Digital Orrery

Mark Cummins (Balliol)
Candidate No. 20871

27th April 2004
Abstract

The task for this 3rd Year ECS project was to construct a graphical simulation of the solar system, intended for use by the general public. This report outlines the development of the back end of the simulation. Our goal was to achieve a flexible, fast and accurate simulation suitable for real-time graphical display. In the process we develop a tailored force determination algorithm, investigate several numerical integration schemes and provide support for various appealing visualizations of our data.
Contents

1 Introduction ................................................. 1
   1.1 Problem Definition ................................. 1
   1.2 Partitioning the Problem ......................... 2
   1.3 Project Management ................................. 2
   1.4 Existing Systems ..................................... 2
   1.5 Objective .............................................. 3
   1.6 Language Choice ....................................... 3
   1.7 Structure of the Report ............................... 3

2 Defining the Task .......................................... 4
   2.1 Feature List ............................................ 4
   2.2 Interface Specification ............................... 4

3 Design of the Simulation .................................. 5
   3.1 Simulation Model ..................................... 5
      3.1.1 Physical Model ................................. 5
          3.1.1.1 Newton vs. General Relativity ............ 5
          3.1.1.2 Tidal Effects ............................ 5
          3.1.1.3 Non-Gravitational Effects ............... 6
      3.1.2 System of Units .................................. 6
          3.1.2.1 Time ..................................... 6
          3.1.2.2 Other Units .............................. 6
   3.2 Class Structure ....................................... 7

4 Force Algorithm ............................................. 8
   4.1 Direct Summation ..................................... 8
      4.1.1 Tweaks to Direct Summation .................... 9
   4.2 Hierarchical Force Determination ................. 10
      4.2.1 Strategies ..................................... 10
      4.2.2 Grouping Algorithm ............................ 12
      4.2.3 PseudoBodies and Force Estimation .......... 13
   4.3 Collisions .......................................... 15

2
5 Numerical Integrators 17
5.1 Introduction .................................. 17
5.2 Accuracy Criteria ............................. 18
5.3 Simple Euler ................................ 19
5.4 Leapfrog ..................................... 19
5.5 Time Symmetry ............................... 21
5.6 Hermite Predictor-Corrector Scheme .... 23
5.7 Further Integrator Issues ................. 23
5.7.1 Variable Timesteps ...................... 23
5.7.2 Individual Timesteps .................... 25
5.7.3 Time Control and the Front End ...... 25

6 Further Aspects of the Simulation 26
6.1 Rotation .................................. 26
6.2 Orbit Visualization ......................... 27
6.2.1 Strategy ................................ 27
6.2.2 Implementation .......................... 28
6.2.2.1 Finding the Central Body ........... 29
6.2.2.2 Determining the Equation of the Orbit 30
6.2.2.3 Plotting the Orbit ................... 31
6.3 Ephemerides ............................... 32
6.3.1 The JPL Ephemeris ....................... 32
6.3.2 JupiDate with a Custom Simulation .... 32
6.3.3 Other Ephemerides ....................... 33

7 Conclusions and Evaluation 34
7.1 Achievements ............................... 34
7.2 Further Developments ...................... 34
7.3 Problems ................................ 34
7.4 Parting Remark ............................ 35

8 Acknowledgements 36

A Test Results 39
A.1 Force Algorithm ......................... 39
A.1.1 Grouping Stage ......................... 39
A.1.2 Force Calculation Stage ............... 41

B Specification of the Back End 46
B.1 Feature List for the Back End .......... 46
B.2 Public Interface of the Back End ...... 47
B.2.1 Overview of the Class Structure ..... 47
B.2.2 The Vec3 Class ......................... 47
B.2.3 The BasicBody Class ................. 48
B.2.4 The Body Class ......................... 48
B.2.5 The PseudoBody Class .............................. 49
B.2.6 The Orbit_Parameters Class ...................... 49
B.2.7 The TimeAndDate Class ............................ 51
B.2.8 The Universe Class ............................... 51

C Code 54
C.1 Force_Algorithm.cpp ............................... 54
C.2 Force_Tree_Algorithm.cpp ......................... 55
C.3 Integrator.cpp .................................... 59
Chapter 1

Introduction

Figure 1.1: A mechanical orrery.

1.1 Problem Definition

The task for this project was to produce a "digital orrery". Originally an orrery was a device that illustrated the motion of the planets and their moons by means of an arrangement of mechanical gears. Our aim is to create a similar simulation on a PC; a program that will display the motion of the solar system in a way
that is attractive and appealing to a casual observer. Though accuracy is an important factor, our aim is to please the lay observer, so our design must prize an intuitive display and responsiveness to the user over accuracy of simulation.

1.2 Partitioning the Problem

As this was a joint project, our first task was to partition the problem so that each partner had a clearly defined area of responsibility. Happily, in this case there is a clear division; into the *frontend* which consists of the display engine, and the *backend* which handles the details of the physical simulation. Each of these is a substantial programming task, and we judged that this split provided an equitable division of labour, with plenty of scope for both partners. My partner Oli Wright worked on the front end, while I developed the backend.

1.3 Project Management

With such a clear split in the areas of responsibility, project management was not troublesome. After sitting down together to agree the large scale design choices, and having defined an interface for the backend, the development of the two sections could proceed essentially independently. During this time we exchanged code and ideas by email and with meetings when necessary. Meetings with our supervisor were arranged about every two weeks to monitor progress. The final phase of development consisted of integrating the two modules and producing a working program, for which we again worked ensemble.

1.4 Existing Systems

Not surprisingly, there already exist several programs that deserve the name “digital orrery”. Many of these are Open Source. Naturally, we would like our program to do something novel, rather than re-invent the wheel. So, our design process began by examining what was already available.

The two most prominent open source programs in this area are *Celestia* and *ORSA*. To explain how they differ requires a short introduction to how the simulation engine of a program of this type can be constructed. There are essentially two strategies.

One way is to make use of predefined paths for bodies in the system. This is the approach used by *Celestia*. The advantage of this approach is that it is computationally very cheap. The problem, however, is that the motion of the objects in the simulation is essentially hard-coded. It would not be possible, for example, to add a spacecraft and have its trajectory determined by the gravitational forces of the planets. While this is a faithful translation of the mechanical orrery into the digital domain, it seems somewhat a missed opportunity.

The other approach is to determine the motion of the bodies by direct numerical integration of the forces acting upon them. This is the approach used by
ORSAT. The advantage of this approach is that a true simulation is being carried out. Thus the user is free to modify, add to, or indeed completely re-define the physical set-up of the gravitational system being simulated. Programs based on this type of backend could simulate the solar systems of extra solar planets just as easily as our own solar system, and could easily handle the addition of spacecraft, new comets, etc., to the simulation.

1.5 Objective

Given the above, where can we improve on what already exists? The answer lies in the GUI. Celestia provides a high-quality 3D display engine, much like high-end computer games. However, its simulation is somewhat limited. ORSA, on the other hand, provides a powerful and flexible simulation, but is geared toward scientific data analysis. Its presentation style resembles Matlab. As far as we are aware, there is no program available that provides a 3D graphical game-like display driven by a numerical-integration based backend. This is what we set out to develop in this project.

The result should be useful as an educational tool. What would happen if the Moon were twice as massive? If a planet were added or removed from the solar system? If we orbited a binary star? Our program should allow the user to engage easily in this kind of exploration, and have the results presented immediately and in an appealing way. We have somewhat expanded upon our original aim of translating an orrery into a computer program. The objective now is to provide an engaging educational tool for the exploration of the physics of gravitational systems.

1.6 Language Choice

Language choice was relatively simple. As our front-end is heavily focused on 3D rendering, C++, with the mature OpenGL 3D graphics library, seemed an obvious choice. Java does provide some 3D graphics capabilities in the Java3D library, but it is neither as extensive nor as fast as OpenGL. This choice having been made for the front end, it was natural to write the backend in C++ also. The speed advantage provided by C++ was also important, as the simulation is quite computationally demanding.

1.7 Structure of the Report

This concludes the shared introduction. The following chapters outline the development of the backend. I begin by addressing what services the backend will be expected to provide, and define its public interface. I then discuss the modeling decisions, and the machinery that implements them, before moving on to an evaluation and conclusion.
Chapter 2

Defining the Task

A starting point in the design process was to define the task of the backend, what features it will be expected to support, and its public interface. Essentially the job of the backend is to handle the details of the simulation for the front end. It should deal with such issues as storing data about the bodies in the system, calculating their position at a given time, and monitoring any errors in the physical simulation. How complex it needs to be depends on exactly what features we want our program to provide. An early part of our design process was a brain-storming session to identify desirable features.

2.1 Feature List

Rather than produce a formal specification at this point, we decided to produce a list of target features. Due to constraints of time, it did not seem likely that we would be able to implement every feature. Nevertheless, consideration was given in the design of the system to their inclusion. The framework created should be such that no major changes should be need to support them. Our original feature list is reproduced in Appendix B.1.

2.2 Interface Specification

From this feature list, the backend was more formally specified, and an interface was agreed. This interface is outlined in Appendix B.2.
Chapter 3

Design of the Simulation

This chapter discusses the modeling and representation decisions that had to be made before beginning to code the simulation. Chapters 4, 5 and 6 go on to discuss the internal mechanics of our simulation engine.

3.1 Simulation Model

3.1.1 Physical Model

One of the first issues in designing a simulation is to decide exactly what aspects of the physical process to include in the model. This section outlines the major considerations.

3.1.1.1 Newton vs. General Relativity

There are two major theories of gravity, Newtonian and General Relativity. General relativity is certainly a better model, but under the conditions that exist within the solar system the predictions of the two theories are nearly indistinguishable.\(^\text{1}\)

For our purposes the Newtonian theory is an excellent model of gravity. It is also far less computationally complex than general relativity. We thus constructed our simulation on a Newtonian framework.

3.1.1.2 Tidal Effects

The motion of planetary satellites is significantly affected by tidal interaction with the planet they orbit. These effects arise because the bodies are not rigid

\(^{1}\text{There is one specific case within the solar system where the Newtonian framework is less than perfect; famously it underestimates the precession of the point of perihelion of Mercury by 43 arc-seconds per century. This was one of the facts used to justify the new theory relativity. A correction to the Newtonian formula has been developed that results in exact agreement with general relativity [1, 2]. We could incorporate this correction quite easily - however, since the effect is only apparent over long timescales, and we are aiming for speed rather than absolute precision, it was decided not to include it.}\)
spheres of uniform density, as in the Newtonian scheme, so cannot truly be considered as point masses.

Directly determining tidal effects on orbital motion is a problem of significant complexity, requiring detailed knowledge of the form and internal physical properties of the bodies. The author is not aware of any simulation that currently handles these effects dynamically.

Semi-empirical corrections to the Newtonian field can be used [2], however these are essentially a fit to observed data, and need to be developed on a case-by-case basis - the approximation used for the Earth's moon would be useless for a moon of Jupiter, for example.

It was considered that the modeling of tidal effects was too complex an issue to include in the simulation.

3.1.1.3 Non-Gravitational Effects

Several non-gravitational effects, such as course correction by spacecraft and out-gassing by comets, were considered for inclusion in the simulation. Including these effects would comprise a periodic adjustment of the bodies velocity vector, and incorporates naturally and easily into the simulation. However, due to limited available project time, these effects were never coded.

3.1.2 System of Units

3.1.2.1 Time

Calendars are complicated beasts, and normal dates are not a convenient way to represent time in our simulation. Within the backend the date is represented by serial day numbers. This is a serial numbering of days where (for obscure calendrical reasons) SDN 1 is November 25th, 4714 BC in the Gregorian calendar. Days after this are numbered sequentially. Allowing days to have a fractional component means that one number can be used to represent the time within the simulation. Most ephemerides also give data using this system.

