MNRAS 000, 1–9 (2022) Preprint 20 May 2022 Compiled using MNRAS LATEX style file v3.0

An Astrophysical N-Body Simulation in FORTRAN90 Using the Adams-Bashforth-Moulton Predictor-Corrector

Daniel Shadrack

20 May 2022

ABSTRACT

N-Body simulations of astrophysical systems act as an invaluable tool in the study of astrophysics, from planetary dynamics to cosmology. In this paper, we introduce the mathematics and programming methodology required to produce a sophisticated algorithm capable of simulating millions of years of dynamic evolution of an astrophysical system. Included is a sample code, programmed in FORTRAN90 that implements these techniques. This algorithm is flexible and can be adapted to suit any number of applications. Based on tests performed using the code, we show that this simulation produces results with impressive accuracy, which can be fine tuned based on the user's needs. Included are a number of optimisations that have been made to the code in order to speed up either its run-time or precision. We also discuss some improvements not included here that can be made to the included algorithm to evolve it even further.

1 INTRODUCTION AND MOTIVATION

There are several applications in which an algorithm capable of simulating the interactions of a set of particles is a desirable tool to have available. The existence of increasingly powerful computers allows simulations of increasingly complex systems to be designed and performed.

One particular field which sees extensive use of these techniques is astrophysics. In this field, one is often interested in the time evolution of large scale gravitational systems such as star clusters or galaxies. Since in many cases these systems are governed almost entirely by the force of gravity, they are well suited to being modelled numerically via iterative force-integration techniques which take advantage of the relatively simple mathematics involved.

The most obvious advantage of this is that the time-scales that these systems evolve over make it impossible for humans to make meaningful observations. Programming a computer with an algorithm that takes into account all of the relevant physics allows this timescale to be shrunk from, in some cases, several million years of real time to a few minutes of compute time, depending on the complexity of the system. Additionally, a well made algorithm should allow its user to modify input parameters, thereby allowing them to easily construct any possible system they wish to study. This convenience is of course not available in real-world observations.

There is a vast array of examples to cite when discussing the use of N-body simulations in modern research. These include investigations into the formation of binary star systems (Arnold (2019)), which develop our understanding of the origins of star and planetary systems. Applications also extend to furthering our understanding of planetary formation, a field which has seen much development thanks to computational algorithms - for example the pebble accretion model of planet formation discussed in Matsumura et al. (2017). Larger scale projects also utilise these techniques, investigating the evolution of dark matter halos in the early Universe and ultimately providing crucial data in developing early galaxy formation models e.g Limousin et al. (2009).

There are many more such studies that could be called upon to evidence the immense impact that N-body methods have had on astrophysical research. However, the intention of this paper is to introduce the algorithmic building blocks necessary to produce a code that could be further adapted into any one of these applications.

2 THEORY AND MATHEMATICS

The main building blocks of the algorithm that has been designed in this project are mathematical formulae that are well defined and have been understood for some time. The complexity comes from integrating these formulae into an algorithm that can utilise them efficiently. The following section will introduce the relevant mathematics that have been applied.

2.1 Newtonian Physics

This piece discusses particle interactions in the context of gravitationally bound systems. However, this algorithm could be applied to any interaction for which the inter-particle forces can be calculated or estimated, such as the coulomb interaction between charged particles or the vanderwaal forces within gases.

The most fundamental defining interaction within an astrophysical system is the force of gravity exerted between two bodies. With an understanding of the forces between particles, a very precise picture of how their velocities, accelerations and positions evolve with time can be produced. In this case, the formula used to model this is that of Newton's law of universal gravitation, cited in eqn. 1.

$$F_{ij} = \frac{Gm_i m_j}{|r_{ij}^2|} \vec{r_{ij}} \tag{1}$$

Eqn. 1 describes the mutual force exerted between two particles i and j, along the vector ij, where G is Newton's gravitational constant, m is the mass of each particle, and r_{ij} is their separation. When studying the dynamics of a particle, it is more useful to re-frame this equation to describe the acceleration of one particle i as a result of the gravitational field of particle j, shown in equation 2.

$$a_i = \frac{GM_j}{|r_{ij}^2|}\vec{r_{ij}} \tag{2}$$

By differentiating this equation we arrive at eqn. 3. Which describes the jerk - rate of change of the acceleration with respect to time, or in other words the third time-derivative of the particle's position.

$$\dot{a}_i = -GM_j \frac{r_{ij}^2 v_i - 3r_{ij}(r_{ij}.v_i)}{|r_{ij}^5|} \tag{3}$$

Equations 2 and 3 provide all of the information about the instantaneous motion of a particle in a gravitational field that we need in order to move onward with the numerical techniques in the subsequent sections.

2.2 Taylor Series

The Taylor Series is one numerical technique that serves to approximate the value of a function at a point to an arbitrary precision, given numerical values of the function's derivatives. The general form of a Taylor Series is shown in eqn. 4

$$f(x) = \sum_{n=0}^{n=\infty} \frac{f^n(a)}{n!}(x-a)^n \tag{4}$$

The function produced from this is in most cases significantly simpler than the original, and though technically only exactly accurate with an infinite number of terms, generally provides reasonable accuracy with only a few. Combining this with the quantities described by eqns. 2 and 3 with eqn. 4 to bring this into the context of a moving particle, we arrive at very simple second order expressions for its position and velocity.

$$r_1 = r_0 + v_0 dt + \frac{1}{2}a_0 dt^2 + \frac{1}{6}\dot{a}_0 dt^3 \tag{5}$$

$$v_1 = v_0 + a_0 dt + \frac{1}{2}\dot{a}_0 dt^2 \tag{6}$$

Where v_0 and r_0 describe a particle's intial velocity and position, and v_1 and r_1 describe these values after a time interval of dt has elapsed.

Equations 5 and 6 alone are sufficient to form the backbone of a simple N-Body simulation by iteratively performing this calculation using a particle's position, velocity and acceleration and using the returned values to repeat the process, thereby advancing time in steps of dt until the desired duration has elapsed. If necessary, the Taylor Series can be expanded to higher orders to provide better accuracy. However, this provides diminishing returns as higher order derivatives tend to get increasingly complex, and begin to require more computational expense to determine than is worth it. A second order simulation may be sufficient to study the overall qualitative behaviour of a reasonably stable system, such as the Solar System, to a reasonable degree of accuracy. More dynamic systems require more sophisticated techniques to solve this problem.

2.3 The Adams-Bashforth-Moulton Predictor-Corrector

An algorithm that employs solely a Taylor series, as discussed in the previous section, is known as a single-step method. This means that, in order to calculate the parameters of a particle at step $k+1$, only the values at step k are required. This is advantageous in the context of computer algorithms, as it minimises the amount of data that must be stored in memory when simulating. However, this advantage is insignificant in comparison to the merits of using a multi-step method. These methods use several data points from previous steps in order to create a much more accurate estimate of the subsequent one.

One such numerical method (and the one employed here) is the Adams-Bashforth-Moulton (ABM) Predictor-Corrector - a method of solving differential equations. This method estimates the value of a function by taking four previous equidistant points of the function's derivative and extrapolating these to 'predict' the following point, and then interpolating backwards using this predicted point to 'correct' it further. In the context of the motion of a particle, we can consider the particle's position to be a function of time, therefore leaving its velocity, acceleration and jerk as the function's derivatives with respect to time. This makes the ABM very well suited to an application in an N-Body simulation.

The foundation of the ABM is that of the fundamental theorem of calculus, eqn. 7

$$y(t_{k+1}) = y(t_k) + \int_{t_k}^{t_{k+1}} f(t, y(t))dt \tag{7}$$

Where y is some function of t, and f is the derivative of y with respect to t.

