



EXTENSIBLE SIMULATION OF PLANETS AND COMETS

---

A Thesis  
Presented to the  
Faculty of  
California State University,  
San Bernardino

---

In Partial Fulfillment  
of the Requirements for the Degree  
Master of Science  
in  
Computer Science

---

by  
Natalie Anne Wiser-Orozco  
December 2008

EXTENSIBLE SIMULATION OF PLANETS AND COMETS

---

A Thesis  
Presented to the  
Faculty of  
California State University,  
San Bernardino

---

by  
Natalie Anne Wiser-Orozco  
December 2008

Approved by:

\_\_\_\_\_  
Dr. Keith Evan Schubert, Chair,  
Department of Computer Science and  
Engineering

\_\_\_\_\_  
Date

\_\_\_\_\_  
Dr. Ernesto Gomez

\_\_\_\_\_  
Dr. Richard Botting

© 2008 Natalie Anne Wiser-Orozco

## ABSTRACT

The research conducted is intended to enhance the way we view and study our solar system and others, by allowing scientists of astronomy, physics, and computer science to accurately simulate many different celestial systems. By creating this extensible simulator, we can organize the celestial bodies to be studied into groups called projects, calculate their positions, graphically visualize their movement using the computed positions, and finally, be able extend this tool to accommodate additional numerical methods, body shapes and behaviors, and camera views. The underlying science behind the Extensible Simulator is the n-body problem, which is derived from the laws of Kepler and Newton. The n-body equation along side Runge-Kutta's Fourth-Order ordinary differential equation solver as the numeric method, are the methods we use for the calculation of the body positions. A thorough explanation of the Extensible Simulator will be given, paying close attention to its extensible components, which will allow others in this field of study to contribute a multitude of features in the form of plug-ins as their needs arise. These extensible components are what separates the Extensible Simulator from its predecessors, and will make a significant contribution to the fields of astronomy, physics, and computer science.

## ACKNOWLEDGEMENTS

To Dr. Schubert, I owe my deepest gratitude to you for having faith in me, and pushing me beyond what I thought was attainable. It was an honor to have your instruction and guidance throughout my under-graduate and graduate studies. To Dr. Gomez and Dr. Botting, the passion you exhibit for life and science is inspiring. Thank you so much for your input and insight on this project. I would like to also thank the entire Computer Science Department for this absolutely wonderful, encouraging and empowering experience. Thank you for everything.

To my husband Angel, words can't describe how thankful I am for such a wonderful partner. You stood beside me when times were rough, celebrated with me when times were great, and for the duration, you always kept me smiling. Thank you so very much and for being there with me every step of the way. To my parents, thank you for all of the conversations we had that were full of inquiry and support. Since my life began, you instilled the confidence in me that I would need, making this a reality. To my brother, Erick, without your "encouragement" to get this thesis finished, I'd probably still be working on it, and yes, I can move back home now! To my best friend Becca, the long nights on the couch, the constant emails back and forth, and the reassuring snail mails kept my spirits up, and always gave me that extra push when I needed it the most. Last, but certainly not least, Korin, thank you for always being there, and lending an ear. I am forever grateful and honored to have such a wonderful support system.

## DEDICATION

To my husband Angel.

## TABLE OF CONTENTS

<i>Abstract</i> . . . . .	iii
<i>Acknowledgements</i> . . . . .	iv
<i>List of Tables</i> . . . . .	viii
<i>List of Figures</i> . . . . .	ix
<b>1. Introduction</b> . . . . .	1
1.1 Background . . . . .	1
1.2 History . . . . .	6
1.3 Significance . . . . .	8
1.4 What Was Accomplished . . . . .	8
1.5 Outline . . . . .	9
<b>2. Science Behind The Simulation</b> . . . . .	10
2.1 Orbits According to Kepler . . . . .	10
2.2 Orbits According to Newton . . . . .	13
<b>3. Computer Science Behind The Simulation</b> . . . . .	16
3.1 Numerical Analysis . . . . .	16
3.1.1 Runge-Kutta Fourth-Order Method . . . . .	17
<b>4. Extensible Simulator</b> . . . . .	21



4.1	Introduction . . . . .	21
4.2	Basic Structure of the Graphical User Interface . . . . .	22
4.3	Python Graphical User Interface - Project Organizer . . . . .	23
4.3.1	Creating A Project . . . . .	24
4.3.2	Creating A Celestial Body . . . . .	25
4.3.3	File Editing . . . . .	28
4.3.4	File Saving . . . . .	29
4.3.5	Project Copying . . . . .	29
4.3.6	History Between Two Projects . . . . .	30
4.4	Python Graphical User Interface Simulation . . . . .	30
4.5	Application Programming Interface . . . . .	35
4.5.1	Bodies . . . . .	37
4.5.2	Cameras . . . . .	38
4.5.3	Gravitational Functions and Numeric Methods . . . . .	39
5.	<i>Benchmarks and Error Analysis</i> . . . . .	46
5.1	Benchmarks . . . . .	46
5.2	Error Analysis . . . . .	47
6.	<i>Conclusion</i> . . . . .	57
	<i>APPENDIX A: MY CODE</i> . . . . .	60
	<i>APPENDIX B: INSTALLATION</i> . . . . .	98
	<i>References</i> . . . . .	102

## LIST OF TABLES

4.1	Keystrokes for the Simulator. . . . .	36
5.1	Pre-application programming interface with large step size. . . . .	46
5.2	Post-application programming interface with small step size. . . . .	47
5.3	Average error of ‘FiveBodyProject’ at 40 years/old step size. . . . .	48
5.4	Average error of ‘FiveBodyProject’ at 40 years/new step size. . . . .	49
5.5	Average error of ‘FiveBodyProject’ at 100 years/new step size. . . . .	50
5.6	Average error of ‘SevenBodyProject’ at 100 years/new step size. . . . .	51

## LIST OF FIGURES

1.1	Orbit with eccentricity of .01. . . . .	2
1.2	Orbit with eccentricity of .99. . . . .	3
1.3	Aftermath of Shoemaker-Levy 9 colliding with Jupiter. . . . .	4
2.1	An orbit as an ellipse with the Sun at one foci. . . . .	11
2.2	Kepler's second law. . . . .	11
2.3	Ellipse in standard position. . . . .	12
2.4	Geometry of a small portion of a circular orbit. . . . .	14
4.1	Screenshot of the Extensible Simulator. . . . .	23
4.2	Screenshot of a project configuration file. . . . .	25
4.3	Screenshot of a body configuration file. . . . .	27
4.4	Unified modeling language diagram of simulation components. . . . .	31
5.1	Relative error of 'FiveBodyProject' at 40 years/old step-size. . . . .	52
5.2	Relative error of 'FiveBodyProject' at 40 years/new step-size. . . . .	53
5.3	Relative error of 'FiveBodyProject' at 100 years. . . . .	54
5.4	Relative error of 'SevenBodyProject' at 100 years. . . . .	55
5.5	Relative error of 'SevenBodyProject' at 100 years continued. . . . .	56

## 1. INTRODUCTION

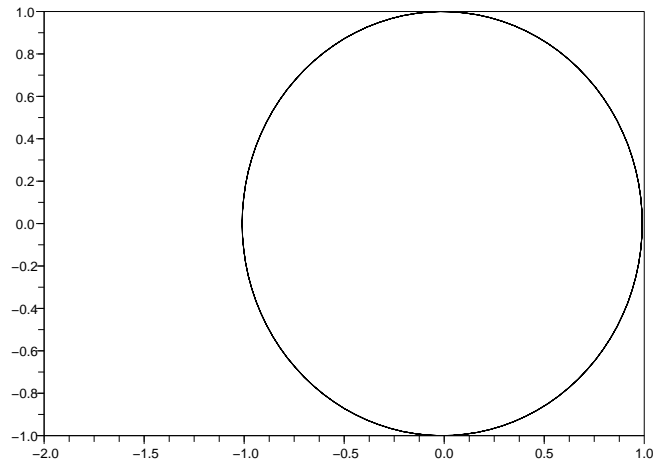
### 1.1 *Background*

There is no question that the bodies in space we call planets and comets are mystical and intriguing, as are the tools that help us visualize them. Understanding the movement of celestial bodies is paramount to continuing the discussion of this tool, as is the numerical method that is used to calculate this movement. I will aim to give the reader a background that will serve as a solid foundation of these two concepts.

In order to understand the movement of celestial bodies, I will discuss their two main characteristics, mass and orbit. The mass of celestial bodies can vary greatly. To give an idea of how much they vary, let's first define what is most familiar to us, our Earth. The mass of the Earth is on the order of  $10^{24}$  kilograms(kg). Our Sun is much larger than that, at  $10^{30}$ kg. The very large star Antares is 15.5 solar mass units, which means that it is 15.5 times the mass of our Sun, while smaller asteroids can be but 1 kg. We care about mass because it plays a large role in how these bodies interact with one another through the laws of gravity. If a body with a large mass comes in close proximity of another body with a lesser mass, then the orbit of the body with lesser mass will be affected, and its orbit is affected.

This brings us to the second property that celestial bodies possess, which is their orbits. Orbits are in the form of an ellipse, which means that they take on an oval

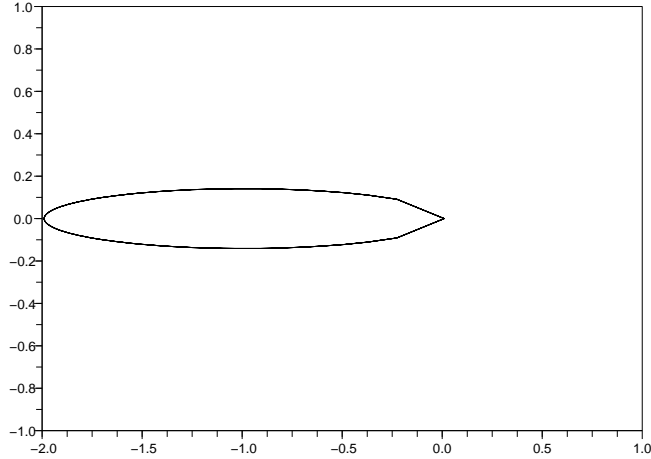
shape. Ellipses can be nearly circular or very flat, which is determined by their eccentricity.



*Fig. 1.1:* Orbit with eccentricity of .01.

This value can range between zero to one exclusively, meaning they can be anything in-between zero and one, but exclude the actual values zero or one. If the value is 1 or greater, then the shape would no longer be an ellipse, but a parabola. If the value is zero, it would be a perfect circle, which follows that the nearer the value is to zero, the more the shape looks like a circle. The nearer it is to one, it becomes more flattened. I've included two figures to illustrate the two extremes of a low value (Fig. 1.1) and a high value (Fig. 1.2) for the eccentricity. Since the planets in our solar system have a low eccentricity, they stay fairly equidistant from the Sun along their path. The paths of bodies that have a higher eccentricity are capable of traveling very far, as in the case of Halley's comet, which travels 35.33 AU from the Sun. This equates to just over 5 billion kilometers or just over 3 billion miles.

The orbits with a higher eccentricity enable the body to reach very long distances within the solar system.



*Fig. 1.2:* Orbit with eccentricity of .99.

The planets and Sun are very large masses, and due to the gravitational field of these great masses, the orbits of smaller bodies can be altered. As an example of a large mass altering an orbit, we look at comet Shoemaker-Levy 9. It was discovered by Carolyn and Eugene Shoemaker and David Levy on March 24th, 1993, but was deduced that this comet had been captured by Jupiter's orbit between the range of 1920 - 1938. What they discovered were fragments of the comet, and was deduced that under the influence of tidal forces, it had split up on July 8, 1992. Due to observations, it was calculated that it would reach its closest distance from Jupiter in the month of July, 1994, and sure enough, from July 16-22, the fragments of Shoemaker-Levy 9 had passed too close, and due to Jupiter's large mass, the fragments became very attracted to it due to Newton's laws of motion, and crashed right into the planet[16][2].

This particular event can be attributed to Newton's three laws of motion that are found in *Physics For Scientists and Engineers with Modern Physics* [10].

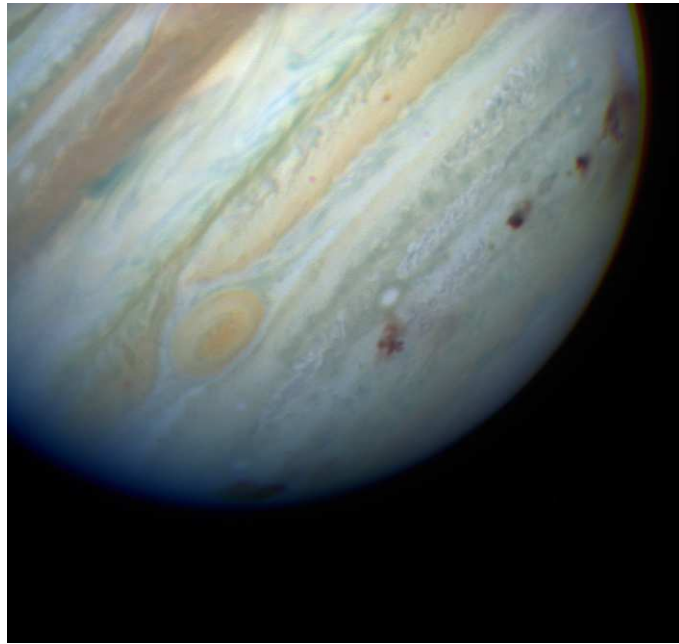


Fig. 1.3: Aftermath of Shoemaker-Levy 9 colliding with Jupiter.

1. An object at rest will remain at rest, and an object in motion, will continue in that same motion with a constant velocity.
2. The acceleration of an object is directly proportional to the net force acting on it and inversely proportional to its mass ( $\sum F = m\mathbf{a}$ ).
3. If two forces interact, the force  $F_{12}$  exerted by object 1 on object 2 is equal in magnitude to and opposite in direction to the force  $F_{21}$  exerted by object 2 on object 1:  $F_{12} = -F_{21}$ [10],

where the gravitational forces of Shoemaker-Levy 9 and Jupiter acted upon each other, drawing each other so near to one another that they collided.

This is precisely one of the reasons that we care about orbits. We want to be able to determine the trajectories of planets or comets in our solar system to avoid disasters such as this, as well as determine the orbits of the bodies of other solar systems to see if they are stable and inhabitable. Our aim here is to create a simulator that can model the behaviors of arbitrary celestial bodies within a solar system, initially paying attention to how their orbits can be affected by one another, and to create such a platform that would enable their calculation by different numerical methods, that would also be able to be compared with one another.

Numerical methods are methods used to solve ordinary differential equations(ODE). An ODE is an equation that describes continuous change over time. While some simple ODE's can be solved analytically, meaning that they have a closed-form solution and can be calculated exactly, most require very tedious calculations, which is why we resort to a numerical solution. To solve using numerics means to start with some initial condition, and subsequently solve the equation at small steps across the interval that is desired. These approximations that we make at these small steps, will no doubt cause some error, because we are not dealing with the exact closed-form solution.

There are many ways of going about the approximations of the ODE, and for this particular project, the Runge-Kutta Fourth-Order(RK4) ODE solver will be used. This particular method is discussed in depth in Chapter 3.



