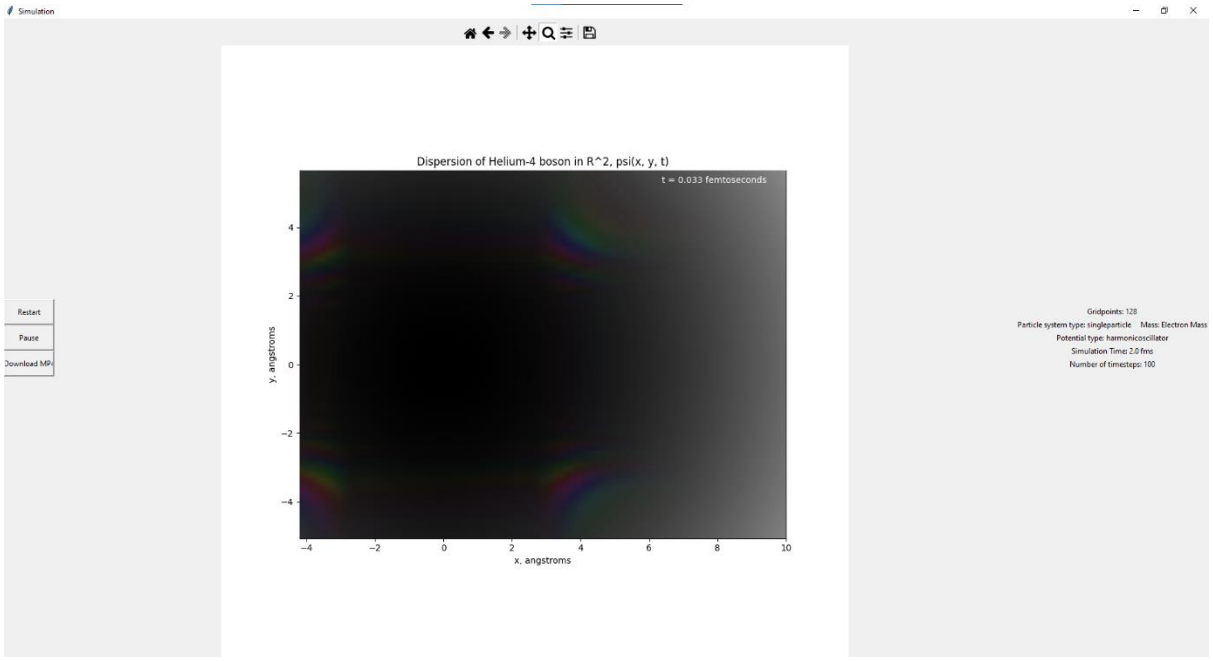
Clearly this program is a significant improvement on the first iteration. Test 16 is a persisting issue, but this will be explained and rectified in the next section.

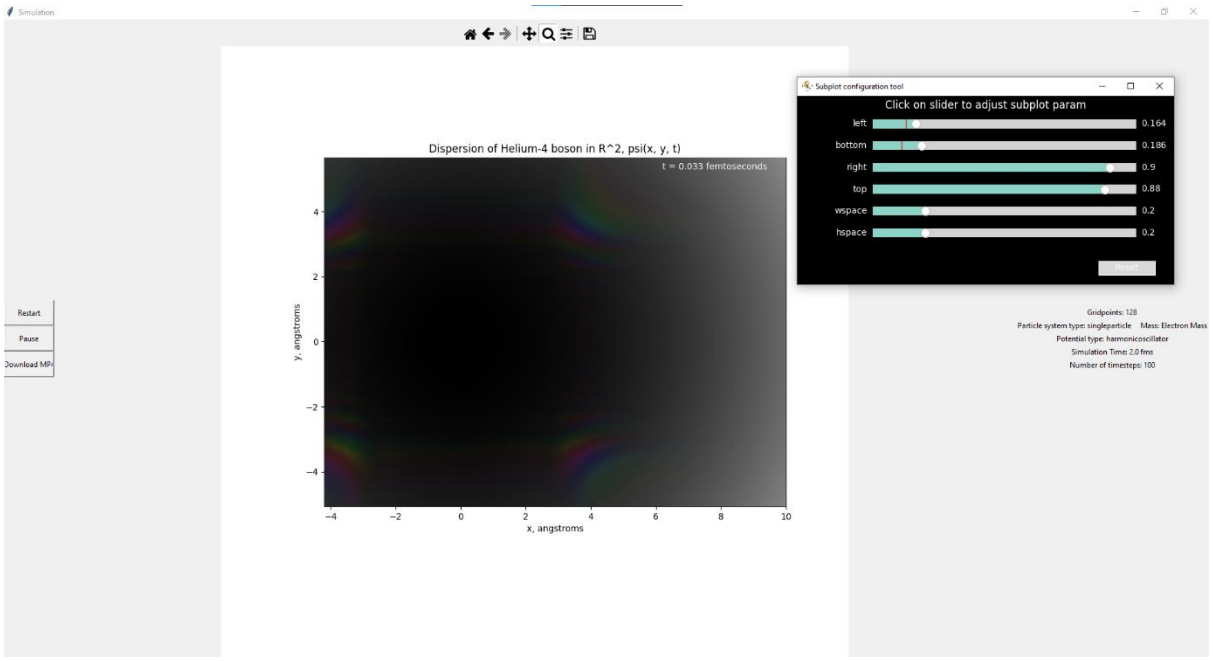
Test 16ABC – excessive duration.

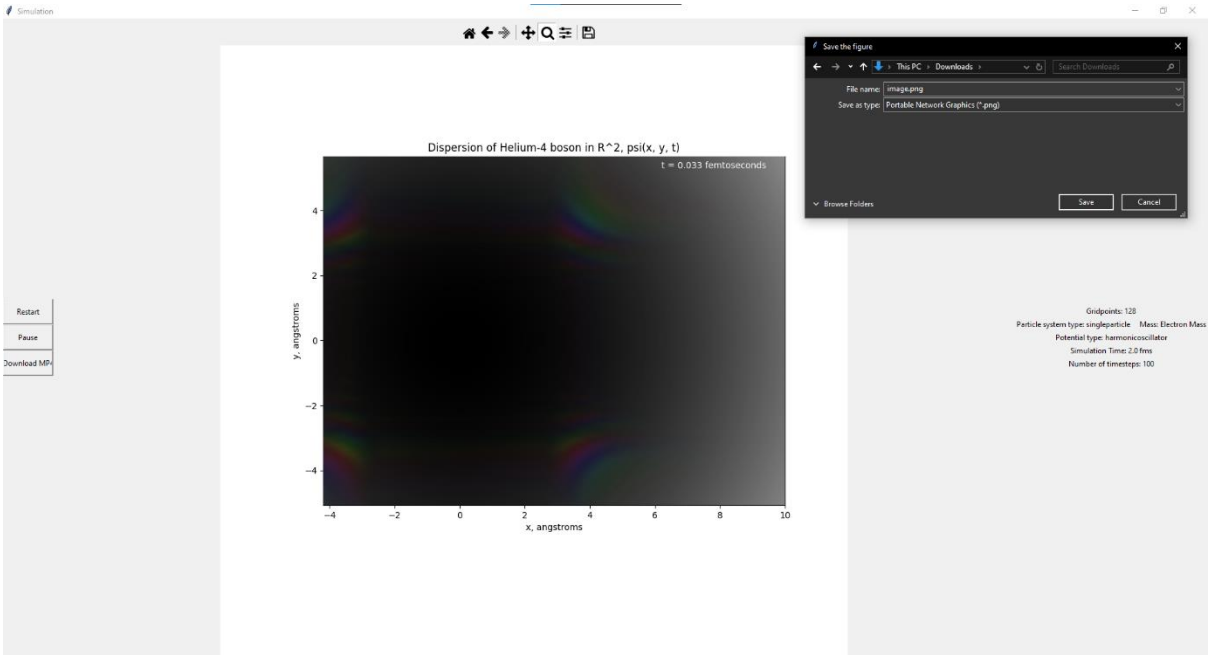
Test 16 took 31 seconds to save, which in my testing was outlined as an issue. The time taken by ffmpeg to encode the video was significantly higher than expected, amounting to roughly 3 seconds per second of video to be created (average from Final Black-Box Testing). However, capturing and encoding the video is a time intensive process which simply requires a lot of CPU resources. The resolution being captured is 896x658, determined by the size of the matplotlib figure, and so reducing the resolution would significantly reduce the quality of the final product as it is already below standard HD. Hence, I have decided that despite the runtime, this test should be considered as a pass as there is no acceptable compromise which I can make which is also suitable for the user. Whilst discrete graphics cards can process the video much faster, the user will be using a school provided laptop (similar to the device on which I conducted the testing). The video saving still serves its purpose too, as it remains faster (for large timesteps) to save a video and simply replay it rather than running the simulation again each time.

