Damien De Courcy

[Email address]

Abstract

This report looks at basic n-body integration methods and applies them to a model of the Solar System to predict orbits for up to N number of days. It compares the algorithm implementations in Haskell and C++ for run times over various simulation parameters, such as logging energy changes on every step, no logging with final data output, and printing to console at every step. I also explore how energy changes in the system are direct results of the integration method used.

Integrating the Solar System

A look at n-body integration methods in C++ and Haskell

Contents

[1. Introduction 2](#_Toc95395092)

[2. N-Body simulations 2](#_Toc95395093)

[3. N-Body simulation in C++ 11](#_Toc95395094)

[4. N-Body simulation in Haskell 15](#_Toc95395095)

[5. Comparisons 24](#_Toc95395096)

[6. Discussion 31](#_Toc95395097)

[7. References 33](#_Toc95395098)

[8. List of Figures 34](#_Toc95395099)

# 1. Introduction

This report looks at basic n-body integration methods and applies them to a computational simulation of the Solar System to predict orbits for up to N number of days. The integration methods used are the Euler method (see 2.4.1) and the Midpoint Method (see 2.4.2).

Computer simulations are used to predict behaviors of systems in response to other systems or stimuli. They use a mathematical and computational description of a real system placed within a mathematical model that simulates the stimuli on the system over time. This in turn calculates how the system itself reacts and allows us to make accurate predictions for real world systems of the same nature.

There are various options when deciding which language to write our simulation in. Various computer languages exist, and span from imperative languages to functional languages, with some languages having features from both. In an imperative language, all computational behavior, as well as model algorithmic behavior, is defined. This means we can run algorithms with side effects, which can be both a helpful feature and a detriment, depending on our intentions within the algorithm. In functional languages, only the algorithms are defined, and no side effects can be administered. This process is known as referential transparency. A functional language, then, suits algorithms that have a need for accuracy.

Computer simulations are often built in a language best suited to the computational domain. As explored in section 2.2, our simulation has a high complexity (slow runtime), however we can manipulate side effects in our algorithm to speed it up. Thus, writing the simulation in and imperative language such as C++ stands out as a default choice. However, mathematical models should be as accurate as possible, and this points us towards a functional language. Thus, writing the simulation in a language such as Haskell is also an option.

For the purposes of this paper, I write the simulation in both languages and compare the runtime and prediction results.

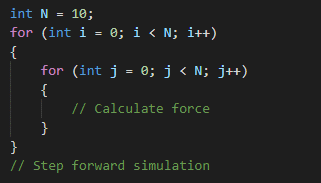
# 2. N-Body simulations

Physicists use N-body simulations to model systems of particles, usually under the influence of physical forces (N-body Simulation, 2022). In each step of the simulation every particle is calculated against every other particle before stepping forward the simulation.

For the purposes of this paper, we use an N-body simulation to build a model of our solar system, predicting the orbits for each planet. Each planetary body represents a ‘particle’, and the force calculated between them is gravity. The gravity is then used to calculate the velocity and position of a planet in an instance.

The simplest way to represent particles in an N-body simulation is in a fixed array the size of **N**, where **N** can be a prescribed number, or captured from user input. In our simulation, as explored in section 3, our **N** is ten. We arrive at this number because we have nine planets, Pluto, and the sun.

The simplest algorithm for an N-body simulation is a nested for loop. Each time we step the outer for loop forward, we calculate the force between the particle at index i, and every other particle.



*Basic N-Body Simulation Algorithm.*

Given a nested for loop iterating over an array, this algorithm has the complexity of O(N^2). When assessing complexity of array algorithms, the complexity is often listed as O(i(**N**)), where i(**N**) represents the functional iterations over the array. Time complexities are listed by worst case scenarios, with the basics in ascending complexities being as below.

**Algorithm complexities:**

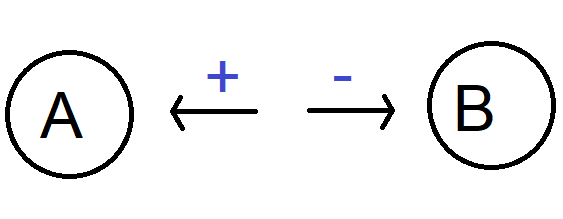
* O(1) : No iteration
* O(log(N)) : Iterate through continual halving’s of the array
* O(N) : One iteration of the array
* O(N log(N)) : Iterate through one iteration, with another iteration through continual

halving’s of the array

* O(N^2) : Iterate through an array once, and once for each item in the array.

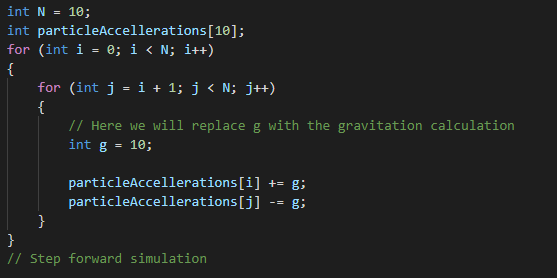
While we cannot reduce the algorithms complexity, we can manipulate side effects to lower the runtime of the simulation, albeit quite negligibly. We do this by adding a cumulative function to the calculation of the forces, calculating only once for each particle, henceforth called a body.

When calculating gravity between two bodies, the direction of the force on each body must be considered. In physics calculations, this direction is denoted with the sign of a numerical value of a force. Subsequently, for body A the resultant force will be in a positive direction. For body B the force will be negative:



*Resultant gravity acting on each body.*

To factor this in cumulatively, we must write the gravitational force back to body A positively, and body B negatively. In C++, this is as simple as writing it back to the array at the bodies index, where the array at index of i is body A and index of j is body B:

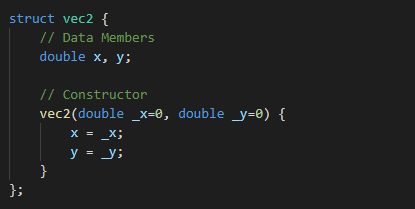


*Modified core algorithm.*

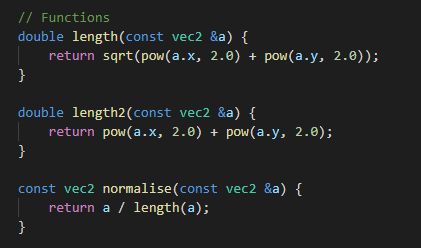
However, in Haskell arrays are represented using Lists and are immutable. That is, we can’t write back the resultant gravity to the original array. This presents us with considerable difficulties in redesigning the algorithm, as explored in section X.X.

When representing a bodies acceleration in 2D, we break the acceleration down to a 2D vector of its cartesian (x, y) components. However, the calculation for gravity requires a single numerical value for the distance between two objects. Subsequently, the first step of the gravitational calculation is to find this value.

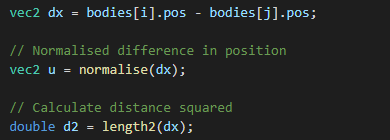
To do this, we subtract the position of body B from body A, then normalize the length to a single vector value using trigonometry as found overleaf.



*A 2d vector of its x and y components.*



*Trigonometry functions for finding and normalizing a single value for a 2d vector.*



*Applying the above code.*

Once this is done, we can now apply the calculation for gravity:

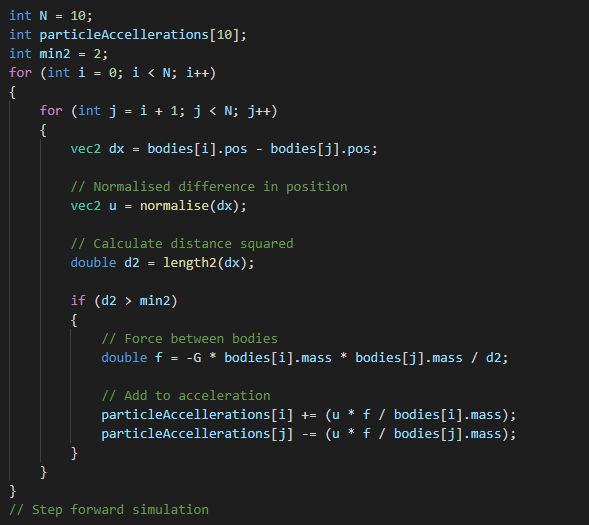
F = -G x mass(body 1) x mass(body 2) / (distance between bodies)

Where:

G = the gravitational constant

However, this equation raises the problem of the inverse square law. In simple terms, the closer two objects are the closer to infinity the force between them becomes. While this is not possible in the physical world it does present the computational restriction that, while unlikely, should two bodies occupy the same point the simulation will calculate for infinity and subsequently break.

To accommodate for this, our algorithm must check two bodies are not in the same point at the time of calculation. We do this by setting an arbitrary distance that must exist between bodies before calculating the forces. Our resulting basic algorithm is as follows.



*Our basic O(N) algorithm for the Solar System simulation in C++.*

“Integration method” refers to a formula applied to points on a curve to predict the positions of future points on the same curve. In our simulation we have two separate curves to predict; the position of a planetary body based on its velocity, and the velocity of the planet based on its acceleration. In both curves, the X axis represents the timestep in the simulation, the Y axis represents the value of what we are predicting (velocity or position).

In the simulation, we explore the two simplest methods for integration: the Euler and Midpoint methods.

The Euler method is the most basic integration method available. Its formula is:

y(at t + ts) = y(at t) + ts(dy/dx)

In simple terms:

y(at t) is the value of y at our current timestep (where x/t is our timestep)

y(at t+ts) is the value of Y at the next time step

ts is our change between timesteps

dy/dx is our function for the slope at the current time

A basic walk through of the method can be done as follows, with the initial conditions of:

x/t = 0

y = 1.5

dx/dy = 3x – 2y

ts = 1

To find the next timestep, or x = 1, we follow the formula remembering that t is the time, which is the value of x.

