International Baccalaureate Diploma Programme

Extended Essay

Python for N Body

“To what *extent can python be optimised for running N body simulations?*”

Word Count: 3695

Computer Science Extended Essay

Table of Contents

1. Introduction

As programming becomes an increasingly important tool for a person to be successful in any career, the need to learn programming is on the rise. However, programming has not been known for its beginner friendliness, on the contrary, according to HESA[1], computing has the highest non continuation rate at 7.7% in the year 2019/20. With that said, some programming languages are still more user friendly than others; there is a general consensus within the programming community that Python is one of the most friendly programming languages while Java and C++ is one of the hardest to learn. However, convenience comes with a trade-off in performance; according to The benchmark games, C++ is 20 to 30 times faster than Python[2]. This performance difference is due to the program being translated to the computer, which only understands text in a language called binary. An interpreted language like python, will translate line by line to the computer. Interpreting a language like this has allowed python to be flexible and easy to learn, attracting a huge community of new and seasoned programmers. A compile program like C++ will be compiled or be translated to assembly language as a whole, then that will again be translated to binary and fed into the computer. The translation process might take more time, but when the program needs to run, it will be faster. Along with that, C++ gives a lower level control over how data is used within a program. This, while making it harder to learn the language, makes it possible to optimise the program to a further extent. This performance is preferred when the program is required in applications where scientific calculations are needed.

One example of these applications is the N body simulation, where a simulation is run to approximate the motions of particle over time, and especially when dealing with force such as gravitational force, where the force act over an infinite distance, where an object on one side of the universe can still feel the extremely small force by an object from the other side, given that the size of universe is finite. During my research, I will determine the answer to the question of whether python can be optimised for N body simulation (a task that is traditionally done in C++)? The research method that is used in this research is an experimental method to evaluate different Python libraries and algorithms against each other and a C++ CUDA program utilising the GPU. These Python libraries, which are coded in different programming languages, can be easily accessed in Python with a few lines of code and can help the program to surpass its own performance limit. The extent of how various and powerful these libraries are only possible because of the huge community that is formed due to how user friendly the language is. Through the research, it will be concluded that Python even though still falling short from C++, still be able to get sufficient performance to calculate n-body simulations in real time.

1.1 N body simulation

N Body simulation was pioneered by Hoomsburg in 1941, where he simulated a system of n = 37 particles using light bulbs and a galvanometer, with n is defined as the number of particles within the system. It's first use in computers started in the early sixties with up to a n = 100 particle. [4] N body simulation is widely used in the field of astrophysic, used to approximate the motions of particles in planetary system, ie. solar system, or simulating the dynamics within a galaxy and multiple galaxies, or even the interaction within the clusters, ie, a group of galaxies. With the growth of hardware devices, like the CPU or the GPU, and the invention of more and more efficient algorithms, Nbody simulation could now reach a scale of n = 2031^3 or over 1 billion particles in the Millenium run [5]. The simulation takes a supercomputer about a month to produce, and as our simulation code will be executed on a laptop, we are aiming for a number of n ≈ 10^3 or some multiple of a thousand particles. There are also some other applications for N-body simulations, such as electrostatic simulations or molecular dynamic n-body simulations for gases, however, our research will be focusing only on gravitational N-body simulations.

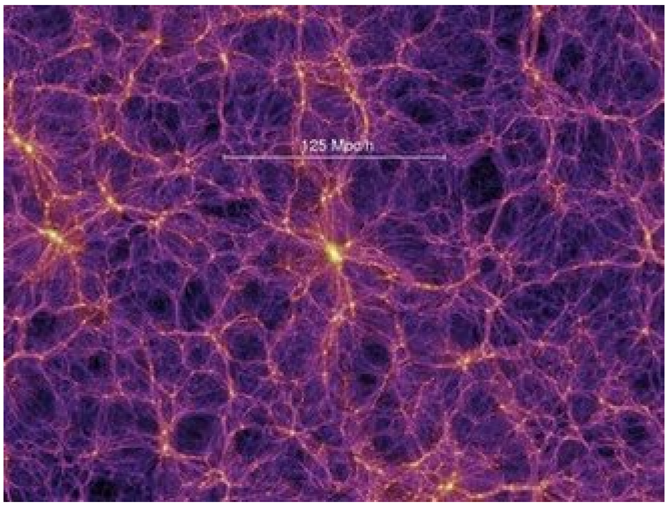


Figure 1: A snapshot of the Millenium Run, Jun of 2005 [5]

One reason why gravitational N body simulations require such computational power and research effort is due to its time complexity. In any program, there are a number of statements needed to be executed, these statements are things like print a text to the screen, or to add A and B together. The statements can then be put into loops, this can offer greater control over how many times these statements will be executed along with increasing the readability of the program. The number of statements that a program needs to be executed in its worst case scenario is called time complexity, and we notate time complexity by O(N), where N is the number of statements that need to be executed. We can use these concepts to calculate the time complexity of the gravitational N-body simulation. For each object, an inner loop will run a number of statements to calculate the force between itself and every other object. We then use an outer loop to run the process over every particle each time step, making the time complexity of an Nbody simulation per frame of time $O(n^2)$.