2.3.1 Prediction

For the prediction stage, the ABM uses the Lagrange polynomial approximation for *f(t)*, taking into account four previous, known values of *f(t)* (denoted by negative subscripts in eqn 8). After some tedious mathematics not reproduced here, this provides the following function when integrated over $[t_k, t_{k+1}]$.

$$p_{k+1} = y_k + \frac{h}{24}(-9f_{k-3} + 37f_{k-2} - 59f_{k-1} + 55f_k) \tag{8}$$

Here p_{k+1} represents the predicted value for the function $y(t_{k+1})$ where $t_{k+1} = t_k + h$, with h being the interval between the points of *f(t)* that were fed into the equation. This is illustrated in Fig. 1

This step alone provides a huge improvement in accuracy over a simple Taylor Series, for a given interval (*dt or h*). The disadvantage is of course that an algorithm employing this method must hold in memory four data points for every parameter being predicted, for every particle. However, if sufficient memory is available, this method is overwhelmingly more efficient.

2.3.2 Correction

The next stage of the ABM method is the Adams-Moulton corrector. Where the predictor uses extrapolation to predict the next data point, the corrector takes this predicted value and interpolates backwards in order to further refine the estimate. Using similar mathematical derivation, we arrive at the formula for the predictor. It should be noted that the corrector does not directly use p_{k+1}, but instead f_{k+1}, which is the derivative of the function corresponding to the point p_{k+1}. This means there must be some intermediate calculation to determine this value in between the prediction and correction steps.

$$y_{k+1} = y_k + \frac{h}{24}(f_{k-2} - 5f_{k-1} + 19f_k + 9f_{k+1}) \tag{9}$$

The data points utilised in this step are illustrated in Fig. 2.

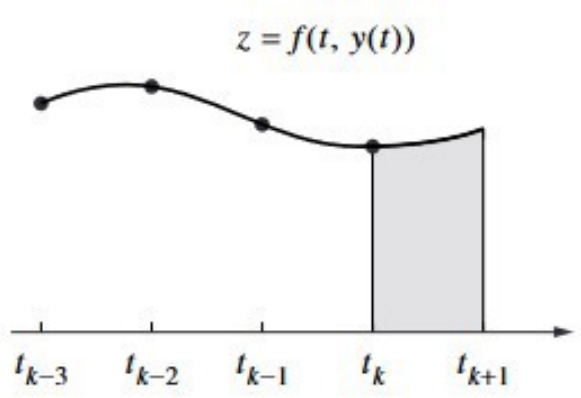

Figure 1. An illustration of the data points used in the prediction phase of the ABM. t_{k-3} through t_k (marked by solid dots) are known equidistant data points from the function. t_{k+1} is the point being predicted. From Mathews (2004)

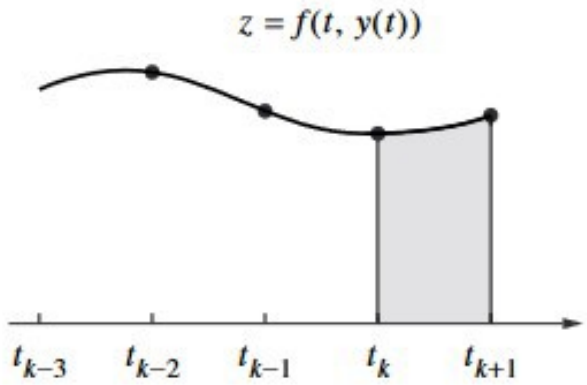

Figure 2. An illustration of the data points used in the correction phase of the ABM. t_{k-2} through t_{k+1} (marked by solid dots) are the same points discussed in fig 1, shifted once to the right.t_{k+1} has been determined from the predicted value p_{k+1} From Mathews (2004)

2.3.3 *Error Estimation*

Another significant merit of this method is that it is possible to estimate a quantifiable error on the calculated value. The errors on the predictor and the corrector are calculated via equations 10 and 11, respectively.

$$y(t_{k+1}) - p_{k+1} = \frac{251}{720} y^{(5)}(c_{k+1}) h^5 \tag{10}$$

$$y(t_{k+1}) - y_{k+1} = \frac{-19}{720} y^{(5)}(d_{k+1}) h^5 \tag{11}$$

These equations include terms that we have not yet calculated, which makes them undesirable in their current form. Fortunately, we can make some assumptions to make them more useful. If we assume that h is sufficiently small that $y^{(5)}$ is approximately constant, we can eliminate all of the new terms. This leads to eqn. 12 for the error on the corrector.

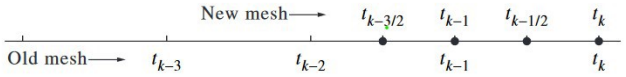

Figure 3. An illustration of how the interval between points can be halved while still retaining the four previous points required for the ABM formulae. From Mathews (2004)

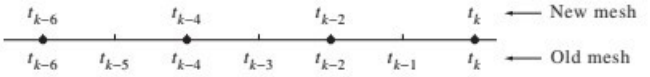

Figure 4. An illustration of how the interval between points can be doubled while still retaining the four previous points required for the ABM formulae. From Mathews (2004)

$$y(t_{k+1}) - y_{k+1} = \frac{-19}{270}(y_{k+1} - p_{k+1}) \tag{12}$$

This gives a good estimate for the inaccuracy on the corrected term, which will prove to be a very useful value to know for the purposes of a simulation.

2.3.4 *Variation of the Interval*

As discussed so far, the ABM is only applicable in cases where the interval between points is constant. If we wish to predict the function at some point that does not line up exactly with an integer multiple of h, this is impossible without some additional steps.

An intuitive way of solving this problem is to allow the interval to shrink or grow by powers of 2. This is done quite simply by employing a fourth degree interpolation formula - equations 13 and 14 .

$$f_{k-1/2} = \frac{-5f_{k-4} + 28f_{k-3} - 70f_{k-2} + 140f_{k-1} + 35f_k}{128} \tag{13}$$

$$f_{k-3/2} = \frac{3f_{k-4} - 20f_{k-3} + 90f_{k-2} + 60f_{k-1} - 5f_k}{128} \tag{14}$$

Where $f_{k-1/2}$ represents the newly generated point between f_k and f_{k-1}. This is illustrated in Fig. 3

The opposite can be done by simply storing twice as many previous data points. If we have 8 data points, we can simply disregard points 1,3, 5 and 7 to arrive again at four points equally

spaced with double the interval. This is also illustrated in Fig 4 .

Variable	Data Type + Dimensions	Purpose
`n`	Integer, 1	Total number of particles in system
`r`	Double-precision, n x 3 x 8	Stores the position vectors of each particle, in each dimension
`v`	Double-precision, n x 3 x 8	Velocity vectors, each dimension
`a`	Double-precision, n x 3 x 8	acceleration vectors
`ai`	Double-precision, n x 3 x 2	interpolated acceleration
`vi`	Double-precision, n x 3 x 2	interpolated velocity
`M`	Double-precision, n	Masses of each particle
`s`	Double-precision, n x n	Absolute separation of each particle from each other particle
`d`	Double-precision, n x n x 3	Vectors from particle i to particle j
`E0 and E1`	Double-precision, 1	Total system energy at start and end of sim
`CurrentTime`	Double-precision, 1	Total time elapsed in sim
`Step`	Double-precision, 1	Current timestep
`relerr`	Double-precision, 1	Arbitary condition for error on the prediction
`length`	Double-precision, 1	User defined length of sim
`PCerror`	Double-precision, 1	Error on predictor-corrector
`t`	Integer, 1	Total steps

3 PROGRAMMING METHODOLOGY - CORE COMPONENTS

The following section serves as a catalogue of the methods used to build the mathematical theorems discussed in the previous section into a working N-Body simulation. It also highlights some of the important factors to consider when implementing such an algorithm onto computer hardware. The programming language used in this paper is FORTRAN90, chosen for its balance between efficiency in number-crunching applications and simplicity of syntax. A language such as C may take the edge in speed, although it is generally more technical and difficult to pick up.