## 1.2 History

Johannes Kepler (1571-1630) started modern astronomy by using an observationally derived mathematical model to describe the laws of planetary motion. These laws are the foundation of the many tools used to understand our solar system, and ultimately, the universe. His laws are as follows:

Kepler's first law states that the orbit of a celestial body is an ellipse, where one foci of the ellipse is the Sun. The second law was introduced during his study of the orbit of Mars, where the radius vector sweeps out equal areas in equal times. Finally, in 1618, Kepler's third planetary law was introduced, which states that the square of the period of any planet about the Sun is proportional to the cube of its mean distance from the Sun[15][9].

In the context of this paper, these foundations have been used to simulate the orbits of planets and comets by a great number of people, including Sir Isaac Newton (1643-1727). He showed that celestial bodies and objects on Earth are governed by the same set of laws, by showing the correlation between Kepler's laws and his theory of gravitation. Newton's law of universal gravitation states that "Every particle in the universe attracts every other particle with a force which is proportional to the product of their masses and inversely proportional to the square of their distance apart" [15].

Joseph-Louis Lagrange(1736-1813) was named the greatest mathematician of the 18th century because he made great contributions to the fields of analysis, number theory, classical mechanics, and celestial mechanics[3]. He introduced the theory of differential equations, and transformed Newtonian Mechanics into a branch of analysis

called Lagrangian Mechanics. He also studied the 3-body problem, which is directly related to our subject matter, where he found special cases to this problem, and named them Lagrangian Points[4]. These points are 5 points in an orbital configuration where a small object only affected by gravity will remain stationary, relative to the two larger objects.

Jules Henri Poincaré (1854-1912) made a great contribution to the field of Celestial Mechanics when he submitted a solution to the n-body problem to the King of Sweden, who offered a prize to the first to solve it. Poincaré did not solve the problem as specified, but was awarded the prize because it was of great significance to celestial mechanics, and contained many important ideas that eventually led to chaos theory. This submission contained a serious error, however, that was later corrected by Karl F. Sundman[5]. In the 1990's, Quidong Wang generalized the 3-body problem, solving for n bodies[17].

Today, there has been a great deal of research that aims to simulate the motion of planets and comets [14] [11] [6] [1], but the most significant contribution to this area that I found is the Millennium Run project, where the aim was to investigate how the universe has evolved over time. This particular project traced 10 billion particles over a simulated volume space of 2 billion light years, and in that, contained 20 million galaxies[18]. The aim of these projects was not to offer an interactive simulator where any number of arbitrary celestial bodies can be defined, and where there is potential to extend the simulator to incorporate different numerical methods for computing the orbits of a multi-body system.

### *1.3 Significance*

The significance of this project is its ability to be a useful tool to computer scientists, astronomers, and physicists by determining how an arbitrary set of celestial bodies interact with one another. It allows definition of bodies that belong to a system, and creates a graphical simulation of them based on their specific properties. Ideas are then easily conveyed not only to colleagues, but to the public as well through the simulation. The most interesting feature of this tool however, is its extensibility over time. Those scientists who wish to extend this tool can do so by incorporating different numerical methods, gravitational functions, or more detailed algorithms describing behavior of a certain body or force. The contributions of myself and others will help this evolve into a very robust simulator, that will be capable of endless detailed simulations as opposed to being limited to simulating one particular scenario.

### *1.4 What Was Accomplished*

A graphical tool, simulator, and Application Programming Interface(API) comprise the basic components of this Extensible Simulator. The creation of “projects” within the graphical tool allow the user to manage a specific set of bodies. Bodies are added to the project, which encompasses the initialization of all the data needed by the program, and are then read by a Scilab script that calculates the positions of the bodies based on the initial conditions for each body using the gravitational function defined in a separate script. Using the initial conditions, these scripts utilize a built-in RK4 ODE solver that ships with Scilab. This solver returns back the position of each body for each time step, which is then read by the simulation script. Depending on

the definitions of the bodies and calculated positions, the simulation script renders the objects and their movement. An analysis of a project that contains the Sun, Mercury, Venus, Earth, and Earth's Moon was performed by comparing the measured values to the true values for perihelion and aphelion, as well as the angle at which they occur. These errors were compared to a project that contains Mars and Jupiter as additional planets to see how they affect the orbits of the others.

### *1.5 Outline*

As you may have already guessed, the introductory chapter introduces the background, history, significance, and what was accomplished in this paper. Chapter 2, "Science Behind the Simulation" covers orbits according to both Kepler and Newton. In "Computer Science Behind the Simulation," Chapter 3, we will dive into the numerical analysis that we employed to calculate the orbits of the bodies. All of the functions that make up the Extensible Simulator will be discussed in Chapter 4, including the Graphical User Interface(GUI), and the API that will be the component that makes the Extensible Simulator extensible. All testing and benchmark data will fill Chapter 5, and finally the conclusion will summarize our findings and include directions of further study in Chapter 6.

## 2. SCIENCE BEHIND THE SIMULATION

### 2.1 *Orbits According to Kepler*

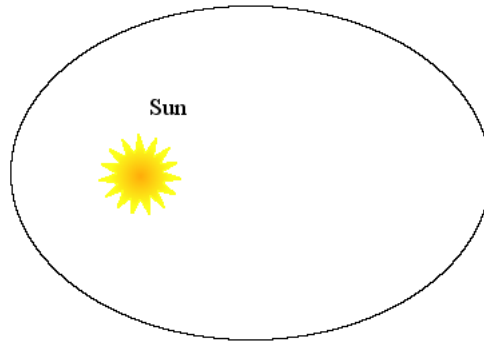
The orbit of a body in space is governed by the laws of planetary motion as described by Kepler and the laws of gravitational motion as described by Newton.

Kepler's laws are as follows:

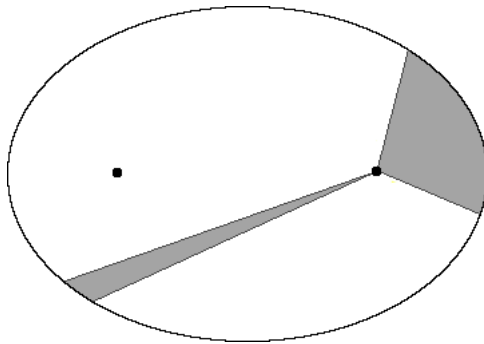
1. The orbits of planets and comets are described as an ellipse, with the Sun at one of the foci of the ellipse(See Fig. 2.1).
2. A line that joins the planet and the Sun will fill equal areas on a regular time interval(Fig. 2.2). This means that the closer the planet is to the Sun, or perihelion, the faster it will travel.
3. The squares of the orbital periods of a body are directly proportional to the cubes of the semi-major axes(the distance from the center to the farthest point along the length of the ellipse), which means that the larger the semi-major axes, the longer the orbital period will be.

$$T^2 = \frac{4\pi^2}{GM}r^3$$

Where  $T$  is the orbital period of the planet,  $G$  is the gravitational constant,  $M$  is the mass of the Sun, and  $r$  is the radius of the planet's circular orbit[10].



*Fig. 2.1:* An orbit as an ellipse with the Sun at one foci.



*Fig. 2.2:* Kepler's second law.

To explore the shape of an ellipse, we can define it as the set of all points in a plane where the sum of the distances of two fixed points remain constant. These two fixed points are the foci of the ellipse, and in the case of our planets, one of those foci would be the Sun. The eccentricity is what determines a circular, or a very flat ellipse, which describes the different trajectories of planets and comets respectively.

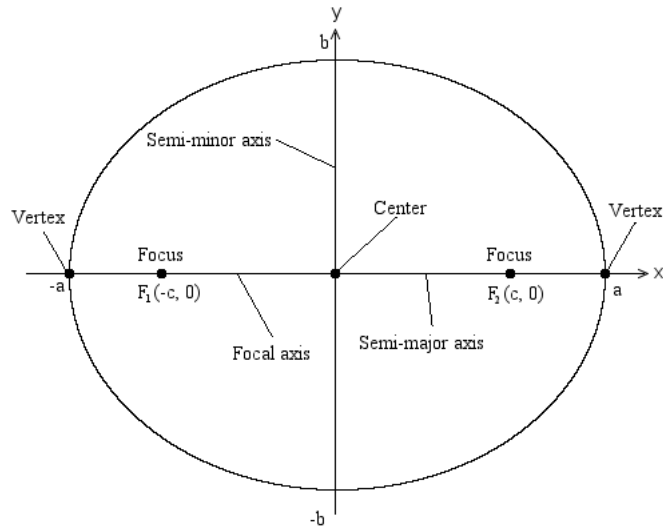


Fig. 2.3: Ellipse in standard position.

Here is the mathematical equation for an ellipse in the cartesian coordinate system,

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1, a > b > 0,$$

where  $a$  is the semi-major axis and  $b$  is the semi-minor axis. The center to focus distance,  $c$  is defined by  $c = \sqrt{a^2 - b^2}$ . The eccentricity is therefore defined as  $e = c/a, 0 < e < 1$ . See Figure 2.3[12].

## 2.2 Orbits According to Newton

Newton was able to solidify the relationship between falling objects near to the Earth and the motion of planetary orbits by using the inverse-square relationship that Kepler had suggested in his third law. After the apple fell off the tree in his garden, he pondered that the Earth's gravity that caused the apple to fall towards the Earth could be extended to bodies like the Moon. Even though the Moon happened to be a greater distance from the Earth than any Earth-bound object, he believed that they may behave the same way. If at any point along a bodies orbit, in Newton's particular case, the Moon's relationship to the Earth, the body is freed from all of its forces, it would continue in a straight line tangent to the orbit at that point. Since the body does not follow that tangent line, and the distance between the two bodies doesn't change, then the body in orbit must be effectively "falling" towards the body that is at the foci of the orbit's ellipse. Figure 2.4 illustrates this by showing a deviation of  $y$  from the line  $AB(=x)$  that would be followed ( $AP$ ) if gravity was not present[7].

From this assertion, Newton derived his law of universal gravitation,

$$F_g = G \frac{m_1 m_2}{r^2}$$

where  $F_g$  is the gravitational force,  $G$  is the gravitational constant,  $m_1$  and  $m_2$  are the masses of the respective planets, and finally,  $r$  is the distance between the two masses. To extend this to describe the interactions between any number of bodies using Newton's laws of universal gravitation, the equation of motion for  $n$  bodies



becomes

$$\begin{aligned}
 F_g &= -G \sum_{j=1, j \neq i}^{j=N} \frac{m_i m_j (\mathbf{r}_i - \mathbf{r}_j)}{|\mathbf{r}_i - \mathbf{r}_j|^3} \\
 F_g &= m_i \ddot{\mathbf{r}}_i \\
 m_i \ddot{\mathbf{r}}_i &= -G \sum_{j=1, j \neq i}^{j=N} \frac{m_i m_j (\mathbf{r}_i - \mathbf{r}_j)}{|\mathbf{r}_i - \mathbf{r}_j|^3} \\
 \ddot{\mathbf{r}}_i &= -G \sum_{j=1, j \neq i}^{j=N} \frac{m_j (\mathbf{r}_i - \mathbf{r}_j)}{|\mathbf{r}_i - \mathbf{r}_j|^3}
 \end{aligned} \tag{2.1}$$

where  $\mathbf{r}_i$  is the position vector of the  $i$ th particle relative to some inertial frame,  $m_i$  is the mass of the  $i$ th particle,  $G$  is the gravitational constant, and  $m_j$  is the mass of the  $j$ th particle.

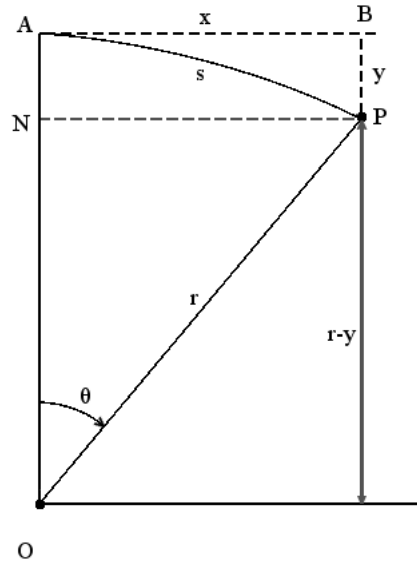


Fig. 2.4: Geometry of a small portion of a circular orbit.

In the cases of one and two bodies, this problem can be solved analytically, however once we move to three or more bodies, we must resort to numerical analysis. Because

every body has an effect on one another, the system can become very chaotic past two bodies when trying to solve analytically, meaning that the math becomes far too complicated. Solving by numerical analysis, we can eliminate the tediousness of the analytical solution, and can obtain an approximate, yet still very accurate result.

### 3. COMPUTER SCIENCE BEHIND THE SIMULATION

#### 3.1 Numerical Analysis

Newton's laws of universal gravitation as described in Section 2.2 help us describe the trajectory of a smaller body about a larger body. Using equation 2.1 from that section, we find the following system of differential equations[8]:

$$\begin{aligned}\ddot{x}_i &= -G \sum_{j=1, j \neq i}^{j=N} \frac{m_j(x_i - x_j)}{|r_{i,j}|^3} \\ \ddot{y}_i &= -G \sum_{j=1, j \neq i}^{j=N} \frac{m_j(y_i - y_j)}{|r_{i,j}|^3},\end{aligned}$$

where  $r_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ .

This system of differential equations describes the continuous change of acceleration of each of the planets involved. We need to solve this using numerical analysis because trying to find a closed-form, or analytical solution would be nearly impossible due to the limitless amount of forces acting on each one of the bodies. While a closed-form solution would give us an exact answer as to the specific positions and velocities of the planets, a numeric solution gives us a reasonable approximation without the tediousness of a closed-form solution. This invariably introduces error into the calculation, which we will discuss in Chapter 5. The order of an ODE is determined by its highest derivative, and the equations that we happen to be dealing with are a system of second-order ODE's. In order to solve this using numerical analysis, we

must transform these equations into an equivalent first order system:

$$\begin{aligned}\frac{d}{dt}x_{pos} &= x_{vel} \\ \frac{d}{dt}x_{vel} &= -G \sum_{j=1, j \neq i}^{j=N} \frac{m_j(x_{posi} - x_{posj})}{|r_{i,j}|^3} \\ \frac{d}{dt}y_{pos} &= y_{vel} \\ \frac{d}{dt}y_{vel} &= -G \sum_{j=1, j \neq i}^{j=N} \frac{m_j(y_{posi} - y_{posj})}{|r_{i,j}|^3}\end{aligned}$$

The decision to use the RK4 solver over other solvers such as Adams-Bashforth, Euler, Fehlberg, or Adams-Moulton[13], was due to my familiarity with it in previous course-work, as well as allowing larger time steps and better accuracy than lower order methods. Other solvers can be implemented quite easily however, in the Extensible Simulator, which will be discussed in Chapter 4.

### 3.1.1 Runge-Kutta Fourth-Order Method

In this project we use the RK4 method that ships with Scilab, however an explanation of the method is in order. If we consider the general differential equation

$$\dot{y}(x) = f(x, y(x)), y(0) = y_0$$

then the RK4 method for the problem is given by the following equation:

$$y(x+h) = y(x) + \frac{1}{6}(F_1 + 2F_2 + 2F_3 + F_4) \tag{3.1}$$

where

$$\begin{aligned}F_1 &= hf(x, y) \\F_2 &= hf\left(x + \frac{h}{2}, y + \frac{F_1}{2}\right) \\F_3 &= hf\left(x + \frac{h}{2}, y + \frac{F_2}{2}\right) \\F_4 &= hf(x + h, y + F_3)\end{aligned}\tag{3.2}$$

The four different functions above are used to determine a weighted average of the slope between the present value ( $y(x)$ ) and the next value ( $y(x + h)$ ).  $F_1$  represents the slope at the present value,  $F_2$  represents the slope at the midpoint between the present value and the next step size, using the  $F_1$  to arrive at the  $y$  value,  $F_3$  is the slope at the midpoint again, using  $F_2$  to arrive at the  $y$  value, and finally  $F_4$  is the evaluation of the slope at  $y(x + h)$ , using  $F_3$  to arrive at the  $y$  value. These four slopes are then averaged, with more weight given to the middle values,  $F_2$  and  $F_3$ , to find the next value,  $y(x + h)$ . Below, you will find an example implementation of this method.