Step 1, replace t for our current time, and ts for our timestep:

y(at 0+1) = y(at 0) + 1(dy/dx)

Step 2, replace Dy/Dx for our function:

y(at 1) = y(at 0) + 1(3x – 2y)

Step 3, pass the values of x and y into the function:

y(at 1) = y(at 0) + 1(3(0) – 2(1.5)

Step 4, solve the function:

3(0) – 2(1.5) = -3

y(at 1) = y(at 0) + 1(-3)

Step 5, solve for y(at 1)

y(at 1) = 1.5 + (-3)

y(at 1) = -1.5

The Midpoint Method expands on the Euler Method. It calculates an over approximation and uses it to calculate a half-step. It then calculates the full step from the result of the value at the half-step. The formula is:

S1 = f(x(at t), y(at t))

S2 = f(x(at t) + (ts/2), y(at t) + (ts/2)(S1))

y(at t + ts) = y(at t) + ts(S2)

In simple terms:

S1 is a point calculated straight from the Dx/Dy function applied to our starting x and y values

S2 is a point calculated from that function applied to the starting x value + half the timestep, and the starting y value + half the timestep multiplied by S1

y is the final point calculated by adding the initial value of y to S2 multiplied by the timestep

A basic walkthrough of the formula can be done as follows:

x/t = 0

y = 1.5

Dx/Dy = 3x – 2y

ts = 1

Step 1, find S1

S1 = 3x – 2y

S1 = 3(x at t) – 2(y at t)

S1 = 3(0) – 2(1.5)

S1 = -3

Step 2, find S2

S2 = 3x – 2y

S2 = 3(x(at t) + (ts/2)) -2(y(at t) + (ts/2)(S1))

S2 = 3(0 + ½) -2(1.5 + (1/2)(-3))

S2 = 3(1/2) – 2(0)

S2 = 1.5

Step 3, find y(at t + ts)

y(at t + ts) = y(at t) + ts(S2)

y(at 0 + 1) = y(at 0) + 1(1.5)

y(at 1) = 1.5 + 1.5 = 3

Once we have calculated the force on an object in an instance, we combined the above methods with physics calculations to predict our curves. The change in velocity is one such curve we must predict over a timestep, to avoid measuring it at every possible instance.

Using the Euler method, we can see that velocity should replace Y(at 0), and the change in velocity can be substituted for the physics equation for finding velocity based on acceleration applied to it.

To find an objects velocity, the physic equation is simply:

vel1 = vel + acc/time

Thus, the Euler equation becomes:

vel1 = vel + (acc/time)(timestep)

The same is true for position, the other curve we are estimating, where the physics equation is:

pos1 = pos + change in position/change in time

Thus, the Euler equation becomes:

pos1 = pos + (change in position / change in time)(timestep)

A problem faced with estimates of unknown curves is that we can’t be certain our predictions are correct. Subsequently we face the problem of accounting for accuracy in our model. Thankfully, we can use physics to compensate for this and estimate an accuracy.

The law of the conservation of energy states that in a closed system energy cannot be destroyed or created (Conservation of Energy, 2022). This means that a systems energy will always equal itself, unless acted upon from the outside. For example, a rock spinning in the vacuum of space will spin forever until it collides with another object.

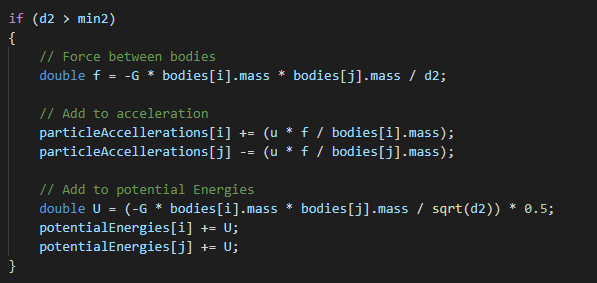
In a model such as our solar system our “energy” is broken into two types and can be calculated by summing both (Playne, 2008). The first is the total kinetic energy of the system. This can be found by finding the kinetic energy of each body and summing the results. The formula for such is as follows:

ke = vel^2 \* mass \* 0.5

The second energy is the total potential energy of the system. In simple terms, unrealized force on an object. For example, lifting something gives it the potential energy of gravity until it is released. Once released this energy is ‘realized’ into kinetic energy causing it to fall. While we calculate the force between bodies, we can also calculate the potential energy of the bodies. The formula is as follows:

pe = -G \* mass1 \* mass2 / distance

Because this energy is shared between bodies, we halve and add it to both. In code, this is as follows:



*Adding potential energy calculations to our algorithm.*

For each step of our simulation the total energy should be the same as its initial value. Our solar system model has exactly ten bodies and is thus counted as a closed system. The only energy the bodies move by is the gravitational force they exact on each other, which varies between kinetic and potential energy based on the positions of the planets. Thus, no energy should be added or removed from the model.

# 3. N-Body simulation in C++

To expand the core algorithm into the Euler method accurately we capture the positions, accelerations, velocities, and energies following these steps:

1. Initialize the simulation
2. Calculate the kinetic energy of all bodies in their starting conditions
3. Until MAXSTEP number of loops, update the simulation
4. Print the final energy calculations and positions of the bodies to a file

For the values to be correct, the following order of operations is required in the update loop:

1. Calculate the accelerations of the bodies in the instance
2. Calculate the potential energy of the bodies in the same instance during this phase
3. Sum the kinetic energy of the bodies from their last update with the potential energy of the planets in this update
4. Update the position of the bodies based on the Euler estimation of the last update velocity over the timestep
5. Update the velocity of the bodies based on the Euler estimation of the current acceleration over the next timestep
6. Return the total energy to the simulation for energy change calculations

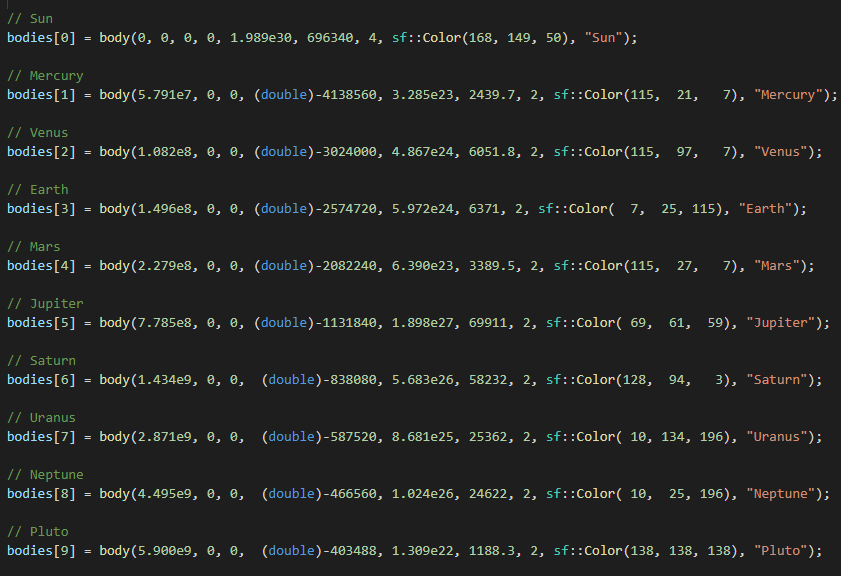
The first step of setting up the simulation is to find the values for each of the bodies and map them to units that work with our timestep. Because our simulation runs at a time step of one day, this means all distances/time values must be converted into km/day values.

The gravitational constant, then, becomes:

6.67408e-11 \* (86400.0\*86400.0) / (1000.0\*1000.0\*1000.0);

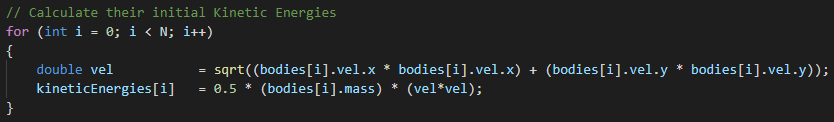
Our bodies then must have their distance from the sun in km, and their velocities worked out to km/day. The resultant values are shown in code overleaf where the values read:

*position x, position y, velocity x, velocity y, mass, radius, render radius, color, and name:*



*The bodies of our simulation at their initial positions and velocities.*

The main initialize function must also do the initial calculation of the kinetic energies:



*Initial capture of the kinetic energies.*

We now have a working Euler model of the solar system as seen in submission 01.Euler.C++.

To convert the Euler algorithm to a Midpoint algorithm we first need to identify the physics equations from the midpoint formula.

Remembering we must estimate for both velocity and position, we apply these to the midpoint method in two places such that

S1 = f(x(at t), y(at t))

S2 = f(x(at t) + (ts/2), y(at t) + (ts/2)(S1))

Simplifies into both:

pos1 = pos + vel \* ts \* 0.5

vel1 = vel \* acc \* ts \* 0.5

And:

y(at t + ts) = y(at t) + ts(S2)

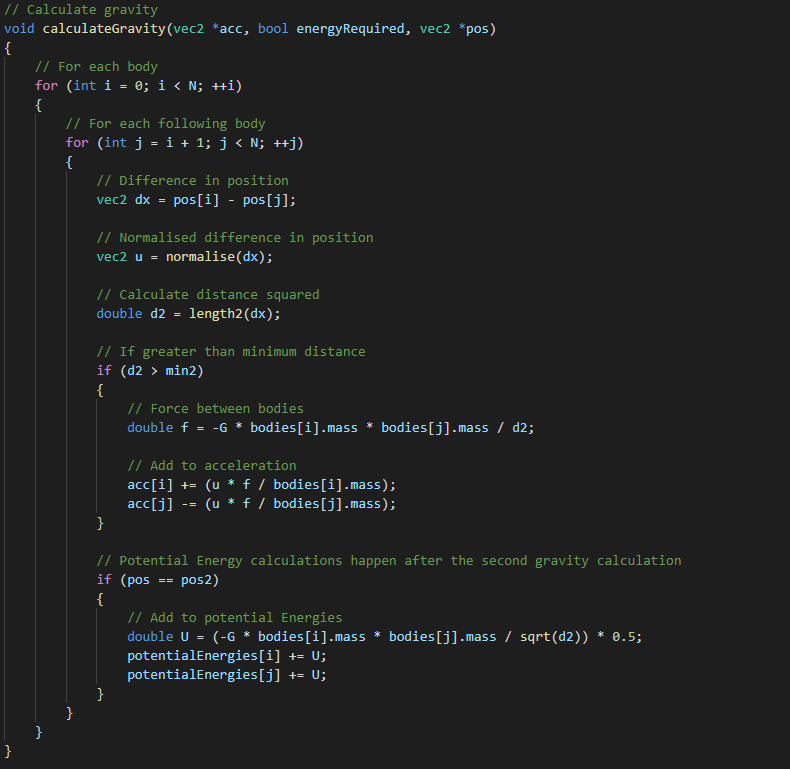
Becomes:

pos = pos + vel2 \* ts

vel = vel + acc2 \* ts

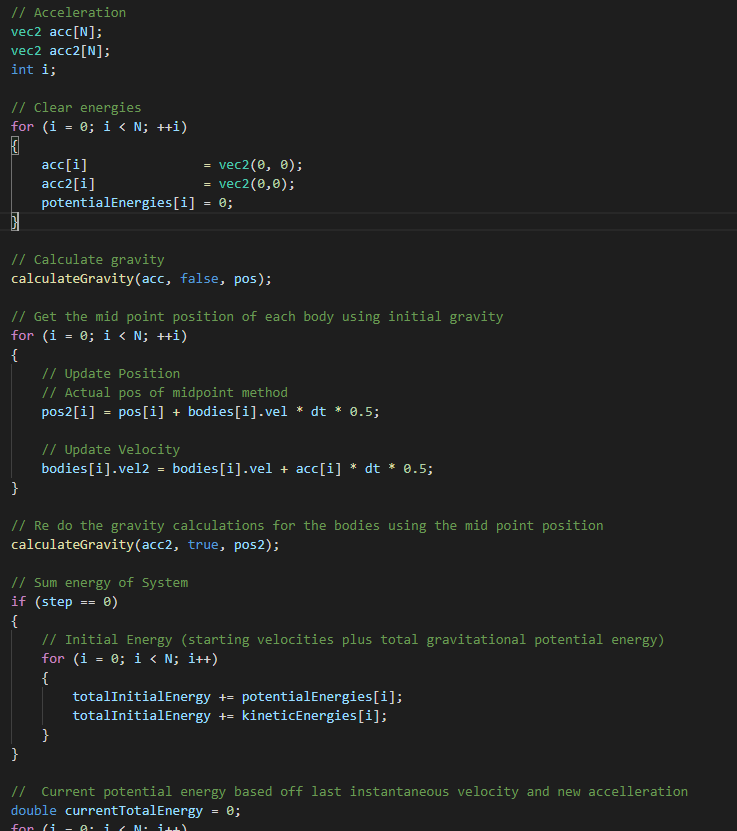
However, a keen eye will notice we have an unaccounted-for variable: acc2.

Acc2 is calculated from the midpoint found in S2, and the simplest way to do this is to abstract the code out to its own function, to be called twice in the update loop. When calling this code, we pass in either the actual acceleration array or the midpoint acceleration array depending on which stage of the update loop we are in. In the second call of this function, we must also calculate the potential energy of the system. The subsequent function looks as follows:



*Calculating the gravitational force between bodies as a callable function.*

This is called twice from the update loop as follows:

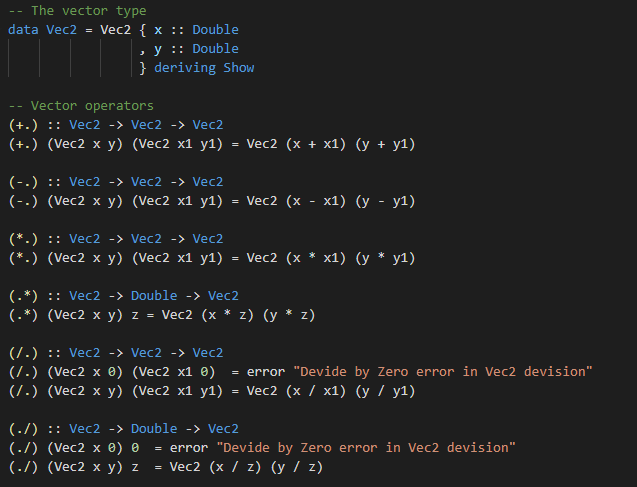


*Snippet of update loop calling the gravitational calculation twice as per the midpoint method.*

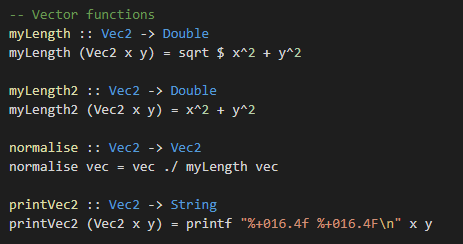
We now have a working midpoint simulation of the solar system, as found in submission 02.Midpoint.C++.

# 4. N-Body simulation in Haskell

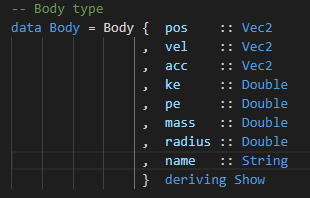
Setting up the simulation in Haskell is much the same as C++. We have a custom datatype for both a body and a vec2:



*Vec2 data type and operators.*

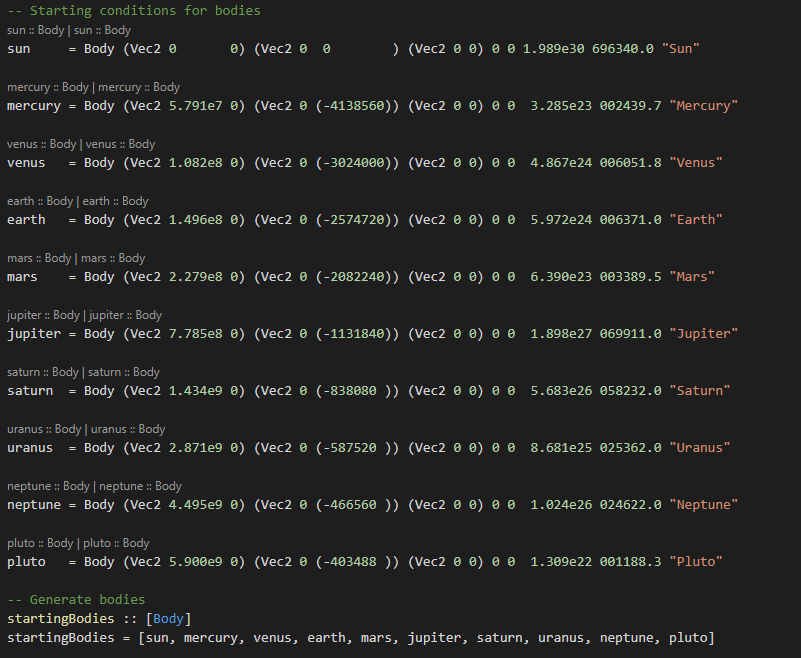


Vec2 physics functions.



*The body datatype.*

We set up our starting conditions with the same values as the C++ model:

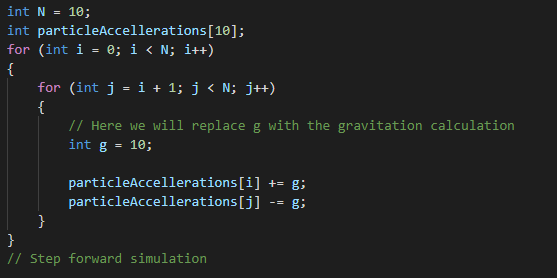


*Starting conditions for the simulation in Haskell.*

And we run the same steps as the C++ simulation as described in section 3.

There will be two major difference between the languages: The evaluation methods of each language, and the Haskell implementation of the core algorithm for interacting the bodies with each other.

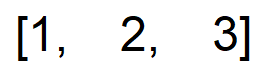
When revisiting the core algorithm, we can see that we must design a cumulative nested for loop in Haskell:



*Revisiting the core algorithm in C++.*

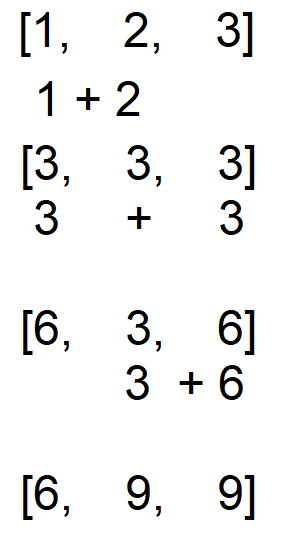
However, in Haskell arrays are represented using Lists and are immutable. That is, we can’t write back the resultant gravity to the original array. Subsequently we must write an algorithm to fold through a list once, producing a new list with the cumulative results.

The first step in designing the Haskell algorithm is to ensure it can be done. To do this, a list of integers representing either acceleration or velocity is used:



*List of integers.*

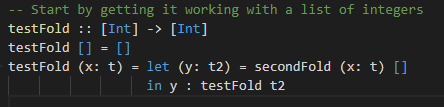
To prove they can be cumulatively folded into a new list a simple operator is chosen, and the algorithm designed using it. The operator and values can later be substituted out however the resultant algorithm should follow this pattern:



*Process of folding the list cumulatively.*

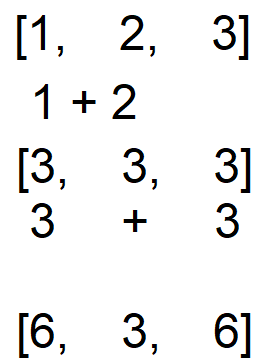
First, we visualize this process using a simple + operator between two integers. The integers will later be substituted for bodies, and the + operator for a physics calculation of gravitation between two bodies. For now, we need two functions.