3.1 Initial Conditions and Setup

The first things to consider when designing the algorithm are the data structures that will be utilised throughout to store all of the necessary information in memory, as well as ensuring that the algorithm is flexible enough to receive whatever input parameters are desired by the user.

3.1.1 Variables

Firstly, the variables that will be used to hold all of the information about the simulated system must be defined. This allocates locations in memory that can be referenced by the program quickly. It is important to carefully consider which data types to employ for each one in order to balance precision with efficient memory use.

For example, a parameter such as the position vector of a given particle must be known to a very high precision. This necessitates the use of a floating point variable, specifically a double-precision. Conversely, a variable which simply keeps track of how many iterations have occurred in a loop can be stored as an integer. An integer requires significantly less storage space in memory than a float. As we simulate larger and larger systems, the issue of memory becomes much more prevalent. Ensuring that variables are of the right type is one easy optimisation step that can be taken immediately.

The table across lists the most important variables used in the example code provided, along with their data types and uses. Variables defined to hold constants are not included. Some additional variables are used in intermediate calculations, though these are the fundamental ones.

3.1.2 Initial Conditions

In the interest of making the algorithm as flexible as possible, it is important to minimise the use of parameters that are hard-coded. That is, cannot be adjusted by the user. In the example code provided, the first block serves to initialise the system and load all of the initial conditions into memory. In this case, it is the Solar System. Values for the semi-major axes of the planets along with their velocities and masses and loaded in. For general use, this could be easily adjusted to accommodate any system. The number of possible particles is not fixed by the algorithm.

In this case, the starting angle of the planets of the Solar System are randomised, so that they will begin in a different position in their orbit with each run. This is useful for testing purposes as it ensures that any results gained are not due to some coincidence in the initial conditions. This also illustrates a potential application for the program, which could involve randomly generating a system such as a star cluster.

After the initial conditions have been set, the algorithm determines both the centre of mass and the centre of velocity of the system. This is performed by looping through each body, and carrying out the calculations described by Equations 15 and 16.

$$COM_{\text{sys}} = \frac{1}{M_{\text{tot}}} \sum r_i m_i \tag{15}$$

Where r_i is the position vector of particle i and m_i is its mass. M_{tot} is the total mass of all particles in the system.

$$COV_{\text{sys}} = \frac{1}{M_{\text{tot}}} \sum v_i m_i \tag{16}$$

Where v_i is the velocity vector of particle i.

The centre of mass is then subtracted from all of the position vectors of each particle in the system, and the centre of velocity is subtracted from their velocities. This ensures that the system as a whole has a net linear momentum of zero, and as such will not drift in coordinate space over time. Without this step, the Solar System

would drift at a speed of around 12m/s due to Jupiter's momentum. This drift can add up significantly when simulating long time periods.

Another step taken at this stage is to calculate the total energy of the system in its initial state. This is done similarly to the calculation for instantaneous acceleration with a nested loop, instead using the equation for gravitational potential energy. This is summed with the kinetic energies of each particle. According to fundamental laws of physics, orbital bodies will trade energy between kinetic and gravitational potential as they interact, but the total sum will remain constant. The energy calculated in the initial state can be compared to the total energy in the final state after the simulation has run its course. If physically correct, the ratio of these two energies should be exactly unity. Any deviation from unity gives an indication of the overall accuracy of the simulation.

The code provided in this project is capable of fulfilling this metric to a precision in excess of ten significant figures, though its accuracy can be fine tuned by the user based on the balance between speed and accuracy that they desire.

3.2 Bootstrapping

When using the ABM, we require four equidistant data points in order to predict a fifth. The problem with this is that our initial conditions only provide one point, and we have no information from which to extrapolate a polynomial.

This is solved by employing a simple second order Taylor series (eqns 5 and 6) for the first three iterations of the algorithm. An extremely small time-step is utilised here since we are using a more rudimentary approximation, so that we sacrifice as little accuracy as possible. The importance of determining the correct time-step size is discussed further in section 3.4.

Since equation 3 is quite complex and contains a large number of multiplication terms, we avoid using it to directly calculate the jerk on a body. Instead, we use the average acceleration between the current step and the previous step (which is held in memory). We also truncate equation 5 to second order so that we do not have to consider the jerk. Since we are using a tiny time-step to bootstrap into the simulation this truncation is an acceptable approximation.

The process involves utilising a nested loop. Each particle i has the acceleration calculation described by equation 2 evaluated for every other particle j (not including when i = j to avoid self-counting). This provides us with the instantaneous accelerations felt by every particle in the system. Given that we now have r, v and a for every particle in the system, we can evaluate the Taylor Series and estimate the values of r and v after one time-step. These can be used to again determine the instantaneous accelerations, as above, and so on.

Each time we complete a step, we store the calculated values in an array. After the next step, these values are retained but pushed backwards in the array by one position, being replaced by the new ones. After three steps using the Taylor Series, we now have an array containing the four previous data points necessary to move on with the ABM method.

3.3 Iterative Application of the ABM

This component of the algorithm is by far the largest. The vast majority of compute time is spent repeatedly evaluating the ABM and advancing time until the end of the simulation.

Initially, the four data points found from the bootstrap are used in the predictor equation. (8). Translating this equation so that it is in the context of particle dynamics yield the following two equations.

$$p_r = r_0 + \frac{dt}{24}(-9v_{-3} + 37v_{-2} - 59v_{-1} + 55v_0) \tag{17}$$

$$p_v = v_0 + \frac{dt}{24}(-9a_{-3} + 37a_{-2} - 59a_{-1} + 55a_0) \tag{18}$$

Where p_r and p_v are the predicted position and velocity vectors of a particle, respectively. Subscripted a and v are the previous steps' values of the acceleration and velocity of the particle.

Now we have predicted values of r and v for each particle, but not a. To find a we simply apply the direct-summation force calculation loop (as in the bootstrap), using these predicted values. This gives a predicted value for the acceleration of the particle in the next step.

Now that we have predicted values for each parameter, we can apply the ABM corrector. Again bringing the equations into the correct context we arrive at the following.

$$r_1 = r_0 + \frac{dt}{24}(v_{-2} - 5v_{-1} + 19v_0 + 9v_p) \tag{19}$$

$$v_1 = v_0 + \frac{dt}{24}(a_{-2} - 5a_{-1} + 19a_0 + 9a_p) \tag{20}$$

The acceleration double-loop is then performed one more time using the corrected positions and velocities to arrive at a value for the 'corrected' acceleration.

These corrected values of r_1 and v_1 serve as our final estimates of how the system has changed after the time-step dt. The arrays containing the kinematic parameters of the particles are shifted backwards by one position, and these new values take the place of r_0, v_0 and a_0.

The loop then resets back to the prediction stage, with each iteration advancing simulation time by dt.

The loop exits once a user-defined amount of simulation time has elapsed. This is implemented very simply by adding a variable that tracks the total time simulated, and adding an exit condition for the loop that checks if this value exceeds the intended duration.

3.4 Timestep Optimisation

The choice of time-step is incredibly important when designing an N-Body simulation algorithm. Since the errors on the predictor and the corrector depend strongly on the interval between points, the smaller the time-step, the smaller the error. However, a smaller time-step demands a longer execution time for a given real amount of time simulated. A simulation of 100 years would require 100 steps if our time-step size is 1 year, but would require 1000 steps and therefore take ten times as long to run if we choose a time-step of 0.1 years. The 0.1 year case would be more accurate than the other, but this increased accuracy may not be necessary for the intended application, therefore wasting valuable computing time. We must therefore compromise between the need for a precise code and the need for a fast one.

