Decoding Celestial Dynamics: A Python-based Exploration of N-Body Simulations

Wouter Dobbenberg s1722980

January 30, 2024

1 Introduction

Engaged in a university assignment, our attention shifts to celestial mechanics through a Python-based simulation. This educational exercise emphasizes proficiency in Python coding while unraveling the nuanced dynamics of gravitational interactions. Beyond that, we delve into performance optimizations, specifically with a focus on the Barnes-Hut method.

Gravity, a fundamental force in the cosmos, governs the movements between celestial bodies. To dissect these gravitational interactions within our simulated framework, celestial bodies are treated as particles. This approach enables us to utilize particle kinematics for precise calculations, allowing the application of integration methods, such as the velocity Verlet technique, to update the positions of these particles.

2 The N-Body Problem

The N-Body problem involves predicting the individual motions of a group of objects interacting with each other through gravity, with the primary objective being the accurate projection of their trajectories over time based on given initial conditions. The challenge arises when dealing with a substantial number of bodies, as computational efficiency diminishes exponentially. This study delves into the theoretical foundations and practical implementation of the N-Body problem. Furthermore, an enhancement, namely the Barnes-Hut Algorithm, will be introduced to address and significantly improve computational efficiency in the simulation.

3 Celestial Mechanics Theory

In the simulation of gravitational interactions among astronomical bodies, a fundamental approach involves modeling celestial bodies as particles. Each particle is characterized by its mass (m) and position ($\vec{x}$). These properties play a crucial role in clarifying the complex dynamics governed by gravity. Furthermore, the first and second-time derivatives of a particle's position yield its velocity ($\vec{v}$) and acceleration ($\vec{a}$), respectively. The quantitative representation of these particle properties is encapsulated by the following formulas (Equations 1 to 4).

This particle-centric perspective, where mass and position are central attributes, forms the basis for employing integration methods such as the Velocity Verlet technique to precisely compute the evolving trajectories of celestial bodies within the gravitational field.

$$mass := \{m\} \tag{1}$$
$$position := \{x_i, x_j, x_k, \} \tag{2}$$
$$velocity := \{v_i, v_j, v_k, \} \tag{3}$$
$$acceleration := \{a_i, a_j, a_k, \} \tag{4}$$

3.1 Gravitational Potential

To chart the trajectories of celestial bodies, it is imperative to deduce the gravitational force governing their motion. This derivation emanates from the formula for potential energy:

$$U = -\int_{ref}^{r} \vec{F} \cdot d\vec{r} \tag{5}$$
$$U(r_{ij}) = -\int_{\infty}^{r_{ij}} \frac{Gm_i m_j}{{r'_{ij}}^2} dr'_{ij} = -\frac{Gm_i m_j}{||r_{ij}||} \tag{6}$$

Expressed in vector notation, the gravitational force ($\vec{F}_{ij}$) acting between two particle masses m_i and m_j, separated by a vector $\vec{r}_{ij}$, can be written as:

$$\vec{F}_{ij} = -\frac{dU}{d\vec{r}_{ij}} = \frac{Gm_i m_j}{||r_{ij}||^2} \cdot \frac{\vec{r}_{ij}}{||r_{ij}||} \tag{7}$$

In this representation, $\vec{F}_{ij}$ denotes the gravitational force vector, $\vec{r}_{ij}$ represents the displacement vector pointing from m_i to m_j, $||\vec{r}_{ij}||$ stands for the magnitude of the displacement vector, and G is the gravitational constant.

This formulation comprehensively incorporates both the magnitude and direction of the gravitational force. The negative sign signifies the attractive nature of the force, indicating its alignment along the line joining the two masses. The term $\frac{\vec{r}_{ij}}{||\vec{r}_{ij}||}$ serves as a unit vector directed along $\vec{r}_{ij}$, ensuring the gravitational force is oriented along the line connecting the two masses.

3.2 Particle Kinematics: Timestep-based Updates

Integration is a fundamental aspect of simulating astronomical particles influenced by gravity, as it enables the prediction of their trajectories over time. Integration allows for the discretization of the equations of motion, which are often complex and nonlinear, into manageable time steps. By iteratively updating positions and velocities based on gravitational interactions, integration methods provide a computationally feasible approach to simulating the dynamic and interconnected behaviour of astronomical particles under the influence of gravity. This process is essential for simulating gravitational interactions that shape the structure and evolution of the cosmos.