To please our human users, dates are received from and returned to the frontend using the TimeAndDate class (see Appendix B.2.7), which represents dates in the normal way. Conversion to and from SDN makes use of some publicly available code [17].

3.1.2.2 Other Units

Since the JPL Horizons online ephemeris system was used to source much of the data for testing, it made sense to match our units to theirs. Consequently internally we use Kilometer-Kilogram-Second as our basic units. All fundamental quantities are stored in double precision.\footnote{For speed we might consider using float, but the accuracy is unacceptable, and the speed advantage is negligible on modern machines[15].}
3.2 Class Structure

The \texttt{Universe} object is the fundamental class in our simulation. To start a new simulation, the front end creates a new \texttt{Universe} object, which then provides all the methods used to control the simulation. A full listing of its methods and data members is provided in Appendix B.2.8.

Each body within the simulation is represented by an instance of the \texttt{Body} class, outlined in Appendix B.2.4. The \texttt{Universe} stores a vector containing all the \texttt{Body} objects in the simulation. Further classes and data structure will be described in the sections following.
Chapter 4

Force Algorithm

The primary task of the backend is to calculate updated positions for the bodies in the simulation when the front end calls \texttt{Advance()}. Performing the calculation is a two-step process. First, we must determine the acceleration of each particle, by calculating the gravitational force acting upon it. This is the job of the force algorithm, discussed in this chapter. Then, knowing the acceleration, we must determine the positions and velocities of the particles at the next timestep. This is the numerical integration step, discussed in Chapter 5.

The force algorithm performs the first stage of the \texttt{Advance()} process - determining the acceleration of the particles at the current timestep. For the N-Body case, the acceleration of the \(i^{th}\) particle is given by:

\[
\vec{a}_i = \sum_{j \neq i}^N \frac{GM_j}{|\vec{r}_{ij}|^3} \vec{r}_{ij}
\]

Some of our numerical integration procedures also require the first derivative of acceleration, known as the \textit{jerk}. This could be estimated from the difference in acceleration between timesteps, but it is more accurate to differentiate the above expression to get the explicit formula below:

\[
\vec{j}_i = \sum_{j \neq i}^N \frac{GM_j}{|\vec{r}_{ij}|^3} \left( \vec{v}_{ij} - \frac{3 (\vec{r}_{ij}, \vec{v}_{ij}) \vec{r}_{ij}}{|\vec{r}_{ij}|^2} \right)
\]

We will now consider various algorithms for determining the acceleration and jerk of particles.

4.1 Direct Summation

The simplest force evaluation algorithm is to directly encode the above summation. This is what the function \texttt{SimpleForceAlg()} does. We can eliminate
a little work by noting that $|\vec{r}_{ij}|^3 = |\vec{r}_{ji}|^3$, and hence cut the number of calculations in half. However, we must still evaluate the force expression for every particle pair. Since for $n$ particles there are $\frac{n(n-1)}{2}$ pairs, the algorithm is unavoidably $O(n^2)$. Evaluating the force expression is costly, since for ever particle pair we must calculate $|\vec{r}_{ij}|^3$, or in components $(x^2 + y^2 + z^2)^{\frac{3}{2}}$. This involves an expensive square root calculation and as $n$ grows it comes to dominate the running time of the entire program.

4.1.1 Tweaks to Direct Summation

We can tweak the direct summation process slightly to improve performance. We divide the bodies in the simulation into two groups - interacting and non-interacting. Non-interacting bodies are test particles that feel the pull of the gravitational field, but do not generate their own field. Consider, for example, a simulation containing the planets and a spacecraft. It would be ridiculous to calculate the pull of the spacecraft on the planets. Force calculations become $O(n^2)$, where $m$ is the number of interacting bodies only.

Where should we set the division between interacting and non-interacting bodies? It helps to have a little perspective on the masses of solar system objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>Mass (Kg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun</td>
<td>$2 \times 10^{30}$</td>
</tr>
<tr>
<td>Planet</td>
<td>$10^{24} - 10^{27}$</td>
</tr>
<tr>
<td>Pluto (Lightest Planet)</td>
<td>$1.3 \times 10^{24}$</td>
</tr>
<tr>
<td>Typical Asteroid Mass</td>
<td>$10^{17} - 10^{19}$</td>
</tr>
<tr>
<td>Deimos (Smallest Moon)</td>
<td>$1.8 \times 10^{13}$</td>
</tr>
<tr>
<td>Typical Comet Mass</td>
<td>$10^{13}$</td>
</tr>
<tr>
<td>Typical Space Probe Mass</td>
<td>$10^3$</td>
</tr>
</tbody>
</table>

We set our threshold at $10^{17} Kg$ (a factor of $10^{-5}$ smaller than the lightest planet). Bodies smaller than this have no gravitational influence in the simulation. However, this is not such a gross approximation; even at the surface of a body of this mass (assuming a typical density), the acceleration due to gravity is $< 0.07 m s^{-2}$. So even in the case of two light objects making very close approaches, in most cases the error will be negligible and well worth the increase in speed. Thus with the default threshold, comets, most asteroids, and the minor moons, will be considered as test particles.\footnote{There is a slight worry that the aggregate mass of many of these small bodies could have a significant effect on the solar system, and we would be overlooking this effect. However, the total mass of all asteroids is only 2% of that of the Moon, and about 75% of this mass is concentrated in the 5 largest asteroids, whose mass is above our cutoff and hence will interact. So, even in aggregate, the light bodies are not gravitationally significant.}

The distinction between interacting and non-interacting bodies is unknown to the front end. All bodies in the simulation are stored together in the bodies vector of the Universe object. However, for speed we segregate interacting and
non-interacting bodies within this vector. This allows the force calculation routine to loop over the relevant part of the vector only, rather than looping through all the bodies and checking if they interact. The AddBody(), AlterBody() and RemoveBody() methods of Universe ensure that this structure is maintained.

4.2 Hierarchical Force Determination

The speed of the $O(n^2)$ algorithm is quite acceptable if our simulation contains only a small number of bodies. However, as our simulation grows it becomes the limiting factor on backend speed. A simulation containing most major solar system object requires 35 interacting particles. If our simulation is to scale smoothly to handle this number of particles, we require a faster force determination algorithm.

For exact determination of forces, it is hard to do better than $O(n^2)$. However, if we permit a small degree of approximation, there are large gains to be made.

4.2.1 Strategies

In the $O(n^2)$ algorithm, often the strength of many of the interactions computed will be completely negligible. For example, the motion of the Earth is dominated by the influence of, at most, 3 or 4 other objects. However, we calculate the effect of all 35. Most of these other interactions are comparatively miniscule.

A natural strategy would be to discard many of these minor influences. However, the structure of the solar system allows us to do better than this.
Figure 4.2: A remote group of particles can be replaced by a single particle.

Consider the effect on a particle of a remote, compact group of bodies. Instead of calculating the pull of each of these particles individually, we could replace the group by a single aggregate particle (a particle with the group’s total mass, located at its centre of mass or barycentre). What we mean by remote and compact will depend on how much error we are willing to tolerate in our results. In many cases the accuracy penalty involved can be very small.

Figure 4.3: Group-Group interaction approximations

The above reasoning can be extended to interactions between groups. Say we have two groups, which are both compact and mutually remote. Instead of computing the pull of the remote group on each of the local particles, we instead compute the interaction between the two barycentres. This force can then be applied to all the particles in the group. Interactions within the local group are computed directly.

Since a group can be internally subdivided into further groups, this suggests a divide-and-conquer approach to computing the gravitational interactions.
Figure 4.4: A greatest attractor tree on an example system. Planets are shown in blue, moons in grey. The graph is not connected due to the presence of binaries (e.g. Pluto/Charon and Neptune/Triton within the solar system). Links shown in red are unwanted.

### 4.2.2 Grouping Algorithm

The approach outlined relies on identifying groups of particles within our simulation. Several properties could form the basis of a grouping algorithm: physical separation, velocity, local density, gravitational attraction, etc. One approach would be to examine these properties and group objects which are in some sense close. However, this leads to the problem of thresholding. To group things which are close we then need to define some type of cutoff where close becomes far.

We decided against pursuing this route, and instead tried to exploit the natural hierarchical structure of planetary systems. Consider the graph resulting from linking each body to that body which attracts it most strongly (Fig. 4.4). Many natural grouping show up in this graph. For example, moons are grouped with their planets, etc. However, the graph contains cycles and is not connected. We would like to generate a tree structure to which we can apply a recursive force-determination scheme along the lines outlined earlier, so these features need to be eliminated from the graph.

We can obtain a much more useful graph if we change the construction criteria so that each body links to that body with greater mass that attracts it most strongly (See Fig. 4.5). Given that there are no two bodies of the same mass (a physically reasonable assumption) it is simple to show that this graph cannot contain cycles and has a unique root.

This tree gives us the kind of groupings we need for our force algorithm. It
is constructed by the `buildGreatestInfluenceTree()` method of the `Universe` class. To record this tree structure, the `Body` class contains an attribute `vector<int>` `children`, which is a list of all those bodies that are direct descendants of that body in the tree. The `Universe` object has an attribute `int rootNode` that records the index of the root.

### 4.2.3 PseudoBodies and Force Estimation

For the approximation scheme, we need to know the mass, position and velocity of the barycentres of every group. Rather than calculate these each time, we insert new `pseudobodies` into our tree to record this information (See Fig.4.6).
Figure 4.6: Our modified groupings tree. PseudoBodies are shown in white. Each node in the tree is the barycentre of the nodes below that point.

PseudoBodies do not refer to any physical body, and are invisible to the front end. They simply record the barycentre information. Like the Body class, PseudoBody extends BasicBody. Since our direct summation function uses only the attributes of BasicBody, it will also work over vectors of PseudoBody.

Once this tree has been constructed the operation of the algorithm is basically as outlines in Section 4.2.1, and is illustrated in Fig. 4.7. Starting on the level below the root, we calculate the interaction of all the bodies on that level using the $O(n^2)$ algorithm. We then examine the bodies on that level. If a body has no descendants, its acceleration has been completely determined and we need to do nothing more. Otherwise, we “trickle down” its acceleration to the bodies directly below it, then recursively call the evaluation function on that level.
Figure 4.7: Interacting groups in the tree-force algorithm (shown in green).

4.2.4 Performance and Review

With only 35 bodies in the system we were doubtful if our algorithm would offer much benefit over direct summation. However, it actually resulted in a \( \times 2.5 \) speedup, with very modest accuracy penalty. See Appendix A for full test results.

Our scheme appears to be successful in pragmatic terms, though asymptotically it’s still \( O(n^2) \). This is because, in the worst case, the grouping algorithm could determine that all the bodies in the system formed a single interacting group. With no partitioning of the system, we have a flat tree and no performance benefit. Pragmatically, however, the algorithm should offer performance gains on any system with a hierarchical structure.

That said, our grouping algorithm could be more aggressive. For example, within the solar system the Sun’s attraction is so dominant that many objects have it as their greatest attractor. As a result our tree is not very deep, with many objects on the first level. Tweaking the algorithm to further partition large blocks like this offers scope for further performance enhancements.

A final word - while this algorithm was designed from first principles, a literature review unsurprisingly revealed that many similar algorithms exist [3, 4]. In particular, the force approximation scheme is almost identical to that described by Appel [5], which was later developed into the Fast Multipole Method.

We have not come across a grouping algorithm exactly like our own, though Hut and Eisenstein’s HOP algorithm [6] bears many similarities.

4.3 Collisions

A final issue to consider regarding force evaluations is the issue of collisions. Since collisions are rare within the solar system, this issue was not considered a
priority and is currently not handled by our program; at present particles pass through each other.