Screenshots for missed tests:




Simulation
Dispersion of Helium-4 boson in R^2, psi(x, y, t)
t = 0.033 femtoseconds
y, angstroms
x, angstroms
Restart
Pause
Gridpoints: 128
Simulation Time: 2.0 fms
Number of timesteps: 100


Simulation
Dispersion of Helium-4 boson in R^2, psi(x, y, t)
t = 0.033 femtoseconds
y, angstroms
x, angstroms
Subplot configuration tool
Click on slider to adjust subplot param
left 0.164
bottom 0.186
right 0.9
top 0.88
wspace 0.2
hspace 0.2
Restart
Pause
Gridpoints: 128
Simulation Time: 2.0 fms
Number of timesteps: 100



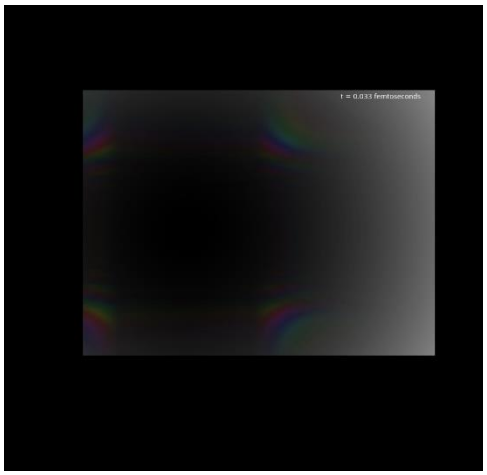
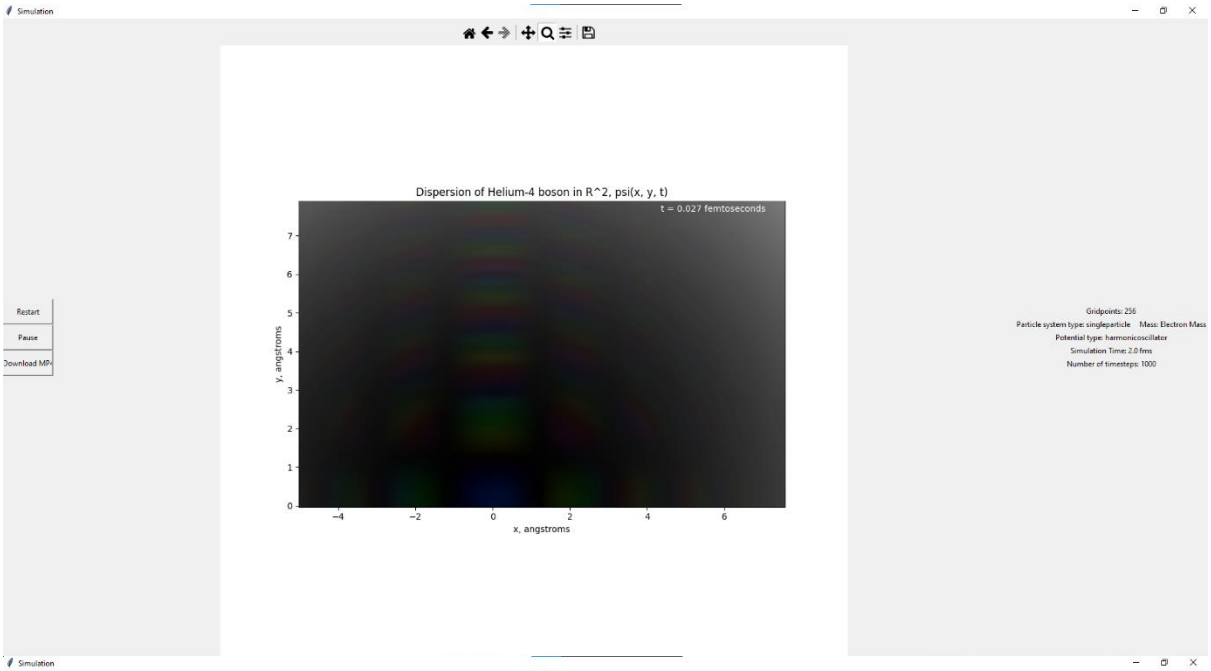
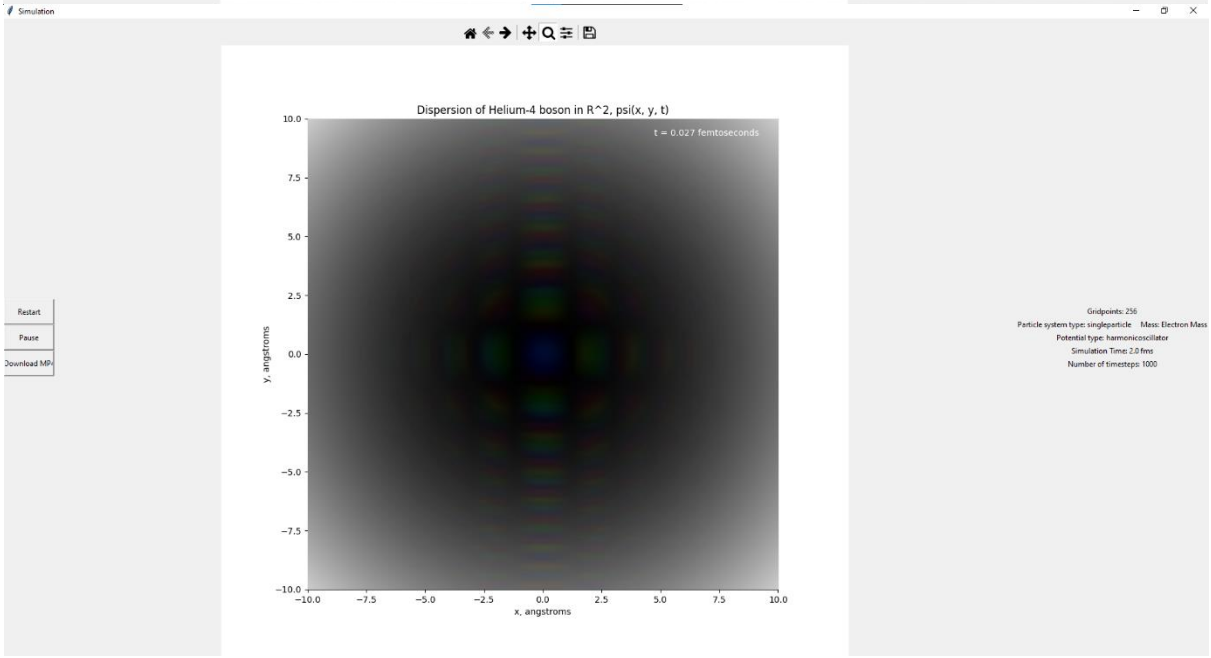
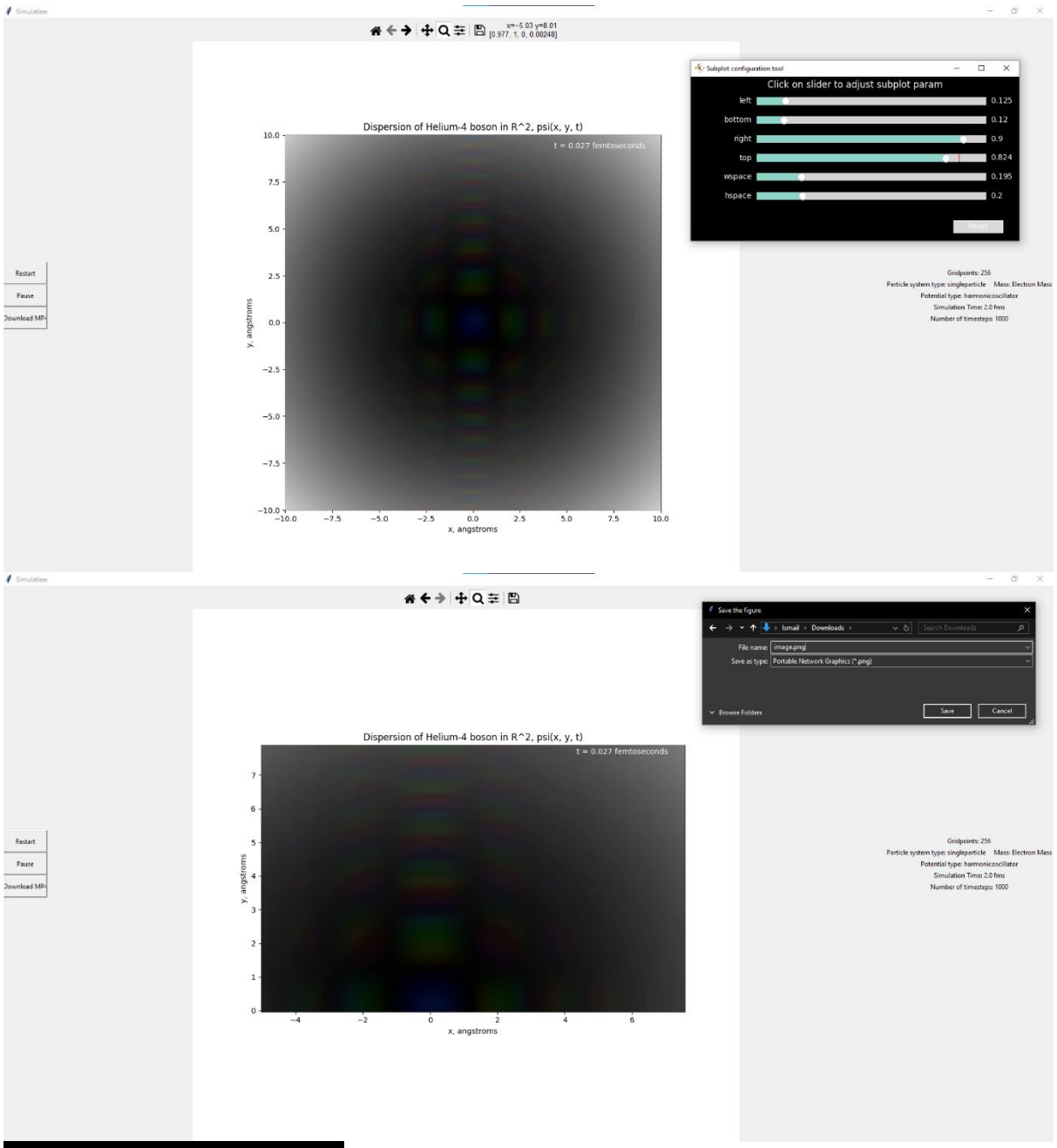


Figure 34 - Tests A10-13 evidence, along with image saved for A13.


