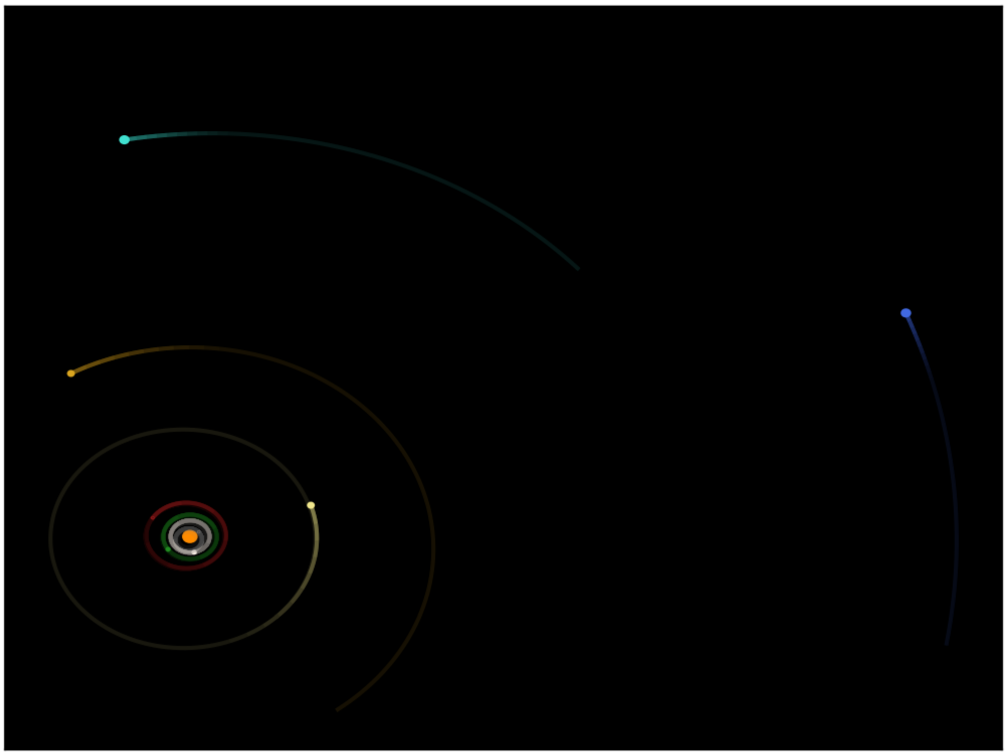
THE UNIVERSITY OF MELBOURNE

Asteroid Trajectories

Matthew Thomas, 831343

June 4, 2021



Contents

1

Introduction

Asteroids colliding with Earth are capable of inflicting enormous amounts of damage. It is widely accepted that an asteroid approximately 10 km wide was the cause of the mass extinction event responsible for eradicating the dinosaurs.
In 2004 astronomers discovered Apophis, an asteroid roughly 300 metres wide. This caused quite a stir when initial calculations indicated a possibility of an impact with Earth. After further investigation, this possibility was ruled out for at least the next hundred years [4]. A recent example of an asteroid colliding with the Earth is the Chelyabinsk meteor in 2013. While this meteor was only about 20m wide it still created a large shock-wave than injured over 1,500 people. There was also the Tunguska event in 1908, after the impact of a 100 metre wide meteoroid which effected an area of 2,150 km^2, fortunately this occurred in the sparsely populated Eastern Siberian Taiga so very few people were hurt.

Given the large number of asteroids that exist, a large asteroid impacting with Earth and causing cataclysmic damage is inevitable [7]. While it is unlikely to occur in the near future, in 2016 NASA warned that Earth is woefully unprepared for defending against an asteroid [1]. Many methods have been proposed to alter the trajectory of an impending asteroid. This includes using nuclear weapons, ramming with spacecraft and focusing solar energy. While these proposals may prove effective, they all rely on being able to adjust the trajectory by a tiny amount over a long period of time. If the asteroid is found early enough, a tiny adjustment to the trajectory is enough to prevent a collision. For the best chance of success, the asteroid and its collision course needs to be determined as early as possible such that humanity has as much time as possible to alter the trajectory [11].

Finding asteroids that may collide with Earth is a very difficult prospect, organisations such as B612 [7], which is dedicated to the discovery and deflection of asteroids, have been created to map the locations of asteroids within the solar system. Even once asteroids are found, calculating their trajectory through space is an immense computational challenge.

Every body within the solar system will have a gravitational effect on the asteroid and the other celestial bodies. This results in an n-body problem, famous for its chaotic nature and unpredictability. For such a problem, the computational cost scales with the number of bodies squared [6].

Predicting asteroid trajectories as early as possible has lead to many ingenious methods to reduce the computational complexity, such as the Barnes-Hut algorithm [8] and the Fast Multipole Method [10]. Simulations of asteroid trajectories have also driven the advancement of computational hardware, and current simulations use state of the art computational systems. However, knowing you are using the most mathematically efficient method and the best hardware does not guarantee the most efficient simulation. Eventually scientists reach the point where they are using the most efficient known method on the best current hardware and in order to improve the simulation they must improve their implementation of the model and how it takes advantage of the hardware. For computationally expensive tasks such as asteroid simulations, even a slight improvement in efficiency can be an enormous difference, in the case of an asteroid, it could provide the additional time required to alter the path of a cataclysmic asteroid.

This article will focus on the point that, given a specific method and hardware, how can the implementation be optimised? By simulating an asteroids trajectory through the solar system with different implementations, we will show that not all implementations are equal.

2

Planetary motions

To simulate the motion of the planets in the solar system we will use Newtons law of universal gravitation,

$$F = \frac{Gm_1m_2}{r^2} \tag{2.1}$$

where m_1, m_2 are the masses of the two bodies, r is the separation of the bodies and G is the gravitational constant. This tells us the gravitational force experienced by one body due to another body. Using this equation we are able to predict the trajectories of celestial bodies by summing the force of a given body from all the other bodies in the system. By Newtons second law,

$$F = m\mathbf{a} = m\ddot{\mathbf{r}} \tag{2.2}$$

meaning we can write the law of gravitation as a differential equation. For an n body system, this results in a set of n differential equations:

$$\ddot{\mathbf{r}}_i = \sum_{j\neq i}^{n-1} \frac{Gm_j}{|\mathbf{r}_i - \mathbf{r}_j|^3}(\mathbf{r}_i - \mathbf{r}_j) \tag{2.3}$$

We will simulate the Sun, Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune and an Asteroid giving us a value of $n = 10$. To numerically solve this system of ODE's we will use Euler's method.

2.1 Euler's Method

Many sophisticated methods have been created for numerically solving differential equation. Our focus in this article is not on the method of solution, but how it is implemented. Because of this, we will restrict ourselves to just one method and vary the implementation.