2. Algorithm Design

In order for Python to be as fast and efficient as C++, multiple non-traditional methods are used to make the program run faster. In this section, I will describe different algorithms that are being evaluated.

$$\sum \frac{Gm_o m_i}{\left|r_{oi}\right|^2} \widehat{r_{oi}}$$

Newtonian gravitational force equation [3]

2.1 Classic Algorithm

First, it is necessary to understand the data architecture and the brute force algorithm for an N-body simulation. There are many approaches to building a data structure that stores information about the particles. The approach I decided to use contains two 2-dimensional matrices: one matrix for storing information of position at time t, or P(t), and the other position at t -1, P(t-1), and one matrix storing the mass, and float storing dt. P(t) is used to calculate the distance between two particles, r, the magnitude of r, $|r|$, and the unit vector of r, $\hat{r}$, will be calculated to obtain the force. Other notable notations are the notations for indexing, as we need a label for the particle we are dealing within a loop. When we loop over particles in the algorithm (the outer loop), we are taking a particle at a time, we call that particle i, and take it's position with every other particle, we do that by calling another loop within the outer loop, the inner loop would take every other particle, notated as o and find the difference in P(t) between i and o. With the notation defined, we can do that set the equation used for each time step by first setting up Newtonian gravitational force equation equal to Newton second law below:

$$F = m_i a \text{ (Newton's second law of motion)}$$

$$\sum \frac{Gm_o m_i}{\left|r_{oi}\right|^2} \widehat{r_{oi}} = F$$

$$\sum \frac{Gm_o m_i}{\left|r_{oi}\right|^2} \widehat{r_{oi}} = = m_i a$$

$$\sum \frac{Gm_o}{\left|r_{oi}\right|^2} \widehat{r_{oi}} = a \text{ (Cancel each other out)}$$

We can then skip a step in calculation by cancelling mass to obtain acceleration, a. a can then be plugged into the verlet integration method:

$$P(t + 1) = 2P(t) - P(t - 1) + a\Delta t^2$$

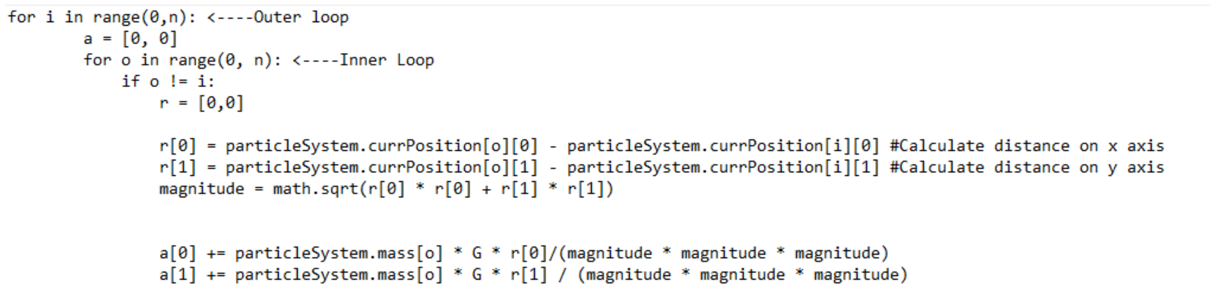
Verlet Integration [6] is used to derive a position with less error because it helps to get rid of Δt, decreasing the error when approximating the position of the particle over time. The process above is then repeated for particles at i. The pseudocode of the process is as follow:

```
1. For all particle i:
2. Create 2d matrix of new position, $P_{new}$
3. For all particle o, calculate distance with particle i
4. Acceleration is calculated with the individual distance
5. Sum the acceleration
6. $P_{new}(t + 1)[i]$ = Approximate $P(t + 1)$ with verlet integration
7. i increment by 1
8. Repeat until i = n
9. Set $P(t-1) = P(t)$, $P(t) = P_{new}$
```

2.2 Partial Vectorization

The next type of algorithm is called vectorization. [7] Vectorization, or array algorithm is a way the libraries called Numpy and Cupy are used to describe a simple operation on all elements of an array. An array is a list storing elements with the same number type. For loops are the traditional method to apply an operator on an array, which is slower in languages like Python and faster in languages like C and C++. To provide an analogy for why this is, think of a train and its track, the train is the for loop run on a track [8]. C, C++ and Java are able to send millions of workers working in parallel to build track tens of thousands of steps ahead, while Python is only able to send one worker at a time, hence why languages like Python are much slower to run. Numpy is able to push its loops onto another language, C, hence why it is able to run much faster than a traditional loop in Python. To utilise this increase in speed, we need to make a few modifications to our classic N Body solver algorithm.

In a partial vectorization code, we aim to vectorize the inner loop that was used to calculate distance between o and i, and we can go straight to calculating the distance between every other particle and i with Numpy. We can treat this calculation as a traditional all element in an array minus a constant for numpy vectorization, and that leaves us with an array of distance between all particles and i. To calculate the magnitude of the distance 2d matrix, we can use Numpy Indexing to multiply two 1d arrays together element wise, then use the numpy square root function to get an array of magnitude. This approach can be used to calculate the acceleration and the new position. Figure 2 shows the code for the traditional method, showing the two loops used in calculation, and Figure 3 shows us that we have already eliminated the inner loop with partial vectorization.