---

*Listing 3.1: Implementation of Runge-Kutta Fourth-Order solver.*

---

```
function y=rk4(f, t0, y0, t)
//we agree that we're going to
//put t0 at the beginning at the list.
//the y values will be the height at
//the t values.
m=max(size(y0));
```

```

n=max(size(t));
y=zeros(n,m);

if t(1) == t0 then
    y(1,:) = y0;
else
    h = t(1)-t0;

    F1 = h .* f(t0, y0);
    F2 = h .* f(t0+h/2, y0+F1/2);
    F3 = h .* f(t0+h/2, y0+F2/2);
    F4 = h .* f(t0+h, y0+F3);

    y(1,:) = y0 + (F1 + 2 .* F2 + 2 .* F3 + F4)./6;
end

for i=1:n-1
    h = t(i+1)-t(i);

    F1 = h .* f(t(i), y(i,:));
    F2 = h .* f(t(i)+h/2, y(i,)+F1/2);
    F3 = h .* f(t(i)+h/2, y(i,)+F2/2);
    F4 = h .* f(t(i)+h, y(i,)+ F3);

    y(i+1,:) = y(i,:) + (F1 + 2.*F2 + 2.*F3 + F4)./6;
end

```

**end**

endfunction

---

## 4. EXTENSIBLE SIMULATOR

### 4.1 *Introduction*

The Extensible Simulator is a tool that was created using a combination of the Python programming language and Scilab, and is able to simulate celestial bodies. It contains three major parts: the GUI, the simulation, and the API.

In order to simulate an arbitrary set of bodies, we start with the GUI which is where the main constructs called “projects” are created. Projects encapsulate the set of bodies to be simulated, while also providing the duration of the simulation, and other items that are essential to the simulation running properly. The bodies that are a part of a project are flat files, which are edited by hand within the Extensible Simulator, and specify the different physical properties of each body, such as mass, initial velocity, and many other attributes which we shall discuss in depth in Section 4.3.2. Based on the bodies defined in a project, we are able to calculate their positions for the specified time interval, by choosing the desired numerical integration method and gravitational function.

Once the calculation of the bodies’ positions are complete, the user is able to run the simulation, which animates the bodies that were specified by the user. In addition to the bodies being set in motion, there are also different camera views that focus on each one. One can also move the camera up, down, forward, backward, left and right



using specified keystrokes (Table 4.1).

In order to make this simulator extensible, an API is available to the prospective programmer. The four main characteristics that makes this simulator interesting and most useful are its camera views, the shapes and behavior of the celestial bodies, the gravitational functions, and the numerical methods used to calculate the positions of the bodies. It follows naturally then, that making these characteristics extensible will be of great benefit, and will allow many different and highly detailed simulations to be possible. One will be able to essentially plug in different camera and body definitions that will be derived from their respective base classes, and specify different numerical methods and gravitational functions for calculating the bodies' positions. This will be discussed in depth in Section 4.5.

#### *4.2 Basic Structure of the Graphical User Interface*

As stated before, the main function of the user interface is to manage sets of bodies contained in a project. As you can see in Figure 4.3, the user interface is composed of two major panels, a file menu, a help menu, two buttons, a few toolbar icons, and finally two drop-down menus. The left panel lists the different bodies in the project, while the right panel is reserved for the editing of the body configuration files. The file menu allows the execution of various functions such as creating new projects and files, opening existing projects and files, copying a project to a new project, showing the differences between two projects, and saving files. You can also exit the application through the file menu or the close button in the top right corner of the application. The two buttons that are below the main panels allow one to calculate the values

needed by the simulation and simulate the project given its contents. The toolbar icons perform some of the same functions as the file menu, such as saving a file, creating a new file, and opening a file, and finally the two drop down menus allow the selection of the numerical method and gravitational function to be employed.

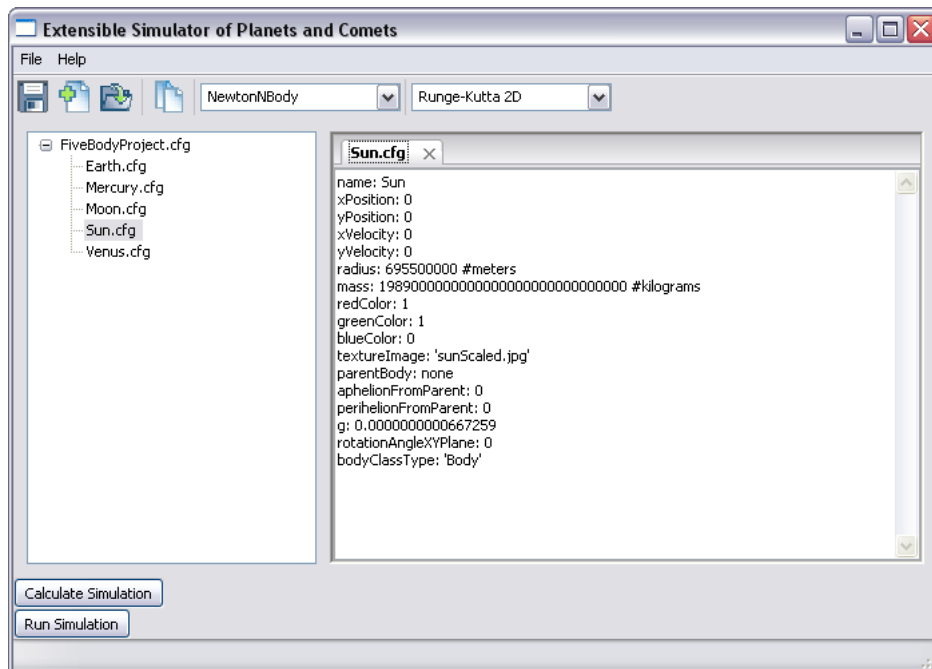


Fig. 4.1: Screenshot of the Extensible Simulator.

### 4.3 Python Graphical User Interface - Project Organizer

Now that the basic structure is defined, we can get into some of the more hair-splitting details of the GUI. To keep this in the most logically flowing order, I'll begin by explaining how to create a project, and move on to file creation, editing, saving, project copying, and showing the history between two projects.

### 4.3.1 *Creating A Project*

When creating a new project, one would start by selecting New → New Project from the file menu. An “Add New Project” dialog manifests itself, and asks for three items:

1. Choose a Directory
2. Project Name
3. Elapsed Time in Earth Years

The directory choice along with project name are self explanatory, however the elapsed time in Earth years refers the length of time for the project simulation. The real-time duration obviously does not take this long due to the rate which OpenGL renders its frames, and the step size used to perform the calculations. The rate at which the simulation renders in real time is also dependant on other programs running on the machine that are using up resources, and the amount of objects rendered on screen, which I will discuss in section 4.4.

Once the values for the new project have been specified, many things occur behind the scenes as preventative measures to errors, and providing essential components for the project. First, we check to see if a project of the same name exists in the project folder, and if so, we prevent that from happening and ask the user to rename the project. Once we get validation that a unique project name has been selected, a project folder is created to house all files and folders related to it.

A project configuration file is created, and is named in accordance with the project name. This file contains the values specified during the creation of the project, which are the absolute path of the project, the duration that the simulation is to last, and

finally the configuration type. This final value is used to differentiate between project configuration files and body configuration files that make up the files of a project. These files are parsed by a specific configuration module of Python, so within the scope of this application, the configuration type prevents body configuration files from being opened as projects.

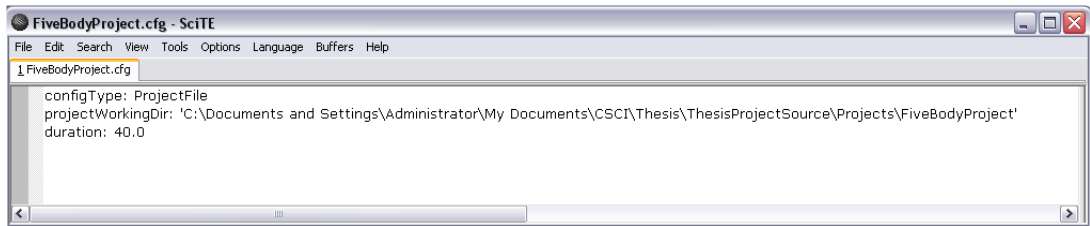


Fig. 4.2: Screenshot of a project configuration file.

Still within our project directory, four more sub-folders are created. A body definition folder is created that houses the body configuration files that I spoke of above. The positions of each body after the calculation are stored in their own folder named “BodyCoordinates”, while the different statistics of each calculation/simulation of the project are housed in a “Benchmarks” folder. Finally, an “Images” folder is created that is to contain any images to be used for texture-mapping the bodies during the simulation.

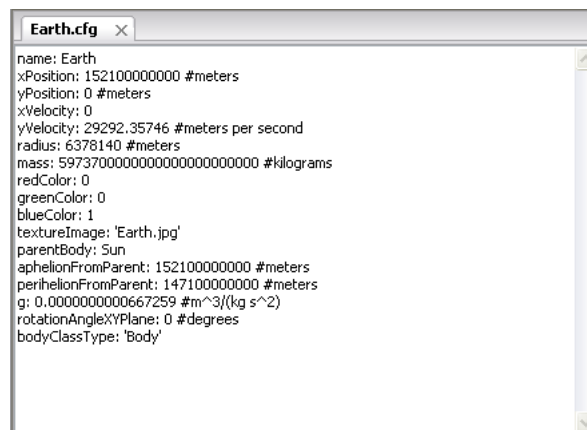
### 4.3.2 *Creating A Celestial Body*

This is where the body configuration files come into play. These files are very significant to the simulator, as they not only provide the specific information for the calculation which leads to the exact positions of the planets to render, but it is also used in the simulator itself, by telling it how the body should be rendered on the

screen, for example, the body's color and shape. Each body that is created has its own configuration file with the following values:

1. Name - The name of the celestial body.
2. Initial  $x$ -coordinate position - The  $x$ -coordinate of where the body will start its simulation.
3. Initial  $y$ -coordinate position - The  $y$ -coordinate of where the body will start its simulation.
4. Initial  $x$ -coordinate velocity - The initial velocity of the body in the  $x$  direction with meters/second as the units.
5. Initial  $y$ -coordinate velocity - The initial velocity of the body in the  $y$  direction with meters/second as the units.
6. Radius - The radius of the body in meters.
7. Mass - The mass of the body in kilograms.
8. Red color component - The following color components are used in the OpenGL script to render the colors of the planets if no image is given for texture mapping. These values range from zero to one.
9. Green color component - See above.
10. Blue color component - See above.
11. Texture image - The filename of the image to use in texture mapping the body. If there is no plan to use texture mapping, a value of 'none' should be put in its place.

12. Parent body - The name of the body who is the parent to this body.
13. Aphelion from parent - The distance that is furthest away from the parent in meters.
14. Perihelion from parent - The distance that is the closest to the planet in meters.
15. Gravitational constant - This should remain the same for all planets in all solar systems, at  $6.67259 \times 10^{-11}$ .
16. Rotation angle on the  $xy$  plane - A degree value that you wish to rotate your initial conditions stated above.
17. Body class type - 'Body' is the base class type used for any celestial body. We will discuss different possible values in section 4.5.



```
Earth.cfg x
name: Earth
xPosition: 152100000000 #meters
yPosition: 0 #meters
xVelocity: 0
yVelocity: 29292.35746 #meters per second
radius: 6378140 #meters
mass: 5973700000000000000000 #kilograms
redColor: 0
greenColor: 0
blueColor: 1
textureImage: 'Earth.jpg'
parentBody: Sun
aphelionFromParent: 152100000000 #meters
perihelionFromParent: 147100000000 #meters
g: 0.000000000667259 #m^3/(kg s^2)
rotationAngleXYPlane: 0 #degrees
bodyClassType: 'Body'
```

Fig. 4.3: Screenshot of a body configuration file.

There are a few things worth noting here in regards to these body values. First, I would like to address the initial  $x$  and  $y$  positions and velocities. Take care to make sure that the correct values are used. If we have a planet starting off at aphelion, then

we must make sure that the appropriate velocity is applied. This is not the planet’s mean velocity, but its velocity at aphelion, which would be traveling at its slowest velocity. The equations for the velocities at perihelion and aphelion are as follows:

$$v_p = \frac{2\pi a}{p} \sqrt{\frac{1+e}{1-e}}$$

$$v_a = \frac{2\pi a}{p} \sqrt{\frac{1-e}{1+e}},$$

where  $a$  is the length of the semi-major axis of the ellipse,  $p$  is the sidereal period of the planet in seconds, and  $e$  is the eccentricity of the orbit.

Second, the ‘Parent Body’ name of the child must coincide with the ‘Name’ value of the parent. If there is no parent body to a particular body, for example, the Sun has no parent body, then a value of ‘none’ should be inserted. If we are creating the Earth as one of our bodies, then this value for the parent body should be ‘Sun’, or whatever the ‘Sun’ was named in its body configuration file.

### 4.3.3 File Editing

File editing is fairly straight-forward. You can open a file for editing by either double-clicking the file you wish to edit in the celestial body tree, right-clicking on the file in the tree, and selecting “Edit” from the drop-down menu, or by opening the file menu, and selecting Open → Open File. You may edit the file however you wish, just keep in mind that the Config module reads the data in a specific format. See APPENDIX B for the specifics of the Config module. Note that scientific notation is not yet supported.

#### 4.3.4 *File Saving*

The file saving feature currently only saves body configuration files, and does not save project files or anything else. The GUI does not let the user know if a file has been modified from its original version, which could be a feature implemented in future versions. What it does exactly, is take what is on the screen, and re-writes it all back to the file it originated from. It's not very fancy, but it does the job.

#### 4.3.5 *Project Copying*

When building this simulator, we thought it would be nice to include the ability to copy an existing project to a brand new project. This feature yields an exact copy of the project you specify, only with a different project name (all project variables reflect the name change). It may sound like it has no purpose initially, but the intention was to cut out time wasted in creating a project that was slightly different from another, in order to view how much of an effect that slight difference would have between the two projects. For example, if we had a project that contained celestial bodies such as the Sun, Mercury, Venus, Earth, and Earth's Moon, and we wanted to see what effect adding Mars and Jupiter into the equation would have on the system, we would create a new project from the initial project, and simply add in Mars and Jupiter to the new project. This begs the question, how do we see how these two projects differ? This brings us to the next section, the history between two projects.



#### 4.3.6 *History Between Two Projects*

As stated above, it would be nice to see how two projects differ when other elements are tweaked, like adding in an additional two planets in the example above. The history feature compares the difference between the orbits of the corresponding bodies in two different projects and outputs a graph indicating those differences.

