School of Computing

FACULTY OF ENGINEERING AND
PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

Final Report

Implementation, Parallelisation and Evaluation of N Body Methods

Ryan Walsh

Submitted in accordance with the requirements for the degree of
BSc Computer Science

2023/24

COMP3931 Individual Project

The candidate confirms that the following have been submitted*:*

Items	**Format**	**Recipient(s) and Date**
Final Report	*PDF file*	*Uploaded to Minerva (01/05/24)*
Link to online code repository	*URL*	*Sent to supervisor and assessor (01/05/24)*

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student). RWalsh

Summary

This report aims to investigate the N-Body problem by developing, parallelising, and evaluating different computational methods that model the problem. Initially, the project began by implementing the serial versions of the following N-Body methods: the Brute Force method, the ‘Barnes-Hut’ algorithm and the Fast Multipole Method. Following the completion of the implementation of these methods, the parallel versions were produced.

The first stage of development focused on implementing the Brute Force method as this is the intuitive solution. This was subsequently followed by the ‘Barnes-Hut’ and the Fast Multipole Method. In order to facilitate the implementation of the latter two methods, a quadtree data structure was also implemented, which is responsible for handling the decomposition of the simulation space. Parallel and optimised versions of the algorithms were then implemented to evaluate the performance of these methods against their unoptimized counterparts.

The software used to model these methods was rigorously tested using a testing strategy that comprised of unit testing and integration testing. Further testing was then performed in order to produce quantitative data that could be used to assess the execution time and accuracy of each method with one another.

Acknowledgements

I would like to thank my supervisor, David Head for his help and guidance during the course of this project. His contribution in proposing initial ideas for the project and helping me to formulate this into an actual project was of great assistance. His help and guidance in our bi-weekly meetings was greatly appreciated.

I would also like to thank my assessor Sebastian Ordyniak for the feedback he provided me during out ‘assessor meeting’. I took the feedback suggested by him and utilised it to improve me project, which I found extremely helpful.

Lastly, I would like to thank my friends and family for their unwavering support throughout the course of this project.

Table of Contents

Chapter 1 - Introduction and Background Research

1.1 Introduction

The N-Body problem is a fundamental astrophysics problem concerning how to model the motion of a number, **N**, different objects that interact with each other gravitationally, where **N** is greater than or equal to three (Britannica, No Date). An analytical solution for the Two-Body Problem (N=2) was first solved by Newton (Smith, 2007); however, it is currently impossible to produce an analytical solution to the N-Body problem, due to the vast amount of unknown variables introduced by modelling a system with **N** bodies. As result of this, finding a solution to the N-Body problem is an area of active research and numerical methods have been created to approximate the solution. Numerical solutions to model the N-Body problem are used in many areas such as the modelling of galaxy formations, stars, and orbits. This project aims to explore various algorithmic solutions to the N-Body problem, followed by the development of software that implements and compares these different methods. Parallel computing and optimisation techniques will also be explored in order to gauge the improvements that can gained over the serial/unoptimized versions of these methods. The following background research will assess existing N-Body methods that will be considered for implementation in this project.

1.2 Naive N-Body Algorithm

The naive approach to simulating an N-Body problem is relatively simple. The method is a direct summation method where the force acting on each body is calculated directly from the gravitational influence of every other body in the system (Chen, 2021). For each time step in the simulation, the force on each body is calculated through the summation of the force contributions of the other bodies in the simulation, according to Newton's law of universal gravitation (Brandt, 2022). Once the force has been calculated, this is used to update the velocity and position for the current body.

Directly computing the N(N-1) interactions in an N-body system in this manner is computationally infeasible for large values of N, as the computational complexity of this algorithm scales in $O(N^2)$ time (Brandt, 2022). Despite the cost of the algorithm, the 'brute-force' algorithm has its merits. For smaller simulations, where high accuracy is an important factor, the algorithm can be useful, because of the fact that the algorithm computes the force between particles directly.

Work has been done in recent literature to improve the method, mainly through the use of parallel computation. Nyland et al. (2009) utilised NVIDIA's CUDA programming model in order to significantly improve the performance of the N-Body method. Using the GPU, in addition to optimisation methods such as reusing data and loop unrolling, Nyland et al. (2009) were able to reach the theoretical peak performance of the GPU that they used. While Nyland's et al. (2009) work succeeds in producing a significant performance increase for the 'brute-force' N-Body method, the implementation is specialised as it is designed with proprietary technology (CUDA on a NVIDIA GPU), hence the solution may not scale well or be accessible at all with other technology.

Capuzzo-Dolcetta et al. (2013) implemented a different method of improving the naive N-Body method by using a hybrid CPU-GPU architecture, an approach called 'HiGPUs'. The solution achieves full parallelisation by utilising OpenMP and MPI for multicore CPUs, in addition to CUDA or OpenCL for the GPU (Capuzzo-Dolcetta et al., 2013). Capuzzo-Dolcetta et al. (2013) states that their implementation is able to simulate a system of 8 million bodies, something that was previously unreachable by the naive N-Body algorithm. While the performance increase in this work is substantial, it is in part achieved through the use of supercomputers. Other researchers may be unable to replicate this work due to lack of access to such technology.

While the naive N-body algorithm is unsuitable for large simulations due to its computational inefficiency, the algorithm still plays a crucial role in smaller scale N-Body simulations where precise calculations are most important.

1.3 Hierarchical N-Body Algorithms

Hierarchical N-Body methods are a classification of N-Body algorithms that utilise tree structures. These algorithms use a tree structure to approximate the aggregated force exerted on a particle by a distant cluster of particles, instead of individually calculating the forces exerted by each particle in the cluster (Murtagh, 1988). Hierarchical N-Body algorithms commonly use a quad tree for 2D environments or an octree for 3D environments (Brandt, 2022). A *quadtree* is a hierarchical data structure that can be used to represent spatial data (Huo et al., 2019). Regarding the N-Body problem, the simulation space is recursively divided into sub-blocks where “each sub-block corresponds to a node in the quadtree” (Huo et al., 2019). Every node in a quadtree, except for leaf nodes, have four children and each node represents a quadrant. The root of a quadtree represents the entirety of the simulation space while the leaf nodes represent a sub-block that does not divide further (Huo et al., 2019).

The primary benefit of utilising a quadtree within hierarchical force calculation algorithms is the significant reduction in computational complexity from $O(N^2)$ to O(N log N) (Murtagh, 1988). Through the use of approximations, hierarchical methods can significantly reduce the computational complexity of an N-Body method in comparison to the naive approach.

The way that a hierarchical algorithm determines if an interaction is ‘long-range’ and can therefore be approximated, differs from algorithm to algorithm. Two of the most well-known hierarchical N-body algorithms are the ‘Barnes-Hut’ algorithm and the ‘Fast-Multipole Method’.

1.3.1 Barnes-Hut Algorithm

In 1986, Josh Barnes and Piet Hut proposed what would become known as the ‘Barnes-Hut’ algorithm. The proposed algorithm is a hierarchical force-calculation algorithm with time complexity O(N log N) (Barnes and Hut, 1986). The algorithm’s execution time represented a significant advancement at the time as up until this point, “the N-Body problem has been modelled numerically using two fundamentally different ways” which both had flaws (Barnes and Hut, 1986). The core innovations of the ‘Barnes-Hut’ algorithm are how the algorithm handles the subdivisions of space and how the algorithm computes 'long-range' interactions.

The algorithm is comprised of two phases. In the first phase, the tree structure is built and populated with the bodies in the system. Each node in the quadtree contains the total mass and the position of the centre of mass of all the particles in the subtree under it (Blelloch and Narlikar, 1997).

In the algorithm's second phase, each body's force is calculated by traversing the tree from the root, assessing whether interactions with nodes are ‘long-range’ using the Barnes-Hut criteria (Blelloch and Narlikar, 1997; Barnes and Hut, 1986). An interaction is deemed ‘long range’ if the ratio of the nodes side length, l, and the distance, D, from the current node’s centre of mass to the current body, is less than θ, where θ is a fixed parameter. If $l/D < \theta$ then the interaction should be approximated. If $l/D > \theta$ then the children of the current node are recursively examined. Since the first introduction of the ‘Barnes-Hut’ algorithm there have been many updates and improvements to the method. In Barnes’ and Hut’s (1986) own article they suggest methods of improvement such as “using a higher order integration scheme”, “including quadrupole moments” or “introducing individual time steps for particles which undergo strongly changing interactions”.

In recent literature, methods have been developed to improve the ‘Barnes-Hut’ algorithm through the use of parallelisation. Nakasato’s (2012) parallel implementation focuses on the optimisation of how the quadtree is constructed, how the nodes in the tree are accessed and how the forces acting on each particle are calculated. One optimisation that Nakasato (2012) uses in his implementation is localised particle ordering. This involves ensuring that particles that are spatially close together in the quadtree, are stored close to one another in memory to improve memory efficiency and reduce cache misses (Nakasato’s ,2012). Nakasato (2012) is able to show that his version of a parallel ‘Barnes-Hut’’ algorithm performs significantly faster than the brute force method for all values of N.

In their paper, Pearce et al. (2014) introduced an improvement to the parallel ‘Barnes-Hut’’ algorithm, in the form of a new method of load balancing. Rather than using the particle count for load balancing, the paper performs load balancing based on the interactions between particles as this is the real computational workload, especially for simulations with a non-uniform distribution of bodies (Pearce et al., 2014). By utilising an adaptive sampling technique to balance the number of interactions evenly across processors, the approach is able to see a “26% improvement in overall performance” (Pearce et al., 2014). However, an issue with this work is its specific tie to N-Body simulations with a highly non-uniform particle distribution. For N-Body simulations with a uniform particle distribution, this method results in little performance increase and, as such, the optimisation is not applicable to all instances where the ‘Barnes-Hut’ algorithm may be used.