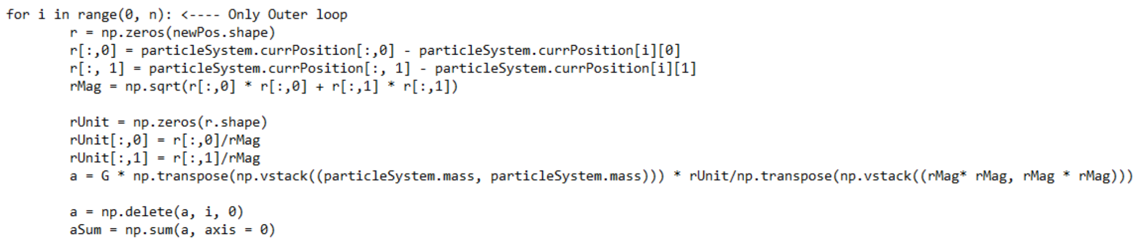


```
for i in range(0,n): <----Outer loop
        a = [0, 0]
        for o in range(0, n): <----Inner Loop
            if o != i:
                r = [0,0]

                r[0] = particleSystem.currPosition[o][0] - particleSystem.currPosition[i][0] #Calculate distance on x axis
                r[1] = particleSystem.currPosition[o][1] - particleSystem.currPosition[i][1] #Calculate distance on y axis
                magnitude = math.sqrt(r[0] * r[0] + r[1] * r[1])


                a[0] += particleSystem.mass[o] * G * r[0]/(magnitude * magnitude * magnitude)
                a[1] += particleSystem.mass[o] * G * r[1] / (magnitude * magnitude * magnitude)
```

Figure 2: Traditional distance calculation



```
for i in range(0, n): <---- Only Outer loop
        r = np.zeros(newPos.shape)
        r[:,0] = particleSystem.currPosition[:,0] - particleSystem.currPosition[i][0]
        r[:, 1] = particleSystem.currPosition[:, 1] - particleSystem.currPosition[i][1]
        rMag = np.sqrt(r[:,0] * r[:,0] + r[:,1] * r[:,1])

        rUnit = np.zeros(r.shape)
        rUnit[:,0] = r[:,0]/rMag
        rUnit[:,1] = r[:,1]/rMag
        a = G * np.transpose(np.vstack((particleSystem.mass, particleSystem.mass))) * rUnit/np.transpose(np.vstack((rMag* rMag, rMag * rMag)))

        a = np.delete(a, i, 0)
        aSum = np.sum(a, axis = 0)
```

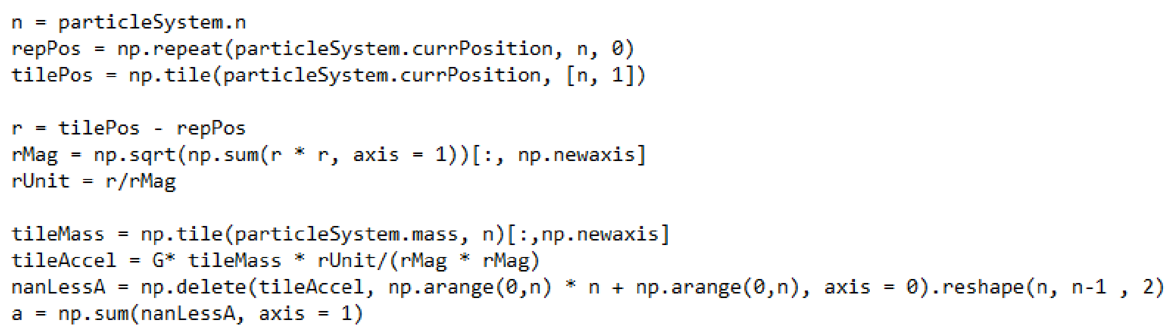
Figure 3: Numpy Partial Distance calculation

This approach also poses new challenges or aspects the programmer has to watch out for. One notable challenge is to get rid of the distance between i and itself, as the simple element wise subtraction will also take the position between i and i. According to the Newtonian gravitational force equation (NGE), the resultant force will be infinity. Implications of this phenomenon is beyond the scope of my research, however, modification to the original NGE can soften the force at small r, this is called a softened NGE, we won't be using this modification in this research as the research is mainly focusing on the performance. To get rid of acceleration calculated with the distance between each i and i, we can just go into the array and set the acceleration at in each acceleration matrix to be equal to 0 instead of inf.

2.3 Total Vectorization

To further optimise the performance of Python, total vectorization was developed in this research to eliminate the outer loop by increasing the shape of the data structures within operation. During the traditional and partial vectorization approach, the largest size of one matrix was a multiple of n, during total vectorization approach, the size of one usual matrix are n * n * 2. For this to work, I will need to introduce what a np.tile and np.repeat.

