An investigation into N-body simulations using the Python virtual environment

Patrick John Welsh

BSc (Hons) Computing, 2022

School of Design and Informatics
Abertay University

Table of Contents

Table of Figures

Table of Tables

Acknowledgements

I would like to sincerely thank my honours project supervisor, Dr Ethan Bayne for his steadfast direction and guidance. Without his assistance and support, I am not sure my project would have been successful.

I also want to thank my wife, Ariel Welsh, from the bottom of my heart, who has been my rock and best friend throughout my life and higher education studies. My success is your success. Without you, this would not have been possible. Thank you eternally. I would also like to thank my best friend Rene, and his partner Stephanie. You have been there through it all, thank you for your support and eternal friendship.

Structured Abstract

Context

Simulations are an extremely effective way to visualize how astronomical bodies interact and help to solve the ever-evolving problem of how the universe operates. As a species, the technology to consistently send vessels out into the vast expanse of interstellar space doesn’t currently exist, as such, simulations are an excellent way of being able to take the best guess at how such bodies interact, using existing scientific and mathematical equations. However, there are numerous methods to create such simulations, and it can be difficult to identify the most effective way to proceed. One such method is that of an N-body simulation.

Aim

This project seeks to conduct an investigation into N-body simulations via the development of multithreaded and non-multithreaded versions of two N-body simulation techniques to compare their performance and identify the faster algorithm. The simulations will be developed in a Python virtual environment. One of which will be a Tree Code, particularly, a Barnes-Hut Tree and the other, a naïve Pairwise Interaction, also referred to as an “All-Pairs N-body” algorithm.

Methodology

N-body simulations show how bodies interact under a force, such as gravity. Python was chosen due to its abundance of libraries and ability to configure virtual environments. Each simulation will be engineered to mimic a *“dynamical system”* of objects under the influence of gravity. They will then be compared against each other via monitoring several metrics to calculate their run-time performance.

Results

Upon comparison of the two simulations, it is expected that the results will demonstrate the Tree Code algorithm to perform faster and thus be the optimum solution.

Conclusion

Comparing these algorithms will assist in the overall understanding of the N-body problem, as well as the algorithms themselves, and the reasons why one is more efficient as compared with the other. This will help in the field of computational astrophysics by focusing research on the faster algorithm, as those working on the N-body problem can direct time and resources into the optimum solution, rather than focusing on an inefficient algorithm, showing that it can be used in future related problems associated with the simulation of astronomical bodies and their interactions.

Abbreviations, Symbols and Notations

2D	Two-dimensional
OS	Operating System
CLI	Command Line Interface
GUI	Graphical User Interface
CPU	Central Processing Unit
GPU	Graphics Processing Unit
GPGPU	General-Purpose Computing on Graphics Processing Units
GRAPE	Gravity Pipe Circuits

Chapter 1. Introduction

1.1 Our Universe

The universe is ever-growing and expanding. For every second you experience, the universe has increased exponentially in size. As humanity's desire to understand how the universe works increases, so too does the complexity of the problems involved in broadening that understanding. The universe is expanding at a rate of 67.36 kilometres per second, per megaparsec. A megaparsec, or mega parallax second, equates to 3.26 million light-years (Greshko, 2021).
With such a rapid rate of expansion, the closest neighbours to Earth are travelling farther and further away with each passing day. The fastest spacecraft ever developed by humanity; the NASA-Germany Helios probes possessed a top speed of 250,000 kilometres per second. To reach Earth's nearest star sans the Sun, Alpha Centauri, which is 4.4 light-years away would take 18,000 years travelling at this speed (Clery, 2016).

Figure 1 - Alpha Centauri System (Skatebiker, 2012)

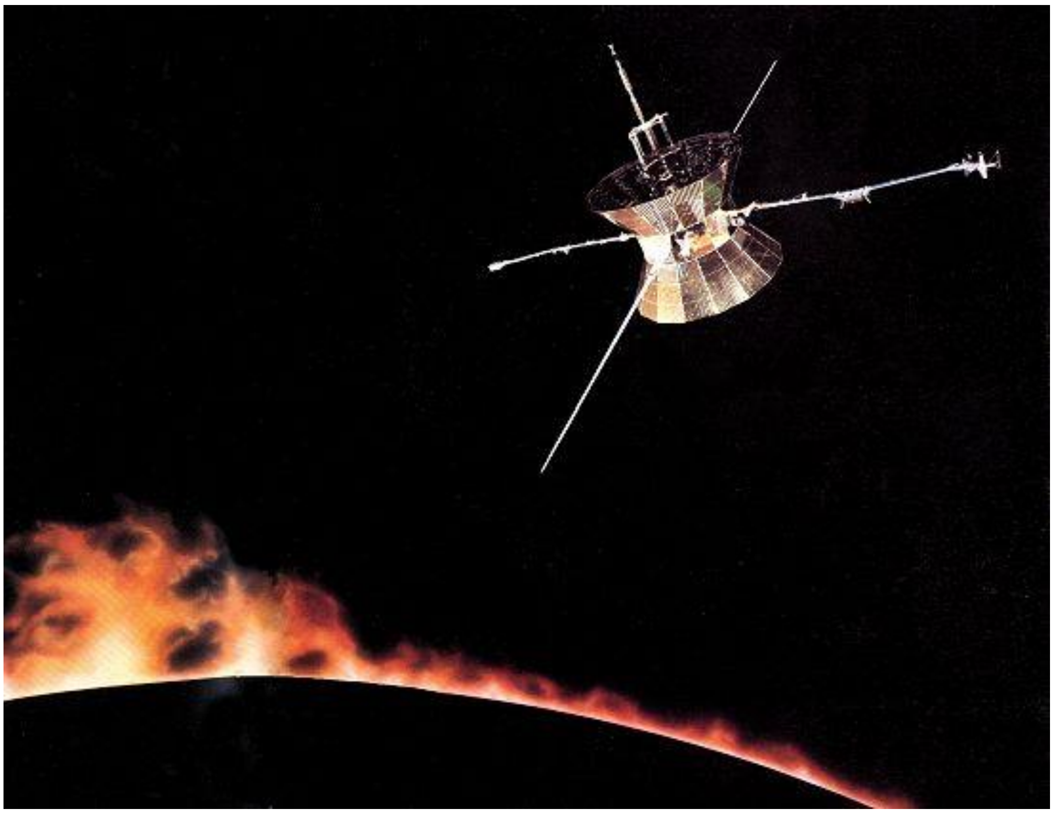


Figure 2 - HELIOS space probe (DLR Space Operations and Astronaut Training, n.d.)

This leads to a situation whereby studying and analysing the closest objects to us is near impossible with current technology. And these “close” objects are moving further away. A technique of simulating far away distant astronomical systems is needed if our understanding of the universe is to be increased, at least until interstellar travel is fully developed and understood in the coming centuries and millenniums.

1.2 Simulate the behaviour of astronomical systems

One such method is the construction of N-body simulations, which are extremely useful because they allow for the study of astronomical systems posessing complex properties and variables, all from the comfort of planet Earth. N-body simulations assist in modelling astronomical systems which themselves are referred to as a “dynamical system”, a system whereby the state of which is unique and specified by a set of variables. The behaviour of such systems adheres to defined rules (Sayama, 2020).

Dynamical systems describe how astronomical bodies interact. Given this is how these bodies behave in nature, and travel in the region of light-years currently is not yet possible, simulations are the best tool available to help broaden one’s understanding. N-body simulations are themselves an umbrella term, several methods exist, and different techniques serve different purposes regarding how the simulation is expected to behave as well as the expected results. They help visualise how these grand systems of objects interact and allow the visualisation of systems that would otherwise be unviewable and incomprehensible.

1.3 Aim of the project

This project proposes an investigation into N-body simulations using the Python virtual environment. This will be achieved through the development of an application comprised of multithreaded and non-multithreaded versions of two N-body simulation algorithms - Pairwise Interaction and a Tree Code implementation via the Barnes-Hut algorithm. Each simulation will be thoroughly analysed to determine which version is faster concerning different situations, such as the number of bodies contained within the system being altered. The application will be named **“*BinaryBody*”,** which refers to the word “***binary***” relating to elements composed of two things, in this case, the comparison of two N-body simulations, as well as the word “***body***”, which is a reference to “N-body simulations”. While this will not solve the N-body problem in general, this project aims to implement and analyse a system that performs the simulation quickly and efficiently from a software engineer’s perspective. The intention of it being used in further N-body research as a component to ***help*** in solving the problem, not to solve the entire problem itself due to the complexities of the research area.
Chapter 2 will consist of a Literature Review, critically analysing the sources that have contributed to the overall understanding of the subject area and subsequent project development, regarding the areas of the N-body problem itself, two distinct methods of N-body simulation; Pairwise Interaction/All-Pairs N-body and the Barnes-Hut Tree, fast N-body

simulations developed on the NVIDIA CUDA platform and experiences with parallel N-body simulation.

Chapter 3 is focused on the overall methodology followed and the approach taken concerning the implementation of the project.

In ***Chapter 4***, the simulation comparison results, and analysis shall be displayed, with further discussion in ***Chapter 5***.

Chapter 6 shall provide the conclusions made, in addition to the future work intended to further develop the project.

Chapter 2. Literature Review

2.1 Chaos Theory and the N-body Problem

Chaotic systems are governed by the principle of “chaos theory”, whereby dynamical systems that consist of states of instability possess underlying patterns and laws that govern them. It refers to the study of differential equations that are sensitive to and dependent on the initial conditions of a system (Stanford Encyclopedia of Philosophy, 2015).

The “*N-body problem”* also referred to as the *“3-body problem”* and *“many-body problem”,* describes the predicament of predicting the movements of an astronomical system with three or more individual bodies which are bound gravitationally. The more bodies are introduced, the more difficult it becomes to predict and calculate an orbit path. Apply this to an entire galactic system, and one can see the complexities involved in such a task and the challenges computational astrophysicists face in modelling astronomical systems. Moreover, this adds to the problem of understanding the operation of our universe. This predicament is what N-body simulations seek to solve.

2.2 The GRAPE Project

In the early 1990s, Junichiro Makino, a professor at the University of Tokyo developed a small inexpensive device known as a GRAPE, or “Gravity Pipe Circuits” (Makino, et al., 1996).

The GRAPE device would be attached to a computer as a co-processor and would handle the “pairwise force calculations”, in other words, the calculations of the forces that objects exert on one another. These calculations were of $O(n^2)$ complexity. The host computer would then complete the “star-pushing” section of the algorithm. GRAPE in the modern area has since fallen into disuse, only to be surpassed by none other than Graphical Processing Units (GPUs). GPUs are extremely helpful in astronomical object simulation and have evolved far past their traditional application in video game rendering. This in essence has given rise to General-Purpose Computing on Graphics Processing Units (GPGPU) and goes far beyond the constraints of image rendering alone, as GPGPU can utilise the GPU hardware to divert power towards scientific computing tasks, such as the simulation that this project proposes. GPGPU processing strengths lie in how they can be used to perform complex calculations and handle parallel processes (Perkins, no date).

GPUs operate by projecting a 3D scene onto a background image comprised of millions of pixels, and then refreshing that image extremely quickly before moving on to the next one, a sort of extremely advanced video. They work by rendering such pictures extremely quickly, displaying the illusion of motion. As such, while GPUs are not specifically designed for N-body simulations, they possess technological advantages given that they can act as the backbone within powerful computers running such complex and massive galactic simulations, in addition to having economic advantages. The GRAPE system is not entirely obsolete, however, given that it is a specialized device for N-body simulations. In 2009, Evghenii Gaburov, Stefan Harfst, and Simon Portegies Zwart managed to bring the GRAPE and GPU together by writing emulation software allowing for programs written on the GRAPE system to be executed on a GPU. This proved to be a huge technological breakthrough, as more recent simulations of galaxies, particularly that of the Milky Way can make the best use of supercomputers and GPUs without the concern of wasting processing power (Hayes, 2015).

Older simulations compiled and created via GRAPE could now be run on GPU hardware.

2.3 N-body simulations

An N-body simulation is a method of modelling a system of objects and numerically approximating how that system evolution. These simulations model the dynamical nature of astronomical systems, a dynamical system being where the system’s state evolves and is dependent on a set of variables and rules (DQ, no date).

For example, if you change the speed of an object, let's say a planet in a localized system, then the system responds to that change and can become unstable.
Furthermore, N-body simulations assist in visualising the interactions of objects under the influence of a force, primarily gravity. This could be the movement of objects such as planets, stars, and galaxies. It is a numerical solution to the mathematical equations that describe the movement of these objects interacting under that force (Hut & Trenti, 2008).
N-body simulations operate on timesteps, whereby every time it calculates the net force on each object in the system, the values are applied to then calculate the acceleration of each object and thus update the new position of the object. The frequency of timesteps can be altered by the user, for example, an N-body simulation could be set up whereby the simulation runs for 10 seconds, and each object position, the timestep, is updated every 0.01 seconds. A common application of N-body simulations is that of modelling how astronomical bodies like stars and galaxies interact under a force such as gravity, and how a system of bodies gravitationally bound to one another evolves. As such, these simulations are extremely helpful in allowing us to understand the dynamics of small-scale systems, such as Pluto and Charon, to large-scale groups of galaxies and superclusters, such as the Laniakea Supercluster. These in turn allow us to understand the structure of the universe on a much grander scale, and ultimately visualize how objects in the universe interact. Furthermore, we can use such simulations to gain a greater understanding of how the universe is expanding. By creating an N-body simulation in a virtualization environment, the proposed project will investigate two methods of N-body simulation construction and compare the performance of one against the other via run-time performance and additional metrics, while adhering to professional software engineering concepts and design. Comparing these techniques will assist in identifying the optimum solution. As discussed in the Abstract, many N-body simulation techniques exist, however, the focus of this project will be on the comparison of the pairwise-force interaction and

tree code techniques. Isaac Newton's Laws of Gravitation state that one can calculate the motion of a system of two bodies that exert a gravitational force on one another, provided the mass, velocity, and position of the bodies are known. From this, the orbit paths of astronomical bodies can be calculated. His equations described and provided evidence that two bodies in proximity can be gravitationally bound, and from which, orbit paths of said bodies can be drawn. The problem arises when trying to calculate the orbit paths and motions of three or more bodies that are gravitationally bound, i.e.: when n (where n = number of bodies) > 3.
In this situation, the orbit paths can become highly erratic due to "chaotic dynamics" (Stephens & Montgomery, 2019).
This topic was discussed in ***2.1 Chaos Theory and the N-body Problem***.

2.4 Pairwise Interaction/All-Pairs N-body

The simplest N-body simulation, All-Pairs N-body – also called a Naïve algorithm, brute force algorithm or pairwise interaction – works by individually calculating the force between each pair of bodies and adding up the resulting forces on each body. In the context of astronomical simulations, these bodies could be planets, moons, stars, asteroids etc.
To successfully execute an All-Pairs N-body simulation, first, compute the forces on each element, and then move the body based on its force, and repeat this step any given number of times. This action is performed simultaneously for every other body in the given system; hence it is considered a parallel computation (Udacity, 2015).
For N objects in a given system, each object computes the forces on it from every other object. This gives a computational complexity for this algorithm of $O(n^2)$, meaning the time taken to complete the algorithm grows based on the square of the input data set. This can be expressed as *"the number of bodies, multiplied by the number of bodies"* or *"the number of bodies squared"*. This simulation technique is extremely inefficient for larger systems. Suppose a system of 1000 bodies were to be constructed, with this complexity, 1000^2 operations would need to take

place, which subsequently gives significant amounts of operations; equal to 1,000,000. If a system of 10,000 bodies were to be constructed, then 100,000,000 operations would be required. The speed of the simulation drastically slows down the larger the system is, and the number of operations required to run each simulation, grows exponentially. A more efficient technique is required here.

2.5 Quadtrees and The Barnes-Hut Algorithm

The most important aspect of speeding up an N-body simulation is that of collectively treating multiple distant bodies, as one body. This is because those distant bodies' gravitational effects on other bodies in the system will be insignificant compared with those which are closer together. Distant bodies can be grouped and treated as one object by calculating the approximate gravitational effects on the rest of the system via the group's center of mass, which will be “the average position of each body in that group, weighted by mass”. For example, if there are two bodies which are sufficiently far away from the rest of the system; which have the positions; x1, x2 and mass; m1 and m2, then the center of mass of those bodies can be calculated via:

$$x = (m_1 * x_1) + (m_2 * x_2) / m_1 + m_2$$

Where:
x = centre-of-mass
m1 = mass of body 1
m2 = mass of body 2
x1 = position of body 1
x2 = position of body 2

The basis of the Barnes-Hut Algorithm is to approximate the gravitational effects of distant bodies by grouping them, and then using the centre-of-mass while calculating the exact gravitational forces of nearby forces on every other nearby body (Cole, 2020)

The algorithm functions by implementing recursion to divide the area in which the bodies reside, based on their position, and store them inside a "quadtree" data structure. A quadtree which implements the Barnes-Hut Algorithm is referred to as a "Barnes-Hut Tree". A quad-tree is a data structure which allows points in 2D space and their data to be stored. In a quad-tree, each node can only have four children at maximum. To engineer a BH Tree, first, divide the 2D space into four quadrants. If any of the quadrants contain more than one body, subdivide that quadrant into four more quadrants. Keep subdividing until each of the quadrants, including those that have been subdivided, contains only one body. Nodes cannot have more than four children, in the context of the 2D space populated with bodies, this means that the root node is the entire 2D space, where the four child nodes are the four-quadrant divisions. Each node is a reference to a position in space, and the recursive subdivision occurs until each subdivision has a maximum of one body only (Oyediran, 2017).

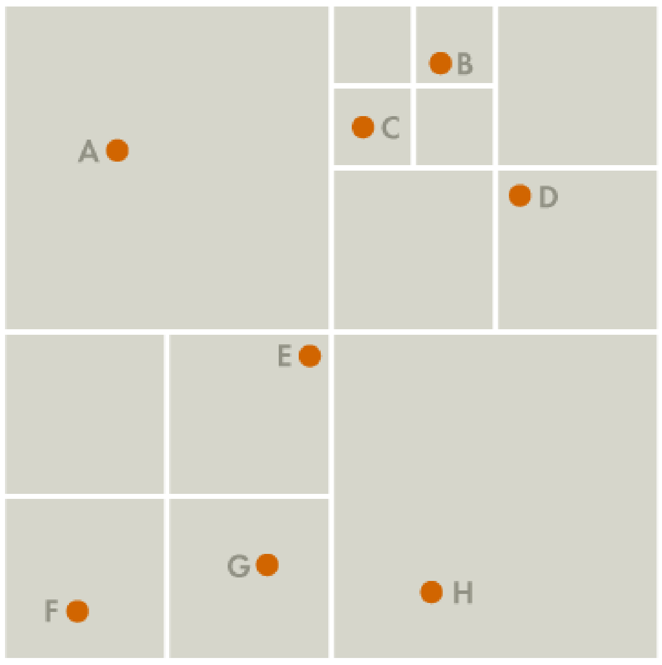


Figure 3 - Barnes-Hut 2D Plane Representation (Ventimiglia & Wayne, n.d.)

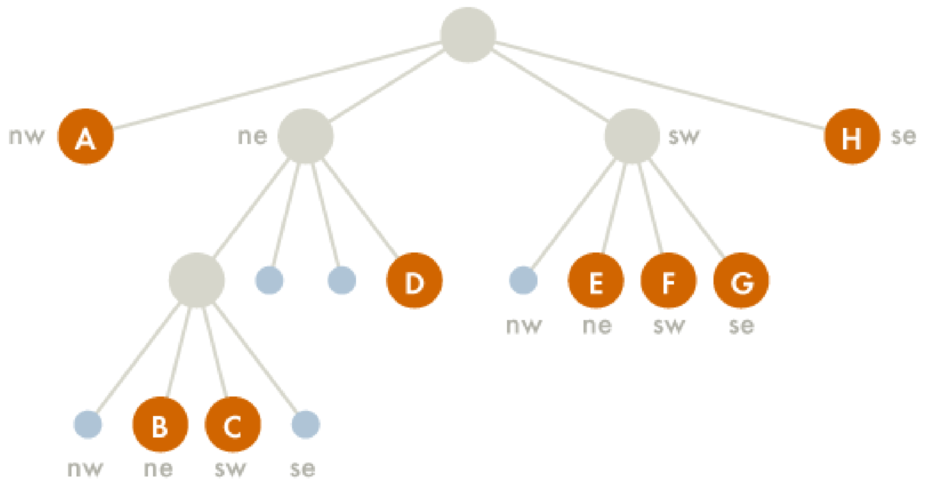


Figure 4 - Barnes-Hut Quadtree Representation (Ventimiglia & Wayne, n.d.)

BH Trees possess a computational complexity of O(n log n) as compared with All-Pairs N-body, which is O(n^2). To put this into perspective, if an N-body simulation of 1000 particles was generated, a BH Tree, being of O(n log n) computational complexity, would require 3000 checks to be performed (1000*log(1000) = 3000), whereas, with an All-Pairs N-body implementation, the number of checks to be performed would be one million (1000^2 = 1,000,000). It is extremely easy to see why quadtrees are used to perform simulations in terms of achieving raw speed, however, a potential disadvantage with this approach is that for larger, more dense systems, poor performance could potentially be observed due to the likelihood of hundreds, if not thousands of recursive subdivisions resulting in deep level nodes and quadrants within the quadtree (Trost, 2016).
BH Trees are great for speed, not so much when grander systems are in question. However, this technique is a massive step forwards compared with All-Pairs N-body, due to fewer operations being required. Therefore, BH Trees possess greater efficiency and achieve superior results in terms of algorithmic speed, with the same number of bodies.

2.6 Fast N-body simulation with NVIDIA CUDA

One approach to N-body simulation is that of the Pairwise method, discussed in ***2.4 Pairwise Interaction/All-Pairs N-body***. In this paper, the focus is on how the All-Pairs algorithm can be used as a kernel and how it can be implemented in the NVIDIA CUDA programming model. Furthermore, it demonstrates how parallelism available within the All-Pairs algorithmic kernel can be expressed in NVIDIA CUDA, to choose certain parameters to make full use of the NVIDIA GeForce 8800 GTX GPU. It must be noted that this paper was published in 2006 and as such the GPU in question is not up to date. When the N-body simulation was executed on both the GPU and CPU, the determination of results showed that the speed of the algorithm possessed a notable increase as compared with the CPU. The primary reason for this conclusion is that Intel CPUs possess the need for many unpipelined clock cycles when performing the square root and division operations within the N-body simulation (Nyland, et al., no date).