3.2.1 Velocity Verlet Integration

The Velocity Verlet Integration method, exemplified by the equations:

$$\vec{x}_{i+1} = \vec{x}_i + \vec{v}_i \Delta t + \frac{1}{2}\vec{a}_i \Delta t^2 \tag{8}$$

$$\vec{x}_{i+1} = \vec{x}_i + (\frac{\vec{a}_{i-1} + \vec{a}_i}{2})\Delta t + \frac{1}{2}\vec{a}_i \Delta t^2 \tag{9}$$

is particularly advantageous in astronomical simulations for its notable speed and energy conservation properties. In contrast to methods like Newton-Euler, where position and velocity evaluations are separate, and Leap Frog, which can exhibit stability concerns, Velocity Verlet stands out as a simplistic integration scheme. Its inherent energy conservation is crucial for preventing the accumulation of numerical errors over prolonged simulation periods. By efficiently updating positions and velocities, as outlined in the equations, Velocity Verlet ensures accuracy in capturing the intricate dynamics of astronomical systems while maintaining computational efficiency.

In this project, a customised version of the Velocity Verlet method, referred to as the Direct Velocity Verlet method, has been implemented, building upon the previously discussed Velocity Verlet Integration equations. In contrast to the original Velocity Verlet method, this modification involves calculating the velocity by subtracting the position of the previous timestep from the current timestep's position. This adjustment results in a slightly simplified formula, and integration constants vanish due to alterations in boundary conditions. The updated equations for the Direct Velocity Verlet method are as follows:

$$\vec{v}_i = \vec{x}_i - \vec{x}_{i-1} \tag{10}$$

$$\vec{x}_{i+1} = \vec{x}_i + \vec{v}_i + \vec{a}_i \Delta t^2 \tag{11}$$

3.3 Enhancing Spatial Partitioning: Quad- and Oct-Trees

Spatial partitioning techniques, exemplified by Quad-Trees and Oct-Trees, play a crucial role in optimizing N-body simulations dealing with a multitude of mass particles. These hierarchical tree structures offer a systematic representation of space by dividing it into smaller regions. Initially, all particles are placed in the root node, and as the simulation progresses, the tree dynamically subdivides regions based on the spatial distribution of particles. Quad-trees, organizing space into quadrants, and Oct-Trees, extending this concept to octants, allow for a finer computation of gravitational interactions. This hierarchical organization proves beneficial in refining the calculation of forces, particularly when dealing with distant particles. The interactions between distant particles are approximated by treating the particles within a distant node as a single particle with a center of mass, thereby reducing computational costs and optimizing the simulation's efficiency. [2]

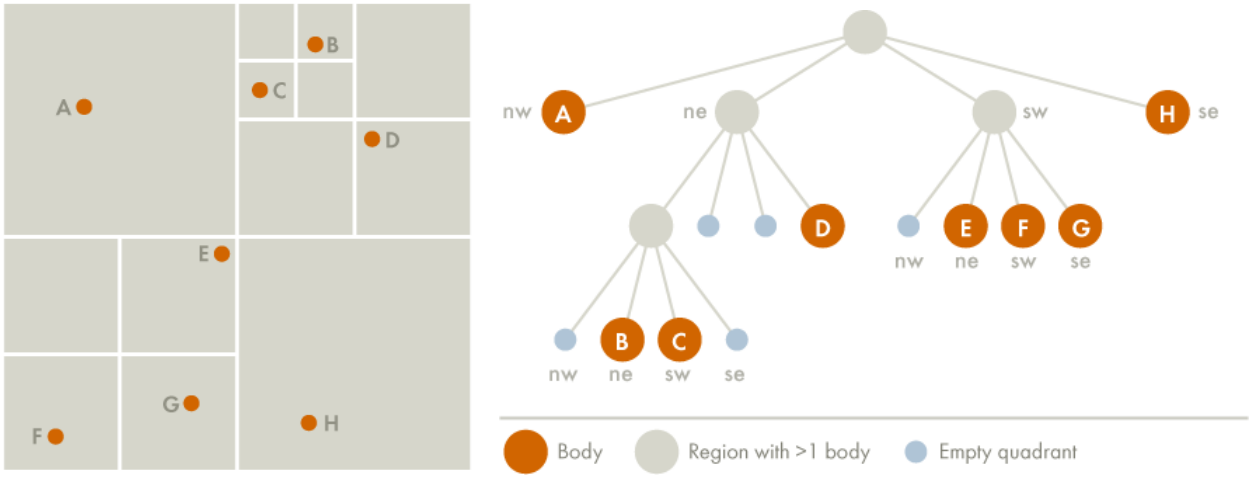

Figure 1: The Figure shows an example of spatial partitioning and related Quad-tree structure respectively. The Images have been sourced from [4]