Simulation
Dispersion of Helium-4 boson in R^2, psi(x, y, t)
t = 0.027 femtoseconds
y, angstroms
x, angstroms
Restart
Pause
Gridpoints: 256
Particle system type: singleparticle Mass: Electron Mass
Potential type: harmonicoscillator
Simulation Time: 2.0 fms
Number of timesteps: 1000


Simulation
Dispersion of Helium-4 boson in R^2, psi(x, y, t)
t = 0.027 femtoseconds
y, angstroms
x, angstroms
Restart
Pause
Gridpoints: 256
Particle system type: singleparticle Mass: Electron Mass
Potential type: harmonicoscillator
Simulation Time: 2.0 fms
Number of timesteps: 1000



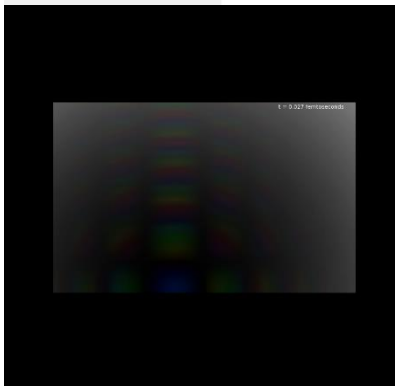


Figure 35 - Tests B10-B13, including the downloaded test image.

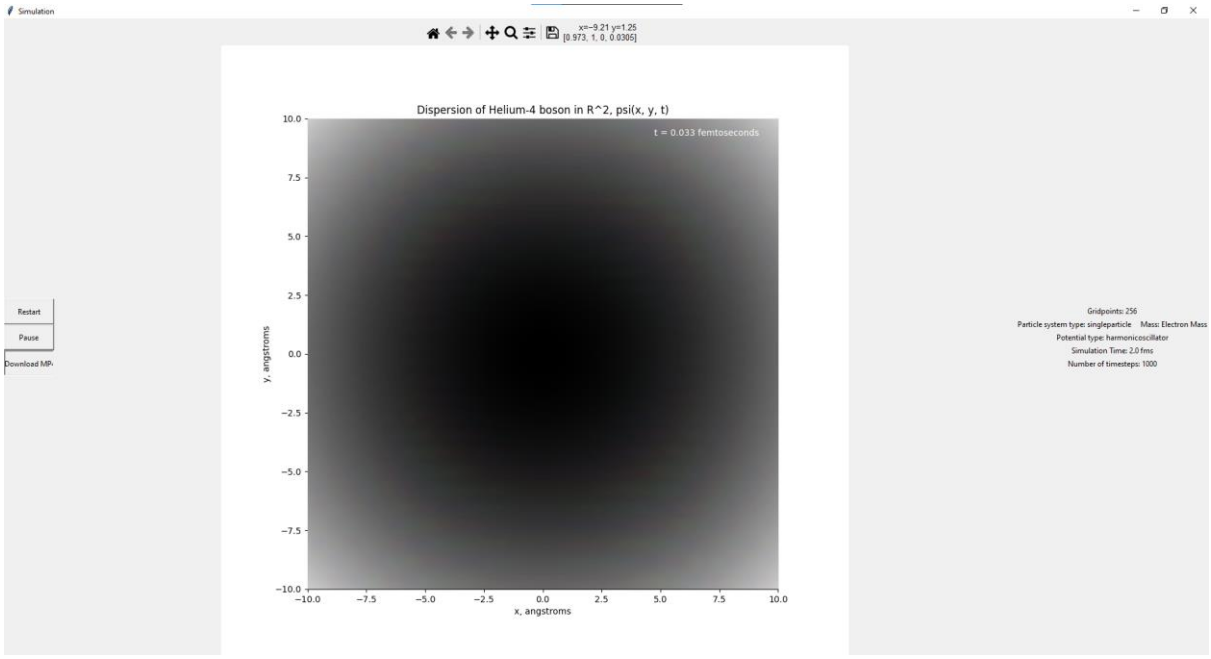


Figure 36 - Saving a video (B16).

Test Video Library:

Video Title	Used for
SSFT Simulator Testing 1.1 Black-box.mp4 SSFT Simulator Testing 1.2 Black-box.mp4	Black-box Test Prototype (Tests A and B)
SSFT Simulator Testing White-Box.mp4	White-box Testing
SSFT Simulator Testing Validation of Inputs.mp4	Validation Testing
SSFT Simulator Testing Modular Testing Fixed Downloads.mp4	Modular Testing – Video Writing
SSFT Simulator Testing Modular Final Review.mp4	Modular Testing – Reviewing final program
SSFT Simulator Testing Black-box Final.mp4	Black-box (Final Testing)

Objectives Met

From this combination of objective oriented testing and function testing, clearly my program meets the vast majority of the objectives outlined. The main exception is to B.2d which works but with an unexpectedly slow runtime. This is also true under certain conditions for A1, as this particle simulator is mostly useful for one particle as opposed to two particle simulations. Two particle simulations would require far more advanced hardware to run at similar conditions to some of the tests that I have conducted.

Evaluation

Requirements

A copy of the User Requirements has been included for reference.

a) Direct user control must be possible, I should be able to forward, rewind and freeze the simulation as well as zoom in and out.
b) I would like the ability to adjust the various parameters of the fluid during the course of the simulation. These should be displayed with an obvious scale and be adjustable as the simulation is live.
c) Interaction with the simulation must be possible. The user should be able to introduce some kind of obstacle into the simulation space.
d) Preferably, the environment should be the configuration space in which the system exists, as opposed to a complex space or single-property representation. This will allow scope to show how a 2D diagram can show many properties at once.
e) The program should output in words what it is calculating at any point, this is in order to show students what types of calculations might be required for such a model.
f) The program should be able to show interactivity between ideal identical particles and the symmetry relationship between bosons. This could be achieved using a two particle simulation.

From testing, it is clear that requirement a) has been achieved (Black-box Tests 7-16), requirement b) through Black-box Tests 1-6. However, requirement b) specifies that the simulation should be adjustable whilst live – but the chosen method leaves very little to control during the course of the simulation. Hence this objective is only partially achieved.

Requirement c) is also partially achieved as the simulation can be interacted with but introducing an obstacle is not possible live – it is possible to add a potential method which includes an object but this has not yet been achieved. I later discuss with the user adding these features in as update packages.

Requirement d) has been fully achieved as the environment used is a 2D configuration space, I heavily drew from the simulation window used in QMSolve, with some modifications to use a more accurate colour mapping and a higher frame rate. I also added some playback functions to the animation routine and wrapped it in a GUI.

Requirement e) is achieved through both the Parameters frame (Black-box test 17) and the console output. The program does report what stage of processing it is at and the progress of the time-stepping algorithm is monitored by a TQDM progress bar which provides an indicator of time elapsed and a good estimate of time remaining. However this could be improved by the ability to relate the operations being carried out to the user in more detail if desired and also including a progress bar for the loading of the animation as with a high number of timesteps, there is a delay between the call to the function and the output due to the time taken to create the frames in the func_animation.

Requirement f) is satisfied by the two particles functions – but this takes an inordinate amount of time to calculate given the sizes of the arrays required (as demonstrated in . Therefore, the two particle function in restricted to the smallest grid size available, and it only executes within a reasonable timeframe if the number of timesteps selected is small (<500 for execution in under 10 minutes).

Objectives

Objective	Met	Comments
A1 - The program should create a two-particle system object for a Helium-4 system.	Partial	The program is capable of creating a two particle system, however this function is limited by the amount of available RAM on the target system and so only a grid size of 128 will work and the runtime is around 3 minutes (from testing). The primary objective and function of this program is single particle systems owing to the device constraints.
A2 - The program should formulate a Hamiltonian for a Helium-4 system as an object and use this information to create a grid and kinetic energy matrix.	Y	The program is fully capable of formulating a Hamiltonian from a set of basic input parameters (a particle system, spatial extent, a potential and a spatial dimension number). Two limitations of this method are the lack of ability to change the spatial dimension (effectively the simulation will only run 2D models) and there is no provision for electron spin or charge potentials. Whilst neither of these were mentioned in the objectives, the design of the program has made it easy to implement these and so the functionality of this product could have been greatly extended. This also applies to the potential method, as whilst the harmonic oscillator grid potential works as outlined in the objective, many other similar methods could have been implemented such as a charged grid or magnetic field. This would have extended the functionality, but more importantly, both of these potential types are part of the A-Level Physics curriculum and this program could be used to demonstrate these concepts, linking the two topics together in the classroom. If I were to write this objective again, I would include the extra potential types, although I have discussed adding these functions with the user.
A3 - The program should formulate an initial dissipative Gross-Pitaevskii equation for a Helium-4 system using a wavefunction Ψ_0.	Y	The program successfully sets up a wavefunction of the correct dimensions and populates it with a non-linear function of the system position. This method was adapted from the quTARANG equation solver to overcome the issue with adding a potential grid to a wavefunction. This implementation has worked well, with no errors from the offset in testing and no problems with wavefunction density in the image (aside from a colouring issue caused by another part of the program). This objective has been totally achieved, and has proven that the GPE can be used with the SSFT to evolve an animated wavefunction.
A4 - The wavefunction should be established and the pre-calculation checks should be implemented (Hamiltonian constructor).	Y	The establishment of the wavefunction has been covered under objective A3 evaluation as it became part of the initial formulation of the wavefucntion arrays, instead of a Hamiltonian method. The Hamiltonian constructor initialises and retains all the attributes necessary to carry out the time evolution (A5). This includes a poor approximation to the spatial derivative