We will use Euler's method, which is one of the simplest numerical methods for solving ordinary differential equations (ODE's). To solve an ODE of the form:

$$\frac{dy}{dt} = f(t, y(t)) \tag{2.4}$$

with initial condition $y(t_0) = y_0$, we make the approximation:

$$\frac{dy}{dt} \approx \frac{\Delta y}{\Delta t} = \frac{y(t+\Delta t) - y(t)}{\Delta t} = f(t, y(t)) \tag{2.5}$$

$$\Rightarrow y(t+\Delta t) = y(t) + \Delta t f(t, y(t)) \tag{2.6}$$

Then by starting at our initial condition $y(t_0) = y_0$ and iterating we can propagate our solution forward in time. Δt is the *step size* of the algorithm, in the limit $\Delta t \to 0$, the equation becomes exact. Euler's method is for solving first order ODE's, whereas our differential equations are second order. By introducing velocity as:

$$\mathbf{v} = \dot{\mathbf{r}} = \frac{d\mathbf{r}}{dt} \tag{2.7}$$

we can convert each second order differential equation into two first order ODE's:

$$\mathbf{v}_i = \frac{d\mathbf{r}_i}{dt}$$

$$\frac{d\mathbf{v}_i}{dt} = \sum_{j \neq i}^{n-1} \frac{Gm_j}{|\mathbf{r}_i - \mathbf{r}_j|^3}(\mathbf{r}_i - \mathbf{r}_j)$$

For our n body system our numerical solution will then be evaluated by:

Algorithm 1: Euler Method

```
for each timestep $t$ do
    for each body $i$ do
        for each body $j > i$ do
            $r^3 \leftarrow |\mathbf{r}_t[j] - \mathbf{r}_t[i]|^3$
            $\mathbf{a}[i] \leftarrow \mathbf{a}[i] + Gm_j(\mathbf{r}_t[j] - \mathbf{r}_t[i])/r^3$
            $\mathbf{a}[j] \leftarrow \mathbf{a}[j] - Gm_i(\mathbf{r}_t[j] - \mathbf{r}_t[j])/r^3$
        end
        $\mathbf{v}_{t+1}[i] \leftarrow \mathbf{v}_t[i] + \mathbf{a}[i]\Delta t$;
        $\mathbf{r}_{t+1}[i] \leftarrow \mathbf{r}_t[i] + \mathbf{v}_t[i]\Delta t$;
    end
end
```

Noting that we are taking advantage of the symmetry of Newton's Laws which reduces the computational complexity. The force on body 1 due to body 2 is equal and opposite to the force on body 2 due to body 1, mathematically $F_{ij} = -F_{ji}$. This allows the innermost loop to run only for $j > i$ reducing the total number of iterations by a half.

2.2 Initial conditions

To determine the initial conditions of the planets we use the the python astroquery package [3]. Within this package is the Horizons class which provides an interface to the Jet Propulsion Laboratory (JPL) [5]. One of the services the JPL provides is the current velocities and positions of known celestial bodies. With this class we can use a simple query that returns a vector containing the positions and velocities of the bodies of interest. Once the initial conditions for each body have been queried, these are then written to a file so that the rest of the software can read this information without re-querying JPL each time.

In order to compare the different implementations of our algorithm we want to compare how long each would take to determine that an asteroid is going to collide with Earth. As there is currently no known asteroid on a collision course with Earth, we need to create one. Fortunately, Newton's Laws are symmetric under time reversal, meaning they work the same if time is reversed. By considering an asteroid with a low mass such that the asteroids gravitational effect on the other planets is negligible. We initialise this asteroid at Earth's location with some arbitrary velocity away from Earth, then we reverse the velocities of each body allowing us to run the simulation backwards. Once the simulation is completed, we can take the final position of the asteroid and the negative of its velocity and we will have the initial coordinates of an asteroid that will collide with Earth when the simulation is run forwards.

2.3 Implementation and Results - `Python`

Now that we have the initial coordinates of the planets and the asteroid we can run the simulation. The simulation is created in `python`, for simplicity we only consider 2 dimensions, x, y. The position, velocity and acceleration of each body is separated into the x and y components, and each is stored as a list. For example, the x positions of every body is stored in one list, where the zeroth element is the x position of the zeroth body. The simulation is run using Euler's algorithm as outlined in algorithm 1, with a timestep value of $\Delta t = 0.01$, with the 10 bodies mentioned earlier, and run for $500,000$ timesteps. Due to the extreme distances between bodies and masses of bodies, working in SI units very quickly leads to numerical errors. To get around this we store distances in Astronomical Units (Au), where 1 Au is 1.496×10^8 km, mass is measured in Solar Masses (SM), with $1\,\text{SM} = 1.988 \times 10^{30}$ kg and time is stored in days. This results in a value for the gravitational constant of $G = 2.96 \times 10^{-4}\,\text{Au}^3\,\text{SM}\,\text{days}^{-2}$. The positions of each body are stored for each timestep, this allows us to plot the trajectories of each body as shown in figure 2.1.

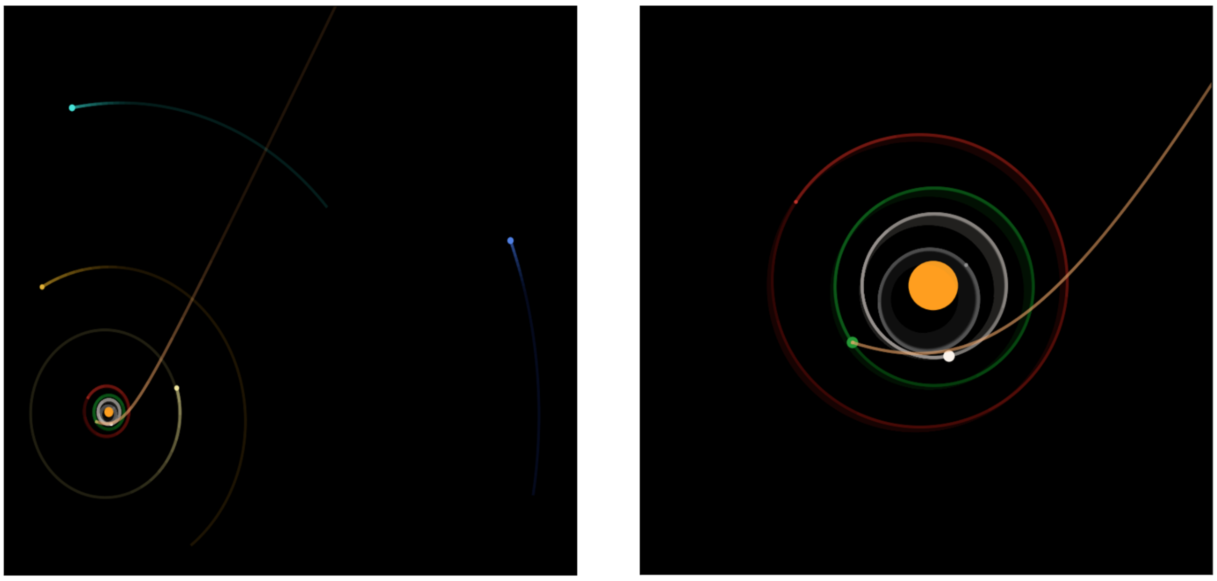


Figure 2.1: Trajectories of the planets and asteroid.

For this simple implementation written in `Python`, the simulation took 131.64 seconds to complete the $500,000$ time steps equivalent to 500 days. Figure 2.2 shows the time taken by the simulation after every $100,000$ time steps or 100 days.