Figure 5 - NVIDIA GeForce 8800 GTX GPU (TechPowerUp, n.d.)

Pipelining refers to parallelism with a single CPU implemented at the instruction level. If a clock cycle is unpipelined, operations and processes such as fetching, decoding, and executing instructions, as well as writing to RAM are merged into a single step, rather than multiple parallel steps. Moreover, only one instruction can be executed per clock cycle (GeeksforGeeks, 2021).

Since modern CPUs are extremely fast, the illusion occurs that unpipelined processors can execute multiple instructions simultaneously, when this is not the case. When running the N-body simulation, if the

CPU requires some unpipelined instructions compared with the GPU, then that could create a notable performance bottleneck and thus explain why the GPU is faster. The overall conclusion here is that there are three main takeaways with regards to the All-Pairs algorithmic kernel’s efficiency after being executed on the GPU. The first is that it implements parallel, sequential memory access patterns, meaning that memory addresses are accessed in sequence, one after the other, and parallelisation to sequential memory addressing in the algorithm has been applied to further increase efficiency. The second is the reuse of data that keeps the ALUs occupied on the GPU, and the third is fully pipelined arithmetic (the process of breaking down the mathematical problems into smaller, more easily calculable problems which can be executed in parallel). This includes utilizing inverse square root operations as a parameter due to those calculation’s execution speed being faster on a GPU as opposed to a CPU, which were current at the time. The resulting implementations allowed for an algorithm that could run 50x quicker than a non-parallel, serial implementation. And up to 250x quicker than the C implementation the author wrote. Given the recorded and observed performance increase, more computationally efficient and complex simulations could be implemented, providing a more detailed and comprehensive N-body simulation of not simply gravitational systems, but also electrostatic or other pairwise-force systems (Yokota & Barber, 2022).

Chapter 3. Methodology

This technical investigation into N-body simulations seeks to determine the optimum algorithm for simulating astronomical systems. In ***Chapter 2 – Literature Review***, it was described in the various sources listed that the Barnes-Hut Algorithm is the more efficient method in practice, as it possesses an algorithmic complexity of O(n log n), compared with O(n^2) for Pairwise Interaction. This methodology seeks to prove the above statement. To achieve this, threaded and unthreaded versions of the Barnes-Hut and Pairwise Interaction Algorithm were compared to identify the most efficient technique.

3.1 Development Methodology

Due to the complexity of this technical investigation, the Agile Incremental Development Methodology, and its subset, the Kanban framework was chosen. This approach allows for multiple versions of the application to be engineered meaning each version is usable. Moreover, this ensures that the current version can be built upon as a superset of all prior versions. Kanban synergises well with Agile as it allows for a visual representation of the project's sections and subsections, as well as what has been completed, is currently being worked on, and what still needs to be engineered. A Gantt chart is used to describe an initial overview of tasks, as well as to provide a visual representation of the project structure (see Appendix A). Furthermore, the requirements are listed in the form of tickets (see Appendix B, C and D).

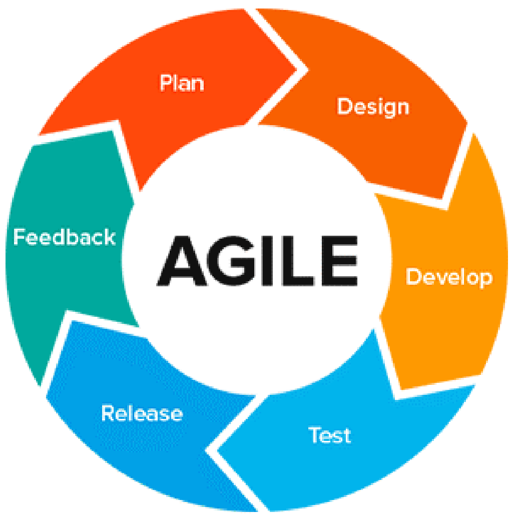


Figure 6 - Agile Development (Iddo, 2017)

3.1.1 Trello

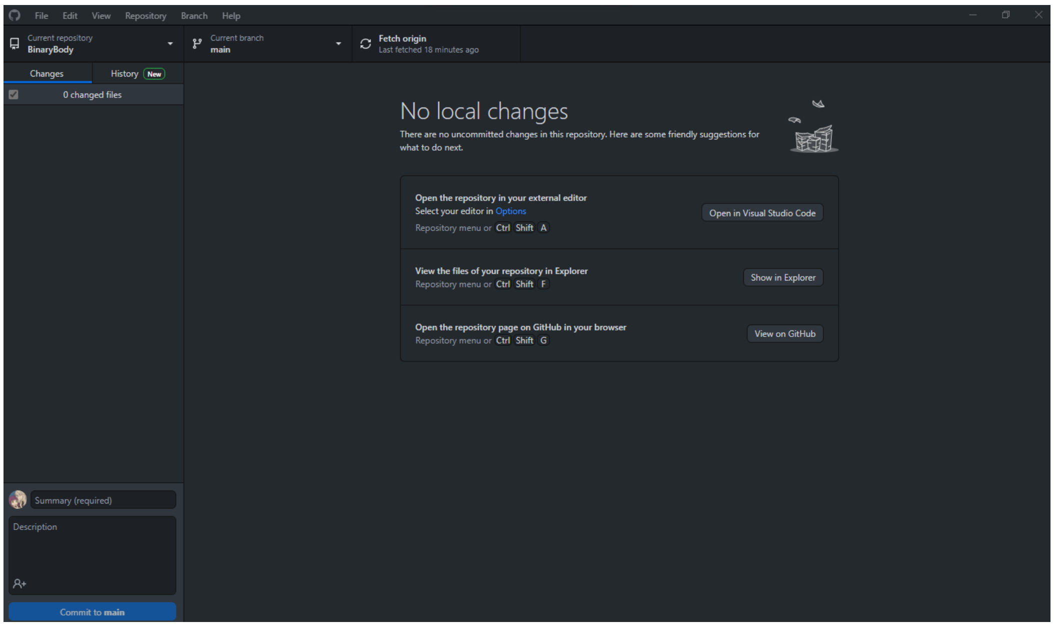
Trello was the service used to create the Kanban board for this project. It helped manage tasks by subdividing the requirements into tickets. As mentioned in ***Chapter 3.1***, Kanban synergises well with Agile due to the recording of tasks to be completed in a visual format. Trello allows for deadline dates for requirements to be completed, as well as the ability to add notes to individual tickets. Examples of the Kanban board for this project can be viewed in Appendix B, C and D.

3.2 Version Control

GitHub was the service used to provide version control within this project. It offers a web interface providing access control for different versions of a project. It is a distributed version control system that can track changes in source code during development. The use of version control within the project is extremely important should any reversions to previous code be necessary. GitHub fits seamlessly with Agile as changes can be made and pushed to the main repository with ease.

3.2.1 GitHub Desktop

To push any changes in the code to GitHub, the desktop client GitHub Desktop was used. GitHub Desktop is extremely powerful as it provides a GUI interface as opposed to the more traditional CLI Git Bash. It uses Git version control to push any changes to GitHub and assists in allowing bulk code changes to be pushed quickly, subsequently increasing the speed of development time.



No local changes

There are no uncommitted changes in this repository. Here are some friendly suggestions for what to do next.

Open the repository in your external editor
Select your editor in Options
Repository menu or Ctrl Shift A

View the files of your repository in Explorer
Repository menu or Ctrl Shift F

Open the repository page on GitHub in your browser
Repository menu or Ctrl Shift G

Figure 7 - GitHub Desktop

3.3 Technologies and Tools Deployed

3.3.1 Python 3.9

The Python programming language is a high-level, object-oriented programming language with a philosophy of code readability and a particular focus on data science (Python Software Foundation, no date). The benefit of using Python in for simulation construction is that due to the complexity of the simulations in question, it was imperative to write them in a language that is easy to learn, but more importantly, easy to read regarding its near-English syntax and code indentation. Since Python is an interpreted language, it makes interactive exploration of the code possible, providing feedback to the user immediately (frontiers in neuroscience, 2009).
Python possesses numerous libraries for use in data science and analysis, such as NumPy, SciPy, Matplotlib and Pandas, as well as modules for CPU profiling, such as cProfile. Additionally, all Python scripts will be written following the conventions set out by the PEP8 code

styling guidelines *(see Appendix E, F, G, H and I)* via the PEP8 code style validator (Bryukhanov, no date).
Moreover, Python allows for the ability to create a Virtual Environment via the `venv` module.

Figure 8 - Python Logo (Python Software Foundation, n.d.)

3.3.2 Python Virtual Environment

A Python Virtual Environment is a self-contained Python environment isolated from the main Python installation. The venv module supports the creation of lightweight virtual environments with their own libraries independent from the main Python installation.
The virtual environment contains its own Python binary that matches the version installed on the machine. When either the Barnes-Hut or Pairwise Interaction simulations are executed, the virtual environment is activated first, then the simulation is executed thereafter.

3.3.3 NumPy

NumPy, a portmanteau of “Numerical Python” is an open-source Python data science library commonly used in scientific computing. NumPy allows for complex mathematical equations and calculations to be used with Python and includes data structures such as multidimensional arrays and matrices (NumPy.org, no date).
NumPy is highly effective as it provides access to functions not available with the base Python installation. Regarding the Pairwise Interaction simulation, it is used to determine body positions and velocities, as well as the array containing those bodies. Concerning the Barnes-Hut simulation, NumPy performs linear algebra to calculate the distance between quadtree nodes. NumPy also performs other calculations within

the simulation scripts *(see **Chapter 3.8 Pairwise Interaction Implementation** and **Chapter 3.9 Barnes-Hut Simulation Implementation**)*.

Figure 9 - NumPy Logo (NumPy, 2020)

3.3.4 Windows Batch Files

Windows Batch files contain a list of commands processed in sequence without the need for user input. They are processed automatically upon execution. Batch files are used in this project to automatically navigate to the venv folder and activate, as well as to deactivate the virtual environment, rather than requiring the user to perform manual activation and deactivation.

3.3.5 timeit

Timeit is a Python module used to monitor the execution time of small sections of Python code (Python Software Foundation, no date). Within this project, it is used to monitor the execution time of the `get_acceleration()` function on the Pairwise Interaction simulation and the `barnes_hut_simulation_loop()` within the Barnes-Hut simulation.

3.3.6 cProfile

cProfile is a Python module which provides *"deterministic profiling"* of Python scripts. Deterministic profiling is the reflection that all function calls, function returns, and exception events are monitored. A profile refers to a set of statistics that describe how frequent and how long

various parts of the script take to finish executing. (Python Software Foundation, no date).
cProfile is used within this project to monitor the function calls in each of the simulation scripts in addition to generating an `output.pstats` file containing profile data for each of the two simulations, which can be converted into a visual format via gprof2dot and Graphviz. It is also used to generate an `output.prof` file from each simulation, which can be viewed online via Snakeviz.

3.3.7 gprof2dot

gprof2dot is a Python tool used to visualise profiler output in the form of a colour-coded call graph. Upon generation, the graph will be in DOT format, however, this can be converted to an image via Graphviz (Nanjappa, 2013).
In the context of this project, it is used in tandem with Graphviz to create an `output.png` file generated from the profile data contained within `output.pstats`.

3.3.8 Graphviz

Graphviz is an open-source graphing visualisation tool that can be used with Python. It has several main graph layout programs such as web and interactive graphical interfaces, and auxiliary tools, libraries, and language bindings. (Graphviz, no date).
Within this project, Graphviz is used in tandem with gprof2dot to create an `output.png` file from each of the two simulations, which visualises the `output.pstats` file as a visual colour-coded call tree.

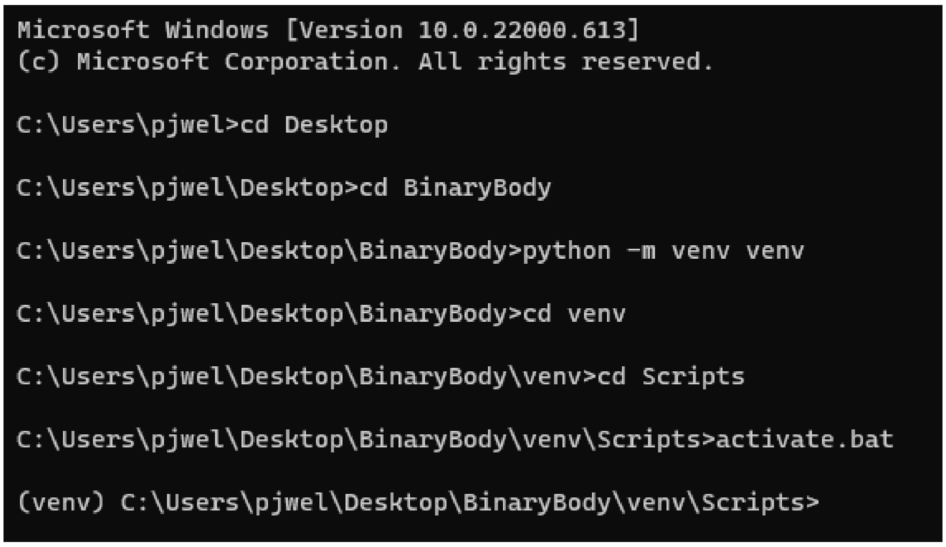
3.3.9 snakeviz

Snakeviz is a browser-based GUI viewer of profile data generated from cProfile (Snakeviz, no date).
The profile data from cProfile will be in the form of an `output.prof` file. This module can output and display profile data on the browser via the Tornado module, which snakeviz installs automatically as part of its

installation. Snakeviz is used here to view the `output.prof` file generated from both the Pairwise and Barnes-Hut simulations.

3.4 Virtual Environment Creation

Before any simulations are run, a virtual environment must first be created along with all necessary module installations. First, a folder named “BinaryBody” will be created, inside which the virtual environment, named `venv` will be created via the Windows Terminal. Once created, the virtual environment will be activated manually via the same terminal window. The brackets surrounding the word `venv` indicate that the virtual environment has been created successfully *(see Figure 10).*



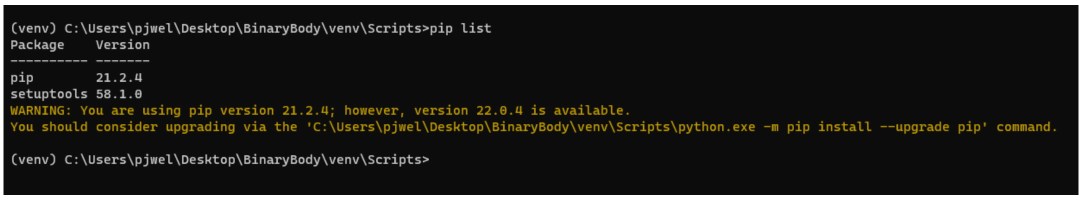
```
Microsoft Windows [Version 10.0.22000.613]
(c) Microsoft Corporation. All rights reserved.

C:\Users\pjwel>cd Desktop

C:\Users\pjwel\Desktop>cd BinaryBody

C:\Users\pjwel\Desktop\BinaryBody>python -m venv venv

C:\Users\pjwel\Desktop\BinaryBody>cd venv

C:\Users\pjwel\Desktop\BinaryBody\venv>cd Scripts

C:\Users\pjwel\Desktop\BinaryBody\venv\Scripts>activate.bat

(venv) C:\Users\pjwel\Desktop\BinaryBody\venv\Scripts>
```

Figure 10 - Create and Activate a Python Virtual Environment

3.4.1 Install modules to the virtual environment

After the creation and activation of the virtual environment, the necessary modules for the application must be installed. Entering `pip list` into the terminal should display the `pip` module itself as well as `setuptools` as no other libraries have been installed yet. NumPy, gprof2dot, psutil and snakeviz will be downloaded and installed via the command line. Graphviz will be downloaded and installed from https://graphviz.org/download/.

Note: Snakeviz will automatically download the Tornado module as part of its installation.

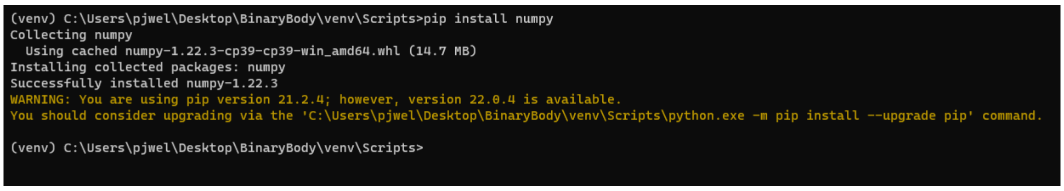


```
(venv) C:\Users\pjwel\Desktop\BinaryBody\venv\Scripts>pip list
Package    Version
---------- -------
pip        21.2.4
setuptools 58.1.0
WARNING: You are using pip version 21.2.4; however, version 22.0.4 is available.
You should consider upgrading via the 'C:\Users\pjwel\Desktop\BinaryBody\venv\Scripts\python.exe -m pip install --upgrade pip' command.

(venv) C:\Users\pjwel\Desktop\BinaryBody\venv\Scripts>
```

Figure 11 – pip list pre module installation

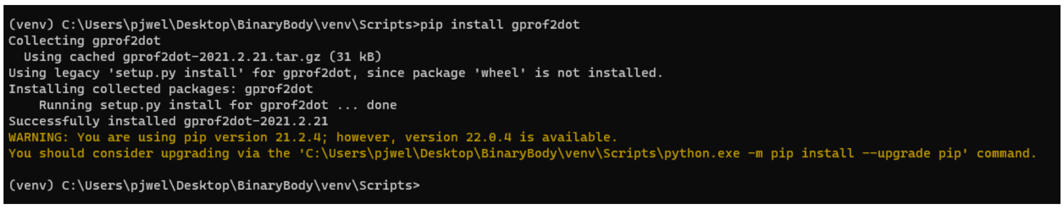
The following commands were entered into the terminal to install the aforementioned modules:

- `pip install numpy`
- `pip install gprof2dot`
- `pip install psutil`
- `pip install snakeviz`



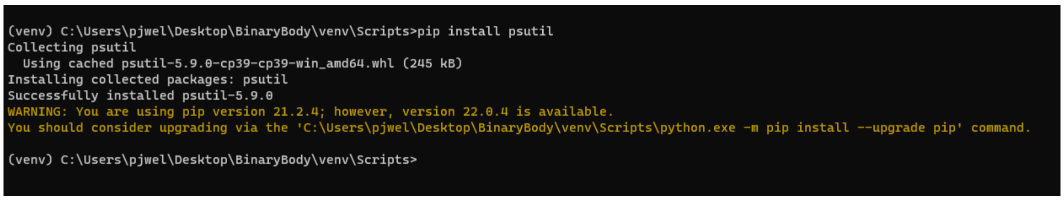
```
(venv) C:\Users\pjwel\Desktop\BinaryBody\venv\Scripts>pip install numpy
Collecting numpy
  Using cached numpy-1.22.3-cp39-cp39-win_amd64.whl (14.7 MB)
Installing collected packages: numpy
Successfully installed numpy-1.22.3
WARNING: You are using pip version 21.2.4; however, version 22.0.4 is available.
You should consider upgrading via the 'C:\Users\pjwel\Desktop\BinaryBody\venv\Scripts\python.exe -m pip install --upgrade pip' command.

(venv) C:\Users\pjwel\Desktop\BinaryBody\venv\Scripts>
```

Figure 12 - pip install numpy



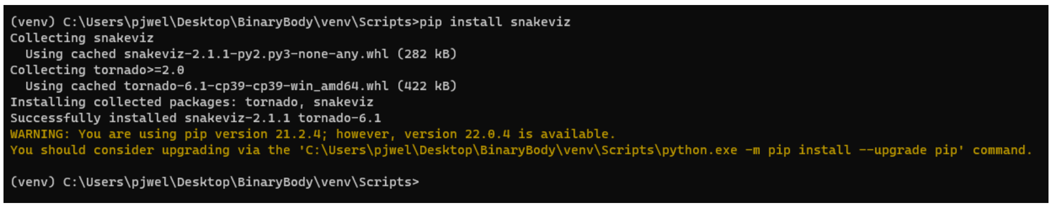
```
(venv) C:\Users\pjwel\Desktop\BinaryBody\venv\Scripts>pip install gprof2dot
Collecting gprof2dot
  Using cached gprof2dot-2021.2.21.tar.gz (31 kB)
Using legacy 'setup.py install' for gprof2dot, since package 'wheel' is not installed.
Installing collected packages: gprof2dot
    Running setup.py install for gprof2dot ... done
Successfully installed gprof2dot-2021.2.21
WARNING: You are using pip version 21.2.4; however, version 22.0.4 is available.
You should consider upgrading via the 'C:\Users\pjwel\Desktop\BinaryBody\venv\Scripts\python.exe -m pip install --upgrade pip' command.

(venv) C:\Users\pjwel\Desktop\BinaryBody\venv\Scripts>
```

Figure 13 - pip install gprof2dot



```
(venv) C:\Users\pjwel\Desktop\BinaryBody\venv\Scripts>pip install psutil
Collecting psutil
  Using cached psutil-5.9.0-cp39-cp39-win_amd64.whl (245 kB)
Installing collected packages: psutil
Successfully installed psutil-5.9.0
WARNING: You are using pip version 21.2.4; however, version 22.0.4 is available.
You should consider upgrading via the 'C:\Users\pjwel\Desktop\BinaryBody\venv\Scripts\python.exe -m pip install --upgrade pip' command.

(venv) C:\Users\pjwel\Desktop\BinaryBody\venv\Scripts>
```

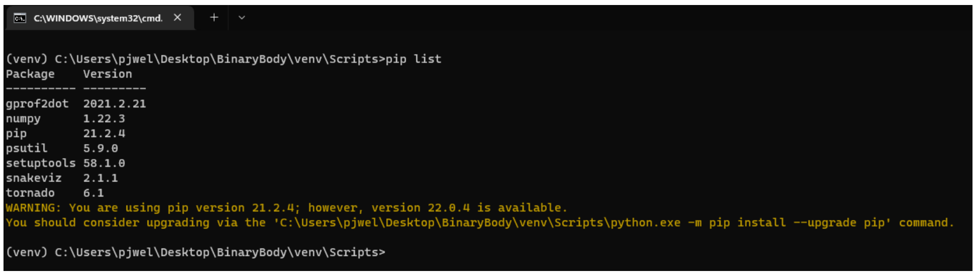
Figure 14 - pip install psutil



```
(venv) C:\Users\pjwel\Desktop\BinaryBody\venv\Scripts>pip install snakeviz
Collecting snakeviz
  Using cached snakeviz-2.1.1-py2.py3-none-any.whl (282 kB)
Collecting tornado>=2.0
  Using cached tornado-6.1-cp39-cp39-win_amd64.whl (422 kB)
Installing collected packages: tornado, snakeviz
Successfully installed snakeviz-2.1.1 tornado-6.1
WARNING: You are using pip version 21.2.4; however, version 22.0.4 is available.
You should consider upgrading via the 'C:\Users\pjwel\Desktop\BinaryBody\venv\Scripts\python.exe -m pip install --upgrade pip' command.

(venv) C:\Users\pjwel\Desktop\BinaryBody\venv\Scripts>
```

Figure 15 - pip install snakeviz

Upon installation of all the necessary libraries, `pip list` was run again to verify that the modules were installed correctly.