However, since we already calculate the separation between particles every timestep, detecting collisions would be straightforward. Accurately modeling what happens during a collision between similarly sized objects is probably too complex an issue to include in any simulation of this type, but it would be easy to model collisions as perfectly inelastic, for example.

An alternative is to include a softening factor in the force equation e.g. 
\[ \left| \mathbf{f}_i \right| = \sum_{j \neq i}^{N} \frac{GM_j}{\left| \mathbf{r}_{ij} \right|^{1+\epsilon}} \] The small factor \( \epsilon \) is chosen so that it has negligible impact at long distances, but prevents infinite forces during collisions.
Chapter 5

Numerical Integrators

5.1 Introduction

The second step in advancing the simulation is the numerical integration step. Knowing the acceleration of bodies at $t$, we need to integrate to get their position and velocity at $t + \delta t$. There exist a large number of procedures for doing this. Even for the specific case of N-Body simulations, which method to use is far from obvious. The algorithms that exist fall into a number of general categories:

1. Predictor-Corrector Schemes
2. Multi-Step Schemes
3. Extrapolation Methods
4. Symplectic Methods

Even within these categories, the number of methods to choose from is very large. The best choice will depend on the level of speed/accuracy required, the number of particles in the simulation, the smoothness of the force field, the type of hardware the algorithm will run on and many other factors.

After a literature review, symplectic methods were ruled out as being mainly suitable for very long-term, qualitative simulations. However, each of the other classes has its proponents. In search of some guidance, we contacted some researchers in the area, however opinions differed widely. One contact, the author of NASA’s ephemeris generation software, recommended multi-step or PC schemes, but specifically warned against extrapolation methods as being inefficient. Another, the author of the well-regarded Solex program, reported excellent results with a extrapolation scheme.

In the absence of any clear favourite method, it was decided that we would attempt to implement and compare several methods. With a little care in the design of the program structure, the nature of the underlying numerical integration procedure can be changed without affecting the interface to front end. The change is also largely transparent to most of the backend methods.
5.2 Accuracy Criteria

"All our measurements and observations are only approximations to the truth."

Gauss, Theoria Motus

Having decided to implement several numerical methods, we need to establish some basis for comparison. This proves to be a difficult issue, and perhaps goes some way toward explaining why there is so little general consensus on which method is best. The underlying reason is that the N-Body problem cannot be solved analytically, except in special cases. The only way to determine the correct outcome of a simulation would be by means of experiment, which is hardly practical. Even the best numerical methods give us only approximations to the truth. Our problem is to decide which of these approximations is best.

Conservation Test

The simplest and most widely used test of accuracy is to check for conservation of energy and angular momentum. These quantities can be directly calculated from simulation data:

\[ E_i = \frac{1}{2} m_i v_i^2 + \sum_{i \neq j}^N \frac{G m_i m_j}{|r_{ij}|} \]

\[ h_i = m_i |r_{ij} \times \vec{v}_{ij}| \]

A simulation that exhibits changes in energy or angular momentum is clearly behaving unphysically. However, the fact that these quantities are preserved is not a guarantee that our simulation is physically realistic. By way of illustration, an algorithm that left the position and velocity of all particles unchanged would exhibit exact conservation of energy and angular momentum, yet would be entirely unphysical. There are other, more subtle ways in which integrators can behave unphysically yet leave these properties unchanged. Despite this, the conservation test was the primary accuracy criteria employed for this project.

Analytic Special Cases Test

Another approach is to make use of the analytic special cases where the solution is known exactly, e.g., the \( N = 2 \) case. We made some use of this test - however, it is not justified to assume that the behaviour of the integrator in this special case will be representative of its behaviour in general.

Forward-Reverse Test

Another test is the forward-reverse test. The simulation is run forward a certain time interval, then backward to the starting time. Since the original positions
at the starting time are known, we can assess the errors made by the integrator. However, for integrators that are time-symmetric, the errors made in the reverse direction will cancel with the forward errors, making this test unsuitable. Since we made heavy use of time-symmetric schemes, this test was not used.

**Reference Solution Test**

Perhaps the best test would be to compute a reference solution using a very high precision integrator, taking a lot of computer time. This reference solution can then be used for the purposes of comparing lower-accuracy methods. However, setting up this type of test is time-consuming, and for our purposes conservation of energy and angular momentum, combined with behaviour that was not obviously unphysical, was decided to be a sufficient test.

### 5.3 Simple Euler

The very simplest numerical integration method is the first-order Simple Euler method. For comparison and testing purposes, we implemented this procedure first. The update equations for Simple Euler are:

\[
v_1 = v_0 + a_0 \delta t
\]

\[
s_1 = s_0 + \frac{\delta t}{2}(v_1 + v_0)
\]

where \( v \) is velocity and \( s \) is displacement, and the subscript indicates the time. As expected, the Simple Euler method does not perform very well. Even for small timesteps, this method exhibits large progressive energy errors. Clearly, something more advanced is required.

### 5.4 Leapfrog

The next method we implemented was the Leapfrog or Verlet method [10, 12]. The update-equations appear very similar to the Simple Euler method:

\[
s_1 = s_0 + v_1 \delta t
\]

\[
v_1 = v_0 + a_1 \delta t
\]

The above update equations, known as the interleaved form, require the position and velocity to be defined a half-timestep apart. While this gives a computationally efficient form of the scheme, it is more convenient to have both position and velocity defined at the same time. The equivalent, non-interleaved form of the scheme, shown below, is the form used by the program.

\[
s_1 = s_0 + v_0 \delta t + \frac{(\delta t)^2}{2}a_0
\]
\[ v_1 = v_0 + \frac{\delta t}{2} (a_1 + a_0) \]

This form brings out more clearly the fact that leapfrog is a second-order scheme. The performance difference compared to Simple Euler can be clearly seen in Fig. 5.1. The figure shows the results of a simple test where a light body is in a circular orbit about a massive central body. The orbital path for the two integrators is shown for 10 periods. The integrations took almost equal amounts of CPU time. However, while the leapfrog orbit is perfectly circular, the Simple Euler method has progressive energy errors which cause the orbiting planet to spiral outward.

In fact, the difference in performance is even more dramatic than the figure might suggest. Comparing the initial and final system energy:

\[
\begin{align*}
\Delta E_{\text{simple euler}} &= 0.3 \\
\Delta E_{\text{leapfrog}} &= 1.3 \times 10^{-10}
\end{align*}
\]

and similarly for angular momentum:

\[
\begin{align*}
\Delta AM_{\text{simple euler}} &= 0.19 \\
\Delta AM_{\text{leapfrog}} &= 4.1 \times 10^{-15}
\end{align*}
\]

The difference in energy error is nine orders of magnitude, and even larger for angular momentum errors! This astonishingly large difference between two apparently similar methods is due to the fact that the Leapfrog method is time-symmetric.
5.5 Time Symmetry

Some integrators have the property of time-symmetry. For a full discussion of
time symmetric integrators and their properties, see [10, 11, 8]. Informally, if a
time symmetric integrator is run in the forward time direction, and then time
is reversed, the integrator will retrace its steps. Errors are made symmetrically,
so that, except for the effects of rounding errors, the particle will return to its
starting point.

Time symmetry is a basic property of the physical laws we are simulating.
Integrators that mirror this property tend to have desirable characteristics. In
particular, they tend to exhibit very good energy conservation. To understand
why this occurs, consider the circular orbit in Fig. 5.1. Since the orbit is
symmetric, the forces experienced by the particle in the second half of its orbit
are equal and opposite to those experienced during the first half. So travelling
the second half of the orbit is essentially the same as travelling the first half of
the orbit in the time-reverse direction. Time-symmetry means the errors made
during the second half cancel with those made during the first half. As a result,
errors are periodic, and do not accumulate between orbits. This can be clearly
seen in Fig. 5.2

![Image of a graph showing energy profile for Simple Euler and Leapfrog]

Figure 5.2: Energy profile for Simple Euler and Leapfrog for the test case shown in
Fig. 5.1. The vertical scale of the Leapfrog error has been magnified by $10^7$. Observe
that the energy error for leapfrog is periodic.

The good performance characteristics of this class of integrators tends to
be exaggerated by orbits which are exactly symmetric, such as our test case.
The solar system, being composed of many near-periodic orbits, is also particu-
larly suited to integration with time-symmetric schemes. However, even in more
general cases they perform very well. The transient energy error tends to man-
ifest itself, for a periodic orbit, as a small linear drift in the time of pericentre
passage; for our purposes a relatively benign type of error.
5.6 Hermite Predictor-Corrector Scheme

Hermite scheme is a fourth-order predictor-corrector scheme [7, 9, 12]. Like the Leapfrog scheme, it is also time symmetric. The update equations are as follows:

The predictor step simply a truncated Taylor series:

\[ s_{\text{pred}} = s_0 + v_0 \delta t + a_0 \frac{(\delta t)^2}{2} + j_0 \frac{(\delta t)^3}{6} \]

\[ v_{\text{pred}} = a_0 \delta t + j_0 \frac{(\delta t)^2}{2} \]

The force algorithm is now called to determine \( a \) and \( j \) at \( s_{\text{pred}} \). This is known as the evaluator step. Finally, we apply the corrector step:

\[ v_1 = v_0 + \frac{\delta t}{2} (a_0 + a_{\text{pred}}) + \frac{(\delta t)^2}{12} (j_0 - j_{\text{pred}}) \]

\[ s_1 = s_0 + \frac{\delta t}{2} (v_0 + v_1) + \frac{(\delta t)^2}{12} (a_0 - a_{\text{pred}}) \]

We can iterate the Evaluator-Corrector stage for higher accuracy. However, each iteration requires a force evaluation, and thus is expensive. Generally, when force evaluations are expensive, the single-iteration \( PEC \) scheme was found to give the best trade-off, though \( P(\text{EC})^2 \) is preferable in situations where we want high accuracy with a small number of bodies.

The accuracy of our implementation was verified by comparison to test data from a reference implementation published by Hut and Makino [12]. For a full comparison of the speed and accuracy of this integrator with Leapfrog and Simple Euler, see Appendix A.2.

Though considerable research time had been spent investigating other integrators (especially the Radau quadrature method [13], and the Burlisch-Stoer method [15, 16]), at this point it was decided that the Leapfrog and Hermite integrators were performing so well that it was time move on to another area of the project. We had already more than met the performance goals set out in our spec.

All three implemented methods are available in the final program, and the existing program structures should support the implementation of further schemes.

5.7 Further Integrator Issues

5.7.1 Variable Timesteps

In the integrators outlined to this point, the timestep \((\delta t)\), has been kept constant during the integration. However, we are free to change the timestep as the integration progresses, and this promises greater speed and accuracy. The basic problem with fixed timesteps is outlined in Fig. 5.3. Since steps are of a
fixed time length, the distance the particle moves during a step is much greater near the central body, since the particle is travelling faster. However, this is precisely the region where the force field is changing most rapidly, so the error per step here is large. By varying the step length so that we take large steps where the force field is changing slowly, and smaller steps where it is changing rapidly, we could have a large gain in accuracy.

Many heuristics exist for choosing a new timestep. We tried a widely used one due to Aarseth [14], shown below:

\[ \Delta t_c = \sqrt{\frac{\eta |a| \cdot |a^{(2)}| + |a^{(1)}|^2}{|a^{(1)}| \cdot |a^{(3)}| + |a^{(2)}|^2}} } \]

where \( \eta \) is an accuracy parameter. Our force algorithm calculates \( a \) and \( a^{(1)} \) (given the symbol \( j \) earlier) explicitly. Since calculating the higher derivatives explicitly is an expensive operation, we estimate them in terms of \( a \) and \( j \).

\[ a^{(2)} = \frac{-6(a_0 - a_1) - \delta t(4j_0 + 2j_1)}{(\delta t)^2} \]
\[ a^{(3)} = \frac{12(a_0 - a_1) + 6\delta t(j_0 + j_1)}{(\delta t)^3} \]

While this scheme correctly varied the length of the timestep, it actually resulted in a large drop in accuracy. This surprising effect is because varying the timestep destroys the time-symmetry of the integrators and consequently their energy-conservation behaviour. Recently methods have been developed that allow time-symmetry to be restored in the variable-timestep case [10, 11].