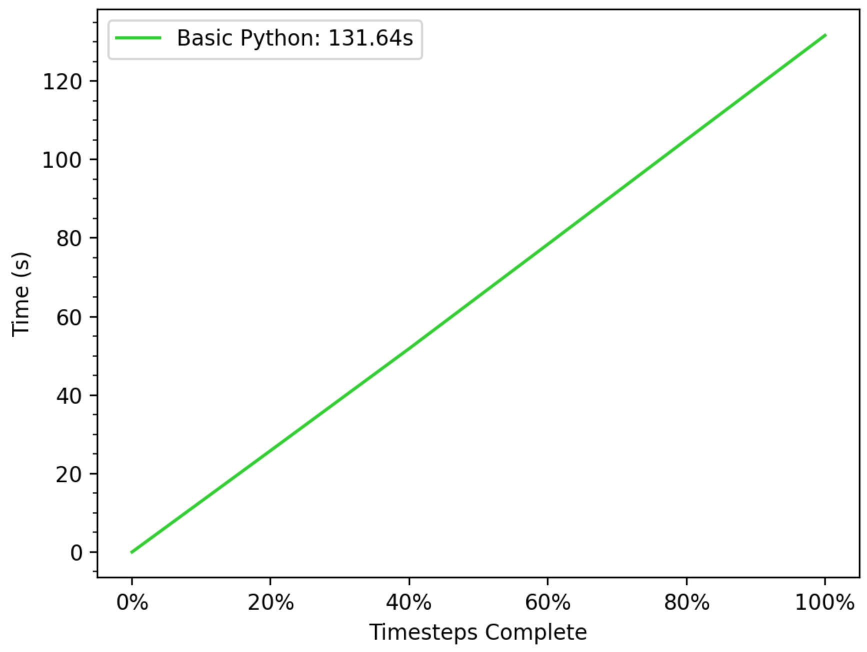


Figure 2.2: Time taken by basic `python` simulation. x axis is the percentage of the $500,000$ timesteps complete.

3

Programming Language

Thousands of programming languages exist today. While only a select few are used by a sizable portion of programmers, they each have different strengths and weaknesses and have been designed for different tasks.

3.1 Interpreted vs Compiled Languages

All high level programming languages such as `python` and `C`, are designed to be readable by humans. The Central Processing Unit (CPU) of a computer does not understand the instructions in this form, and requires a translation into *machine code*. Machine code is a low level language that is able to directly control the CPU, this includes fundamental CPU instructions such as loading, storing or arithmetic logic on memory. The way that the human readable language is converted into machine code can be done in two different ways.

`Python` is an example of an interpreted language, which uses an *interpreter* to translate the program line by line during execution into machine code. Conversely, a compiled language such as `C`, requires a *compilation* step where the entire program is translated into an executable file. Due to the additional overhead associated with interpreting line by line, typically compiled languages are significantly faster than interpreted languages. The additional efficiency is the main advantage that compiled languages have over interpreted languages [2].

Although much slower in general, interpreted languages have many advantages. With no compilation step, they can be modified and re-run much quicker. They are also more portable, as the compilation process that produces the executable file is unique to each CPU. For an interpreted language, each CPU has its own interpreter capable of converting the same file into its own appropriate machine code.

3.2 Typing

Different programming languages also use different *typing* systems. This defines the method in which the language assigns types such as `integer` or `string` to its variables. To ensure a program runs succesfully, the types of variable must be verified to enusre they can be manipulated as instructed. For example, in many languages `strings` cannot be added to `ints`, so the types of these variables must be verified before addition can occur.

`Python` is an example of a *dynamically* typed language meaning that the types are verified during runtime. This allows greater flexibility when declaring variables but introduces extra overhead when checking the types of variables. On the other hand, `C` is an example of a *static* typed language. Variables and functions in `C` must have a predefined types before they can be used or the compiler will throw an error.

We can clearly see the different between `python` and `C` if we consider a function that adds two numbers in both languages. First we consider the dynamically typed `python` function:

`Python` Function

```
def add(a, b):
    result = a + b
    return result
```

In this function the variables a and b could have any type, only when the program is run with specific a and b values will their type be verified. This provides flexibility as this function could be used for multiple types, but increases overhead as the interpreter has to check the type of both a and b and verify that they both have implementations of addition. Comparing this to the statically typed `C` function:

`C` Function

```
int add(int a, int b) {
    int result;
    result = a + b;
    return result;
}
```

Now all variables have to be declared as `int` and the function itself has the be declared as returning an `int`. This drastically reduces flexibility as only `ints` can be passed into this function without producing an error. However this can improve the efficiency as the compiler now knows exactly what types this function will receive and return [9]. This can drastically reduce the memory usage of the program, for example if the compiler knows that `ints` will be used, it can declare only the required number of bytes. This can also lead to further optimization by creating machine code that creates instructions for the CPU that are uniquely optimized for that data type.

3.3 Implementation and Results - C

To compare these differences and measure the effect on predicting asteroid trajectories, the same simulation was run using C. This simulation was run using the same structure as algorithm 1 and the same initial coordinates and all the same parameters. The $500,000$ timesteps took the C simulation only 2.25 seconds. This drastic improvement over the python implementation is shown in figure 2.2.

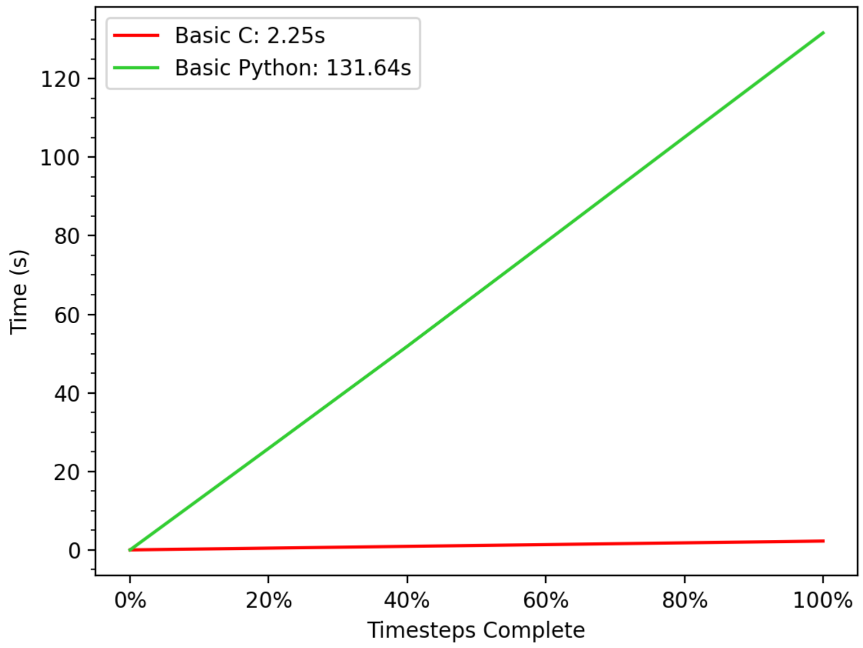


Figure 3.1: Time taken for C and python simulations.

C is typically considered a faster language than python, however the difference in these results is staggering. With the same number of iterations, calculations and array operations we are seeing a massive performance boost just by changing the programming language.

4

Python libraries

Python was originally designed for readability and simplicity not for efficiency. However many libraries have been created for python such as numpy, that are designed to give python the efficiency of other languages while maintaining python's simplicity.

Numpy is able to drastically improve the performance of python code by adding support for large, multi-dimensional arrays using pre-compiled C code. In much the same way as C arrays are defined, numpy arrays are also pre-allocated sections of memory that allow for efficient indexing and operations on the entire memory block, known as *vectorization*. `Python` loops are inherently slow, the vectorization of code allows these loops to be done as efficiently as `C` code. `Numpy` has also been highly optimized for calculations involving matrices, making use of the highly efficient Basic Linear Algebra Subroutine (BLAS) library. BLAS is the gold standard of linear algebra operations, such as dot products and matrix multiplication on CPU's.