#### 4.4 *Python Graphical User Interface Simulation*

After the task of creating the files for each body in a project is complete, it's time to see what the system looks like graphically. Before we can run the simulation, we must first calculate the values of our system in order to run the simulation and plot the bodies where they need to be. It is as simple as pressing the "Calculate Simulation" button, which invokes the Scilab script, using the function selected in the drop-down menu, and solving it using the numeric method specified in the second drop down menu.

The application will let the user know that the calculation is complete by saying so in the application's status bar. Depending on how many bodies are present for the calculation, this may take roughly 5 minutes or 300 seconds for five bodies or less. Once the calculation is complete, the simulation can be run by pressing the "Run Simulation" button. This launches the OpenGL rendering of the planets using the body configuration files, and the coordinates that resulted from the calculation. In the initial stages of development, we were having trouble with the rendering of the planets, as it was very spotty. After much trial and error, we came to realize that the units of position and radius were quite large, so we convert everything to astronomical

units(AU) before rendering. The speed of the simulation itself, also depends on how many objects you render on the screen, as well as how many other programs you have running. When drawing the trails that follow a portion of the trajectory of the body, I realized that the simulation got slower and slower as the simulation progressed. Once I limited the portion of the trail to draw(hence less objects to render), the simulation rendered at a more constant rate.

Now that we've got a simulation rendering, we must discuss scene navigation. To aid in the understanding of the underlying simulation structures, I've included Figure 4.4.

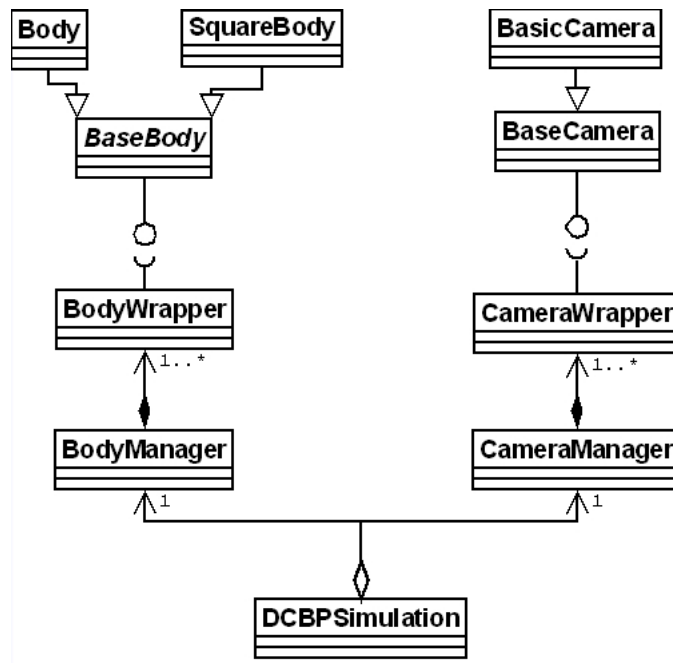


Fig. 4.4: Unified modeling language diagram of simulation components.

There are keystrokes that can be executed during a simulation that help navigate the scene during the simulation. The major simulation functions include switching

the view from body to body, moving the camera left, right, up, down, forward, and backward while looking at a body, scaling the bodies, resetting the camera view, and exiting the application. If at any time the user gets disoriented due to excessive camera movement, then they can re-set the camera to the pre-defined position, which lies at  $x = 0$ ,  $y = 0$ , and  $z$  is the  $z$  viewing distance of the body which is its radius multiplied by 150 meters. I picked this value because it rendered the body at a reasonable viewing distance.

Moving the camera left, right, up, and down will move the camera by 200,000,000 meters each step. Moving forward and backward is a little more involved, because we don't want the user ending up inside a body, where they would see nothing but black space. To move forward, I divide the  $z$  viewing distance in half, which brings the camera closer to the body. This process will repeat until the camera has come within two of the bodies radius. After the camera gets this close, pressing 'f' to move forward will keep you right where you are, two radii away from the body. To move backward, we do the opposite by multiplying the  $z$  viewing distance by two. This posed a problem of clipping if one moved too far back because of the viewport previously defined. This was solved by increasing the viewport length to accommodate all bodies.

The absolute scaling is invoked by the 'a' key, where the bodies get enlarged so we can actually see them during the simulation. The algorithm to compute the scaling factor starts by finding the two bodies that have the shortest distance between each other in the entire system. We then enlarge the radius of the two planets until the distance between them is no more than 30 times the smaller radius. The value of 30 was chosen because by trial and error, it provided the most reasonable rendering.

Below, we include the code for this algorithm.

*Listing 4.1: Algorithm for absolute scaling factor.*

---

```
def findScalingFactor(self):

    #find the shortest distance between any two bodies, take
    #the radius of the larger and smaller and let the
    #algorithm run.

    #Now to find the shortest distance between two bodies.

    #set shortest distance to the maxint value of the machine,
    #so that when our first iteration goes around, it
    #automatically gets set to something lower within our
    #body system.

    shortestDistance = sys.maxint

    i = 0
    j = 0

    for i in range(len(self.bodies)):
        for j in range(len(self.bodies)):
            if i < j:

                distance = sqrt( \
                    pow((self.bodies[j].xPosition - \
```

```

        self.bodies[i].xPosition),2) + \
        pow((self.bodies[j].yPosition - \
            self.bodies[i].yPosition),2))
if distance < shortestDistance:
    shortestDistance = distance
    if self.bodies[i].radius > \
        self.bodies[j].radius:
        radiusSmaller = \
            self.bodies[j].radius
        radiusLarger = \
            self.bodies[i].radius
    else:
        radiusSmaller = \
            self.bodies[i].radius
        radiusLarger = \
            self.bodies[j].radius

newSmallRadius = radiusSmaller
newLargeRadius = radiusLarger
adjustedDistance = shortestDistance
distance = adjustedDistance

```

```

while adjustedDistance > (radiusSmaller * 30.):
    newSmallRadius = newSmallRadius + \
        (radiusSmaller * .1)
    newLargeRadius = newLargeRadius + \
        (radiusLarger * .1)
    adjustedDistance = distance - \
        (newSmallRadius + newLargeRadius)

return newSmallRadius/radiusSmaller

```

---

Table 4.1 contains the different keystrokes available to navigate the scene. You'll notice that there are two groups of keys in the figure. The first group are keystrokes defined in the simulation file, and will always remain a part of the simulation. The second group, however, is dependent on the camera used for the simulation, and as this program grows, these may change.

#### 4.5 *Application Programming Interface*

The API is the very component of the Extensible Simulator that makes the program extensible. Bodies, cameras, gravitational functions and numeric methods can all be extended or replaced with ease. There is a Utilities file that keeps track of different constants and library functions. It is here where the different string variables describing the different subclassed bodies and cameras reside, in addition to string constants describe the different gravitational functions and numerical methods.

Tab. 4.1: Keystrokes for the Simulator.

Key	Description
+	Increments the body index.
-	Decrements the body index.
a	Scales the bodies so they are more visible.
n	Renders the bodies as their normal sizes.
e	Exit the application.
↑	Move in the positive y direction.
↓	Move in the negative y direction.
→	Move in the positive x direction.
←	Move in the negative x direction.
b	Move in the positive z direction (backwards).
f	Move in the negative z direction(forwards).
r	Resets the scene.

These constants are used in the respective wrapper classes of the bodies and cameras to instantiate the proper class when called. The ability to add different bodies and different behaviors of bodies is of great interest, because we are continually learning of the different properties they possess, and the different things that affect them. For a general idea of the relationships between these components, refer to Figure 4.4. The code for the objects referenced in the following sections can be found in APPENDIX A.

#### 4.5.1 Bodies

The *BaseBody* is the object that all bodies are to be derived from. It contains the bare essentials to render a body on screen when called, including the ability to initialize and draw itself, read its respective coordinates file, return its  $z$  viewing distance, load its textures, and draw a trail that traces its path. To create a plug-in that contains your own definitions of a body and its behavior, simply override the given methods, register your new body type in the *Utilities* object as a string constant, followed by including its instantiation in the *BodyWrapper* initializations.

Keep in mind, that in addition to deriving from the *BaseBody* object, we can add new attributes and methods as long as they are incorporated into the *BodyWrapper* object. The script that renders the body (*DynamicCelestialBodyPositionSimulation.py*) only accepts the *BodyWrapper* type, so any new methods you wish to call from your new method or attribute must be “wrapped” inside this object.

While used in the simulation script, the *BodyWrapper* type that encapsulates the many different *BaseBody* subclasses is managed by the *BodyManager* object. The



function of this object is to manage the array of bodies as a whole, ranging from their instantiation by means of their respective configuration, to drawing, and finally finding the scaling factor that suits the system of bodies as a whole.

One question has yet to be answered. Where does the *BodyWrapper* get the information on what type of body to instantiate? When we talked about creating a new body configuration file in Section 4.3.2, we included the mention of a *BodyClassType*. This is the key to creating as many bodies as desired with the new body definition. The name here must coincide with the string constant that was placed in the *Utilities* object.

#### 4.5.2 Cameras

Cameras are structured in the same manner as bodies, meaning that they have a *CameraWrapper* object that instantiates the correct camera by comparing it with the respective string constant values found in the *Utilities* object. Like their *BaseBody* counterpart, the *BaseCamera* contains the basic functions for navigating and viewing a simulation, with can be used to create subclasses of cameras that override the existing methods. The methods that may be overridden include initialization, defining the camera viewport, specifying which body to look at, and the ‘special’ and ‘keyboard’ OpenGL functions.

Just as the bodies, if any *BaseCamera* subclasses include additional methods or attributes, they must be “wrapped” by the *CameraWrapper* object if it is to be called from the simulation script. The cameras also have a *CameraManager* which handles the instantiation of the different cameras, however unlike the bodies, we specify the

camera type explicitly in the *CameraManager*'s *GetCameras* method.

### 4.5.3 Gravitational Functions and Numeric Methods

Creating plug-ins for different gravitational functions and numerical methods is a complete shift from how the body and camera plug-ins are implemented. The only thing that remains the same is the necessity to insert the string constant describing the method into the *Utilities* object. To go about creating a new gravitational function, create a Scilab file that contains only the new function, and a variable declaration, *gravFunc*, which is set to your function's name. This file will live in the root directory of this program. Register this file with the *Utilities* file, and it should appear in the main GUI automatically.

As for the numeric method, this file contains all of the setup needed in order to populate the initial matrix that gets fed into the solver. This means reading all of the data from the body configuration files, using the *BodyConfig.sci* script, reading the project data of the particular project to get the duration of the simulation, which plays a part in the step size of the solver(*ProjectFile.sci*). To determine what gravitational function to apply, we use the *gravFunc* variable that is declared in the previous file that we spoke of. After reading in all of the body configuration files, we then apply Given's rotations to each body, taking into account that the Moon, for instance will have to be rotated the same amount as the Earth in order for it to be in the correct position. The code currently used for setup is found below:

*Listing 4.2: Code snippet of initial matrix setup for the solver.*

---

```
//construct the initial conditions.
```

```

initialPositions=[]

//get all bodies initial conditions prior to rotation ,
//so we can rotate children properly.
for i=1:numBodies
    initialPositions(1,i) = strtod(bodies(xPosition,i))
    initialPositions(2,i) = strtod(bodies(yPosition,i))
end

for i=1:numBodies

    //apply Given's rotations here.
    //for the angle, we have to get the radians from
    //the degrees input into the body config files.
    bodysRotationAngle = (
        strtod(bodies(rotationAngleXYPlane,i)) * %pi)/180 ;
    GivensMatrix = [
        cos(bodysRotationAngle) -sin(bodysRotationAngle)
        sin(bodysRotationAngle) cos(bodysRotationAngle) ];

    //need to find the parent's xy values , so we can
    //subtract from the childs initial conditions.
    //we need to do this so we make sure that we're

```

```

//rotating about the parent, and not anything else.
//example: in the moon's case, we want to rotate about
//the earth, not the sun.
originalXPositionChild = strtod(bodies(xPosition,i))
originalYPositionChild = strtod(bodies(yPosition,i))
originalXPositionParent = 0
originalYPositionParent = 0
for j=1:numBodies
    if bodies(parentBody,i) == bodies(name,j) then
        originalXPositionParent = initialPositions(1,j);
        originalYPositionParent = initialPositions(2,j);
    end
end
adjustedXPosition = originalXPositionChild -
                    originalXPositionParent;
adjustedYPosition = originalYPositionChild -
                    originalYPositionParent;

originalPosition = [ adjustedXPosition
                     adjustedYPosition ];

originalVelocity = [ strtod(bodies(xVelocity,i))
                    strtod(bodies(yVelocity,i)) ];

```

```
adjustedPosition = GivensMatrix * originalPosition
```

```
adjustedVelocity = GivensMatrix * originalVelocity
```

```
adjustedPosition = [ adjustedPosition(1,1) +  
                    originalXPositionParent  
                    adjustedPosition(2,1) +  
                    originalYPositionParent ]
```

```
xPosDifference = adjustedPosition(1,1) -  
                originalXPositionChild
```

```
yPosDifference = adjustedPosition(2,1) -  
                originalYPositionChild
```

```
bodies(xPosition, i) = string(adjustedPosition(1,1))
```

```
bodies(yPosition, i) = string(adjustedPosition(2,1))
```

```
bodies(xVelocity, i) = string(adjustedVelocity(1,1))
```

```
bodies(yVelocity, i) = string(adjustedVelocity(2,1))
```

```
for k=1:numBodies
```

```
    if bodies(name, i) == bodies(parentBody, k) then
```

```
        //we move the smaller (k).
```

```

//first , we have to move the child body the same
//amount as the parent body moved, while not
//rotating it .
//Then, if the child body itself needs rotated ,
//then it will be handled in the previous for
//loop , not this one.

posX = strtod(bodies(xPosition ,k));
posY = strtod(bodies(yPosition ,k));

adjPosx = posX + xPosDifference;
adjPosy = posY + yPosDifference;

bodies(xPosition ,k) = string(adjPosx);
bodies(yPosition ,k) = string(adjPosy);
end

end

end

//this is outside the initial for loop to make sure that
//all the rotated velocities have been
//calculated prior to adding parent velocities to the

```

```

//child.
for i=1:numBodies
//add the velocity of the parent body to the child body.
    for j=1:numBodies

        //ex if the moon's (i) parent body == the earth(j)
        if bodies(parentBody,i) == bodies(name,j) then

            velXI = strtod(bodies(xVelocity,i));
            velXJ = strtod(bodies(xVelocity,j));
            velYI = strtod(bodies(yVelocity,i));
            velYJ = strtod(bodies(yVelocity,j));

            theXSum = velXI + velXJ;
            theYSum = velYI + velYJ;

            bodies(xVelocity,i) = string(theXSum);
            bodies(yVelocity,i) = string(theYSum);

        end

    end

end

//initial condition matrix setup.

```

```

z0 = zeros(max(size(bodies)) * numBodies);
for i=1:numBodies
    z0(xPosition + (numAttributes*(i-1)) ) =
        strtod(bodies(xPosition , i));
    z0(yPosition + (numAttributes*(i-1))) =
        strtod(bodies(yPosition , i));
    z0(xVelocity + (numAttributes*(i-1))) =
        strtod(bodies(xVelocity , i));
    z0(yVelocity + (numAttributes*(i-1))) =
        strtod(bodies(yVelocity , i));
    z0(radius + (numAttributes*(i-1))) =
        strtod(bodies(radius , i));
    z0(mass + (numAttributes*(i-1))) =
        strtod(bodies(mass , i));
    z0(g + (numAttributes*(i-1))) =
        strtod(bodies(g , i));
    z0(rotationAngleXYPlane + (numAttributes*(i-1))) =
        strtod(bodies(rotationAngleXYPlane , i));
end

```

---

Refer to *RungeKutta2D.sci* in APPENDIX A for the specifics.