does. np.tile will tile the positions elements on to each other, which looks something like [0, 1, 2,...n -1, n, 0, 1,2, …n -1, n], whereas np.repeat would place the element duplicated next to themself, which looks like this [0, 0, 1, 1, 2,2 …, n, n]. If we tile and repeat the position vector with n, we can obtain two different (n*n, 2) matrix, where the tile would represent particles at o, and repeat would represent the particles at i. We can then subtract the two tile and repeat element wise, then we would obtain a matrix of (n*n,2) of distance between each particle and every other particle, placed on top of each other. We can then treat them as a very long matrix of elements, where acceleration can be calculated using similar approaches as the partial vectorization technique. To due with a similar inf acceleration problem, we can have an array between [0…n], then multiply by n, then add with [0 … n], we would get the index of which acceleration to get rid of . After, we can then collapse into the net acceleration for each i particle by reshaping the (n*n, 2) to (n, n, 2), then np.sum can take care of the summation over the axis. This approach was later shown to have similar performance as the partial vectorization in Numpy, however was able to produce the best performance of all the techniques with Cupy.



```
n = particleSystem.n
repPos = np.repeat(particleSystem.currPosition, n, 0)
tilePos = np.tile(particleSystem.currPosition, [n, 1])

r = tilePos - repPos
rMag = np.sqrt(np.sum(r * r, axis = 1))[:, np.newaxis]
rUnit = r/rMag

tileMass = np.tile(particleSystem.mass, n)[:,np.newaxis]
tileAccel = G* tileMass * rUnit/(rMag * rMag)
nanLessA = np.delete(tileAccel, np.arange(0,n) * n + np.arange(0,n), axis = 0).reshape(n, n-1 , 2)
a = np.sum(nanLessA, axis = 1)
```

Figure 4: Total Vectorization code

2.4 C++ CUDA

The C++ CUDA algorithm will be a modified version of the mini-nbody original made by Harrism on github. The code originally had 3 dimensions running with 200 bodies and 10 time steps, it was then modified down to 2 dimensions with 1000 bodies running through 2000 timesteps.

3. Experimental procedure

With all the algorithms presented above, we can then set up a controlled experiment to test the performance of all the algorithms against each other. This section will go into details on the independence variables, dependent variables, and controls, along with measurement algorithms for the experiments.

The independent variables in this experiment are the algorithms used to solve the position of the particle over different time and the dependent variable is the time it takes for the simulation to run. Due to time being measured, other variables would also come into play that we need to consider. When an algorithm is run, its hardware comes greatly into play of how fast it can run, especially when we are measuring the performance of code that are actively trying to maximise the speed with specific hardware like Cupy[9]. Hence the program will be run on one laptop, which has the hardware or specification as below.

CPU	Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz 2.21 GHz
GPU	GTX 1050 TI Max-Q design
RAM	8GB DDR4
Window Version	Window 11 home edition
System type	64 bit, CUDA 1.16
SSDs capacity	256 GB

Table 1: Hardware specification.

Along with that, to make the load on the cpu to be constant, the laptop will run no other program in the process. Of course, that excludes programs that have already been run in the background when window is starting up.

Different libraries will also be tested with the same algorithm, in particular 2 different libraries on 5 different algorithms will be measured against the C++ CUDA algorithm. First, we have the python classic model, running on classic python with no other libraries but Numpy to store and generate the particle positions and mass. Then partial vectorization and total vectorization will be measured for both Numpy and Cupy. Numpy is a library containing efficient data structures and algorithms that are being pushed onto another language for faster computation[7]. Cupy is similar to Numpy, however, it opted to use the Nvidia GPU to speed up its array computation. Hence, keeping the GPU a control variable is important and is stored in the specification above. Other libraries are also used in this experiment for storing the data about the experiment. Pandas will be used to save and export data frames containing measured time for solving. That data will be imported from another file to be plotted in Matplotlib. Matplotlib is then also used to render the particles over time. However, the rendering time will not be considered when measuring the performance of each algorithm.

The measurement algorithm will consist of the library datetime, where we can get the time elapsed per time step by taking the time when the program finishes calculation - the time when it starts. The process will be run over 2000 timestep.

The summation about the experiment are as follows:

Procedure:

1. Initialise information about particle
2. Initialise the dependent variable measurement matrix
3. For t before time limit:
4. Measure time start

5. Run algorithm
6. Measure time stop
7. Measurement = Stop - start
8. Set frame to first column and measurement on second column
9. Export to csv

With variables summing up as:

Independence variable	Traditional Python, Numpy Partial Vectorization, Cupy Partial Vectorization, Numpy Total Vectorization, Cupy Total Vectorization, C++ CUDA
Dependent variable	Time elapsed when running the algorithms
Control	● Number of program also running ● Time limit ● Measurement algorithm ● Hardware specification

Table 2: Independent, dependent and control variables of the experiment

All the produced file with the algorithm from this section and algorithm design will be included in https://github.com/Cody-Le/Nbody/tree/main/NbodyPy [10]

4. Result and analysis

After all the algorithm, a program has been created to measure the performance of each algorithm, along with visualisation. This section is meant to showcase that, along with the result of the experiments.

Visualisation of resulting simulation:

First we have the demonstration of the algorithm simulating a binary star system with two stars over arbitrary units of position.