4.1 Implementation - `Numpy`

Because `numpy's` efficiency is based around matrices, we need to adjust our algorithm to capture this efficiency gain. The modified algorithm is shown in algorithm 2:

Algorithm 2: `Numpy` Simulation

```
for each timestep $t$ do
    Compute $\mathbf{a}_t$;
    $\mathbf{v}_{t+1} \leftarrow \mathbf{v}_t + \mathbf{a}_t \Delta t$;
    $\mathbf{r}_{t+1} \leftarrow \mathbf{r}_t + \mathbf{v}_t \Delta t$;
end
```

Importantly, this has reduced the number of loops down to 1. In each timestep the acceleration is computed for all bodies at the same time via matrix multiplication, then the positions and velocities of all bodies are updated at once. To compute $\mathbf{a}$, we first create a matrix of the pairwise distances $\mathbf{dx}$ between each body via:

$$\mathbf{dx} = \mathbf{x}^T - \mathbf{x} \tag{4.1}$$

then the matrix of the inverse distances cubed is calculated by cubing this matrix. Finally this is multiplied by G and a vector of the masses of the bodies, producing the accelerations of each body without any explicit looping.

Because we are now using square matrices to store our data, we are no longer directly taking advantage of the symmetry. However BLAS routines are capable of exploiting symmetries when multiplying matrices so this should only have a minor effect on computation speed.

4.2 Results - `Numpy`

Using this `numpy` method, the simulation was run again and compared to the earlier implementations. This took 18.86 seconds to complete the $500,000$ timesteps as shown in figure 4.1.

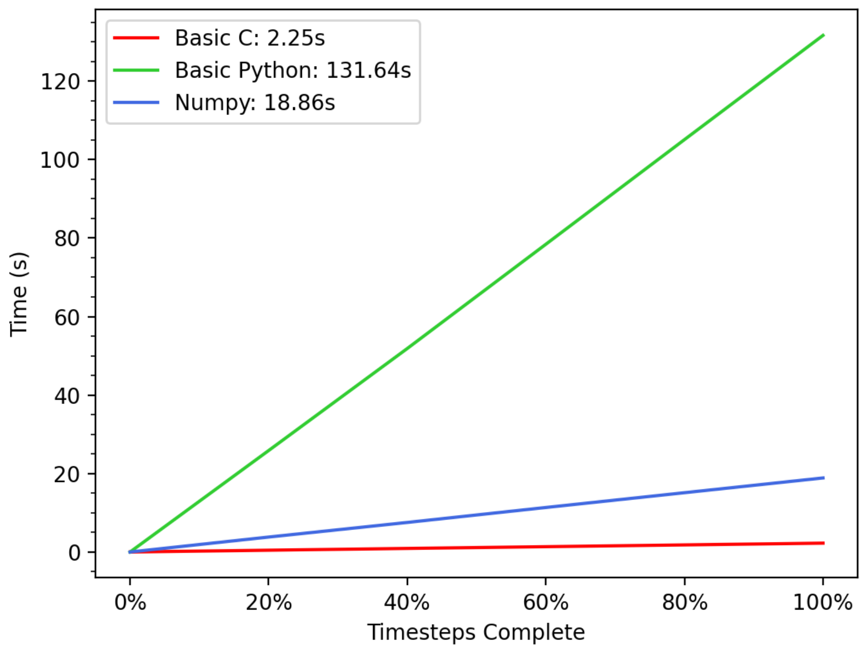


Figure 4.1: Time taken for simulation using `numpy`

While this is an enormous improvement over the basic `python` implementation, it is still significantly slower than the `C` implementation. The `numpy` implementation is able to drastically reduce the computational cost of the inner two loops of algorithm 1, however it has no effect on the outer loop through the timesteps. This outer loop is still run through `python's` inefficient looping, and cannot be vectorized as each iteration of this loop requires information from the previous iteration.

To demonstrate this, we consider a different system. We now consider an n-body simulation that contains many more bodies and far less timesteps. In relation to our algorithm, this will push the computationally expensive parts into the inner loops where `numpy` excels. For the first simulation we used $t = 500,000$ and $n = 10$, due to the single outer loop and the two inner loops, we expect the algorithm complexity to be $\mathcal{O}(t \times n^2)$, this produces $50,000,000$ iterations. For a direct comparison, we choose a system that will have the same number of iterations. Taking $t = 5000$ and $n = 100$, we generate a random set of bodies and initial positions and coordinates and run the simulation with each of our earlier implementations. This simulation has the same number of iterations, so we should expect the `python` and the `C` implementations to take a similar amount of time, but should speed up the `numpy` implementation. This produces the plot shown in figure 4.2.

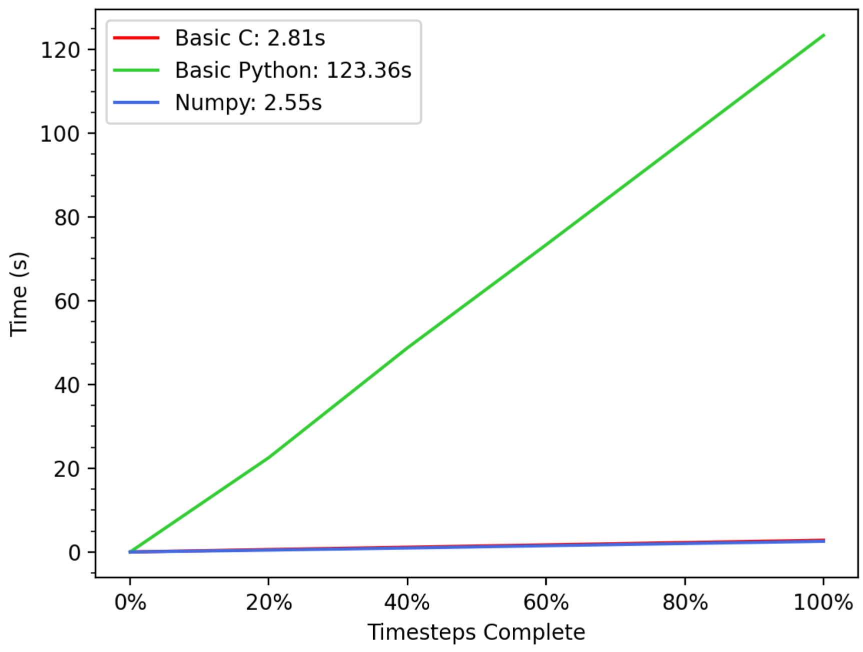


Figure 4.2: Time taken for simulation of 100 bodies over 5000 timesteps.

The `C` and `python` simulations take approximately the same amount of time, but the `numpy` simulation has drastically increased its performance and is now slightly ahead of the `C` simulation, showing that `numpy` can provide a tremendous performance boost when the vectorization of array operations is possible but has a negligible effect when loops cannot be vectorized.

5

Multiprocessing

All most all modern computers are designed with multi-core processors. A multi-cored processor contains multiple processing units, known as cores, that are each capable of running programs independently. This allows computers to more efficiently handle multiple tasks by allocating tasks to each core. Unfortunately, this means that when a computer is instructed to run a program, it will only run on a single core without taking advantage of the full processing power of the multi-core system.

Parallel computing techniques have been devised to spread the task of a single program out to multiple cores. The basic premise is to break the task into smaller chunks which can be computed independently, then recombined once all of the tasks are complete. Each smaller chunk is then run on a different core of the processor, this is known as *multiprocessing*. Multiprocessing can result in massive performance improvements, however it does come with additional overhead, as the tasks have to be distributed to the cores and then be stitched back together for the final result.

5.1 Implementation

In order to adapt this simulation for multiprocessing we need to determine which task can be spread amongst different cores. The main computational effort is calculating the force on each body due to the other bodies. If we consider the force on each individual body as a distinct tasks, they can be run on different cores.

The simulation is modified so that each core is assigned 1 body, and using the positions of the rest, it can compute the force on that body. Once each core has done this, all the results can be combined, the positions and velocities of the bodies are updated and the next iteration can begin.