The first function we will call testFold and is between a map and a fold. It replicates a cumulative double for loop by calling a fold on a list, then recursively calling the fold on the tail of the result while appending that answer to the head of the initial result:



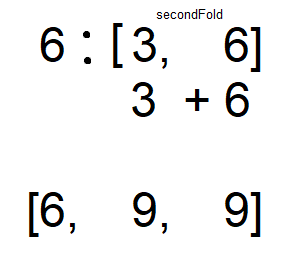
*The map/fold function.*

The second function we will call secondFold. It is a fold that takes a list and folds it into another list that contains one finished item (cumulatively added to every other item in the list), and the cumulative result of each other item in the list. Using our integer list from above, this fold will handle this step:

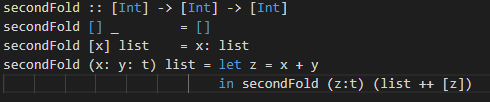


*secondFolds order of operations.*

This resultant list will be passed back to the testFold, which will store the finished item and append it to a recall of secondFold on the cumulative result stored in the tail. Essentially, this step:

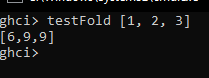


*The recursive call of secondFold by testFold.*



*The secondFold function of the map/fold function.*

Running these functions gives us the expected result:



*testFold run in GHCi.*

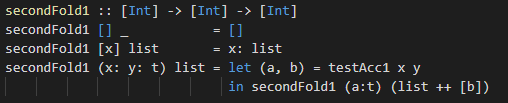
Now that we have proven a cumulative fold is possible, we test to see we can apply a positive result to the left integer, and a negative one to the right integer. In this way, we prove that we can add an acceleration force on the left and subtract one on the right.

First, we need a new function we will call testAcc1. This will later be replaced by the physics calculation for gravity between two bodies. For now, we keep using the simple + operator on integers:



*A proof-of-concept function for adding force on the left, and deducting on the right.*

We tweak our secondFold to use testAcc1 and its resulting “bodies”, still represented as integers:



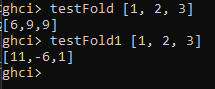
*Our second fold function using the proof-of-concept function testAcc1.*

Our testFold function remains unchanged, although it is renamed in the codebase as testFold1.

To spare the report from algorithm overload, the process has been omitted but the expected result of running this is the following list:

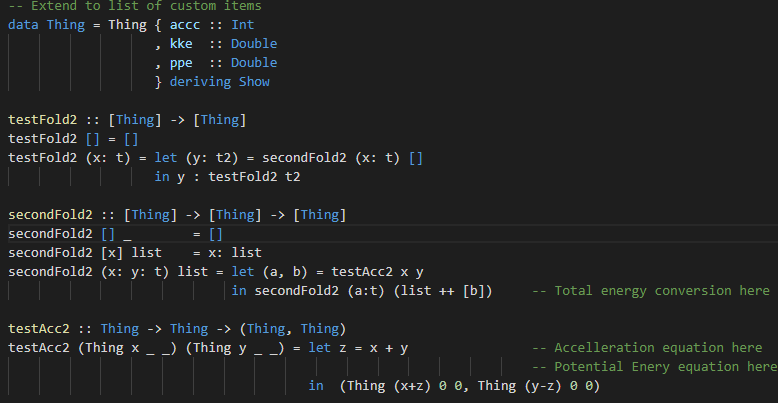
[11, -6, 1]

And running testFold1 in GHCi gives us this exact answer:



*testFold1.*

Having proven we can add on the left and subtract on the right, we now amend the functions to use a custom data type, later to be swapped for the Body data type. This will prove we can access specific attributes of the body, such as acceleration and velocity, without modifying other attributes during the fold:



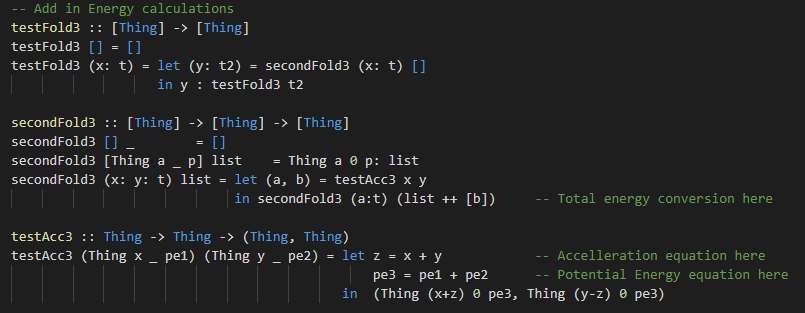
*The functions extended to a custom data type.*

By running testFold2 in GHCi, we can see that the integers representing acceleration are now changed to the expected values as found in the lists above:

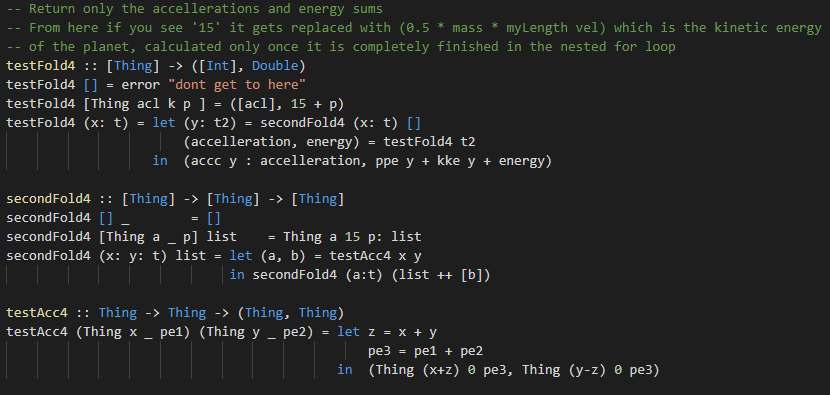


*testFold2 applied to a list of custom data types.*

The penultimate step in finalizing the algorithm is to add in the potential energy summation, while modifying the acceleration. The ultimate step is to run the algorithm and return only the accelerations and sum of all energies:



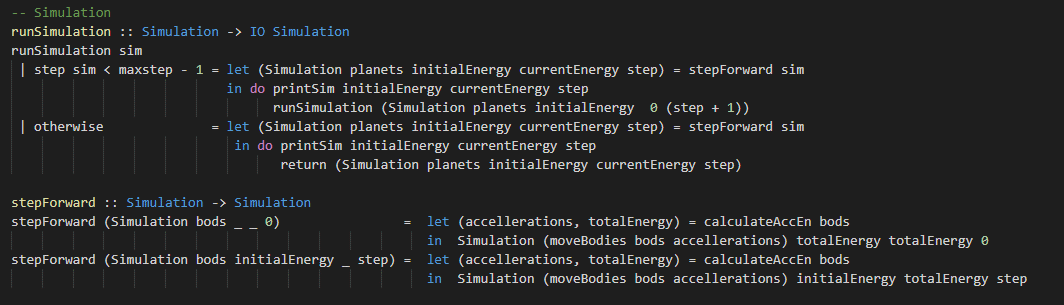
*Add in the energy calculations.*



*The final algorithm in Haskell before substituting for actual data types and operators.*

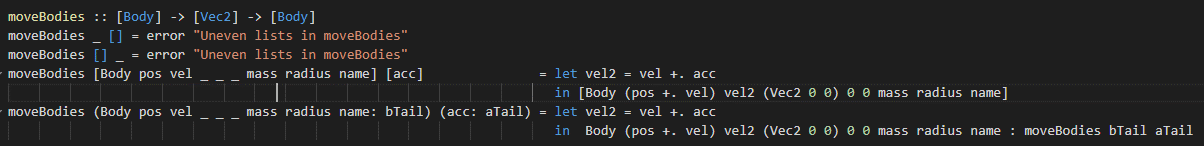
This process can be found in submission 03.Haskell.Algorithm.Design.

The Euler simulation in Haskell follows the same logic as the C++ implementation. The first step of the simulation captures the initial energy, and until the final step runs an update loop which we call “stepForward”. These are run by our IO abstraction layer “runSimulation”:



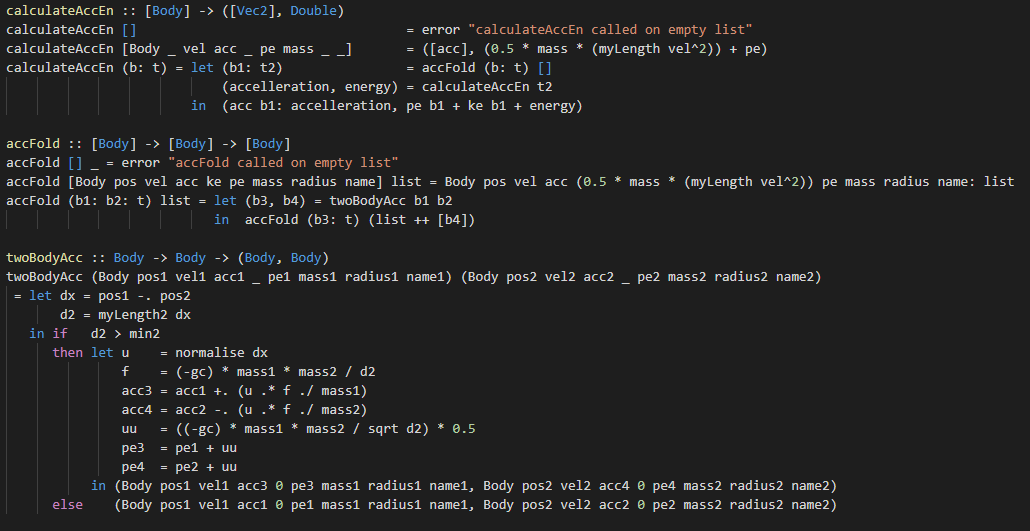
*The basic forward step of a Euler simulation in Haskell.*

The Euler integration is done in the moveBodies function, where the ts is omitted due to it being a factor of 1:



*The Euler integration with ts omitted.*

When stepping forward the simulation, we use the final product of our testFolds, where testFold4 becomes calculateAccEn, secondFold4 becomes accFold and testAcc4 becomes twoBodyAcc. Much of the twoBodyAcc function requires eager evaluation and is the same as the C++ integration:



*The final products of our testFold.*

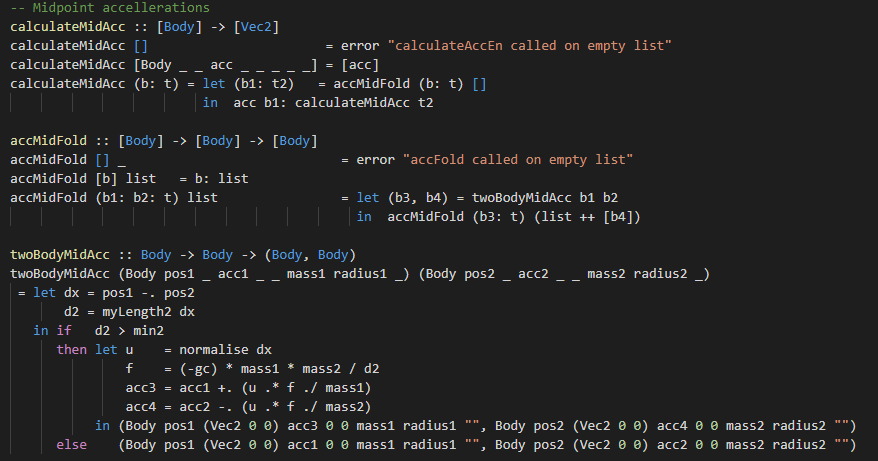
We now have a working Euler simulation of the solar system as found in submission 01.Euler.Haskell.