Choosing a time-step can be a difficult task, considering we do not necessarily know how the system will evolve before we simulate it. More dynamic systems will require more precision in order to trust the results, whereas more stable systems can afford to sacrifice some accuracy in favour of speed. A system that is expected to move between phases of dynamism and stability, such as an elliptical orbit of a planet that spends most of its time moving slowly far away from its parent and shorter periods moving quickly very close to it in an encounter, can cause problems with this compromise.

Fortunately, there are some algorithmic techniques we can employ in order to have the simulation handle this compromise for us.

3.4.1 Variable Time-steps

As mentioned above, choosing a time-step for a system can be complicated if we expect the system to transition between phases of chaos and stability. If we have to choose a single step size for the entire simulation, we either have to sacrifice accuracy in times of chaos, or speed in times of stability. Neither solution is desirable.

Instead, we can introduce a component to the algorithm that allows itself to shrink and grow the step size depending on particular conditions. Through this, we can have a small step size when accuracy is needed and a relaxed one when this is less important. Provided we set the right conditions, this change can happen automatically without any input from the user.

Referring back to Section 2.3.4, we see that the ABM method can be easily adapted to include this feature. At any one time during the execution of the ABM, we have in memory at least four data points tracking the values of each kinematic parameter. If we wish for the time-step to be reduced, we can simply employ the interpolation equations (13) and (14). This provides us with four new data points whose interval is half of that of the originals. These replace the old ones, and the ABM can continue with this new reduced time-step. This process can be repeated as many times as is required in order to reach a sufficiently small step size.

The process for increasing the time-step is slightly nuanced. Though the mathematics is simpler, the memory usage of the program requires an expansion. If we are to disregard every second data point and still end up with four values, we must start with eight. This means that to provide this functionality we must store the previous eight values for each parameter v and a, rather than just four. This is easily implemented, though now these arrays demand twice the memory.

Unlike shrinking the time-step, this cannot be done repeatedly in consecutive steps, since after the step size is doubled, we no longer have eight equidistant data points available. This is remedied easily by setting a condition within the program ensuring that it must wait at least three steps after a doubling occurs before it is allowed to double again. This allows enough time for the array containing the data to populate with newly calculated values.

The criterion the algorithm uses to decide whether to increase, decrease, or maintain the size of the time-step is based on the errors on the predictor and corrector, as defined in section 2.3.3, Eqn. 12. The values produced from the predictor and corrector are substituted into this equation, and the fractional error on both the position and velocity estimates are calculated. The user can define an acceptable relative error parameter, which will be compared to these error estimates. If either calculated error exceeds the accepted amount, the algorithm for shrinking the time-step is applied before the next step. If the error is significantly smaller, say by a factor of 100, we know that we have some precision to spare, and the algorithm for increasing the time-step is applied.

4 PROGRAMMING METHODOLOGY - EXTRA FEATURES

4.1 Optimisation Using OMP Parallelisation

Modern computers almost always possess CPU units with several cores. This means that they can perform multiple operations simultaneously. In order to take advantage of this fact, we can implement the OpenMP (OMP) parallelisation package.

A direct summation N-Body simulation is extremely well suited to the implementation of parallel computing. Each step of the algorithm can be considered like a freeze-frame in time of the system. As time is frozen during each step, the forces applied by particle i on particle j are completely unrelated and separate from the forces applied by particle i on particle k. Since we do not need information about the first calculation to perform the second, there is no reason why they cannot be performed simultaneously.

Implementation is fairly simple. We must define some variables which will be stored globally (are visible to all parallel threads) and some which are local (unique between each thread). We can, for example, make the positions of all particles public, and localise the inner loop of the direct summation. This means that for every particle i, the index j represents a different particle on each thread. Given sufficient threads, we can evaluate the total force on particle i due to all other particles in as much time as it took to evaluate the force from only one. The speedup from enabling this feature is approximately linear with the number of available threads.

One issue that can arise from this is that memory must now be managed between different threads, rather than streamlined into only one. This introduces some overhead 'administrative' expense which can negate the efficiency of this optimisation when applied to a system with a small number of particles. As the number of particles increases, the increase in efficiency becomes proportionally more dominant over this initial slowdown.

4.2 Multiple Particle Time-steps

Even after employing a variable time-step, there is still room for improvement in terms of reducing the number of redundant calculations that must be performed. When using a single, universal, time-step for the entire simulation, the limit is defined by the most 'needy' particle. If we have one particle demanding a small step size and ninety-nine others that would be accurately tracked with a much larger one, all one-hundred particles must still be evaluated using the smallest step-size. This is a situation which is likely to arise in a simulation such as that of a star cluster, where there may be a number of rapidly moving stars in the centre of the system, accompanied by several much less dynamic bodies in the outer regions. Evidently this introduces some wasted CPU time modelling these less demanding particles to an unnecessary precision.

A solution to this problem is found by introducing independent step sizes for each particle. Using this, we can still resolve our needy particle as often as is necessary, while avoiding having to do so for all of the others.

In practise, this is done by organising the particles into a hierarchical system. We can use the same doubling and halving mechanism that is baked into the ABM as before, this time instead evaluating the errors on each particle individually.

Instead of all following the same step-size, each particle is now assigned to a level that is some power of 2 multiple of a defined 'base' time-step. The most demanding particles populate the lower levels, while the others drift up to higher step sizes. The hierarchy ensures that particles remain synchronised over multiple steps, despite having different step sizes. One particle may be in the lowest tier, being addressed every step, while another may be in the third tier, being addressed only every 4th step.

The computation time is saved by evaluating the instantaneous acceleration of a particle only when this integer number of steps has occurred. Since this is by far the most computationally expensive

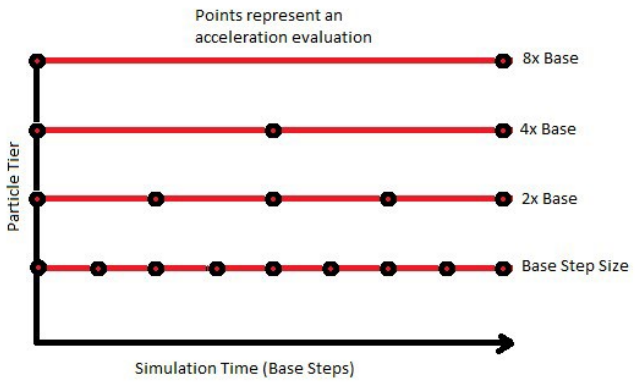

Figure 5. An illustration depicting the hierarchical system employed by the multi-particle technique. Particles are synchronised for the first step and then have their accelerations evaluated at the integer multiple of the base step for the tier that they are in. After a number of steps equal to the highest tier has passed, the particles are again synchronised.

calculation in the entire algorithm, this cuts down on the CPU demand significantly.

The positions and velocities of every particle in the system must still be updated every step, however, since this information is required to evaluate the forces experienced by the particles utilising a smaller step size. This requires only a simple application of the predictor and corrector, since we are essentially assuming the acceleration of the particle has not changed since the previous step.