The ‘Barnes-Hut’ algorithm represents a significant improvement in computational efficiency over the naive approach to N-Body simulations. This reduction in time complexity to O(N long N) allows for modelling large simulations that were previously computationally infeasible. Despite this, the ‘Barnes-Hut’ algorithm may not be appropriate for all simulation types. If a simulation requires an exceptionally high degree of accuracy in its results, then the ‘Barnes-Hut’ method may not be appropriate due to its use of approximation to speed up the computational complexity. In this instance, it may be appropriate to utilise the naive direct approximation method, but the ‘Barnes-Hut’ algorithm is appropriate for most cases.

1.3.2 Fast-Multipole Method

The ‘Fast-Multipole Method’ (FMM) was first introduced by Leslie Greengard and Vladimir Rokhlin (1987). This hierarchical force calculation executes in O(N) time (Greengard and Rokhlin, 1987) which is a significant improvement of the previously discussed naive algorithm and ‘Barnes-Hut’ algorithm. Since the FMM is a hierarchical method, the algorithm functions similarly to the ‘Barnes-Hut’ algorithm in the sense that a quadtree data structure is used to recursively decompose the simulation space. Where the algorithms differ, and how FMM is able to achieve a complexity of O(N), is through FMM’s use of multipole expansions to calculate the forces acting on each body in the system (Blelloch and Narlikar, 1997).

A *multipole expansion* is a series expansion of a function (Brandt, 2022). In the context of FMM, each node in the quadtree stores a truncated multipole expansion series (Board and Schulten, 2000) which expresses the mass distribution of all a node’s children (Brandt, 2022). This allows the calculation of a ‘long-range’ interaction between a particle and a group of distant particles by using a single multipole expansion that represents the group, rather than having to calculate the interactions separately for all the particles in the distant group (Board and Schulten, 2000). The expansions can be evaluated to give the gravitational potential and therefore the gravitational force acting on each particle (Greengard and Rokhlin, 1987).

The algorithm begins by constructing a quadtree for the simulation space (Greengard and Rokhlin, 1987). Once the quadtree has been populated, the algorithm computes the multipole expansions of the leaf nodes. (Blelloch and Narlikar, 1997). These expansions are then recursively ‘shifted up’ and added with neighbouring expansions in order to construct the multipole expansions of parent nodes (Blelloch and Narlikar, 1997).

Once the root of the quadtree is reached, the next phase of the algorithm begins. A depth-first traversal is initiated to calculate the local expansions (‘long-range’ interactions) (Blelloch and Narlikar, 1997). The local expansion for the root node is calculated and then shifted down recursively to the children and added to the multipole expansions of the nodes within the interaction set (Blelloch and Narlikar, 1997). The *interaction set* for a body **B** is the set of bodies in the simulation which are ‘long-range’ interactions with **B**; the particles in the interaction set are far enough away from **B** that the effects from the bodies in the interaction set can be approximated but are not so far away from **B** so that they can be ignored (Greengard and Rokhlin, 1987).

In the algorithm's final phase, the force contributions of nodes which are direct neighbours of the current body in the quadtree are computing directly using the naive O(N^2) method because the bodies in these nodes are too close to the current body to be approximated.

Since the creation of the FMM method there have been many papers which contribute to it. One such recent paper is that of Dehnen (2014). In his paper, Dehnen (2014) details several optimisations for the FMM method that aim to improve the computational efficiency and the accuracy of the algorithm. One improvement suggested is the introduction of an optimised expansion order for the multipole expansions. By using an optimised value, Dehnen (2014) is able to find an optimal trade-off between the computational costs of the expansions and the accuracy of the results. Dehnen's (2014) optimisations increase the efficiency of the FMM method and make it an even more powerful tool for large N-body simulations. An issue with Dehnen's (2014) work is the potential for overfitting with his optimised values; there is a risk that the optimisations Dehnen suggested do not generalise well to other N-body simulations.

In another paper, López-Fernández et al. (2016) describe an optimisation to determine the optimal size (D) of the edge length of the cubes used in the quadtree that store the bodies in the simulation. López-Fernández et al. (2016) propose a method that dynamically determines the optimal value of D which reduces the computational time the most without sacrificing accuracy. López-Fernández et al. (2016) test their method for choosing D across two different FMM frameworks and provide data to show how the computation time is improved with optimised D values. A critique of López-Fernández et al. (2016) is that they do not address the potential overhead introduced when dynamically determining D and discuss how this could affect the time complexity of the FMM.

The FMM method offers a significant leap in time complexity to O(N), compared to the naive approach and 'Barnes-Hut' algorithm. Like the 'Barnes-Hut' algorithm, the FMM utilises a hierarchical approach to achieve this, in addition to the inclusion of the calculation of multipole expansions. However, as with the 'Barnes-Hut' algorithm, because of the use of approximations in the FMM, it may be necessary to utilise the naive approach for N-body simulations that require a high degree of accuracy in their results.

1.4 'Mesh' Based N-Body Algorithms

'Mesh' based N-Body methods are a classification of N-Body algorithms where the core concept is to use a spatial 'mesh' to efficiently calculate the forces acting on each particle

(Bagla, 2004). These methods are able to reduce the time complexity for a simulation from $O(N^2)$ to $O(N \log N)$ by mapping the mass of each particle in the simulation onto the mesh points in the mesh, and then using Poisson's equation to calculate the gravitational potential at each point with Fast Fourier Transforms (Bagla, 2004). Once the gravitational potential is calculated, the gravitational force can then be calculated at each mesh point and be interpolated back to the positions of the actual particles in the simulation. Subsequently, the updated velocities and positions of the particles can be updated.

The method described previously is known as the 'Particle-Mesh' method and was first introduced by Hockney & Eastwood (1989) in their book called '*Computer Simulation Using Particles*'. The 'Particle-Mesh' method serves as a suitable improvement to the naive N-body algorithm as it reduces the time complexity and produces accurate simulations for long-ranged interactions. As noted by Hockney & Eastwood (1989), there is an issue with the 'Particle-Mesh' method due to how it defines the smallest scale of interactions that can be resolved, which can lead to the method being unable to accurately resolve short-range interactions.

Hockney & Eastwood (1989) also proposed a solution to the issues with the 'Particle-Mesh' method: the 'Particle-Particle-Particle-Mesh' method (P^3M method). The 'P^3M' method combines the naive direct summation method with the 'Particle-Mesh' method in order to utilise the advantages of the two different algorithms. This is achieved by utilising the direct summation method for short-range interactions and using the 'Particle-Mesh' method for the long-range interactions.

In recent literature, improvements have been made to optimise the 'P^3M' method. In their 2013 journal article, Harnois-Déraps et al provide a high performance, publicly available implementation of 'P^3M' called 'CUBEP^{3}M'. The 'CUBEP^{3}M' code offers significant scalability, having been ran on 'over 27000 cores' (Harnois-Déraps et al., 2013), making it an important tool for large scale N-Body simulations. In addition to this, the 'CUBEP^{3}M' code is extremely memory efficient as it allocates only 37 bytes per particle in the simulation, which is approximately half of what other N-Body simulations use (Harnois-Déraps et al., 2013).

'Mesh' based algorithms mark a significant advancement in N-Body simulations as they are able to reduce the time complexity of the simulation to $O(N \log N)$, from $O(N^2)$. 'Mesh' based algorithms offer a different approach to hierarchical methods, which use tree structures to achieve their reduced time complexity. Even though methods such as the 'Fast Multipole Method' have a lower time complexity, 'mesh' based methods are particularly useful for modelling simulations with predominately long ranged interactions, offering a balance between computational complexity and accuracy.

1.5 Background Research Findings

The background research conducted has resulted in three methods being chosen: the naive approach, the ‘Barnes-Hut’ algorithm and the Fast Multipole Method (FMM).

Despite having a time complexity of O(N^2), it is clear to see that the naive method remains a relevant tool in the simulation of N-Body models. This is especially the case with smaller scale simulations where accuracy is paramount. It may also be interesting to test the trade-off of execution time vs accuracy that occurs when using the naive method vs more advanced methods, in addition to seeing what performance increases can be achieved.

The next algorithm chosen to implement is the widely adopted ‘Barnes-Hut’ algorithm which is prominent in larger scale simulations due to its hierarchical nature and use of approximations to reach a time complexity of O(N log N). The other hierarchical N-Body method researched, the FMM, will also be implemented due to its remarkable efficiency of O(N) through its use of multipole expansions. Two hierarchical methods were chosen as they are both of the same ‘classification’ of N-Body simulation so it may be interesting to test the similarities/differences of the two algorithms.

It was decided not to implement any ‘mesh’ based N-Body methods. While these methods are good at modelling long ranged interactions and run at an efficient time complexity of O(N log N), we have opted not to include them in tests/optimisations due to the complexity involved with implementing a spatial mesh and utilised Fast Fourier Transforms, which is required for implementing ‘mesh’ based methods.

We have opted to utilise C++ to produce the N-Body software. C++ provides high performance, which will not affect the runtime measurements when performing tests. C++ also allows the efficient utilisation of parallel programming techniques through libraries such as OpenMP, which makes it ideal for this software. In contrast to this, if Python was selected as the language of choice, parallel programming could not be implemented for optimisations due to Python’s Global Interpreter Lock.