Multiprocessing is typically split into two types, *shared memory*, where the same memory is shared between the cores and *distributed memory*, where each core has its own memory. Both types have their advantages, shared memory can reduce the memory required as distributed memory can require duplicating data so every core has access. However distributed memory can be more reliable as sharing memory can lead to *race conditions* where one core modifies data while another is using it, leading to inconsistent results. While many sophisticated methods have been created in order to reduce errors in multiprocessing, it is still very important to consider how each of the cores interact with each other and the data that they are using.

For this simulation, each core needs to have access to the positions of every body in order to compute the acceleration. Because the calculation of acceleration does not alter the positions, the shared memory multiprocessing method is appropriate as we shouldn't need to worry about race conditions.

In `C` we use the `OpenMP` library for shared memory multiprocessing. `OpenMP` provides a simple method for paralysing a loop, preceding a `for loop` with the command `#pragma omp parallel for`, runs the `for loop` in parallel, such that each iteration of the loop is run by a different core. This alters our basic algorithm to the one shown in algorithm 3:

Algorithm 3: `OpenMP` implementation

```
for each timestep $t$ do
    #pragma omp parallel for
    for each body $i$ do
        compute $\mathbf{a}_t$
    end
end
```

Where the compute $\mathbf{a}_t$ function calculates the acceleration using the Euler method as before. Because the accelerations for each body are calculated by individual cores, we are no longer taking advantage of the symmetry which will contribute to the change in performance.

For the `python` simulation, we would still like to use the shared memory method, but `python` contains a Global Interpreter Lock (GIL). The GIL is effectively a lock that prevents the `python` interpreter from receiving multiple instructions at once. This has made multiprocessing historically difficult in `python` and many have called for the GIL to be removed to improve this. Despite this, `python` does include the `multiprocessing` library that uses subprocesses to avoid the GIL. While this library does allow multiprocessing in `python` it only allows for distributed memory. This means that data that needs to be shared between cores, such as the positions of bodies must be duplicated, increasing memory consumption. Because the shared data consists of only lists of `floats`, this is not a large concern to us but can effect the performance when cores need access to large data sets.

The implementation in `python` is very similar to `C`, the multiprocessing library allows a loop to be spread across different cores via the `pool` class. Similarly to algorithm 3, each body has its acceleration computed by individual cores in parallel. Because we have already shown using `numpy's` efficient matrix methods provide a performance boost, we continue to use `numpy` when calculating the acceleration of individual bodies. We don't expect the speed up to be as drastic, as the we are not calculating all accelerations at once in one matrix, but array operations are still faster than base `python` loops.

These simulations were run on a 1.1 GHz Quad-Core Intel Core i5 processor. The quad core means at a theoretical maximum we might expect a 4 times speed up of our simulation. However, with the extra overhead required for multiprocessing, we may not see a performance boost that reflects this theoretical increase.

5.2 Results - Multiprocessing

The multiprocessing implementation of the simulation in `python` and `C` gave some surprising results. Not only did we not see a 4 times speed up, these simulations took significantly longer than the implementations without multiprocessing. This simulation took 21.97 seconds for `C` and 525.73 seconds for `python` as shown in figure 5.1.

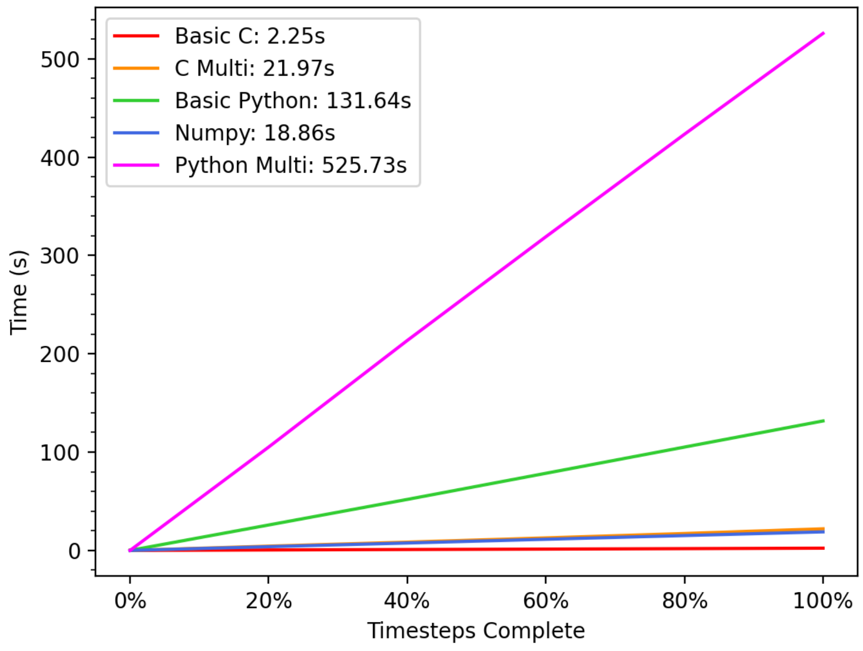


Figure 5.1: Time taken for multiprocessing simulations.

The extreme decrease in performance may seem shocking but can be explained by the overhead in dividing the tasks between cores. In each iteration of the time loop, each core is given the task of determining the acceleration on one body. With only 9 other bodies, this involves a tiny amount of computation on the scale of a cores output. The results in figure 5.1 tell us that the speed up gained from separating these calculations is far outweighed by the added computational overhead of dividing and recombining the results.

In this case, the task of each core is too small for multiprocessing to be beneficial, however if we consider the same system as earlier with $n = 100$ bodies and $t = 5000$ timesteps, computing the acceleration on one body is more computationally demanding. In this system, most of the computation is contained in the inner loops which are run in parallel so we should expect an increase in efficiency despite computing the same number of total iterations. With this alteration, we obtain the results shown in figure 5.2 and we can see the benefits of using multiprocessing.

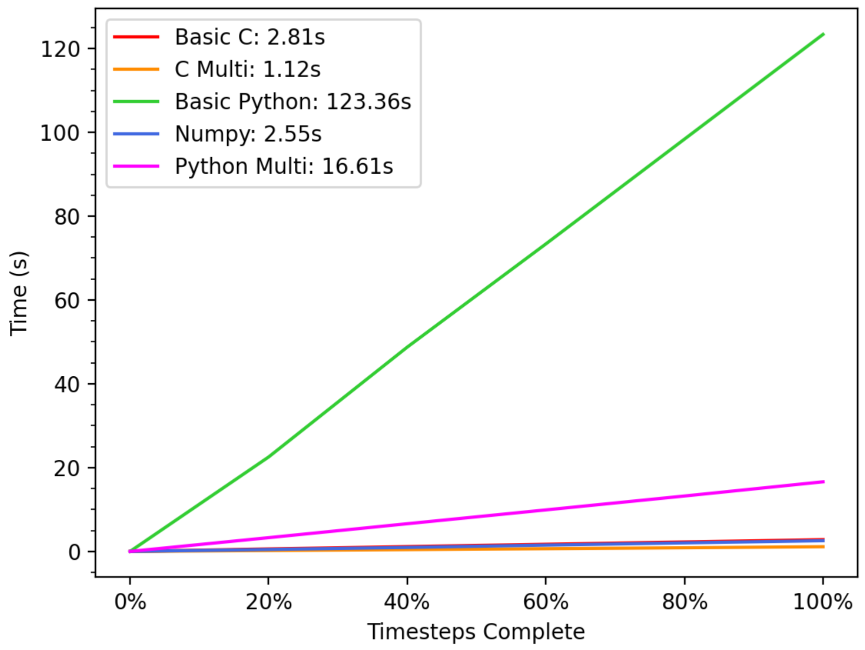


Figure 5.2: Time taken for simulation of 100 bodies over 5000 timesteps with multiprocessing.