## 5. BENCHMARKS AND ERROR ANALYSIS

### 5.1 Benchmarks

The following benchmarks were performed both pre-API and post-API with the smaller step-size, measuring the time to calculate the coordinates of all bodies contained within a project. The calculations were done on an Intel Core Two Duo CPU T7300 @ 2.00 GHz and 1GB of RAM. The only program running while performing the calculations was the Wingware Python Intelligent Development Environment, which was used for the development of this project. It launches the Extensible Simulator, and from there we initialize the calculation by pressing the “Calculate Simulation” button.

Tab. 5.1: Pre-application programming interface with large step size.

Project Name	Duration(seconds)
TwoBodyProject	11.971
ThreeBodyProject	19.528
FourBodyProject	30.227
FiveBodyProject	44.694

Each calculation was run 10 times, and with a duration of 1 year, and the average time was taken.

The difference in the pre and post API calculation times is due largely to the smaller step size in the latter. This is discussed in depth in Section 5.2.

Tab. 5.2: Post-application programming interface with small step size.

Project Name	Duration(seconds)
TwoBodyProject	82.256
ThreeBodyProject	132.539
FourBodyProject	204.187
FiveBodyProject	295.555
SevenBodyProject	542.719

## 5.2 Error Analysis

Error analysis was performed on the accuracy of the measured aphelion and perihelion versus their true values, and also the angle at which aphelion and perihelion occurs versus their starting angle. Specifically, I looked at the relative error between the measured and true values, which can be defined as

$$\epsilon = \frac{|v_{approx} - v_{true}|}{|v_{true}|},$$

where  $\epsilon$  is the resulting error, which portrays the digits of accuracy between the two values,  $v_{approx}$  is the measured value, and  $v_{true}$  is the expected value.

While running the initial tests to see how the RK4 ODE solver was handling the data, I realized that after a simulation time of 40 years, the error of the Moon was sky-rocketing. See Table 5.3 and Figure 5.1. It was then that I realized that my step-size was increasing exponentially to the duration of the simulation. I was dividing

the total duration time into 10,000 steps, and when the duration time increases significantly, the step sizes also become very large.

Tab. 5.3: Average error of ‘FiveBodyProject’ at 40 years/old step size.

	Aphelion	Perihelion	Aphelion Slope	Perihelion Slope
Mercury	$5.93 \times 10^{-5}$	$3.545 \times 10^{-4}$	$8.932 \times 10^{-2}$	$2.049 \times 10^{-1}$
Venus	$3.627 \times 10^{-4}$	$1.130 \times 10^{-3}$	$2.769 \times 10^{-2}$	$2.844 \times 10^{-2}$
Earth	$5.34 \times 10^{-5}$	$1.718 \times 10^{-3}$	1.155	1.737
Moon	$1.457 \times 10^{-2}$	$1.398 \times 10^{-1}$	46.384	45.887
Average Error	$3.655 \times 10^{-2}$	$3.576 \times 10^{-2}$	11.914	11.964
Total Average	5.988			

The step size was so large, that the calculated positions for the Moon became very inaccurate towards the end of a 40 year simulation. In fact, the RK4 solver could not finish integrating.

After finding this, I decided that negotiating step-size was essential. I experimented with a few different values, and came up with a step-size that varied based on simulation duration added to a constant. Using the new step sizes I was able to get a handle on the error for the Moon. This was the only body that had a problem with the errors due to it being a few orders of magnitude smaller than the other planets, and having a much faster period as well. This poses a problem, because with a much larger step size, one step could be a large portion of the Moon’s orbit, thus giving us a highly inaccurate value for its position.

The other factor to consider is the time it takes to calculate a simulation. With

a larger step-size, the time to calculate would be small, because there would be less calculations to perform.

Tab. 5.4: Average error of 'FiveBodyProject' at 40 years/new step size.

	Aphelion	Perihelion	Aphelion Slope	Perihelion Slope
Mercury	$6.400 \times 10^{-6}$	$2.163 \times 10^{-4}$	$1.447 \times 10^{-2}$	$3.276 \times 10^{-2}$
Venus	$3.636 \times 10^{-4}$	$1.132 \times 10^{-3}$	$6.446 \times 10^{-3}$	$6.701 \times 10^{-3}$
Earth	$5.34 \times 10^{-5}$	$1.717 \times 10^{-3}$	$8.923 \times 10^{-1}$	$8.924 \times 10^{-1}$
Moon	$1.867 \times 10^{-2}$	$5.384 \times 10^{-2}$	44.28	44.46
Average Error	$4.774 \times 10^{-3}$	$1.423 \times 10^{-2}$	11.30	11.35
Total Average	5.667			

With a smaller step-size, we would have more calculations to perform, resulting in a longer calculation time. My goal was to find a happy medium, that would give us accurate results in the shortest amount of time possible. This is why, for every Earth year, I multiply by a constant of 500 to get the optimal step size. This particular number yielded the most accurate solutions in the shortest amount of time, as you can see in Table 5.4 and Figure 5.2.

To see how the RK4 solver performs for a longer simulation duration, I've chosen to increase the duration to 100 years (See Figure 5.3 and Table 5.5). We are at an average of 2 digits of accuracy for the aphelion and perihelion of all the planets with the exception of the Moon. Our errors aren't too bad, but aren't great either if we adopt the general rule that 3 digits of accuracy is a fair result. This could be due to the fact that our system contains a mere 5 bodies compared to the eight that are

in our solar system. We are missing Jupiter, which has a very large mass, and as referenced in Section 1.1, could certainly have an affect on the system as a whole.

To test this concept, we do exactly that, and run the same tests again with Mars and Jupiter included(See Table 5.6 and Figures 5.4 and 5.5).

Tab. 5.5: Average error of 'FiveBodyProject' at 100 years/new step size.

	Aphelion	Perihelion	Aphelion Slope	Perihelion Slope
Mercury	$1.250 \times 10^{-5}$	$2.932 \times 10^{-4}$	$3.625 \times 10^{-2}$	$8.215 \times 10^{-2}$
Venus	$3.826 \times 10^{-4}$	$1.125 \times 10^{-3}$	$1.328 \times 10^{-2}$	$1.354 \times 10^{-2}$
Earth	$6.060 \times 10^{-5}$	$1.737 \times 10^{-3}$	$9.766 \times 10^{-1}$	$9.973 \times 10^{-1}$
Moon	$1.972 \times 10^{-2}$	$5.490 \times 10^{-2}$	$7.030 \times 10^{-2}$	45.44
Average Error	$5.043 \times 10^{-3}$	$1.450 \times 10^{-2}$	$2.741 \times 10^{-1}$	11.63
Total Average	2.982			

As a result of including Mars and Jupiter, we don't see much of a difference in the significant digits of our measurements due to relative error. Some factors that could result in less than favorable results are the angles of the true planets' orbits themselves. For these projects, we picked arbitrary angles that didn't necessarily reflect the measured angles at which they occur in our actual solar system. With the exception of the angles of the Moon's orbit, however we still get fair results at about 2 digits of accuracy. What really throws off our overall averaged errors are the angles at which the Moon's perihelion and aphelion occur. This could be due to a minor distance change in aphelion/perihelion that greatly affects the angle at which it occurs. Our step size in the RK4 algorithm may still be too large for the Moon's

orbit, which may be skipping over a significant enough area, which could skip right over a true aphelion/perihelion point resulting in a significant error at the angles at which they occur.

Tab. 5.6: Average error of ‘SevenBodyProject’ at 100 years/new step size.

	Aphelion	Perihelion	Aphelion Slope	Perihelion Slope
Mercury	$1.820 \times 10^{-5}$	$2.465 \times 10^{-4}$	$3.677 \times 10^{-2}$	$8.275 \times 10^{-2}$
Venus	$3.916 \times 10^{-4}$	$1.118 \times 10^{-3}$	$1.312 \times 10^{-2}$	$1.338 \times 10^{-2}$
Earth	$6.730 \times 10^{-5}$	$1.750 \times 10^{-3}$	1.004	$9.889 \times 10^{-1}$
Moon	$1.972 \times 10^{-2}$	$5.490 \times 10^{-2}$	45.35	45.32
Mars	$5.590 \times 10^{-5}$	$1.455 \times 10^{-4}$	$1.566 \times 10^{-2}$	$1.991 \times 10^{-2}$
Jupiter	$8.250 \times 10^{-5}$	$1.312 \times 10^{-3}$	$2.574 \times 10^{-4}$	$2.899 \times 10^{-4}$
Average Error	$3.389 \times 10^{-3}$	$9.912 \times 10^{-3}$	7.736	7.738
Total Average	3.872			

This can perhaps be avoided by employing a numerical method that treats systems containing varying periods.



Fig. 5.1: Relative error of 'FiveBodyProject' at 40 years/old step-size.

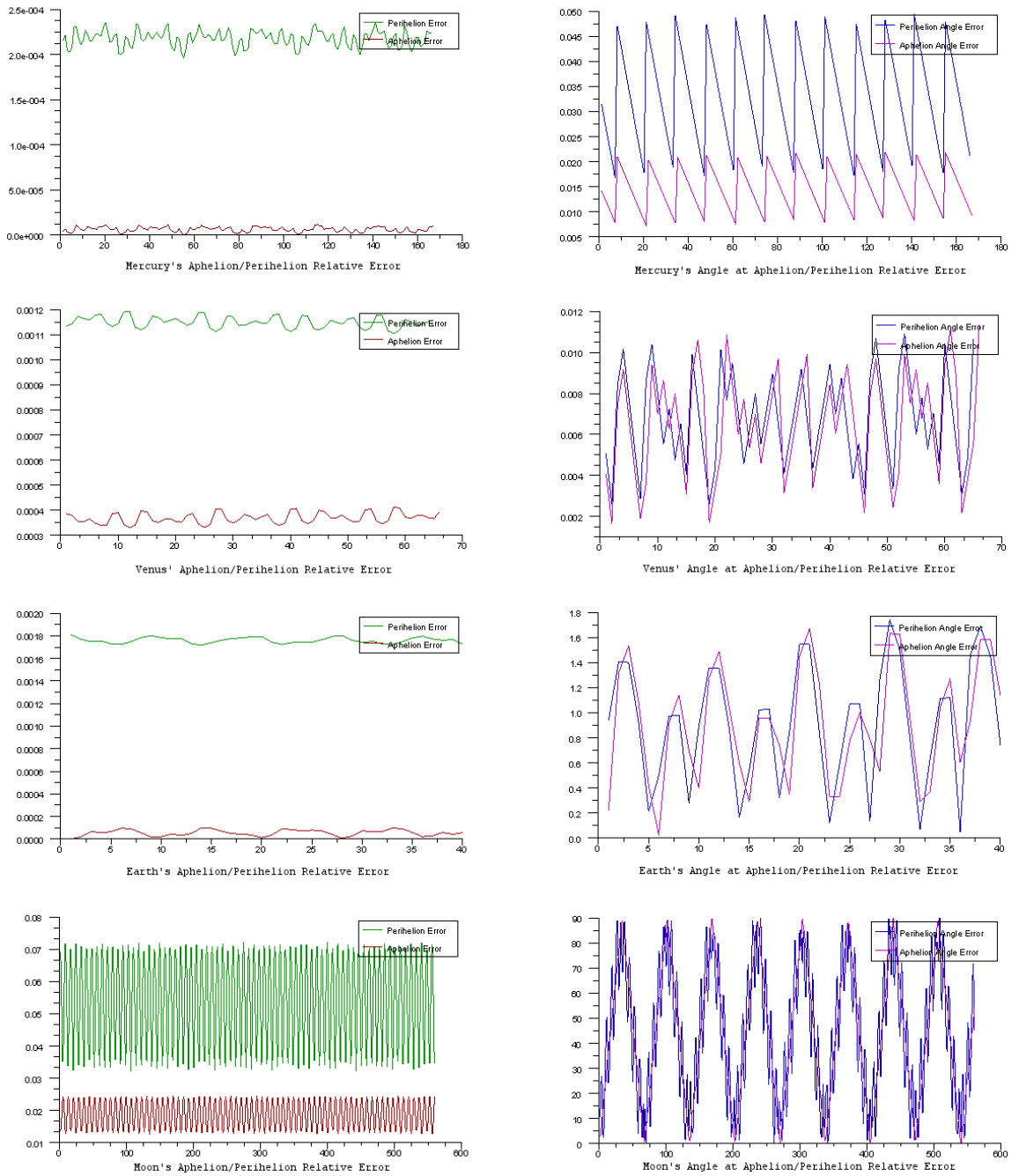


Fig. 5.2: Relative error of 'FiveBodyProject' at 40 years/new step-size.



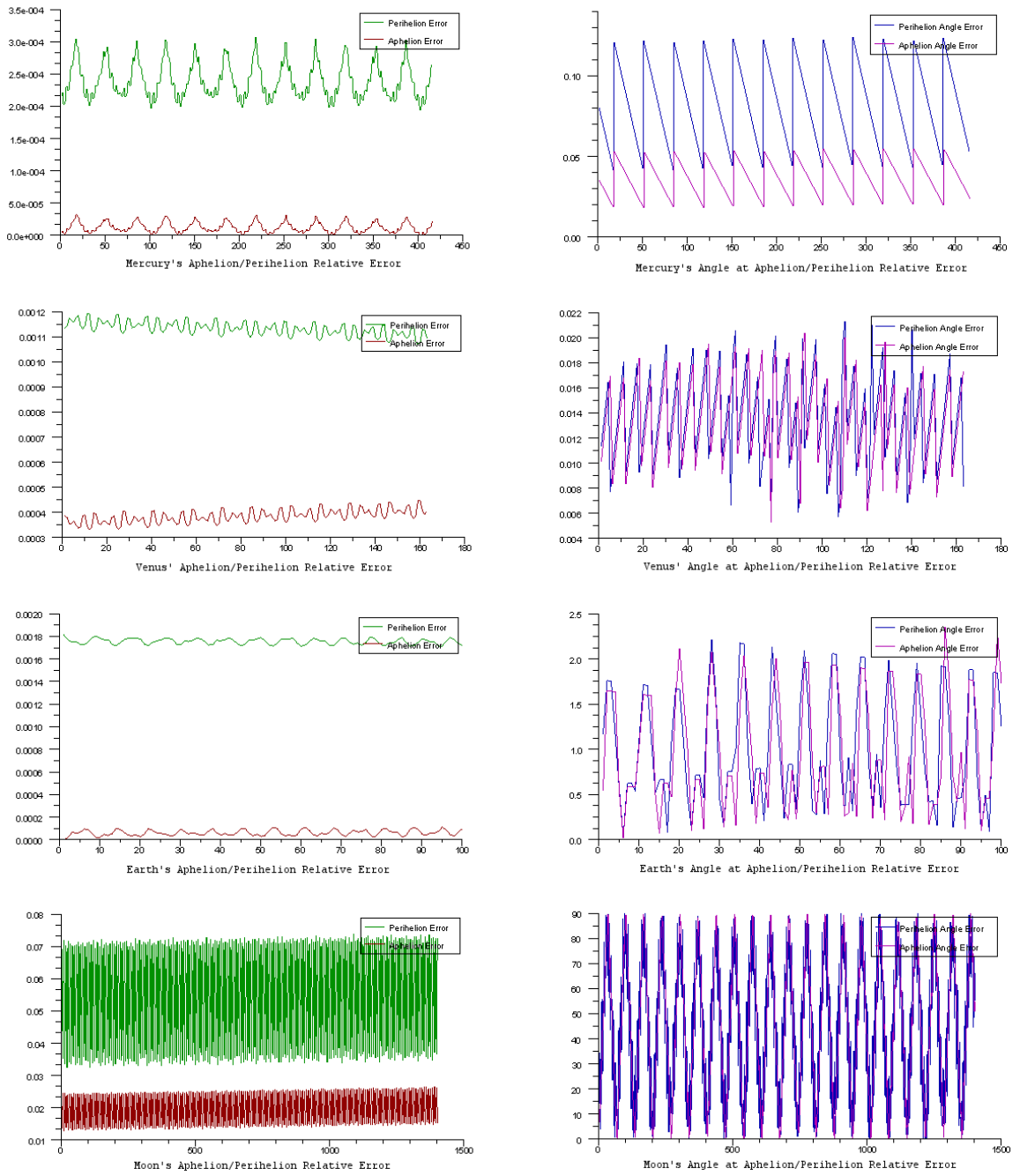


Fig. 5.3: Relative error of 'FiveBodyProject' at 100 years.