		operator as the accuracy of the normalisation integral is dependent on the relative width of the system in comparison to the grid. This will have the effect of making smaller grid size simulations increasingly inaccurate over longer times. All of the sub-objectives of this group asre successfully accomplished however, and every test relating to normalisation has passed.
A5 - A synchronised split-step Fourier method should be implemented to determine a set of solutions.	Y	The split step Fourier method is carried out using an approximation to an exact solution to the NLSE obtained from a research paper. [47] This method was successfully tested against QMSolve animations for accuracy (roughly as the systems were different), and proved to be reliable during all testing. The time evolution limites the program by forcing potential types to be a function of position, but the user agrees that these are typically more useful for the classroom environment.
B1 - In order to animate the time evolution of the wavefunctions, a visualisation object must be instantiated.	Y	A visualisation object has been used to manage the GUI, the controls and the simulation figure. This object controls writing the animation to a file, pausing, restarting and the figure integration. This object successfully produces an animation in a GUI given a simulation object, but it is mainly assessed in B2 and B3.
B2 - The visualisation system must plot the results of the wavefunction evolution on a grid using a suitable animation method.	Y	The animation plotting is accomplished using MatPlotLib and a FuncAnimation frame plotting routine. This method was an adaptation of the method from QMSolve, with added controls for pausing, downloading and restarting the animation. The figure background colour and colour conversion method were also adapted to provide a clearer, brighter image. The figure was also integrated into a GUI to allow for scaling, extra controls and reusability of code without restarting the program.
B3 - The solution filled grid should be exported and displayed in the GUI.	Y	The MatPlotLib figure is exported to a TKinter GUI window which allows for resizing, controls and having a separate input dialogue. White-Box testing showed that this integration using a MatPlotLib backend aggregator was successful, and allows the program to run new animations without restarting the program or storing old simulation information. The GUI also allows for further customisations in the future, though I am unsure what this would look like.

User Consultation

As can be seen from the testing section, my user was consulted regularly throughout the testing section leading to changes such as the inclusion of an input menu dialogue and a restart button (thanks to Helen in Physics for the spot). However, I have conducted a final interview with Mr Powell to determine the overall usefulness of the project, and any further suggestions.

Mr Powell: Having spent some time playing with the final version of the simulation, I'm very pleased with the overall outcome. The simulation window is easy to manipulate and the output looks

excellent. This would also make a good resource for teaching Young's double slit diffraction if it were possible to add a slit into the window.

Me: Fortunately, the system is very modular, and the harmonic oscillator could be easily swapped out for a slit or even a magnetic potential. This could be released in the form of an update which adds the module into the program if desired.

Mr Powell: I understand you ran out of time to incorporate the eigenstate solver into the program. Would it be possible to incorporate this into the update too?

Me: This would require a new routine to plot the calculated eigenstates in the visualisation routine but given time would be possible. However, I am satisfied with how much of the project I accomplished, and it completes most of the tasks I'd hoped for. Is there anything you'd change about the simulation or anything that doesn't meet your expectations?

Mr Powell: Whilst the simulation is suitable for my needs, I would like to see the inclusion of "presets" where there is a menu to select types of particles and systems and the input dialogue is pre-populated for me instead of having to work out suitable values to input. I noticed that some values can seemingly destroy the simulation causing lots of flashing lines and something similar to TV static.

Me: A preset menu would be a welcome inclusion, this could be implemented with a different TKinter menu where you select a system, possibly with an icon to see a preview. The flashing lines are unfortunately unavoidable as this is the consequence of inputting invalid input data which causes the time stepping routine to "overload" and explode.

Mr Powell: Overall, I'm very pleased with this resource and I look forward to using it and possibly expanding it with you in the future!

In the future, the modular design of these projects means that it would be capable of solving and plotting for eigenstates, solving for multiple particle systems, or adding different types of potentials. Limitations to this would include the restriction to grid based potentials only due to the SSFT method requiring the potential to be a function of the position in space. Also, efficiency will limit the feasibility of multiple particle systems as the increase in runtime will be exponential.