Decreasing the tier of a particle (and therefore halving its step-size) can be done at any time, since we simply employ the same interpolation methods in the previous section, and a particle in a high tier will always be synchronised with particles in every tier below it. Moving particles to a higher tier (larger step size) is slightly more complicated. If we were to move a particle to tier 2 in a step when tier 2 is not due to be evaluated, this would cause this level to become out of sync with itself. Therefore, we must wait until the evaluation step in the first tier is synchronised with the tier above it. Since the tiers are powers of two apart, this occurs every second step. This is best illustrated with a diagram: Fig. 5 shows the organisation of the particle hierarchy. In this diagram, moving to a higher tier is only possible when a dot is vertically aligned with another above it.

A check to ensure particles are permitted to move tiers can be implemented with a simple integer variable for each tier that counts up once per base step, resetting each time acceleration is calculated. A boolean variable can also be used to track whether tier shifting is allowed on the current step.

4.3 Outputs

To be useful for any kind of experimental application we must implement some functionality to log and export data regarding the outcome of the simulation. This is another feature that is relatively simple to implement in FORTRAN. Due to the small size of the time-steps compared to the length of the simulation, there is no need to log data every step. Instead, we introduce a variable that counts up by 1 each time an iteration occurs. If we want the algorithm to log the state of the system every 500th step, we simply check each iteration if this variable is a multiple of 500 using a conditional statement.

If true, we take the values currently stored in the most recent position of each array and write them to a .csv file, along with the corresponding time that has elapsed in the simulation. When the simulation terminates, this provides us with a file containing a set of approximately equal spaced points in time, and every piece of information we require to construct a picture of the system at that time (the positions of each particle, their trajectories / energy etc).

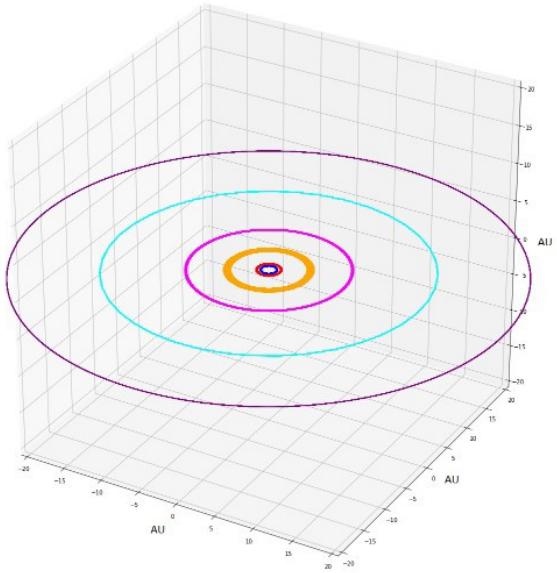

Figure 6. A 3D model depicting the orbital paths of Solar System bodies (Earth to Neptune) over the course of a few hundred years. This system is very stable and so the path of each particle overlaps itself with each orbit.

From this csv. file we can use any data analysis software we choose to extract this information and reconstruct a replay of the simulation. In the case of this project, python code utilising the matplotlib.pyplot and numpy packages has been employed to construct 3-D graphs depicting the orbital tracks of each particle from the beginning to the end of the simulation.

Figure 6 illustrates this in the case of the Solar System.

5 POSSIBLE IMPROVEMENTS TO THE ALGORITHM

5.1 N-Body Units

One optimisation that can be made to the provided code is to change the base system of units that is used to store the variables. Currently they use SI units, i.e the position vectors are expressed in metres and the masses in kilograms.

This can be improved upon by choosing a different set of base units. When programming, it is desirable to use units which result in the magnitudes of the variables being as close to unity as possible. This is a consequence of the way computers store float data types. A double-precision will always use the same quantity of allocated memory. However, sharing of the memory between different pieces of information about the number can change. For a decimal number, part of the memory must be used to store the digits / significant figures as a binary representation. An additional piece of memory must be used to store the order of magnitude of the number. If a number is preceded by 100 zeroes, the program must store that fact in part of its allocated memory. Conversely, if the number is around unity, the program does not need as much room to store the information

regarding the order. Instead, more memory can be allocated to storing the significant figures. This allows the precision limit of the data type to be tightened, therefore introducing a smaller error in calculations.

The desired units can vary by system. For example, in the Solar System, length units of AUs and mass units of Solar masses may be good choices. In something like a star cluster it may make more sense to use parsecs.

5.2 Spatial Decomposition

A more algorithmically complex optimisation that could be applied to a simulation of this kind is spatial decomposition. When there are several 'clumps' of particles some significant distance apart from each other (for example two star clusters or galaxies significantly separated), we can model the gravitational field produced by one of these clumps as originating from a single point source, with a mass equal to the total of its constituent particles. When valid, this assumption allows us to avoid resolving every individual particle in the cluster. The increase in efficiency from this method can be profound in some scenarios.

Though not implemented in the provided code, the methodology for enabling this feature is well understood. One such solution to this problem is the Barnes-Hut tree. The simulation space is first broken down into a three-dimensional grid of cubic cells, and the number of particles in each cell is determined. Each cell containing more than one particle is then broken down into smaller sub-cells. This process is repeated until we have only cells containing a single particle. When evaluating the acceleration calculation for a particle, instead of looping through all other particles as usual, we loop through other cells. We first consider the largest cell level, which may contain several particles. If the size of this cell is insignificant compared to its distance to the primary particle, we can model the entire cell as a point mass thereby skipping having to go though all of its constituent particles. If this is not the case, we break this large cell into its eight sub cells and repeat the process until the assumption is either true or we have cells with only a single particle within them. This process is illustrated in the branching diagram shown in the right side of Fig. 7.

The process of breaking down the entire simulation space into a grid must be done after every step, since particles may have moved between cells. This means that a significant overhead of computational expense is introduced. As a result, this optimisation should only be employed for systems which specifically take advantage of it. A system that does not contain clumped sets of particles (e.g a planetary system) may become less efficient with this addition.

6 CONCLUSION

Generally, the product of this project is a robust and sophisticated N-Body algorithm. This algorithm is capable of accurately simulating very long time-periods of dynamic interactions between particles in a wide variety of astrophysical systems. In fact, depending on the user's need to balance accuracy and speed, the algorithm can be easily tweaked to adjust this compromise. In the attached code, the algorithm is capable of modelling the Solar System for several tens of millions of years with impressive accuracy. Using the metric of energy retention discussed in section 3.1.2, it is capable of achieving an energy conservation that is unity to 10 significant figures after one million years of simulation time. Using a typical mid-range consumer CPU, and employing the optimisation features discussed here, it achieves this easily within less than one hour of real-time.

Naturally, the Solar System is relatively simple, and as the number of bodies being simulated increases, the CPU time required also increases exponentially. However, the optimisation features employed here reduce the impact of this significantly, and the result is an algorithm which could just as easily be used to study the evolution of far more complex systems in a time frame that is reasonable.

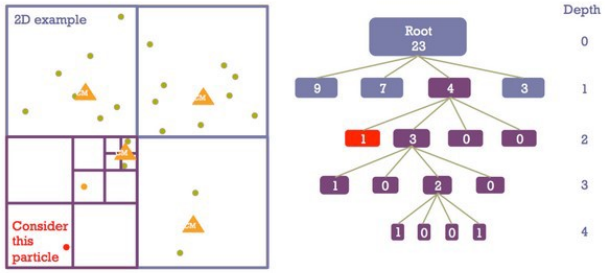

Figure 7. An illustration of the Barnes-Hut algorithm, wherein a particle has its instantaneous acceleration determined based on neighbouring cells, rather than all other particles. The larger cells are far enough away from the particle highlighted in red such that they can be approximated as a single particle at positions shown by the yellow triangle. From http://portillo.ca/nbody/barnes-hut (n.d)

As discussed previously, there are several improvements that could be made to the provided code. Some of these improvements are beyond the scope of this project, but could be implemented were it to be used in an actual experimental setting. There is always room for optimisation in programming, however it is not always an efficient use of time to pursue optimisation endlessly. It may take more time to implement an optimisation feature than it saves in being more efficient. All of the components discussed here provide good returns on the amount of labour that should reasonably be required to implement them, and an algorithm which utilises all of them should meet all of the requirements needed to prove to be an invaluable tool in experimental astrophysics.