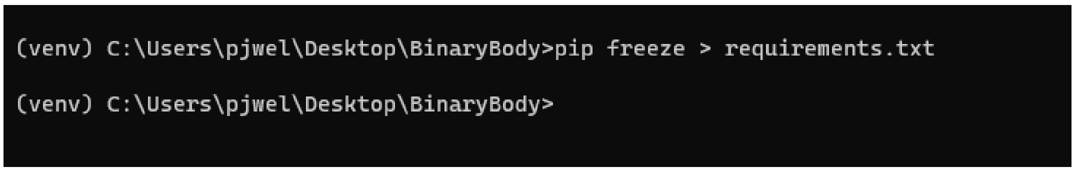


```
(venv) C:\Users\pjwel\Desktop\BinaryBody\venv\Scripts>pip list
Package    Version
---------- ---------
gprof2dot  2021.2.21
numpy      1.22.3
pip        21.2.4
psutil     5.9.0
setuptools 58.1.0
snakeviz   2.1.1
tornado    6.1
WARNING: You are using pip version 21.2.4; however, version 22.0.4 is available.
You should consider upgrading via the 'C:\Users\pjwel\Desktop\BinaryBody\venv\Scripts\python.exe -m pip install --upgrade pip' command.

(venv) C:\Users\pjwel\Desktop\BinaryBody\venv\Scripts>
```

Figure 16 - pip list post-module installation

Once the module installations to the virtual environment have been verified, a `pip freeze > requirements.txt` action will be performed to generate a requirements.txt file. This file stores all the library and module information relating to the virtual environment. This file is important because if the virtual environment must be discarded for any reason, then requirements.txt can be used to download all the libraries, dependencies and modules automatically via the `pip install -r requirements.txt` command. Alternatively, if this fails, the `pip3 install -r requirements.txt` command can also be used.

This is particularly useful for projects which contain tens or even hundreds of libraries.

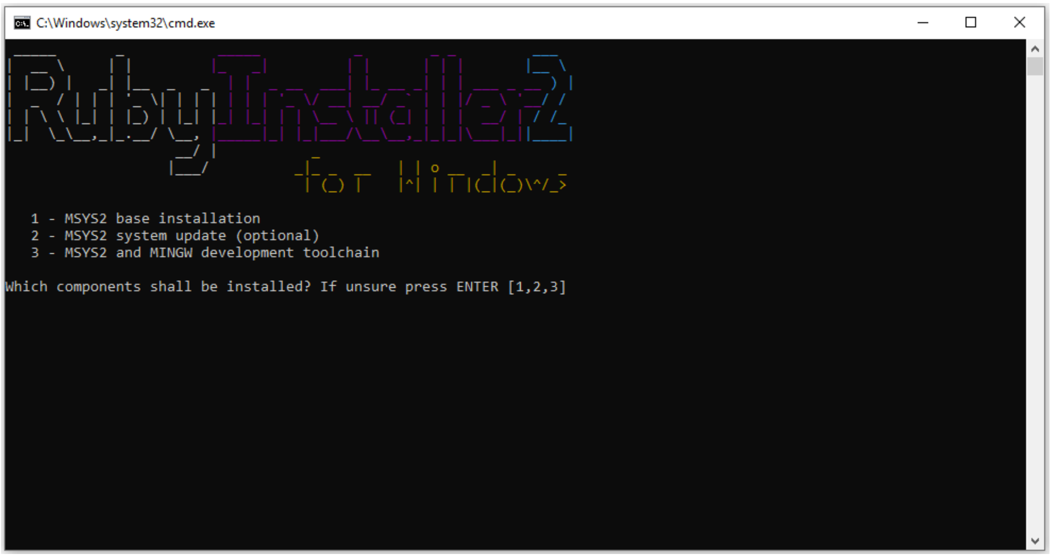


```
(venv) C:\Users\pjwel\Desktop\BinaryBody>pip freeze > requirements.txt

(venv) C:\Users\pjwel\Desktop\BinaryBody>
```

Figure 17 - pip freeze > requirements.txt

3.5 User Interface and UX

The user interface for the BinaryBody application is entirely CLI-based, which includes an ASCII art rendition of the BinaryBody logo. It was inspired by the Ruby programming language installer for Windows and can be found on all Python scripts within this project.



```
  1 - MSYS2 base installation
  2 - MSYS2 system update (optional)
  3 - MSYS2 and MINGW development toolchain

Which components shall be installed? If unsure press ENTER [1,2,3]
```

Figure 18 - Ruby Installer for Windows (Lib-Static Howtos, n.d.)

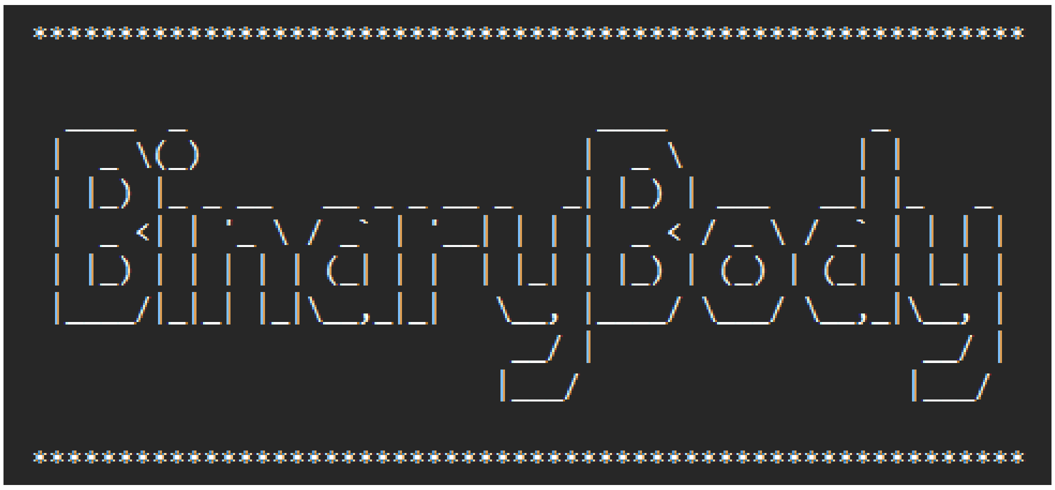
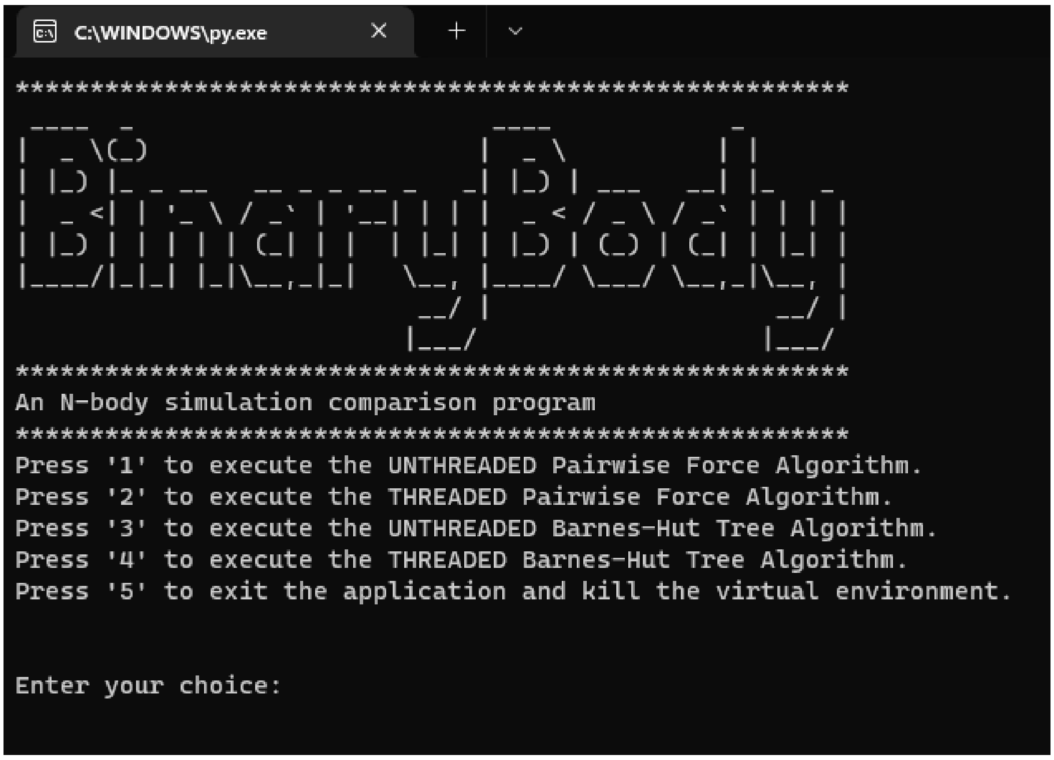


Figure 19 - BinaryBody ASCII Art Logo

3.6 main.py

This is the script that launches the Pairwise and Barnes-Hut simulations. Upon launching the script, the user will be greeted by a window asking them the following:

- Press “1” to execute the UNTHREADED Pairwise Force Algorithm
- Press “2” to execute the THREADED Pairwise Force Algorithm
- Press “3” to execute the UNTHREADED Barnes-Hut Tree Algorithm
- Press “4” to execute the THREADED Barnes-Hut Tree Algorithm
- Press “5” to exit the application and kill the virtual environment



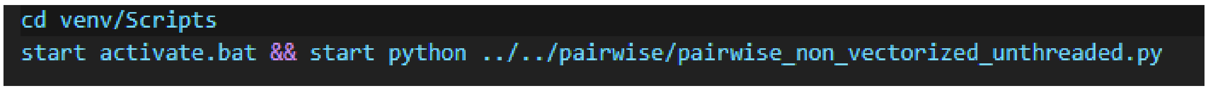
```
********************************************************
 ____  _                 ____            _
|  _ \(_)               |  _ \          | |
| |_) |_ _ __   __ _ _ __ _   _| |_) | ___   __| |_   _
|  _ <| | '_ \ / _` | '__| | | |  _ < / _ \ / _` | | | |
| |_) | | | | | (_| | |  | |_| | |_) | (_) | (_| | |_| |
|____/|_|_| |_|\__,_|_|   \__, |____/ \___/ \__,_|\__, |
                           __/ |                   __/ |
                          |___/                   |___/
********************************************************
An N-body simulation comparison program
********************************************************
Press '1' to execute the UNTHREADED Pairwise Force Algorithm.
Press '2' to execute the THREADED Pairwise Force Algorithm.
Press '3' to execute the UNTHREADED Barnes-Hut Tree Algorithm.
Press '4' to execute the THREADED Barnes-Hut Tree Algorithm.
Press '5' to exit the application and kill the virtual environment.


Enter your choice:
```

Figure 20 - BinaryBody Main

3.7 Batch files to activate the virtual environment

To execute each simulation successfully, the virtual environment containing the necessary libraries must be activated. Instead of manually activating the virtual environment every time, batch files were created, which will be launched from `main.py` and subsequently, the specific batch file will execute the appropriate simulation script based on which one was selected. For example, if the Threaded Barnes-Hut simulation was selected from `main.py`, the batch file `activate_venv_threaded_barnes_hut.bat` will be called, which activates the virtual environment and then that batch file will call `barnes_hut_quadtree_threaded.py`, thus executing the script.

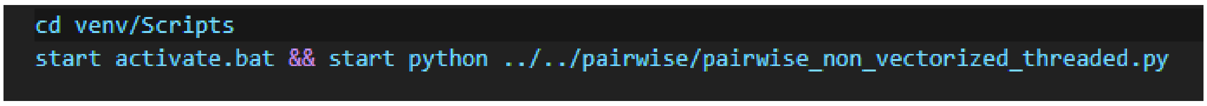
3.7.1 activate_venv_unthreaded_pairwise.bat



```
cd venv/Scripts
start activate.bat && start python ../../pairwise/pairwise_non_vectorized_unthreaded.py
```

Figure 21 - activate_venv_unthreaded_pairwise.bat

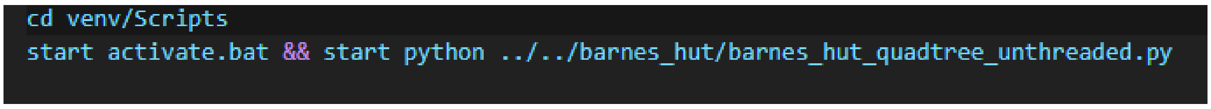
3.7.2 activate_venv_threaded_pairwise.bat



```
cd venv/Scripts
start activate.bat && start python ../../pairwise/pairwise_non_vectorized_threaded.py
```

Figure 22 - activate_venv_threaded_pairwise.bat

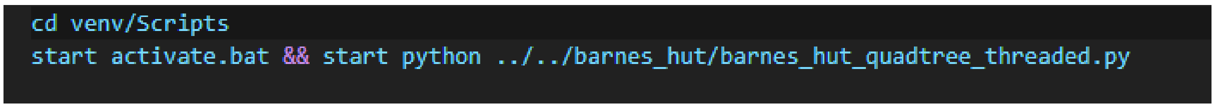
3.7.3 activate_venv_unthreaded_barnes_hut.bat



```
cd venv/Scripts
start activate.bat && start python ../../barnes_hut/barnes_hut_quadtree_unthreaded.py
```

Figure 23 - activate_venv_unthreaded_barnes_hut.bat

3.7.4 activate_venv_threaded_barnes_hut.bat



```
cd venv/Scripts
start activate.bat && start python ../../barnes_hut/barnes_hut_quadtree_threaded.py
```

Figure 24 - activate_ven_threaded_barnes_hut.bat

3.8 Pairwise Interaction Implementation

A dynamical system of bodies interacting gravitationally will be implemented here. The force calculations must be performed, followed by the leapfrog method integration and initial conditions configuration.

3.8.1 Force Calculations

According to Isaac Newton's Law of Universal gravitation, every object will experience a force on itself from all other bodies. "G" in the figure below represents Newton's Gravitational Constant, equal to 6.67×10^-11 $m^3/kg/s^2$. In short, G determines the strength of the inverse square law. (The Royal Society Publishing, 2014)

$$F = G\frac{m_1 m_2}{r^2}$$

Figure 25 - Equation for Newton's Law of Universal Gravitation / The Inverse Square-Law (BBC News, 2016)

F = Force

G = Newton's Gravitational Constant

m1 = Mass of first body

m2 = Mass of second body

r^2 = Radius, squared

Each body possesses a mass, position and velocity. Concerning the position, this will be an array containing the body's x, y, and z-coordinates. With the velocity, it will also be an array, except it will contain the velocity along the x-axis, y-axis and z-axis. The simulation will contain ***"N point bodies"*** where N will represent the number of bodies determined by the user. These bodies will be indexed by ***"i=1"*** up to ***"i=N"***.

3.8.2 Get Acceleration

The `get_acceleration()` function was implemented to calculate the acceleration of objects within the simulation per Newton's Law of Universal Gravitation. First, an array is created called N, which represents the number of objects in the simulation. Acceleration is an N x 3 matrix of positions. This is because the acceleration will be along the x, y and z-axes. The “position” parameter is also an N x 3 matrix of positions due to each position being comprised of an x, y and z-coordinate. Mass is an N x 1 one-dimensional array of masses. This is because each body in the simulation can only possess one mass value. Softening refers to the number added to the force calculation to avoid potential issues when two bodies are nearby. If omitted, the acceleration can theoretically scale up to infinity.

3.8.3 Leapfrog Method

The positions and velocities of the bodies in the simulation will be updated via the “leapfrog method”. This follows the sequence: “kick-drift-kick”. The simulation follows a timestep cycle. Timestep describes the change in the time between simulation cycles, also referred to as frames. For every timestep, denoted by Delta (Δ), all bodies will receive:

A half-step kick:

$$\mathbf{v}_i = \mathbf{v}_i + \frac{\Delta t}{2} \times \mathbf{a}_i$$

Figure 26 - Half-Timestep Kick Formula

V = velocity

Δt = timestep

a = acceleration

A full-step drift:

$$\mathbf{r}_i = \mathbf{r}_i + \Delta t \times \mathbf{v}_i$$

Figure 27 – Full-Timestep Drift Formula

R = radius

Δt = timestep

V = velocity

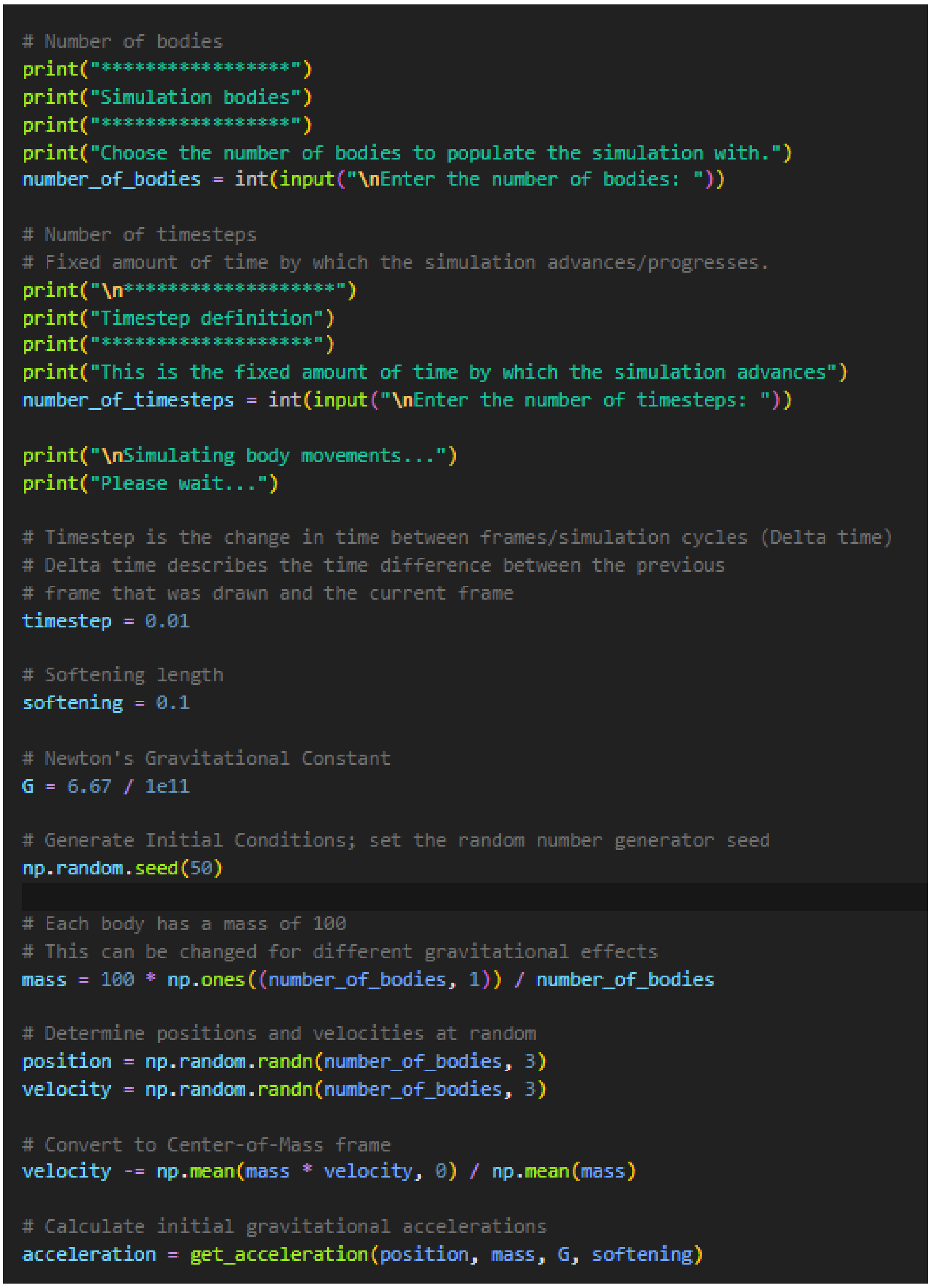
Followed by another half-step kick.

3.8.4 Configure Pairwise Simulation Initial Conditions

The initial conditions for the simulation consists of specifying the following: the number of bodies and number of timesteps via user entry, the timestep value; pre-set to 0.01, the softening length, pre-set to 0.1, Newton’s Gravitational Constant; pre-set to G = 6.67 / 1e11 and the random number seed. The mass of each body will then be set to 100 for all bodies before each body’s random position and velocity will be determined. Once this has been performed, the velocity will be converted to the centre-of-mass reference frame.

The centre-of-mass reference frame is the set of coordinates centred on and moving with the centre-of-mass of the system being studied. In the centre-of-mass frame, by definition, the centre of mass of the system is at rest, and the total momentum is zero (encyclopedia.com, no date).

After the conversion of the velocity to the centre-of-mass reference frame, invocation of the `get_acceleration()` function will occur.