We planned to implement this scheme, but time did not permit. Also, save for comets, orbits within the solar system are only moderately eccentric, so this issue is not critically important.
5.7.2 Individual Timesteps

As currently implemented, all bodies in the simulation move with the same global timestep. This is quite inefficient; in general some bodies will be in a region where the gravitational force is slowly varying, so could afford to take large steps, whereas others will be undergoing close encounters and will require smaller timesteps. With global timesteps all particles must move at the speed of the smallest step. Allowing individual timesteps thus offers large performance gains. However, as far as we are aware, there is no known way to maintain time symmetry with individual timesteps; though performance gains are often enough to offset this [7]. Again, this is an area that time did not permit us to explore fully.

5.7.3 Time Control and the Front End

One issue that proved tricky to resolve was the interaction of the backend and the front end regarding time control. The front end calls \texttt{Advance(T)} once per frame to update the position of bodies. The problem is that the frame-rate of the display is dependant upon the number of bodies in the field of view. If the front end called \texttt{Advance(T)} with a constant \( T \) per frame, then bodies would appear to move at varying rates depending on the number of bodies on-screen, even though their speed in the underlying simulation remains the same.

This suggests that the front end should vary the \( T \) per frame to maintain a fixed ratio between simulated time and real time. However, this presents problems too. As explained in Section 5.7.1, varying the timestep interferes with time-symmetry and destroys the accuracy of the simulation. So we cannot link \( T \) directly to the integrator timestep \( \delta t \).

Our saving grace is that the backend runs considerably faster than the front end: generally one or two orders of magnitude, and up to three orders of magnitude in some simple test cases. The solution that was developed was to have an internal timestep \( \delta t \), perhaps 10 to 100 times smaller than the typical \( T \) per frame. Then, when the front end requests an advance of \( T \), this actually causes the backend to take \( T \div \delta t \) steps (each of length \( \delta t \)). Since \( \delta t \ll T \), this is almost equal to the advance requested. This allows us to maintain an almost constant ratio between real time and simulated time, while allowing the backend to retain control over its own timestep.
Chapter 6

Further Aspects of the Simulation

6.1 Rotation

The force evaluation and numerical integration algorithms deal with updating the positions of the bodies in the system, but we must also consider their rotation. To deal with rotation, each body stores three attributes: its rotation axis, its angular velocity about that axis, and an angle by which it is currently rotated about the axis. (See details of the Body class in Appendix B.2.4). When we advance a timestep, we simply update the angle attribute by an appropriate amount. The axis and rate of rotation are fixed.

This model of rotation is very simplistic. Over long timescales, both the rotation rate and rotation axis are affected by gravitational interactions. This is the origin of such phenomenon as locked rotation of moons and the precession of the equinoxes on Earth. However, the origin of these effects is tidal gravitational interaction. As explained in Section 3.1.1.2, tidal effects were not modeled.

It would be relatively straightforward and computationally inexpensive to incorporate an empirical correction into our model to include these rotational effects without actually modeling the underlying processes. It was decided not to include these corrections. Any corrections would necessarily be tied to the present structure of the solar system. We are giving the user free reign to modify the simulation, and after any modification our empirical corrections would be invalid. Furthermore, the timescale over which the effects are noticeable is on the order of thousands of years, so not modeling these effects should not be noticeable to most users.
6.2 Orbit Visualization

6.2.1 Strategy

Another task of the backend is to support orbit visualization. There are several approaches that can be taken here. One simple solution is to have each body store a record of its past locations. This would allow a trail to be drawn, as in Fig. 6.1. This is simple to implement, and is included in the program. The number of points stored in the history, and how frequently a point is added, are controllable by the front end.

![Figure 6.1: Orbit trails.](image)

However, trails alone do not provide exactly the effect we are seeking. We would like to be able to visualize the entire orbit of the body, not just where it has been recently, as in Fig. 6.3.

To draw an orbit line, the front end needs to know many points along the orbit of the body. Some of these points will be ahead of the body’s current position. This presents a problem for the backend - after all, positions ahead of the body lie in the future and have not yet been calculated.

One possible solution would be to buffer our output, so that our “current” position lies in the middle of our position history. This would allow us to draw some of the points that lie in front of the body. However, the problem then becomes to ensure that position histories are large enough that the points stored encompass the entire orbit. Unless we are very careful to account for the
variations in speed and length of orbit for the different bodies, and adjust the size of their histories accordingly, the effect we achieve is likely to be as in Fig. 6.2.

Figure 6.2: Buffered Orbit Trails

A more satisfactory approach is to determine an analytic equation for the orbit of the body; then it is easy to calculate any number of equally spaced points along its orbit, resulting in an output as in Fig. 6.3. In an N-body situation this will not be possible in general, since only 2-body problems are analytically solvable. However, in cases where the force from one body dominates, we can approximate our N-Body problem by a collection of 2-body problems, which are then solvable analytically. This works well within the solar system; it is, after all, why Kepler’s Laws are successful.

6.2.2 Implementation

To support orbit visualizations, we require the analytic solution of the 2-body orbit that most closely matches the body’s real orbit. This task is handled by the class `OrbitParameters`, outlined in Appendix B.2.6. Each body stores its own orbit parameters, which the front end uses to draw its orbit line.
6.2.2.1 Finding the Central Body

Given a body $B$, the first thing we must determine is what the second body in our 2-body approximation will be. The best approximation is given by the body that attracts $B$ most strongly. This will be the central body of our orbit, $C$. Normally, for a planet this will be the sun, and for a moon it will be its planet, etc.\(^1\)

At this point we perform a small defensive check. Sometimes it will not make sense to draw an orbit line. This may be because $B$ cannot sensibly be said to be in orbit about anything (What does the Sun orbit within the solar system?), or because our system is not well-approximated by a collection of 2-body problems (for example, the situation in Fig. 6.4). In these cases it would be best if we did not draw any orbit line, and fell back upon the more robust orbit-trails mechanism. So, if $C$ is less massive than $B$ (case for the sun example) or if no one body dominates the behaviour of $B$ (case for the figure-of-eight orbit) then

\(^1\)Surprisingly, this is not always the case. When writing the code to determine the greatest attractor for a given body, I spent some time debugging the fact that the moon seemed to be more attracted to the Sun than to the Earth. After much head scratching, I discovered that the “bug” was in fact with my own knowledge. Somewhat unexpectedly, the sun actually attracts our moon 2.2 times more strongly than the Earth, so that in at least one important sense of the word, our moon is not really a “moon” at all. Discovering this was quite exciting: my program was beginning to teach me things about the solar system that I had not anticipated. However, it does mean that to see the orbit lines the user expects, it is sometimes necessary to force the program to pick a central body other than the one that would be chosen automatically.
we set the `validOrbit` flag to false and exit. This tells the front end to draw no orbit line for this body.

Otherwise, we will continue and determine the orbit of \( B \) with respect to \( C \).

![Orbit Diagram](image)

Figure 6.4: Some situations, such as this figure-of-eight orbit, cannot be well-approximated as a collection of two-body problems. In this situation, orbit lines are not drawn, and we revert to drawing trails only.

### 6.2.2.2 Determining the Equation of the Orbit

2-body orbits are all conic sections. The polar equation of a conic section is:

\[
\rho = \frac{l}{1 + e \cos \theta}
\]

where \( l \) is the semilatus rectum, \( e \) is the eccentricity, and \( \theta \) is the true anomaly. Knowing these parameters we can plot the orbit. We need to determine these parameters from the data available in the simulation.

The eccentricity is given by:

\[
e = \sqrt{1 + 2Eh^2}
\]

and the semilatus rectum is given by:

\[
l = \frac{h^2}{GM}
\]

where \( h \) is the body’s angular momentum, \( |\vec{r} \times \vec{v}| \), \( E \) is its specific energy, \( 0.5v^2 + \frac{GM}{r} \), \( M \) is the mass of the central body, and \( G \) is the gravitational constant. All positions and velocities are relative to the central body. These quantities can be directly obtained from the data available in the simulation.

The values of these parameters should not change very much. They are stored in an `Orbit_Parameters` attribute of a body, and re-calculated only occasionally.
Finally, to draw the orbit, we also need to know $\theta$, the present true anomaly. This will change as the body moves, so we calculated it each time we need to draw the orbit. It is given by:

$$\theta = \cos^{-1}\left(\frac{1}{e}\left(\frac{l}{|\vec{r}|} - 1\right)\right)$$

This will give us a value between 0 and $\pi$. Whether the angle between the $\vec{r}$ and $\vec{v}$ vectors is acute or obtuse resolves the ambiguity of whether we are in the first or second half of the orbit, to give us a value between 0 and $2\pi$.

### 6.2.2.3 Plotting the Orbit

Having all the above parameters, we can draw the orbit line. This is illustrated in Fig. 6.5. We know the unit radial vector and the unit vector normal to the plane of the orbit (shown in blue). We also know the current value of the true anomaly $\theta$, and using the polar equation of the orbit, we can calculate the length of the radial vector for a given value of the true anomaly.

To draw the orbit, we take the current radial vector, and incrementally rotate it through 360 degrees (using a small angle rotation matrix, in conjunction with the unit normal to the orbit). At each point, we then know the true anomaly and the unit radial vector. We calculate the orbital radius at that point from the polar equation, and hence we generate the orbit line.

![Diagram of an orbit with radial and velocity vectors](image.png)

Figure 6.5: The operation of the Plot_Orbit function.
SHOW COMPARISON BETWEEN TRUE AND PREDICTED.

6.3 Ephemerides

6.3.1 The JPL Ephemeris

We now have outlined the functioning of a working simulation. However, we are missing a vital ingredient: *data*. Our simulation, so far, is a general gravitational simulator. It knows nothing about the solar system. Our final task was to read from a standard data source.

A dataset giving the position of the bodies within the solar system is known as an *ephemeris*. Several ephemerides are publicly available, most commonly used are those published by the JPL at NASA, and the Bureau De Longitudes in Paris. Alternative sources publish ephemerides for comets, asteroids, etc. We chose to use the JPL DE406 ephemeris, which gives the positions of the nine planets, plus Earth’s moon, over the interval 3000 BC to 3000 AD, to an accuracy of 25m (and 1m for the Moon). The file provides exact positions every 64 days, and Chebyshev polynomials to interpolate between these points, so that the ephemeris can generate a position for any date within its range. The size of the data file is approx 5 MB per 150 years. C++ code to read from the NASA datafiles is publicly available [18]. This code, with minor modification, was incorporated into the project for this task.

Reading from the dataset is controlled by the `JumpToDate(TimeAndDate &t)` method of the `Universe` class. A call to this method causes the backend to reset simulation, and reinitialize at the new date by reading from the ephemeris.

6.3.2 JumpToDate with a Custom Simulation

One somewhat thorny question arises when considering the `JumpToDate()` function. What behaviour is expected when the user has added custom bodies to the solar system? The ephemeris knows positions for the standard solar system bodies only, and cannot know where the custom body will be on the requested date.

**Possible Behaviours:**

1. Custom bodies are removed from the simulation.

2. Ignore the ephemeris and try to run the simulation backward or forward to the requested date. While this will certainly work, for a long jump of 1000 years, say, it could easily take 30 minutes or more. The user is unlikely to be pleased by our diligence!