REFERENCES

Arnold, B. (2019) N-body simulations of star forming regions. University of Sheffield.

Mathews, J.H. and Fink, K.D. (2004). Numerical methods using MATLAB. Upper Saddle River, N.J. ; London: Pearson Prentice Hall.

Dierickx, M. and Portillo, S. (n.d.). Barnes-Hut | N-Body Building. [online] Portillo.ca. Available at: http://portillo.ca/nbody/barnes-hut/ [Accessed 13 Dec. 2021]

Limousin, M., Sommer-Larsen, J., Natarajan, P. and Milvang-Jensen, B. (2009). PROBING THE TRUNCATION OF GALAXY DARK MATTER HALOS IN HIGH-DENSITY ENVIRONMENTS FROM HYDRODYNAMICALN-BODY SIMULATIONS. The Astrophysical Journal, 696(2), pp.1771–1779. doi:10.1088/0004-637x/696/2/1771.

N-body simulations of planet formation via pebble accretion - I. First results Soko Matsumura, Ramon Brasser, Shigeru Ida AA 607 A67 (2017) DOI: 10.1051/0004-6361/201731155

APPENDIX A: REFLECTION ON PROJECT PLAN

For this project I progressed very much in line with the aims of the project plan as initially set out. The main core algorithm was complete by the end of semester 1, including the predictor corrector

scheme being fully functional and the variable timestep in place. This left semester 2 to implement the 'extras' as initially planned.

OMP parallelisation ended up causing some issues related to using a Windows operating system as a coding platform, though this was anticipated. This had a minimal effect on the completion of the rest of the project as this is not an integral feature and its benefits are very predictable and well understood, hence there was no need to waste any more time on it, as this would likely have required testing the algorithm on an entirely new operating system.

N-Body units also caused some issues with implementation, as an unforeseen bug was encountered. Since this is such a minor improvement to the program, this feature was simply dropped instead of spending valuable time trying to debug the problem for mininmal benefits.

APPENDIX B: FORTRAN CODE

The actual FORTRAN90 code for the algorithm is attached below this page.