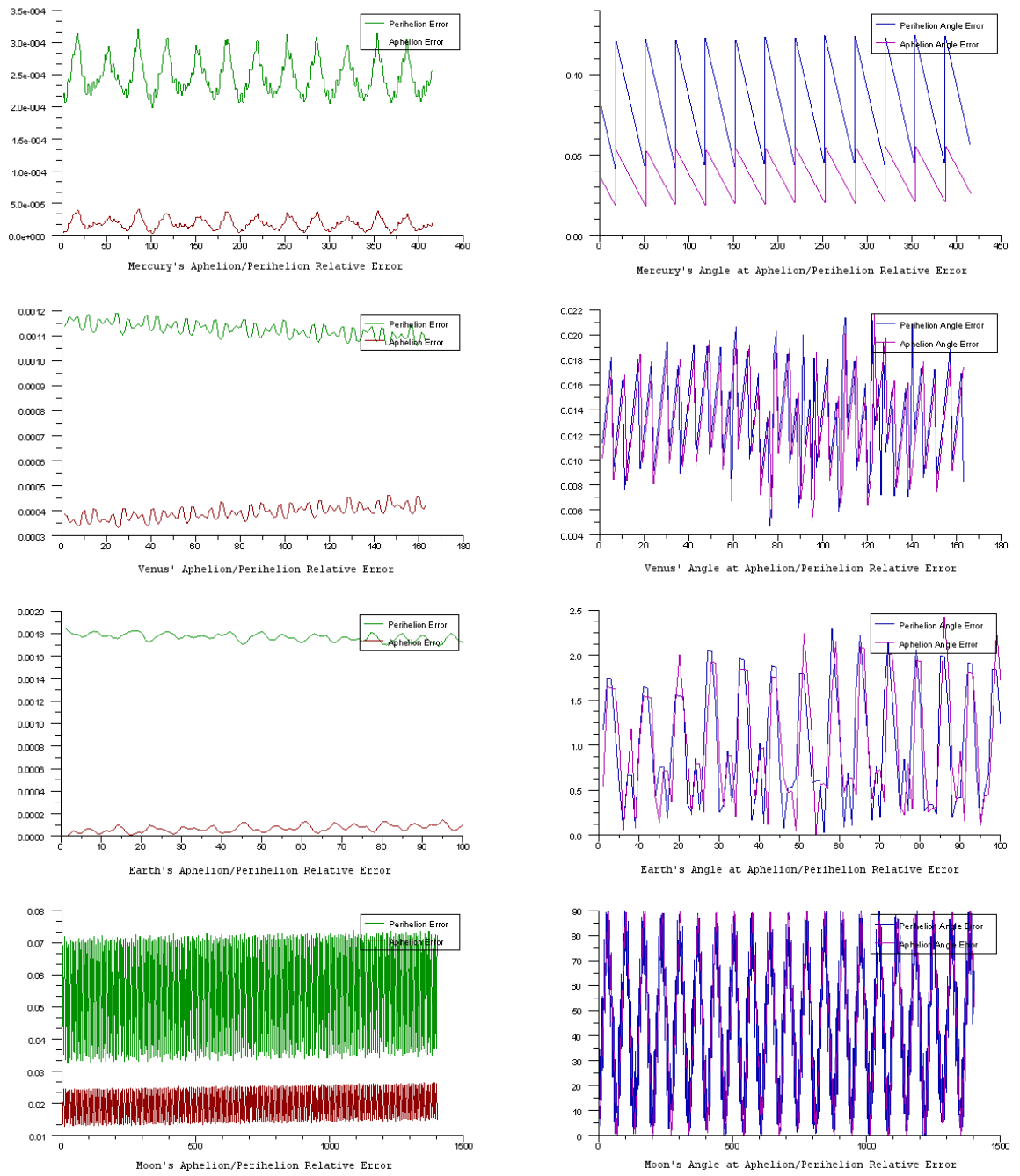


Fig. 5.4: Relative error of 'SevenBodyProject' at 100 years.

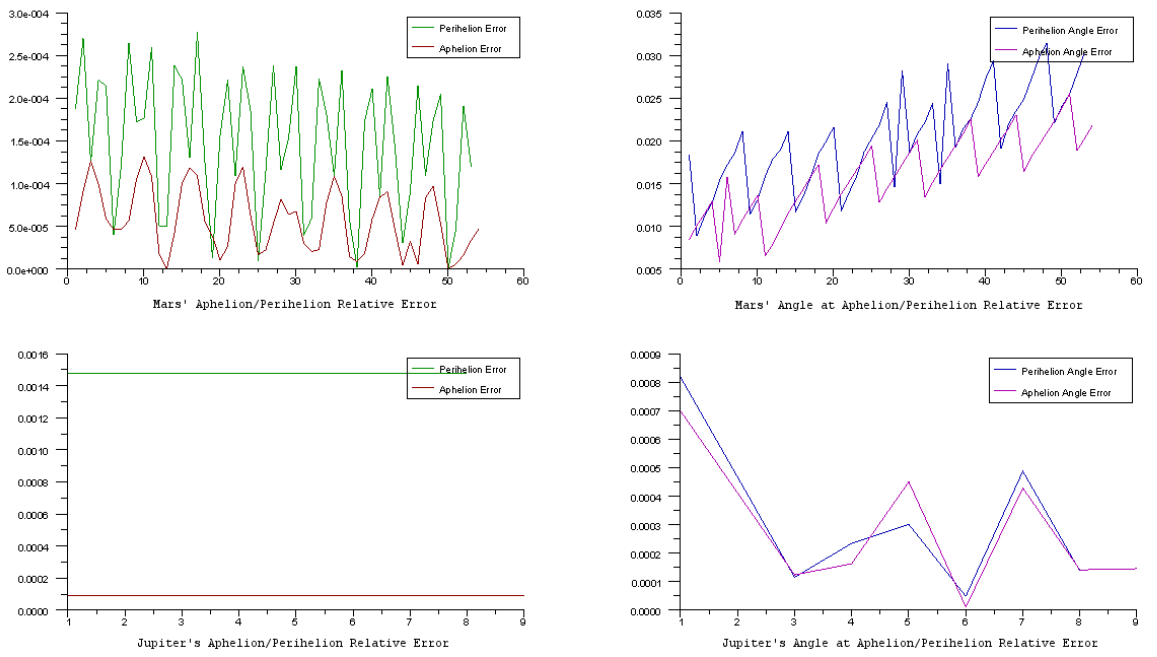


Fig. 5.5: Relative error of 'SevenBodyProject' at 100 years continued.

## 6. CONCLUSION

The aim of this research was to create an extensible simulator that will not only calculate the trajectories of planets and comets using a numerical method, but be able to graphically simulate them. This will be particularly useful to colleagues in the fields of astronomy, physics, and computer science, by being able to convey ideas easily to one another with confidence that the results are accurate to within an average of two digits with the exception of the Moon.

Additionally, we structured this program in such a way to make it easily extensible by providing an API to allow the extending of cameras, bodies, gravitational functions and numerical methods. These features are also very useful, because they allow scientists to add more features as needed with a great deal of ease. If one wanted to define a different camera that follows a specific path throughout the simulation, that could be easily achieved. If one wanted to define a body of a different shape, or add rotation to the shape, they can do so through the body portion of the API. The gravitational functions and numerical methods are scripts that can be interchanged easily. These factors give this program the potential to become a very robust simulator, capable of many different and intricate simulations.

The research conducted here, along with the created program can also facilitate research in the following ways:

1. Add the third dimension to the calculation of the trajectory taking into account each body's inclination to the ecliptic.
2. Dynamically speed up the simulation while the simulation is running.
3. Add a GUI to the actual simulation screen that allows easier ways of switching between cameras and determining which body the camera is looking at.
4. Add more numeric methods to the simulation to give the user different options. For instance, methods that treat different periods differently. This could aid greatly with the inconsistency of the Moon's results compared to the other bodies' results.
5. In the existing GUI, add the ability to edit project files and body configuration files more easily, instead of having to edit them by hand. Adding in the ability to input values in scientific notation would also improve the usability tremendously.
6. Programmatic video capture of a simulation would be of great interest, allowing one to capture a specific simulation, and be able to re-play it anywhere, eliminating the need to have a machine that contains the software needed to run the Extensible Simulator. This would aid greatly in presentations of specific scenarios.
7. Automatically calculate the necessary velocities at aphelion and perihelion for the user.
8. Develop an algorithm to determine an ideal step size based on the magnitude of the bodies, their orbits, and numerical method used.

In closing, this tool can help us understand a bit more, how our universe really moves. It enables communication over a broad spectrum of backgrounds and experiences from those in highly technical or scientific fields to those who are not, and can, but is certainly not limited to, helping us detect disasters that may threaten our existence on Earth, or aiding in the prediction of stable systems elsewhere.

## APPENDIX A

### MY CODE

The code contained here is only the code of the API that I thought relevant enough to include, as it is referenced extensively in Chapter 4. For instruction on obtaining the code in its entirety, please refer to APPENDIX B.

*Listing 1.1: Utilities.py*

---

```
import string
import sys
from config import Config
import os
import os.path

class Utilities(object):
    'This is a class that has generic utilities used within the project'

    #Used by the DynamicCelestialBodyPositionSimulation.py
    #file, along with others to resolve paths to the respective
    #directories
    BODY_DEFINITION_DIR = 'BodyDefinitions'
    BODY_COORDINATES_DIR = 'BodyCoordinates'

    #The project file type to determine that a .cfg is indeed
    #a project file, and not a body configuration file, or
    #anything else.
    PROJECT_FILE = 'ProjectFile'

    #Descriptions of the various body class types. These must be the same that
    #the user puts in each of the body configuration files.
    SQUARE_BODY = 'SquareBody'
    BODY = 'Body'

    #Descriptions of the various camera types.
    #This gets used in the Camera Manager and Camera Wrapper.
    BASIC_CAMERA = 'BasicCamera'
```



```

#Here we'll put the different strings that signify the different scilab
#calculation routines that are offered. The string should be what you want
#displayed in the combo box that selects it.
NUMERIC_METHODS = ["Runge-Kutta 2D"]
NUMERIC_METHOD_FILES = ["RungeKutta2D.sci"]

GRAVITY_FUNCTIONS = ["NewtonNBody"]
GRAVITY_FUNCTION_FILES = ["NewtonNBody.sci"]

AUSCALING_FACTOR = 1./149596000000000.

def IsGravFunction(self, scriptName):
    if scriptName == self.NEWTON_GRAV_FUNC:
        return True
    else:
        return False

def IsScilabScript(self, scriptName):
    #here we would add more conditionals as the number of scilab scripts
#increases.
    if scriptName == self.RUNGE_KUTTA_2D:
        return True
    else:
        return False

def ProjectExists(self, directoryName):
    return os.exists(directoryName)

def IsAProjectDirectory(self, directoryPath):
    if os.path.exists(directoryPath):
        filename = os.path.split(directoryPath)
        projectFile = filename[1] + '.cfg'
        if self.IsProject(projectFile):

```

```

        return True
    else:
        return False
else:
    return False

def IsProject(self, projectFileName):
    cfg = Config(projectFileName)
    try:
        projectType = cfg.configType
    except Exception, e:
        return False

    if projectType == self.PROJECT_FILE:
        return True
    else:
        return False

def NewProjectTemplate(self):
    return 'name: \n' + \
        'xPosition: \n' + \
        'yPosition: \n' + \
        'xVelocity: \n' + \
        'yVelocity: \n' + \
        'radius: \n' + \
        'mass: \n' + \
        'redColor: \n' + \
        'greenColor: \n' + \
        'blueColor: \n' + \
        'textureImage: \n' + \
        'parentBody: \n' + \
        'aphelionFromParent: \n' + \
        'perihelionFromParent: \n' + \
        'g: \n' + \

```

```
'rotationAngleXYPlane: \n' + \  
'bodyClassType: \n'
```

---

Listing 1.2: BaseBody.py

---

```
from __future__ import with_statement  
from OpenGL import *  
from OpenGL.GL import *  
from OpenGL.GLU import *  
from OpenGL.GLUT import *  
import string  
import sys  
from Utilities import Utilities  
import Image  
  
class BaseBody(object):  
    'This represents a celestial body, its attributes and functions/methods'  
  
    nextCoordInc          = 0  
    def __init__(self, bodyName, bodyRadius, bodyMass, bodyRedColor, \  
                 bodyGreenColor, bodyBlueColor, bodyTextureImage, \  
                 bodyParentBody, bodyAphelionFromParent, \  
                 bodyPerihelionFromParent ):  
        'Initialize a new celestial body, complete with radius, mass, colors, \  
        texture images, it\'s parent body, and the aphelion & perihelion from \  
        the parent object.'  
        #initialize everything we can from instantiation  
        self.name          = bodyName  
        self.radius        = float(bodyRadius) #meters  
        self.mass          = float(bodyMass) #kilograms  
        self.redColor      = float(bodyRedColor)  
        self.greenColor    = float(bodyGreenColor)  
        self.blueColor     = float(bodyBlueColor)  
        self.textureImage  = bodyTextureImage
```

```

self.parentBody          = bodyParentBody
self.aphelionFromParent  = float(bodyAphelionFromParent) #meters
self.perihelionFromParent = float(bodyPerihelionFromParent) #meters
self.actualRadius        = self.radius #meters
# initialize all positions and velocities to zero, as we will set these
# later; they will change frequently anyway, and I know of no better
# way. Mayhap this will change later, if I can think of a better way
# to handle the drawing of a body. Still need to think about how we
#are going to handle the texture mapping of the particular planet.

#keep the values that have already been drawn in arrays, so that we can
#produce a visual trail to see the trajectory of the planets more clear.
self.xTrailValues        = []
self.yTrailValues        = []
with open(Utilities.BODY.COORDINATES_DIR + "/" + self.name + ".txt") \
    as f:
    for line in f:
        coords = line.split()
        self.xTrailValues.append(float(coords[0]))
        self.yTrailValues.append(float(coords[1]))

#the x coordinate of the body
self.xPosition = self.xTrailValues[self.nextCoordInc]
# the y coordinate of the body
self.yPosition = self.yTrailValues[self.nextCoordInc]
print "xPosition on setup: " + str(self.xPosition)

def loadTextures(self):

    if self.textureImage != "none":

        image = Image.open(self.textureImage)

```

```

ix = image.size[0]
iy = image.size[1]
image = image.tostring("raw", "RGBX", 0, -1)

# Create Texture
self.textures = glGenTextures(3)
# 2d texture (x and y size)
glBindTexture(GL_TEXTURE_2D, int(self.textures[0]))

glPixelStorei(GL_UNPACK_ALIGNMENT,1)
glTexImage2D(GL_TEXTURE_2D, 0, 3, ix, iy, 0, GL_RGBA, \
             GL_UNSIGNED_BYTE, image)

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST)
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL)

def draw(self, scalingFactor, AUScalingFactor):
    'Draw the celestial body. It\'s coordinates must be set by the caller \
    before this function is called.'

    self.actualRadius = (self.radius + (self.radius * scalingFactor))

    self.drawingRadius = self.actualRadius * AUScalingFactor

    glPushMatrix()
    self.material(self.redColor, self.greenColor, self.blueColor, 1, .8, 70)
    glTranslated(self.xPosition* AUScalingFactor, self.yPosition* \
                AUScalingFactor, 0)

    # handle color here.