3.4 Barnes-Hut Criteria for Accelerated Simulations

The Barnes-Hut optimization introduces specific criteria to accelerate N-body simulations, particularly in the context of gravitational interactions between mass particles. By leveraging the hierarchical structure of Quad-Trees or Oct-Trees, the Barnes-Hut algorithm strategically identifies nodes that are sufficiently distant from a target particle to be approximated. This is determined by evaluating the size-distance ratio $\frac{s}{d}$, where s is the size of the region (width or diameter) and d is the distance between the center of mass of the region and the terget particle. The Barnes-Hut criterion states that if $\frac{s}{d} < \theta$, the region is considered "sufficiently far away" from the target particle, allowing the algorithm to streamline the simulation process. in such cases, the distant groups of particles within a node are treated as a collective entity with a centor of mass. This approximation significantly reduces the computational load, enhancing the overall speed and efficiency of the simulation. [1][4]

This performance improvement becomes particularly crucial when dealing with a large number of particles, allowing for accurate and expedited simulations of complex celestial systems. Adjusting the user-defined parameter θ provides a means to control the trade-off between accuracy and computational efficiency in the simulation.

4 Code: Python Implementation Insights

The forthcoming subsections unravel the construction of the Python code, showing the intricacies of the celestial mechanics simulation. It's crucial to highlight that the oct-tree data structure discussed in subsections 4.4 to 4.7 is adapted directly from a JavaScript quadtree developed by Daniel Shiffman [3]. Moreover, both the implementation of the velocity Verlet method and the integration of the Barnes-Hut algorithm draw directly from insights obtained from informative text-based webpages referenced earlier in Section 3.

It's essential to note that the provided code snippets, while serving as a foundational model, do not offer a one-to-one representation of the actual code. The presented snippets are intentionally simplified to enhance readability and comprehension. The complete implementation goes beyond this

foundational model, incorporating a more sophisticated file structure, facilitating the storage of celestial body positions in a vtu file, and supporting multiple integration methods for a comprehensive exploration of celestial dynamics.

4.1 Setting up a *"Point Mass"* class

In the context of the celestial mechanics simulation, the "**Point**" class plays a pivotal role as a representation of celestial bodies. This class encapsulates essential properties such as position, velocity, acceleration, gravitational force, and mass. Instances of the "**Point**" class serve as particles within the simulation framework, allowing for the modeling of gravitational interactions. The `__init__` method initializes a Point object with these properties, enabling the instantiation of individual celestial bodies with specific attributes. As we progress through the paper, you will observe how instances of the "**Point**" class become integral to the implementation of various simulation aspects, including the Velocity Verlet integration method and the subsequent optimizations such as spatial partitioning with Quad- and Oct-Trees, leading to an efficient simulation of celestial dynamics. This modular and object-oriented approach enhances code readability and facilitates a comprehensive understanding of the simulation's inner workings.

```
class Point:
    def __init__(self, pos, vel, acc, force, mass):
        self.pos = pos
        self.vel = vel
        self.acc = acc
        self.force = force
        self.mass = mass
```

4.2 Setting up a *"Simulation Box"* class

Introducing the "**Box**" class, this component defines the spatial constraints of the celestial mechanics simulation. Instantiated with a position (**pos**) and size (**size**), the "**Box**" class characterizes a confined region within the simulation. The "**centre**" attribute represents the center of the box, calculated as the sum of the position and half of the size. The "**Contains**" method determines whether a given "**Point**" object resides within the boundaries of the box, allowing spatial partitioning.

```
class Box:
    def __init__(self, pos, size):
        self.pos = pos
        self.size = size
        self.centre = pos + size/2

    def Contains(self, point):
        if ((point.pos > self.pos) and (point.pos <= self.pos + self.size)):
            return True
        return False
```

4.3 Setting up the interactions

In the context of the traditional N-Body problem, this code snippet orchestrates the pairwise calculation of gravitational forces between celestial bodies. Employing a nested loop over the list of celestial bodies (**List**), it iterates through each unique pair of bodies and calls the "**CalculateBodyForces**" function to compute the gravitational forces between them. This straightforward approach is effective for small to moderate numbers of particles, providing a clear representation of the gravitational interactions. However, it's important to note that as the number of particles increases, the computational demands of this structure grow exponentially, resulting in reduced efficiency.