3. Planets move to new ephemeris positions for the date, custom bodies stay where there are. Potentially, the custom bodies might end up inside a planet!
While not ideal, our function implements option 1, as being the most acceptable alternative. There is scope for more sophisticated handling of this problem - perhaps presenting the above alternatives to the user, etc.

6.3.3 Other Ephemerides

The JPL ephemeris contains no data for the natural satellites of the planets (e.g. the moons of Jupiter, etc). While this data is available from other sources, it exits in a variety of different file formats. Having already included one ephemeris in the program, the work involved in dealing with these extra files was judged not to be a useful way to spend limited project time. However, in order to give our program as full a test as possible, the state vectors for all the natural satellites and several comets were hand-entered for a particular date. This seemed sufficient proof of concept - there would be no barrier to including the extra data, but with project time limited, it was decided invest it elsewhere.
Chapter 7

Conclusions and Evaluation

7.1 Achievements

Overall, I believe this project has been a success. We achieved the primary things we set out to do: a working simulation that meets our performance and accuracy goals. In the process we developed a tailored force determination scheme, investigated several numerical integration methods and provided support for various visualizations. The final simulation is fast, flexible and accurate.

Also, importantly, we maintained an arms-length separation between the frontend and backend. The backend is a general simulation engine that is entirely agnostic on the nature of the frontend (indeed for much of the development process Matlab was used as the front end).

7.2 Further Developments

While the project achieved a solid core of functionality, by nature this problem is very open ended, and there remains plenty of scope for future development. Some of the more ambitious features in our initial spec (notably the gravity map and eclipse/conjunction prediction) could still be developed. Also, as noted in the body of the report, there are several avenues open for improving the force algorithm and the integration schemes. Finally, many simple features, such as load/save, etc, need to be implemented to improve the user experience.

7.3 Problems

They say, best men are moulded out of faults,
And, for the most, become much more the better
For being a little bad.

William Shakespeare, "Measure for Measure", Act 5 scene 1
Any extended development process will run into problems, and this project was no exception. At the most basic level, a sizable proportion of project development time was spent debugging; but this is to be expected, and I don’t believe we suffered more in this department than is normal. The most problematic issue was in debugging the numerical methods; it can be hard to determine if poor behaviour is due to a coding bug or is a feature of the underlying algorithm. This was an area where I feel I learned a lot.

At a more general level, in hindsight some issues could have been handled better. Most prominent in my mind is the internal structure of the backend. I developed this project concurrently with learning C++ and taking my first course in OOP. Consequently, in retrospect, I feel the backend could do with a refactoring.

### 7.4 Parting Remark

I leave the final word to the original orbit modeller:

\[
\begin{align*}
\text{It is not knowledge, but the act of learning, not possession but the} \\
\text{act of getting there, which grants the greatest enjoyment. When I} \\
\text{have clarified and exhausted a subject, then I turn away from it, in} \\
\text{order to go into darkness again; the never-satisfied man is so strange} \\
\text{- if he has completed a structure, then it is not in order to dwell in} \\
\text{it peacefully, but in order to begin another.}
\end{align*}
\]

*Carl Friedrich Gauss, Letter to Bolyai, 1808.*

Onward to the 4th Year Project!
Chapter 8

Acknowledgements

First and foremost, we would like to thank our supervisor Jeff Sanders for providing sure and sane guidance through the wilderness, for teaching us the real meaning of design and most importantly for teaching us the skill of thinking for ourselves.

We would also like to acknowledge:

• The members of sci.comp.numanalysis and the Minor Planet Mailing List for some useful words of advice.
• Pasquale Tricarico, author of ORSA, for some tips on ephemerides.
• The maintainers of the wonderful resources that is arXiv.
• and finally the authors of all the Open Source utilities that have helped us develop this project.
Bibliography


Appendix A

Test Results

These tests were carried out on a full solar system containing 35 bodies. Data sourced from the JPL Horizons ephemeris system, for the configuration of the solar system on Jan 1st 2004. The simulation contains the sun, the planets, their major moons, and the comet Wild 2.

A.1 Force Algorithm

A.1.1 Grouping Stage

The grouping algorithm found the following partition of the solar system. Note that, with the exception of Earth’s moon, planets and their moons are correctly grouped together. The fact that Earth’s moon is not grouped with the Earth is not an error in the code, but rather due to a quirk of the Earth-Moon system. See footnote on page 29.
PseudoBody Tree

Barycentre
Wild2
The Sun
Mercury
Venus
Earth
Luna
Barycentre
Phobos
Deimos
Mars
Barycentre
Jupiter
Io
Europa
Ganymede
Callisto
Barycentre
Saturn
Mimas
Enceladus
Tethys
Dione
Rhea
Titan
Hyperion
Iapetus
Phoebe
Barycentre
Uranus
Miranda
Ariel
Umbriel
Titania
Oberon
Barycentre
Neptune
Triton
Barycentre
Pluto
Charon
A.1.2 Force Calculation Stage

This test assesses the quality of the force approximation scheme. Using the test system, we first calculated all the forces using the Direct Summation scheme. Then, we calculated the forces using the Force-Tree scheme. The force tree scheme was 2.5 times faster than the direct method. As can be seen from the table, the accuracy penalty is very slight.

**Relative Magnitude Error** The fractional difference between the magnitude of the true and approximated forces - i.e. \( \frac{|f_{\text{approx}} - |f_{\text{true}}|}{|f_{\text{true}}|} \)

**Direction Error** The angle in degrees between the true force vector and the calculated one.

<table>
<thead>
<tr>
<th>Body</th>
<th>Relative Magnitude Error</th>
<th>Direction Error (degrees)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wild 2</td>
<td>(8.7 \times 10^{-14})</td>
<td>(8.5 \times 10^{-7})</td>
</tr>
<tr>
<td>The Sun</td>
<td>(5.6 \times 10^{-10})</td>
<td>(&lt; 10^{-15})</td>
</tr>
<tr>
<td>Mercury</td>
<td>(2.7 \times 10^{-15})</td>
<td>(8.5 \times 10^{-7})</td>
</tr>
<tr>
<td>Venus</td>
<td>(3.9 \times 10^{-15})</td>
<td>(&lt; 10^{-15})</td>
</tr>
<tr>
<td>Earth</td>
<td>(1.6 \times 10^{-14})</td>
<td>(&lt; 10^{-15})</td>
</tr>
<tr>
<td>Luna</td>
<td>(8.0 \times 10^{-15})</td>
<td>(&lt; 10^{-15})</td>
</tr>
<tr>
<td>Mars</td>
<td>(3.4 \times 10^{-13})</td>
<td>(1.2 \times 10^{-6})</td>
</tr>
<tr>
<td>Phobos</td>
<td>(2.3 \times 10^{-7})</td>
<td>(5.8 \times 10^{-6})</td>
</tr>
<tr>
<td>Deimos</td>
<td>(4.9 \times 10^{-6})</td>
<td>(2.8 \times 10^{-4})</td>
</tr>
<tr>
<td>Jupiter</td>
<td>(8.9 \times 10^{-8})</td>
<td>(1.6 \times 10^{-5})</td>
</tr>
<tr>
<td>Io</td>
<td>(1.0 \times 10^{-9})</td>
<td>(1.2 \times 10^{-5})</td>
</tr>
<tr>
<td>Europa</td>
<td>(8.1 \times 10^{-7})</td>
<td>(4.2 \times 10^{-5})</td>
</tr>
<tr>
<td>Ganymede</td>
<td>(1.6 \times 10^{-6})</td>
<td>(1.3 \times 10^{-4})</td>
</tr>
<tr>
<td>Callisto</td>
<td>(1.1 \times 10^{-5})</td>
<td>(5.2 \times 10^{-4})</td>
</tr>
<tr>
<td>Saturn</td>
<td>(1.2 \times 10^{-7})</td>
<td>(1.2 \times 10^{-5})</td>
</tr>
<tr>
<td>Mimas</td>
<td>(7.7 \times 10^{-9})</td>
<td>(8.5 \times 10^{-7})</td>
</tr>
<tr>
<td>Enceladus</td>
<td>(2.6 \times 10^{-8})</td>
<td>(1.2 \times 10^{-6})</td>
</tr>
<tr>
<td>Tethys</td>
<td>(3.6 \times 10^{-8})</td>
<td>(8.5 \times 10^{-7})</td>
</tr>
<tr>
<td>Dione</td>
<td>(5.7 \times 10^{-8})</td>
<td>(3.6 \times 10^{-6})</td>
</tr>
<tr>
<td>Rhea</td>
<td>(2.9 \times 10^{-7})</td>
<td>(1.4 \times 10^{-5})</td>
</tr>
<tr>
<td>Titan</td>
<td>(1.7 \times 10^{-6})</td>
<td>(1.2 \times 10^{-4})</td>
</tr>
<tr>
<td>Hyperion</td>
<td>(4.7 \times 10^{-6})</td>
<td>(2.1 \times 10^{-4})</td>
</tr>
<tr>
<td>Iapetus</td>
<td>(5.8 \times 10^{-6})</td>
<td>(9.4 \times 10^{-4})</td>
</tr>
<tr>
<td>Phoebe</td>
<td>(3 \times 10^{-4})</td>
<td>(8.9 \times 10^{-2})</td>
</tr>
<tr>
<td>Uranus</td>
<td>(1.8 \times 10^{-8})</td>
<td>(&lt; 10^{-15})</td>
</tr>
<tr>
<td>Miranda</td>
<td>(3.2 \times 10^{-10})</td>
<td>(&lt; 10^{-15})</td>
</tr>
<tr>
<td>Ariel</td>
<td>(1.0 \times 10^{-8})</td>
<td>(&lt; 10^{-15})</td>
</tr>
<tr>
<td>Umbriel</td>
<td>(1.3 \times 10^{-8})</td>
<td>(1.7 \times 10^{-6})</td>
</tr>
<tr>
<td>Titania</td>
<td>(1.2 \times 10^{-7})</td>
<td>(3.3 \times 10^{-6})</td>
</tr>
<tr>
<td>Oberon</td>
<td>$2.3 \times 10^{-7}$</td>
<td>$1.2 \times 10^{-3}$</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Neptune</td>
<td>$8.0 \times 10^{-9}$</td>
<td>$8.5 \times 10^{-7}$</td>
</tr>
<tr>
<td>Triton</td>
<td>$2.5 \times 10^{-9}$</td>
<td>$&lt; 10^{-15}$</td>
</tr>
<tr>
<td>Pluto</td>
<td>$8.2 \times 10^{-9}$</td>
<td>$8.5 \times 10^{-7}$</td>
</tr>
<tr>
<td>Charon</td>
<td>$8.1 \times 10^{-9}$</td>
<td>$&lt; 10^{-15}$</td>
</tr>
</tbody>
</table>

### A.2 Numerical Integrators

To test the numerical integrators, we performed an integration of the test solar system for a period of 1 year, varying various parameters. The results are presented below.