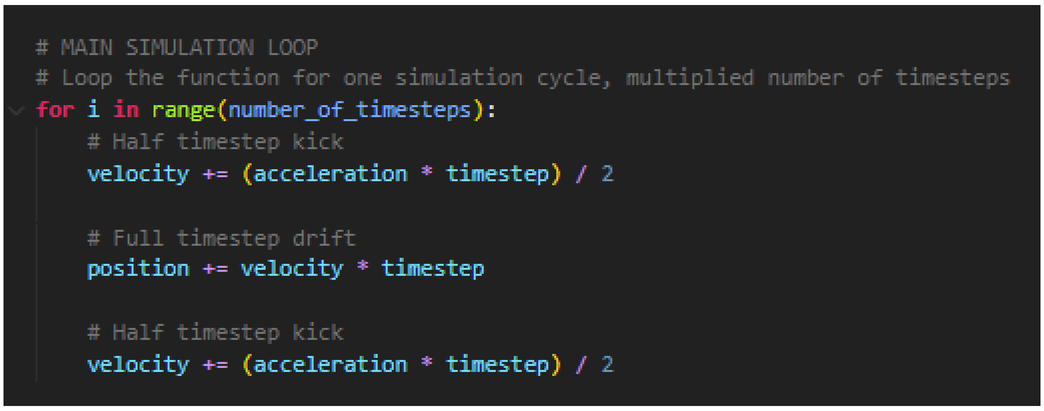


```python
# Number of bodies
print("*****************")
print("Simulation bodies")
print("*****************")
print("Choose the number of bodies to populate the simulation with.")
number_of_bodies = int(input("\nEnter the number of bodies: "))

# Number of timesteps
# Fixed amount of time by which the simulation advances/progresses.
print("\n*******************")
print("Timestep definition")
print("*******************")
print("This is the fixed amount of time by which the simulation advances")
number_of_timesteps = int(input("\nEnter the number of timesteps: "))

print("\nSimulating body movements...")
print("Please wait...")

# Timestep is the change in time between frames/simulation cycles (Delta time)
# Delta time describes the time difference between the previous
# frame that was drawn and the current frame
timestep = 0.01

# Softening length
softening = 0.1

# Newton's Gravitational Constant
G = 6.67 / 1e11

# Generate Initial Conditions; set the random number generator seed
np.random.seed(50)

# Each body has a mass of 100
# This can be changed for different gravitational effects
mass = 100 * np.ones((number_of_bodies, 1)) / number_of_bodies

# Determine positions and velocities at random
position = np.random.randn(number_of_bodies, 3)
velocity = np.random.randn(number_of_bodies, 3)

# Convert to Center-of-Mass frame
velocity -= np.mean(mass * velocity, 0) / np.mean(mass)

# Calculate initial gravitational accelerations
acceleration = get_acceleration(position, mass, G, softening)
```

Figure 28 - Pairwise Simulation Initial Conditions

3.8.5 Unthreaded Pairwise Interaction

Unthreaded Pairwise Interaction operates by looping the simulation for a number of iterations equal to the number of timesteps determined by the user, effectively controlling how long the simulation takes to conclude its execution. The kick-drift-kick leap-frog calculations are defined inside this for-loop, as it is the basis for determining the behaviour of all the bodies within the simulation.

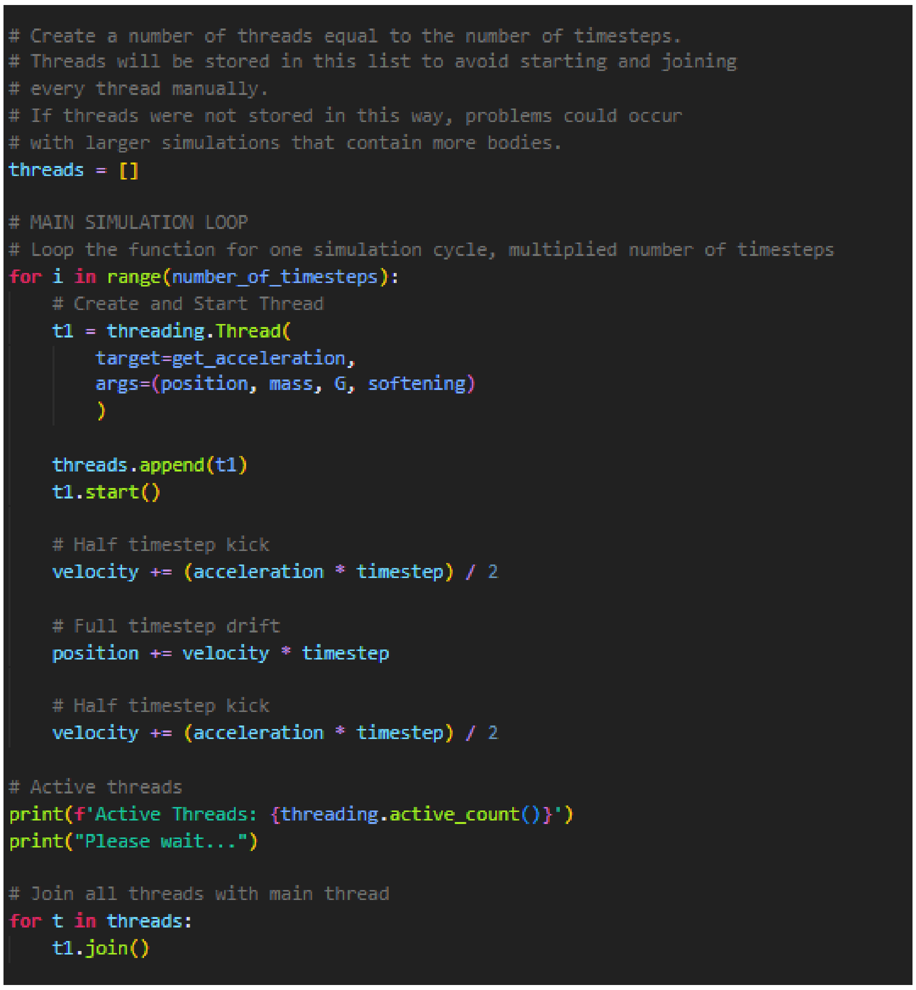


```
# MAIN SIMULATION LOOP
# Loop the function for one simulation cycle, multiplied number of timesteps
for i in range(number_of_timesteps):
    # Half timestep kick
    velocity += (acceleration * timestep) / 2

    # Full timestep drift
    position += velocity * timestep

    # Half timestep kick
    velocity += (acceleration * timestep) / 2
```

Figure 29 - Unthreaded Pairwise Simulation Loop

3.8.6 Threaded Pairwise Interaction

Multithreading was implemented into the Pairwise Interaction simulation via instantiation of a list object called `threads` to contain all threads, whereby a thread object called `t1` is then defined and configured inside the main simulation loop to target the `get_acceleration` function, appending the `thread` object to the `threads` list and starting that thread, and then performing the leapfrog calculation. See ***Chapter 3.8.1 Force Calculations***. This will occur for a number of iterations defined by the user. A further for-loop is used to wait until all the threads have finished executing before joining back to the main thread. This is done to avoid starting and joining every single thread manually.



```python
# Create a number of threads equal to the number of timesteps.
# Threads will be stored in this list to avoid starting and joining
# every thread manually.
# If threads were not stored in this way, problems could occur
# with larger simulations that contain more bodies.
threads = []

# MAIN SIMULATION LOOP
# Loop the function for one simulation cycle, multiplied number of timesteps
for i in range(number_of_timesteps):
    # Create and Start Thread
    t1 = threading.Thread(
        target=get_acceleration,
        args=(position, mass, G, softening)
        )

    threads.append(t1)
    t1.start()

    # Half timestep kick
    velocity += (acceleration * timestep) / 2

    # Full timestep drift
    position += velocity * timestep

    # Half timestep kick
    velocity += (acceleration * timestep) / 2

# Active threads
print(f'Active Threads: {threading.active_count()}')
print("Please wait...")

# Join all threads with main thread
for t in threads:
    t1.join()
```

Figure 30 - Threaded Pairwise Simulation Loop

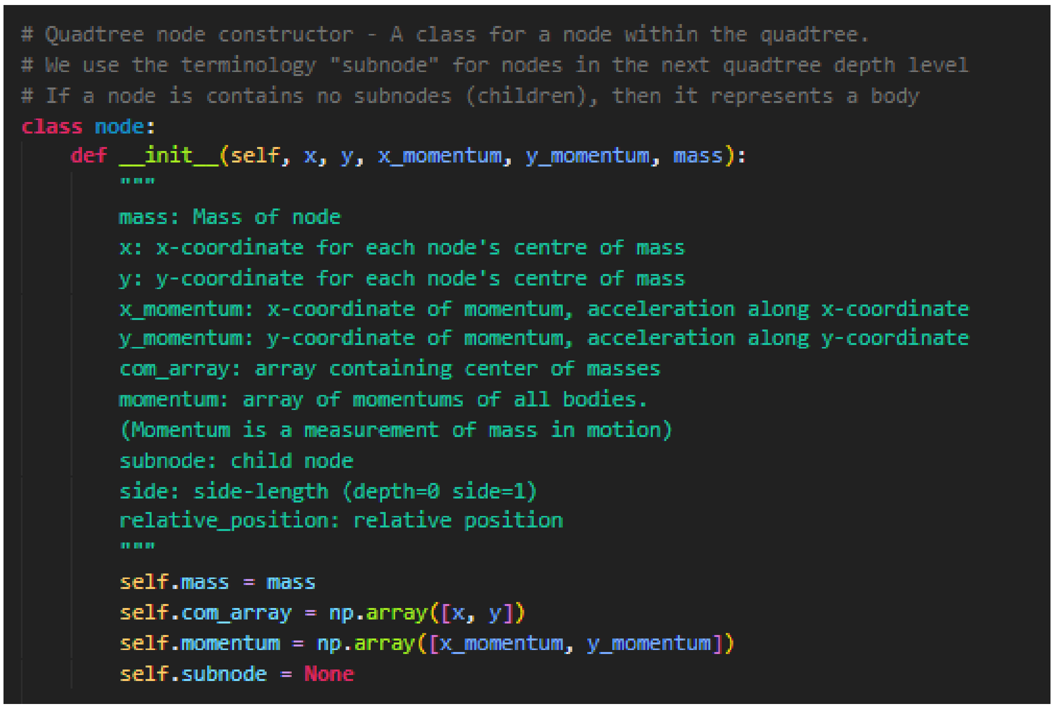
3.9 Barnes-Hut Simulation Implementation

The Barnes-Hut Algorithm operates by grouping nearby bodies together and approximating that group as a single body if it is sufficiently far away, determined by a parameter named Theta, discussed in ***Chapter 3.9.6 Configure Barnes-Hut Simulation Initial Conditions***. Recursive division occurs, whereby the bodies contained within the simulation space are split into groups and stored within a quadtree data structure. Quadtrees are similar to binary search trees, except each node consists of four children as opposed to two. It is important to note that some nodes may be empty within a quadtree, and each node can only have a maximum of four children. The root node references the entire simulation area, and the four quadrants are the child nodes of the root node

To create the Barnes-Hut Simulation, a quadtree must first be engineered and for each body within the simulation area, traverse the quadtree to a sufficient depth level. This will provide the centre-of-mass values required to calculate distances and thus will determine each body's acceleration. Numerically integrate using an algorithm such as Verlet, Euler etc.

3.9.1 Quadtree Node Constructor

This subsection describes the implementation of the quadtree node constructor class and its associated functions. The initialisation (`init`) function handles the initialisation of class attributes, as well as instantiation of the mass, centre-of-mass array (`com_array`), momentum and subnode objects.

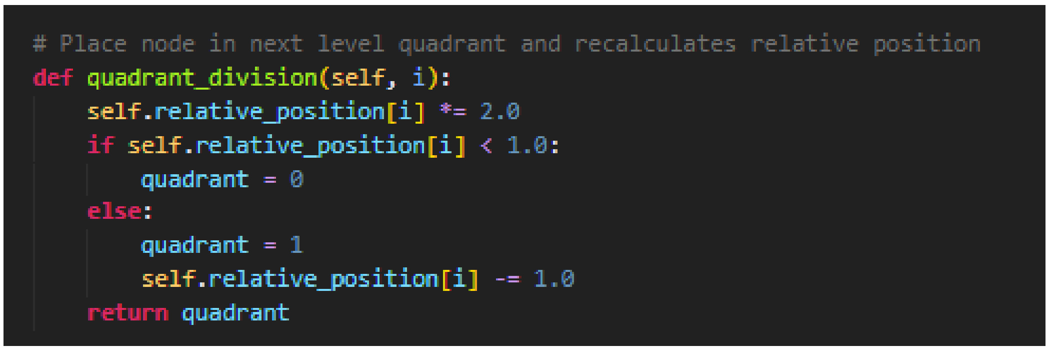


```python
# Quadtree node constructor - A class for a node within the quadtree.
# We use the terminology "subnode" for nodes in the next quadtree depth level
# If a node is contains no subnodes (children), then it represents a body
class node:
    def __init__(self, x, y, x_momentum, y_momentum, mass):
        """
        mass: Mass of node
        x: x-coordinate for each node's centre of mass
        y: y-coordinate for each node's centre of mass
        x_momentum: x-coordinate of momentum, acceleration along x-coordinate
        y_momentum: y-coordinate of momentum, acceleration along y-coordinate
        com_array: array containing center of masses
        momentum: array of momentums of all bodies.
        (Momentum is a measurement of mass in motion)
        subnode: child node
        side: side-length (depth=0 side=1)
        relative_position: relative position
        """
        self.mass = mass
        self.com_array = np.array([x, y])
        self.momentum = np.array([x_momentum, y_momentum])
        self.subnode = None
```

Figure 31 - Barnes-Hut Quadtree Node Constructor

3.9.1.1 Quadrant Division

The Barnes-Hut Algorithm uses recursion to divide the simulation space down until each subdivision contains zero bodies or one body. Refer to ***Chapter 2.5 Quadtrees and the Barnes-Hut Algorithm*** for more information.

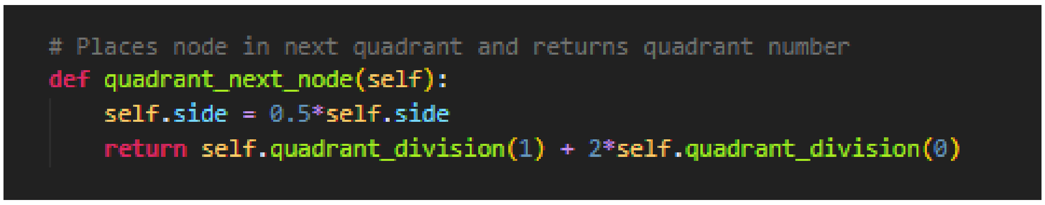


```python
# Place node in next level quadrant and recalculates relative position
def quadrant_division(self, i):
    self.relative_position[i] *= 2.0
    if self.relative_position[i] < 1.0:
        quadrant = 0
    else:
        quadrant = 1
        self.relative_position[i] -= 1.0
    return quadrant
```

Figure 32 - Barnes-Hut Quadtree Quadrant Division

3.9.1.2 Quadrant Next Node

This function places the node in the next quadrant and calculates the relative position within the quadtree.

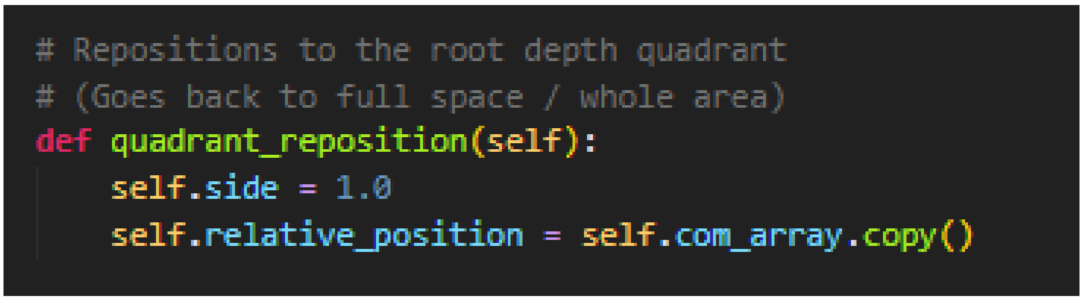


```python
# Places node in next quadrant and returns quadrant number
def quadrant_next_node(self):
    self.side = 0.5*self.side
    return self.quadrant_division(1) + 2*self.quadrant_division(0)
```

Figure 33 - Barnes-Hut Quadtree Quadrant Next Node

3.9.1.3 Quadrant Reposition

This function traverses back to the root node of the quadtree.

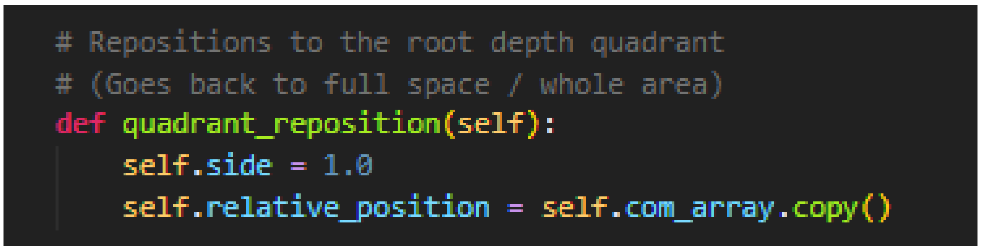


```python
# Repositions to the root depth quadrant
# (Goes back to full space / whole area)
def quadrant_reposition(self):
    self.side = 1.0
    self.relative_position = self.com_array.copy()
```

Figure 34 - Barnes-Hut Quadtree Quadrant Reposition

3.9.1.4 Node Distance

Function allowing for the calculation of distance between quadtree nodes.

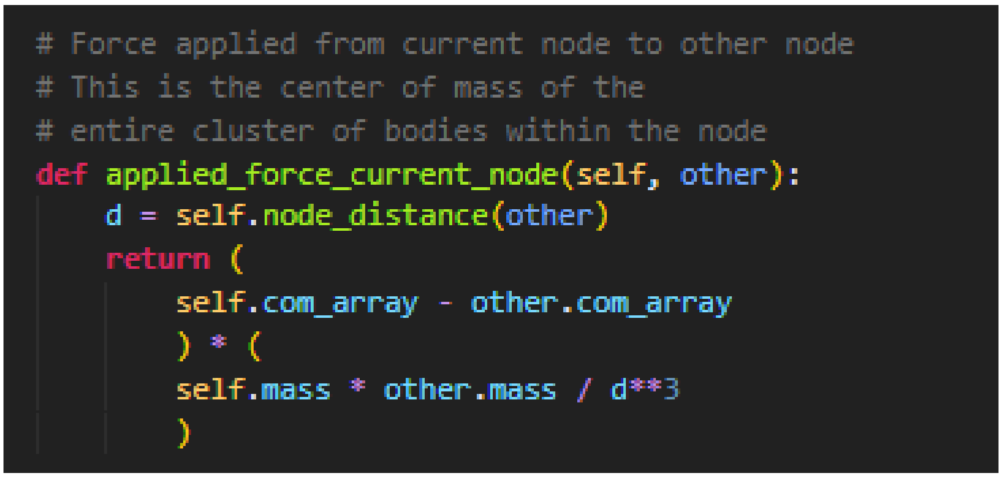


```python
# Repositions to the root depth quadrant
# (Goes back to full space / whole area)
def quadrant_reposition(self):
    self.side = 1.0
    self.relative_position = self.com_array.copy()
```

Figure 35 - Barnes-Hut Quadtree Quadrant Node Distance

3.9.1.5 Applied Force on the Current Node

This function calculates the force exerted from the current node on another node.

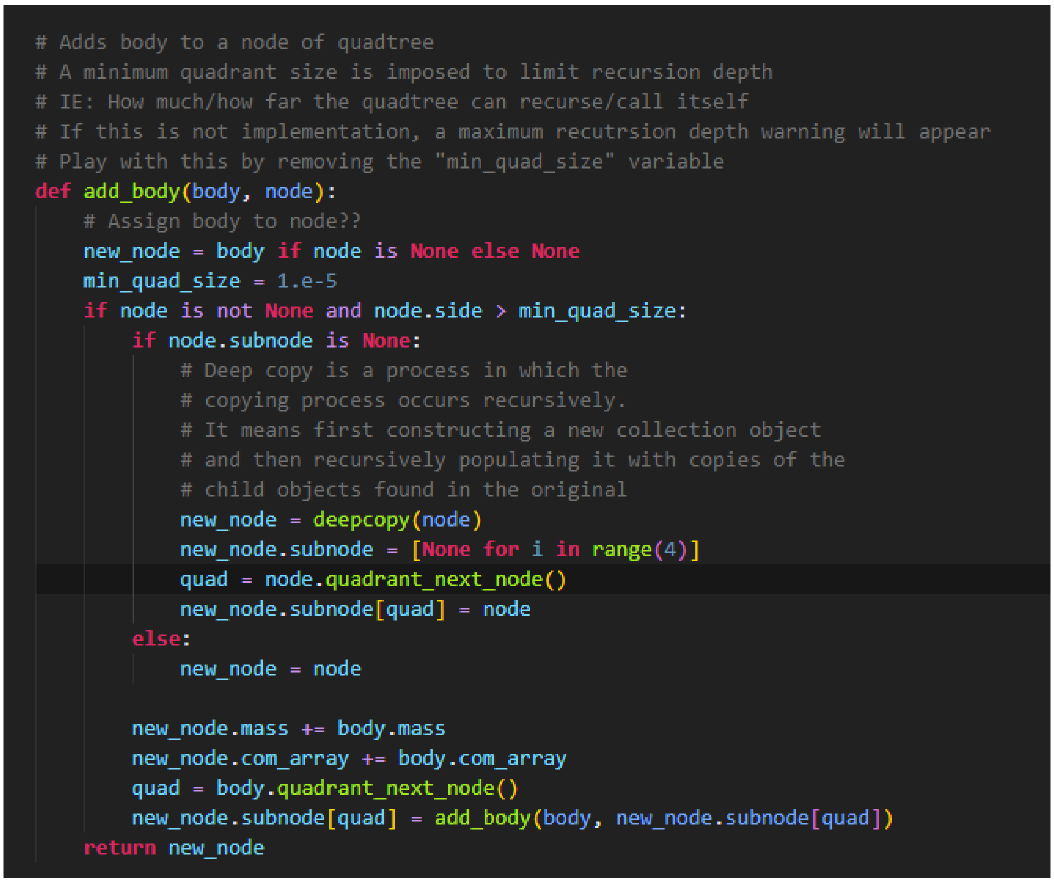


```python
# Force applied from current node to other node
# This is the center of mass of the
# entire cluster of bodies within the node
def applied_force_current_node(self, other):
    d = self.node_distance(other)
    return (
        self.com_array - other.com_array
        ) * (
        self.mass * other.mass / d**3
        )
```

Figure 36 - Barnes-Hut Quadtree Applied Force on the Current Node

3.9.2 Add Body

This function controls how bodies are added to a specific node within the quadtree. When adding bodies, it will create a new node which will be equal to the new body if that node does not already contain one. A minimum recursion depth has been implemented to limit how far the quadtree can recurse. If this was not implemented, the quadtree could recurse into infinity.



```python
# Adds body to a node of quadtree
# A minimum quadrant size is imposed to limit recursion depth
# IE: How much/how far the quadtree can recurse/call itself
# If this is not implementation, a maximum recutrsion depth warning will appear
# Play with this by removing the "min_quad_size" variable
def add_body(body, node):
    # Assign body to node??
    new_node = body if node is None else None
    min_quad_size = 1.e-5
    if node is not None and node.side > min_quad_size:
        if node.subnode is None:
            # Deep copy is a process in which the
            # copying process occurs recursively.
            # It means first constructing a new collection object
            # and then recursively populating it with copies of the
            # child objects found in the original
            new_node = deepcopy(node)
            new_node.subnode = [None for i in range(4)]
            quad = node.quadrant_next_node()
            new_node.subnode[quad] = node
        else:
            new_node = node

        new_node.mass += body.mass
        new_node.com_array += body.com_array
        quad = body.quadrant_next_node()
        new_node.subnode[quad] = add_body(body, new_node.subnode[quad])
    return new_node
```

Figure 37 – Barnes-Hut Quadtree Add Body Function

3.9.3 Force Acting On a Body

Calculate the net force acting on a body from a given quadrant by applying Newton's force calculation, using the centre of mass. For example, to calculate the net force acting on body x, the following recursive function can be used:

- If the current node is an external node and not x itself, calculate the force exerted by the current node on x, and add this amount to x's net force.
- Otherwise, calculate the ratio s/d. If s/d is less than the Theta parameter, treat this internal node as a sole body and calculate the force it exerts on x; add this value to x's net force
- Alternatively, execute this recursive function on the node's children.