Final Project

```
# Ismail Mehmood - Simulating a Helium-4 boson in 2D real space
#
### Imports
from abc import ABC, abstractmethod
# abc (Abstract Base Class) library used to make abstract base class for the particle system
import numpy as np
# numpy is used for the following: zeros, dtype, pi, exp, linspace, meshgrid, amin, amax, round, abs, fft.fftn,
fft.ifftn, array, ndarray, angle, moveaxis, where, concatenate
from tkinter import *
# tkinter is the python interface to the Tk GUI toolkit. This is used for StringVar, Label, Button, OptionMenu,
focus_set, pack, Frame, title, geometry, get, set.
import matplotlib.pyplot as plt
# Used for plotting program output:
from matplotlib import widgets
# Used to create the toolbar
from matplotlib import animation
# Used to animate the frames of the simulation generated from the plot routine.
```

```
from matplotlib.colors import hsv_to_rgb
# module from matplotlib which converts the HSV colours obtained from the complex components of the energy matrices to RGB.
from matplotlib.backends.backend_tkagg import (FigureCanvasTkAgg, NavigationToolbar2Tk)
# this backend plugin is used to integrate the plot from matplotlib into the tkinter GUI.
from tqdm import tqdm
# tqdm is a wrapper for iteratives which generates a progress bar. This gives the user an indication of the time taken for the time-dependent solver routine.
#import ffmpeg
# video library for handling video (used to download animation)
from scipy import ndimage # integrate, sparse
# scipy is used for the following: ndimage (for the laplace integral routine to approximate the second spatial derivative) Previously used integrate (to calculate the normalisation constant of the wavefunction) and sparse (to create a sparse 2D array from a matrix)
from functools import partial
# creates a partial function from a function and parameters in order to package parameters into tkinter buttons

### Physical contants (to 11dp)

# Four fundamental physical constants
# note, these constants are in Hartree atomic units for simpler and faster calculations.
hbar = 1.0 # reduced Plank's constant
e = 1.0 # elementary charge
k_e = 1.0 # Coulomb constant
m_e = 1.0 # electron rest mass

# Interaction Potential of Helium-4 boson
a_s = 1.6782483e-9
#a_s = 1.0
# Time constants - ***seconds to atomic time unit (hbar/hartree energy)
picoseconds = 4.134137333518212e4
femtoseconds = 4.134137333518212 * 10.

# each constant here is to 17sf
Å = 1.8897261246257702 # angstrom
m_p = 1836.1526734400013

k = hbar**2 / (2*m_e) # force constant for oscillator.

# Classes for particle system, simulation and time evolution:
# Step 2: Define the objects


class GPE:
    def __init__(self, gridpointtuples, H):
        #spatialext is a list of tuples
        self.H = H
        self.gridpoint = H.N
        if len(gridpointtuples) == 2:
            self.nX, self.nY = gridpointtuples
        elif len(gridpointtuples) == 4:
            self.nX, self.nY, self.nXX, self.nYY = gridpointtuples
        self.create_wfc_arrays()
```

```
    def create_wfc_arrays(self):
        if type(self.H.particle_system) == singleparticle:
            self.wfc = np.zeros((self.nX, self.nY), dtype = np.complex128) # wavefunction for real space
        elif type(self.H.particle_system) == twoparticles:
            self.wfc = np.zeros((self.nX, self.nY, self.nXX, self.nYY), dtype = np.complex128)

    def wfc_0(self, x1, y1): # initial wavefunction initialiser, takes in an x and y and outputs a function of the two representing the wavefunction from that position
        eps1 = 0.25
        const = 8**0.25/(np.pi * eps1)**(3/4)
        return const * np.exp(-(x1**2 + 2*y1**2/(2*eps1)))

    def wfcd_0(self, x1, y1, x2, y2): # initial wavefunction initialiser for dual particle systems, outputs a function of the 4 xy parameters representing the wavefunction from that position.
        #This wavefunction correspond to two stationary gaussian wavepackets. The wavefunction must be symmetric: Ψ(x1,x2) = Ψ(x2,x1)
        σ = 0.4 * Å
        μ01 = -7.0*Å
        μ02 = 0.0*Å
        return (np.exp(-(x1 - μ01)**2/(4*σ**2))*np.exp(-(x2 - μ02)**2/(4*σ**2)) + np.exp(-(x1 - μ02)**2/(4*σ**2))*np.exp(-(x2 - μ01)**2/(4*σ**2))) + (np.exp(-(y1 - μ01)**2/(4*σ**2))*np.exp(-(y2 - μ02)**2/(4*σ**2)) + np.exp(-(y1 - μ02)**2/(4*σ**2))*np.exp(-(y2 - μ01)**2/(4*σ**2)))

    def calc_normalisation_constant(Hm, psi): # normalisation constant of wavefunction
        n_c = Hm.dV * (np.sum(np.abs(psi)))**2 # The probability of the particle being in the real space is 1.
        return n_c

    def renormalise(Hm, normconstant, wfc): # normalisation routine - tales in the calculated normalisation constant and divides wfc by it
        wfc /= normconstant
        return wfc

    def initialsinglewavefunction(self, Hm): # defines each position in the wfc array as a function from wfc_0
        self.wfc[:] = self.wfc_0(Hm.particle_system.x1, Hm.particle_system.y1) # the colon fetches the entire array
        print("wavefunction:", self.wfc) # WB 3.1.3
        return self.wfc

    def initialdualwavefunction(self, Hm): # defines each position in the wfc array as a function from wfc_0
        self.wfc[:] = self.wfcd_0(Hm.particle_system.x1, Hm.particle_system.y1, Hm.particle_system.x2, Hm.particle_system.y2) # the colon fetches the entire array
        return self.wfc


# The class particle_system is used for the object representing the particle system. To allow for further development, the particle_system class will have methods only general to all particle systems.

class particle_system(ABC): # abstract base particle class, this is used to pass the methods into each particle system.
    def __init__(self):
        pass

    @abstractmethod
    def get_system_observables(self, H): # creates the 1D vectors and meshgrid to form a linear space.
        pass
```

```
# The class singleparticle is used for the object representing any single particle system. It requires the mass of the particle and the spin to instantiate, but will assume the mass to be m_e and the spin to be None if not provided.
# Methods here are: get_system_observables - x

class singleparticle(particle_system):
    def __init__(self, m=m_e): # this initialiser assumes a single particle has the mass of an electron if left blank.
        self.m=m # mass of particle to be modelled

    def get_system_observables(self, H): # returns a meshgrid which functions as a composition of the linear spaces needed to represent the system
        #create meshgrid of space equivalent to hamiltonian based on 2D
        x1 = np.linspace(-H.spextent/2, H.spextent/2, H.N)
        y1 = np.linspace(-H.spextent/2, H.spextent/2, H.N)
        self.x1, self.y1 = np.meshgrid(x1, y1) # The meshgrid is a co-ordinate matrix of the numpy arrays provided.
        print(self.x1) #WB 1.2.2

# The class twoparticle is used for the object representing any single particle system. It requires the mass of the particle to instantiate, but will assume the mass to be m_e if not provided.

class twoparticles(particle_system):
    def __init__(self, m=m_e):
        self.m=m # mass

    def geteigenstates():
        return

    def get_system_observables(self, H): # returns a meshgrid which functions as a composition of the linear spaces needed to represent the system
        #create meshgrid of space equivalent to hamiltonian based on 2D - 4 1D arrays for 2D, each.
        x1 = np.linspace(-H.spextent/2, H.spextent/2, H.N)
        y1 = np.linspace(-H.spextent/2, H.spextent/2, H.N)
        x2 = np.linspace(-H.spextent/2, H.spextent/2, H.N)
        y2 = np.linspace(-H.spextent/2, H.spextent/2, H.N)
        self.x1, self.y1, self.x2, self.y2 = np.meshgrid(x1, y1, x2, y2) # The meshgrid is a co-ordinate matrix of the numpy arrays provided.
        #print(self.x1) #WB 1.2.2


# The class Hamiltonian represents the operator object corresponding to the total energy of the system. The Hamiltonian is a collation of the informtion needed to create a wavefunction independent of the particle position.

class Hamiltonian:
    def __init__(self, particles, potential, pot_time, pot_amp, gridpoints, spextent, spatial_ndim, E_min=0):
        self.N = gridpoints # number of gridpoints to use
        self.spextent = spextent # spatial extent of the system, in angstroms
        self.dx = self.spextent / self.N # space width, estimate for spatial derivative operator for x
        self.dy = self.spextent / self.N # space width, estimate for spatial derivative operator for y
        self.dV = self.dx * self.dy # 2D derivative operator
        self.particle_system = particles # particle system being used
```

```
        self.particle_system.H = self # the Hamiltonian of the particle system, for easy reference
        self.spatial_ndim = spatial_ndim # why? – unused but could be used in future methods
        self.ndim = 4  # total number of observables
        self.potential = potential # assigns the potential type
        self.pot_time_period = pot_time # time period of harmonic oscillator, if present
        self.pot_max_amp = pot_amp # max amplitude of harmonic oscillator, if present

        # The following method calls are part of the constructor.

        self.particle_system.get_system_observables(self) # initialise the obervables, uses a particle method

        self.V = self.getpotentialasamatrix() # gets the potential in a matrix form in order to

    def getpotentialasamatrix(self): # converts potential into a matrix
        V = self.potential(self.particle_system, self.pot_time_period, self.pot_max_amp) # gets the potential
from the global method as defined by the type of potential required.
        self.Vgrid = V # The global potential routine returns the potential as a function of the particle position
in space, which is stored as a grid.
        self.E_min = np.amin(V) # determines the minimum energy value - needed to animate the function
        if type(self.particle_system) == singleparticle:
            V = V.reshape(self.N, self.N) # shapes the potential into a N^2 point matrix
        elif type(self.particle_system) == twoparticles:
            V = V.reshape(self.N, self.N, self.N, self.N) # shapes the potential into a N^4 point matrix
        print(V) #WB 2.2.1
        return V # returns the generated matrix in an array form


# The class Simulation takes in the time parameters to initialise the time conditions for the simulation.

class Simulation:
    def __init__(self, H, total_time, store_steps):
        # simple creation of attributes assigned to object based on input parameters
        self.dt  = total_time/store_steps # width of time step
        self.total_time = total_time # total time to run simulation
        self.store_steps = store_steps # steps to store
        self.H = H # associated Hamiltonian
        print(self.dt, " - dt, ", self.total_time, " - total time, ", self.store_steps, " - steps to store.") # WB 4.1.1

    def sim(self, psi, Hmt): # The sim method takes in a wavefunction and a Hamiltonian to add to the
simulation object and evolve using a specified method. In this case, the only method is the
split_step_fourier.
        time_evolution = split_step_fourier(self) # instantiates an object of type split_step_fourier
        self.results = time_evolution.run(psi, Hmt) # uses the run() method of the time evolution object
created, passing in an initial wavefunction and a Hamiltonian to process the time evolution.
        print(self.results[0], self.results[int(self.store_steps/2)], self.results[self.store_steps], " - evolved wfcs at t
= 0, t = t/2 and t = t") # WB 4.2.1
        return self.results # returns the array storing the calculated wavefunctions.


# The class split_step_fourier is a method to carry out the time evolution of the simulation. It takes in a
simulation object and evolves the associated wavefunction for the number of timesteps required.

class split_step_fourier:
    def __init__(self, simulation):
        self.simulation = simulation # assigns simulation as an attribute
```

```
        self.simulation.Vmin = np.amin(self.simulation.H.Vgrid) # finds minimum potential
        self.simulation.Vmax = np.amax(self.simulation.H.Vgrid) # finds maximum potential

    def run(self, wfc, Hm): # carries out the time evolution
        self.Hm = Hm # assigns the Hamiltonian attribute
        steps_to_make = self.simulation.store_steps # takes in the number of steps to carry out from the simulation
        dt = self.simulation.dt # takes in the step width to use from the simulation
        total_time = self.simulation.total_time # takes in the total time to evolve over from the simulation
        dt_store = total_time/dt # the number of steps to take by width
        dt = dt_store # time step width
        psi = [wfc] # import initial wavefunction
        c_0 = 1 # constant for split-operator approximation to GPE
        c_1 = -c_0 # constant for split-operator approximation to GPE
        c_2 = 1 # constant for split-operator approximation to GPE
        g = 4*np.pi*hbar**2*a_s/self.Hm.particle_system.m # coupling constant
        potential = self.Hm.V # takes the potential matrix from the Hamiltonian

        for i in tqdm(range(0, steps_to_make), desc = "Progress (of steps taken): "): # tqdm wrapper for progress bar + iteration for number of steps to carry out
            TPSI = [] # temporary array to store wavefunctions being manipulated
            TPSI.append(psi[i]) # add the current array psi into TPSI

            operatorone = np.exp(-0.5j*dt*(potential+(g*np.abs(TPSI[0]**2)))) # first operator, for position space
            operatortwo = np.exp(0.5j*dt)*(ndimage.laplace(TPSI[0])) # second operator, for momentum space

            TPSI.append(operatorone*(np.fft.fftn(TPSI[0]))) # TPSI[1] is psi_1, the first wavefunction from the partial operator application, here psi is stored in momentum space using the fft.fftn n-dimensional fourier transform
            if i == 1:
                print(TPSI[1], " - step 1 for Test 5.2.1a") # WB Test 5.2.1

            TPSI.append(operatortwo*(np.fft.ifftn(TPSI[1]))) # TPSI[2] is psi_2, the second wavefunction from the partial operator application, here psi is stored in real space using the fft.ifftn n-dimensional inverse fourier transform
            if i == 1:
                print(TPSI[2], " - step 2 for Test 5.2.1b") # WB Test 5.2.1
            operatorthree = np.exp((-0.5j*dt*(potential+g*np.abs(c_0*TPSI[0]+c_1*TPSI[1]+c_2*TPSI[2])**2))) # the third operator is now built - as it uses psi_1 and psi_2 as part of it's definition, the first two operations had to be carried out first.

            TPSI.append(np.fft.ifftn(operatorthree*(TPSI[1]))) # the third operator can now be applied to the psi_1 wavefunction
            if i == 1:
                print(TPSI[3], " - step 3 for Test 5.2.1c") # WB Test 5.2.1
            norm = GPE.calc_normalisation_constant(self.Hm, TPSI[3]) # uses the GPE method to calculate a normalisation constant to the wavefunction.
            final_psi = GPE.renormalise(self.Hm, norm, TPSI[3]) # uses the GPE method to normalise the final wavefunction
            if i == 1:
                print("Normalisation constant: ", norm) # WB 3.2.1
                print(self.Hm.dV * (np.sum(np.abs(final_psi)))**2, " - area under normalised wavefunction, should be 1.") # WB 3.2.1
            psi.append(final_psi) # appends the final wavefunction to the psi array
```

```
        self.simulation.psi_max = np.amax(np.abs(psi)) # finds the maximum of psi (for plot)
        return psi # returns psi - the array full of stored steps

# Method for defining the potential, here a potential correlating to a harmonic oscillator is used.

def harmonicoscillator(particle_system, pot_time, pot_amp):
    T = pot_time * femtoseconds # time period, Enter a suitable time period for the harmonic oscillator in
femtoseconds
    w = np.pi*2/T # omega
    A = pot_amp * Å / femtoseconds * w # max magnitude of the oscillation
    if type(particle_system) == singleparticle:
        PE = 0.5 * A**2 * particle_system.x1**2 + 0.5 * A**2 * particle_system.y1**2 # total potential energy
from position in harmonic oscillator
    elif type(particle_system) == twoparticles:
        PE = 0.5 * A**2 * particle_system.x1**2 + 0.5 * A**2 * particle_system.y1**2 + 0.5 * A**2 *
particle_system.x2**2 + 0.5 * A**2 * particle_system.y2**2
    return PE
# In order to define the potential of the grid, we use the particle mass, time period (if oscillatory), omega. k
is m*omega².
# This subroutine will return the energy due to a particles position in a harmonic oscillator.


# The object visualise_wavefunction contains methods to plot the simulation data using matplotlib in a
tkinter GUI.

class visualise_wavefunction():
    def __init__(self):
        self.pause_status = False # pause status of animation
        self.restart = False # restart status of animation
        self.mpeg_write_status = False # video writing status of animation
        self.download_count = 0
        print(self.pause_status, " - pause status, ", self.restart, " - restart status, ", self.mpeg_write_status, " -