**Test 1, $\delta t = 1$ Day, 1 year**

<table>
<thead>
<tr>
<th>Integrator</th>
<th>Force Scheme</th>
<th>Run Time</th>
<th>Energy Error</th>
<th>Angular Momentum Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hermite $P(\mathcal{EC})^k$</td>
<td>Direct Sum</td>
<td>1.6 secs</td>
<td>$4.4 \times 10^{-4}$</td>
<td>$4 \times 10^{-5}$</td>
</tr>
<tr>
<td></td>
<td>Tree</td>
<td>0.8 secs</td>
<td>$4.1 \times 10^{-4}$</td>
<td>$5 \times 10^{-5}$</td>
</tr>
<tr>
<td>Hermite $PEC$</td>
<td>Direct Sum</td>
<td>1.0 secs</td>
<td>$1.1 \times 10^{-3}$</td>
<td>$9.0 \times 10^{-5}$</td>
</tr>
<tr>
<td></td>
<td>Tree</td>
<td>0.5 secs</td>
<td>$8.9 \times 10^{-4}$</td>
<td>$1.0 \times 10^{-4}$</td>
</tr>
<tr>
<td>Leapfrog</td>
<td>Direct Sum</td>
<td>0.5 secs</td>
<td>$2.5 \times 10^{-4}$</td>
<td>$5.9 \times 10^{-6}$</td>
</tr>
<tr>
<td></td>
<td>Tree</td>
<td>0.2 secs</td>
<td>$2.3 \times 10^{-4}$</td>
<td>$8.0 \times 10^{-6}$</td>
</tr>
<tr>
<td>Simple Euler</td>
<td>Direct Sum</td>
<td>0.6 secs</td>
<td>$6.7 \times 10^{-3}$</td>
<td>$4.7 \times 10^{-4}$</td>
</tr>
<tr>
<td></td>
<td>Tree</td>
<td>0.3 secs</td>
<td>$6.9 \times 10^{-3}$</td>
<td>$4.9 \times 10^{-4}$</td>
</tr>
</tbody>
</table>

A number of things can be seen for the above test. Firstly, the performance gain from using the tree-based force determination scheme is considerable, and the accuracy penalty is slight. With a timestep as large as 1 Day, there is little difference in accuracy between the integrators. The difference becomes more apparent at higher levels of accuracy.

**Test 2, $\delta t = 0.02$ Day, 1 year**

<table>
<thead>
<tr>
<th>Integrator</th>
<th>Force Scheme</th>
<th>Run Time</th>
<th>Energy Error</th>
<th>Angular Momentum Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hermite $P(\mathcal{EC})^2$</td>
<td>Direct Sum</td>
<td>46 secs</td>
<td>$2 \times 10^{-12}$</td>
<td>$1.9 \times 10^{-6}$</td>
</tr>
<tr>
<td></td>
<td>Tree</td>
<td>17.8 secs</td>
<td>$4.3 \times 10^{-10}$</td>
<td>$1.9 \times 10^{-6}$</td>
</tr>
<tr>
<td>Hermite $PEC$</td>
<td>Direct Sum</td>
<td>24 secs</td>
<td>$2 \times 10^{-9}$</td>
<td>$2.0 \times 10^{-6}$</td>
</tr>
<tr>
<td></td>
<td>Tree</td>
<td>10 secs</td>
<td>$2.7 \times 10^{-9}$</td>
<td>$2.0 \times 10^{-6}$</td>
</tr>
<tr>
<td>Leapfrog</td>
<td>Direct Sum</td>
<td>14 secs</td>
<td>$2.9 \times 10^{-9}$</td>
<td>$1 \times 10^{-6}$</td>
</tr>
<tr>
<td></td>
<td>Tree</td>
<td>6.2 secs</td>
<td>$3.4 \times 10^{-9}$</td>
<td>$1.1 \times 10^{-6}$</td>
</tr>
<tr>
<td>Simple Euler</td>
<td>Direct Sum</td>
<td>14 secs</td>
<td>$3.1 \times 10^{-4}$</td>
<td>$4.7 \times 10^{-6}$</td>
</tr>
<tr>
<td></td>
<td>Tree</td>
<td>5.8 secs</td>
<td>$3.2 \times 10^{-4}$</td>
<td>$4.7 \times 10^{-6}$</td>
</tr>
</tbody>
</table>
At this smaller timestep, the accuracy difference between the schemes begins to grow. As might be expected, the use of the force approximation scheme also begins to have a more significant impact when we want the very highest levels of accuracy - e.g. with the $P(EC)^2$ Hermite scheme in this example. However, we will rarely want a level of accuracy so high in practice.

Surprisingly, for our purposes Leapfrog appears to be superior to the Hermite scheme. It delivers comparable levels of accuracy and a significant speed advantage. In other tests conducted with a smaller simulation (e.g. A two body elliptical orbit, and figure-of-eight orbit), Hermite outperformed Leapfrog significantly in most cases. The anomalously good behaviour of Leapfrog in the above test can be accounted for by the fact that it tends to perform particularly well on circular or near-circular orbits. Since the orbits of most bodies in our simulation are very close to circular, we might expect Leapfrog to do better than in the general case. Also, the improved performance of Hermite was often not noticeable until we used small timesteps and looked for levels of accuracy approaching machine precision (e.g. $10^{-15}$). With this large simulation, the force evaluations are costly and we are restricted to larger timesteps, so the Hermite scheme does not really come into its own.

For the purposes of our simulation, Leapfrog/Tree with $\delta t = 0.02$ seems to provide a happy medium between high speed and high accuracy. With a calculation time of 6.3 secs per year, and a an Energy error per year of $3.4 \times 10^{-6}$ (not necessarily cumulative), this configuration meets both our performance goal and our accuracy goal (See B.1). This was chosen as the default setting for the backend.

**Test 2, $\delta t = 0.02$ Day, 100 years**

To demonstrate the behaviour over the longer term, the above simulation was extended to 100 years. Results are presented below. It can be seen that while Simple Euler errors are cumulative, for the Leapfrog and Hermite $P(EC)^2$ schemes, the magnitude of the final does not seem to grow with time. 

<table>
<thead>
<tr>
<th>Integrator</th>
<th>Force Scheme</th>
<th>Run Time</th>
<th>Energy Error</th>
<th>Angular Momentum Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hermite $P(EC)^2$</td>
<td>Tree</td>
<td>c. 30 mins</td>
<td>$8.1 \times 10^{-10}$</td>
<td>$2.7 \times 10^{-7}$</td>
</tr>
<tr>
<td>Hermite $PEC$</td>
<td>Tree</td>
<td>c. 20 mins</td>
<td>$5.6 \times 10^{-7}$</td>
<td>$9.9\times 10^{-7}$</td>
</tr>
<tr>
<td>Leapfrog</td>
<td>Tree</td>
<td>c. 10 mins</td>
<td>$1.26 \times 10^{-5}$</td>
<td>$3.2 \times 10^{-6}$</td>
</tr>
<tr>
<td>Simple Euler</td>
<td>Tree</td>
<td>c. 10 mins</td>
<td>$9.0 \times 10^{-3}$</td>
<td>$7.0 \times 10^{-4}$</td>
</tr>
</tbody>
</table>

However, the symmetric schemes have complex error behaviour, so examining the final energy error is not very informative. Much more can be seen from the graph of energy error against time. The energy error vs time, sampled every 2 days, is shown below.
Figure A.1: Energy errors in the 100-year test. The error of the Simple Euler scheme (green) is comparatively so large that both Hermite and Leapfrog errors appear flat.
Figure A.2: Energy errors in the 100-year test, showing Hermite PEC in blue, Hermite $P(EC)^2$ in black, and Leapfrog in red. The Hermite PEC integration seems to have suffered a close encounter. While Hermite $P(EC)^2$ shows the best performance, it took 3 times longer to compute than Leapfrog.

The second figure shows a zoomed view, demonstrating the periodic nature of the error.
Appendix B

Specification of the Back End

B.1 Feature List for the Back End

Features of the back end:

• Evolve body positions based on a realistic physical model. Accurately model the motion of planets, moons, asteroids, comets, space probes, etc.

• Provide initial conditions for the solar system at a given date from a standard database.

• Include the rotation and inclination of bodies in the simulation.

• Support modification of the gravitational system by the user, while the simulation is running. (e.g. Add a body, etc).

• Allow user to adjust the simulation speed (possibly at the cost of decreasing the accuracy.)

• Provide a method that allows the front end to visualize the orbit of a body.

• Run at an acceptable speed. A target of less than 10 seconds per year of simulated time was decided on.
  At the outset it was not clear what a reasonable expectation was in this regard. We decided that in order to be useful to a user, the simulation should run at least fast enough that a year of simulated time would take no more than 10 seconds on an average PC. This seemed a reasonable performance goal.

• Develop only small errors in body positions over timescales of several thousand years.
  For reasons discussed in detail in the report, a precise measure of accuracy is hard to define. However, in order to be of any value, obviously our simulation must be a close representation of the physics of the solar system.
• Calculate diagnostic data for use in testing (System energy and system angular momentum, etc.)

• Keep track of the magnitude of the errors in the simulation.

• Predict eclipses, conjunctions, transits and other events of astronomical interest.

• Allow the front end to visualize the gravitational field (“Rubber sheet” view).

• Handle collisions between objects.

B.2 Public Interface of the Back End

B.2.1 Overview of the Class Structure

Insert pretty diagram here. Learn UML?

B.2.2 The Vec3 Class

This is a basic utility class that handles 3D-vectors, and operations upon them. Vectors are represented internally as Cartesian x,y,z coordinates.

<table>
<thead>
<tr>
<th>Public Methods</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>double X()</td>
<td>Returns the x co-ordinate.</td>
</tr>
<tr>
<td>double Y()</td>
<td>Returns the y co-ordinate.</td>
</tr>
<tr>
<td>double Z()</td>
<td>Returns the z co-ordinate.</td>
</tr>
<tr>
<td>Vec3 operator =</td>
<td></td>
</tr>
<tr>
<td>Vec3 operator +=</td>
<td></td>
</tr>
<tr>
<td>Vec3 operator -=</td>
<td></td>
</tr>
<tr>
<td>Vec3 operator /=</td>
<td></td>
</tr>
<tr>
<td>double Mag()</td>
<td>Returns the magnitude: $\sqrt{x^2 + y^2 + z^2}$</td>
</tr>
<tr>
<td>double MagSq()</td>
<td>For efficiency, this routine that returns the magnitude squared is also included: $(x^2 + y^2 + z^2)$</td>
</tr>
<tr>
<td>double distTo(Vec3 &amp;other)</td>
<td>Returns the distance between this vector and the vector other.</td>
</tr>
<tr>
<td>Vec3 Unit()</td>
<td>Returns the unit vector in the same direction as this vector.</td>
</tr>
</tbody>
</table>

**Associated Global Methods**

| Vec3 operator =         |                                              |
| Vec3 operator -=        |                                              |
| Vec3 operator *=        |                                              |
| Vec3 operator /=        |                                              |
| Vec3 Cross(Vec3 v1, Vec3 v2) | Cross product                               |
B.2.3 The BasicBody Class

This is the base class for bodies. It contains only the essential information needed by the force determination algorithms, which operate over this class. Both Body and PseudoBody inherit from BasicBody.

<table>
<thead>
<tr>
<th>Public Data Members</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>double mass</td>
<td>For efficiency, we actually store (mass*G), where G is the Universal Gravitational Constant.</td>
</tr>
<tr>
<td>Vec3 s, v, a</td>
<td>s, v, a correspond to the body's position, velocity and acceleration.</td>
</tr>
</tbody>
</table>

Further Public Data Members

- Vec3 j: Jerk, the first derivative of acceleration. Required by the Hermite integrator.

B.2.4 The Body Class

The Body class extends BasicBody.

Body makes use of the utility class Colour defined by the front end.

<table>
<thead>
<tr>
<th>Public Data Members</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>string name</td>
<td>Assumed unique</td>
</tr>
<tr>
<td>double radius</td>
<td></td>
</tr>
<tr>
<td>Vec3 rotaxis</td>
<td>The axis of rotation of the body</td>
</tr>
<tr>
<td>double angle</td>
<td>The angle in radians by which the body has rotated, relative to its base position.</td>
</tr>
<tr>
<td>double w</td>
<td>The angular velocity of the body about its own axis.</td>
</tr>
<tr>
<td>Colour baseColour</td>
<td>The colour of the body.</td>
</tr>
</tbody>
</table>

- Orbit_Parameters op: Contains data that defines the approximate orbit the body is following, if any. Allows the front end to plot an orbit trail. See the definition of the Orbit_Parameters class.