To convert our Euler simulation to the midpoint method we must consider the new logic:

1. Find the velocity, positions, and accelerations of the midpoint
2. Find the velocity, positions, and accelerations next step from the midpoint

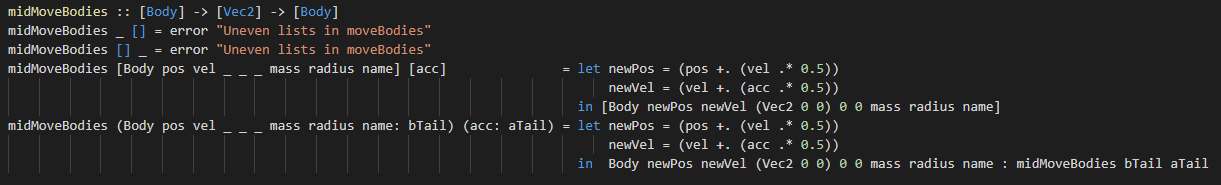
Subsequently, the first step is to find the midpoint.

We cannot easily function-ify the gravity calculation for dual usage as we did in C++. Haskell is a pure language, and functions must have a singular use while maintaining referential transparency. Subsequently, we must strip calculateAccEn back to another function:



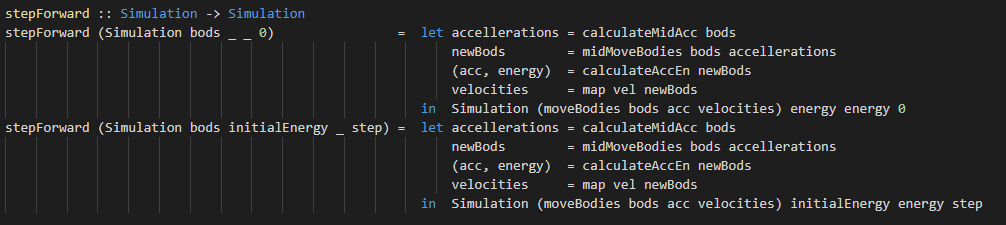
*A function to find the accelerations used to calculate the midpoint.*

With these accelerations, we can obtain the midpoint using a new moveBodies function:



*Obtaining the midpoint.*

Finally, with the midpoint calculated, we can calculate the next timestep in the simulation:



*The midpoint method in Haskell.*

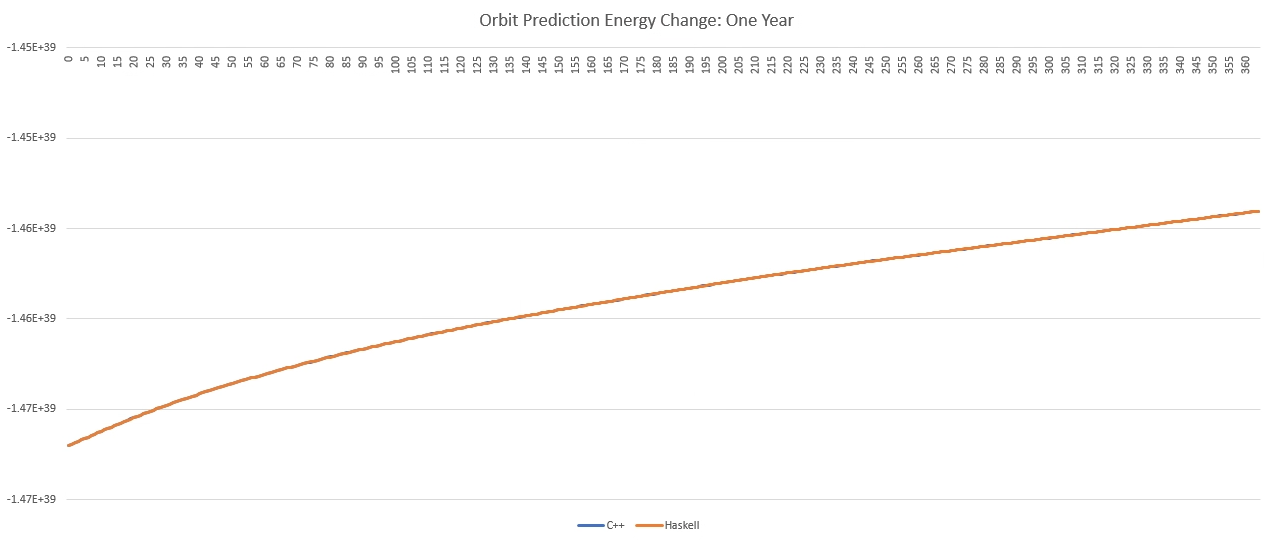
# 5. Comparisons

Now that we have two methods of integration coded in two different languages, we have two comparisons we can make: Method to method, and language to language.

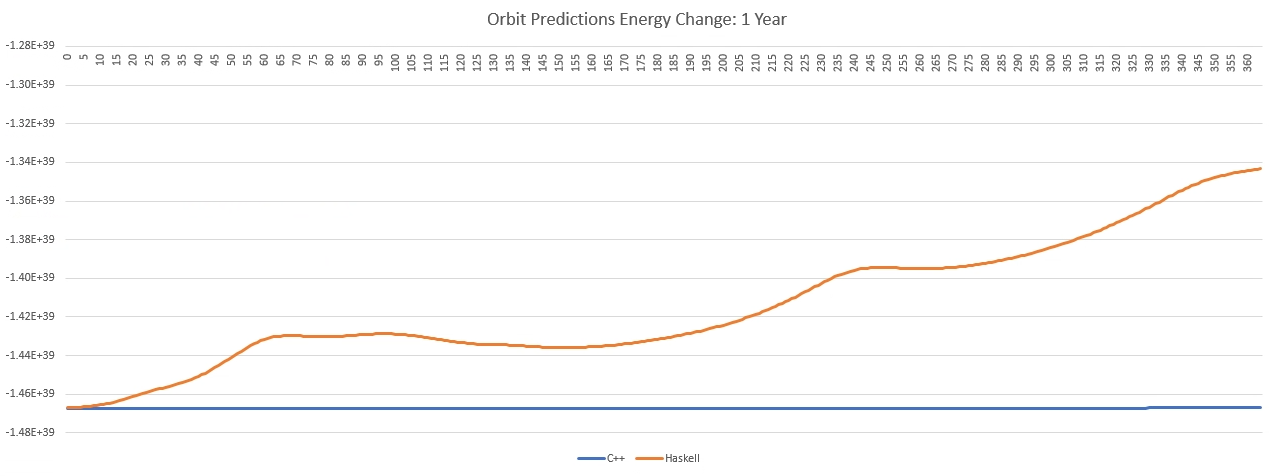
When comparing the Euler method to the Midpoint method, we want focus on the energy trends of the simulation over various time spans. As a starting prediction, we focus on which method results in less energy change. The closer to zero the energy change is, the more accurate the model is assumed to be.

Over the course of a year, the energy change in both models written in Haskell is 0.009~. In C++, the Euler method also results in 0.009~, however the Midpoint method has an energy change of 0.000007~.

*Note: The results that follow have discrepancies between languages that will be explored later.*



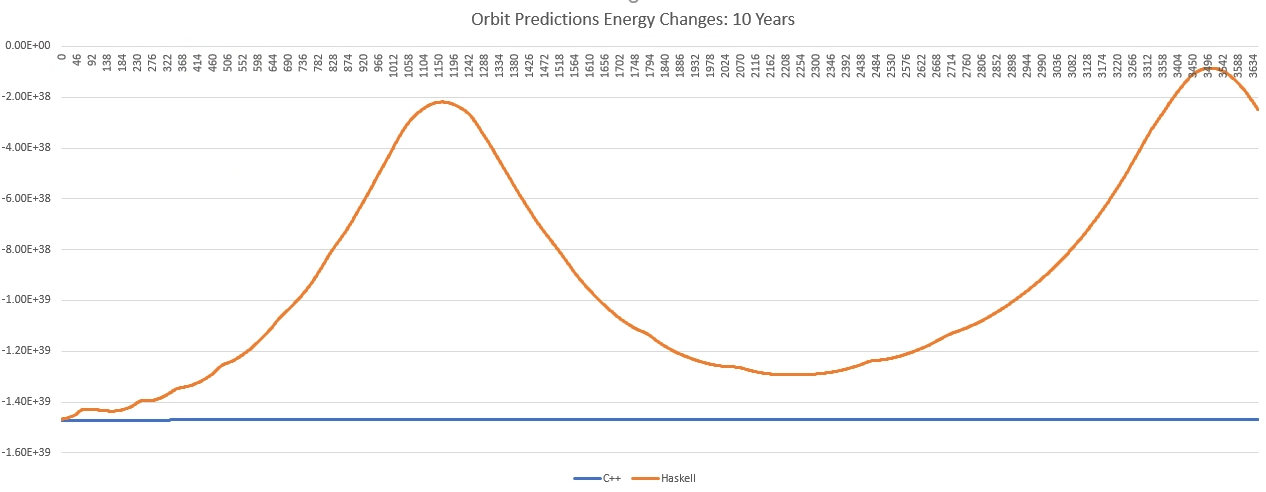
*Euler method change in energy over 365 days of orbit.*