video writing status, ", self.download_count, " - download count.") # WB 6.1.1

    def complex_to_rgb(self, Z: np.ndarray, max_val: float = 1.0) -> np.ndarray:
        mod = np.abs(Z) # takes modulus of complex value
        arg = np.angle(Z) # takes argument of complex value
        h = (arg + np.pi)  / (2 * np.pi) # hue value determined by polar mapping of argument
        #s = np.reshape((r/self.simulation.psi_max), (256, 256)) - this is the precise mapping of the colour
space, but it leads to washed out colours and too low values of saturation to see clearly.
        #v = np.reshape((r/self.simulation.psi_max), (256, 256))
        s = np.ones(h.shape) # fills s, v with the value 1 - which is not colour accurate but for the purposes of
the simulation provides a much higher quality output.
        v = np.ones(h.shape)
        rgb = hsv_to_rgb(np.moveaxis(np.array([h,s,v]) , 0, -1)) # moves h, s, v into an array, rearranges the axis
and uses the hsv_to_rgb routine from matplotlib to convert the HSV value obtained into an RGB value.

        abs_z = mod / max_val # Takes the wavefunction saturation from the function call and sets the
maximum value of the magnitude of the complex value to the
        abs_z = np.where(abs_z> 1., 1. ,abs_z) # where the magnitude of the complex number is greater than
1, replace abs_z with 1
        return np.concatenate((rgb, abs_z.reshape((*abs_z.shape,1))), axis= (abs_z.ndim))

    def animate(self, figuresize=(10, 10), animation_duration = 5, fps = 30, potential_saturation=0.8,
wavefunction_saturation=0.8):
```

```
        total_frames = int(fps * animation_duration) # total number of frames required
        sim_dt = self.simulation.total_time/total_frames # time to display each frame for
        self.simulation.psi_plotarea = self.simulation.results/self.simulation.psi_max # makes a suitable plot
area as an attribute of the simulation
        graphfig = plt.figure(figsize=figuresize) # creates a figure
        axes = graphfig.add_subplot(111) # adds axes to the figure as part of a subplot arrangement
        plt.style.use("dark_background") # background colour of simulation graph



        length = self.simulation.H.spextent/Å # length of system
        # plot of the potential on the axes of the potential grid over the range of the potential system, using a
grey colourmap and using bilinear interpolation into
        plot_potential = axes.imshow((self.simulation.H.Vgrid + self.simulation.Vmin)/(self.simulation.Vmax-
self.simulation.Vmin), vmax = 1.0/potential_saturation, vmin = 0, cmap = "gray", origin = "lower",
interpolation = "bilinear", extent = [-length/2, length/2, -length/2, length/2])
        # plot of the wavefunction
        plot_wavefunction = axes.imshow(self.complex_to_rgb(self.simulation.psi_plotarea[0], max_val=
wavefunction_saturation), origin = "lower", interpolation = "bilinear", extent = [-length / 2, length/ 2,-length
/ 2,length / 2])
        print(self.complex_to_rgb(self.simulation.psi_plotarea[0], max_val= wavefunction_saturation), " - initial
RGB of wavefunction") # WB 6.2.1

        axes.set_title("Dispersion of Helium-4 boson in R^2, psi(x, y, t)") # axes title
        axes.set_xlabel("x, angstroms") # x label
        axes.set_ylabel("y, angstroms") # y label

        #time label change
        time_ax = axes.text(0.96, 0.96, "",  color = "white", transform=axes.transAxes, ha="right", va="bottom")
        time_ax.set_text(u"t = {} femtoseconds".format("%.3f"  % 0.00))

        animation_data = {'t': 0.0, 'ax':axes ,'frame' : 0}
        def func_animation(*arg): # increments the time, and calculates the next frame to display by
calculating a frame for each time step, sim_dt. Used for the FuncAnimation call
            if self.restart == True:
                self.restart = False
                animation_data['t'] = 0.0 # reset the animation time to 0
            if not self.pause_status: # stops next frame being calculated when pasued
                time_ax.set_text(u"t = {} femtoseconds".format("%.3f"  % (animation_data['t']/femtoseconds))) #
update time axis value

                animation_data['t'] = animation_data['t'] + sim_dt # increment time
                if animation_data['t'] > self.simulation.total_time:
                    animation_data['t'] = 0.0 # replays simulation when toal time reached

                animation_data['frame'] +=1 # increments frame number
                index = int((self.simulation.store_steps)/self.simulation.total_time * animation_data['t']) # t/dt

                plot_wavefunction.set_data(self.complex_to_rgb(self.simulation.psi_plotarea[index], max_val=
wavefunction_saturation)) # the potential of the wavefunction is plotted over the plot area
                return plot_potential, plot_wavefunction, time_ax
            else:
                return plot_potential, plot_wavefunction, time_ax
```

```
        # call to animation function, takes in figure, animation function, frames and the interval to display the frame for. Uses blitting, a procedure outlined in the design for the animation function.
        self.a = animation.FuncAnimation(graphfig, func_animation, blit=True, frames=total_frames, interval= 1/fps * 1000) # Call to FuncAnimation with the parameters: graphfig - the plot figure, func_animation - the user defined function describing how to animate the simulation
        # blitting - renders all non-changing points in the animation into a single background image and renders it in one go.

        # creating the Tkinter canvas
        # containing the Matplotlib figure
        self.canvas = FigureCanvasTkAgg(graphfig, master = self.root)
        self.canvas.draw()

        # placing the canvas on the Tkinter window
        self.canvas.get_tk_widget().pack()

        # creating the Matplotlib toolbar
        self.toolbar = NavigationToolbar2Tk(self.canvas, self.topframe)
        self.toolbar.update()

        # placing the toolbar on the Tkinter window
        self.canvas.get_tk_widget().pack()

    def mpeg_write(self): # writes video to mped
        self.mpeg_write_status = not self.mpeg_write_status # set mpeg write status
        print(self.mpeg_write_status, " - updated mpeg write status") # WB 6.1.1
        self.ToggleRestartStatus()
        if self.pause_status:
            self.TogglePauseStatus()
        Writer = animation.writers['ffmpeg'] # use ffmpeg as the writer
        writer = Writer(fps=30, bitrate=1800) # write file using given parameters

self.a.save(r"C:\Users\ismail\Videos\Simulation_Downloads\animation"+str(self.download_count)+".mp4", writer=writer) # write animation to file using defined writer
        self.download_count += 1
        print(self.download_count, " - updated download count")
        self.mpeg_write_status = not self.mpeg_write_status # unset mpeg write status
        print(self.mpeg_write_status, " - updated mpeg write status") # WB 6.1.1


    def TogglePauseStatus(self):
        self.pause_status = not self.pause_status # changes pause status
        print(self.pause_status, " - new pause status") # WB 6.1.1

    def ToggleRestartStatus(self):
        self.restart = not self.restart # changes restart status
        print(self.restart, " - restart status change") # WB 6.1.1


    def InputDialogue(self):
        self.inputwindow = Tk()
        self.inputwindow.geometry("450x600")
        self.inputwindow.title("Input Dialogue")
        grd_options = [
            "128",
```

```
            "256",
            "512",
        ]

        # datatype of menu text
        grd_clicked = StringVar()

        # initial menu text
        grd_clicked.set("128")

        # Create Dropdown menu
        grd_drop = OptionMenu(self.inputwindow, grd_clicked, *grd_options )
        grd_drop.pack()

        # Create Label
        self.gridpoints = Label(self.inputwindow, text = " " )
        self.gridpoints.pack()

        # Create button, it will change label text
        grd_button = Button(self.inputwindow, text = "Update", command = partial(self.show, self.gridpoints,
grd_clicked)).pack()

        ### NUMBER OF PARTICLES

        # Dropdown menu options
        par_options = [
            "1",
            "2 - Superposing",
        ]

        # datatype of menu text
        par_clicked = StringVar()

        # initial menu text
        par_clicked.set("1")

        # Create Dropdown menu
        par_drop = OptionMenu(self.inputwindow, par_clicked, *par_options )
        par_drop.pack()

        # Create Label
        self.particles = Label(self.inputwindow, text = " " )
        self.particles.pack()

        # Create button, it will change label text
        par_button = Button(self.inputwindow, text = "Update", command = partial(self.show, self.particles,
par_clicked)).pack()


        ### POTENTIAL

        # Dropdown menu options
        pot_options = [
          "Harmonic Oscillator",
          "No potential",
```

```
        ]
        # datatype of menu text
        pot_clicked = StringVar()

        # initial menu text
        pot_clicked.set("Harmonic Oscillator")

        # Create Dropdown menu
        pot_drop = OptionMenu(self.inputwindow, pot_clicked, *pot_options )
        pot_drop.pack()

        # Create Label
        self.potential = Label(self.inputwindow, text = " " )
        self.potential.pack()

        # Create button, it will change label text
        pot_button = Button(self.inputwindow, text = "Update", command = partial(self.show, self.potential,
pot_clicked)).pack()

        ### TIME_PARAMETERS

        time_label = Label(self.inputwindow, text = "Enter total time to run the simulation, in femtoseconds
(example: 2)")
        time_label.pack()
        self.time_slider = Scale(self.inputwindow, from_=0.1, to=10, resolution=0.1, orient = HORIZONTAL)
        self.time_slider.pack()

        steps_label = Label(self.inputwindow, text = "Enter amount of timesteps to carry out (example: 1000)")
        steps_label.pack()
        self.timesteps_slider = Scale(self.inputwindow, from_=100, to=10000, resolution = 100, orient =
HORIZONTAL)
        self.timesteps_slider.pack()

        osc_time_label = Label(self.inputwindow, text = "Enter a time period for the harmonic oscillator
(example: 1)")
        osc_time_label.pack()
        self.osc_time_slider = Scale(self.inputwindow, from_=0.0, to=5, resolution=0.1, orient = HORIZONTAL)
        self.osc_time_slider.pack()

        osc_amp_label= Label(self.inputwindow, text = "Enter amplitude of harmonic oscillator in angstroms
(example: 50)")
        osc_amp_label.pack()
        self.osc_amp_slider = Scale(self.inputwindow, from_=1, to=80, orient = HORIZONTAL)
        self.osc_amp_slider.pack()


        create_button = Button(self.inputwindow,
                    command = partial(self.create_redraw, self.inputwindow),
                    height = 2,
                    width = 15,
                    text = "Create Sim button")
        create_button.pack(side = BOTTOM)
        self.inputwindow.attributes('-topmost',True)
        self.inputwindow.mainloop()
```

```
    def show(self, labeltoupdate, gettertouse):
        labeltoupdate.config( text = gettertouse.get() )


    def create_redraw(self, window):
        if self.particles["text"] != "1":
            self.gridpoints["text"] = "128"
            print("Two particle simulation only available with 128 gridpoints.")
        redraw_button = Button(window,
                    command = partial(self.update_animation, self.gridpoints["text"], self.potential["text"],
self.osc_time_slider.get(), self.osc_amp_slider.get(), self.time_slider.get(), self.particles["text"], 20,
self.timesteps_slider.get()),
                    height = 2,
                    width = 10,
                    text = "Execute")
        redraw_button.pack(side = BOTTOM)

    def update_animation(self, gridlabel, potentiallabel, hmtimeperiod, hmamplitude, totaltimelabel,
particlesystemtypelabel, spatialextentlabel, timestepstodolabel):
        if hasattr(self, "canvas"):
            self.canvas.get_tk_widget().destroy()
            #for widget in self.toolbar.winfo_children():
            #    widget.destroy()
        self.inputwindow.destroy()
        #empty frames
        for widget in self.leftframe.winfo_children():
            widget.destroy()
        for widget in self.rightframe.winfo_children():
            widget.destroy()
        for widget in self.topframe.winfo_children():
            widget.destroy()
        if particlesystemtypelabel == "1":
            particlesystemtype = singleparticle
        else:
            particlesystemtype = twoparticles

        if potentiallabel == "Harmonic Oscillator":
            potential = True
        else:
            potential = False

        print("Creating particle system...")
        sys = particlesystemtype()
        print("Done")

        print("Creating Hamiltonian...")
        if potential:
            H = Hamiltonian(sys, harmonicoscillator, float(hmtimeperiod), int(hmamplitude),  int(gridlabel),
int(spatialextentlabel)*Å, 2)

        elif not potential:
```

```
            H = Hamiltonian(sys, harmonicoscillator, 0, 0,  int(gridlabel), int(spatialextentlabel)*Å, 2)
        print("Done")

        print("Formulating initial wavefunction...")
        if type(sys) == singleparticle:
            wfc = GPE([H.N, H.N], H)
            psi = wfc.initialsinglewavefunction(H)
        elif type(sys) == twoparticles:
            wfc = GPE([H.N, H.N, H.N, H.N], H)
            psi = wfc.initialdualwavefunction(H)
        print("Done")

        print("Beginning time evolution...")
        total_time = float(totaltimelabel) * femtoseconds
        simulationone = Simulation(H, float(totaltimelabel)*femtoseconds, int(timestepstodolabel))

        evolvedwfcs = simulationone.sim(psi, H)
        print("Done")
        self.simulation = simulationone
        self.H = simulationone.H # creates Hamiltonian attribute
        self.animate((10, 10), 10, 30, 0.8, 0.8)
        self.Buttons()
        self.DisplayParameters()
        self.InputDialogue()


    def Buttons(self):
        # Buttons
        restart_button = Button(self.leftframe,
                        command = self.ToggleRestartStatus,
                        height = 2,
                        width = 10,
                        text = "Restart")
        restart_button.pack(side = TOP)

        pause_button = Button(self.leftframe,
                        command = self.TogglePauseStatus,
                        height = 2,
                        width = 10,
                        text = "Pause")
        pause_button.pack(side = TOP)

        download_button = Button(self.leftframe,
                        command = self.mpeg_write,
                        height = 2,
                        width = 10,
                        text = "Download MP4")
        download_button.pack()
        return

    def DisplayParameters(self):
        # parameter labelling

        N_lbl=Label(self.rightframe, text="Gridpoints: " + str(self.simulation.H.N))
        N_lbl.pack(side = TOP)
```

```
        if self.simulation.H.particle_system.m == 1:
            masstype = "Electron Mass"
        else:
            masstype = "Helium Boson Mass"

        parsys_lbl=Label(self.rightframe, text="Particle system type: " +
str(self.simulation.H.particle_system.__class__.__name__) + "     Mass: " + masstype)
        parsys_lbl.pack(side = TOP)

        pot_lbl=Label(self.rightframe, text="Potential type: " + str(self.simulation.H.potential.__name__))
        pot_lbl.pack()

        if self.simulation.H.potential == "harmonicoscillator":
            pottmp_lbl=Label(self.rightframe, text="Time Period of Harmonic Oscillator: " +
str(self.simulation.H.pot_time_period))
            pottmp_lbl.pack()
            potamp_lbl=Label(self.rightframe, text="Maximum Amplitude of Harmonic Oscillator: " +
str(self.simulation.H.pot_max_amp))
            potamp_lbl.pack()

        sim_time_lbl=Label(self.rightframe, text="Simulation Time: " +
str(self.simulation.total_time/femtoseconds)+ " fms")
        sim_time_lbl.pack()

        time_steps_lbl=Label(self.rightframe, text="Number of timesteps: " + str(self.simulation.store_steps))
        time_steps_lbl.pack(side = BOTTOM)
        return

    def MainWindow(self):
        # GUI
        # visualisation
        # create root window
        self.root = Tk()

        # root window title and dimension
        self.root.title("Simulation")
        #Set geometry (widthxheight)
        self.root.geometry('1920x1080')

        self.frame = Frame(self.root)
        self.frame.pack()

        self.leftframe = Frame(self.root)
        self.leftframe.pack(side = LEFT)

        self.rightframe = Frame(self.root)
        self.rightframe.pack(side = RIGHT)

        self.topframe = Frame(self.root)
        self.topframe.pack(side = TOP)

        self.InputDialogue()
        self.root.mainloop()
```

```
# Main program

print("SSFT Particle Simulator - Enter the desired parameters to run a simulation.")
print("Testing Mode: ") # White-Box Testing statement
# Instantiate visualisation object
graph = visualise_wavefunction()
# Create simulation window
graph.MainWindow()

# Main Program END
```