In anticipation of addressing efficiency concerns associated with a larger number of particles, the interaction function will be redesigned from scratch. The forthcoming implementation will incorporate advanced techniques such as the Oct-Tree and Barnes-Hut algorithm to significantly improve the computational efficiency of gravitational force calculations in the celestial mechanics simulation.

```
for i, point1 in enumerate(List):
    for point2 in List[i+1:]:
        CalculateBodyForces(point1, point2)
```

4.4 Setting up a *"Data Structure"* class

Introducing the "**DataStruct**" class, this component is used as a data storage medium for the celestial bodies. Instantiated with a simulation box (**Box**), this class dynamically updates the center of mass (**massPos**) and accumulated mass (**accumMass**) as celestial bodies are stored into the "**points**" List. Specifically, "**massPos**" represents the weighted sum of the positions of all stored particles within the box, considering each particle's mass as its weight. Meanwhile, "**accumMass**" accumulates the total mass of all particles within the box.

```
class DataStruct:
    def __init__(self, box):
        self.points = []
        self.massPos = box.centre
        self.accumMass = 0

    def InsertPoint(self, point):
        if (not self.box.Contains(point)):
            return False
        else:
            self.points.append(point)
            self.massPos = (point.mass * point.pos + self.accumMass * ...
            → self.massPos) / (self.accumMass + point.mass)
            self.accumMass += point.mass
```

The "**InsertPoint**" method, a key function for the Oct-Tree structure, checks whether a given celestial body falls within the boundaries of the associated simulation box through its "**Contains**" function. If outside the box, it returns "**False**", indicating that the body cannot be inserted. If inside the box, the body is added to the points list, and the center of mass and accumulated mass are updated.

In the next section, the "**DataStruct**" class will be updated to allow the implementation of advanced techniques such as the Barnes-Hut algorithm.

4.5 Transforming the Data Structure into an Oct-Tree

To expand the capabilities of the "**DataStruct**" class, several attributes have been introduced to facilitate the transition to an Oct-Tree structure. The class now includes "**divided**" to indicate whether the current box has been subdivided, "**depth**" to track the depth of the tree, "**pointCap**" to set a limit on the number of points a box can hold before subdividing, and the list "**subTree**" to store the resulting sub-boxes upon division.

The "**InsertPoint**" method remains pivotal, dynamically updating the center of mass (**massPos**) and accumulated mass (**accumMass**) as celestial bodies are inserted into the data structure. Additionally, a call to the newly introduced "**InsertOrSubdivide**" function has been incorporated. This function, to be defined in the subsequent subsection, manages the decision-making process of either inserting a point into the current box or subdividing it further, based on defined criteria.

```
class DataStruct:
    def __init__(self, box, depth = 0, pointCap = 4 ):
        self.points = []
        self.massPos = box.centre
        self.accumMass = 0

        self.divided = False
        self.depth = depth
        self.pointCap = pointCap

        self.subTree = []

    def InsertPoint(self, point):
        if (not self.box.Contains(point)):
            return False
        else:
            self.massPos = (point.mass * point.pos + self.accumMass * ...
            → self.massPos) / (self.accumMass + point.mass)
            self.accumMass += point.mass
            InsertOrSubdivide(point) # call to insert a point or subdivide ...
            → the current box in the data structure
```

4.6 Managing Point Insertion and Subdivision in the Oct-Tree Structure

The "**InsertOrSubdivide**" function governs the decision-making process within our Oct-Tree structure, determining whether to insert a celestial body into the current box or to subdivide it for further refinement. The function begins by evaluating two conditions: if the current box has not been divided and the number of points within the box is below the specified capacity, or if the depth of the tree has reached a predetermined limit (in this case, 8). In either case, the celestial body is directly added to the current box (**points** list).

If the conditions for direct insertion are not met, and the box has not been divided yet, the function triggers subdivision using the "**Subdivide**" method and reallocates previously assigned points to the newly created sub-boxes using "**ReallocatePoints**".

Subsequently, the function recursively calls "**InsertPoint**" on each of the eight sub-boxes (**subTree**) to attempt the insertion of the celestial body. The logical OR operator is employed to ensure that the point is inserted into the first available sub-box that satisfies the conditions for insertion. This recursive process continues until the celestial body finds a suitable location within the Oct-Tree structure.

```
def InsertOrSubdivide(self, point):
    if (len(self.points) < self.pointCap and self.divided == False) or ...
    ↪ self.depth == 8:
        self.points.append(point)
    elif self.divided == False:
        self.Subdivide()
        self.ReallocatePoints()
    return (self.subTree[0].InsertPoint(point) or
            self.subTree[1].InsertPoint(point) or
            self.subTree[2].InsertPoint(point) or
            self.subTree[3].InsertPoint(point) or
            self.subTree[4].InsertPoint(point) or
            self.subTree[5].InsertPoint(point) or
            self.subTree[6].InsertPoint(point) or
            self.subTree[7].InsertPoint(point))
            # try and insert the point into the subTree recursively
```