With this system, the multiprocessing simulations have now shown a performance boost and the multiprocessing `C` implementation has become the most efficient. The implementation in `python` is still not giving the performance boost we expected due to the outer time loop still being inefficiently done in basic `python` and the additional overhead of duplicating the data. If we now consider an even more contrasting system to the original one, such that almost all of the computation is contained in the inner loops, where we have $n = 1000$ bodies and $t = 50$ timesteps we obtain the results shown in figure 5.3.

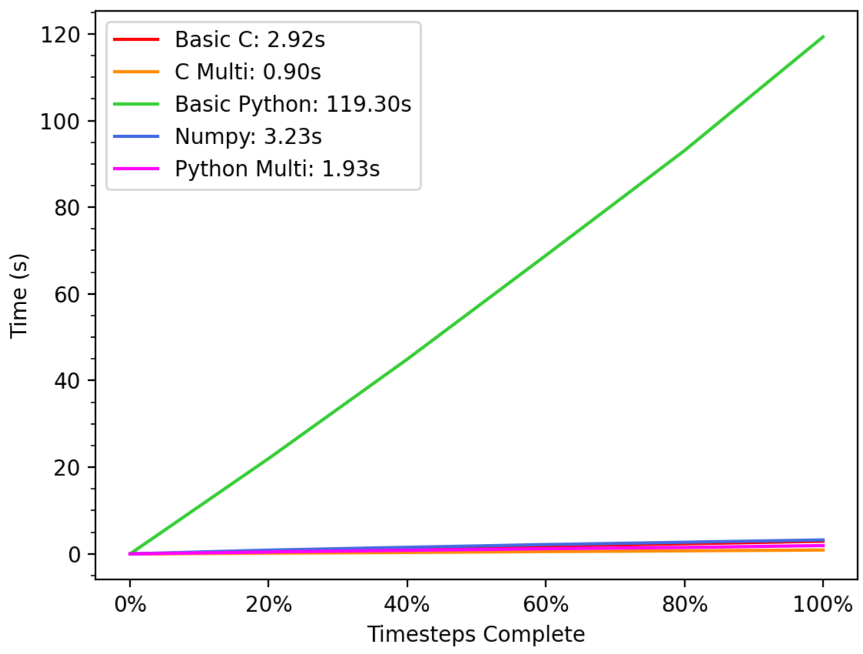


Figure 5.3: Time taken for simulation of 1000 bodies over 50 timesteps with multiprocessing.

Now we can really see the power of multiprocessing. In this system almost all of the computation done is in computing accelerations where the computation is done in parallel, having divided this up amongst different cores produces a significant performance boost and now the simulations taking advantage of multiprocessing are the two most efficient.

Now that each core is doing a sizeable calculation, we can see that the extra computation required for multiprocessing becomes worthwhile. This shows that multiprocessing can be a powerful tool to increase performance but it is not the solution to every system.

6

Comparison of Methods

So far we have implemented a variety of different simulations of asteroid trajectories and seen a variety of results. But now we want to determine which method is truly the best for computing asteroid trajectories.

6.1 Efficiency

We have seen that the most efficient method to use can depend on the number of bodies and the number of timesteps in the simulation. While it is clear the basic `python` simulation is not comparable and will be excluded from this discussion, the other simulations are all capable of being very efficient. To further investigate when each simulation is best, and which should be used for asteroid trajectories, we ran each simulation for a variaty of combinations of timesteps and number of bodies. For each simulation we normalised the time taken by the time taken for the slowest method. This produced figure 6.1.

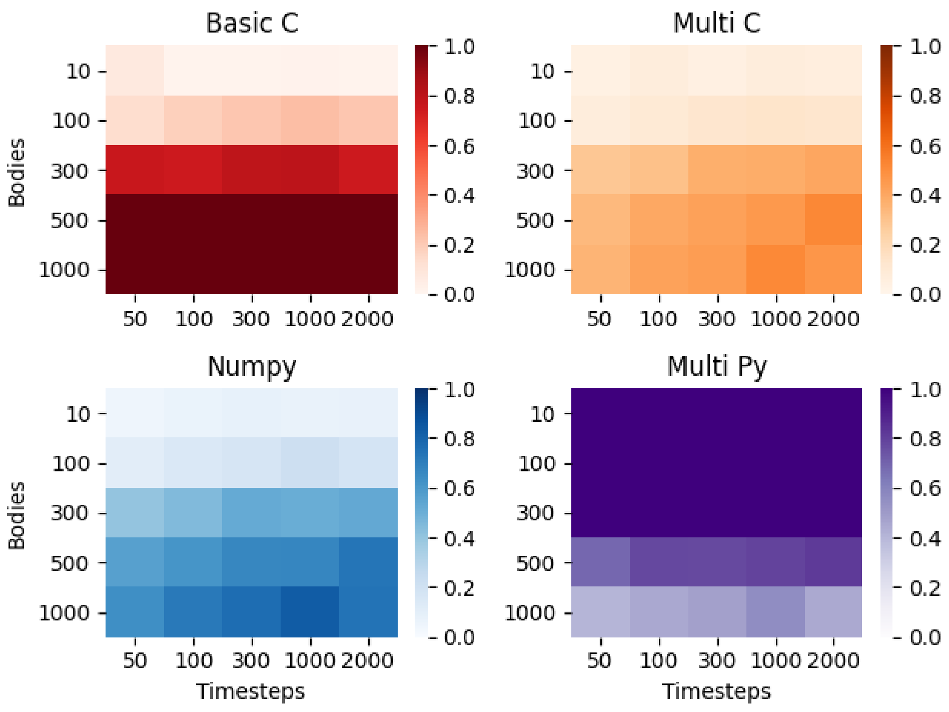


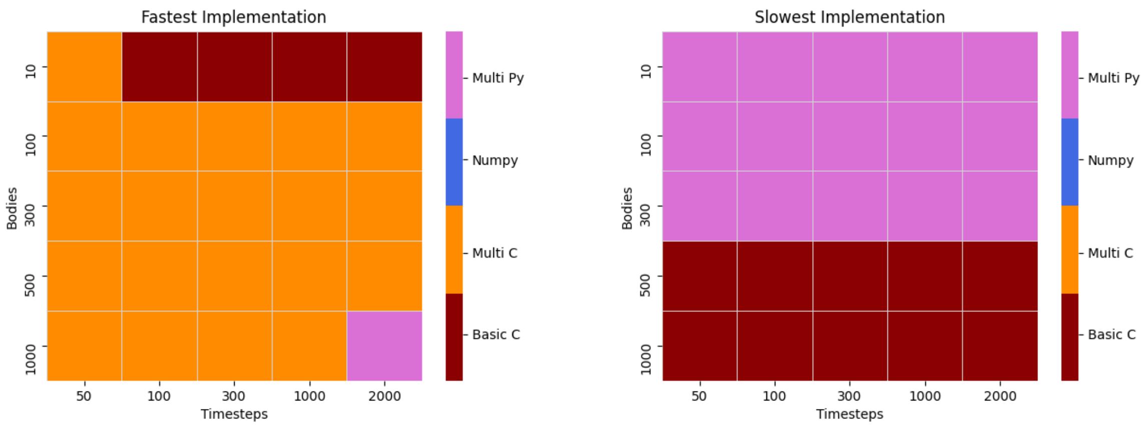
Figure 6.1: Time taken for each implementation for different numbers of timesteps and bodies. Lower values correspond to faster simulations and the times have been normalised by the maximum time in each test.