3.9.4 Verlet Integration

Verlet integration is a numerical method for the integration of Newton's equations describing the laws of motion. It is typically implemented in the trajectory calculation of particles or bodies in simulations, such as those used in orbital mechanics. An example of Verlet integration in real life was that of the research conducted by Philip Herbert Cowell and Andrew Claude de la Cherois Crommelin in 1909 to calculate the orbit path of Halley's Comet (Nina.az, 2021).

3.9.5 Single Timestep Cycle

This function simulates one timestep cycle, taking the number of bodies, the Theta value, Newton's Gravitational constant and timestep as parameters.

3.9.6 Configure Barnes-Hut Simulation Initial Conditions

Upon quadtree implementation, the simulation can now be constructed. For every quadrant within the quadtree, the centre-of-mass can be calculated. Then, given a body, the force acting on it from a given quadrant can be calculated by applying Newton's force calculation using the centre of mass from that quadrant. For example, this approach could be applied to the four quadrants present at the first-depth level to approximate the force acting on the body. This is a potentially inaccurate calculation if there are many bodies present. To overcome this predicament, the Barnes Hut algorithm specifies a critical distance parameter called "Theta" (θ). If the critical distance between the body and the centre of mass of the quadrant is greater than Theta, then it is used as an approximation, if not the algorithm moves to the next depth level of the quadtree and tries again by way of recursion. The recursion takes place until the distance becomes below Theta, or there is only one body in the quadrant. Moreover, setting Theta = 0 returns to the $O(N^2)$ pairwise interaction approach (Cole, 2020).
In summary, it is extremely important to set this parameter to a minimum critical distance greater than 0 so the Barnes-Hut algorithm can approximate the distance of distant body clusters via their centre of mass.

The initial conditions of the Barnes-Hut simulation will follow a constant in line with the conditions configured in the Pairwise Simulation, in addition to the Theta parameter. The number of bodies and number of timesteps can be defined by the user. The value for G will be set to 6.67 / 1e11 and the timestep will be set to 0.01. Theta will be set to 0.5.

3.9.7 Unthreaded Barnes-Hut

This simulation operates by looping for a number of iterations equal to the number of timesteps defined via user entry. The main Barnes-Hut simulation loop is defined by the function `barnes_hut_simulation_loop`. Contained within this function is a for-loop which calls the function responsible for controlling a single simulation cycle; `single_timestep_cycle`, which iterates for a number of cycles equal to the number of timesteps. The single timestep function contains the Theta parameter, number of bodies, the timestep value itself and the value corresponding to Newton's Gravitational Constant, represented by "G".

3.9.8 Threaded Barnes-Hut

Regarding multithreading, inside the function `barnes_hut_simulation_loop`, much like with the threaded Pairwise Interaction simulation, a list called `threads` was created to contain all threads. A thread object called `t1` is then defined and configured to target the `single_timestep_cycle` function. The `t1` thread object is then appended to the threads list before the thread itself starts. This will occur for a number of iterations defined by the user.
Much the same as with the multithreaded Pairwise Interaction simulation, an additional for-loop is used to wait until all the threads have finished executing before fusing back with the main thread.

Chapter 4. Results

Covered in this section is the overall effectiveness of the Threaded and Unthreaded variants of the Pairwise Interaction and Barnes-Hut Simulations. During the testing phase of this application, three attempts were made to measure the execution time for a range of bodies (250, 500, 750, 1000) for each simulation. The mean average execution time was then calculated from these results for every simulation. To measure algorithmic efficiency, the mean average execution time for each number of bodies was used as the basis to calculate the percentage change in execution time. All results were recorded with the number of timesteps as a constant. 100 timesteps were inserted into each simulation for every test. Regarding the CPU Profiling testing phase, profile data was generated using the number of bodies and timesteps as constants. 100 of each were used during this phase of testing. For all simulations, cProfile was used to create a *pstats* file, gprof2dot was used to take that *pstats* file and convert it to a *png* image displaying a visible call tree, and cProfile was then used again to create a prof file, whereby this prof file can be viewed online via Snakeviz to visualise the script's call tree differently to that of the *png* image.

4.1 Execution Times

4.1.1 Unthreaded Pairwise Interaction

Number of Bodies	Attempt 1 Execution Time (s)	Attempt 2 Execution Time (s)	Attempt 3 Execution Time (s)	Mean Average Execution Time (s)
250	1.168	1.273	1.04	1.16
500	4.5	4.405	4.191	4.365
750	12.561	10.707	18.414	13.894
1000	17.894	18.323	17.534	17.917

Table 1 - Unthreaded Pairwise Interaction Execution Times

4.1.2 Unthreaded Barnes-Hut

Number of Bodies	Attempt 1 Execution Time (s)	Attempt 2 Execution Time (s)	Attempt 3 Execution Time (s)	Mean Average Execution Time (s)
250	2.08	2.08	2.313	2.158
500	4.527	4.717	4.317	4.52
750	7.106	7.981	6.953	7.347
1000	9.275	9.315	9.095	9.228

Table 2 - Unthreaded Barnes-Hut Execution Times

4.1.3 Threaded Pairwise Interaction Execution Times

Number of Bodies	Attempt 1 Execution Time (s)	Attempt 2 Execution Time (s)	Attempt 3 Execution Time (s)	Mean Average Execution Time (s)
250	1.181	1.267	1.025	1.158
500	3.997	4.307	4.928	4.411
750	8.827	10.591	9.511	9.643
1000	35.188	38.345	31.343	34.959

Table 3 - Threaded Pairwise Interaction Execution Times

4.1.4 Threaded Barnes-Hut

Number of Bodies	Attempt 1 Execution Time (s)	Attempt 2 Execution Time (s)	Attempt 3 Execution Time (s)	Mean Average Execution Time (s)
250	5.08	6.713	6.086	5.96
500	9.801	7.741	8.696	8.746
750	12.995	12.62	12.539	12.718
1000	14.475	13.688	13.401	13.855

Table 4 - Threaded Barnes-Hut Execution Times

4.2 Percentage Changes

4.2.1 Unthreaded Barnes-Hut vs. Unthreaded Pairwise Interaction

Number of Bodies	% Change in Mean Average Execution Time
250	86%
500	4%
750	89%
1000	94%

Table 5 - Percentage Change between UNTHREADED Barnes-Hut and UNTHREADED Pairwise Interaction Mean Average Execution Times

4.2.2 Threaded Barnes-Hut vs. Threaded Pairwise Interaction

Number of Bodies	% Change in Mean Average Execution Time
250	415%
500	98%
750	32%
1000	152%

Table 6 - Percentage Change between THREADED Barnes-Hut and THREADED Pairwise Interaction Mean Average Execution Times

4.2.3 Unthreaded vs. Threaded Pairwise Interaction

Number of Bodies	% Change in Mean Average Execution Time
250	0%
500	1%
750	31%
1000	49%

Table 7 - Percentage Change between UNTHREADED and THREADED Pairwise Interaction Mean Average Execution Times

4.2.4 Unthreaded vs. Threaded Barnes-Hut

Number of Bodies	% Change in Mean Average Execution Time
250	176%
500	93%
750	73%
1000	50%

Table 8 - Percentage Change between UNTHREADED and THREADED Barnes-Hut Mean Average Execution Times

4.3 Line Graphs for Mean Average Execution Times

4.3.1 Unthreaded Pairwise Interaction vs. Unthreaded Barnes-Hut

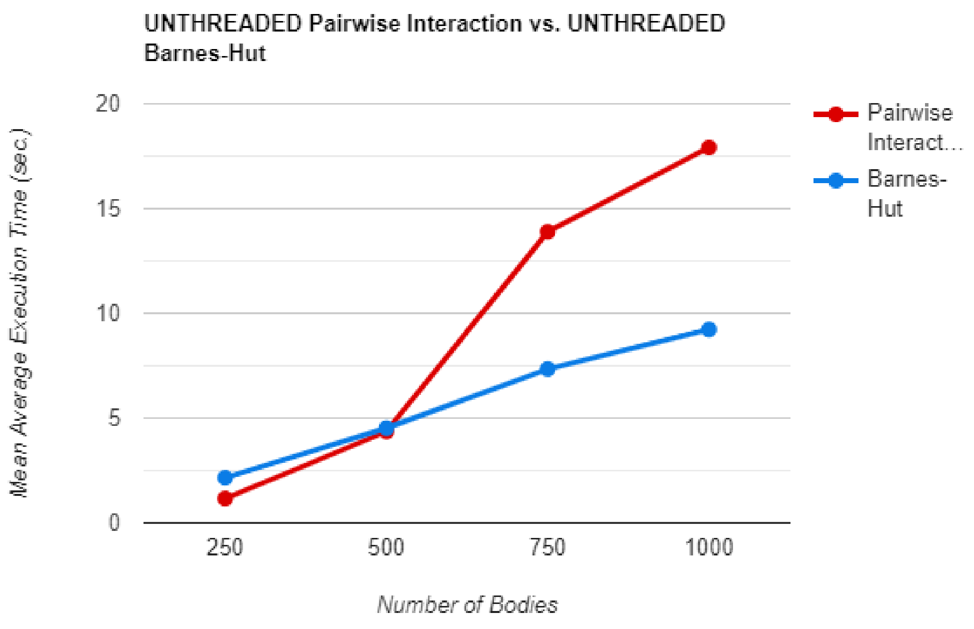


Figure 38 - UNTHREADED Pairwise Interaction vs. UNTHREADED Barnes-Hut

4.3.2 Threaded Pairwise Interaction vs. Threaded Barnes-Hut

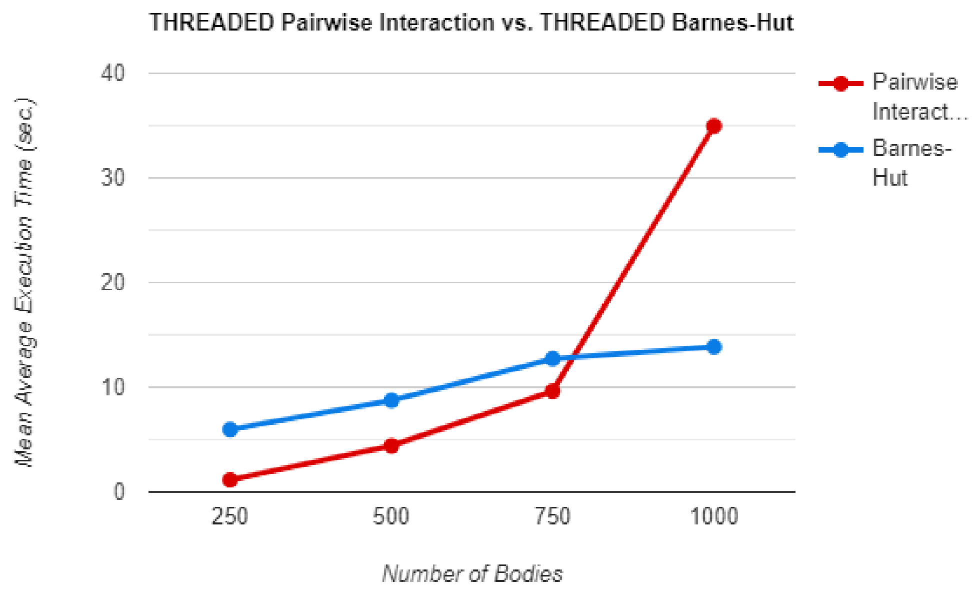


Figure 39 - THREADED Pairwise Interaction vs. THREADED Barnes-Hut

4.3.3 Unthreaded vs. Threaded Pairwise Interaction

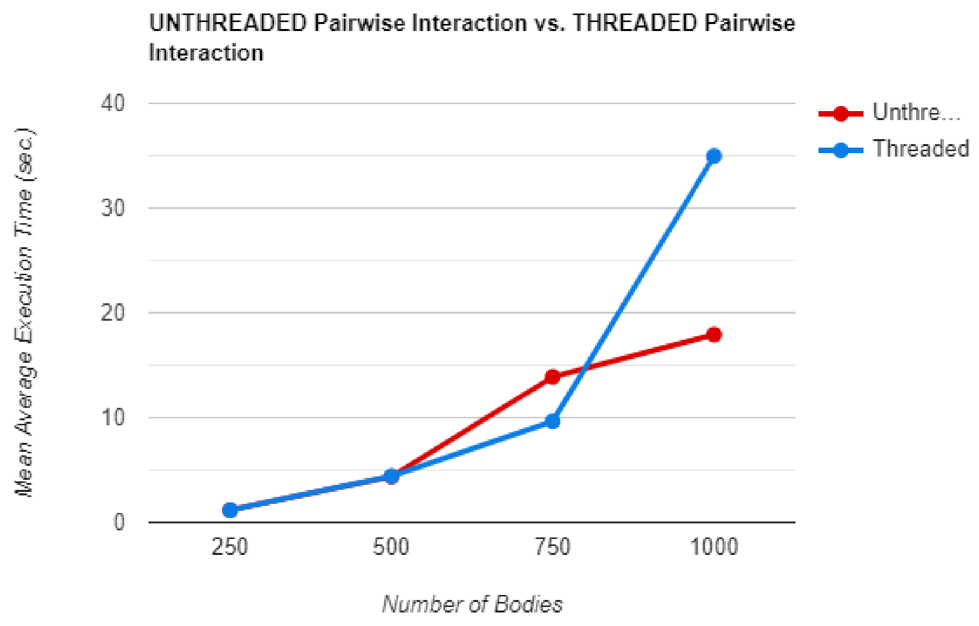


Figure 40 - UNTHREADED vs. THREADED Pairwise Interaction

4.3.4 Unthreaded vs. Threaded Barnes-Hut

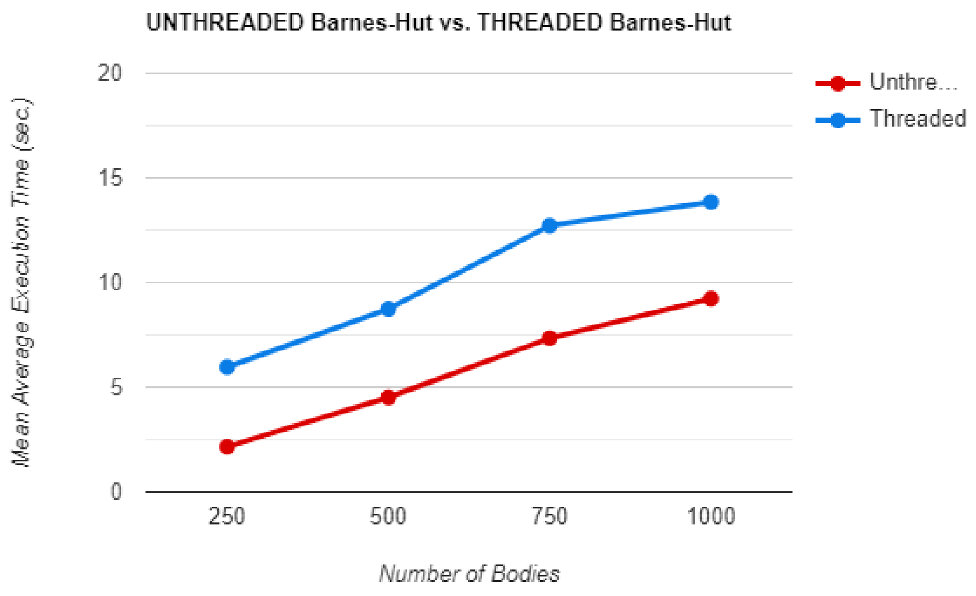


Figure 41 - UNTHREADED vs. THREADED Barnes-Hut

4.4 CPU Profiling

4.4.1 Unthreaded Pairwise Interaction

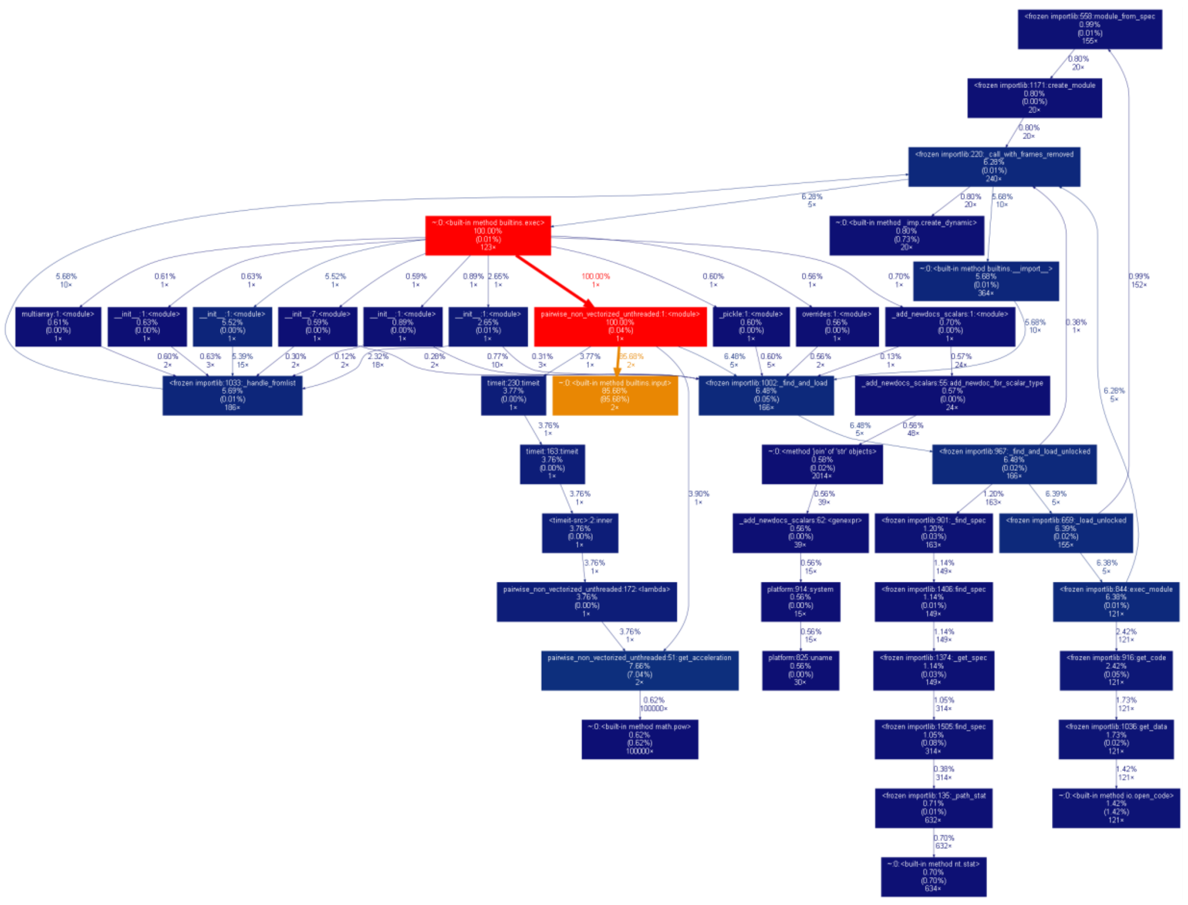
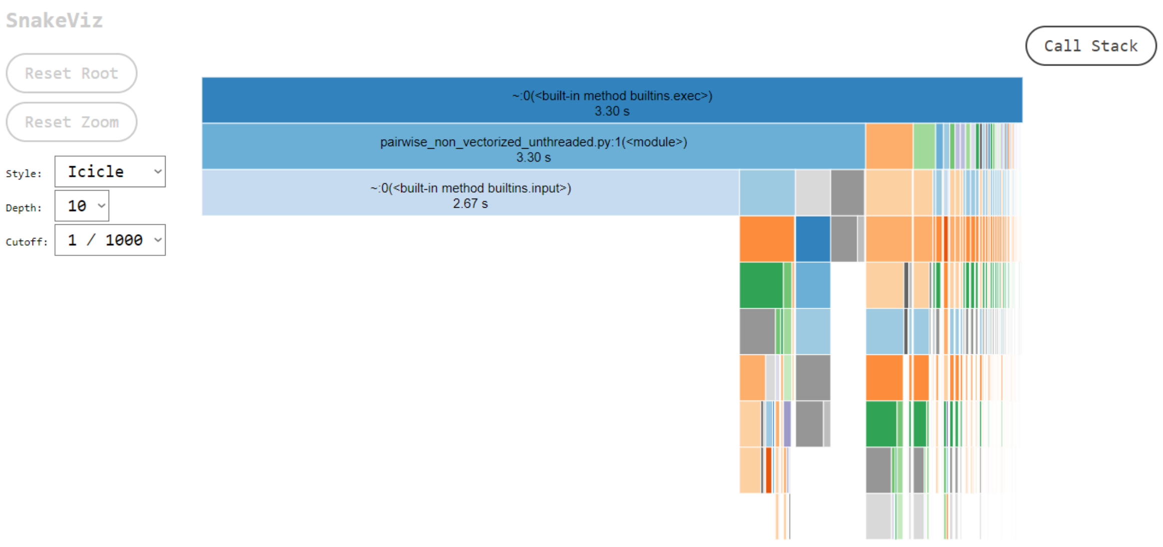