4.7 Point Reallocation in the Oct-Tree Structure

The "**ReallocatePoints**" function mirrors the approach utilized in Section 4.6 "Managing Point Insertion and Subdivision in the Oct-Tree Structure". It serves as a vital element of the Oct-Tree structure, ensuring the smooth transition of celestial bodies to their appropriate sub-boxes following a parent box's subdivision. Similar to the logic in "**InsertOrSubdivide**", for each point stored in the parent box, the function iterates through the eight sub-boxes (**subTree**) and recursively attempts to insert the point into each sub-box using "**InsertPoint**". The logical OR operator guarantees that the point is allocated to the first available sub-box that meets the insertion criteria. This recursive reallocation process ensures that each celestial body finds its new residence within the subdivided structure.

```
def ReallocatePoints(self):
    for oldPoint in self.points:
        if (self.subTree[0].InsertPoint(oldPoint) or
            self.subTree[1].InsertPoint(oldPoint) or
            self.subTree[2].InsertPoint(oldPoint) or
            self.subTree[3].InsertPoint(oldPoint) or
            self.subTree[4].InsertPoint(oldPoint) or
            self.subTree[5].InsertPoint(oldPoint) or
            self.subTree[6].InsertPoint(oldPoint) or
            self.subTree[7].InsertPoint(oldPoint))
            # try and insert the oldPoint into the subTree recursively
            pass
        else:
            print("ERROR, old points haven't been reassigned")
```

4.8 Subdividing

The "**Subdivide**" method is a pivotal step in the Oct-Tree structure, facilitating the creation of eight sub-boxes from a single parent box. This method is instrumental in enhancing the adaptability and spatial resolution of our celestial mechanics simulation. Within a triple-nested loop iterating over the dimensions (x, y, z), the function dynamically generates eight new sub-boxes corresponding to the eight octants of the parent box and appends them to the "**subTree**" list. Each sub-box is instantiated as a new "**DataStruct**" object, corresponding to the newly defined position (**newPos**) and size (**newSize**). This recursive subdivision process ensures a hierarchical organisation of space, optimising the simulation's precision, particularly in densely packed regions, where a concentration of celestial bodies requires more detailed representation. The Oct-Tree's refined spatial partitioning contributes to higher accuracy in capturing intricate gravitational interactions within these concentrated areas.

```
def Subdivide(self):
    for x in range(2):
        for y in range(2):
            for z in range(2):
                self.subTree.append(DataStruct(Box(newPos, newSize, depth + 1)))
```

The "**newPos**" snippet calculates the position of each sub-box within the parent box. For each dimension (x, y, z), the position is adjusted based on the size of the parent box, effectively dividing it into two halves. This calculated position ensures that each sub-box is appropriately positioned within the hierarchical structure, creating a spatially organized hierarchy in our Oct-Tree.

The "**newSize**" snippet determines the size of each sub-box by halving the size of the parent box along each dimension. This halving process ensures that the sub-boxes are proportionally smaller, refining the spatial granularity and accommodating a more detailed distribution of celestial bodies within the Oct-Tree structure.

```
newPos = [(
            self.box.pos[0] + x * self.box.size[0]/2,
            self.box.pos[1] + y * self.box.size[1]/2,
            self.box.pos[2] + z * self.box.size[2]/2
         )]
```

```
newSize = [(
            self.box.size[0]/2,
            self.box.size[1]/2,
            self.box.size[2]/2
          )]
```

4.9 Barnes-Hut Method: Optimizing Gravitational Force Calculations

The "**CalculateBodyForcesInTree**" function introduces a strategic enhancement to our gravitational force calculations, serving as an alternative to the previously mentioned "**CalculateBodyForces**" function in Section 4.3. Unlike its counterpart, this function requires only a single celestial body point as input, as the other relevant points are already stored within the Oct-Tree data structure. This streamlined approach leverages the Barnes-Hut method, optimizing the efficiency of force calculations in the context of the N-Body simulation.

The function begins by calculating the distance between the center of mass (**massPos**) of the current Oct-Tree node and the position (**pos**) of the given celestial body point. The magnitude of this distance is then computed. Subsequently, the function compares the distance against the Barnes-Hut criteria (in this case, 0.5), determining whether the node's size is sufficiently large compared to the distance. If the criteria are met, the function recursively applies the Barnes-Hut method to nested nodes within the Oct-Tree structure.