*Midpoint method change in energy over 365 days of orbit.*

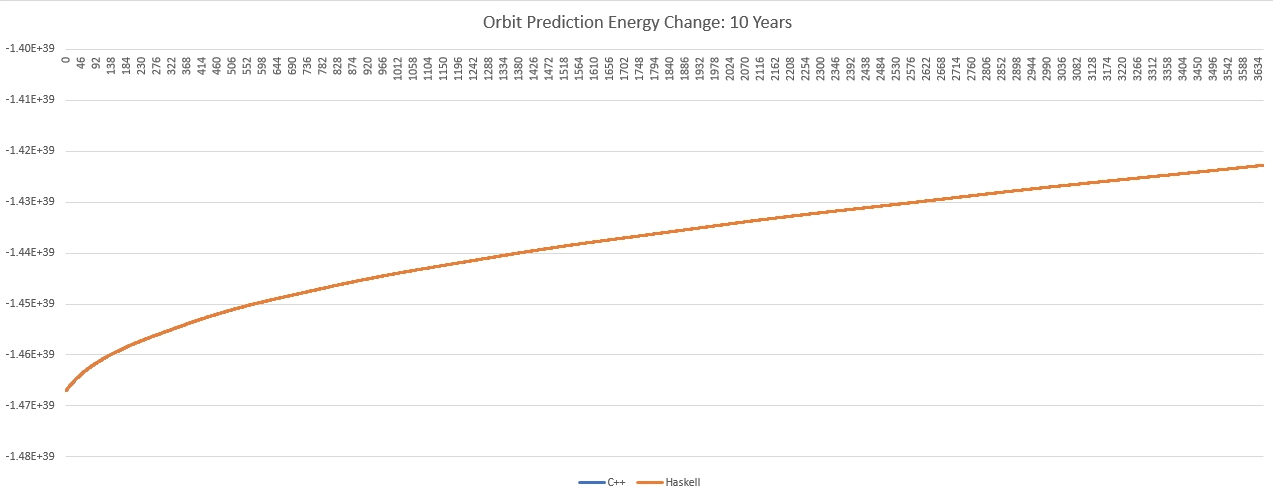
In the Haskell Midpoint method, we can see the energy beginning to fluctuate up and down. These fluctuations potentially represent the model over estimating orbits while the planets are close to each other, and under estimating orbits while the planets are further away from each other. It may also represent a faulty algorithm. We can explore this furth by extending the runtime of the simulation.

When extended to a runtime of 10 years, the fluctuations in the Midpoint method become both more apparent and consistent. This is a small indication the algorithm is working as intended, and the orbits are stabilized. As the planets repeat their orbits the energy is rhythmically over or under calculated.



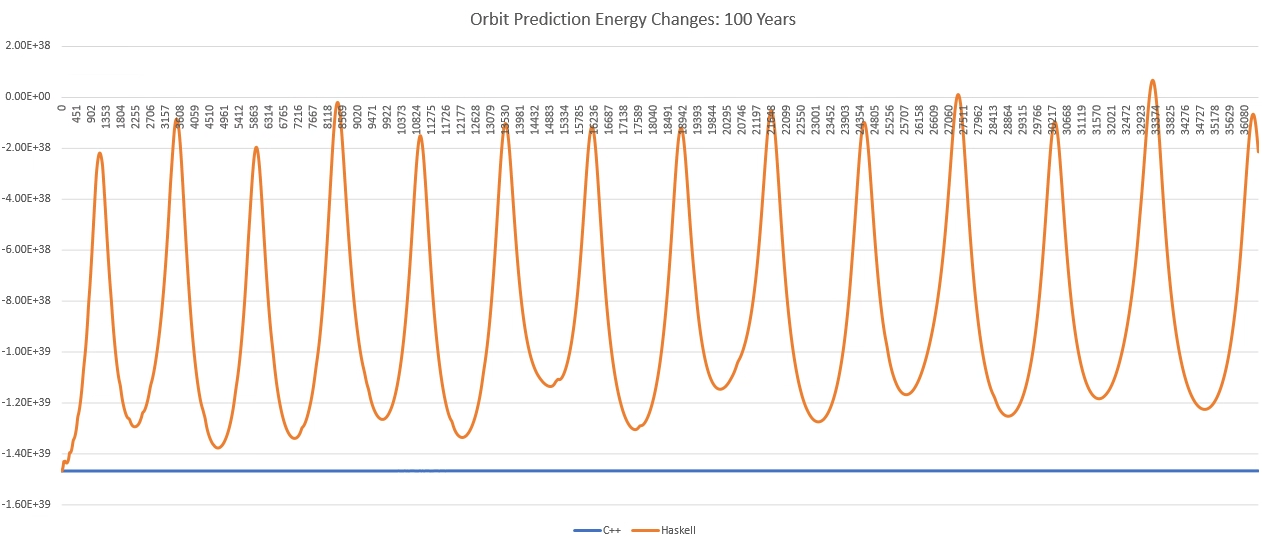
*Midpoint method energy change over 10 years.*

However, the Euler method remains a consistent upward track, with no discrepancies between the languages.

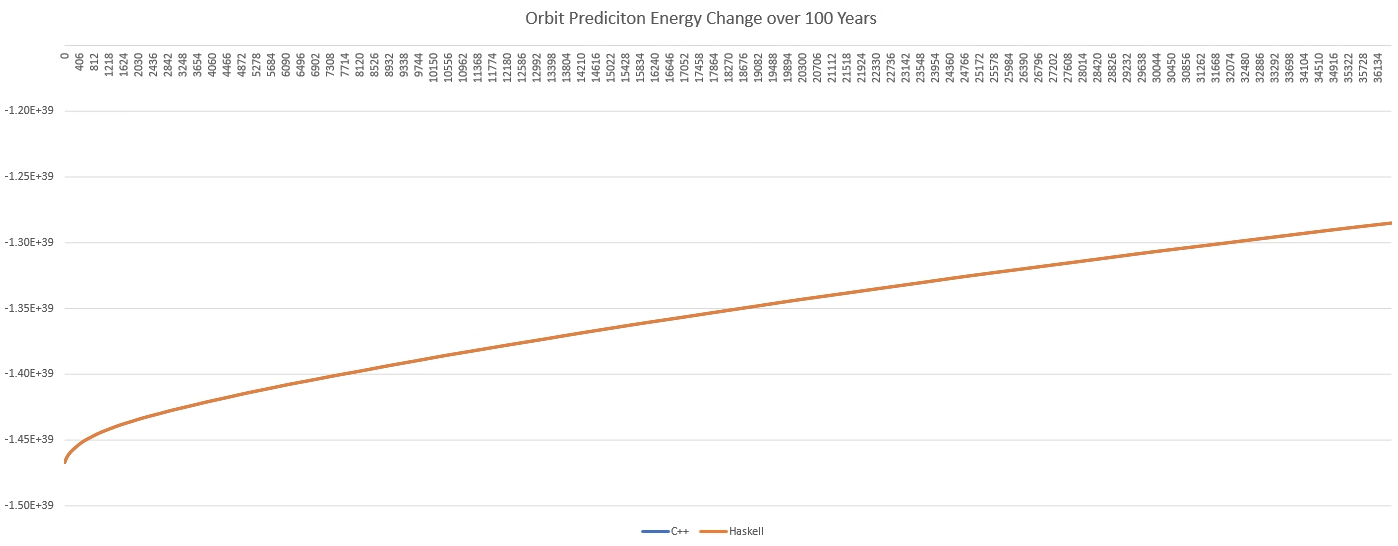


*Euler method energy change over 10 years.*

Extending the methods run time to an orbit of 100 years shows this under/over estimation of the orbits is truly rhythmic, with an average upwards trend. The average energy over the 100 years is -8.71E+38, a change of 5.96E+38. Why it doesn’t occur in the C++ simulation, nor the Euler method simulations, is unknown.



*Midpoint method energy change over 100 years.*

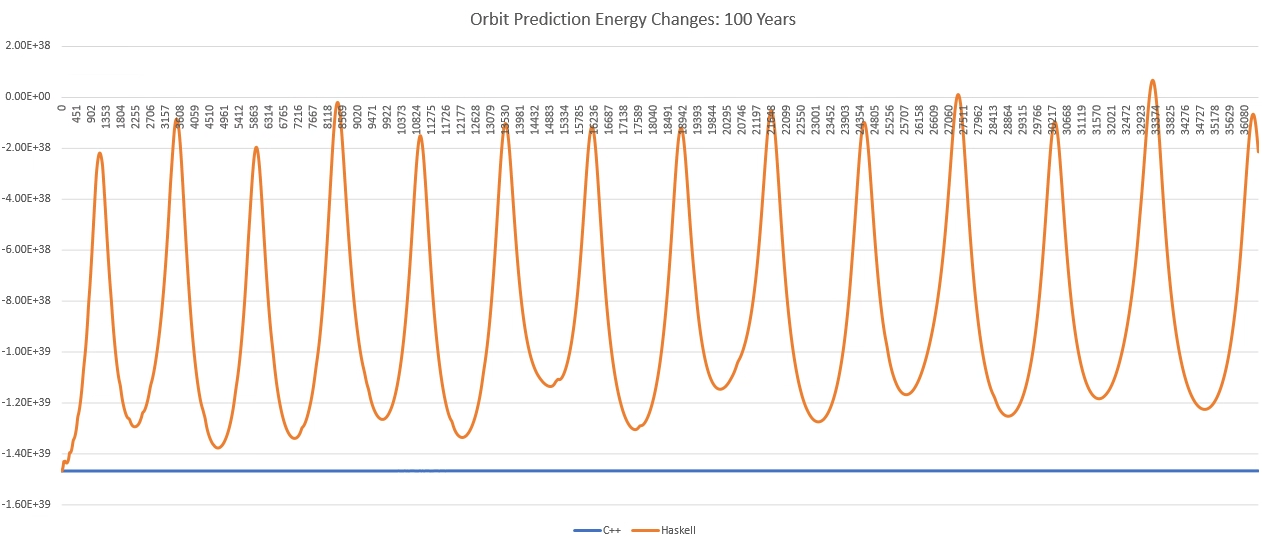


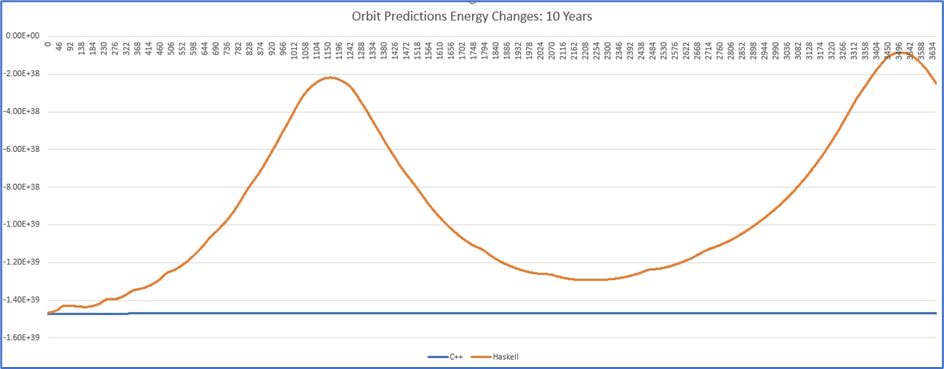
*Euler method energy change over 100 years.*