- vector<Vec3> posHistory: A vector containing the last few positions the body has occupied. The number of entries and the interval between entries is set by the front end.

Further Public Data Members

- These are members not specified in the agreed interface, and are intended for internal backend use only.
bool interacts

Set true if the body influences other bodies gravitationally.

Vec3 oldv, olda, oldj

Predictor-corrector integrators need to know the values of s,v,a,j from the previous timestep.

Public Methods

void putHistory(Vec3 &v)

Add the body’s current position to the positionHistory list.

### B.2.5 The PseudoBody Class

The PseudoBody class is used by the force-tree algorithm. It inherits from BasicBody.

<table>
<thead>
<tr>
<th>Public Data Members</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>int index</td>
<td>A PseudoBody is either associated with a real Body, or represent a barycentre. In the former case index is the index of that body in the bodies vector. In the latter it is set to -1.</td>
</tr>
<tr>
<td>vector&lt;PseudoBody&gt; subtrees</td>
<td>PseudoBodies directly below this PseudoBody in the tree.</td>
</tr>
<tr>
<td>int interactLim</td>
<td>The index of the division between interacting and non-interacting bodies in subtrees.</td>
</tr>
</tbody>
</table>

### B.2.6 The Orbit_Parameters Class

Each body stores details of the conic section that most closely matches it’s orbit (if any). This allows the front end to plot orbit lines. This class handles the relevant data and methods.

**Conic section terms:**

**True Anomaly** The angular distance of the point in the orbit past the pericentre.

**SemiLatus Rectum** The chord through a focus parallel to the conic section directrix.

**SemiMajor Axis** Half the distance across an ellipse along the longest of its three principal axes. Loosely speaking, the 'average' radius of a non-circular orbit. Applies only to closed orbits.

**Pericentre** The point on an orbit closest to the central body.

**Apocentre** The point on an orbit farthest from a central body. For parabolic and hyperbolic orbits, this will be at ∞.
<table>
<thead>
<tr>
<th>Public Methods</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>double OrbitRadius(double theta)</td>
<td>Returns the radius of the orbit for the given value of theta, where ( \theta ) is the true anomaly.</td>
</tr>
<tr>
<td>double calcTrueAnomaly()</td>
<td>Returns the current value of the true anomaly.</td>
</tr>
<tr>
<td>Vec3 getR()</td>
<td>Returns the radial position vector from the central body to the orbiting body.</td>
</tr>
<tr>
<td>Vec3 getV()</td>
<td>Returns the velocity vector for the orbiting body relative to the central body.</td>
</tr>
<tr>
<td>Vec3 getCentrePos()</td>
<td>Returns the position vector of the central body.</td>
</tr>
<tr>
<td>double SemiMajorAxis()</td>
<td>Calculates the semimajor axis of the orbit. If ( e \geq 1 ) this quantity is undefined.</td>
</tr>
<tr>
<td>double rPericentre()</td>
<td>Calculates the radius of the orbit at the pericentre. If ( e &gt; 1 ) this quantity is undefined.</td>
</tr>
<tr>
<td>double rAporcentre()</td>
<td>Calculates the radius of the orbit at the apocentre. If ( e \geq 1 ) this quantity is undefined.</td>
</tr>
<tr>
<td>void PlotOrbit(double phi)</td>
<td>Drawing function used by the front end. Calculates points at phi degree intervals around the orbit, and hence draws the orbit line.</td>
</tr>
</tbody>
</table>

### Associated Methods in Universe Class

<table>
<thead>
<tr>
<th>Orbit_Parameters GetOrbitParameters(int i)</th>
<th>Given the index of a body, calculates the orbit parameters of that body. Automatically determines the central body ( c ).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orbit_Parameters GetOrbitParameters(int i, int c)</td>
<td>Calculates the orbit parameters of body ( i ), relative to the central body ( c ).</td>
</tr>
<tr>
<td>int GetGreatestAttractor(int i)</td>
<td>Finds the body that attracts body ( i ) most strongly. Used to determine the central body of the orbit.</td>
</tr>
</tbody>
</table>

### Public Data Members

<table>
<thead>
<tr>
<th>Universe *u</th>
<th>A pointer to the Universe which the body belongs to.</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool validOrbit</td>
<td>Some bodies cannot sensibly be said to orbit anything (e.g. the Sun within the solar system). In this case, the flag is set, and the front end draws no orbit line.</td>
</tr>
<tr>
<td>int thisBody</td>
<td>The index of the body which these Orbit_Parameters describe.</td>
</tr>
<tr>
<td>int centralBody</td>
<td>The index of the central body of the orbit.</td>
</tr>
<tr>
<td>Vec3 axUnit</td>
<td>The unit normal vector to the plane of the orbit, from the primary focus.</td>
</tr>
<tr>
<td>double energy</td>
<td>The specific energy of the orbiting body relative to the central body: ( 0.5v^2 + \frac{GM}{r} )</td>
</tr>
<tr>
<td>double h</td>
<td>The specific angular momentum of the orbiting body: (</td>
</tr>
<tr>
<td>double GM</td>
<td>The ( G \times ) Mass value of the central body.</td>
</tr>
</tbody>
</table>
double e
The eccentricity of the orbit, given by:
\[ \sqrt{1 + 2Eh^2} \]
double slr
The semilatus rectum of the orbit, given by: \[ \frac{h^2}{GM} \]

B.2.7 The TimeAndDate Class
This structure is used to return the time and date to the front end. Time is not represented internally in this format, instead serial day numbering is used.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>int year</td>
<td>A negative value corresponds to B.C.</td>
</tr>
<tr>
<td>int month</td>
<td></td>
</tr>
<tr>
<td>int month</td>
<td></td>
</tr>
<tr>
<td>int day</td>
<td></td>
</tr>
<tr>
<td>int hour</td>
<td></td>
</tr>
<tr>
<td>double second</td>
<td></td>
</tr>
</tbody>
</table>

B.2.8 The Universe Class
The Universe class is the central class of the backend. To begin a simulation, the front end creates a new object of type Universe. The methods supplied by the universe are then used to control the simulation. The public interface of universe is as follows:

<table>
<thead>
<tr>
<th>Method</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Accessors</strong></td>
<td></td>
</tr>
<tr>
<td>TimeAndDate GetTime()</td>
<td></td>
</tr>
<tr>
<td>double GetTimeStep()</td>
<td></td>
</tr>
<tr>
<td>Vec3 getObserverAdjustedPos(int &amp;index)</td>
<td>Returns the position of the i\textsuperscript{th} body, relative to the observer position.</td>
</tr>
<tr>
<td>Body getBody(string &amp;name)</td>
<td></td>
</tr>
<tr>
<td><strong>Assignment Operators</strong></td>
<td></td>
</tr>
<tr>
<td>void SetTimeStep(double &amp;tstep)</td>
<td>Sets the internal timestep ( \delta t )</td>
</tr>
<tr>
<td>void SetAccuracy(double &amp;aparam)</td>
<td>Sets the accuracy parameter for variable timestep scheme.</td>
</tr>
<tr>
<td>void AddBody(Body &amp;b)</td>
<td></td>
</tr>
<tr>
<td>void RemoveBody(string &amp;name)</td>
<td></td>
</tr>
<tr>
<td>void AlterBody(string &amp;name, Body &amp;b)</td>
<td>Positions are returned to the front end in the frame of reference of body name. Body name appears fixed at the origin, and other bodies move relative to this.</td>
</tr>
<tr>
<td>void SetObserverPosition(string &amp;name)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>General Methods</strong></td>
<td></td>
</tr>
</tbody>
</table>
### void Initialize()
Records the initial system energy and angular momentum, and performs some other setup tasks that cannot be done in the constructor.

### void Advance(double &interval)
Advances the simulation to t + interval. This may result internally in several smaller timesteps of length Δt being taken.

### void JumpToDate(TimeAndDate &t)
Reads from an ephemeris, if available.

#### Diagnostic Methods

- **double EnergyError()**
  Returns the fractional energy error since the simulation started.

- **double AngularMomentumError()**

#### Public Data Members

- **vector<Body> bodies**
  Rather than having to pass all the body positions back to the front end through an accessor function, this vector is left public. It should be treated as read-only by the front end.

---

Also, the private structure of the universe:

<table>
<thead>
<tr>
<th>Private Data Members</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>double time</td>
<td>Internal representation of time. Serial day number.</td>
</tr>
<tr>
<td>double tstep</td>
<td>Internal base timestep, in seconds.</td>
</tr>
<tr>
<td>double ap</td>
<td>Accuracy parameter for the variable timestep scheme.</td>
</tr>
<tr>
<td>int observerPos</td>
<td>The index of the body that corresponds to the observer position.</td>
</tr>
<tr>
<td>string observerLoc</td>
<td>The name of the above body.</td>
</tr>
<tr>
<td>int forceAlgorithm</td>
<td>Selects between direct summation and treecode.</td>
</tr>
<tr>
<td>int integrator</td>
<td>Selects between Simple Euler, Leapfrog and Hermite.</td>
</tr>
<tr>
<td>bool initialized</td>
<td>Set true when the universe has been initialized.</td>
</tr>
<tr>
<td>int histUpdateCount</td>
<td>Counter to determine when to add a position to body’s history list.</td>
</tr>
<tr>
<td>int orbUpdateCount</td>
<td>Counter to determine when to refresh the Orbit_Parameters of bodies.</td>
</tr>
<tr>
<td>double initAngularMomentum</td>
<td></td>
</tr>
<tr>
<td>double initEnergy</td>
<td></td>
</tr>
<tr>
<td>map&lt;string, int&gt; nameToIndex</td>
<td>Mapping from the names of bodies to their position in the bodies vector.</td>
</tr>
<tr>
<td>PseudoBody SSBC</td>
<td>Root of the PseudoBody tree</td>
</tr>
<tr>
<td>int rootnode</td>
<td>Root of the Greatest Influence tree</td>
</tr>
<tr>
<td><strong>Private Methods</strong></td>
<td>For a full listing of the private methods, see the header file.</td>
</tr>
</tbody>
</table>
Appendix C

Code

Due to space considerations, only a selection of project code has been included.