```

```

glColor3f(self.redColor, self.greenColor, self.blueColor)

# NOTE: if there is no color, then handle texture here.
if self.textureImage != "none":
    quadratic = gluNewQuadric()
    gluQuadricNormals(quadratic, GLUSMOOTH)
    gluQuadricTexture(quadratic, GLTRUE)
    glBindTexture(GL_TEXTURE_2D, int(self.textures[0]))
    gluSphere(quadratic, self.drawingRadius, 50, 50)
else:
    #draw the object with no texture.
    glutSolidSphere(self.drawingRadius, 50, 50)

glPopMatrix()

try:
    self.nextCoordInc = self.nextCoordInc + 1
    #the x coordinate of the body
    self.xPosition = self.xTrailValues[self.nextCoordInc]
    # the y coordinate of the body
    self.yPosition = self.yTrailValues[self.nextCoordInc]
except Error:
    print 'We have an error!'
    raise Error

def drawTrail(self, AUScalingFactor, numBodies):
    'Draw the trail that the celestial body leaves behind itself.'

    glPushMatrix()
    glBegin(GL_LINE_STRIP)
    glColor3f(self.redColor, self.greenColor, self.blueColor)
    for i in range(self.nextCoordInc):
        if i >= (self.nextCoordInc - 1000):
            glVertex2f(self.xTrailValues[i]* AUScalingFactor, \

```

```

        self.yTrailValues[i]* AUScalingFactor)

    glEnd()
    glPopMatrix()

#private
    def material(self, red, green, blue, alpha, spec, shiny):
        ambient_diffuse = [red, green, blue, alpha]
        specular = [spec, spec, spec, spec]
        shininess = [shiny]
        emission = [0, 0, 0, 1]

        glMaterialfv(GLFRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, ambient_diffuse)
        glMaterialfv(GLFRONT_AND_BACK, GL_SPECULAR, specular)
        glMaterialfv(GLFRONT_AND_BACK, GL_SHININESS, shininess)
        glMaterialfv(GLFRONT_AND_BACK, GL_EMISSION, emission)

    def getZViewingDistance(self):
        return self.radius * 150

```

---

*Listing 1.3: Body.py*

---

```

from BaseBody import BaseBody
from OpenGL import *
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *

class Body(BaseBody):
    'This represents a celestial body, its attributes and functions/methods'

    def __init__(self, bodyName, bodyRadius, bodyMass, bodyRedColor, \
                 bodyGreenColor, bodyBlueColor, bodyTextureImage, \
                 bodyParentBody, bodyAphelionFromParent, \

```

```

        bodyPerihelionFromParent):
    'Initialize a new celestial body, complete with radius, mass, colors, \
    texture images, it\'s parent body, and the aphelion & perihelion from \
    the parent object.'
    super(Body, self).__init__( bodyName, bodyRadius, bodyMass, \
                                bodyRedColor, bodyGreenColor, \
                                bodyBlueColor, bodyTextureImage, \
                                bodyParentBody, bodyAphelionFromParent, \
                                bodyPerihelionFromParent )

```

---

*Listing 1.4: BodyWrapper.py*

---

```

from __future__ import with_statement
from OpenGL import *
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
import string
import sys
import Image
from Utilities import Utilities
from config import Config

#For each new body that is derived from BaseBody, it must be imported here.
#That's pretty standard, but just in case.. had to include that.
from Body import Body
from SquareBody import SquareBody

class BodyWrapper(object):
    'This is the class that manages which type of body subclass to instantiate.'

    #the body type is intended to be private. I don't know how to force this
    #right now, but if you're the one extending this simulator, just don't use
    #it after you instantiate it elsewhere. It's used to supply the data for

```



*#the wrapper functions here. Nothing else.*

```
name           = None
radius        = None
mass          = None
redColor      = None
greenColor    = None
blueColor     = None
textureImage  = None
parentBody    = None
aphelionFromParent = None
perihelionFromParent = None
coordinatesFile = None
xPosition     = 0
yPosition     = 0

xTrailValues  = []
yTrailValues  = []
nextCoordInc  = 0
textures      = None
bodyClassType = None
getZViewingDistance = 0
```

```
def __init__(self, bodyName, bodyRadius, bodyMass, bodyRedColor, \
             bodyGreenColor, bodyBlueColor, bodyTextureImage, \
             bodyParentBody, bodyAphelionFromParent, \
             bodyPerihelionFromParent, bodyClassType):

if bodyClassType == Utilities.SQUAREBODY:
    self.body = SquareBody( bodyName, bodyRadius, bodyMass, \
                            bodyRedColor, bodyGreenColor, \
                            bodyBlueColor, bodyTextureImage, \
                            bodyParentBody, bodyAphelionFromParent, \
                            bodyPerihelionFromParent )
```

```

elif bodyClassType == Utilities.BODY:
    self.body = Body( bodyName, bodyRadius, bodyMass, bodyRedColor, \
                      bodyGreenColor, bodyBlueColor, bodyTextureImage, \
                      bodyParentBody, bodyAphelionFromParent, \
                      bodyPerihelionFromParent )

#if one wanted to add an additional body type, they'd have to
#instantiate it here. Additionally, if any extra methods were added to
#the new body type, you'd have to include a wrapper for that, while
#also performing any error handling that may arise if someone calls
#your method for your body while actually dealing with a body type other
#than your own.

else:
    self.body = BaseBody( bodyName, bodyRadius, bodyMass, bodyRedColor, \
                           bodyGreenColor, bodyBlueColor, \
                           bodyTextureImage, bodyParentBody, \
                           bodyAphelionFromParent, \
                           bodyPerihelionFromParent )

self.name           = self.body.name
self.radius         = self.body.radius
self.mass           = self.body.mass
self.redColor       = self.body.redColor
self.greenColor     = self.body.greenColor
self.blueColor      = self.body.blueColor
self.textureImage   = self.body.textureImage
self.parentBody     = self.body.parentBody
self.aphelionFromParent = self.body.aphelionFromParent
self.perihelionFromParent = self.body.perihelionFromParent
self.bodyClassType  = self.body.bodyClassType
self.getZViewingDistance = self.body.radius * 150
self.actualRadius   = self.body.actualRadius
self.xPosition      = self.body.xPosition
self.yPosition      = self.body.yPosition

```

```

def loadTextures(self):
    self.body.loadTextures()

def draw(self, scalingFactor, AUScalingFactor):
    self.body.draw(scalingFactor, AUScalingFactor)
    #these are the values that need updating after the base drawing
    #function.
    self.xPosition = self.body.xPosition
    self.yPosition = self.body.yPosition
    self.actualRadius = self.body.actualRadius
    self.drawingRadius = self.body.drawingRadius

def drawTrail(self, AUScalingFactor, numBodies):
    self.body.drawTrail(AUScalingFactor, numBodies)

```

---

*Listing 1.5: BodyManager.py*

---

```

from __future__ import with_statement
from OpenGL import *
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
import string
import sys
import Image
from Utilities import Utilities
from config import Config
from BodyWrapper import BodyWrapper
from math import *

class BodyManager(object):
    bodies = []
    scalingFactor = 0
    normalScaling = True

```

```

absoluteScaling = False

def AddBody(self, bodyName, bodyRadius, bodyMass, bodyRedColor, \
            bodyGreenColor, bodyBlueColor, bodyTextureImage, \
            bodyParentBody, bodyAphelionFromParent, \
            bodyPerihelionFromParent, bodyClassType):
    self.bodies.append( BodyWrapper(bodyName, bodyRadius, bodyMass, \
                                    bodyRedColor, bodyGreenColor, \
                                    bodyBlueColor, bodyTextureImage, \
                                    bodyParentBody, bodyAphelionFromParent, \
                                    bodyPerihelionFromParent, \
                                    bodyClassType ) )

#eventually we need to get to the part where bodies are manipulated through
#this class only. Therefore we won't be returning the bodies to the main
#program hopefully.

def GetBodies(self, directory):
    for filex in directory:
        if os.path.splitext(filex)[1] == '.cfg':
            f = file(Utilities.BODY_DEFINITION_DIR + '\\\ ' + filex)
            cfg = Config(f)
            self.AddBody( cfg.name, cfg.radius, cfg.mass, cfg.redColor, \
                          cfg.greenColor, cfg.blueColor, cfg.textureImage, \
                          cfg.parentBody, cfg.aphelionFromParent, \
                          cfg.perihelionFromParent, cfg.bodyClassType )

    self.scalingFactor = self.findScalingFactor()
    return self.bodies

def DrawBodies(self, AUScalingFactor):
    if self.normalScaling == True and self.absoluteScaling == False:
        theScalingFactor = 0
    elif self.absoluteScaling == True and self.normalScaling == False:
        theScalingFactor = self.scalingFactor

```

```

else:
    theScalingFactor = 0

for bodyx in self.bodies:
    bodyx.draw(theScalingFactor, AUScalingFactor)
    if bodyx.parentBody != "none":
        bodyx.drawTrail(AUScalingFactor, len(self.bodies))

def findScalingFactor(self):

    #find the shortest distance between any two bodies, take the radius of
    #the larger and smaller and let the algo run.

    #Now to find the shortest distance between two bodies.

    #set shortest distance to the maxint value of the machine, so that when
    #our first iteration goes around, it automatically gets set to something
    #lower within our body system.
    shortestDistance = sys.maxint
    i = 0
    j = 0
    for i in range(len(self.bodies)):
        for j in range(len(self.bodies)):
            if i < j:

                distance = sqrt( pow((self.bodies[j].xPosition - \
                                     self.bodies[i].xPosition),2) + \
                                pow((self.bodies[j].yPosition - \
                                     self.bodies[i].yPosition),2))

                if distance < shortestDistance:
                    shortestDistance = distance

                    if self.bodies[i].radius > self.bodies[j].radius:
                        radiusSmaller = self.bodies[j].radius
                        radiusLarger = self.bodies[i].radius

```

```

        else:
            radiusSmaller = self.bodies[i].radius
            radiusLarger = self.bodies[j].radius

newSmallRadius = radiusSmaller
newLargeRadius = radiusLarger
adjustedDistance = shortestDistance
distance = adjustedDistance

while adjustedDistance > (radiusSmaller * 30.):
    newSmallRadius = newSmallRadius + (radiusSmaller * .1)
    newLargeRadius = newLargeRadius + (radiusLarger * .1)
    adjustedDistance = distance - (newSmallRadius + newLargeRadius)

return newSmallRadius/radiusSmaller

```

---

*Listing 1.6: BaseCamera.py*

---

```

from __future__ import with_statement
import string
from OpenGL import *
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from Utilities import Utilities

class BaseCamera(object):
    'This represents a camera, which can be extended'

    def __init__(self, eyex, eyey, eyez, lookx, looky, lookz, windowx, windowy):
        'Initialize where we want the camera to look.'
        self.eyex = eyex
        self.eyey = eyey

```

```

self.eyeZ = eyez
self.lookX = lookx
self.lookY = looky
self.lookZ = lookz
self.windowX = windowx
self.windowY = windowy

#This is 1/AU where AU is the astronomical unit. (AU/meters)

def CameraViewport(self, angle, near, far):
    'Create the viewport definition of the camera'
    # Reset The Projection Matrix
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    # Calculate The Aspect Ratio Of The Window
    ratio = self.windowX/self.windowY

    gluPerspective(angle, ratio, near, far)
    glMatrixMode(GL_MODELVIEW)

def LookAt(self, body):
    self.eyeX = body.xPosition * Utilities.AU_SCALING_FACTOR
    self.eyeY = body.yPosition * Utilities.AU_SCALING_FACTOR
    self.eyeZ = body.getZViewingDistance * Utilities.AU_SCALING_FACTOR
    self.lookX = body.xPosition * Utilities.AU_SCALING_FACTOR
    self.lookY = body.yPosition * Utilities.AU_SCALING_FACTOR
    #this is where we decide what body to look at.

    gluLookAt(self.eyeX, self.eyeY, self.eyeZ, self.lookX, self.lookY, \
              self.lookZ, 0, 1, 0)

def keyboard(self, args, body):
    pass

```

```
def special(self, key, x, y):
    #print "we made it to base camera"
    pass
```

---

*Listing 1.7: BasicCamera.py*

---

```
from __future__ import with_statement
import string
from OpenGL import *
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from Utilities import Utilities
from BaseCamera import BaseCamera

class BasicCamera(BaseCamera):
    'This represents a camera, which can be extended'

    def __init__(self, eyex, eyey, eyez, lookx, looky, lookz, windowx, windowy):
        'Initialize where we want the camera to look.'

        super(BaseCamera, self).__init__( eyex, eyey, eyez, lookx, looky, \
                                           lookz, windowx, windowy)

        self.eyeXInc = 0
        self.eyeYInc = 0
        self.eyeZInc = 0

    def LookAt(self, body):
        self.eyeX = body.xPosition * Utilities.AU_SCALING_FACTOR
        self.eyeY = body.yPosition * Utilities.AU_SCALING_FACTOR
        self.eyeZ = body.getZViewingDistance * Utilities.AU_SCALING_FACTOR
        self.lookX = body.xPosition * Utilities.AU_SCALING_FACTOR
        self.lookY = body.yPosition * Utilities.AU_SCALING_FACTOR
```



```

self.eyeX = self.eyeX + self.eyeXInc
self.eyeY = self.eyeY + self.eyeYInc
if self.eyeZInc != 0:
    self.eyeZ = self.eyeZInc * Utilities.AUSCALINGFACTOR

#this is where we decide what body to look at.

gluLookAt(self.eyeX, self.eyeY, self.eyeZ, self.lookX, self.lookY, \
          self.lookZ, 0, 1, 0)

def AdjustCamera(self, body):
    if self.eyeZInc != 0:
        #print "Actual Radius: " + str(body.actualRadius)
        #print "eyeZInc: " + str(self.eyeZInc)
        if self.eyeZInc < body.actualRadius*2:
            self.eyeZInc = body.actualRadius * 2
            self.CameraViewport(130, .00000001, \
                                (152630000000 + self.eyeZInc)\
                                *Utilities.AUSCALINGFACTOR)

def keyboard(self, args, body):

    if args[0] == 'f':
        print "We're moving forward"
        if self.eyeZInc == 0:
            zDist = body.getZViewingDistance/2
            self.eyeZInc = zDist
        else:
            self.eyeZInc = self.eyeZInc/2
            if self.eyeZInc < body.actualRadius*2:
                self.eyeZInc = body.actualRadius*2
    #move camera backward
    elif args[0] == 'b':
        if self.eyeZInc == 0:
            zDist = body.getZViewingDistance*2