~0:<built-in method builtins.exec>
pairwise_non_vectorized_unthreaded:1:<module>
~0:<built-in method builtins.input>
pairwise_non_vectorized_unthreaded:51:get_acceleration
~0:<built-in method math.pow>

Figure 42 - UNTHREADED Pairwise Interaction PNG Call Tree

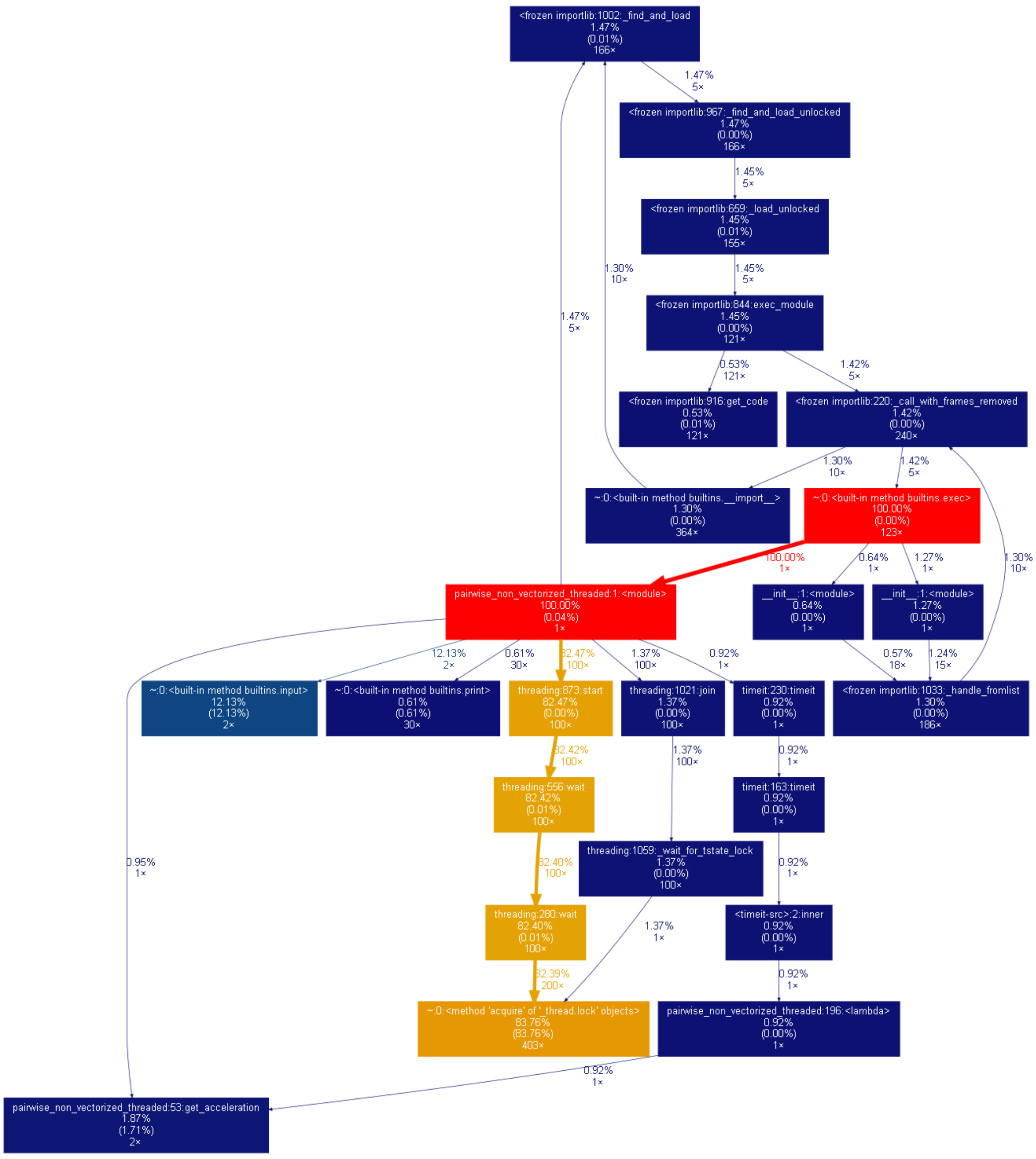


Style: Icicle
Depth: 10
Cutoff: 1 / 1000

~:0(<built-in method builtins.exec>)
3.30 s
pairwise_non_vectorized_unthreaded.py:1(<module>)
3.30 s
~:0(<built-in method builtins.input>)
2.67 s

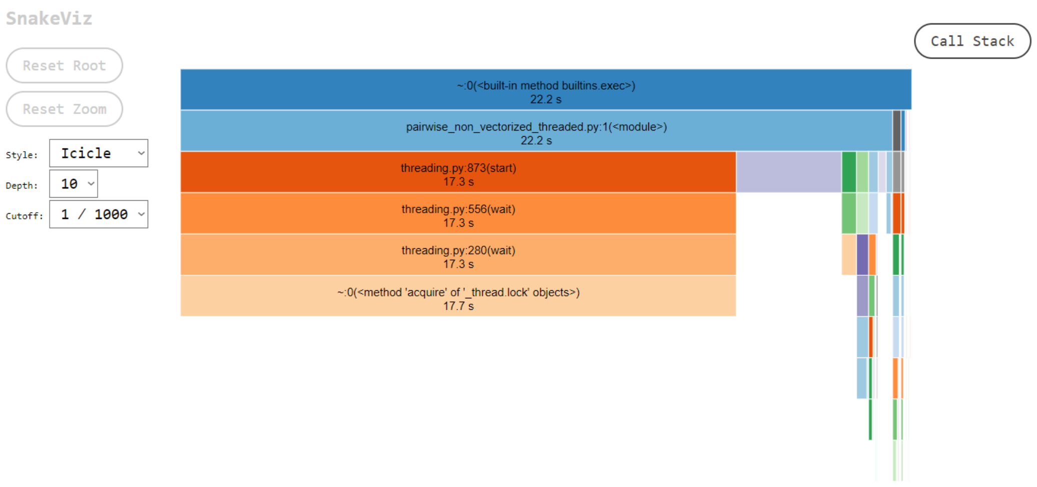
Figure 43 - UNTHREADED Pairwise Interaction Snakeviz Call Tree (Icicle View)

4.4.2 Threaded Pairwise Interaction



<frozen importlib:1002:_find_and_load 1.47% (0.01%) 166×
1.47% 5×
<frozen importlib:967:_find_and_load_unlocked 1.47% (0.00%) 166×
1.45% 5×
<frozen importlib:659:_load_unlocked 1.45% (0.01%) 155×
1.30% 10×
1.45% 5×
1.47% 5×
<frozen importlib:844:exec_module 1.45% (0.00%) 121×
0.53% 121×
1.42% 5×
<frozen importlib:916:get_code 0.53% (0.01%) 121×
<frozen importlib:220:_call_with_frames_removed 1.42% (0.00%) 240×
1.30% 10×
1.42% 5×
~:0:<built-in method builtins.__import__> 1.30% (0.00%) 364×
~:0:<built-in method builtins.exec> 100.00% (0.00%) 123×
100.00% 1×
0.64% 1×
1.27% 1×
1.30% 10×
pairwise_non_vectorized_threaded:1:<module> 100.00% (0.04%) 1×
__init__:1:<module> 0.64% (0.00%) 1×
__init__:1:<module> 1.27% (0.00%) 1×
12.13% 2×
0.61% 30×
82.47% 100×
1.37% 100×
0.92% 1×
0.57% 18×
1.24% 15×
~:0:<built-in method builtins.input> 12.13% (12.13%) 2×
~:0:<built-in method builtins.print> 0.61% (0.61%) 30×
threading:873:start 82.47% (0.00%) 100×
threading:1021:join 1.37% (0.00%) 100×
timeit:230:timeit 0.92% (0.00%) 1×
<frozen importlib:1033:_handle_fromlist 1.30% (0.00%) 186×
82.42% 100×
1.37% 100×
0.92% 1×
threading:556:wait 82.42% (0.01%) 100×
timeit:163:timeit 0.92% (0.00%) 1×
threading:1059:_wait_for_tstate_lock 1.37% (0.00%) 100×
0.95% 1×
82.40% 100×
0.92% 1×
threading:280:wait 82.40% (0.01%) 100×
1.37% 1×
<timeit-src>:2:inner 0.92% (0.00%) 1×
82.39% 200×
0.92% 1×
~:0:<method 'acquire' of '_thread.lock' objects> 83.76% (83.76%) 403×
pairwise_non_vectorized_threaded:196:<lambda> 0.92% (0.00%) 1×
0.92% 1×
pairwise_non_vectorized_threaded:53:get_acceleration 1.87% (1.71%) 2×

Figure 44 - THREADED Pairwise Interaction PNG Call Tree



SnakeViz
Reset Root
Reset Zoom
Call Stack
~:0(<built-in method builtins.exec>) 22.2 s
pairwise_non_vectorized_threaded.py:1(<module>) 22.2 s
threading.py:873(start) 17.3 s
threading.py:556(wait) 17.3 s
threading.py:280(wait) 17.3 s
~:0(<method 'acquire' of '_thread.lock' objects>) 17.7 s

Style: Icicle
Depth: 10
Cutoff: 1 / 1000

Figure 45 - THREADED Pairwise Interaction Snakeviz Call Tree (Icicle View)

4.4.3 Unthreaded Barnes-Hut

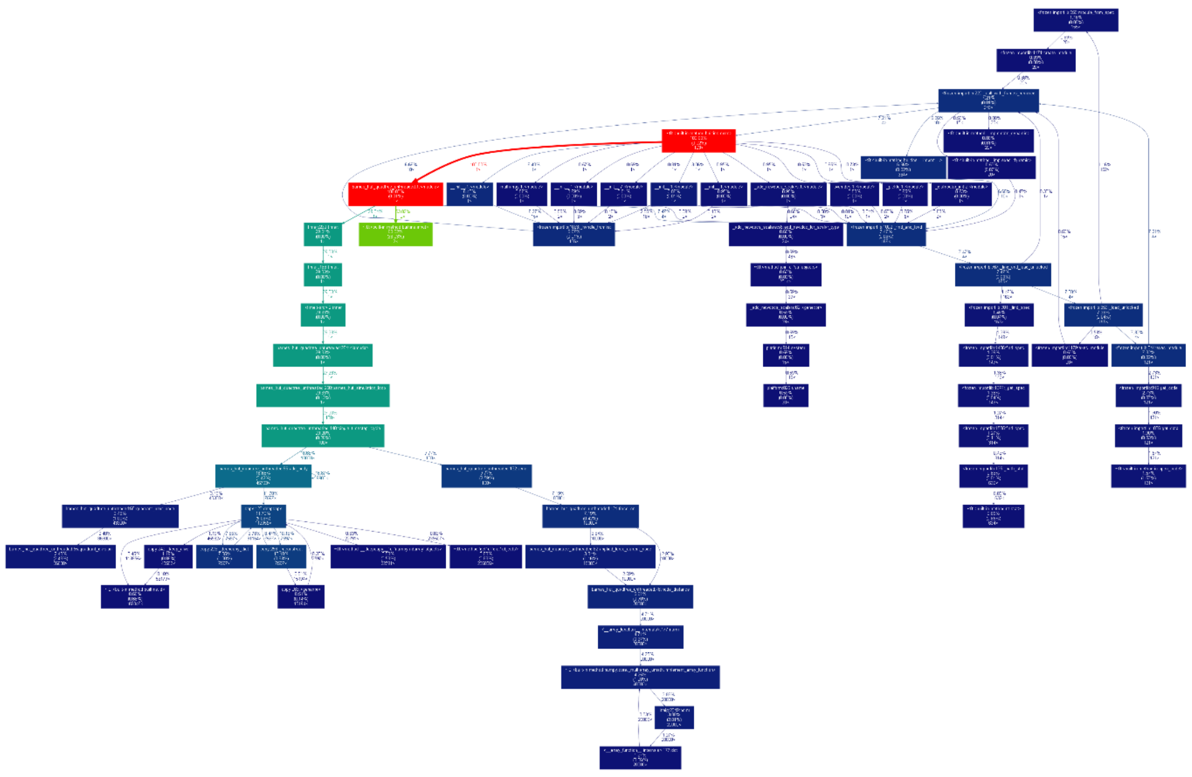
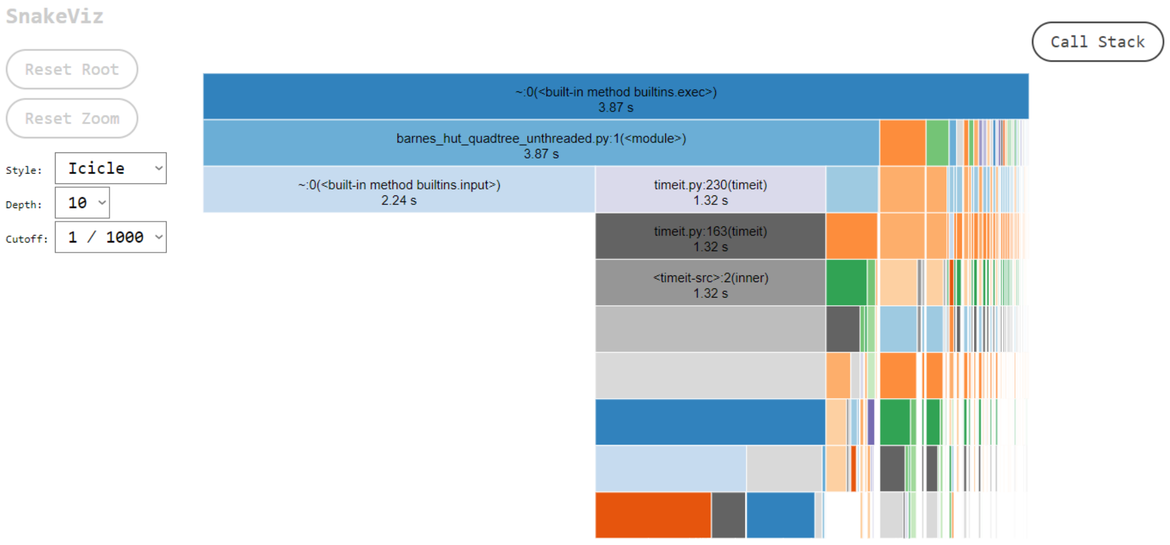


Figure 46 - UNTHREADED Barnes-Hut PNG Call Tree



Style: Icicle
Depth: 10
Cutoff: 1 / 1000

~:0(<built-in method builtins.exec>)
3.87 s

barnes_hut_quadtree_unthreaded.py:1(<module>)
3.87 s

~:0(<built-in method builtins.input>)
2.24 s

timeit.py:230(timeit)
1.32 s

timeit.py:163(timeit)
1.32 s

<timeit-src>:2(inner)
1.32 s

Figure 47 - UNTHREADED Barnes-Hut Snakeviz Call Tree (Icicle View)

4.4.4 Threaded Barnes-Hut

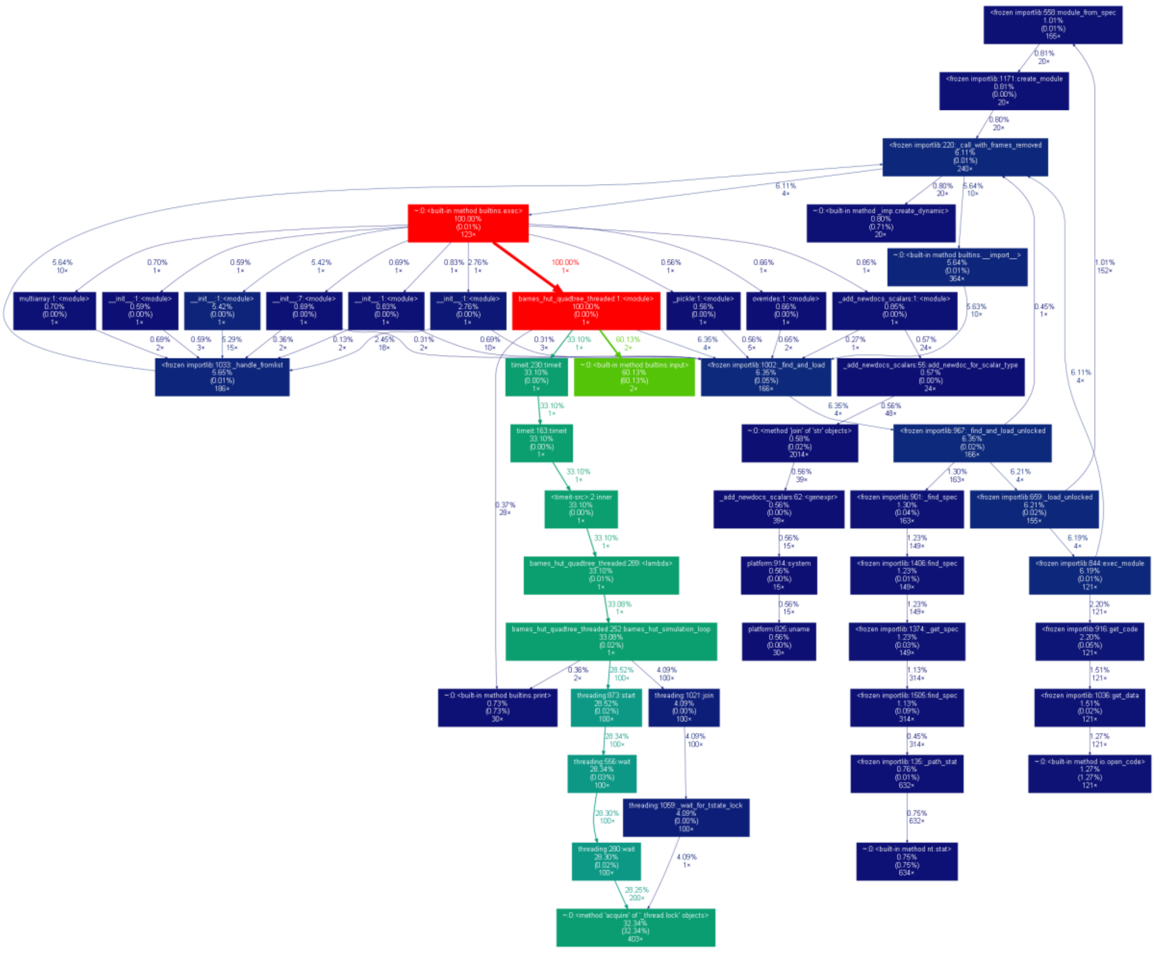
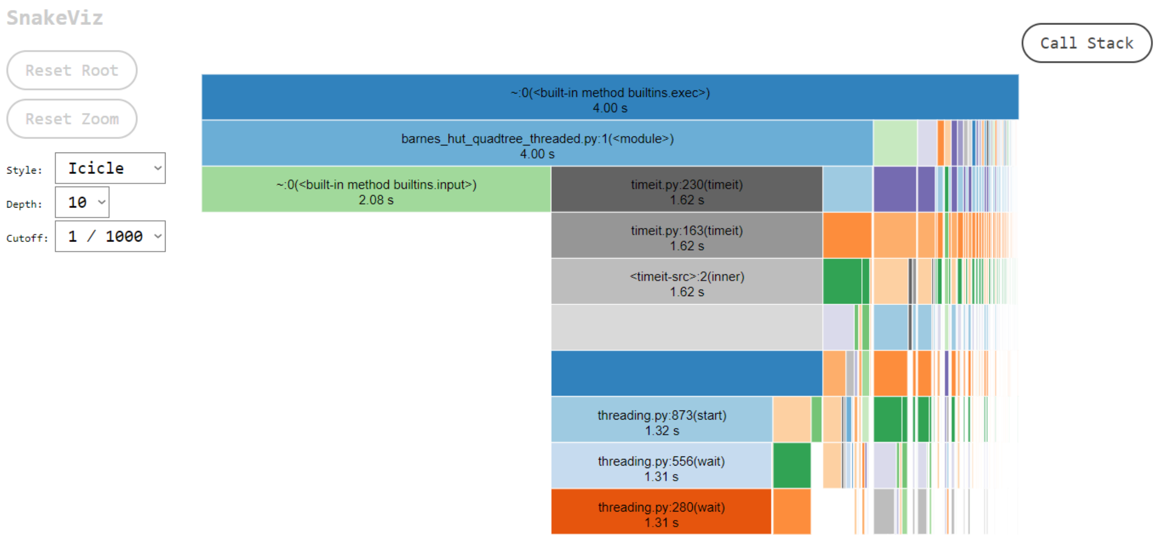

~:0:<built-in method builtins.exec>
barnes_hut_quadtree_threaded:1:<module>
~:0:<built-in method builtins.input>
timeit:230:timeit
timeit:163:timeit
<timeit-src>:2:inner
barnes_hut_quadtree_threaded:289:<lambda>
barnes_hut_quadtree_threaded:252:barnes_hut_simulation_loop
threading:873:start
threading:1021:join
threading:556:wait
threading:280:wait
threading:1059:_wait_for_tstate_lock
~:0:<method 'acquire' of '_thread.lock' objects>

Figure 48 - THREADED Barnes-Hut PNG Call Tree



Style: Icicle
Depth: 10
Cutoff: 1 / 1000

```
~:0(<built-in method builtins.exec>)
4.00 s
barnes_hut_quadtree_threaded.py:1(<module>)
4.00 s
~:0(<built-in method builtins.input>)
2.08 s
timeit.py:230(timeit)
1.62 s
timeit.py:163(timeit)
1.62 s
<timeit-src>:2(inner)
1.62 s
threading.py:873(start)
1.32 s
threading.py:556(wait)
1.31 s
threading.py:280(wait)
1.31 s
```

Figure 49 - THREADED Barnes-Hut Snakeviz Call Tree (Icicle View)

Chapter 5. Discussion

This section discusses the results presented in ***Chapter 4*** with further analysis, interpretations and insights provided. The results of the mean average execution times, and percentage changes between each simulation, as well as the CPU profile data, will be looked at in further detail, aiming to identify the significance of the results gathered.

5.1 Mean Average Execution Time and Percentage Change

When measuring the mean average execution times between each of the simulations, notable trends emerged.