References

[All references are in the Harvard Referencing Format, author information is provided where

[1] Braley, C. and Sandu, A. (no date) Fluid Simulation for Computer Graphics: A tutorial in grid based and ... Available at: https://cg.informatik.uni-freiburg.de/intern/seminar/gridFluids_fluid-EulerParticle.pdf (Accessed: 04 August 2023).

[2] Building collision simulations: An introduction to computer graphics (2021) YouTube. Available at: https://www.youtube.com/watch?v=eED4bSkYCB8&ab_channel=Reducible (Accessed: 04 August 2023).

[3] Gangardt, D. (no date) Department of Physics, Bose-Einstein Condensation in Ultra Cold Gases. Available at: https://warwick.ac.uk/fac/sci/physics/mpags/modules/theory/becucg (Accessed: 04 August 2023).

[4] Jones, A. (2021) Visualizing Differential Equations in python, Andy Jones. Available at: https://andrewcharlesjones.github.io/journal/differential-equation-viz.html (Accessed: 04 August 2023).

[5] Moorehead, J. (2023) Masters degree show 2016 - smooth particle hydrodynamics fluid solver, Vimeo. Available at: https://vimeopro.com/nccaanimation/mastersdegreeshow2016/video/185476692 (Accessed: 04 August 2023).

[6] Stagg, G. (no date) WebGL superfluid simulation using dGPE, WebGL GPE. Available at: https://georgestagg.github.io/webgl_gpe/ (Accessed: 04 August 2023).