Chapter 2 - Methods

2.1 Initial Project Decisions

For the development of this software, the waterfall methodology was chosen over other popular software approaches, such as the agile methodology. The reason for this is due to the project's clearly defined requirements prior to development. Approaches such as the agile methodology take an iterative sprint approach, where at the end of each sprint, user feedback/technical evaluation is gathered to redefine the software requirements. Given that the requirements of this software are well defined and predetermined (to produce a comparison, evaluation, and optimisation of N-Body methods), it would be inappropriate to utilise an iterative software methodology. The waterfall methodology allows for stage-by-stage development, which the nature of the project lends itself to.

The project will utilise 'Simple DirectMedia Layer' (SDL) for producing a simple GUI for the software. SDL has been chosen as it works natively with C++ and is a well-established, efficient library for managing graphics. SDL also offers low overhead, which is essential for this project, in order to test the real time complexity and execution time of N-Body methods without the results being affected by other factors.

Version control will also be utilised throughout this project in order to manage the completion of different stages. This will be done in the form of a GitHub repository which can be found at the following link: https://github.com/uol-feps-soc-comp3931-2324-classroom/final-year-project-RyanWalsh18

2.2 Requirements Analysis

2.2.1 Stakeholders

This project does not follow a traditional user centric model as the aims of this project are to test the performance of different algorithms. However, one such stakeholder group are Peer Researchers in the field of computational physics, as they have a significant interest in the efficiency and accuracy of N-Body simulations. This means that the project should not only deliver optimised algorithms for performance, but it should also ensure that the implemented algorithms are validated against benchmarks to ensure their correctness. This is important as Peer Researchers expect findings that can be replicated, scrutinised, and expanded upon.

Another potential stakeholder is future developers. N-Body methods are typically difficult to implement so a well-documented, modular implementation with comments and clear explanations about optimisations and underlying mathematics will benefit developers trying to build on this work.

2.2.2 Functional Requirements

A functional requirement is a task that this piece of software must be capable of completing. After considering the scope of the project and the stakeholders involved, these are the functional requirements of the software:

- It must implement several N-Body algorithms: the naive method, the ‘Barnes-Hut’ method, and the ‘Fast Multipole’ method
- It must accept user inputs that specify the initial conditions for simulations
- It must be able to run an N-Body simulation based on the initial conditions provided
- It must produce a visualisation of the simulation
- It must keep track and report performance metrics such as execution time and memory usage

2.2.3 Non-Functional Requirements

A non-functional requirement specifies the performance characteristics and constraints for a piece of software. After considering the scope of the project and the stakeholders involved, these non-functional requirements of the software:

- It should produce simulation results that are within an acceptable error margin, as determined by established bench marks
- It should be well documented, modular, and easy for future researchers/developers to extend
- It should utilise the command line to receive input parameters from the user
- It should produce a visualisation that it clear and easy for the user to interpret
- It should output performance metrics in a clear and presentable way, such as through the terminal or as a text file

2.3 System Design

2.3.1 System Architecture Overview

The system architecture used for this project is a modular monolithic approach. This approach was chosen primarily because it combines the straightforward deployment of a monolith with the organized structure of modularity. The modular monolithic architecture allows for components to be situated in distinct, well-defined modules, while maintaining the software as a single executable. This type of architecture is more appropriate over a distributed architecture, such as a microservice architecture, because while modularity is at the centre of the design behind a distributed architecture, it comes at the cost of the system being distributed. In the context of this project, it may harm the results produced by testing as it could potentially introduce latency; this would be avoided by adopting a modular monolithic approach. The modular monolithic architecture is also of benefit to one of the stakeholder groups for this project: Future developers. By utilising a modular approach, this stakeholder group will benefit by being able reuse and adapt the code produced as part of this project.

2.3.2 Component Design

The simulation software will be broken down into several components that perform separate tasks and interact with each other. Figure 1 is a proposed component diagram of the system:

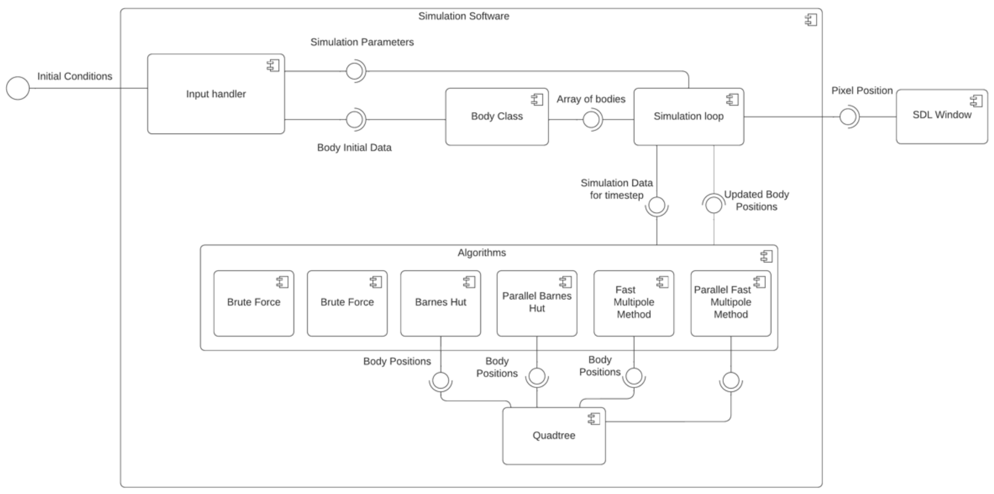


Figure 1: Component for proposed software

The ‘Input Handler’ component is responsible for parsing the input provided by the user on the command. The input handler will check for the type of algorithm to use and how many bodies to simulate, ensuring that it is a valid format. The ‘Body Class’ defines structure that represents the celestial bodies within the simulation. The ‘Simulation Loop’ is the main part of the software. It is responsible for managing each timestep of the simulation and ensuring that the positions/velocities of the bodies are updated. The ‘Algorithm’ component encapsulates the algorithms that can be used to simulate the N-Body problem. The simulation loop will execute the algorithm specified by the user input. The ‘Quadtree’ is a component that represents a quadtree data structure. This data structure is used as part of the ‘Barnes-Hut’ algorithm and the Fast Multipole method to store spatial data. The ‘SDL Window’ component is part of the SDL library and is used in this project to present a visual representation of the simulation. It receives the updated positions of bodies every timestep from the simulation loop and translates this data into pixel positions on the screen for rendering.

2.3.2 Class Design

The development of this project will utilise object orientated programming principles (OOP) to define several distinct classes. The utilisation of classes in this way contributes significantly to the modularity of this simulation software, which is aligned with the modular monolithic architecture chosen for this project. Figure 2 contains the class diagram designed for this system.

An instance of the ‘Body’ class is designed to represent a celestial body within the simulation. The class has private attributes such as mass, position (x,y), velocity (x,y) and acceleration (x,y) that can be accessed using the corresponding public get/set method. Encapsulation has been used here in order to control the access and modification of these attributes, which ensures these values are not mistakenly accessed. The class features several public constructors for initialising these attributes as either default or specified values. Lastly, the class features two operator overrides: the ‘==’ and ‘!=’ operators which have the purpose of allowing the comparison of the two body objects.

The ‘Simulation’ class is responsible for managing the data for the simulation. It has private attributes that track the properties of the simulation; as these attributes are private, there are public getter/setter methods for all of them. The class features a default constructor and a second constructor that initialises the previously mentioned private attributes. The remaining public methods are methods that run the simulation with the provided data for the current timestep, for example, the ‘runBarnesHuts’ method runs the ‘Barnes-Hut’ algorithm.

In order to run the two hierarchical algorithms, a quadtree implementation is needed. This is achieved with the ‘Quad’ and ‘QuadTree’ classes. The ‘Quad’ class simply represents a quadrant of the simulation space. It has private members that store the boundary coordinates of the section, a constructor that initialises them and public getter methods for the private attributes. The ‘QuadTree’ class stores and manages the data for each node in the quadtree. The class is designed that so each ‘QuadTree’ instance represents a singular node in the quadtree. In order to facilitate this, the class has a private unique pointer attribute that points to the ‘Quad’ instance which the node represents. The ‘QuadTree’ class also has pointer attributes that point to other ‘QuadTree’ instances, in order to store the parent and children on the current node, which will be used for tree traversal. The other private attributes store information about the node, such as if the node is the root of the tree, the level of the node in the tree and a vector array of the ‘Body’ instances that are stored in the node. The class features two public constructors: one for the root node of the tree and one for other nodes in the tree.

The methods of the ‘QuadTree’ class are used to perform actions on the quadtree structure. These methods include mechanisms to insert bodies into the quadtree structure and calculating the current forces acting on each ‘Body’ according to either the Fast Multipole Method or the ‘Barnes-Hut’ method. The class also includes private methods that subdivide the quadtree structure further when necessary, and helper methods for the Fast Multipole Method to aid with its implementation.