When no more nested nodes exist, the function calculates the gravitational interaction forces between the given celestial body point and the stored points within the "**subtree**" list. The gravitational interaction force is determined using the Equation 7 mentioned in Section 3.1. Notably, a minimum and maximum magnitude are enforced to prevent numerical instability caused by the gravitational singularity inherent in point masses. If the Barnes-Hut criteria are not triggered, the function calculates the interaction forces with the accumulated mass of the tree node, further optimizing the simulation's computational efficiency.

```
def CalculateBodyForcesInTree(self,point):
    distance = self.massPos - point.pos
    magnitude = numpy.sqrt(distance.dot(distance))

    # Comparing the distance against the barnes hut criteria
    if (((self.box.size[0] + self.box.size[1] + self.box.size[2])/3) ...
    → /magnitude) >= 0.5:
            # repeat criteria for nested nodes
            for child in self.subTree:
                child.CalculateBodyForcesInTree(point)

            # if no more nested nodes exist, calculate the interaction forces
            # against the points stored in the subtree
            if (not self.subTree):
                for treePoint in self.points:
                    if point != treePoint:
                        distance = treePoint.pos - point.pos
                        magnitude = numpy.sqrt(distance.dot(distance))
                        magnitude = max(min(magnitude,50),5)
                        bodyForces = (GRAV_CONST * point.mass * ...
                        → treePoint.mass * distance)/(numpy.power(magnitude,3))
                        point.force += bodyForces

    # if the criteria is not triggered,
    # Calculate the interaction forces with the accumulated mass of the tree node
    else:
        bodyForces = (GRAV_CONST * point.mass * self.childMass * ...
        → distance)/(numpy.power(magnitude,3))
        point.force += bodyForces
```

4.10 Enhancing the Point Class for Direct Velocity Verlet Integration

Expanding upon the previously introduced "**Point**" class in Section 4.1, we enrich its capabilities to accommodate the "Direct Velocity Verlet" method discussed in Section 3.2.1. This extension is essential for calculating the trajectories of particles or celestial bodies within our gravitational simulation. The "**DirectVelocityVerlet**" function, along with the inclusion of "**oldPos**" and "**firstTimeStep**" variables, forms a critical component of the position update process.

The "**DirectVelocityVerlet**" method not only updates the particle position but also handles nuances in velocity calculations. specifically during the initial time step, ensuring a precise evolution of celestial body trajectories over time. The "**oldPos**" variable serves the crucial role of tracking the previous position, a necessity for the Direct Velocity Verlet technique. It allows for accurate velocity computations, integral to maintaining the fidelity of the simulation. Simultaneously, the "**firstTimeStep**" flag plays a vital role in addressing the absence of previous position data during the initial iteration. This boolean variable enables adjustments in velocity to align with the simulation's starting conditions, ensuring the accurate application of the Direct Velocity Verlet formula.

```
class Point:
    def __init__(self, pos, vel, acc, force, mass):
        self.pos = pos
        self.oldPos = pos
        self.vel = vel
        self.acc = acc
        self.force = force
        self.mass = mass
        self.firstTimeStep = True

    def DirectVelocityVerlet(self):
        if self.firstTimeStep == False:
            self.vel = self.pos - self.oldPos
        else:
            self.vel *= DELTA_TIME
            self.firstTimeStep = False
        self.oldPos = numpy.copy(self.pos)
        self.pos += self.vel + self.acc * settings.DELTA_TIME ** 2
```

4.11 Writing a "update" method

With all the essential classes and their corresponding functions in place, we can now construct the "**Update**" method that orchestrates the simulation. This method oversees the dynamic evolution of celestial bodies by generating a "**DataStruct**" Oct-Tree at each time step. The process begins by initializing the oct tree within the simulation box, setting its position to the center of the box and its size to encompass the entire simulation space.

The function then iterates through the list of celestial bodies, inserting each point into the "**DataStruct**" Oct-Tree. Once integrated into the oct tree, gravitational interaction forces are calculated for each celestial body based on the hierarchical structure of the tree. This is facilitated by the "**CalculateBodyForcesInTree**" function.

Subsequently, the positions of celestial bodies are updated using the "Direct Velocity Verlet" method through the "**DirectVelocityVerlet**" function for each point in the list. This method ensures the precise calculation of new positions, incorporating gravitational interactions into the trajectory of each celestial body.

To optimize memory usage and prepare for the next iteration, the created "**DataStruct**" Oct-Tree is deleted. The entire process is repeated for a user-defined number of iterations, allowing for the simulation to unfold over the specified time steps. This "**Update**" method serves as the core mechanism for advancing the simulation, capturing the dynamic interplay of gravitational forces, and revealing the evolving trajectories of celestial bodies within the simulated cosmos.

```
for i in range(1000):
    Update(List)
```

```
def Update(List):
    DataTree = DataStruct(Box( ...
        → pos = numpy.double([SIZE/2,SIZE/2,SIZE/2]), ...
        → size = numpy.double([SIZE,SIZE,SIZE])))
    for point in List:
        DataTree.insertPoint(point)
    for point in List:
        DataTree.CalculateBodyForcesInTree(point)
    for point in List:
        point.DiractVelocityVerlet()
    del DataTree
    return
```

5 Simulation Results

Utilizing the implemented code, two distinct simulation scenarios have been executed, providing dynamic visualizations of celestial interactions. The first simulation centers around a three-body problem, unveiling the synchronized orbits of two planets revolving around a central sun. This depiction captures the elegance of gravitational interactions in a stable celestial system. The initial conditions are shown in Table 1.