This paper has been typeset from a TEX/LATEX file prepared by the author.

```
program NBody
!Start Program
!Version 2
!Variable Declaration
Implicit none
!Define Variables
doubleprecision :: r(0:6,0:2,0:2),v(0:6,0:2,-6:2),a(0:6,0:2,-6:2), ai(0:6,0:2,0:1),
vi(0:6,0:2,0:1),pcu, munit,runit,tunit,vunit
doubleprecision :: G, M(0:6),s(0:6,0:6), d(0:6,0:6,0:2), AU,
E0,E1,Vabs_Squared(0:6),COM(0:2),Mtot,Cov(0:2), Aerr, Verr
doubleprecision :: P(0:6), Theta(1:6), pi, Msol, TimeFactor, CurrentTime,Step, year, Small,
relerr,length,PCerror,start,finish
Integer(8) :: i,j,k,t,done, n, Logging, counter, frac,systime
!Assign constant values
Do
n = 7
t = 0
year = 3600.*24.*365.25
AU = 1.496e11
G = 1*6.67408e-11
pi = 4.*atan(1.)
counter = 0
Small = 1.0e-10
relerr = 5.0e-12
frac = 0
!
!pcu=3.086d18
!munit = 1.99e33
!runit = AU
!tunit=SQRT((runit*pcu**3)/(G*munit*Msol))/year
!vunit=runit*pcu/(tunit*year*1.d5)
!write(6,*) 'tunit/yrs', tunit
!write(6,*) 'vunit/km/s', vunit

!Randomise Starting angles in xy
Call init_random_seed()
Call random_number(Theta)
Theta = Theta * 2 * pi
Msol = 1.99e30
Timefactor = 1.
!Indexing of Planets In the following order: Sun, Earth, Jupiter, Mars, Saturn, Uranus, Neptune
CurrentTime = 0.
!Masses of bodies
M(0) = 1.*Msol
M(1) = 1.*5.97e24
M(2) = 1.*1.898e27
M(3) = 1.*6.4171e23
M(4) = 5.683e26
M(5) = 8.681e25
M(6) = 1.*1.024e26

v = 0.
!Starting Velocities
v(0,0,0) = 0.
v(0,1,0) = 0.
v(0,2,0) = 0.
v(1,0,0) = 29780. * -1. * cos(Theta(1))
v(1,1,0) = 29780. * -1. * sin(Theta(1))
v(1,2,0) = 0.
v(2,0,0) = 13060. * -1. *cos(Theta(2))
```

```
v(2,1,0) = 13060. * -1. *sin(Theta(2))
v(2,2,0) = 0.
v(3,0,0) = 24070. * -1. *cos(Theta(3))
v(3,1,0) = 24070. * -1. *sin(Theta(3))
v(3,2,0) = 0.
v(4,0,0) = 9680. * -1. *cos(Theta(4))
v(4,1,0) = 9680. * -1. *sin(Theta(4))
v(4,2,0) = 0.
v(5,0,0) = 6800. * -1. *cos(Theta(5))
v(5,1,0) = 6800. * -1. *sin(Theta(5))
v(5,2,0) = 0.
v(6,0,0) = 5430. * -1. *cos(Theta(6))
v(6,1,0) = 5430. * -1. *sin(Theta(6))
v(6,2,0) = 0.

!Starting positions
r = 0.
r(0,0,0) = 0.
r(0,1,0) = 0.
r(0,2,0) = 0.
r(1,0,0) = AU * sin(Theta(1)) * -1.
r(1,1,0) = AU * cos(Theta(1))
r(1,2,0) = 0.
r(2,0,0) = 5.2*AU * sin(Theta(2)) * -1.
r(2,1,0) = 5.2*AU * cos(Theta(2))
r(2,2,0) = 0.
r(3,0,0) = 1.52*AU * sin(Theta(3)) * -1.
r(3,1,0) = 1.52*AU * cos(Theta(3))
r(3,2,0) = 0.
r(4,0,0) = 9.583*AU * sin(Theta(4)) * -1.
r(4,1,0) = 9.583*AU * cos(Theta(4))
r(4,2,0) = 0.
r(5,0,0) = 19.201*AU * sin(Theta(5)) * -1.
r(5,1,0) = 19.201*AU * cos(Theta(5))
r(5,2,0) = 0.
r(6,0,0) = 30.07 * AU * sin(Theta(6)) * -1.
r(6,1,0) = 30.07 * AU * cos(Theta(6))
r(6,2,0) = 0.

!Ask for length of time sim should run for
Write(6,*) ("Enter Sim Duration (Years)")
Read *, length



!Empty Arrays
a = 0.
s = 0.
d = 0.
P = 0.
Vabs_Squared = 0.
COM = 0.
Mtot = 0.
Cov = 0.




!Calculate centre of mass / centre of velocity information
Do i = 0,n-1
        Do k = 0,2
```

```
                COM(k) = COM(k) + r(i,k,0)*M(i)
                Cov(k) = Cov(k) + v(i,k,0)*M(i)
        End Do
Mtot = Mtot + M(i)
End Do

!Correct for COM and COV
Do i = 0,n-1
    r(i,:,0) = r(i,:,0) - COM/Mtot
    v(i,:,0) = v(i,:,0) - Cov/Mtot
    Do k = 0,2
        Vabs_Squared(i) = Vabs_Squared(i) + v(i,k,0)**2
    End Do
End Do

!Calculate Initial Conditions
    Do i = 0,n-1
             !For each body
             Do j = 0,n-1
                 !For each other body
                 If (i /= j) then
                 !Ignore self
                     !Calculate separation from vectors
                     s(i,j) = ((r(i,0,0)-r(j,0,0))**2 + (r(i,1,0) -r(j,1,0))**2 + (r(i,2,0) -
r(j,2,0))**2)**0.5
                     !Running total GPE
                     P(i) = P(i) - G*M(i)*M(j)/s(i,j)
                     !Find vectors i -> j
                     d(i,j,:) = r(j,:,0) - r(i,:,0)
                     !Calculate forces in dimension k, add to force vector array
                     a(i,:,0) = a(i,:,0) + (G*M(j)/(s(i,j))**2)*d(i,j,:)/s(i,j)
                 End If
             End Do
    End Do
P = P/2.


Write(6,*) ""

!Report initial energy of system
write(6,*) "Initial Energy of System (J):"
E0 = 0
Do i = 0,n-1
    E0 = E0 + 0.5*M(i)*Vabs_Squared(i) + P(i)
End Do
Write(6,*) E0
!Small timestep while bootstrapping
Step = 1
!Bootstrap into ABM using 2nd order taylor expansion
Do k = 0,3
    !Find new r pos after time-step
    r(:,:,1) = r(:,:,0) + v(:,:,0)*step + 0.5*a(:,:,0)*step**2

    !Reset arrays for current loop
    s = 0.
    d = 0.
    a(:,:,1) = 0.

    !Accleration calculation loop
    Do i = 0,n-1
        !For each other body
        Do j = 0,n-1
```

```
            If (i /= j) then
                !Find absolute distance between bodies i and j
                s(i,j) = ((r(i,0,1)-r(j,0,1))**2 + (r(i,1,1) -r(j,1,1))**2 + (r(i,2,1) -
r(j,2,1))**2)**0.5
                !For each dimension xyz
                !Find vectors i -> j
                d(i,j,:) = r(j,:,1) - r(i,:,1)
                !Calculate field strength in dimension, add to acceleration vector array
                a(i,:,1) = a(i,:,1) + (G*M(j)/(s(i,j))**2)*d(i,j,:)/s(i,j)
            End if
        End Do
    End Do

    !Update velocity using newfound acceleration
    v(:,:,1) = v(:,:,0) + 0.5*(a(:,:,0) + a(:,:,1))*Step

    !Shift all datapoints backwards in time one step
    a(:,:,-6:0) = a(:,:,-5:1)
    v(:,:,-6:0) = v(:,:,-5:1)
    !Keep track of time elapsed
    t = t+1
    CurrentTime = CurrentTime + Step
    counter = counter + 1

End Do

!Ask for user defined parameters

Write(6,*) ""
Write(6,*) "Log Data to Files? (Type 1 for yes, 0 for no)"
read *, Logging
Write(6,*) ""

!Open Log Files
If (Logging == 1) then
    open(2,file = 'Sun Motion.csv')
    open(3,file = 'Earth Motion.csv')
    open(4,file = 'Jupiter Motion.csv')
    open(7,file = 'Mars Motion.csv')
    open(8,file = 'Saturn Motion.csv')
    open(9,file = 'Uranus Motion.csv')
    open(10,file = 'Neptune Motion.csv')
End IF


!Begin Simulation
systime = Time()
Call cpu_time(start)
Do while ((CurrentTime / year) < length)
    s = 0.
    d = 0.

    !Predict r pos using ABM predictor
    r(:,:,1) = r(:,:,0) + (Step/24.) * (-9.* v(:,:,-3) +37.* v(:,:,-2) -59. * v(:,:,-1) +55. *
v(:,:,0))
    v(:,:,1) = v(:,:,0) + (Step/24.) * (-9.* a(:,:,-3) +37.* a(:,:,-2) -59. * a(:,:,-1) +55. *
a(:,:,0))
    !Reset predicted acceleration values for current calculation
    a(:,:,1) = 0.

    !Acceleration Loop
    !For each body
```

```
    Do i = 0,n-1
        !For each other body
        Do j = 0,n-1
            If (i /= j) then
                !Find absolute distance between bodies i and j
                s(i,j) = ((r(i,0,1)-r(j,0,1))**2 + (r(i,1,1) -r(j,1,1))**2 + (r(i,2,1) -
r(j,2,1))**2)**0.5
                !For each dimension xyz
                !Find vectors i -> j
                d(i,j,:) = r(j,:,1) - r(i,:,1)
                !Calculate field strength in dimension k, add to acceleration vector array
                a(i,:,1) = a(i,:,1) + (G*M(j)/(s(i,j))**2)*d(i,j,:)/s(i,j)
            End if
        End Do
    End Do

    !Correct using predicted values using ABM corrector
    r(:,:,2) = r(:,:,0) + (Step/24.) * ( v(:,:,-2) - 5.* v(:,:,-1) + 19.*v(:,:,0) + 9.* v(:,:,1) )
    v(:,:,2) = v(:,:,0) + (Step/24.) * ( a(:,:,-2) - 5.* a(:,:,-1) + 19.*a(:,:,0) + 9.* a(:,:,1) )

    a(:,:,2) = 0
    !Corrected Acceleration Loop
    Do i = 0,n-1
        !For each other body
        Do j = 0,n-1
            If (i /= j) then
                !Find absolute distance between bodies i and j
                s(i,j) = ((r(i,0,2)-r(j,0,2))**2 + (r(i,1,2) -r(j,1,2))**2 + (r(i,2,2) -
r(j,2,2))**2)**0.5
                !For each dimension xyz