These figures show clear regions where different methods shine. The basic `C` implementation, shown in red, scales well as the number of timesteps are increased, but becomes very inefficient when the number of bodies become large. The other 3 implemnetations all have methods to enhance the performance for larger numbers of bodies, `numpy` uses efficient matrix methods, multiprocessing `C` splits the computation up for different cores, and the multiprocessing `python` does both, whereas basic `C` continues with simple loops.

The plot in purple in figure 6.1 is the multiprocessing `python` implementation. For low numbers of bodies this method is very slow due to the outer loop through time. Once the number of bodies becomes larger, the decrease in performance due to the time loop, is outweighed by the performance boost from multiprocessing and matrix operations.

The other two implementations, multiprocessing `C` shown in orange, and `numpy` shown in blue, seem to be viable in all circumstances. Earlier results showed that `C` implementations were faster than `numpy` for low numbers of bodies. Because the multiprocessing `python` simulation is very inefficient for low numbers of bodies, and these results have been normalised, figure 6.1 does not capture `C's` advantage over `numpy` for small numbers of bodies.

To more specifically see which implementation is the fastest in each scenario, figure 6.2 shows which implementation was the fastest and which was the slowest in each case. It is now clear that the `C` methods are best for low numbers of bodies, and the multiprocessing `C` implementation is fastest in most cases. The lower right square with $n = 1000$ and 2000 timesteps shows that multiprocessing in `python` with efficient array operations can overtake `C` however the difference between these two was minimal.



(a) Fastest implementation (b) Slowest implementation

Figure 6.2: Comparison of the slowest and fastest implementation for different numbers of timesteps and bodies.

Figure 6.2 also shows which method was the slowest for each simulation. Here the clear distinction for different numbers of bodies becomes obvious for `C` and multiprocessing `python`. It also shows that `numpy` may never be the fastest implementation, it is also never the slowest.

From these comparisons, along with earlier results, we can deduce that the multiprocessing implementation in `C` is the best all purpose method for simulating asteroid trajectories. In almost all cases it is the fastest implementation, and for the few cases where it is beaten, it is by a small margin.

It should also be noted the cases where `python` implementation exceed `C`, are when `numpy's` matrix multiplication shines. `Numpy` is a library that is written in `C` or `C++` and using BLAS. Since `numpy` is written in `C`, it must be possible to capture the performance boost of `numpy` for matrices in `C` with a more sophisticated simulation.

6.2 Simplicity

While it is clear that implementations using `C` are more efficient in most cases, efficiency should not be the only consideration. The implementations in `C` were far more difficult than the `python` implementations. Due to `C's` memory allocation, static typing and reduced readability these implementations took far longer and resulted in many more bugs. `Python` was created with simplicity and readability in mind, resulting in a far simpler coding experience. There is a very good reason that many introductions to programming are now done in `python`.

The same can be said of multiprocessing. We have seen that multiprocessing is capable of incredible performance boosts in the right circumstances, but the added complexity can be daunting. When considering more advanced components of multiprocessing such as race conditions, multiprocessing can result in bugs that are extremely difficult to find and fix. Multiprocessing can also be very difficult to get working properly, while the `python` implementation was relatively straight forward due to `python's` simplicity, many problems arose during the `C` implementation. This included issues with the compilers applicability with multiprocessing and the install directory of the `OpenMp` library which could be to much for a beginner programmer to deal with. If the final results only see a 5% speed up, multiprocessing may more hassle than it is worth.

Figure 6.2 showed us that `numpy` was never the fastest method, but was also never the slowest. The implementation using `numpy` was one of the simplest. For smaller scale simulations, `numpy` may be the best balance between efficiency and difficulty. We also mentioned that the efficiency benefits of `numpy` are possible to reproduce in `C`, however this does not consider the ease of use of `numpy`. By being a simple `python` library, `numpy` provided simple and well documented access the the highly optimized `C` codes that it is built from. Many scientists and programmers have contributed to `numpy` to squeeze in every possible performance boost. Outperforming `numpy` in situations where matrix operations are key is possible, but it is far beyond the capabilities of the average scientist in any reasonable amount of time.

6.3 Reality

In reality, not all programs need to be executed at peak of efficiency. This had lead to countless programs being implemented in `python` and using `numpy` to get the performance boost of compiled languages without the hassle.

However, for simulations of asteroid trajectories, the enormous computational cost and requirement for as early as possible prediction mean that ease of use is sacrificed for pure efficiency. Modern simulation of asteroids are run on state of the art High Performance Clusters (HPC) using compiled languages such as `C` or `fortran` taking advantage of the best known shared and distributed multiprocessing methods, the most advanced forms of BLAS and the most efficient computations of body accelerations currently known. In addition to the computational complexity shown so far, real simulations must include extra complexity. On the scale of the solar system, the Earth is tiny, any small deviation of the trajectory can be the difference between an impact with Earth, or safely floating past. Real Simulation must include every body in the solar system, including moons and other asteroids, they must also be run for an enormous amount of timesteps for long term predictions. These simulations must also consider the material and density of the asteroid, any rotations of the asteroid and alterations of the calculations due to general relativity. Additionally the coordinates and velocities of asteroids are not known to infinite precision, so many simulations are run of the same asteroid with slightly different initial parameters. All of this adds up to an enormous amount of computation required and any tiny improvement in efficiency could be the difference in humanity having enough time to deflect an incoming asteroid.

7 Conclusion

An asteroid colliding with Earth may be humanity's greatest threat. Currently, preventing an asteroid colliding with Earth would not be possible, but given enough time we would be able to alter its trajectory and save our planet. Searching for nearby asteroids remains an ongoing task, and once found, determining their trajectories remains an enormous computational task. In order to give humanity its best chance, an incoming asteroid needs to be found as early as possible. In order to efficiently simulate asteroids, not only do you need the best scientific methods and the most advanced computers you also need to implement those methods as efficiently as possible.

We have seen that compiled languages have a significant performance advantage over interpreted languages and that under certain circumstances, multiprocessing can lead to enormous performance boosts. The simulations done using multiprocessing in `C` were fastest in almost all cases and is the clear choice when efficiency is the most important factor. Real asteroid simulations require that efficiency is the top priority and any other factors are negligible compared to efficiency. This leads to super efficient programs written in compiled languages like `C` or `fortran` using advanced multiprocessing methods.

For asteroid simulations, efficiency is key. However, for many other applications of programming efficiency is not as important. It may be that readability and flexibility are far more important in which case languages like `C` may not be the answer. Instead a less efficient language like `python` that excels in simplicity and ease of use may be the answer, especially with the creation of more and more `python` libraries like `numpy` that provide efficiency.

There is a good reason more than one programming language exists. There is no best language or best method of implementation. Efficiency can be extremely important, and in the case of asteroids may save lives. But it is not the only thing to consider. The most important thing to remember when choosing a programming language or method, is not which one is best, it is which one suits the task at hand the best.

Appendix A

Reflection

This project was self motivated with the aim of gaining a better understanding of commonly used techniques for improving efficiency in scientific programming. Before beginning the project I had a very basic understanding of why compiled languages are faster than interpreted and why `numpy` improved the speed of `python`. These are both far clearer now, and I have a better understanding of just how much of a speed up can be expected and under which circumstances the added efficiency will be worth the extra effort.

I also had a rudimentary understanding of multiprocessing, but now I understand how and when multiprocessing should be implemented. For my research I will be making use of the multiprocessing library `mpich`, before this project I viewed this as a black box. While I have not used `mpich` directly here, `mpich` is the distributed memory multiprocessing library for `C`, so I am now aware of the differences and similarities to `OpenMP` which was used here.