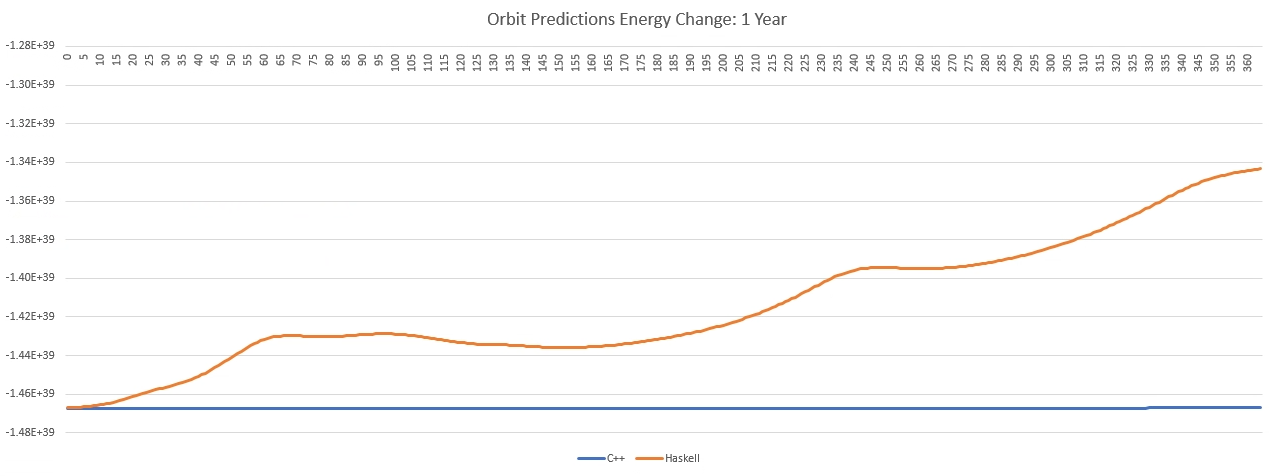
Not many conclusions can be drawn from this information, until we review the discrepancies between languages.

To ensure the algorithms are accurate to each other, we want to reverse the runtimes to a runtime of a single day. If the algorithms are accurate, the closer the run time is to a single iteration the closer to an exact match we should see the energy change between simulations.

When reversing the runtimes, we do in fact see this trend. We start by revising the last graphs in reverse order:

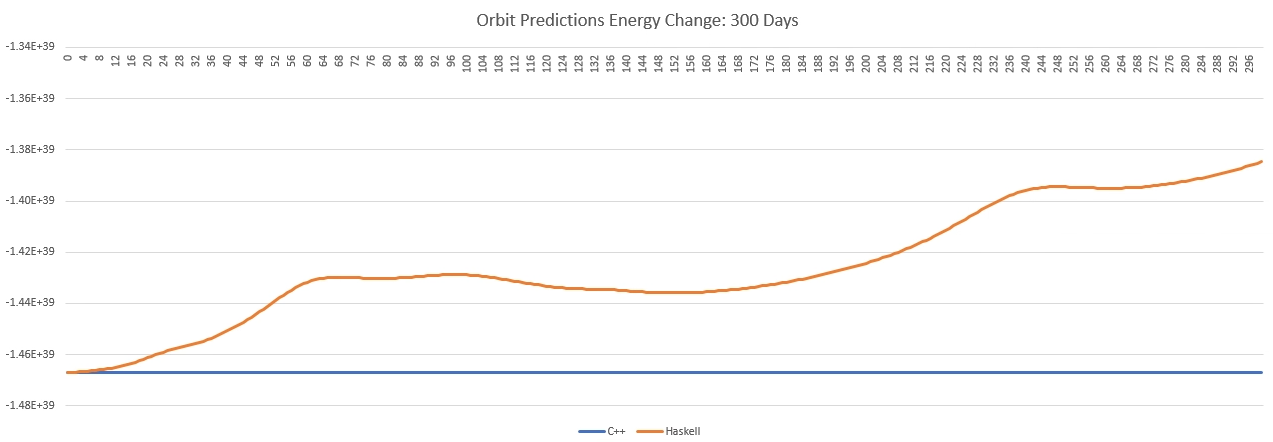


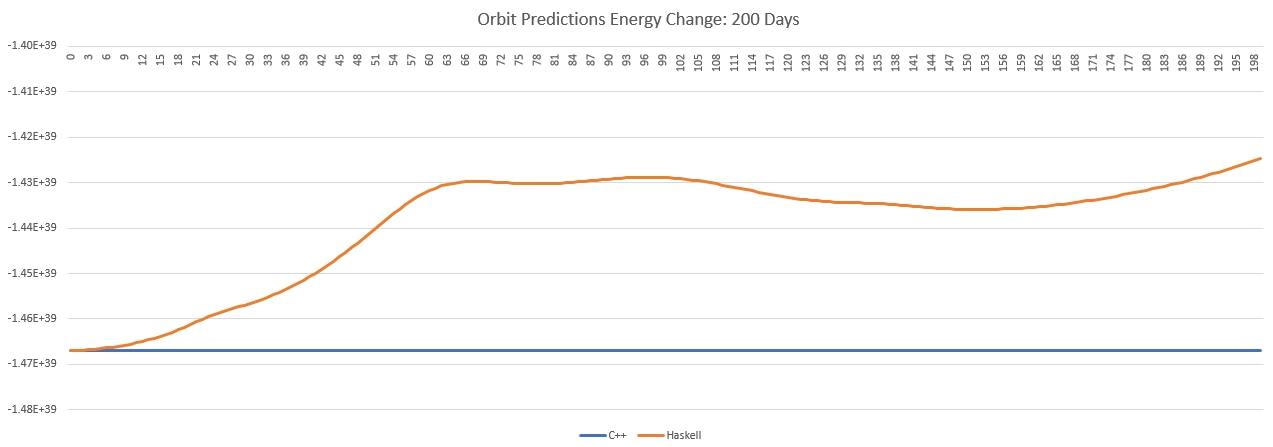


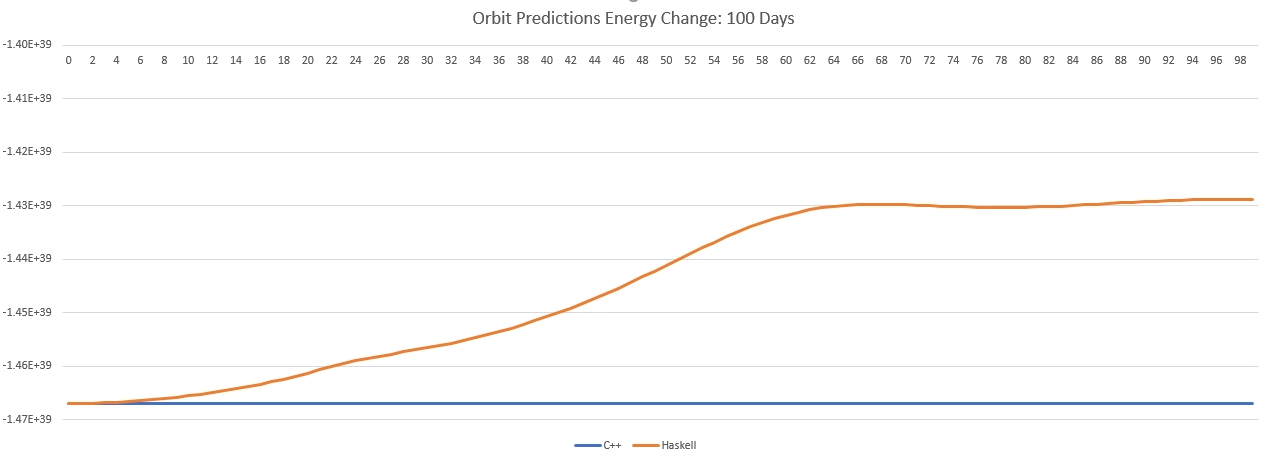


*The lowering of runtime shown to reduce discrepancies in energy calculations between languages.*

And extend these graphs by reducing the run times to 300 days, 200 days, 100 days and finally, a single day:

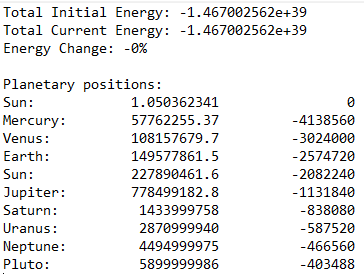




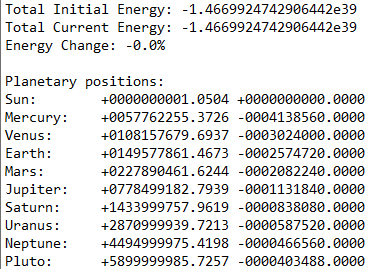


*Energy changes in the Midpoint method per language over 300-, 200- and 100-day intervals.*

Finally, if we compare the output of a single day, we can see the results are almost identical:



*Results from a single day’s simulation in C++ using the Midpoint method.*



*Results from a single day’s simulation in Haskell using the Midpoint method.*

All taken together, this indicates the algorithms are accurate to each other but does not shed light on the behavioural differences.