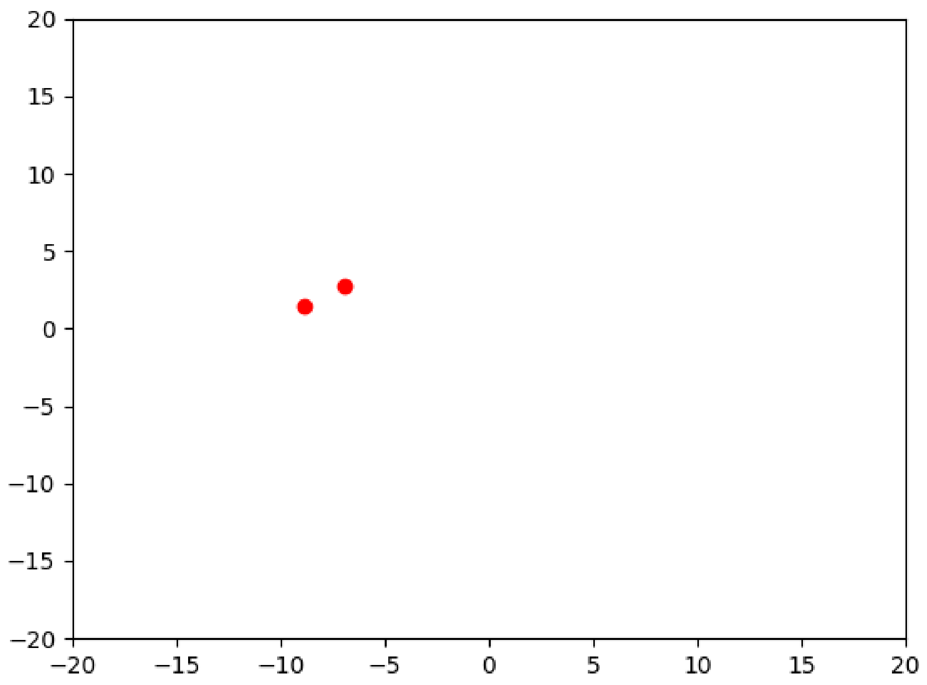


Figure 5: Binary star system simulation visualised with Matplotlib and CTV algorithm

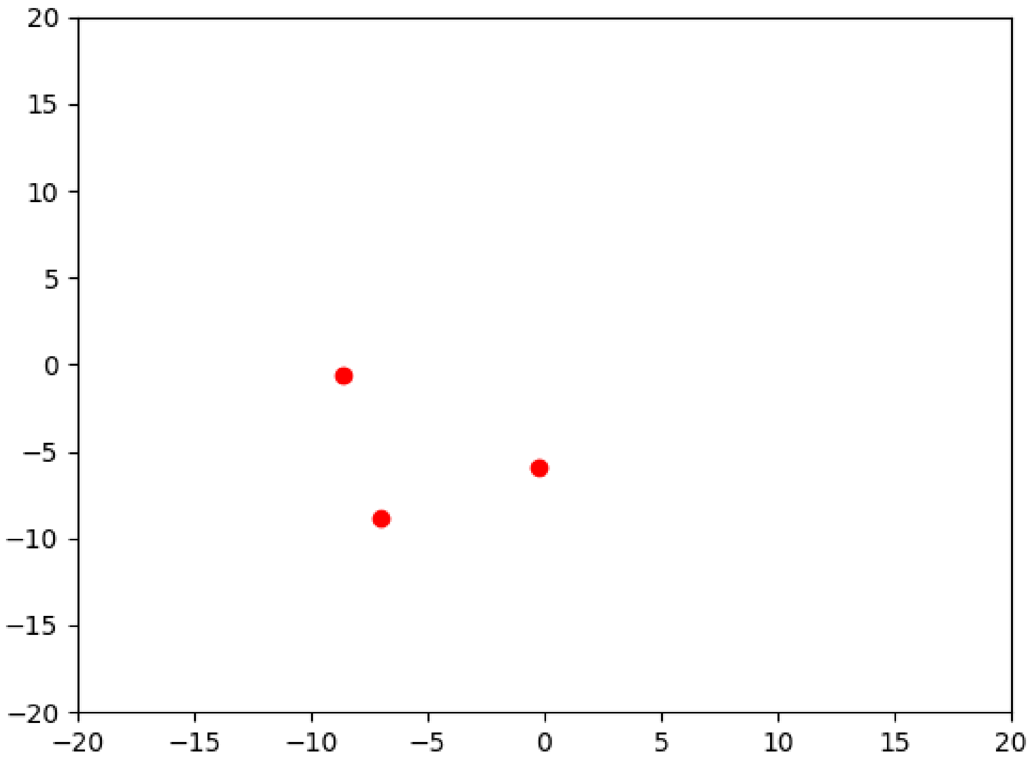


Figure 6: Three body simulation visualised with Maplotlib and CTV algorithm

Note that simulation wasn't meant to be accurate to simulating real life, as we are aiming instead for performance, hence factors such as randomly generated mass, velocity and position were chosen for its convenience in error detection instead of accuracy. The gravitational constant features in NGE are also replaced by another constant to promote gravitational-like behaviour.

Experimental result:

The results of the experiments which ran over 2000 timesteps to test the result of each other were put into different csv files, which are on link to the github page [10]. The general statistics on the result are shown below.