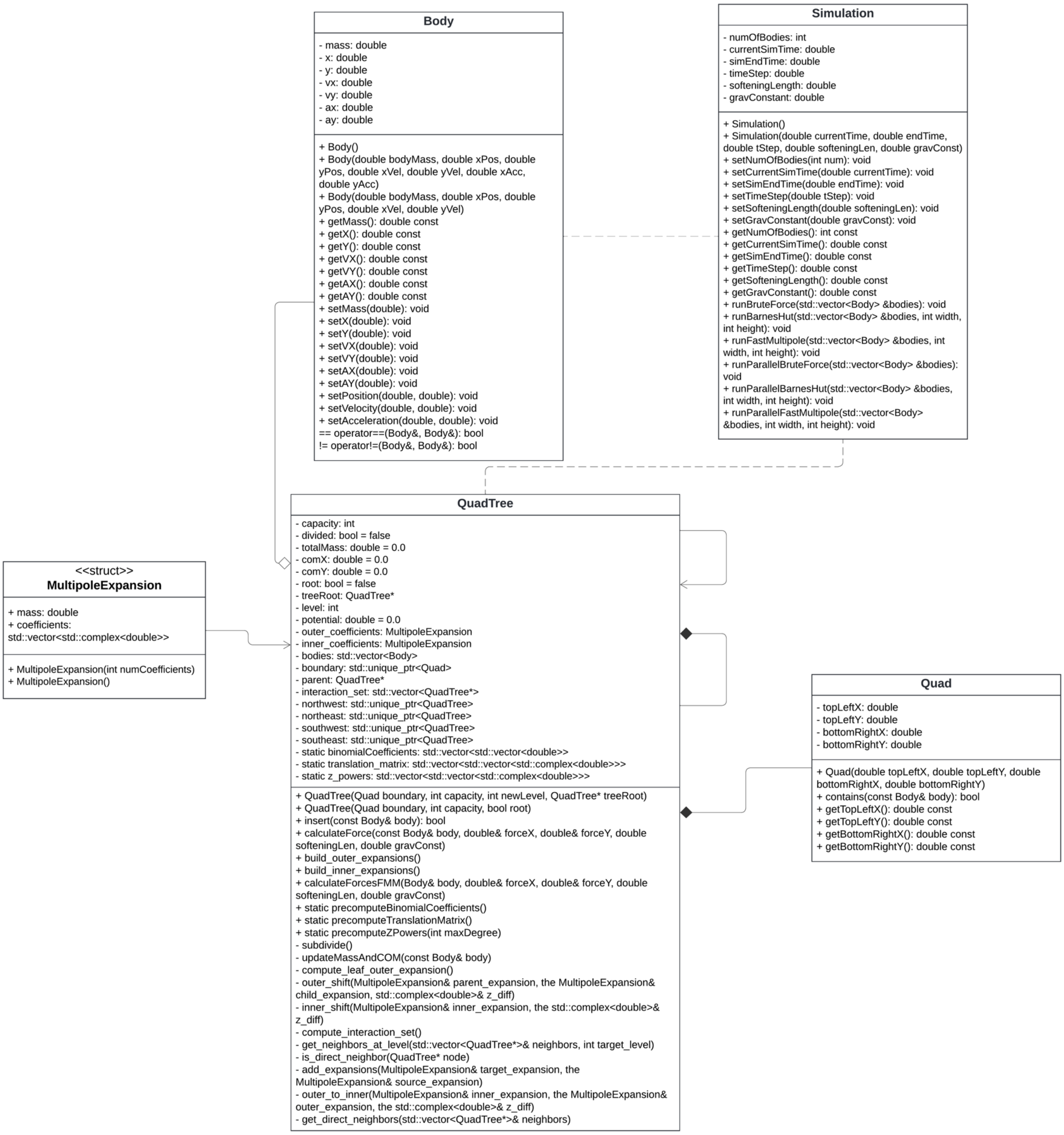

Body
- mass: double
- x: double
- y: double
- vx: double
- vy: double
- ax: double
- ay: double
+ Body()
+ Body(double bodyMass, double xPos, double yPos, double xVel, double yVel, double xAcc, double yAcc)
+ Body(double bodyMass, double xPos, double yPos, double xVel, double yVel)
+ getMass(): double const
+ getX(): double const
+ getY(): double const
+ getVX(): double const
+ getVY(): double const
+ getAX(): double const
+ getAY(): double const
+ setMass(double): void
+ setX(double): void
+ setY(double): void
+ setVX(double): void
+ setVY(double): void
+ setAX(double): void
+ setAY(double): void
+ setPosition(double, double): void
+ setVelocity(double, double): void
+ setAcceleration(double, double): void
== operator==(Body&, Body&): bool
!= operator!=(Body&, Body&): bool
Simulation
- numOfBodies: int
- currentSimTime: double
- simEndTime: double
- timeStep: double
- softeningLength: double
- gravConstant: double
+ Simulation()
+ Simulation(double currentTime, double endTime, double tStep, double softeningLen, double gravConst)
+ setNumOfBodies(int num): void
+ setCurrentSimTime(double currentTime): void
+ setSimEndTime(double endTime): void
+ setTimeStep(double tStep): void
+ setSofteningLength(double softeningLen): void
+ setGravConstant(double gravConst): void
+ getNumOfBodies(): int const
+ getCurrentSimTime(): double const
+ getSimEndTime(): double const
+ getTimeStep(): double const
+ getSofteningLength(): double const
+ getGravConstant(): double const
+ runBruteForce(std::vector<Body> &bodies): void
+ runBarnesHut(std::vector<Body> &bodies, int width, int height): void
+ runFastMultipole(std::vector<Body> &bodies, int width, int height): void
+ runParallelBruteForce(std::vector<Body> &bodies): void
+ runParallelBarnesHut(std::vector<Body> &bodies, int width, int height): void
+ runParallelFastMultipole(std::vector<Body> &bodies, int width, int height): void
<<struct>>
MultipoleExpansion
+ mass: double
+ coefficients: std::vector<std::complex<double>>
+ MultipoleExpansion(int numCoefficients)
+ MultipoleExpansion()
QuadTree
- capacity: int
- divided: bool = false
- totalMass: double = 0.0
- comX: double = 0.0
- comY: double = 0.0
- root: bool = false
- treeRoot: QuadTree*
- level: int
- potential: double = 0.0
- outer_coefficients: MultipoleExpansion
- inner_coefficients: MultipoleExpansion
- bodies: std::vector<Body>
- boundary: std::unique_ptr<Quad>
- parent: QuadTree*
- interaction_set: std::vector<QuadTree*>
- northwest: std::unique_ptr<QuadTree>
- northeast: std::unique_ptr<QuadTree>
- southwest: std::unique_ptr<QuadTree>
- southeast: std::unique_ptr<QuadTree>
- static binomialCoefficients: std::vector<std::vector<double>>
- static translation_matrix: std::vector<std::vector<std::complex<double>>>
- static z_powers: std::vector<std::vector<std::complex<double>>>
+ QuadTree(Quad boundary, int capacity, int newLevel, QuadTree* treeRoot)
+ QuadTree(Quad boundary, int capacity, bool root)
+ insert(const Body& body): bool
+ calculateForce(const Body& body, double& forceX, double& forceY, double softeningLen, double gravConst)
+ build_outer_expansions()
+ build_inner_expansions()
+ calculateForcesFMM(Body& body, double& forceX, double& forceY, double softeningLen, double gravConst)
+ static precomputeBinomialCoefficients()
+ static precomputeTranslationMatrix()
+ static precomputeZPowers(int maxDegree)
- subdivide()
- updateMassAndCOM(const Body& body)
- compute_leaf_outer_expansion()
- outer_shift(MultipoleExpansion& parent_expansion, the MultipoleExpansion& child_expansion, std::complex<double>& z_diff)
- inner_shift(MultipoleExpansion& inner_expansion, the std::complex<double>& z_diff)
- compute_interaction_set()
- get_neighbors_at_level(std::vector<QuadTree*>& neighbors, int target_level)
- is_direct_neighbor(QuadTree* node)
- add_expansions(MultipoleExpansion& target_expansion, the MultipoleExpansion& source_expansion)
- outer_to_inner(MultipoleExpansion& inner_expansion, the MultipoleExpansion& outer_expansion, the std::complex<double>& z_diff)
- get_direct_neighbors(std::vector<QuadTree*>& neighbors)
Quad
- topLeftX: double
- topLeftY: double
- bottomRightX: double
- bottomRightY: double
+ Quad(double topLeftX, double topLeftY, double bottomRightX, double bottomRightY)
+ contains(const Body& body): bool
+ getTopLeftX(): double const
+ getTopLeftY(): double const
+ getBottomRightX(): double const
+ getBottomRightY(): double const

Figure 2: Class diagram for proposed software

2.4 Testing and Validation Design

The goal of this section of the report is to detail the planned testing and validation for the software. The goal is to implement a multi-tiered testing strategy that ensures the simulation software is reliable, robust and produces accurate N-Body models. We plan on utilising ‘Catch2’, a popular testing framework for C++. ‘Catch2’ is a versatile library, easy to integrate and has a wide range of assertations, so is an appropriate choice of testing library for this project. Testing is particularly important given the stakeholders of this project: Peer researchers and future developers. It is important to ensure that this software is robust and reliable so that researchers are able to rely of the accuracy of the results, and future developers are able to build upon this work without being burdened by bugs.

2.4.1 Unit Testing

The first planned section of the testing strategy is unit testing, which will be used to test the core components of this system, such as the previously described classes. The benefit of utilising unit tests in this way is that they ensure that each component functions correctly in isolation and allow for confidence in the quality of the code in later stages of development.

2.4.2 Integration Testing

The next stage of the testing strategy is to perform integration testing. The utilisation of integration tests in this project ensures that components are working together cohesively. In the context of this project, an example of an integration test would be to ensure that ‘Body’ objects are correctly managed within the ‘QuadTree’ class/data structure.