One possibility is that Haskell has the more accurate estimates. In the simulation we use the data type of Double for our decimal numbers. Based on the outputs, Haskell is capable of handling more decimal points than C++ accurately. Both languages handle 15 decimal digits of precision (Singh, 2018), however C++ rounds to exponential form where it deems subsequent decimals negligible. This is to favour performance over precision (Bloodfield, 2015). Haskell, compiled using GHC, ensures precision at the expense of performance (**This taken from copious readings on engineering forums but nothing I would feel comfortable sourcing**). With such large numbers in our calculations, over many iterations, it is obvious the disparity between calculations stacks up quite fast. This is one possible explanation for the difference in energy behaviours and planetary positions.

To compare runtimes between simulations I ran three different scenarios using the Midpoint method:

1. Both simulations with only an output file
2. Both simulations writing the energy of every iteration to screen
3. Both simulations logging the energy of every iteration to file

The numbers below are the averages across a run of ten simulations for each scenario described.

When printing to screen, Haskell is substantially faster than C++. For a simulation period of one year, Haskell has a run time of 0.4659564s to C++’s 0.655429s. For a period of five years, Haskell comes in at 1.8660104s to C++’s 3.44622s. For a period of 100 years, Haskell comes in at 16.9519643s to C++’s 33.3521s.

When logging to file, and having no step-based IO, Haskell starts off slower than C++, and the runtime gets substantially slower the longer the simulation’s orbital period is increased. For a period of one year, Haskell clocks in at 0.709285s to C++’s 0.005s. For a period of 10 years, Haskell runs for 17.3705571s compared to C++’s 0.02s. Due to its overly long runtime, the Haskell simulation for a 100-year period did not complete.

With no IO except the final output, Haskell is slower still. For a simulation period of one year Haskell takes 0.0733987s while C++ is so fast the system clock cannot accurately capture a runtime and reports 0s. For a period of 10 years, Haskell clocks in a 0.5707856s, where C++ take 0.005038s. Finally, for a period of 100 years Haskell takes 5.8800302s, where C++ takes 0.043908s.

# 6. Discussion

*This section serves as a conclusion and talks from a personal standpoint.*

When first approaching this project, I had two expectations. The first is that Haskell would be the faster, more accurate simulation language when no step-based IO was involved. Haskell, with its lazy evaluation, has been constantly touted as a language built for speed and accuracy. However, it is also widely regarded as useless with IO. Subsequently Haskell programs should have a single entry and exit point for the IO, abstracted into the main program through an abstraction function that calls pure functions and returns an IO() object.

The second expectation is that it would be easier than it was to integrate physical and mathematical functions into code. At a mathematical level such as integration calculations, mathematics itself becomes a sort of functional language. Each function has a domain (input parameters) and range (output). Physics calculations are also largely the same, although the domain of a function is usually another function.

My findings were interesting. As per my predictions, Haskell appears to be the more accurate language. Sacrificing performance for more accurate handling of decimal precision, and with the high numbers involved in the simulation, Haskell appears to have a more accurate grasp on the numbers. When combined with the Midpoint method, this appears to be the optimal simulation for accurate orbital predictions, despite its runtime issues.

However, Haskell is slower than C++ except when printing to screen each step. As explored in section 5 Haskell runs slower than its C++ equivalent. Possible reasons are the adherence to numerical accuracy, as previously described, and the inappropriate match of algorithm to evaluation type. That is, the algorithm for the N-body simulation forces an eager evaluation of each body-to-body interaction before stepping the simulation forward. This doesn’t align with the lazy evaluation Haskell offers and instead Haskell should be applied to a problem set which does not require an eagerly evaluated, cumulative interaction between each body before continuing.

Another finding is that the midpoint approximation method, given enough time, is somewhat self-correcting. While the energy change in the simulation did trend upwards, the cyclical nature of the total energy shows that the method self corrects its over estimations with underestimations. There is probably a way to average the simulation into a more accurate one.

I have a top two positive takeaways from this project as follows, the first of which is learning to integrate physics and mathematical functions into imperative and functional code. This has been intriguing to me from the second year of university where I studied an introductory Physics course. While I obtained an A for the paper, I realized my brain did not match Physics calculations. This has been confirmed while trying to wrap my head around the Euler and Midpoint methods in physics simulations, simultaneously with their code implementations. However, the practice and experience obtained through this project has helped immensely, and I have thoroughly enjoyed it. In a few years, when I can afford to study my Masters, this field (along with the imperative vs. functional computation exploration) will be a high contender for its focal point.

The second was deriving the cumulative for-loop algorithm for immutable lists. While there is no-doubt a better suited way of running this simulation in functional code, I challenged myself to come up with this algorithm and am glad I saw it to completion, all on my own merit. Exploring functional languages and experimenting with algorithms is by far my favorite aspect of Computer Science, as opposed to writing software, manipulating AI, or data science e.g. For example, other highly enjoyable algorithm assignments during my studies were the RSA Blockchain Encryption assignment for Operating Systems and Networks paper, and the Lift-Semaphore assignment for the Programming Languages, Algorithms and Concurrency paper.

I only had one negative takeaways from this project: Assessing the results for comparison. Once the algorithms were deduced and proven to work accurately, I spent two days running, rerunning, logging, and comparing results of both the Euler and Midpoint methods across both languages. I’m still not entirely convinced I have understood the result implications correctly. In a perfect world, I would offload this work to someone else, and focus on finding a better way to run the simulation in functional code. The aim would be to improve performance with the Haskell runtimes. One such avenue already researched would be using the JHC compiler over GHC. This would compile a C like stack implementation, over the heap-based approach GHC produces.

There were two unexpected findings in this project, the first of which is that Haskell runs faster than C++ when writing to the console every step. As stated in section 6 I have always heard Haskell to be insufferably slow when IO is involved. Subsequently, when the runtimes for the console-printed simulation were consistently less than half the C++ run times I was quite shocked. I estimate what is happening is the forced energy calculation at each step is completing the value in the world space, so each time it is referenced it is already calculated. This feature of lazy IO cumulates in time savings, as opposed to C++ who calculates it each time. However, I would not have expected it to result in such substantial time savings.

The second unexpected finding is that Haskell runs slower than C++ with only one IO at either side (the final output only simulation). For the same reasons above, I had assumed Haskell would fly through the simulation where minimal IO was concerned. This is compared to the simulation that logs on each step, where C++ holds the file stream open as opposed to Haskell whose lazy IO causes it to open and close it each step. With those results being as predicted, I had thought that one write in Haskell would cause one evaluation of the main algorithm and one file-open, write, file-close sequence.

# 7. References

N-body simulation. (2022, January 26). In Wikipedia

*https://en.wikipedia.org/wiki/N-body\_simulation*

Conservation of Energy. (2022, January 26). In Wikipedia

*https://en.wikipedia.org/wiki/Conservation\_of\_energy*

Playne, D.P. (2008, June). *Notes on Particle Simulation and Visualization.* Massey University, 2008. PDF/Print.

Bloomfield, P. (2015, October). Decimal places in a floating-point number. Found at

https://peter.bloomfield.online/decimal-places-in-a-floating-point-number/#:~:text=Hypothetically%2C%20it's%20possible%20under%20certain,and%20324%20in%20a%20double.

Singh, M. (2018, April). Difference between float and double in C/C++. Found at

https://www.geeksforgeeks.org/difference-float-double-c-cpp/#:~:text=double%20is%20a%2064%20bit,15%20decimal%20digits%20of%20precision.

# 8. List of Figures

1. Basic N-Body Simulation Algorithm. 2

2. Resultant gravity acting on each body. 3

3. Modified core algorithm. 3

4. A 2d vector of its x and y components. 4

5. Trigonometry functions for finding and normalizing a single value for a 2d vector. 4

6. Applying the above code. 4

7. Our basic O(N) algorithm for the Solar System simulation in C++. 5

8. Adding potential energy calculations to our algorithm. 9

9. The bodies of our simulation at their initial positions and velocities. 11

10. Initial capture of the kinetic energies. 11

11. Calculating the gravitational force between bodies as a callable function. 12

12. Snippet of update loop calling the gravitational calculation twice as per the midpoint... 13

13. Vec2 data type and operators. 14

14. Vec2 physics functions. 14

15. The body datatype. 15

16. Starting conditions for the simulation in Haskell. 15

17. Revisiting the core algorithm in C++. 16

18. List of integers. 16

19. Process of folding the list cumulatively. 16

20. The map/fold function. 17

21. secondFolds order of operations. 17

22. The recursive call of secondFold by testFold. 17

23. The secondFold function of the map/fold function. 18

24. testFold run in GHCi. 18

25. A proof-of-concept function for adding force on the left, and deducting on the right. 18

26. Our second fold function using the proof-of-concept function testAcc1. 18

27. testFold1. 18

28. The functions extended to a custom data type. 19

29. testFold2 applied to a list of custom data types. 19

30. Add in the energy calculations. 19

31. The final algorithm in Haskell before substituting for actual data types and operators. 20

32. The basic forward step of a Euler simulation in Haskell. 20

33. The Euler integration with ts omitted. 20

34. The final products of our testFold. 21

35. A function to find the accelerations used to calculate the midpoint. 22

36. Obtaining the midpoint. 22

37. The midpoint method in Haskell. 22

38. Euler method change in energy over 365 days of orbit 23

39. Midpoint method change in energy over 365 days of orbit 23

40. Midpoint method energy change over 10 years. 24

41. Euler method energy change over 10 years. 24

42. Midpoint method energy change over 100 years. 25

43. Euler method energy change over 100 years. 25

44. The lowering of runtime shown to reduce discrepancies.. 26

45. The lowering of runtime shown to reduce discrepancies.. 26

46. The lowering of runtime shown to reduce discrepancies.. 26

47. Energy changes in the Midpoint method per language over 300, 200 and 100 day intervals. 27

48. Energy changes in the Midpoint method per language over 300, 200 and 100 day intervals. 27

49. Energy changes in the Midpoint method per language over 300, 200 and 100 day intervals. 27

50. Results from a single day’s simulation in C++ using the Midpoint method. 28

51. Results from a single day’s simulation in Haskell using the Midpoint method. 28