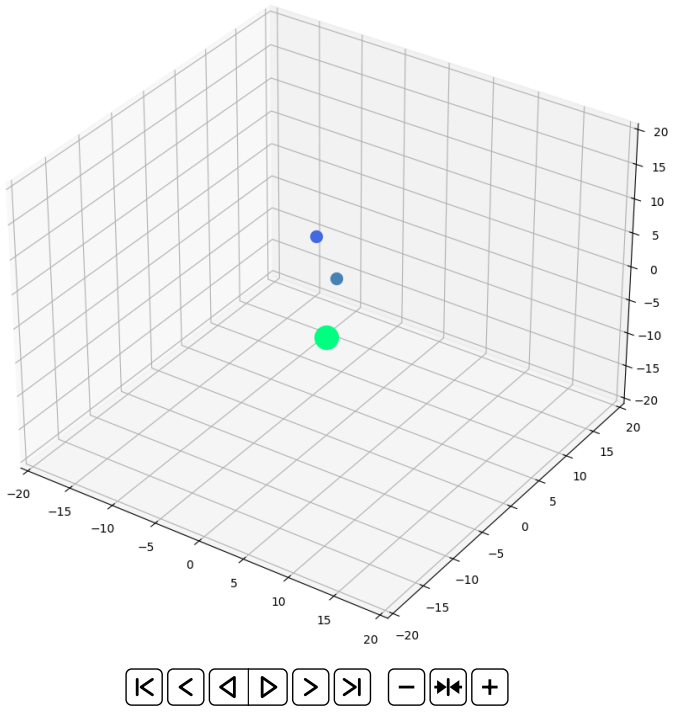

Figure 2: Animated png of the three body problem simulation within python. The animation is rendered within "matplotlib". If by any chance the animation doesn't play, consider using a different pdf reader such as adobe acrobat.

Body nr.	mass (kg)	$\vec{pos}$ (m)	$\vec{vel}$ $(\frac{m}{s})$	$\vec{acc}$ $(\frac{m}{s^2})$	$\vec{force}$ $(kg\frac{m}{s^2})$
1	$9e^9$	$[-10, 0, 0]$	$[0, -1, -2]$	$[0, 0, 0]$	$[0, 0, 0]$
2	$1.8e^{10}$	$[0, 0, 0]$	$[0, 0, 2]$	$[0, 0, 0]$	$[0, 0, 0]$
3	$9e^9$	$[10, 0, 0]$	$[0, 1, -2]$	$[0, 0, 0]$	$[0, 0, 0]$

Table 1: Three body simulation starting conditions.

In the second simulation, a more intricate scenario unfolds, featuring a diverse ensemble of 100 asteroids alongside two planets and a sun. The asteroids initiate their journey from random starting positions around the sun, each equipped with a random velocity perpendicular to the sun's direction. The other three bodies within the simulation bounds use the starting conditions shown in Table 3. The resulting animations, presented in Figure 2 and Figure 3, offer a captivating display of the evolving trajectories within this complex gravitational field.