[7] Tsuzuki, S. (2021) ArXiv:2105.03177v3 [physics.flu-dyn]. Available at: https://arxiv.org/pdf/2105.03177.pdf (Accessed: 04 August 2023).

[8] Xagoraris, K. (no date) Superfluids and Fluid Dynamics, Kate Xagoraris. Available at: https://www.katexagoraris.com/super-fluids (Accessed: 04 August 2023).

[9] Nakayama, A. and Makri, N. (2005) Simulation of dynamical properties of normal and superfluid helium, PNAS. Available at: https://www.pnas.org/doi/10.1073/pnas.0501127102 (Accessed: 04 August 2023).

[10] Xagoraris, K. (no date) Hydrodynamics, liquid crystals, and VFX, Kate Xagoraris. Available at: https://www.katexagoraris.com/hydrodynamics-liquid-crystals-and-v (Accessed: 30 September 2023).

[11] Ecole de Physique des Houches (2019) Quantum gases and superfluidity, lecture 1, YouTube. Available at: https://www.youtube.com/watch?v=oT3ueGAQa2Y (Accessed: 30 September 2023).

[12] Reducible (2021) Building collision simulations: An introduction to computer graphics, YouTube. Available at: https://www.youtube.com/watch?v=eED4bSkYCB8 (Accessed: 30 September 2023).

[13] Bruce, R. et al. (2017) 3D CFD Transient Numerical Simulation of Superfluid Helium, IOP Science. Available at: https://iopscience.iop.org/article/10.1088/1757-899X/278/1/012057/pdf (Accessed: 30 September 2023).

[14] G, P. (2022) Why Quantum Mechanics Makes No Sense (but still works) - collapse of the wave function (Parth G), YouTube. Available at: https://www.youtube.com/watch?v=Is_QH3evpXw (Accessed: 30 September 2023).

[15] Tajima, H. (2021) Gross-Pitaevskii equation(1), YouTube. Available at: https://www.youtube.com/watch?v=_gvfbahcqsk (Accessed: 30 September 2023).

[16] Solver, M.P. (2021) Time-dependent Schrodinger equation in python: Two different techniques, YouTube. Available at: https://www.youtube.com/watch?v=kVjg3jbM3Pw (Accessed: 30 September 2023).

[17] Strang, G. and Herman, E.J. (2022) 16.5: Divergence and curl, Mathematics LibreTexts. Available at: https://math.libretexts.org/Bookshelves/Calculus/Calculus_(OpenStax)/16%3A_Vector_Calculus/16.05%3A_Divergence_and_Curl (Accessed: 30 September 2023).

[18] Muruganandam, P. et al. (2016) OpenMP fortran and C programs for solving the time-dependent gross–pitaevskii equation in an anisotropic trap, Computer Physics Communications. Available at: https://www.sciencedirect.com/science/article/abs/pii/S001046551630073X (Accessed: 30 September 2023).

[19] Comput. Phys. Commun. 184, 201-208 (2013)

[20] Wikipedia (2023) Crank–Nicolson method, Wikipedia. Available at: https://en.wikipedia.org/wiki/Crank%E2%80%93Nicolson_method (Accessed: 30 September 2023).

[21] Wikipedia (2023b) Gross–Pitaevskii equation, Wikipedia. Available at: https://en.wikipedia.org/wiki/Gross%E2%80%93Pitaevskii_equation (Accessed: 30 September 2023).

[22] Wikipedia (2023b) Finite difference, Wikipedia. Available at: https://en.wikipedia.org/wiki/Finite_difference#Relation_with_derivatives (Accessed: 30 September 2023).

[23] Wikipedia (2023a) Correlation function (statistical mechanics), Wikipedia. Available at: https://en.wikipedia.org/wiki/Correlation_function_(statistical_mechanics) (Accessed: 30 September 2023).

[24] Wikipedia (2023e) Path integral formulation, Wikipedia. Available at: https://en.wikipedia.org/wiki/Path_integral_formulation (Accessed: 30 September 2023).

[25] Wikipedia (2023e) Helium Atom, Wikipedia. Available at: https://en.wikipedia.org/wiki/Helium_atom (Accessed: 30 September 2023).

[26] Ackland, G. (2020) 14.1: Impact parameter and classical analogies, Physics LibreTexts. Available at: https://phys.libretexts.org/Bookshelves/Quantum_Mechanics/Quantum_Physics_(Ackland)/14%3A_Using_Partial_Waves/14.01%3A_Impact_Parameter_and_Classical_Analogies (Accessed: 30 September 2023).

[27] Collimator (no date) What is the finite difference method?, Collimator. Available at: https://www.collimator.ai/reference-guides/what-is-the-finite-difference-method (Accessed: 30 September 2023).

[28] Wikipedia (2022) Adaptive mesh refinement, Wikipedia. Available at: https://en.wikipedia.org/wiki/Adaptive_mesh_refinement (Accessed: 30 September 2023).

[29] Fuenta, R. de la (2022) Quantum-visualizations/qmsolve: ⚛ a module for solving and visualizing the schrödinger equation., GitHub. Available at: https://github.com/quantum-visualizations/qmsolve (Accessed: 30 September 2023).

[30] (No date) 3. numerically solving PDE's: Crank-Nicholson algorithm. Available at: https://www.sfu.ca/~rjones/bus864/notes/notes2.pdf (Accessed: 30 September 2023).

[31] Wikipedia (2023) Quantum Harmonic Oscillator, Wikipedia. Available at: https://en.wikipedia.org/wiki/Quantum_harmonic_oscillator (Accessed: 05 November 2023).

[32] Wikipedia (2023a) Potential energy, Wikipedia. Available at: https://en.wikipedia.org/wiki/Potential_energy (Accessed: 05 November 2023).

[33] Wikibooks (no date) Modern physics/potential momentum, Wikibooks, open books for an open world. Available at: https://en.wikibooks.org/wiki/Modern_Physics/Potential_Momentum (Accessed: 05 November 2023).

[34] Wikipedia (2023c) Wave packet, Wikipedia. Available at: https://en.wikipedia.org/wiki/Wave_packet (Accessed: 05 November 2023).

[35] Wikipedia (2023a) Cyclotron, Wikipedia. Available at: https://en.wikipedia.org/wiki/Cyclotron (Accessed: 05 November 2023).

[36] School of Physics and Astronomy (2020) Chemical potential μ - treatment of open systems, School of Physics, Edinburgh University. Available at: https://www2.ph.ed.ac.uk/~gja/thermo/course_notes/topic17.pdf (Accessed: 05 November 2023).

[37] numpy (no date) Numpy user guide, NumPy user guide - NumPy v1.26 Manual. Available at: https://numpy.org/doc/stable/user/index.html#user (Accessed: 05 November 2023).

[38] Wikipedia (2023b) Hilbert space, Wikipedia. Available at: https://en.wikipedia.org/wiki/Hilbert_space (Accessed: 05 November 2023).

[39] Wikipedia (2023e) Split-step method, Wikipedia. Available at: https://en.wikipedia.org/wiki/Split-step_method (Accessed: 05 November 2023).

[40] Wikipedia (2023a) Baker–Campbell–Hausdorff formula, Wikipedia. Available at: https://en.wikipedia.org/wiki/Baker%E2%80%93Campbell%E2%80%93Hausdorff_formula (Accessed: 05 November 2023).

[41] Wikipedia (2023d) LOBPCG, Wikipedia. Available at: https://en.wikipedia.org/wiki/LOBPCG (Accessed: 05 November 2023).

[42] Sanwo, S. (2022) How to set up a virtual environment in python – and why it's useful, freeCodeCamp.org. Available at: https://www.freecodecamp.org/news/how-to-setup-virtual-environments-in-python/ (Accessed: 05 November 2023).

[43] Wikipedia (2023c) Electromagnetism, Wikipedia. Available at: https://en.wikipedia.org/wiki/Electromagnetism (Accessed: 05 November 2023).

[44] Wikipedia (2023d) Euler method, Wikipedia. Available at: https://en.wikipedia.org/wiki/Euler_method (Accessed: 05 November 2023).

[45] Wikipedia (2023b) Crank–nicolson method, Wikipedia. Available at: https://en.wikipedia.org/wiki/Crank%E2%80%93Nicolson_method (Accessed: 05 November 2023).

[46] Arfken, G.B., Weber, H.-J. and Harris, F.E. (2021) '16 - Integral Equations', in Mathematical methods for physicists: A comprehensive guide. New Delhi: Elsevier.

[47] Xu, K., 2004. Generalized Gradient Approximation Made Simple. Physical Review Letters, 92(21), p.213201. Available at: https://arxiv.org/pdf/cond-mat/0411154.pdf (Accessed: 05 November 2023).

[48] Rawat, S. (2024) Sachinrawat2207/QUTARANG_DEP: A GPU-enabled Gross-Pitaevskii equation solver. GitHub. Available at: https://github.com/sachinrawat2207/quTARANG_dep?tab=readme-ov-file#capablities-of-qutarang (Accessed: 18 March 2024).

[49] PaNOSC (no date) *MyHDF5 · EXPLORE & visualize HDF5 files, myHDF5 · Explore & Visualize HDF5 Files*. Available at: https://myhdf5.hdfgroup.org/ (Accessed: 23 March 2024).

[50] *Visual Quantum Mechanics* (no date) *Homepage of Visual Quantum Mechanics*. Available at: https://vqm.uni-graz.at/pages/colormap.html (Accessed: 24 March 2024).

[51] *Matplotlib.animation.funcanimation#* (no date) *matplotlib.animation.FuncAnimation - Matplotlib 3.8.3 documentation*. Available at: https://matplotlib.org/stable/api/_as_gen/matplotlib.animation.FuncAnimation.html (Accessed: 24 March 2024).