C.1 Force_Algorithm.cpp

template <class T> void SimpleForceAlg(vector<T> &b, const int &interactLim)
{
    int i,j;
    int size = b.size();
    Vec3 invsq;
    //Update acceleration fields of non-interacting bodies
    for(i=0; i<interactLim; ++i)
    {
        for(j=interactLim; j<size; ++j)
        {
            b[i].a = (b[j].mass)*InverseSquare(b[i].s,b[j].s);
            //Add the accel component due to the Jth interacting body.
            //There is no equal but opposite force, since body i does not interact
        }
    }
    //Now update acceleration fields of interacting bodies
    for(i=interactLim; i<size; ++i)
    {
        for(j=i+1; j<size; ++j)
        {
            //Note, potential roundoff error. Should get all accel
            //for a given body, sort, and add in order
            //smallest to largest, otherwise we lose precision.
            invsq = InverseSquare(b[i].s,b[j].s);
            b[i].a = invsq*(b[i].mass);
            b[j].a = invsq*(b[j].mass); //Body j experiences an equal but opposite force
        }
    }
    //Similar to SimpleForceAndJerk, but also calculates Jerk (da/dt).
    //Would be cleaner to get jerk in it's own function, calculates this.
    template <class T> void SimpleForceAndJerkAlg(vector<T> &b, const int &interactLim)
}
```cpp
{
    int i,j;
    int size = b.size();
    Vec3 accels, jerkcs;
double r2,r3;
    //Update a and j fields of non-interacting bodies
    for(i=0; i<interactLim; ++i)
    {
        for(j=interactLim; j<size; ++j)
        {
            Vec3 sab = b[j].s - b[i].s;
            Vec3 vab = b[j].v - b[i].v;
            r2 = 1/sab.MagSq();           //1/r^2
            r3 = r2/sab.Mag();            //1/r^3
            b[i].a ^= (b[j].mass)*(sab*r3);
            b[i].j ^= (b[j].mass*r3)*(vab - 3*r2*(Dot(sab,vab))+sab);
            //Same for jerk. Might be nicer to parcel off jerk to a function,
            //as in SimpleForceAlg.
            //but for efficiency, we'll do it all here.
        }
    }
    //Update acceleration fields of interacting bodies
    for(i=interactLim; i<size; ++i)
    {
        for(j=i+1; j<size; ++j)
        {
            Vec3 sab = b[j].s - b[i].s;
            Vec3 vab = b[j].v - b[i].v;
            r2 = 1/sab.MagSq();           //1/r^2
            r3 = r2/sab.Mag();            //1/r^3
            accels = sab*r3;
            jerkcs = r3*(vab - 3*r2*(Dot(sab,vab))+sab);
            b[i].a ^= accels*(b[j].mass);
            b[j].a ^= accels*(b[i].mass);
            b[i].j ^= jerkcs*(b[j].mass);
            b[j].j ^= jerkcs*(b[i].mass);
        }
    }
}

C.2 Force_Tree_Algorithm.cpp

//-------------------------------
//
//Contents: A custom tree-based force determination scheme.
//
//Author: Mark Cummins
//
//Created: 2004/03/31
//

#include "backend.h"
int Universe::GetGreatestAttractor(int i) const
{
    int c = -1;
    ```
double F=0.0;
double maxF=0.0;
int size = bodies.size();
for(int j=interactLimit; j<size; ++j)
{
    if(j==i)
    {
        if(bodies[j].mass>bodies[i].mass)
        {
            F = (bodies[j].mass)*InverseSquare(bodies[i].s.bodies[j].s).Mag();
            //Find the acceleration toward the jth interacting body
            if(F>maxF)
            {
                c = j;
                maxF = F;
            }
        }
    }
}
return c;
}
void Universe::BuildGreatestInfluenceTree()
{
    int size = bodies.size();
    //Reset rootnode
    rootnode = -1;
    //First, reset children fields to be empty
    for (int i=0; i<size; ++i)
    {
        bodies[i].children.clear();
    }
    //Now set them appropriately
    for(i=0; i<size; ++i)
    {
int c = GetGreatestAttractor(i);
    if(c==i)
    {
        if((rootnode < 0)||(bodies[rootnode].mass < bodies[i].mass))
        {
            //Set body as new rootnode
            rootnode = i;
        }
    }
else
    {
        //Not a binary
        bodies[c].children.push_back(i);
    }
    }
}
void Universe::BuildPseudoBodyTree()
{
    BuildGreatestInfluenceTree();
    SSBEC = makePseudoBody(bodies[rootnode]);
}
PseudoBody Universe::makePseudoBody(const Body &b)
PseudoBody pb;
int size = b.children.size();
if(size==0)
{
  //Leaf node
  //Constructor copies across all the info except the index.
  pb = PseudoBody(b);
  //Look up the index of this body and record it
  pb.index = getBodyIndex(b.name);
}
else
{
  //Non-leaf node
  //Make this body a pb
  //Don't need to worry about interactLim,
  //since non-interacting bodies can't have children
  PseudoBody thisb = PseudoBody(b);
  thisb.index = getBodyIndex(b.name);
  pb.subtrees.push_back(thisb);

  //Then recurse to all it's children
  for(int i=0; i<size; ++i)
  {
    PseudoBody childb = makePseudoBody(bodies[b.children[i]]);
    if((childb.index==i)||(bodies[childb.index].interacts))
    {
      pb.subtrees.push_back(childb);
    }
    else
    {
      vector<PseudoBody>::iterator pos;
      pos = pb.subtrees.begin();
      advance(pos, pb.interactLim);
      pb.subtrees.insert(pos,childb); //Insert the new body.
      pb.interactLim +=1; //Increment our index
    }
  }

  //Now update the details of the barycentre
  size=1;
  //Size + 1 since we added the body itself, plus children
  for(i=0; i<size; ++i)
  {
    pb.mass += pb.subtrees[i].mass;
    pb.s += pb.subtrees[i].s+pb.subtrees[i].mass;
    pb.v += pb.subtrees[i].v+pb.subtrees[i].mass;
  }
  pb.s /= pb.mass; //Average position
  pb.v /= pb.mass; //Average velocity
}
return(pb);
}

void Universe::RefreshPseudoBodyTree(PseudoBody &pb)
{
  //Leaves the tree structure the same, but updates the s and v fields of the PB's
  pb.a = Vec3(0,0,0);
pb.j = Vec3(0,0,0);
if(pb.index>=0)
{
    
    //Leaf node
    pb.s = bodies[pb.index].s;
    pb.v = bodies[pb.index].v;
    // cout << "Updated " << bodies[pb.index].name << endl;
}
else
{
    
    //Barycentre node
    //Set s,v to zero
    pb.s = Vec3(0,0,0);
    pb.v = Vec3(0,0,0);
    //Update children recursively
    int size = pb.subtrees.size();
    for(int i=0; i<size; ++i)
    {
        RefreshPseudoBodyTree(pb.subtrees[i]);
        //Recalculate this barycentre's position as the average of its children
        pb.s += pb.subtrees[i].s*pb.subtrees[i].mass;
        pb.v += pb.subtrees[i].v*pb.subtrees[i].mass;
    }
    pb.s /= pb.mass; //Average position
    pb.v /= pb.mass; //Average velocity
}
}
void Universe::TreeCodeAlg(PseudoBody &pb, const bool needJerk)
{
    
    //This function updates the acceleration fields of all the
    //bodies using the force tree algorithm
    //It assumes a valid tree already exists
    //First, update all the accelerations at this tree level(considering jerks if needed)
    if(needJerk)
    {
        SimpleForceAndJerkAlg(pb.subtrees, pb.interactLim);
    }
    else
    {
        SimpleForceAlg(pb.subtrees, pb.interactLim);
    }
    //Now, examine the level
    int size=pb.subtrees.size();
    for(int i=0; i<size; ++i)
    {
        if(pb.subtrees[i].index>=0)
        {
            
            //This PB refers to a real body.
            //Update the real body's acceleration field
            bodies[pb.subtrees[i].index].a = pb.subtrees[i].a;
            if(needJerk)
            {
                bodies[pb.subtrees[i].index].j = pb.subtrees[i].j;
            }
        }
        else
        {
            
        }
        
    }
}
// This PB is a barycentre
// Trickle down its acceleration to all its children
// Then recurse
PseudoBody bary = pb.subtrees[i];
int num = bary.subtrees.size();
for(int j=0;j<num;++j)
{
    bary.subtrees[j].a += bary.a;
    if(needJerk)
    {
        bary.subtrees[j].j += bary.j;
    }
}
TreeCodeAlg(bary, needJerk);

C.3 Integrator.cpp

//...................................................................................//
//
// Contents: Definitions of the numerical integration routines which
// update the positions and velocities of all the Bodies in bodies
// based on the acceleration field and the timestep.
// Author: Mark Cummins
// Created: 2004/01/06
//...................................................................................#
#include "backend.h"
#include <cmath>
void Universe::InitializeIntegrator(const int &integrator)
{
    switch(integrator)
    {
    case SIMPLE:
        break;
    case LEAPFROG:
        GetForces(); //DEFENSIVE - Make sure the acceleration fields are current.
        //May do a little needless work.
        break;
    case HERMITE:
        GetForcesAndJerk();
        //Because the call to update acceleration is in the middle of the Hermit loop,
        //rather than at the start, unless we do this.
        //the first time we call Hermite we may have no valid accel data.
        break;
    default:
        cerr << "ERROR in InitializeIntegrator(): Integrator " << integrator
             << " does not exist." << endl;
        break;
    }
    // This simple integrator is basic school level applied maths.
    // It updates the velocity and positions of bodies according to.
    //
// vnnew = v + at
// s = 0.5(t(vnnew + v)) (Equivalent to ut + 0.5a(t^2))
// v = vnnew;
// This is a simple Euler method. It's not very accurate unless tstep is very small.
void Universe::SimpleIntegratorStep()
{
    int i;
    int size = bodies.size();
    Vec3 vnnew;
    for(i=0; i<size; ++i)
    {
        vnnew = bodies[i].v + tstep*bodies[i].a;
        bodies[i].s += tstep*0.5*(vnnew + bodies[i].v);
        bodies[i].v = vnnew;
    }
}
// Non-Interleaved Version
// The non-interleaved representation of leapfrog, where v and s are all defined at the same timestep.
// s[1] = s[0] + v[0]*t + 0.5*a[0]*t;
// v[1] = v[0] + 0.5*t*(a[0] + a[1])
// Though this is slightly more computationally expensive, it's much more convenient.
void Universe::LeapFrogStepPartA()
{
    int i;
    int size = bodies.size();
    for(i=0; i<size; ++i)
    {
        bodies[i].v += 0.5*tstep*bodies[i].a; //At this point we have vtemp = v[0] + 0.5*t*a[0]
        bodies[i].s += tstep*bodies[i].v; // This gives us s[1] = s[0] + t*(v[0] + 0.5*t*a[0])
    }
}
// Acceleration field is updated between PartA and PartB.
void Universe::LeapFrogStepPartB()
{
    int i;
    int size = bodies.size();
    for(i=0; i<size; ++i)
    {
        bodies[i].v += 0.5*tstep*bodies[i].a;
        //Now v[1] = vtemp + 0.5*t*a[1] , i.e. v[1] = v[0] + 0.5*t*(a[0] + a[1])
    }
}
**********Hermite Integrator**********
//This is a 4th order time-symmetric Predictor-Corrector integrator
//For a full explanation see the report.
//Phases:
//0) Store values of a,v,s,j

60
/1) Predictor (simple Taylor series)
//2) Calc Acceleration and Jerk at predicted s[i]
//3) Correct positions and velocities
//4) Optional: New correct Acceleration and Jerk.
//(Expensive to do this twice. Not necessarily worth it.)

void Universe::HermiteStoreOldValues()
{
    int i;
    int size = bodies.size();
    // Store a,v,s,j
    for(i=0; i<size; ++i)
    {
        bodies[i].ssold = bodies[i].s;
        bodies[i].vsold = bodies[i].v;
        bodies[i].asold = bodies[i].a;
        bodies[i].jsold = bodies[i].j;
    }
}

void Universe::HermitePredictStep()
{
    int i;
    int size = bodies.size();
    for(i=0; i<size; ++i)
    {
        // Predict new positions
        bodies[i].s = tstep*(bodies[i].v + tstep*0.5*(bodies[i].a + (tstep/3)*bodies[i].j));
        bodies[i].v = tstep*(bodies[i].a + tstep*0.5*bodies[i].j);
    }
}

void Universe::HermiteCorrectStep()
{
    int i;
    int size = bodies.size();
    for(i=0; i<size; ++i)
    {
        bodies[i].v = bodies[i].vsold +
            tstep*0.5*(bodies[i].a + bodies[i].asold + (tstep/6)*(bodies[i].jsold - bodies[i].j));
        bodies[i].s = bodies[i].ssold +
            tstep*0.5*(bodies[i].vsold + bodies[i].v + (tstep/6)*(bodies[i].asold - bodies[i].a));
    }
}