2.4.3 Algorithmic Validation

The purpose of this phase is to rigorously validate that the implemented algorithms perform correctly. Since the goal of the project is to compare the implementation of various N-Body Methods, it is crucial to ensure that the results produced by these implementations are correct. This is so that we are able to accurately compare them. The correctness of these algorithms will be validated by comparing the output of the implementations against known results, sourced from established and peer-reviewed sources. For example, the implementation of the ‘Barnes-Hut’ algorithm may be cross referenced by comparing it to a known celestial movement, such as the orbit of the Moon around the Earth.

2.4.4 Comparison Testing and Metrics

As discussed previously, the goal of this project is to implement, optimise and compare the performance of N-Body methods. In order to perform these comparisons, there are several metrics that can be used in order to quantify the performance of each implementation. The execution time of an algorithm will provide insight into the scalability of the algorithms as the number of bodies increase. The accuracy of the algorithms can be evaluated by using the root-mean-squared error (RSME) to measure the deviation from a known solution. The RSME is a good choice of metric in this case as it can be used to quantify the deviation of the coordinates produced by the N-Body implementations from the coordinates produced by an analytical solution.

Chapter 3 - Results

3.1 Implementation

3.1.1 Body Class

The first element of this solution to be implemented was ‘Body’ class. The reason for implementing this first is that the ‘Body’ class is a fundamental component that all of the other components in the system will be built to interact with.

The class is declared in the ‘body.h’ file and defined in the ‘body.cpp’ file. As per the class diagram, the ‘Body’ class stores several attributes that contain vital information about the specific instance of a ‘Body’. The class stores the mass, ‘x’ & ‘y’ position, ‘x’ & ‘y’ velocity, and ‘x’ & ‘y’ acceleration as doubles, in order to account for the precision needed for an N-Body simulation. The class features a default constructor and a parameterised one; the latter allows for the initialisation of ‘Body’ objects with specific property values and is the constructor that will be utilised to create specific cases of gravitational bodies for simulating. In this implementation, getter and setter methods have been included. These methods allow other components in the simulation to safely access/update the attributes of a ‘Body’ object as needed during the course of the simulation.

Two overloaded operators have been implemented as part of the ‘Body’ class: “==” and “!=”. These comparison operators make it possible to check whether two instance of the ‘Body’ class are identical or if they differ. The reason for this implementation is for use with the N-Body methods in order to check that when calculating the forces acting on a body, the current body does not include itself in the calculation.

This implementation of the ‘Body’ class provides a simple and clear abstraction of a celestial body that can be used to effectively manage the properties of each body in the system.

3.1.2 Simulation Class

The next element to be implemented for the N-Body simulation is the ‘Simulation’ class. This class is implemented by following the design outlined in the previously discussed class diagram; this is achieved by declaring the class in the ‘simulation.h’ header file and defining it in the ‘simulation.cpp’ file.

The ‘Simulation’ class features several private attributes to store the parameters for the simulation. An integer data type is used to store the number of bodies in the simulation while doubles are used to store the following: the current simulation time, the end time of the simulation, the time step, the softening length, and the gravitational constant. The softening length is a small value that is added to the distance between bodies, in order to prevent the distance becoming zero and the gravitational force between two bodies becoming infinite. Since the attributes of the class are private, the ‘Simulation’ class features implemented getter/setter methods to enable encapsulation.

Besides the default and parametrised versions of the class’s constructor methods, the ‘Simulation’ class features six other public methods. These methods refer to the six N-Body algorithms implemented in this software: the Brute Force algorithm, ‘Barnes-Hut’ algorithm and the Fast Multipole Method, in addition to their optimised versions. The simulation class is designed so that with each timestep of the simulation, the relevant function for the N-Body method is called, in order to calculate the current forces acting on the bodies at that moment. In order to facilitate this, each method function accepts a vector containing all of the ‘Body’ objects in the system. The vector is passed by reference in order to modify the velocity and position of the actual ‘Body’ objects. For the hierarchical methods, these functions also accept the dimensions of the simulation space as a parameter in order to construct the quad tree data structure used during these methods.

3.1.3 Input Handling

The decision was made during the implementation of this software for it to be launched via the command line with command line arguments used to configure the system. By utilising command line arguments in this way, the N-Body program is easily integrated into testing frameworks and automated systems, which is of benefit to the stakeholder groups. The software requires one mandatory command line argument, with an optional second. The first argument specifies the N-Body method to be used for this instance of the simulation. The second optional argument specifies the number of bodies in the simulation. If the second argument is not provided, then the simulation defaults to a two simulation of a stable orbit.

```
Usage: ./simulation brute-force | barnes-hut | fast-multipole | parallel-brute-force | parallel-
barnes-hut | parallel-fast-multipole [numbers-of-bodies-simulated]
```

Figure 3: Command line usage of software

The handling of inputs and the simulation loop is implemented within the ‘main.cpp’ file. At the start of the main function, the program checks the arguments are valid and if not, a usage message is returned informing the user of the correct command line format. A helper function called ‘simMode’ is used to translate the first argument (a string input of the N-Body method) into an integer that represents the method that the simulation is to utilise. The ‘simMode’ function achieves this by performing conditional checks that compare the user input to the recognised simulation methods. If a match is found then the corresponding integer is set in the ‘mode’ global variable, otherwise the usage message is returned to prevent runtime errors.

If the second command line argument is provided, which specifies the number of bodies to be simulated, then the software converts this from a string to an integer using the standard C++ ‘atoi’ function which is used for this conversion due to its robustness and simplicity. If a valid integer is returned by ‘atoi’ then the global ‘defaultBodies’ flag is set to ‘false’, and it indicates that the number of bodies to simulate in the simulation has been specified. A usage message is returned if ‘atoi’ indicates an invalid integer has been provided. The usage message is shown in figure 3.

3.1.4 SDL Window, Body, and Simulation Initialisation

Once the input handling has concluded, the setup for the visualisation and the simulation begins. Key variables are defined such as ‘running’, a Boolean flag that controls the main simulation loop; ‘HEIGHT’ and ‘WIDTH’ which store the dimensions of the SDL window and auxiliary variables ‘X_MID’, ‘Y_MID’, ‘X_BOUND’, and ‘Y_BOUND’ which are used for ensuring that the bodies modelled in the simulation are placed within the bounds of the SDL window.

An SDL window and renderer object are created, based on the previously declared dimensions, using the ‘SDL_CreateWindowAndRenderer’ function. In this implementation, gravitational bodies have been abstracted to individual pixels, so to improve the clarity of the visualisation, the scale of the renderer is doubled using the ‘SDL_RenderSetScale’ function which results in the size of the drawn points being doubled.

The vector ‘bodies’ is then declared in order to store all of the body objects that are to be created for the simulation. A check is performed on the ‘defaultBodies’ flag to determine if the number of bodies to be simulated was provided by the user of the software. If the flag is ‘true’ then two ‘Body’ objects are created with pre-set attributes that result in a two-body simulation of a stable orbit (figure 4). If the flag is ‘false’, a grid distribution is calculated, and the number of bodies specified by the user are evenly distributed within the bounds of the SDL window. The use of a grid distribution (figure 5), rather than random distribution, is to ensure that the initial positions of bodies in the simulation is repeatable, allowing for the collection of accurate and robust data for algorithmic comparisons. Once the bodies have been created, for both cases, the ‘Body’ objects are added to the ‘bodies’ vector array.

Figure 4: Stable Two-Body orbit

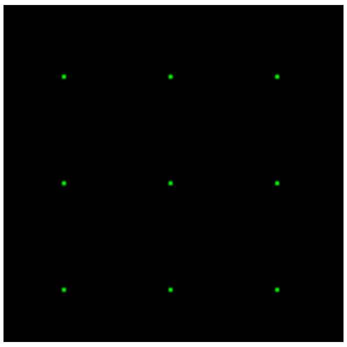


Figure 5: Grid distribution of bodies

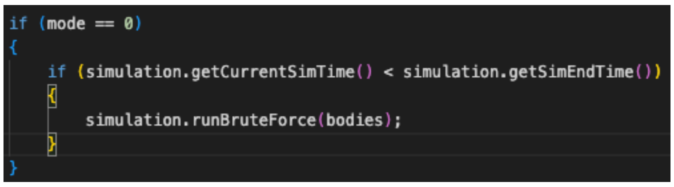
Once the ‘Body’ objects are defined, an instance of the ‘Simulation’ class is created. In this implementation, the ‘Simulation’ class is defined with hard coded values for the simulation properties such as end time’ and ‘time step’. The ‘setNumOfBodies’ method is then invoked so that the simulation object is aware of how many ‘Body’ objects it will be managing.

3.1.5 Simulation Loop

The main simulation loop is responsible for continuously handling events, updating the simulation state and rendering the pixels in the SDL window until the program is closed. The simulation loop is implemented with a ‘while’ loop that runs continuously while the ‘running’ flag is ‘true’. Every loop iteration, events are polled for. Event handling is handled through the use of the ‘SDL_PollEvent’ function. The only event that this function is polling for in this implementation is to check if the SDL window is closed. If this is so, then ‘running’ is set to ‘false’ and the simulation software exits gracefully.

After the event handling, the rendering of ‘Body’ objects follows. The render is cleared with a call to the ‘SDL_RenderClear’ function, to ensure that bodies from the previous frame do not remain on the screen. Once the window is cleared, the ‘bodies’ vector is iterated over, and each body is drawn to the screen using the ‘SDL_RenderDrawPoint’ function.