Algorithm	Mean (ms)	min(ms)	Max (ms)	range(ms)
Classic Python	2385	2236	3108	872
Numpy Partial	112	100	263	163
Numpy Total	118	100	181	83
Cupy Partial	1039	845	1561	715
Cupy Total	30	27	112	85
C++ CUDA	3	2	6	4

Table 4: Experiment results statistics

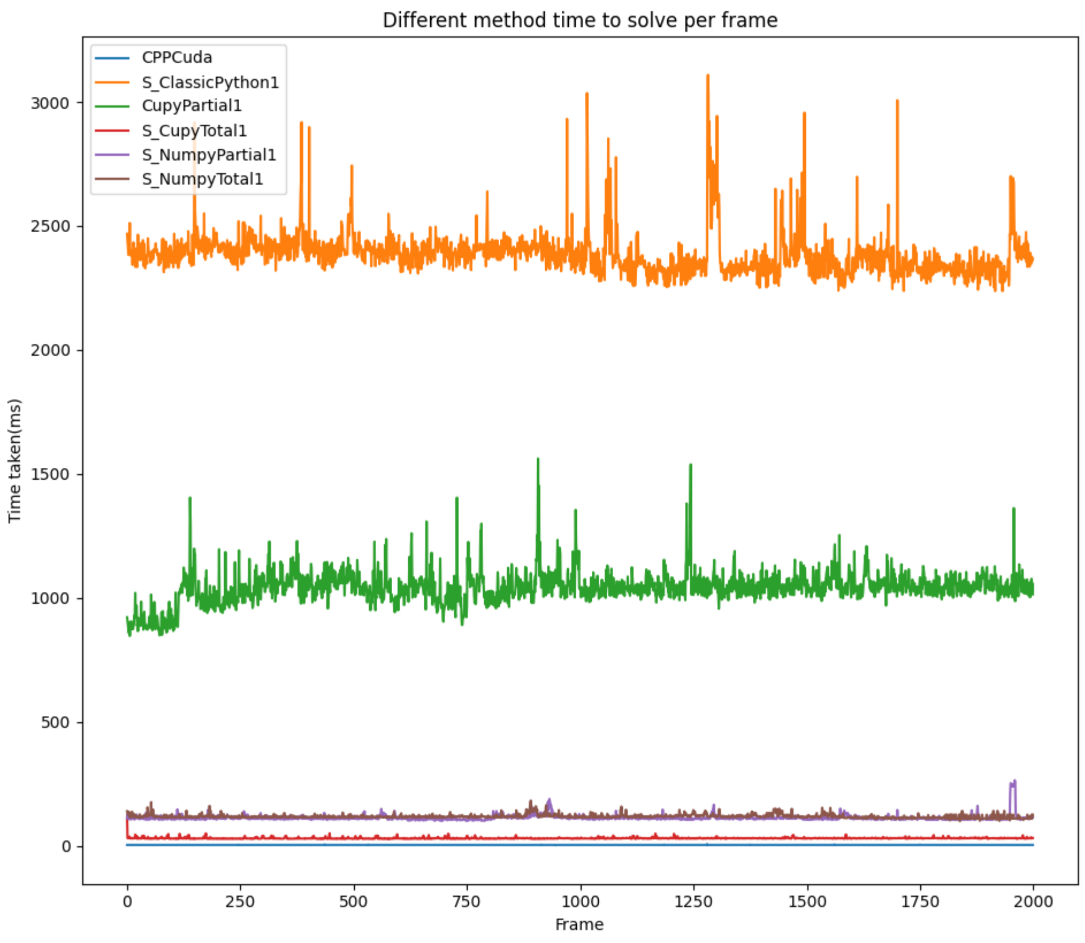


Figure 7: Time of calculation over timesteps

Table 3 shows the fastest, slowest and range of the time it takes to solve each time step over the time span of 2000 frames. As we can see the performance of Classic Python seems to be worse in all of the cases as predicted, with the highest frame fluctuation (range) and means. Numpy Partial and Numpy Total have higher performance than the classic Python. Interestingly, they have similar performance, this might be because of the same number of operations that the CPU needed to perform even if we eliminate the outer loop. The most interesting point, however, is that CuPy Partial has relatively low performance, where it is hypothesised that Cupy will be at least faster than NumPy Partial, due its utilisation of parallel computation. One possible reason as to why this is might relate back to the how Cupy go from CPU based data to GPU based data, Cupy is using CUDA based computation [9], which is a framework that [10] are use by developers to utilise the computation power of GPU for

C++, within the program, there is step in which we need to allocate memory in GPU, which require the developer to create an empty slot on GPU's VRAM, then copy the CPU memory from that the CPU's RAM to the VRAM, the data then get computed in kernels, then retrieved back by copying the data in VRAM to RAM. Transporting memory like this might take a long time, which is why Cupy Partial is slower, because the algorithm divides the data into chunks, then uploads it to the GPU n amount of times. In contrast, algorithms such as Cupy Total where there are only a few instances of converting memories between VRAM and RAM would prove to have the best performance of all the Python Algorithms.

With the data above, we can deduce the amount of frame rate that each of the average time would produce, this is a different amount of frame rate when we are visualising however, because rendering might take extra time. The equation to convert from average time to solve, μT to frame per second is as below:

$$F_{ps} = \frac{1000ms}{\mu T}$$

The data calculated is shown below:

Algorithm	FPS
Classic Python	0.419
Numpy Partial	8.93
Numpy Total	8.47
Cupy Partial	0.962
Cupy Total	33.3
C++ CUDA	333

Table 5: Frame Rate for each algorithm

For a standard of a video on the internet is shown with 30 fps, which with the Fps equation above, the time to produce each frame is about 33 millisecond. Table 4 shows that with the Cupy total algorithm, the solve time was able to produce barely above 30 frames per second without accounting for the rendering/visualising time, which means that it can render a simulation real time. However it still significantly falls short from the performance of C++ CUDA, as the performance of C++ CUDA would be around ten times faster than that of the Cupy Total Vectorization algorithm.