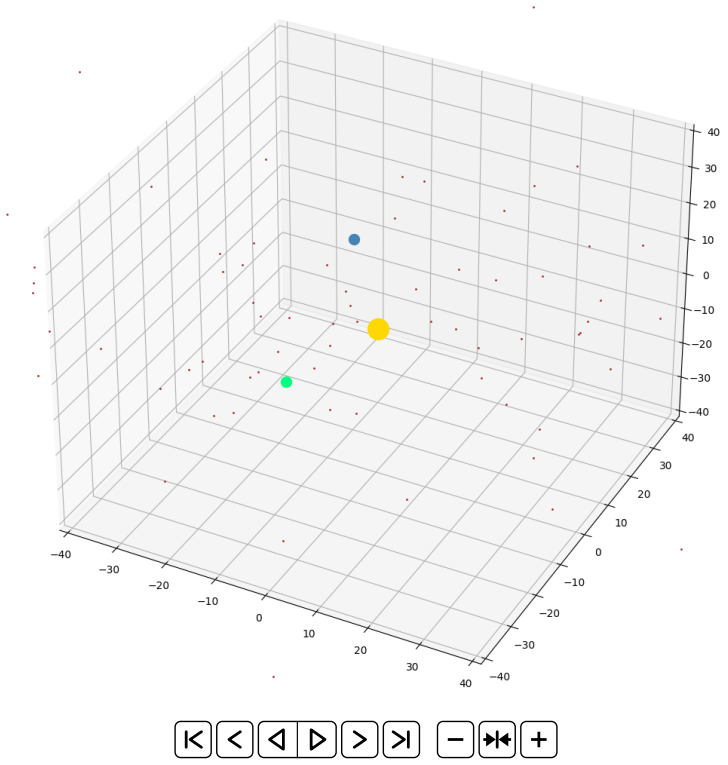

Figure 3: The Figure shows the second simulation scenario. It shows one sun (yellow) in the center surrounded by two planets and 100 asteroids. The animation is rendered within "matplotlib". If by any chance the animation doesn't play, consider using a different pdf reader such as adobe acrobat.

Body nr.	mass (kg)	$\vec{pos}$ (m)	$\vec{vel}$ $(\frac{m}{s})$	$\vec{acc}$ $(\frac{m}{s^2})$	$\vec{force}$ $(kg\frac{m}{s^2})$
1	$9e^{11}$	$[0,0,0]$	$[0,0,0]$	$[0,0,0]$	$[0,0,0]$
2	$2.7e^{10}$	$[30,0,0]$	$[0,-10,0]$	$[0,0,0]$	$[0,0,0]$
3	$2.7e^{10}$	$[-30,0,0]$	$[0,10,0]$	$[0,0,0]$	$[0,0,0]$

Table 2: galaxy simulation starting conditions, excluding asteroids.

The second simulation preset, in particular, provides a compelling portrayal of the intricate and chaotic nature of N-body simulations. Initially, the system maintains stability due to the carefully chosen initial conditions. However, as time progresses, the system undergoes a gradual acceleration toward increased instability. Eventually, the system exhibits signs of re-stabilization, intriguingly achieved by pushing one of the planets to a higher orbit. This intriguing behavior demonstrates the nuanced dynamics of N-body systems, showcasing transient phases of instability and subsequent re-stabilization, providing valuable insights into the underlying gravitational interactions.

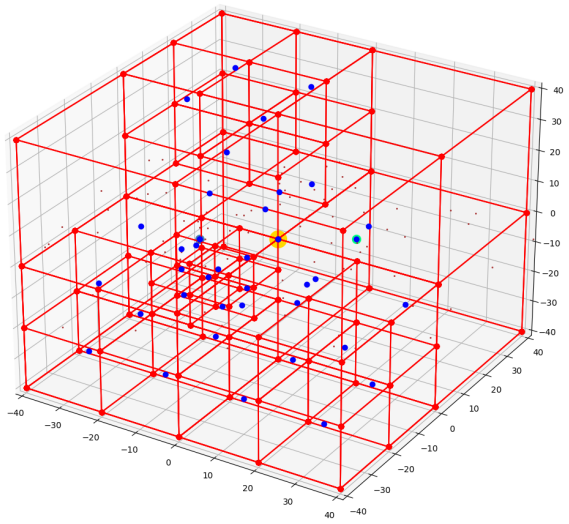

Figure 4: A representation of the renderer Oct-Tree structure and the respective weighted average mass position.

6 Discussion

The instability observed in the simulation of the galaxy problem, particularly with 100 asteroids, does not seem to stem from errors in the code or functions. Different integration methods, as well as simulations without the Barnes-Hut optimization, have produced similar results. Notably, when no asteroids are present, the three-body system initialized in the galaxy simulation shows no signs of instability. This observation points towards the likelihood that the chosen simulation size and celestial body masses may be contributing factors to the observed instability. The current simulation confines celestial bodies within a relatively small 50 by 50-meter box, coupled with high mass numbers. It is conceivable that adjusting simulation sizes and maintaining accurate ratios between distance, velocity, and mass could address the observed instability. Fine-tuning these parameters could lead to a more realistic representation of celestial dynamics, offering improved stability and accuracy in simulating complex interactions within the galaxy. Further exploration and optimization of simulation parameters, including variations in integration methods and the consideration of Barnes-Hut optimization, could yield valuable insights for achieving a more reliable and accurate simulation of celestial systems.

References

[1] J. Heer. The barnes-hut approximation, efficient computation of n-body forces. URL https://jheer.github.io/barnes-hut/.

[2] J. Kang. An interactive explanation of quadtrees. URL https://jimkang.com/quadtreevis/.

[3] D. Shiffman. Mutual attraction quadtree by codingtrain. URL https://editor.p5js.org/codingtrain/sketches/joXNoi9WL.

[4] T. Ventimiglia and K. Wayne. The barnes-hut algorithm. URL http://arborjs.org/docs/barnes-hut.