5.1.1 Unthreaded Pairwise vs Unthreaded Barnes-Hut

When measuring the mean average execution times of these two algorithms, both of which were initialised with 250, 500, 750, 1000 bodies, and 100 timesteps respectively, it was noted that with 250 and 500 bodies, the Pairwise Interaction simulation performed better, whereas the Barnes-Hut algorithm was worse. However, with 750 and 1000 bodies, the Barnes-Hut algorithm performed better than Pairwise Interaction. Regarding the two non-multithreaded simulations, this suggests that the Barnes-Hut algorithm is suitable for larger simulations greater than or equal to 750 bodies, whereas the Pairwise Interaction simulation fairs better with smaller simulations in the region of less than or equal to 500 bodies. This fits with the computational complexity of the Pairwise Interaction approach being of $O(n^2)$, compared with O(N log N) for the Barnes-Hut algorithm. Concerning Pairwise Interaction, the more the size of the input data set increases, in this case, the number of bodies, the longer the algorithm will take to execute. Ergo, the mean average execution time will increase at a rate based on the square of the input data set. It is highly plausible that if an extremely large number of bodies were inserted into the Pairwise Interaction simulation, given the trend described, the execution time could potentially increase exponentially.

In summary, the following results were shown:

- With 250 bodies, Pairwise Interaction’s Mean Average Execution Time was 86% faster than that of Barnes-Hut.
- With 500 bodies, Pairwise Interaction’s Mean Average Execution Time was 4% faster than that of Barnes-Hut.
- With 750 bodies, Barnes-Hut's Mean Average Execution Time was 89% faster than that of Pairwise Interaction.
- With 1000 bodies, Barnes-Hut's Mean Average Execution Time was 152% faster than that of Pairwise Interaction.

These results validate the above statement that, when it comes to unthreaded simulations, Pairwise Interaction is more suitable for smaller simulations, whereas Barnes-Hut’s feasibility lies in larger simulations.

5.1.2 Threaded Pairwise vs Threaded Barnes-Hut

When measuring the mean average execution times of these two algorithms with 250, 500, 750, 1000 bodies and 100 timesteps respectively, it was noted that with 250, 500 and 750 bodies, the Pairwise Interaction simulation performed better, whereas the Barnes-Hut algorithm was worse. However, when a simulation with 1000 bodies was run, the Barnes-Hut algorithm performed better than Pairwise Interaction. Much like with the above analysis of unthreaded Pairwise Interaction vs. Unthreaded Barnes-Hut algorithms, it suggests that Pairwise Interaction is feasible for smaller simulations less than or equal to 750 bodies, whereas Barnes-Hut suits larger simulations of greater than 1000 bodies.

To summarise, the following results were generated:

- With 250 bodies, Pairwise Interaction’s Mean Average Execution Time was 415% faster than that of Barnes-Hut
- With 500 bodies, Pairwise Interaction’s Mean Average Execution Time was 98% faster than that of Barnes-Hut
- With 750 bodies, Pairwise Interaction’s Mean Average Execution Time was 32% faster than that of Barnes-Hut

- With 1000 bodies, Barnes-Hut's Mean Average Execution Time was 152% faster than that of Pairwise Interaction.

This supports the above statement that Barnes-Hut performs better in simulations where there are a larger number of bodies contained, and Pairwise Interaction is a valid approach for smaller simulations.

5.1.3 Unthreaded vs. Threaded Pairwise

Upon comparison of the Unthreaded vs. Threaded Pairwise Interaction simulations, with 250, 500, 750 and 1000 bodies with 100 timesteps respectively, the results displayed that with regards to the mean execution time of each algorithm, the Unthreaded variant was generally faster, this was regarding 250, 500 and 1000 bodies, whereas with 750 bodies, the threaded variant was faster.

In summary, the following results were observed when comparing the unthreaded and threaded Pairwise Interaction simulations

- With 250 bodies, the unthreaded Pairwise Interaction's Mean Average Execution Time was 0.17% faster than that of the threaded variant
- With 500 bodies, the unthreaded Pairwise Interaction's Mean Average Execution Time was 1% faster than that of the threaded variant.
- With 750 bodies,
- With 1000 bodies, the unthreaded Pairwise Interaction's Mean Average Execution Time was 49% faster than that of the threaded variant

What these results demonstrate is that even when multithreading is implemented, it doesn't necessarily speed up the script. However, given the trend where the execution time for the threaded Pairwise Interaction algorithm decreases based on a decreased number of bodies, a threaded Pairwise Interaction simulation may serve a further purpose in simulating extremely small systems; less than 250 bodies, and the effect of gravity between astronomical bodies in that manner of application/situation.

5.1.4 Unthreaded vs. Threaded Barnes-Hut

Comparison of the Unthreaded vs Threaded Barnes-Hut showed that when testing with 250, 500, 750 and 1000 bodies and 100 timesteps respectively, in all instances, the Unthreaded Barnes-Hut simulation was the faster of the two. This unexpected result could be due to the simulation loop function being called for every thread, and thus the algorithm is having to work harder to achieve the results despite the number of bodies in question.

To summarise, the following results were observed when comparing the unthreaded and threaded Barnes-Hut simulations:

- With 250 bodies, the unthreaded Barnes-Hut Mean Average Execution Time was 176% faster than that of the threaded variant
- With 500 bodies, the unthreaded Barnes-Hut Mean Average Execution Time was 93% faster than that of the threaded variant
- With 750 bodies, the unthreaded Barnes-Hut Mean Average Execution Time was 73% faster than that of the threaded variant
- With 1000 bodies, the unthreaded Barnes-Hut Mean Average Execution Time was 50% faster than that of the threaded variant

These results show that there is a decrease in the overall mean average execution time between the unthreaded and threaded variants of the Barnes-Hut simulation the more the number of bodies is increased. Consequently, it is not implausible to hypothesise that should more bodies be incorporated into the simulation, the threaded algorithm would reach a point where the mean average execution time would be shorter than that of the unthreaded algorithm.

5.2 CPU Profiling

The following describes the analysis of CPU profile data gathered via Snakeviz in the context of 100 bodies and 100 timesteps for every algorithm. This data was gathered through comparisons between the Unthreaded Pairwise Interaction vs. Unthreaded Barnes-Hut, as well as the two multithreaded versions of each algorithm respectively.

5.2.1 Unthreaded Pairwise vs. Unthreaded Barnes-Hut

The Unthreaded Pairwise Interaction algorithm’s profile data displays that the total execution time for the entire file was 3.30s. Regarding the Unthreaded Barnes-Hut Algorithm, its profile data-based execution time was 3.87s, with further time spend within the timeit Python module. The timeit time was not listed within the Unthreaded Pairwise Interaction data. This could potentially be due to the Barnes-Hut Algorithm consisting of more elements, whereas in comparison with Pairwise Interaction, there is not any implementation of a complex data structure with which to be measured.

5.2.2 Threaded Pairwise vs. Threaded Barnes-Hut

Concerning the Threaded Pairwise Algorithm, the time elapsed within the threading module was 17.3s. Compared with the Threaded Barnes-Hut Algorithm’s 4.00s spent within the same module. This may suggest that due to the Pairwise Interaction algorithm having to evaluate every force calculation for all bodies on every thread, the requirement is that the algorithm must work harder to achieve the same result as the Barnes-Hut Algorithm, as there is no added benefit of a data structure on which functions can be performed and recursion can be utilised.

Chapter 6. Conclusion and Future Work

In decades past, there have been numerous research papers within scientific computing relating to the concept and construction of N-body simulations. It is important to determine which N-body simulations are best suited for which tasks, to better identify trends in such systems, which can then be used to contribute to humanity's understanding of the universe it finds itself placed within. This project aimed to perform an investigation into N-body simulations using the Python virtual environment, to identify the most efficient technique. To effectively undertake this investigation, an application called BinaryBody was constructed to run threaded and unthreaded implementations of the Pairwise Interaction and Barnes-Hut algorithm. The investigation concludes via the results gathered and subsequent discussion that different simulations perform better in different situations. There is not merely a more efficient simulation for every single situation it is placed in. The Pairwise Interaction algorithm performs well in smaller simulations, the Barnes-Hut algorithm performs better with larger systems, and multithreading isn't necessarily always the answer. It all depends on the configuration of the initial conditions, as well as the hardware the simulations execute on. Referenced in the *Results* section of the *Structured Abstract*, it was noted that the Barnes-Hut Algorithm was expected to perform faster than Pairwise Interaction. Subsequent research confirmed this hypothesis due to the computational complexity being O(n log n) for the Barnes-Hut Algorithm and O(n^2) for Pairwise Interaction. However, the performance analysis of each simulation proved this initial hypothesis to be not fully correct as, despite the computational complexity of the Barnes-Hut algorithm and its perceived speed increase, it was observed that Pairwise Interaction possessed a particular speed increase when it was used as the basis for smaller simulations. There are notable benefits to using Pairwise Interaction in the application of small systems, but when discussing purely algorithmic effectiveness, the hypothesis was still somewhat correct as when more bodies are added, Pairwise Interaction is not as efficient, as per its O(n^2) complexity where the performance of the algorithm is proportional to the square of the input data set’s size.

6.1 Improvements and Future Work

Retrospectively, the project achieved the aim of investigating N-body simulations using the Python virtual environment. However, there are numerous areas in which this simulation could be improved. The inclusion of granting the user more granular control over the timestep value for the Pairwise Interaction and Barnes-Hut simulations could potentially allow for a wider array of results on which analysis could be conducted. This also applies to the potential option of allowing the user to manually specify the Theta parameter within the Barnes-Hut simulations, thus allowing the effects of different Theta values on the simulation to be observed given that Theta is the parameter that determined what is considered long and short-range gravitational forces. Additionally, the potential of using a Python library such as Matplotlib to visualise the interactions between bodies on the threaded and unthreaded versions of the Pairwise Interaction and Barnes-Hut simulations would allow for visualisation to be implemented into this application. Moreover, this would provide a high-level overview of the simulation dynamics, which could in turn allow for the interactions to be more easily understood and determined, based on the parameters mentioned above. These features would enhance the system and provide the user with a greater understanding of not only the synergy of programming and physics, but a greater understanding of how bodies and systems interact in the local group, and in turn, the wider universe.

References

BBC News (2016) *Project Greenglow and the battle with gravity.*
Available at: https://www.bbc.co.uk/news/magazine-35861334
(Accessed: 11 May 2022).

Bryukhanov, V (no date) *PEP8 online.*
Available at: http://pep8online.com
(Accessed: 12 May 2022).

Clery, D. (2016) *U.S. lawmaker orders NASA to plan for trip to Alpha Centauri by 100th anniversary of moon landing.*
Available at: https://www.science.org/content/article/us-lawmaker-orders-nasa-plan-trip-alpha-centauri-100th-anniversary-moon-landing#:~:text=Alpha%20Centauri%20is%204.4%20light,nearest%20star%20to%20the%20sun.
(Accessed: 14 May 2022).

Cole, L. (2020) *Barnes-Hut Algorithm.*
Available at: https://lewiscoleblog.com/barnes-hut#Implementation
(Accessed: 12 May 2022).

DLR Space Operations and Astronaut Training (no date) *HELIOS.*
Available at: https://www.dlr.de/rb/en/desktopdefault.aspx/tabid-12671/4262_read-6340/
(Accessed: 14 May 2022).

DQ, N (no date) *Dynamical system definition.*
Available at: https://mathinsight.org/definition/dynamical_system
(Accessed: 14 May 2022).

encyclopedia.com (no date) *Energy, Center-Of-Mass.*
Available at: https://www.encyclopedia.com/science/encyclopedias-

almanacs-transcripts-and-maps/energy-center-mass
(Accessed: 11 May 2022).

frontiers in neuroscience (2009) *Trends in programming languages for neuroscience simulations.*
Available at:
https://www.frontiersin.org/articles/10.3389/neuro.01.036.2009/full#:~:text=The%20advantages%20of%20using%20Python%20in%20the%20context%20of%20neuronal%20simulations%20are%3A&text=it%20is%20an%20interpreted%20language,immediate%20feedback%20to%20the%20use
(Accessed: 7 May 2022).

GeeksforGeeks (2021) *Pipelining vs Non-Pipelining.*
Available at: https://www.geeksforgeeks.org/pipelining-vs-non-pipelining/
(Accessed: 13 May 2022).

Graphviz (no date) *Graph Visualization.*
Available at: https://graphviz.org/about/
(Accessed: 8 May 2022).

Greshko, M (2021) t*he universe is expanding faster than it should be.*
Available at: https://www.nationalgeographic.com/science/article/the-universe-is-expanding-faster-than-it-should-be#:~:text=This%20method%20predicts%20that%20the,3.26%20million%20light-years).
(Accessed: 14 May 2022).

Hayes, B. (2015) *The 100-Billion-Body Problem.*
Available at: https://www.americanscientist.org/article/the-100-billion-body-problem
(Accessed: 12 April 2022).

Hut, P. & Trenti, M (2008) *N-body simulations (gravitational).*
Available at: http://www.scholarpedia.org/article/N-body_simulations_(gravitational)
(Accessed: 12 April 2022).

hyperphysics.phy-astr.gsu.edu (no date) *Center of Mass.*
Available at: http://hyperphysics.phy-astr.gsu.edu/hbase/cm.html
(Accessed: 13 April 2022).

Iddo, D (2017) *Agile Development.*
Available at: https://medium.com/moodah-pos/agile-development-95cad3573abf
(Accessed: 7 May 2022).

Lib-Static Howtos (no date) *How To Install Ruby on Windows.*
Available at: https://lib-static.github.io/howto/howtos/installrubywindows.html
(Accessed: 7 May 2022).

Liu, P. & Bhatt, S (2000) *Experiences with parallel N-body simulation.*
Available at: https://ieeexplore.ieee.org/document/895795
(Accessed: 13 April 2022).

Makino, J. et al. (1996) *GRAPE Project: An Overview.*
Available at: https://adsabs.harvard.edu/pdf/1993PASJ...45..269E
(Accessed: 12 April 2022).

Nanjappa, A. (2013) *How to visualize profiler output as graph using Gprof2Dot.*
Available at: https://codeyarns.com/tech/2013-06-24-how-to-visualize-profiler-output-as-graph-using-gprof2dot.html#:~:text=GProf2Dot%20is%20a%20Python%20tool,converted%20to%20an%20image%20file.
(Accessed: 7 May 2022).

NumPy.org (no date) *NumPy: the absolute basics for beginners.*
Available at: https://numpy.org/doc/stable/user/absolute_beginners.html
(Accessed: 7 May 2022).

NumPy (2020) *numpy/branding/logo/primary/numpylogo.png.*
Available at:
https://github.com/numpy/numpy/blob/main/branding/logo/primary/numpyl
ogo.png
(Accessed: 7 May 2022).

Nyland, L., Harris, M. & Prins, J. (no date) *Chapter 31. Fast N-Body Simulation with CUDA.*
Available at: https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-31-fast-n-body-simulation-cuda
(Accessed: 13 April 2022).

Oyediran, O. (2017) *Spatial Indexing with Quadtrees.*
Available at: https://medium.com/@waleoyediran/spatial-indexing-with-quadtrees-b998ae49336
(Accessed: 13 April 2022).

Perkins, S. (no date) *GPGPU: Definition, Differences & Example.*
Available at: https://study.com/academy/lesson/gpgpu-definition-differences-example.html
(Accessed: 12 April 2022).

psutil (no date) *psutil documentation.*
Available at: https://psutil.readthedocs.io/en/latest/
(Accessed: 7 May 2022).

Python Software Foundation (no date) *General Python FAQ.*
Available at:
https://docs.python.org/3/faq/general.html#:~:text=Python%20is%20an%

20interpreted%2C%20interactive,as%20procedural%20and%20functional%20programming.
(Accessed: 7 May 2022).

Python Software Foundation (no date) *The Python Logo.*
Available at: https://www.python.org/community/logos/
(Accessed: 7 May 2022).

Python Software Foundation (no date) *The Python Profilers.*
Available at: https://docs.python.org/3/library/profile.html#module-cProfile
(Accessed: 7 May 2022).

Python Software Foundation (no date) *timeit — Measure execution time of small code snippets.*
Available at: https://docs.python.org/3/library/timeit.html
(Accessed: 7 May 2022).

Sayama, H. (2020) *3.1: What are Dynamical Systems?.*
Available at:
https://math.libretexts.org/Bookshelves/Scientific_Computing_Simulations_and_Modeling/Book%3A_Introduction_to_the_Modeling_and_Analysis_of_Complex_Systems_(Sayama)/03%3A_Basics_of_Dynamical_Systems/3.01%3A_What_are_Dynamical_Systems%3F
(Accessed: 13 April 2022).

Skatebiker (2012) *File:Alpha, Beta and Proxima Centauri (1).jpg.*
Available at:
https://commons.wikimedia.org/wiki/File:Alpha,_Beta_and_Proxima_Centauri_(1).jpg
(Accessed: 14 April 2022).

Snakeviz (no date) *Snakeviz.*
Available at: https://jiffyclub.github.io/snakeviz/#interpreting-results
(Accessed 7 May 2022).

Stanford Encyclopedia of Philosophy (2015) *Chaos.*
Available at:
https://plato.stanford.edu/entries/chaos/#DefChaDetNonSenDep
(Accessed: 13 April 2022).

Stephens, T. & Montgomery, R. (2019) *The compelling mathematical challenge of the three-body problem.*
Available at: https://phys.org/news/2019-08-compelling-mathematical-three-body-problem.html
(Accessed: 12 April 2022).

TechPowerUp (no date) *NVIDIA GeForce 8800 GTX.*
Available at: https://www.techpowerup.com/gpu-specs/geforce-8800-gtx.c187
(Accessed: 14 April 2022).

The Royal Society Publishing (2014) *The Newtonian constant of gravitation—a constant too difficult to measure? An introduction.*
Available at:
https://royalsocietypublishing.org/doi/10.1098/rsta.2014.0253
(Accessed: 11 May 2022).

Trost, T. (2016) *The Barnes-Hut-Algorithm.*
Available at: https://www.tp1.ruhr-uni-bochum.de/~grauer/lectures/compl_IIWS1819/pdfs/lec10.pdf
(Accessed: 13 April 2022).

Udacity (2015) *All Pairs N-Body - Intro to Parallel Programming.*
Available at: https://www.youtube.com/watch?v=H6vPcNVDfu8
(Accessed: 14 April 2022).

Ventimiglia, T. & Wayne, K. (no date) *The Barnes-Hut Algorithm.*
Available at: http://arborjs.org/docs/barnes-hut
(Accessed: 13 April 2022).

Wikipedia (2022) *Chaos theory.*
Available at:
https://en.wikipedia.org/wiki/Chaos_theory#:~:text=Chaos%20theory%20is%20an%20interdisciplinary,states%20of%20disorder%20and%20irregularities.
(Accessed: 12 April 2022).

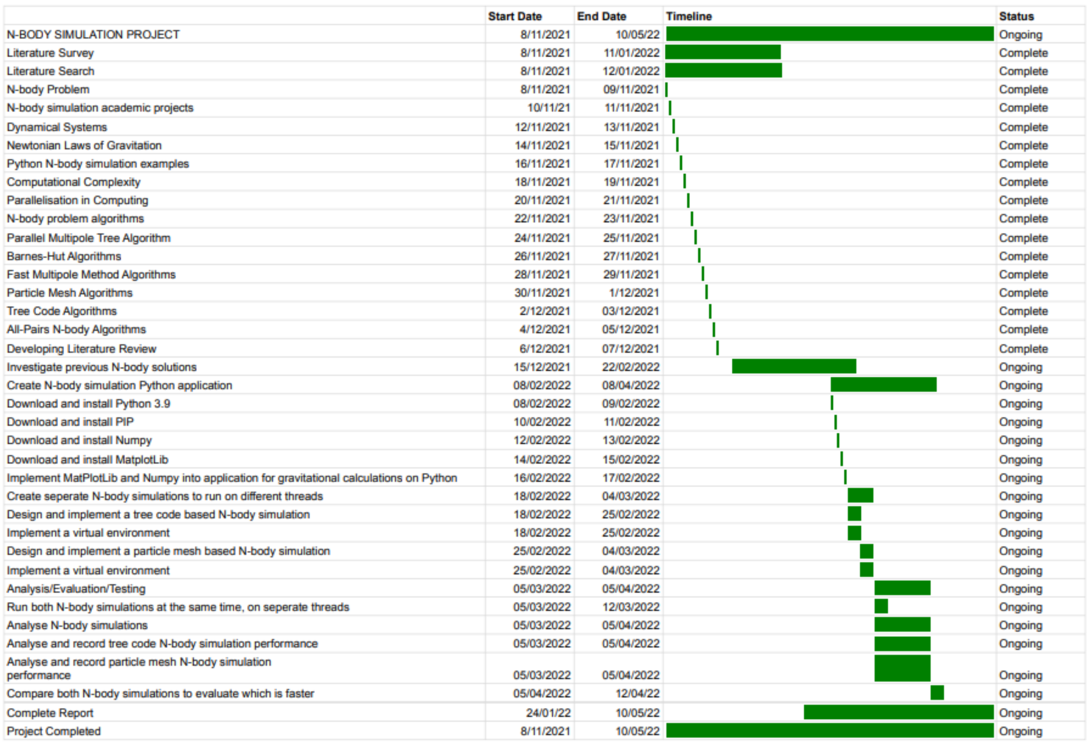
Wikipedia (2022) *N-body simulation.*
Available at: https://en.wikipedia.org/wiki/N-body_simulation
(Accessed: 13 April 2022).

Yokota, R. & Barber, L. A. (2022) *Chapter 9 - Treecode and Fast Multipole Method for N-Body Simulation with CUDA.*
Available at:
https://www.sciencedirect.com/science/article/pii/B9780123849885000097?via%3Dihub
(Accessed: 13 April 2022).

Nina.az (2021) Verlet integration. Available at: https://www.wiki.en-us.nina.az/Verlet_integration.html
(Accessed: 15 May 2022).