5. Conclusion

The data above shows that Python was able to reach a bare minimum frame rate for real time gravitational N-body simulation, despite falling short from C++ CUDA. However, the frame rate is not possible without the indirect use of other languages such as C in Numpy or CUDA for Cupy, or the deep understanding of libraries specific functions such np.tile and np.repeat, which could be argued to have increase the complexity in creating the program, which is something we are trying to eliminate with Python. Another point to acknowledge is that solving the program in real time isn't the only approach to solving a simulation, another approach would be solving them before saving the data and to visualise the data point separately, which would have increased the performance significantly. Therefore, we can say that although Python capacity has proven itself to be useful, it is not yet ready to be used to develop heavier tasks used in simulation.

Nevertheless, there are ways in which new methods could be developed to even further enhance the speed of Python. Python is now the most widely used language for applications such as Machine Learning for its user friendliness, Machine learning is also now has its uses in applications for molecular simulation [13], then it might be possible for a machine learning GAN model to generate an entire video base on initial conditions at t=1[14]. Python also harbours libraries like Quiskit, a popular library for quantum computing. A new quantum computing circuit may be developed to be suitable to further optimise the simulation to run on quantum computers [15]. If we look for a solution on a more general scale, there are other faster rising programming languages, one of them being Julia, which have similar syntax with Python, but have speed near C and C++ [16][2]. These solutions provide us with many possibilities for research to increase in performance, with Julia particularly showing the trend will still continue pass Python to increase the performance and user friendliness at the same time for both languages, hoping to bring a more future where getting started in programming could be considered friendly.

6.Works Cited

[1] "Non-Continuation: UK Performance Indicators." *HESA*, https://www.hesa.ac.uk/data-and-analysis/performance-indicators/non-continuation.

[2] *Which Programming Language Is Fastest?*, https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html.

[3] "N-Body Simulations." *Princeton University*, The Trustees of Princeton University, https://physics.princeton.edu//~fpretori/Nbody/intro.htm.

[4] Trenti, Michele, and Piet Hut. "N-Body Simulations (Gravitational)." *Scholarpedia*, http://www.scholarpedia.org/article/N-body_simulations_(gravitational)#:~:text=The%20history%20of%20N%2Dbody,of%20electromagnetic%20and%20gravitational%20interactions.

[5] Springel, Volker, et al. “Simulations of the Formation, Evolution and Clustering of Galaxies and Quasars.” *NASA/ADS*, https://ui.adsabs.harvard.edu/abs/2005Natur.435..629S/abstract.

[6]“Verlet Method.” *Computational Methods of Physics*, https://www.physics.udel.edu/~bnikolic/teaching/phys660/numerical_ode/node5.html .

[7]Van Der Walt, Stefan, et al. “The NumPy Array: A Structure for Efficient Numerical Computation.” *ArXiv.org*, 8 Feb. 2011, https://arxiv.org/abs/1102.1523 .

[8]BaskayaBaskaya 7, et al. “Why Python Is so Slow for a Simple for Loop?” *Stack Overflow*, 1 Aug. 1959, https://stackoverflow.com/questions/8097408/why-python-is-so-slow-for-a-simple-for-loop.

[9]“Cupy.” *CuPy*, https://cupy.dev/ .

[10]Cody-Le. “Nbody/NbodyPy.” *GitHub*, https://github.com/Cody-Le/Nbody/tree/main/NbodyPy .

[11] “CUDA Toolkit Documentation v11.7.1.” *CUDA Toolkit Documentation*, https://docs.nvidia.com/cuda /.

[12]“Welcome to Python.org.” *Python.org*, https://www.python.org /.

[13]*Machine Learning for Molecular Simulation | Annual Review of Physical ...* https://www.annualreviews.org/doi/10.1146/annurev-physchem-042018-052331 .

[14]Synced. “DeepMind DVD-Gan: Impressive Step toward Realistic Video Synthesis.” *Medium*, SyncedReview, 17 July 2019, https://medium.com/syncedreview/deepmind-dvd-gan-impressive-step-toward-realistic-video-synthesis-12027d942e53 .

[15]“Qiskit.org.” *Qiskit.org*, https://qiskit.org/ .

[16]Jeff Bezanson, Stefan Karpinski. “The Julia Programming Language.” *The Julia Programming Language*, https://julialang.org/ .

[17]Harrism. “Harrism/Mini-Nbody: A Simple Gravitational N-Body Simulation in Less than 100 Lines of C Code, with CUDA Optimizations.” *GitHub*, https://github.com/harrism/mini-nbody .

[18]“Gravitational Constant.” *Oxford Reference*, https://www.oxfordreference.com/view/10.1093/oi/authority.20110810105019672#:~:text=The%20constant%20of%20proportionality%20relating,11%20Nm2%2Fkg2 .