                !Find vectors i -> j
                d(i,j,:) = r(j,:,2) - r(i,:,2)
                !Calculate field strength in dimension k, add to acceleration vector array
                a(i,:,2) = a(i,:,2) + (G*M(j)/(s(i,j))**2)*d(i,j,:)/s(i,j)
            End if
        End Do
    End Do


    !Shift all datapoints one step backwards
    v(:,:,-6:0) = v(:,:,-5:1)
    a(:,:,-6:0) = a(:,:,-5:1)
    r(:,:,0) = r(:,:,2)
    v(:,:,0) = v(:,:,2)
    a(:,:,0) = a(:,:,2)


    !Find the largest discrepancy between predictor and corrector
    PCerror = (19./270.) * (maxval(  abs( r(:,:,2) - r(:,:,1) )/(abs(r(:,:,2))+ Small )  ))
    Aerr = (19./270.) * (maxval(  abs( a(:,:,2) - a(:,:,1) )/(abs(a(:,:,2))+ Small )  ))
    Verr = (19./270.) * (maxval(  abs( v(:,:,2) - v(:,:,1) )/(abs(v(:,:,2))+ Small )  ))

    PCerror = max(Aerr,PCerror)
    PCerror = max(Verr,PCerror)


    !Check if predictor and corrector agree to overly precise level
    If ((PCerror <(relerr * 0.01))) then
        !Ensure there are enough previous points in memory
        If ((counter >= 7)) then

            !Omit every second historic point
```

```
            a(:,:,-1) = a(:,:,-2)
            a(:,:,-2) = a(:,:,-4)
            a(:,:,-3) = a(:,:,-6)

            v(:,:,-1) = v(:,:,-2)
            v(:,:,-2) = v(:,:,-4)
            v(:,:,-3) = v(:,:,-6)

            !Double the timestep and reset the counter
            Step = Step * 2.
            counter = 0
            !write(6,*) "Growing dt to ", Step, t, (maxval(abs(r(:,:,2) - r(:,:,1))/
(abs(r(:,:,2)) + Small)) )
        End If
    else

       If (PCerror > relerr) then
            If (counter >= 4) then
                ai(:,:,0) = (1./128.) * (-5.* a(:,:,-4) + 28.* a(:,:,-3) - 70.* a(:,:,-2) + 140.*
a(:,:,-1) + 35.* a(:,:,0))
                ai(:,:,1) = (1./128.) * (3.* a(:,:,-4) - 20.* a(:,:,-3) + 90.* a(:,:,-2) + 60.*
a(:,:,-1) - 5.* a(:,:,0))

                vi(:,:,0) = (1./128.) * (-5.* v(:,:,-4) + 28.* v(:,:,-3) - 70.* v(:,:,-2) + 140.*
v(:,:,-1) + 35.* v(:,:,0))
                vi(:,:,1) = (1./128.) * (3.* v(:,:,-4) - 20.* v(:,:,-3) + 90.* v(:,:,-2) + 60.*
v(:,:,-1) - 5.* v(:,:,0))

                a(:,:,-2) = a(:,:,-1)
                a(:,:,-1) = ai(:,:,0)
                a(:,:,-3) = ai(:,:,1)

                v(:,:,-2) = v(:,:,-1)
                v(:,:,-1) = vi(:,:,0)
                v(:,:,-3) = vi(:,:,1)


                Step = Step * 0.5
                counter = 0
                !write(6,*) "Shrinking dt to", Step,t, (maxval(abs(r(:,:,2) - r(:,:,1))/
(abs(r(:,:,2)) + Small)))
            end if
        End If
    End If
!
    counter = counter + 1

    !Write Every 2000th step to log files
    If ((Mod(t,2000) == 0) .and. (Logging == 1)) then

        P = 0.
        Vabs_Squared = 0.
        !For each body
        Do i = 0,n-1
                !For each other body
                Do j = 0,n-1
                !Ignore self
                If (i /= j) then
                    !Find absolute separation from position vectors
                    s(i,j) = ((r(i,0,0)-r(j,0,0))**2 + (r(i,1,0) -r(j,1,0))**2 + (r(i,2,0) -
r(j,2,0))**2)**0.5
                    !Sum calculated GPE
```

```
                    P(i) = P(i) - G*M(i)*M(j)/s(i,j)
                End if
                End Do
                !Sum kinetic energy in each dimension (k)
                Do k = 0,2
                    Vabs_Squared(i) = Vabs_Squared(i) + v(i,k,0)**2
                End Do
        End Do
        !Half to account for doubling up of counting in loop
        P = P/2.
        E1 = 0.
        !Sum total energy
        Do i = 0,n-1
            E1 = E1 +   0.5*M(i)*Vabs_Squared(i) + P(i)
        End Do

        Write(2,*) CurrentTime,',', d(0,0,0),',',d(0,0,1),',',d(0,0,2),',', s(0,0),',', E1/E0
        Write(3,*) CurrentTime,',', d(0,1,0),',',d(0,1,1),',',d(0,1,2),',', s(0,1)
        Write(4,*) CurrentTime,',', d(0,2,0),',',d(0,2,1),',',d(0,2,2),',', s(0,2)
        Write(7,*) CurrentTime,',', d(0,3,0),',',d(0,3,1),',',d(0,3,2),',', s(0,3)
        Write(8,*) CurrentTime,',', d(0,4,0),',',d(0,4,1),',',d(0,4,2),',', s(0,4)
        Write(9,*) CurrentTime,',', d(0,5,0),',',d(0,5,1),',',d(0,5,2),',', s(0,5)
        Write(10,*) CurrentTime,',', d(0,6,0),',',d(0,6,1),',',d(0,6,2),',', s(0,6)


    End If

!Keep track of how much time has elapsed
CurrentTime = CurrentTime + Step
t = t + 1

!Mechanism for printing out a progress update in 20% increments
!Can probably be done in a better way
If (CurrentTime / (length * year) > 0.2) Then
    If (CurrentTime / (length * year) > 0.4) Then
        If (CurrentTime / (length * year) > 0.6) then
            If (CurrentTime / (length * year) > 0.8) then
                If (frac == 6) then
                    Write(6,*) "80% Done"
                    frac = 8
                End If
            else
            If (frac == 4) then
                    Write(6,*) "60% Done"
                    frac = 6
                End If
            End If
        else
            If (frac == 2) then
                    Write(6,*) "40% Done"
                    frac = 4
            End If
        End If
    else
        If (frac == 0) then
            Write(6,*) "20% Done"
            frac = 2
        End If
    End If
End If
End Do
Write(6,*) "100% Done!"
```

```
!Close Log files
If (Logging == 1) then
    close(2)
    close(3)
    close(4)
    close(7)
    close(8)
    close(9)
    close(10)
End if

!State Final Conditions
Write(6,*) ""
Write(6,*) "Final Conditions"
Write(6,*) ""
Write(6,*) "Time Elapsed", CurrentTime / year, "Years"
Write(6,*) ""
Write(6,*) "Absolute distance to Sun (AU)"
Write(6,*) "Sun",s(0,0)/AU
Write(6,*) "Earth",s(0,1)/AU
Write(6,*) "Mars",s(0,3)/AU
Write(6,*) "Jupiter",s(0,2)/AU
Write(6,*) "Saturn",s(0,4)/AU
Write(6,*) "Uranus",s(0,5)/AU
Write(6,*) "Neptune",s(0,6)/AU


!Report Final Energy of System / Compare to Initial

!Loop through each body to find GPE of each
P = 0.
Vabs_Squared = 0.
!For each body
Do i = 0,n-1
        !For each other body
        Do j = 0,n-1
        !Ignore self
        If (i /= j) then
            !Find absolute separation from position vectors
            s(i,j) = ((r(i,0,0)-r(j,0,0))**2 + (r(i,1,0) -r(j,1,0))**2 + (r(i,2,0) -
r(j,2,0))**2)**0.5
            !Sum calculated GPE
            P(i) = P(i) - G*M(i)*M(j)/s(i,j)
        End if
        End Do
        !Sum kinetic energy in each dimension (k)
        Do k = 0,2
            Vabs_Squared(i) = Vabs_Squared(i) + v(i,k,0)**2
        End Do
End Do
!Half to account for doubling up of counting in loop
P = P/2.
E1 = 0.
!Sum total energy
Do i = 0,n-1
    E1 = E1 +   0.5*M(i)*Vabs_Squared(i) + P(i)
End Do

!Report final energy and accuracy test
Write(6,*) ""
write(6,*) "Final Energy of System (J)"
write(6,*) E1
```

```
Write(6,*)
Write(6,*) "Percentage of Initial Energy Retained (Indicative of Sim Accuracy)"
write(6,*) E1/E0 * 100
Write(6,*) Step
Write(6,*) t
!Stops program from closing automatically
Write(6,*) ""
Write(6,*) "Clock Time (s) ~", (Time() - systime)
call cpu_time(finish)
Write(6,*) "CPU time (s)", finish - start
Write(6,*) ""
End Do
read *, done



!Subroutine for seeding the random number generator
!From https://gcc.gnu.org/onlinedocs/gcc-4.6.1/gfortran/RANDOM_005fSEED.html
Contains
subroutine init_random_seed()

      INTEGER :: i, n, clock
      INTEGER, DIMENSION(:), ALLOCATABLE :: seed

      CALL RANDOM_SEED(size = n)
      ALLOCATE(seed(n))

      CALL SYSTEM_CLOCK(COUNT=clock)

      seed = clock + 37 * (/ (i - 1, i = 1, n) /)
      CALL RANDOM_SEED(PUT = seed)

      DEALLOCATE(seed)
end



end program NBody
```