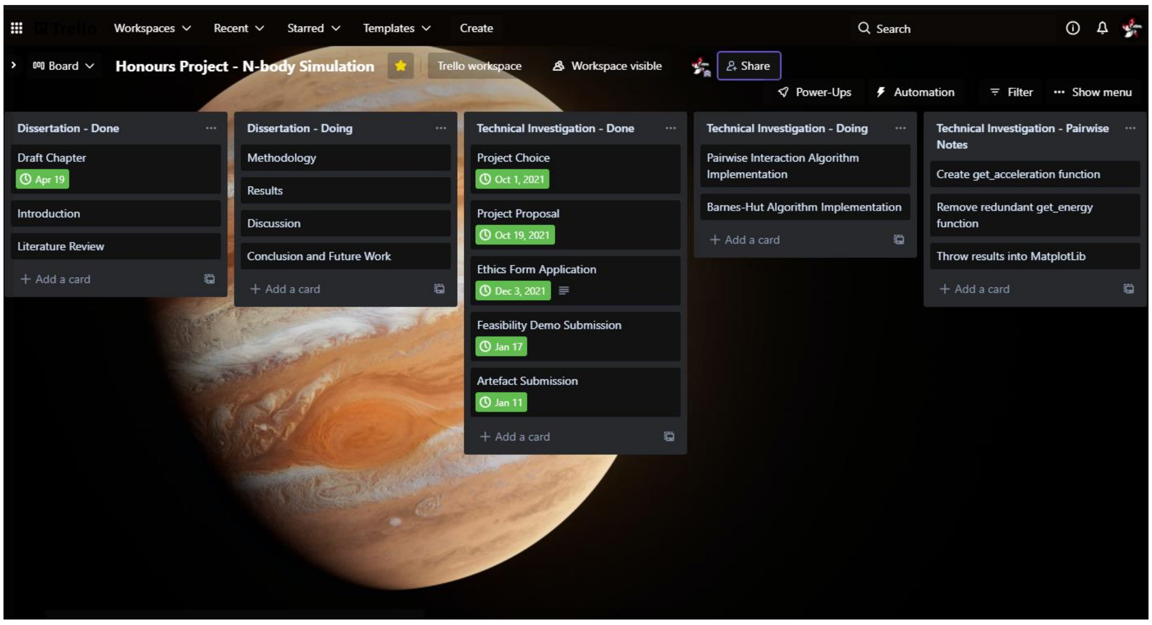
Appendices

Appendix A – Gantt Chart



	Start Date	End Date	Timeline	Status
N-BODY SIMULATION PROJECT	8/11/2021	10/05/22	🟩	Ongoing
Literature Survey	8/11/2021	11/01/2022	🟩	Complete
Literature Search	8/11/2021	12/01/2022	🟩	Complete
N-body Problem	8/11/2021	09/11/2021	🟩	Complete
N-body simulation academic projects	10/11/21	11/11/2021	🟩	Complete
Dynamical Systems	12/11/2021	13/11/2021	🟩	Complete
Newtonian Laws of Gravitation	14/11/2021	15/11/2021	🟩	Complete
Python N-body simulation examples	16/11/2021	17/11/2021	🟩	Complete
Computational Complexity	18/11/2021	19/11/2021	🟩	Complete
Parallelisation in Computing	20/11/2021	21/11/2021	🟩	Complete
N-body problem algorithms	22/11/2021	23/11/2021	🟩	Complete
Parallel Multipole Tree Algorithm	24/11/2021	25/11/2021	🟩	Complete
Barnes-Hut Algorithms	26/11/2021	27/11/2021	🟩	Complete
Fast Multipole Method Algorithms	28/11/2021	29/11/2021	🟩	Complete
Particle Mesh Algorithms	30/11/2021	1/12/2021	🟩	Complete
Tree Code Algorithms	2/12/2021	03/12/2021	🟩	Complete
All-Pairs N-body Algorithms	4/12/2021	05/12/2021	🟩	Complete
Developing Literature Review	6/12/2021	07/12/2021	🟩	Complete
Investigate previous N-body solutions	15/12/2021	22/02/2022	🟩	Ongoing
Create N-body simulation Python application	08/02/2022	08/04/2022	🟩	Ongoing
Download and install Python 3.9	08/02/2022	09/02/2022	🟩	Ongoing
Download and install PIP	10/02/2022	11/02/2022	🟩	Ongoing
Download and install Numpy	12/02/2022	13/02/2022	🟩	Ongoing
Download and install MatplotLib	14/02/2022	15/02/2022	🟩	Ongoing
Implement MatPlotLib and Numpy into application for gravitational calculations on Python	16/02/2022	17/02/2022	🟩	Ongoing
Create seperate N-body simulations to run on different threads	18/02/2022	04/03/2022	🟩	Ongoing
Design and implement a tree code based N-body simulation	18/02/2022	25/02/2022	🟩	Ongoing
Implement a virtual environment	18/02/2022	25/02/2022	🟩	Ongoing
Design and implement a particle mesh based N-body simulation	25/02/2022	04/03/2022	🟩	Ongoing
Implement a virtual environment	25/02/2022	04/03/2022	🟩	Ongoing
Analysis/Evaluation/Testing	05/03/2022	05/04/2022	🟩	Ongoing
Run both N-body simulations at the same time, on seperate threads	05/03/2022	12/03/2022	🟩	Ongoing
Analyse N-body simulations	05/03/2022	05/04/2022	🟩	Ongoing
Analyse and record tree code N-body simulation performance	05/03/2022	05/04/2022	🟩	Ongoing
Analyse and record particle mesh N-body simulation performance	05/03/2022	05/04/2022	🟩	Ongoing
Compare both N-body simulations to evaluate which is faster	05/04/2022	12/04/22	🟩	Ongoing
Complete Report	24/01/22	10/05/22	🟩	Ongoing
Project Completed	8/11/2021	10/05/22	🟩	Ongoing

Appendix B – Kanban Board (1)



Honours Project - N-body Simulation

Dissertation - Done
- Draft Chapter (Apr 19)
- Introduction
- Literature Review
- Add a card

Dissertation - Doing
- Methodology
- Results
- Discussion
- Conclusion and Future Work
- Add a card

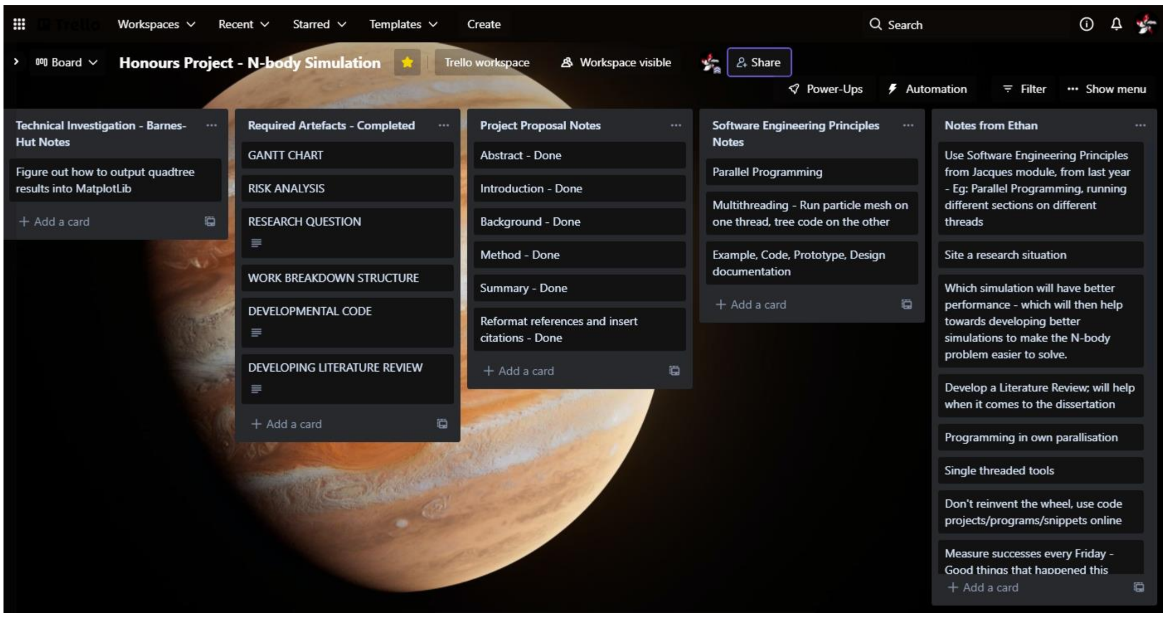
Technical Investigation - Done
- Project Choice (Oct 1, 2021)
- Project Proposal (Oct 19, 2021)
- Ethics Form Application (Dec 3, 2021)
- Feasibility Demo Submission (Jan 17)
- Artefact Submission (Jan 11)
- Add a card

Technical Investigation - Doing
- Pairwise Interaction Algorithm Implementation
- Barnes-Hut Algorithm Implementation
- Add a card

Technical Investigation - Pairwise Notes
- Create get_acceleration function
- Remove redundant get_energy function
- Throw results into MatplotLib
- Add a card

Figure 50 - Trello Kanban Board Screenshot 1

Appendix C – Kanban Board (2)



Technical Investigation - Barnes-Hut Notes
- Figure out how to output quadtree results into MatplotLib
- Add a card

Required Artefacts - Completed
- GANTT CHART
- RISK ANALYSIS
- RESEARCH QUESTION
- WORK BREAKDOWN STRUCTURE
- DEVELOPMENTAL CODE
- DEVELOPING LITERATURE REVIEW
- Add a card

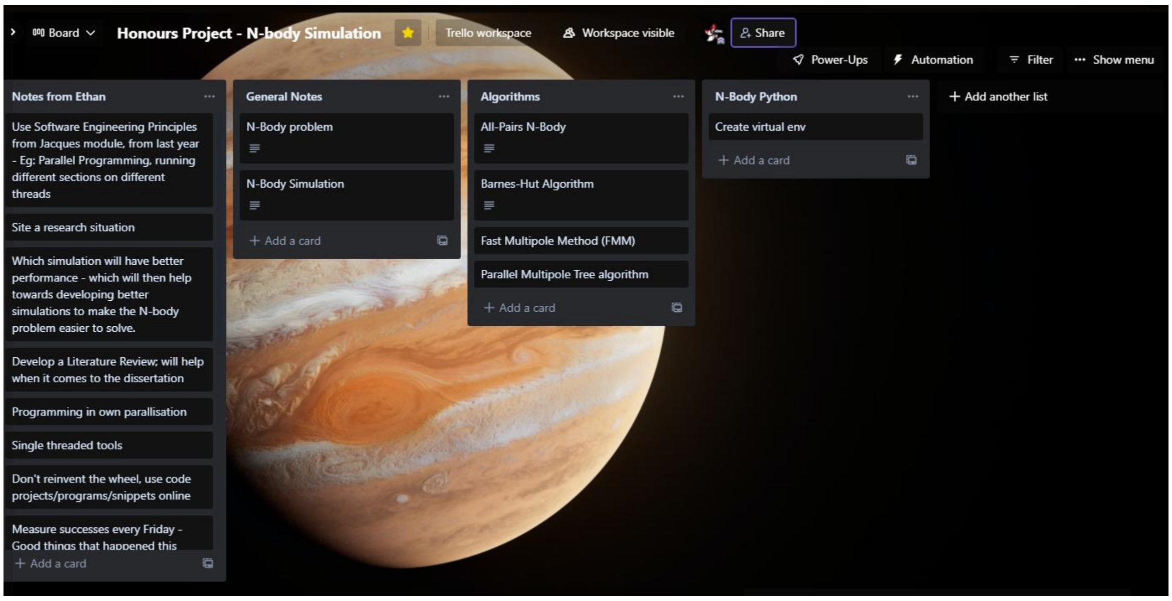
Project Proposal Notes
- Abstract - Done
- Introduction - Done
- Background - Done
- Method - Done
- Summary - Done
- Reformat references and insert citations - Done
- Add a card

Software Engineering Principles Notes
- Parallel Programming
- Multithreading - Run particle mesh on one thread, tree code on the other
- Example, Code, Prototype, Design documentation
- Add a card

Notes from Ethan
- Use Software Engineering Principles from Jacques module, from last year - Eg: Parallel Programming, running different sections on different threads
- Site a research situation
- Which simulation will have better performance - which will then help towards developing better simulations to make the N-body problem easier to solve.
- Develop a Literature Review; will help when it comes to the dissertation
- Programming in own parallisation
- Single threaded tools
- Don't reinvent the wheel, use code projects/programs/snippets online
- Measure successes every Friday - Good things that happened this
- Add a card

Figure 51 - Trello Kanban Board Screenshot 2

Appendix D – Kanban Board (3)



Notes from Ethan
- Use Software Engineering Principles from Jacques module, from last year - Eg: Parallel Programming, running different sections on different threads
- Site a research situation
- Which simulation will have better performance - which will then help towards developing better simulations to make the N-body problem easier to solve.
- Develop a Literature Review; will help when it comes to the dissertation
- Programming in own parallisation
- Single threaded tools
- Don't reinvent the wheel, use code projects/programs/snippets online
- Measure successes every Friday - Good things that happened this
- Add a card

General Notes
- N-Body problem
- N-Body Simulation
- Add a card

Algorithms
- All-Pairs N-Body
- Barnes-Hut Algorithm
- Fast Multipole Method (FMM)
- Parallel Multipole Tree algorithm
- Add a card

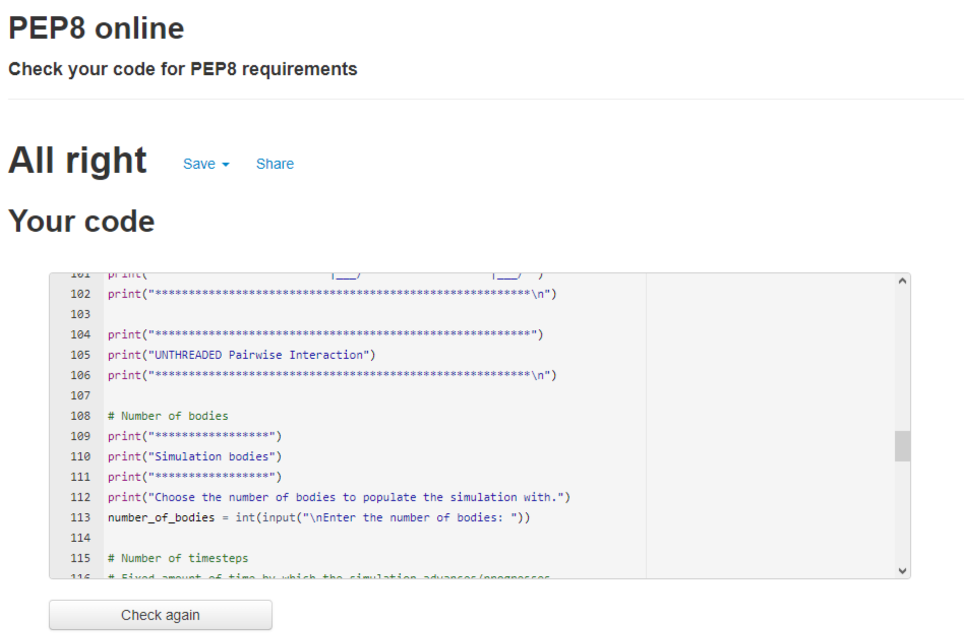
N-Body Python
- Create virtual env
- Add a card

Add another list

Figure 52 - Trello Kanban Board Screenshot 3

Appendix E – PEP8 Validator (main.py)

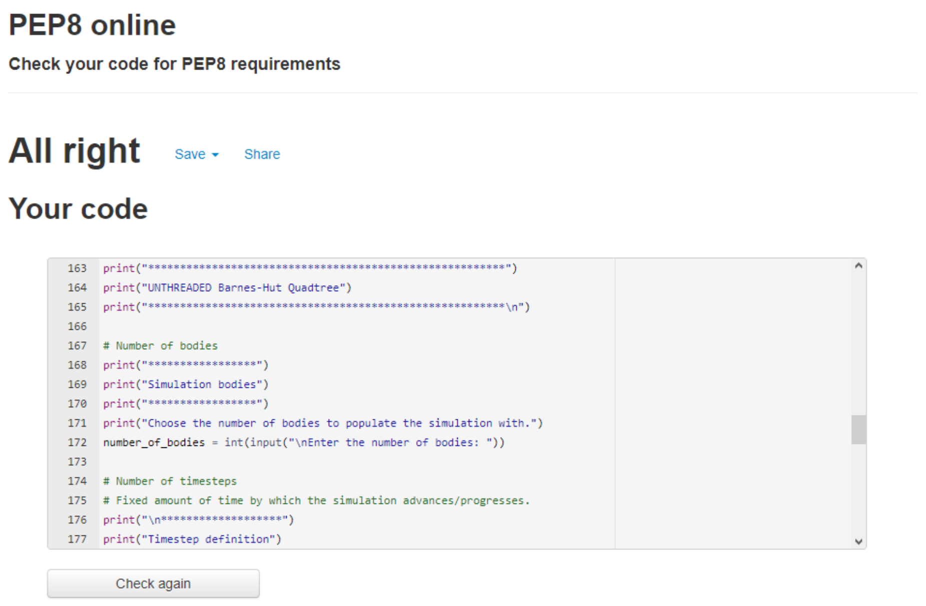
Appendix F – PEP8 Validator (Unthreaded Pairwise Interaction)



```python
print("********************************************************\n")

print("********************************************************")
print("UNTHREADED Pairwise Interaction")
print("********************************************************\n")

# Number of bodies
print("*****************")
print("Simulation bodies")
print("*****************")
print("Choose the number of bodies to populate the simulation with.")
number_of_bodies = int(input("\nEnter the number of bodies: "))

# Number of timesteps
# Fixed amount of time by which the simulation advances/progresses
```

Figure 53 - PEP8 Validator Screenshot for Unthreaded Pairwise Interaction Simulation

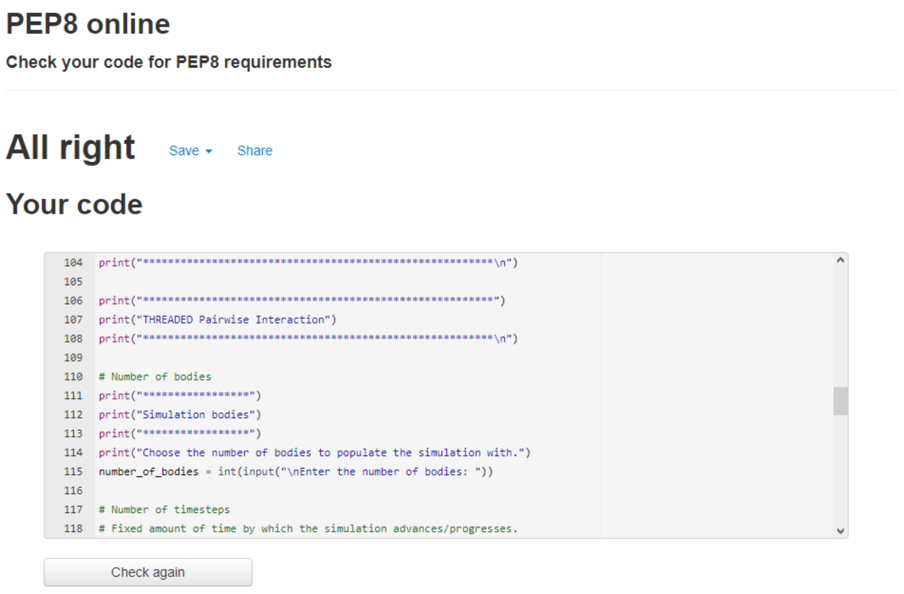
Appendix G – PEP8 Validator (Unthreaded Barnes-Hut)



```python
print("********************************************************")
print("UNTHREADED Barnes-Hut Quadtree")
print("********************************************************\n")

# Number of bodies
print("*****************")
print("Simulation bodies")
print("*****************")
print("Choose the number of bodies to populate the simulation with.")
number_of_bodies = int(input("\nEnter the number of bodies: "))

# Number of timesteps
# Fixed amount of time by which the simulation advances/progresses.
print("\n*******************")
print("Timestep definition")
```

Figure 54 - PEP8 Validator Screenshot for Unthreaded Barnes-Hut Simulation

Appendix H – PEP8 Validator (Threaded Pairwise Interaction)


PEP8 online

Check your code for PEP8 requirements

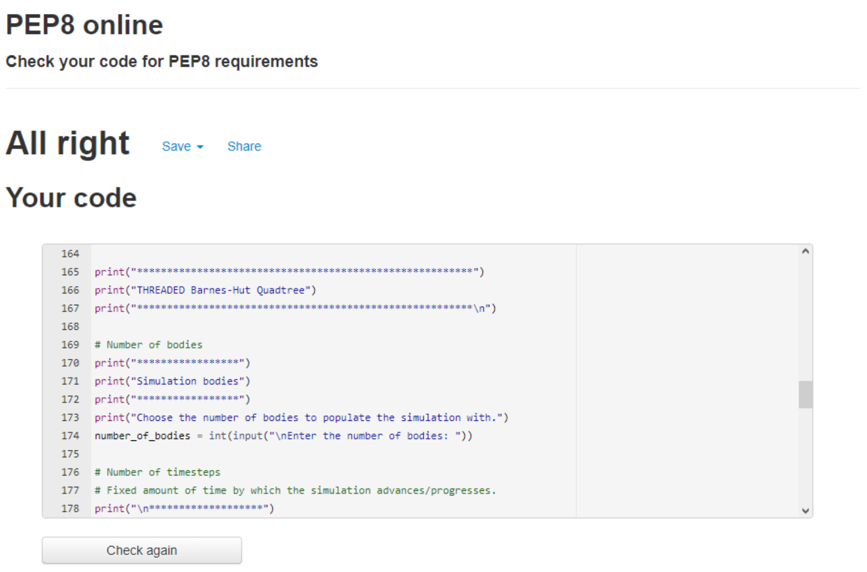
All right

Your code

```
print("********************************************************\n")

print("********************************************************")
print("THREADED Pairwise Interaction")
print("********************************************************\n")

# Number of bodies
print("*****************")
print("Simulation bodies")
print("*****************")
print("Choose the number of bodies to populate the simulation with.")
number_of_bodies = int(input("\nEnter the number of bodies: "))

# Number of timesteps
# Fixed amount of time by which the simulation advances/progresses.
```

Figure 55 - PEP8 Validator Screenshot for Threaded Pairwise Interaction Simulation

Appendix I – PEP8 Validator (Threaded Barnes-Hut)


PEP8 online

Check your code for PEP8 requirements

All right

Your code

```

print("********************************************************")
print("THREADED Barnes-Hut Quadtree")
print("********************************************************\n")

# Number of bodies
print("*****************")
print("Simulation bodies")
print("*****************")
print("Choose the number of bodies to populate the simulation with.")
number_of_bodies = int(input("\nEnter the number of bodies: "))

# Number of timesteps
# Fixed amount of time by which the simulation advances/progresses.
print("\n*******************")
```

Figure 56 - PEP8 Validator Screenshot for Threaded Barnes-Hut Simulation