Several conditional blocks follow the rendering of the ‘Body’ objects. These condition blocks are responsible for checking the global variable ‘mode’ in order to call the correct function of the ‘Simulation’ object that corresponds to the matching N-Body method. Once the correct block is identified, a check is performed to check the simulation end time has not been reached and if so, the N-Body method is called (figure 6).



```
if (mode == 0)
{
    if (simulation.getCurrentSimTime() < simulation.getSimEndTime())
    {
        simulation.runBruteForce(bodies);
    }
}
```

Figure 6: Conditional check for N-Body method to run

3.1.6 Brute Force Algorithm

As discussed previously, the ‘Brute Force’ solution to the N-Body problem involves directly calculating the force exerted in each body by every other body in the simulation. This algorithm has been implemented within the ‘runBruteForce’ method of the ‘Simulation’ class; a call to the ‘runBruteForce’ method computes the forces acting on all the bodies, calculates the new velocity and updates the position of each body for the given time step.

The implementation utilises a nested for loop structure, where ‘i’ and ‘j’ are counter variables, to iterate other every pair of bodies where ‘i’ does not equal ‘j’. The reason for this condition is to prevent a body from calculating the force exerted on itself, by itself. For each unique pair, the function computes the ‘x’ and ‘y’ component of the distance between the two bodies. Using this, the magnitude of the distance is calculated and subsequently, the gravitational force is calculated according to Newton’s law of universal gravitation. Once the force magnitude is calculated, the ‘x’ and ‘y’ components of the force is calculated, followed by the new velocity of the body and the new position of the body. Once all the forces have been calculated, the ‘currentSimTime’ attribute is incremented at the end of the function in order to progress the simulation.

3.1.6.1 Parallelisation and Optimisations

Two optimisations have been implemented for this N-Body method. These come in the form of utilising OpenMP directive “#pragma omp parallel for” to parallelise two sections of the algorithm: the force calculation loop and the position update loop. The aim of these optimisations is to distribute the computational load across multiple cores, and thus reduce the execution time of the algorithm by allowing for work to done concurrently.

3.1.7 Quad and QuadTree Class

The ‘Quad’ and the ‘QuadTree’ classes are, as discussed, this software’s implementation of a quadtree data structure. Both of these classes have been declared in the ‘QuadTree.h’ file and defined in the ‘QuadTree.cpp’ file. The ‘Quad’ class represents a spatial region of the simulation space. The class feature double attributes that store the dimensions of the area that the ‘Quad’ instance is representing and has getter and setter methods to retrieve/alter these attributes.

The ‘QuadTree’ class is a more complex class; each instance of the ‘QuadTree’ class represents a node in a quadtree data structure. It is for this reason that the ‘QuadTree’ class has been implemented as a recursive data structure, where each ‘QuadTree’ instance is able to point to up to 4 other ‘QuadTree’ instances. A ‘QuadTree’ instance that acts as the root of the tree can be defined by using a constructor that takes the following parameters: a ‘Quad’ class representing the whole simulation space, the capacity of nodes in the tree (the maximum number of bodies allowed to be store in a node) and a Boolean set to ‘true’ that identifies the current node as the root of the tree. ‘Body’ objects can be inserted into this quadtree structure by invoking the ‘insert’ method and passing a ‘Body’ instance as a parameter.

The ability to insert into a quadtree has been implemented as follows. The ‘insert’ function receives a reference to the ‘Body’ object that is to be inserted into the quadtree and inserts it if the ‘Body’ is in the region covered by the quad tree. If the current node is a leaf node and the capacity of the node is not exceeded, then the ‘Body’ is simply inserted into this node and ‘true’ is returned. ‘QuadTree’ nodes are able to store what ‘Body’ objects are in them through the use of a vector array attribute. If the capacity of the current node is exceeded and the current node is a leaf node, then the current node in subdivided. This is achieved by calling the private method ‘subdivide’, which creates four pointers to new ‘QuadTree’ objects. These four new objects are children of the current node and represent quadrants of the parent’s space. The ‘Body’ objects in the current node are then redistributed across the child nodes, in addition to the ‘Body’ currently being inserted. If the current node is not a leaf node, then the ‘Body’ object being inserted is recursively passed to the children of the current node.

In addition to the insert method, the ‘QuadTree’ implementation features several other public methods. The ‘calculateForce’ method calculates the gravitational force acting on a ‘Body’ according to the ‘Barnes-Hut’ algorithm. The ‘calculateForcesFMM’, ‘build_outer_expansions’ and ‘build_inner_expansions’ methods of the ‘QuadTree’ class are all implemented similarly to the ‘calculateForce’ method, but they compute components used in the execution of the Fast Multipole Method. In addition to these public methods, the ‘QuadTree’ class feature private helper functions to add with the implementation of the Fast Multipole Method.

3.1.8 Barnes-Hut Algorithm

The ‘Barnes-Hut’ algorithm implementation is located in the ‘runBarnesHut’ method of the ‘Simulation’ class. The implementation begins by defining a ‘QuadTree’ instance that represents the entire simulation space and, utilising a loop, the ‘Body’ objects in the vector array ‘bodies’ (the array is passed by reference to the method) are inserted into the quadtree data structure. With the quadtree constructed, the force acting on each body is calculated. This implemented with another for loop in order to loop through each ‘Body’ in the simulation. With each iteration of the loop, two doubles are defined to store the ‘X’ and ‘Y’ components of the gravitational force. The current ‘Body’ and the force variables are passed to the previously discussed ‘calculateForce’ method of the ‘QuadTree’ class, which traverses the quadtree structure and calculates the components of the gravitational force acting on the current ‘Body’. Once the force components are calculated, they are used to calculate the new position of the current body. Once all the ‘Body’ objects have been looped through, the simulation is progressed.

3.1.8.1 Parallelisation and Optimisations

The ‘Barnes-Hut’ algorithm has been parallelised again by utilising OpenMP. The loop responsible for the calculation and update of the ‘Body’ objects’ new positions has been parallelised using the “#pragma omp parallel for” directive. This parallelisation spreads the load of updating the new positions of the ‘Body’ objects in order to improve performance.

Attempts were made to implement a further optimisation, namely an optimisation of inserting ‘Body’ objects into quadtree structure. This approach was designed to construct ‘sub’ quadtrees concurrently and then merge these together into the main quad tree structure. Issues were faced with the implementation of this approach, primarily surrounding data dependencies, and avoiding race conditions so this was ultimately not implemented.

3.1.9 Fast-Multipole Algorithm

The final N-Body method implemented is the Fast Multipole Method; this was accomplished in the ‘runFastMultipoleMethod’ method of the ‘Simulation’ class. Similarly to the ‘Barnes-Hut’ algorithm, this implementation starts with the definition of a ‘QuadTree’ instance that represents the entirety of the simulation space and the ‘QuadTree’ object is then populated with the ‘Body’ objects in the simulation. This is followed by calls to the ‘build_outer_expansions’ and the ‘build_inner_expansions’ functions to calculate the outer and inner expansions respectively, of each node in the quadtree. Each ‘quadTree’ instance has two private instances of the ‘MultipoleExpanison’ struct in order to store the outer and inner expansions.

The ‘build_outer_expansions’ method is responsible for computing the ‘outer expansions’ for each node in the quadtree. These expansions are calculated from the leaves of the tree and shifted up to the parent node, until the root of the tree is reached. The ‘build_inner_expansions’ method is responsible for computing the ‘inner expansions’ of the quad tree node. This implementation calculates the ‘inner expansion’ of each node by initialising the ‘inner expansion’ of the root node to zero, then performing a preorder traversal to shift the ‘inner expansion’ to the leaf nodes. As the nodes are traversed, the ‘interaction set’ is computed for each node. For each node in the interaction set, the ‘outer expansion’ of the node is converted into an ‘inner expansion’ using the ‘outer_to_inner’ helper function and is added to the current node’s ‘inner expansion’. The purpose of this is to account for gravitational potential that is from nodes close to the current node, but still outside the current node. When the traversal reaches the leaf nodes of the tree, the ‘inner expansions’ for the leaf nodes are directly computed by using the eight nodes that are direct neighbours of the current leaf node. The final step of this implementation is looping through the ‘Body’ objects and evaluating the ‘inner expansions’ in order to calculate the gravitational force acting on the current ‘Body’, so that the updated position of the body can be calculated.

3.1.9.1 Parallelisation and Optimisations

Several optimisations have been implemented for the Fast Multipole Method, in addition to parallelising the force update loop. Beside this, three optimisations have been implemented: the precomputation of the binomial coefficients, the translation matrix, and the powers of complex numbers. The precomputation of these things reduces redundant calculations to improve the efficiency of the Fast Multipole Method. If they were not precomputed, these computations would have to be recomputed for every node in the system. If the system is a large system of tens of thousands of ‘Body’ objects, then this would be a lot of computation time wasted.

3.1.10 Software Compilation

The compilation of this software is handled by a ‘Makefile’. The ‘Makefile’ utilses the ‘clang’ compiler as the software was developed on ‘MacOS’ in order to compile and link all of the source files used in this project. The ‘Makefile’ is responsible for the creation of three executables. The first is the ‘simulation’ executable which is the executable for the N-Body algorithms and simulation software. A second executable is the ‘tests’ executable. This executable is responsible for linking together all of the source files that perform unit testing and integration testing. The final executable is the ‘generate_accuracy_data’ executable. The executable is responsible for autonomously running simulation models in order to produce data that is used for the comparison of the implementations.

3.2 Validation and Testing

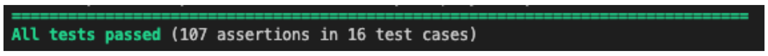
3.2.1 Unit Testing

The unit tests implemented as part of this testing strategy are designed to isolate and test the most basic functionality of the classes in this software. These unit tests ensure the robustness of each component, so that the components can be relied upon to produce accurate results. The unit tests for each class are spread across separate sources files and the different unit tests are implemented as ‘Catch2’ ‘TEST_CASE’ macros.