While the implementation of these methods was done on a simple system of ODE's, having practise with these libraries, and an understanding of when they are beneficial will make their applications to other more complicated systems much simpler.

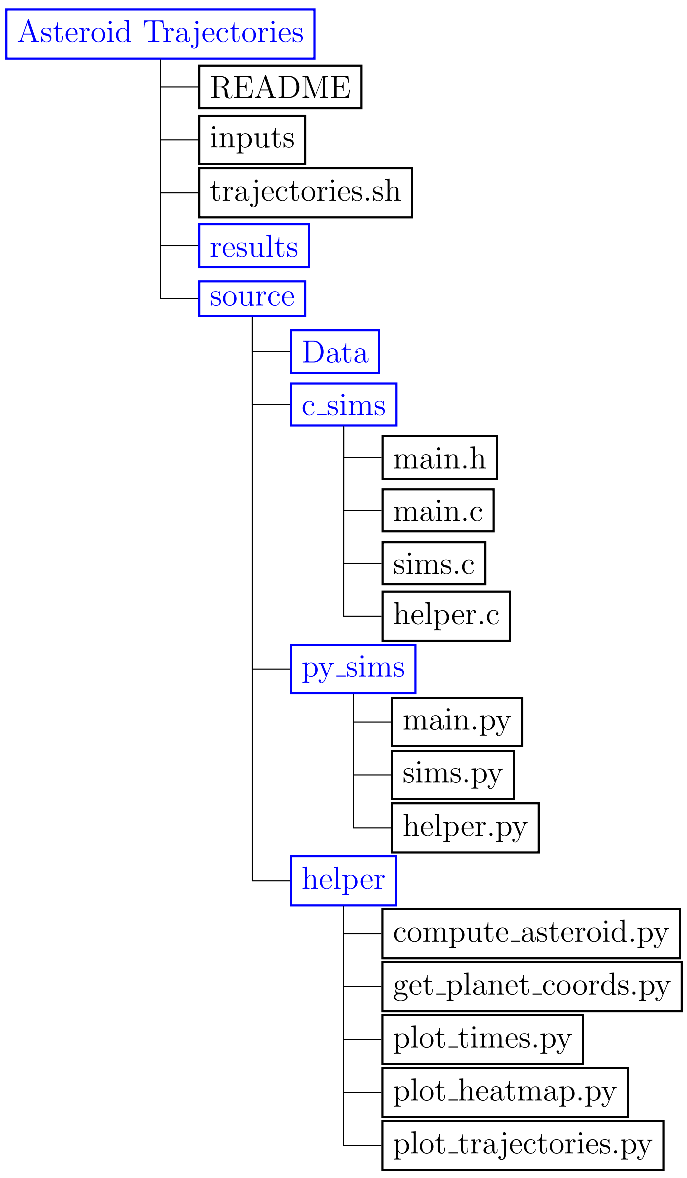
Working with a variety of different simulations in different languages has also given me experience working on a larger project, which was managed using `git`, in the following appendix I discuss the structure of the code and the use of `Bash` scripting to run the code. `Bash` scripts are often used for numerically intensive tasks on super computers, such that a single `bash` script can be run that will run multiple different programs and control the outputs, while the scientist gets on with their day. I believe my research will lead me down this route, so having a understanding of `bash` scripting beyond just copying shell commands will be invaluable.

Should other students taking this subject be more interested in the efficiency side of scientific computing, this project could be for them. As the basic premise of simulating planets with Newton's Laws is relatively straightforward, it allows most of the project to focus on the techniques that can be used to increase the efficiency. This system has also been able to demonstrate the incredible differences for the various implementations. While we have implemented a toy model of the solar system, the n-body problem is found in numerous areas of science, and having a strong understanding of the basic problem will make further research much more streamlined.

Appendix B

Source Code

The code used during this article can be found at `https://github.com/Syndrius/comp90072_project` and can be installed via `git clone`. All the simulation were done on a 2020 MacBook Air with 1.1 GHz Quad-Core Intel Core i5 processor and 16 GB 3733 MHz LPDDR4X RAM. The structure of the software is shown below:



The two key files in this are `inputs` and `trajectories`. The `inputs` file should be the only file that needs to be interacted with to run the simulation. The `input` file contains boolean options for which of the simulations should be run, and the different scenarios. This includes running the simulations with the real planetary data or with the randomly generated data. It also contains the option for creating the heatmap data shown in the comparison section. Once the appropriate inputs are chosen, the file `trajectories` will run the appropriate simulations and save the results into the `results` folder.

Bibliography

[1] *Earth woefully unprepared for surprise comet or asteroid, Nasa scientist warns.* en. Section: Science. Dec. 2016. URL: `http://www.theguardian.com/science/2016/dec/13/space-asteroid-comet-nasa-rocket` (visited on 05/31/2021).

[2] Ampomah Ernest, Ezekiel Mensah, and Abilimi Gilbert. "Qualitative Assessment of Compiled, Interpreted and Hybrid Programming Languages". en. In: *Communications on Applied Electronics* 7.7 (Oct. 2017), pp. 8–13. ISSN: 23944714. DOI: `10.5120/cae2017652685`. URL: `http://www.caeaccess.org/archives/volume7/number7/kwame-2017-cae-652685.pdf` (visited on 06/02/2021).

[3] Adam Ginsburg et al. "astroquery: An Astronomical Web-querying Package in Python". In: *The Astronomical Journal* 157 (Mar. 2019), p. 98. ISSN: 0004-6256. DOI: `10.3847/1538-3881/aafc33`. URL: `http://adsabs.harvard.edu/abs/2019AJ....157...98G` (visited on 06/02/2021).

[4] https://jpl.nasa.gov. *NASA Analysis: Earth Is Safe From Asteroid Apophis for 100-Plus Years.* en. URL: `https://www.jpl.nasa.gov/news/nasa-analysis-earth-is-safe-from-asteroid-apophis-for-100-plus-years` (visited on 05/31/2021).

[5] https://jpl.nasa.gov. *NASA Jet Propulsion Laboratory (JPL) - Robotic Space Exploration.* en. URL: `https://www.jpl.nasa.gov/` (visited on 06/02/2021).

[6] Lubomir Ivanov. "The N-body problem throughout the computer science curriculum". In: *Journal of Computing Sciences in Colleges* 22.6 (June 2007), pp. 43–52. ISSN: 1937-4771.

[7] *OUR MISSION.* en-US. URL: `https://b612foundation.org/our-mission/` (visited on 05/31/2021).

[8] Susanne Pfalzner and Paul Gibbon. *Many-body tree methods in physics.* eng. Cambridge ; New York : Cambridge University Press, 1996. ISBN: 978-0-521-49564-6. URL: `http://archive.org/details/manybodytreemeth00pfal_902` (visited on 06/02/2021).

[9] Alexander Roghult. *Benchmarking Python Interpreters : Measuring Performance of CPython, Cython, Jython and PyPy.* eng. 2016. URL: `http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-183547` (visited on 05/30/2021).

[10] V Rokhlin. "Rapid solution of integral equations of classical potential theory". en. In: *Journal of Computational Physics* 60.2 (Sept. 1985), pp. 187–207. ISSN: 0021-9991. DOI: `10.1016/0021-9991(85)90002-6`. URL: `https://www.sciencedirect.com/science/article/pii/0021999185900026` (visited on 06/02/2021).

[11] *The Effect of Warning Time on the Deflection of Earth-Impacting Asteroids.* en-US. May 2019. URL: `https://b612foundation.org/the-effect-of-warning-time-on-the-deflection-of-earth-impacting-asteroids/` (visited on 05/31/2021).