7. Appendix

```
import numpy as np
import cupy as cp
import math
G = 0.01;

print(G)


def S_classPython(particleSystem):
    n = particleSystem.n;
    newPos = np.zeros(particleSystem.currPosition.shape)
    for i in range(0,n):
        a = [0, 0]
        for o in range(0, n):
            if o != i:
                r = [0,0]

                r[0] = particleSystem.currPosition[o][0] - particleSystem.currPosition[i][0] #Calculate distance on x axis
                r[1] = particleSystem.currPosition[o][1] - particleSystem.currPosition[i][1] #Calculate distance on y axis
                magnitude = math.sqrt(r[0] * r[0] + r[1] * r[1])


                a[0] += particleSystem.mass[o] * G * r[0]/(magnitude * magnitude * magnitude)
                a[1] += particleSystem.mass[o] * G * r[1] / (magnitude * magnitude * magnitude)
        newPos[i][0] = 2 * particleSystem.currPosition[i][0] - particleSystem.prevPosition[i][0] + a[0] * \
                particleSystem.dt ** 2
        newPos[i][1] = 2 * particleSystem.currPosition[i][1] - particleSystem.prevPosition[i][1] + a[
            1] * particleSystem.dt ** 2

    particleSystem.prevPosition = particleSystem.currPosition
    particleSystem.currPosition = newPos







def S_numpyPartial(particleSystem):
    n = particleSystem.n
    newPos = np.zeros(particleSystem.currPosition.shape)
    for i in range(0, n):
        r = np.zeros(newPos.shape)
        r[:,0] = particleSystem.currPosition[:,0] - particleSystem.currPosition[i][0]
        r[:, 1] = particleSystem.currPosition[:, 1] - particleSystem.currPosition[i][1]
        rMag = np.sqrt(r[:,0] * r[:,0] + r[:,1] * r[:,1])

        rUnit = np.zeros(r.shape)
        rUnit[:,0] = r[:,0]/rMag
        rUnit[:,1] = r[:,1]/rMag


        r = 0


        a = G * np.transpose(np.vstack((particleSystem.mass, particleSystem.mass))) * rUnit/np.transpose(np.vstack((rMag* rMag, rMag * rMag)))

        a = np.delete(a, i, 0)

        aSum = np.sum(a, axis = 0)
        newPos[i] = 2 * particleSystem.currPosition[i] - particleSystem.prevPosition[i] + aSum *  np.square(particleSystem.dt)

    particleSystem.prevPosition = particleSystem.currPosition
    particleSystem.currPosition = newPos

def S_cupyPartial(particleSystemCP):
    n = particleSystemCP.n
    newPos = cp.zeros(particleSystemCP.currPosition.shape)
    for i in range(0, n):
        r = cp.zeros(newPos.shape)
        r[:,0] = particleSystemCP.currPosition[:,0] - particleSystemCP.currPosition[i][0]
        r[:, 1] = particleSystemCP.currPosition[:, 1] - particleSystemCP.currPosition[i][1]
        rMag = cp.sqrt(r[:,0] * r[:,0] + r[:,1] * r[:,1])

        rUnit = cp.zeros(r.shape)
        rUnit[:,0] = r[:,0]/rMag
        rUnit[:,1] = r[:,1]/rMag


        r = 0


        a = G * np.transpose(cp.vstack((particleSystemCP.mass, particleSystemCP.mass))) * rUnit/np.transpose(cp.vstack((rMag* rMag, rMag * rMag)))

        a[i,0] = 0
        a[i,1] = 0
```

```
            aSum = cp.sum(a, axis = 0)
            newPos[i] = 2 * particleSystemCP.currPosition[i] - particleSystemCP.prevPosition[i] + aSum *  cp.square(particleSystemCP.dt)


        particleSystemCP.prevPosition = particleSystemCP.currPosition
        particleSystemCP.currPosition = newPos


def S_numpyTotal(particleSystem):

    #acceleration calculation
    n = particleSystem.n
    repPos = np.repeat(particleSystem.currPosition, n, 0)
    tilePos = np.tile(particleSystem.currPosition, [n, 1])

    r = tilePos - repPos
    rMag = np.sqrt(np.sum(r * r, axis = 1))[:, np.newaxis]
    rUnit = r/rMag
    r = 0
    tileMass = np.tile(particleSystem.mass, n)[:,np.newaxis]
    tileAccel = G* tileMass * rUnit/(rMag * rMag)
    nanLessA = np.delete(tileAccel, np.arange(0,n) * n + np.arange(0,n), axis = 0).reshape(n, n-1 , 2)
    a = np.sum(nanLessA, axis = 1)
    #next position
    newPos = 2 * particleSystem.currPosition - particleSystem.prevPosition + a*np.square(particleSystem.dt)
    particleSystem.prevPosition = particleSystem.currPosition
    particleSystem.currPosition = newPos
    pass

def S_cupyTotal(particleSystemCP):

    #acceleration calculation
    n = particleSystemCP.n
    repPos = cp.repeat(particleSystemCP.currPosition, n, 0)
    tilePos = cp.tile(particleSystemCP.currPosition, [n, 1])

    r = tilePos - repPos
    rMag = cp.sqrt(cp.sum(r * r, axis = 1))[:, cp.newaxis]
    rUnit = r/rMag
    r = 0
    tileMass = cp.tile(particleSystemCP.mass, n)[:,cp.newaxis]
    tileAccel = G* tileMass * rUnit/(rMag * rMag)
    zeroIndex = np.arange(0,n) * n + np.arange(0,n)
    tileAccel[zeroIndex, :] = 0
    A = tileAccel.reshape(n, n, 2)
    a = cp.sum(A, axis = 1)
    #next position
    newPos = 2 * particleSystemCP.currPosition - particleSystemCP.prevPosition + a*cp.square(particleSystemCP.dt)
    particleSystemCP.prevPosition = particleSystemCP.currPosition
    particleSystemCP.currPosition = newPos
    pass
```