The unit tests for this ‘Body’ verify the correct implementation of the class’s constructor methods and setter/getter methods, which ensure that properties such as the mass, position and velocity are modified and retrieved correctly. The tests also include unit tests that validate the correctness of the ‘==’ and ‘!=’ overload comparison operators. For all of these unit tests, edge cases such as negative and extreme values are included in order to ensure that these scenarios can be handled without error.

The unit tests for the ‘Quad’ and ‘QuadTree’ class ensure that both of these classes are initialised correctly by their own constructors, in addition to checking the getter/setter methods. Unit tests have also been implemented to validate that calls to the ‘insert’, ‘calculateForce’ and ‘calculateForceFMM’ functions are handled correctly. These tests only verify that the calls to these two functions are handled correctly. The correctness of the actual calculations is checked within the algorithmic validation section of the testing strategy.

The final class that unit tests have been produced for is the 'Simulation' class. As with the other classes, unit tests have been produced to verify the correctness of the constructors, getters, and setters. Unit tests have also been implemented which ensure that calls to the six N-Body method functions do not result in exceptions being thrown. The actual correctness of the results produced by the function is tested in the algorithmic validation section of the testing strategy. All of the unit tests that have been implemented for this testing strategy pass as expected (figure 7)

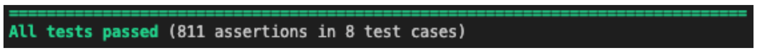


```
===============================================================================
All tests passed (107 assertions in 16 test cases)
```

Figure 7: Unit testing results

3.2.2 Integration Testing

Integration testing has been implemented as the next stage of this testing strategy in order to validate that the various components in this software work together correctly. The testing for this section begins with testing the integration of the 'Body' class and the 'QuadTree' class in order to ensure that 'Body' objects are inserted into the correct node of the 'QuadTree' instance. Integration tests have also been utilised to test the integration of the 'Body' class with the 'Simulation' class by ensuring that 'Body' objects can be created and passed to one of the six N-Body methods which are methods of the 'Simulation' class. The integration tests for this ensure that all the 'Body' objects that are passed to the functions are updated by the 'Simulation' object. All of the integration tests that have been implemented for this testing strategy pass as expected. (figure 8)



```
===============================================================================
All tests passed (811 assertions in 8 test cases)
```

Figure 8: Integration testing results

3.2.3 Algorithmic Validation

In order to validate the correctness of N-Body method implementations, comparisons to a known solution were utilised. In order to source a known solution, it was decided to calculate an analytical solution to the Two-Body problem. This method was used over the initial decision in the design of the testing strategy to use a known solution from other researchers due to being unable to find a solution we thought was appropriate for this software. Kepler's laws of planetary motion were utilised in order to calculate a stable circular orbit for a Two-Body system and then set the software to model this orbit. All of the N-Body Methods implemented were able to model the orbit given the same parameters as the analytical solution. The implemented methods were then used to simulate an N-Body problem and provide visual confirmation that they were correct. The ‘Brute Force’ and ‘Barnes-Hut’ algorithms were correct, however an issue was discovered with the Fast Multipole method. This bug appears part way through the simulation and results in a portion of the ‘Body’ objects having their positions being calculated as ‘NaN’. This bug was thought to be a result of division by zero or an error when calculating the multipole expansions, however this bug could not be fixed and is an area for future work. It is important to ensure that this bug is considered when evaluating and comparing this implementation of the Fast Multipole Method with the other N-Body methods, as the results may be affected.

3.3 Comparative Performance Analysis of N-Body Methods

In order to compare the scalability of the serial and parallelised N-Body methods, the execution times of the algorithms was measured while increasing the number of bodies, **N**, in the system. The execution time of each algorithm was measured using the ‘chrono’ C++ library and tests were executed three times in order to calculate the average execution time.

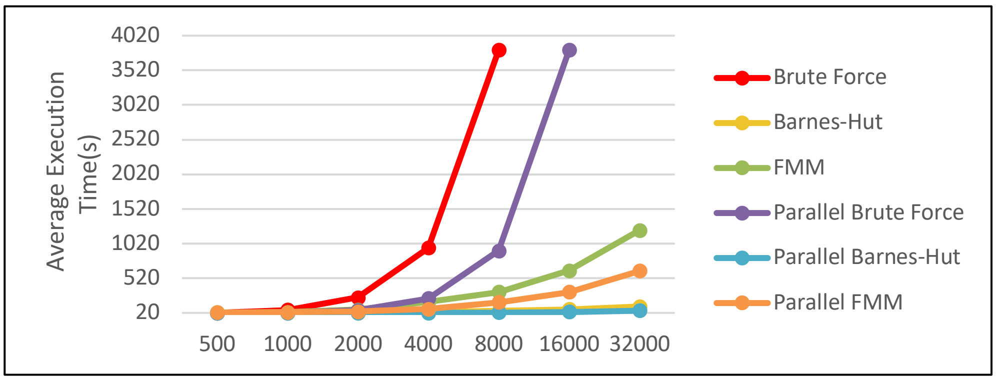


Figure 9: Line chart plotting average execution time vs number of bodies being simulated

Figure 9 shows that the implemented parallel techniques and optimisations had the intended effect of improving the scalability of each of the N-Body algorithms. As expected, the ‘Brute Force’ method generates the steepest curve, showing its poor scalability due to its O(n^2) time complexity. The parallelised version of the ‘Brute Force’ method offers an improvement to its serial version, however the scalability is still poor, especially when compared to the hierarchical methods.

The much-improved efficiency of the ‘Barnes Hut’ and Fast Multipole Method (FMM) can also be seen from figure 9. While this is not necessarily surprising, due to their superior time complexities, what is surprising is that FMM was outperformed by the ‘Barnes Hut’ algorithm. It could be possible that the previously discussed bug in the implementation of the FMM is at fault for this. It is probably more likely however, that the complexity and resultant computational overhead of the calculations in the FMM is not offset by the small numbers of **N** used in testing. This suggests that the ‘Barnes Hut’ algorithm is the best N-Body algorithm in terms of scalability, but future work is needed to compare this method and the Fast Multipole method with larger values of N.

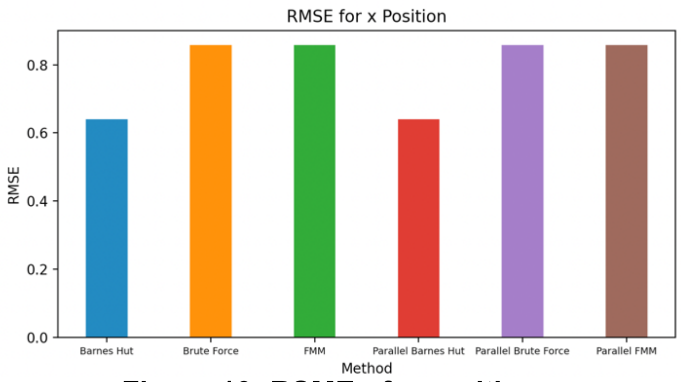


Figure 10: RSME of x position

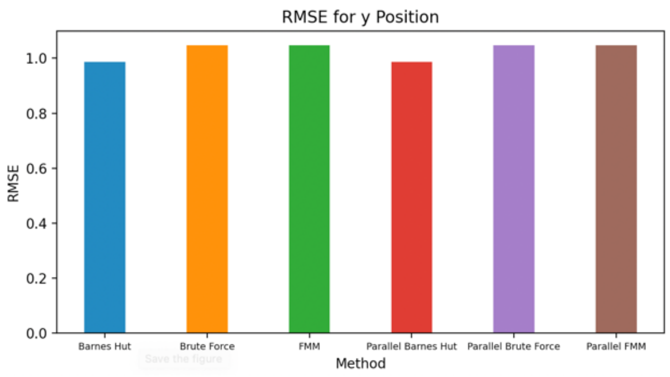


Figure 11: RSME of y position

The accuracy of the N-Body methods was evaluated by comparing the positions of a body at each time step while using the different methods, to the positions of a body during a known analytical solution. The known solution used in this case was the previously discussed Two-Body system that models a stable orbit. A C++ file was written to generate this benchmark data and the data for the N-Body methods. The data was then written to individual CSV files and a python script was then created to read in the data, calculate the Root Mean Squared Error (RSME) for the x and y positions and use the ‘matplotlib’ library to produce bar charts for this data.

Figure 10 and Figure 11 indicate that there is no difference between the serial and parallel versions of the N-Body methods. Interestingly, the ‘Barnes-Hut’ algorithm was the method with the lowest RSME, despite using it approximations. The most surprising thing from these results is the fact that the Brute Force method has higher RSME than the ‘Barnes Hut’ algorithm. A potential reason for this is due to the integration method used for updating the positions of the bodies introducing errors. Based on these results, it is clear to see that the factors which affect the accuracy of N-Body methods is an area for future research.

Chapter 4 - Discussion

4.1 Conclusions

Through thorough testing, a comprehensive comparison and evaluation of different N-Body methods has been achieved. The results from this testing show a clear difference in the execution time of the methods as the number of bodies, **N**, in the simulation scales. Unsurprisingly, the Brute Force algorithm scaled the worse due to its poor O(N^2) efficiency, while the two hierarchical methods offered significant improvement, through their use of spatial decomposition and the quadtree data structure. It is interesting to note that the 'Barnes-Hut' algorithm performed better than the Fast Multipole Method (FMM), despite having a higher theoretical time complexity. This is likely due to the overhead associated with computing the multipole expansions used in the FMM, meaning that the benefits of the FMM were not realised with the values of **N** which the testing was conducted with. As a result of these findings, it is a critic of the testing that the values of **N** used were not large enough to show the true performance of the FMM. The parallel implementations of these algorithms reflected improvements in the execution times, however the degree of the improvement varied between algorithms. This suggests that while the parallel computation techniques offered improvement, the complexity of each algorithm is the overriding factor in the execution time of each algorithm.

The other aspect of testing was to compare the accuracies of each algorithm by using the Root Mean Squared Error (RSME) as a metric to compare the implementations to a known analytical solution. The most surprising conclusion from this aspect of testing was that despite its use of approximations, the 'Barnes-Hut' algorithm produced more accurate results than the Brute Force method, which directly computes all the forces in the system. A possible explanation of this is that the integration method used to update the positions of the bodies in the simulation is introducing errors into the produced results. It is a critique of this evaluation that the integration method was not accounted for or acknowledged during the development of the N-Body software.

It is important to acknowledge the bug in the FMM that causes a proportion of the bodies' positions to be evaluated as 'NaN'. It appears that bug is potentially stemming from the one of two places: An accumulation of error when calculating the multipole expansions or a 'division by zero' when handling cases where bodies are extremely close to each other. This bug undermines the confidence in the accuracy of the FMM method implementation, so it is important that this is addressed.

4.2 Ideas for Future Work

As mentioned throughout the implementation of this project, there are several potential options for areas of future work. The most pressing of these is the previously discussed bug in the implementation of the Fast Multipole Method (FMM). As part of this future work, it would be crucial to perform a thorough review of the implementation of the FMM. The focus of this work would be on ensuring the correctness of the complex calculations used in the FMM, in addition to improving the method’s handling of extreme cases. This future work would aim to benefit both stakeholder groups of this project improving the robustness and accuracy of the FMM implementation.

Another area of future work is the implementations of more advanced parallel techniques and optimisations. The inclusion of these could potentially lead to even more reductions in computation time by improving the scalability of the N-Body implementations. Techniques that could be explored would be utilising GPU technology, parallel load balancing and techniques for parallelising the methods of the ‘QuadTree’ class such as inserting into the data structure.

The final area of future work to be discussed for this project would be the expansion of the tests used for the comparison of the N-Body methods. Firstly, the scope of the testing would be increased to include larger numbers of bodies that are being simulated. This would allow for the ‘Barnes-hut’ and FMM to be compared by how they handle significant values of **N**.

4.3 Summary

In summary, this project has met its objectives of implementing, parallelising, and evaluating the performance of N-Body methods. Quantitative data has been produced in the form of execution time and accuracy measurements, which have allowed for a detailed comparison of the performance of the different N-Body methods. The results from these comparisons have highlighted the trade-offs between accuracy and speed in the simulation of the N-Body problem. This project lays a foundation for exploring more advanced parallel optimisation techniques that can benefit whole sectors of the scientific community.

List of References

Bagla, J. 2004. Cosmological N-Body simulation: Techniques, Scope and Status. *Current Science*. **88**.

Barnes, J. and Hut, P. 1986. A hierarchical O(N log N) force-calculation algorithm. *Nature*. **324**(6096), pp.446–449.

Blelloch, G. and Narlikar, G. 1997. A Practical Comparison of N-Body Algorithms. *Parallel Algorithms*. **30.**

Board, J. and Schulten, L. 2000. The fast multipole algorithm. *Computing in Science & Engineering*. **2**(1), pp.76–79.

Brandt, A. 2022. [Pre-Print]. On Distributed Gravitational N-Body Simulations.

Britannica. No Date. *The n-body problem.* [Online]. [Accessed 14 January 2024]. Avaiable from: https://www.britannica.com/science/Hamiltonian-function

Capuzzo-Dolcetta, R., Spera, M. and Punzo, D. 2013. A fully parallel, high precision, *N*-body code running on hybrid computing platforms. *Journal of Computational Physics*. **236**, pp.580–593.

Chen, J. 2021. Understanding and Analyzing the Performance Scalability of N-body Problem *In*: *2021 4th International Conference on Computer Science and Software Engineering (CSSE 2021)* [Online]. Singapore Singapore: ACM, pp.212–215. [Accessed 27 March 2024]. Available from: https://dl.acm.org/doi/10.1145/3494885.3494924.

Dehnen, W. 2014. A fast multipole method for stellar dynamics. *Computational Astrophysics and Cosmology*. **1**(1), p.1.

Greengard, L. and Rokhlin, V. 1987. A fast algorithm for particle simulations. *Journal of Computational Physics*. **73**(2), pp.325–348.

Harnois-Déraps, J., Pen, U.-L., Iliev, I.T., Merz, H., Emberson, J.D. and Desjacques, V. 2013. High-performance P3M N-body code: cubep3m. *Monthly Notices of the Royal Astronomical Society*. **436**(1), pp.540–559.

Hockney, R.W. and Eastwood, J.W. (1989) *Computer simulation using particles*. Bristol: Hilger.

Huo, S.H., Li, Y.S., Duan, S.Y., Han, X. and Liu, G.R. 2019. Novel quadtree algorithm for adaptive analysis based on cell-based smoothed finite element method. *Engineering Analysis with Boundary Elements*. **106**, pp.541–554.

López-Fernández, J.A., López-Portugués, M. and Ranilla, J. (2016) 'Improving the FMM performance using optimal group size on Heterogeneous System Architectures', The Journal of Supercomputing, **73**(1).

Murtagh, F.D. 1988. Hierarchical trees in *N*-body simulations: Relations with cluster analysis methods. *Computer Physics Communications*. **52**(1), pp.15–18.

Nakasato, N. 2012. Implementation of a Parallel Tree Method on a GPU. *Journal of Computational Science*. **3**(3), pp.132–141.

Nyland, L., Harris, M. and Prins, J. 2009. Fast N-body simulation with CUDA. *GPU Gem, Vol. 3.*, pp.677–695.

Smith, G. 2007. *Newton's philosophiae naturalis principia mathematica.* [Online]. [Accessed 21 February 2024]. Available from: https://plato.stanford.edu/entries/newton-principia/#Bib

Appendix A - Self-appraisal

A.1 Critical self-evaluation

Reflecting on the project, I am happy with the overall execution and achievements of it, relative to the objectives and requirements first laid out. This project required a sophisticated understanding of complex mathematical and physical concepts, that at first, I struggled with. I was pushed to the edge of my technical capabilities but because of this, I was able to expand my knowledge of computational physics and algorithmic optimisations.

Having never undertaken a project of this size before, I initially struggled to develop a robust and actionable plan. However, after conducting background research, I gradually began to form an idea of what the objectives of this project would be.

During semester one, I felt happy with the progress I was making. I was successfully able to implement the brute force method and the visual window. As a result of this, I felt in control of the project and was considering things to add to my initial objectives.

It was in the second semester that I faced issues. I began to face development issues with the implementation of the quadtree data structure and the fast multipole method, as I encountered bugs that I struggled to fix. I feel like I did not create a robust enough plan that included mitigations for me to fall back on in case I lost valuable time due to things like this. It also did not help that I underestimated the time commitment that my other responsibilities required. Whilst I was successful in fixing the issue with the quadtree structure, I was unable to do so for the Fast Multipole Method. This is the thing that I am least satisfied with about my project and wish I could change the most.

A.2 Personal reflection and lessons learned

This project was an invaluable learning experience as it marks my first time tackling such a complex research project. Handling all the aspects of software development, from the initial design all the way through to the testing and evaluation has significant enhanced my technical abilities. In addition to this, writing a report in an academic context is something that I have never done before, and I am personally happy with how it turned out.

If I was to tackle the project again though, there are somethings that I would change due to the lessons that I have learnt. One of the key lessons that I learnt is the importance of realistic planning. I would aim to include realistic time frames for each section of development in order to ensure that my plan is feasible and suitable, given my other academic and personal commitments. As part of this, I would also include risk management, in order to ensure I am able to cope with any setbacks or delays.

In addition to this, another key lesson I learnt was the importance of continuous documentation during the development cycle. This change would aid significantly as it would help to reduce the burden of having to retrospectively produce the documentation.

A.3 Legal, social, ethical, and professional issues

A.3.1 Legal Issues

All libraries that were used in this project were fully licensed and used within the terms of their respective licences.

A.3.2 Social Issues

This project is confined to an academic setting with no direct social implications. The software that was developed was on used for the purpose of testing and evaluation. The software has not been distributed in any form.

A.3.3 Ethical Issues

This project features no ethical issues as it strictly handles theoretical and simulated data. No real individuals or sensitive information is used.

A.3.4 Professional Issues

Good professional practices were followed during the development of this project. Version control was used via GitHub and name conventions were followed throughout for things such as variables, functions.

Appendix B - External Materials

- SDL was used to handle the creation of the visualisation window
    - https://www.libsdl.org/
- OpenMP was used for parallel computing
    - https://www.openmp.org/
- Matplotlib was used to produce bar charts
    - https://matplotlib.org/
- Microsoft Excel was used to produce line charts
    - https://www.microsoft.com/en-gb/microsoft-365/excel
- Catch2 was used for a testing framework
    - https://github.com/catchorg/Catch2
- Pandas was used to read in CSV files
    - https://pandas.pydata.org/
- Sklearn was used to calculate Root Mean Squared Error
    - https://scikit-learn.org/stable/