```

```

        self.eyeZInc = zDist
    else:
        self.eyeZInc = self.eyeZInc*2

        #We're moving backwards by a whole bunch, so we must compensate
        #for the clipping that would occur because the viewport wouldn't
        #be long enough to hold the body objects in it's viewport.
        print "EyeZInc: " + str(self.eyeZInc)
        self.CameraViewport(130, .00000001, \
                               (body.getZViewingDistance + self.eyeZInc)*\
                               Utilities.AU_SCALING.FACTOR)

#reset the view
    elif args[0] == 'r':
        self.eyeXInc = 0
        self.eyeYInc = 0
        self.eyeZInc = 0
    #absolute scaling
    else:
        return

def special(self, key, x, y):
    #move camera up
    if key == GLUT_KEY_UP:
        self.eyeYInc = self.eyeYInc + 200000000. * \
            Utilities.AU_SCALING.FACTOR
    #move camera down
    if key == GLUT_KEY_DOWN:
        self.eyeYInc = self.eyeYInc - 200000000. * \
            Utilities.AU_SCALING.FACTOR
    #move camera right
    if key == GLUT_KEY_RIGHT:
        self.eyeXInc = self.eyeXInc + 200000000. * \
            Utilities.AU_SCALING.FACTOR

```

```

    #move camera left
    if key == GLUT_KEY_LEFT:
        self.eyeXInc = self.eyeXInc - 200000000. * \
            Utilities.AU_SCALING_FACTOR
    else:
        return

```

---

*Listing 1.8: CameraWrapper.py*

---

```

from __future__ import with_statement
from OpenGL import *
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
import string
import sys
import Image
from Utilities import Utilities
from config import Config

#For each new camera that is derived from BaseCamera, it must be imported here.
#That's pretty standard, but just in case.. had to include that.
from BasicCamera import BasicCamera

class CameraWrapper(object):
    'This is the class that manages which type of camera subclass to \
    instantiate.'

    #the body type is intended to be private. I don't know how to force this
    #right now, but if you're the one extending this simulator, just don't use
    #it after you instantiate it elsewhere. It's used to supply the data for
    #the wrapper functions here. Nothing else.

    def __init__(self, cameraSubType, eyex, eyey, eyez, lookx, looky, lookz, \

```

```

        windowx, windowy):

self.cameraSubType = cameraSubType
if self.cameraSubType == Utilities.BASIC.CAMERA:
    self.camera = BasicCamera(eyex, eyeY, eyez, lookx, looky, lookz, \
                               windowx, windowy)

#if one wanted to add an additional camera type, they'd have to
#instantiate it here. Additionally, if any extra methods were added to
#the new camera type, you'd have to include a wrapper for that, while
#also performing any error handling that may arise if someone calls your
#method for your camera while actually dealing with a camera type other
#than your own.

else:
    self.camera = BaseCamera(eyex, eyeY, eyez, lookx, looky, lookz, \
                              windowx, windowy)

self.eyex = self.camera.eyex
self.eyey = self.camera.eyey
self.eyez = self.camera.eyez
self.lookX = self.camera.lookX
self.lookY = self.camera.lookY
self.lookZ = self.camera.lookZ
self.windowX = self.camera.windowX
self.windowY = self.camera.windowY

if self.cameraSubType == Utilities.BASIC.CAMERA:
    self.eyexInc = 0
    self.eyeyInc = 0
    self.eyezInc = 0

def CameraViewport(self, angle, near, far):
    self.camera.CameraViewport(angle, near, far)

def LookAt(self, body):

```

```

self.camera.LookAt(body)

def AdjustCamera(self, body):
    if self.cameraSubType == Utilities.BASIC.CAMERA:
        self.camera.AdjustCamera(body)

def keyboard(self, args, body):
    self.camera.keyboard(args, body)

def special(self, key, x, y):
    self.camera.special(key, x, y)

```

---

*Listing 1.9: CameraManager.py*

---

```

from __future__ import with_statement
from OpenGL import *
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from Utilities import Utilities
from CameraWrapper import CameraWrapper

class CameraManager(object):
    cameras = []

    def AddCamera(self, cameraSubType, eyex, eyey, eyez, lookx, looky, lookz, \
                 windowx, windowy):
        self.cameras.append( CameraWrapper(cameraSubType, eyex, eyey, eyez, \
                                           lookx, looky, lookz, windowx, \
                                           windowy) )

#Give the initial conditions for every camera here. If you need different
#initial conditions for different cameras then use the AddCamera method from
#the calculation script to instantiate it.

```

```

def GetCameras(self, eyex, eyey, eyez, lookx, looky, lookz, windowx, \
               windowy):
    #add as many cameras as you want right here. In the simulation script,
    #you can add keyboard functions that will switch between the different
    #cameras you instantiate if you like.
    self.AddCamera( Utilities.BASIC_CAMERA, eyex, eyey, eyez, lookx, looky, \
                   lookz, windowx, windowy )
    return self.cameras

```

---

*Listing 1.10: DynamicCelestialBodyPositionSimulation.py*

---

```

from __future__ import with_statement
from OpenGL import *
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
import string
import sys
import os
from config import Config
from BodyManager import BodyManager
from BodyWrapper import BodyWrapper
from Utilities import Utilities
import os
import time
from CameraManager import CameraManager

class Simulation(object):
    def __init__(self):
        #get the current working directory set by the main GUI
        thisdir = os.getcwd() + '\\\\' + Utilities.BODY_DEFINITION_DIR
        theListDir = os.listdir(thisdir)

```

```

self.bodyMgr = BodyManager()
self.theBodies = self.bodyMgr.GetBodies(theListDir)

#####

self.bodyIndex = 0
self.cameraIndex = 0

self.window_x=800
self.window_y=800
#This is 1/AU where AU is the astronomical unit.(1AU/149596000m)

self.eyex = 0
self.eyey = 0
self.eyez = self.theBodies[self.bodyIndex].getZViewingDistance * \
    Utilities.AU.SCALING_FACTOR #meters

#self.eyexInc = 0
#self.eyeyInc = 0
#self.eyezInc = 0

self.lookx = 0 #15210000000
self.looky = 0
self.lookz = 0

self.cameraMgr = CameraManager()
self.theCameras = self.cameraMgr.GetCameras( \
    self.eyex, self.eyey, self.eyez, self.lookx, self.looky, \
    self.lookz, self.window_x, self.window_y)

self.normalScaling = bool(True)
self.absoluteScaling = bool(False)
self.categoricalScaling = bool(False)
self.scalingFactor = 0

```

```

#The categorical radii of celestial bodies.
self.category1radius = 200.
self.category2radius = 20.
self.category3radius = 1.0
self.category4radius = .3

self.elapsedTime = 0
self.name = 'Sun Earth & Moon Animation'

def main(self):
    self.timeStart = time.time()

    print 'Main Start Time: ' + str(self.timeStart)

    glutInit(sys.argv)
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_ALPHA | GLUT_DEPTH)
    glutInitWindowSize(self.window_x, self.window_y)
    glutCreateWindow(self.name)

    glutDisplayFunc(self.display)
    glutIdleFunc(self.display)

    #glutReshapeFunc(ReSizeGLScene)
    glutKeyboardFunc(self.keyboard)
    glutSpecialFunc(self.special)

    self.InitGL(self.window_x, self.window_y)

```



```

glutMainLoop()

return

def InitGL(self , Width, Height):

    #initialize all the planets data.
    for bodyx in self.theBodies:
        bodyx.loadTextures()

    glEnable(GL_TEXTURE_2D)

    # We call this right after our OpenGL window is created.
    # This Will Clear The Background Color To Black
    glClearColor(0.,0.,0.,1.)
    # Enables Clearing Of The Depth Buffer
    glClearDepth(1.0)
    # The Type Of Depth Test To Do
    glDepthFunc(GL_LESS)
    glEnable(GL_CULL_FACE)
    glEnable(GL_LIGHTING)
    # Enables Depth Testing
    glEnable(GL_DEPTH_TEST)
    # Enables Smooth Color Shading
    glShadeModel(GL_SMOOTH)

    #initialize the camera viewport.
    self.theCameras[self.cameraIndex].CameraViewport(\
        130, 0.00000001, \
        (self.theBodies[self.bodyIndex].getZViewingDistance)*\
        Utilities.AU_SCALING_FACTOR)

def display(self):
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)

```

```

glLoadIdentity()
self.theCameras[self.cameraIndex].LookAt(self.theBodies[self.bodyIndex])

lightZeroPosition = [self.eyex, self.eyey, self.eyez, 1.]
glLightfv(GLLIGHT0, GLPOSITION, lightZeroPosition)
lightZeroColor = [10.,10.0,10.,1.0]
glLightfv(GLLIGHT0, GLDIFFUSE, lightZeroColor)
ambientIntensity0 = [152100000000., 152100000000., 152100000000., 1.0]
glLightfv(GLLIGHT0, GLAMBIENT, ambientIntensity0)
glLightf(GLLIGHT0, GLCONSTANT_ATTENUATION, 0.1)
glLightf(GLLIGHT0, GLLINEAR_ATTENUATION, 0.05)
glEnable(GLLIGHT0)

#make the bodies draw themselves - CONTROLLER!
try:
    self.bodyMgr.DrawBodies(Utilities.AU_SCALING_FACTOR)

    #when drawing bodies, make sure that the camera doesn't go inside
    #one of the bodies.
    self.theCameras[self.cameraIndex].AdjustCamera(\
        self.theBodies[self.bodyIndex])
    glutSwapBuffers()
return
except Error:
    elapsedTime = (time.time() - self.timeStart) / 60
    print 'Start Time is: ' + str(self.timeStart) + \
        ' Elapsed time is: ' + str(elapsedTime) + ' minutes'
    exit(0)

def idle(self):
    glutPostRedisplay()

def keyboard(self, *args):

```

```

self.theCameras[self.cameraIndex].keyboard(\
    args, self.theBodies[self.bodyIndex])

#absolute scaling
if args[0] == 'a':
    self.bodyMgr.normalScaling = bool(False)
    self.bodyMgr.absoluteScaling = bool(True)
#normal scaling
elif args[0] == 'n':
    self.bodyMgr.normalScaling = bool(True)
    self.bodyMgr.absoluteScaling = bool(False)
elif args[0] == 'e':
    exit(0)

#These two here need to be the exact same as the 'f' and 'b' keys above.
elif args[0] == '+':
    self.bodyIndex = (self.bodyIndex + 1) % len(self.theBodies)
    #self.eyezInc = 0
elif args[0] == '-':
    self.bodyIndex = (self.bodyIndex - 1) % len(self.theBodies)
    #self.eyezInc = 0
else:
    return
glutPostRedisplay()

def special(self, key, x, y):
    self.theCameras[self.cameraIndex].special(key, x, y)
    glutPostRedisplay()

#starts demo..
if __name__ == "__main__":
    #main()

```

---

```
function xyp=f1(t,Z);

    name = 1;
    xPosition = 2;
    yPosition = 3;
    xVelocity = 4;
    yVelocity = 5;
    radius = 6;
    mass = 7;
    redColor = 8;
    greenColor = 9;
    blueColor = 10;
    textureImage = 11;
    parentBody = 12;
    aphelionFromParent = 13;
    perihelionFromParent = 14;
    g = 15;
    rotationAngleXYPlane = 16;

//G=12

//get the size of this beast(all planetary/cometary data)
//number of columns equals the number of planets.
numPlanets = max(size(Z))/16;
numAttributes = 16;

//for each planet , calculate thier coordinates.
for i=1:numPlanets
    eqnSumX = 0;
    eqnSumY = 0;
    calcRadius = 0;
    //calculating each individual component of the gravitational equation.
```

```

for j=1:numPlanets
    if j <> i then
        calcRadius = ((Z(xPosition + (numAttributes*(i-1))) - ...
            Z(xPosition+(numAttributes*(j-1))))^2 + ...
            (Z(yPosition+(numAttributes*(i-1))) - ...
            Z(yPosition+(numAttributes*(j-1))))^2)^(1/2);
        //theradius = disp([calcRadius]);
        eqnSumX = eqnSumX + ((Z(mass+(numAttributes*(j-1))) * ...
            (Z(xPosition+(numAttributes*(i-1))) - ...
            Z(xPosition+(numAttributes*(j-1))) )) / abs(calcRadius)^3);
        eqnSumY = eqnSumY + ((Z(mass+(numAttributes*(j-1))) * ...
            (Z(yPosition+(numAttributes*(i-1))) - ...
            Z(yPosition+(numAttributes*(j-1))) )) / abs(calcRadius)^3);
    end
end

//after each component is summed up (which is a property of the
//gravitational equation, set the values back to what they
// should be:
xyp(xPosition+(numAttributes*(i-1))) = ...
    Z(xVelocity+(numAttributes*(i-1)));
xyp(xVelocity+(numAttributes*(i-1))) = ...
    -Z(g+(numAttributes*(i-1))) * eqnSumX;
xyp(yPosition+(numAttributes*(i-1))) = ...
    Z(yVelocity+(numAttributes*(i-1)));
xyp(yVelocity+(numAttributes*(i-1))) = ...
    -Z(g+(numAttributes*(i-1))) * eqnSumY;

//clear out the mass values, so the function doesn't add to them.
for k=1:numPlanets
    xyp(mass + (numAttributes*(k-1))) = 0;
    xyp(g + (numAttributes*(i-1))) = 0;
    xyp(rotationAngleXYPlane + (numAttributes*(i-1))) = 0;

```

```
end
end
endfunction

gravFunc = f1
```

---

*Listing 1.12: RungeKutta2D.sci*

---

```
//include the configuration file here, so we can actually access the values.
//Read in the values as a two dimensional array so we can keep everything
//as organized as we can.

bodyConfig = "C:\Documents and Settings\Administrator\My Documents\CSCI\" + ...
             "Thesis\ThesisProjectSource\BodyConfig.sci"
exec (bodyConfig);
projectFile = "C:\Documents and Settings\Administrator\My Documents\CSCI\" + ...
             "Thesis\ThesisProjectSource\ProjectFile.sci"
exec (projectFile);

name = 1;
xPosition = 2;
yPosition = 3;
xVelocity = 4;
yVelocity = 5;
radius = 6;
mass = 7;
redColor = 8;
greenColor = 9;
blueColor = 10;
textureImage = 11;
parentBody = 12;
aphelionFromParent = 13;
perihelionFromParent = 14;
```

```

g = 15;
rotationAngleXYPlane = 16;

t0 = 0;
//number of real time seconds in the simulation
tf = 3600 * 24 * 365.24 * getDuration();
//we want to divide the period up into small step-sizes , so the
//larger the denominator, the smaller our step-size will be.
h = 500 * getDuration();
t=0:h:tf;

//initial condition setup for x and y.
//first value is the distance between the Earth and
//the Sun at aphelion(furthest distance)..
bodies = GetAllBodyData();

size(bodies)

[m,numBodies] = size(bodies);
numAttributes = 16;

//construct the initial conditions.
initialPositions=[]

//get all bodies initial conditions prior to rotation ,
//so we can rotate children properly.
for i=1:numBodies
    initialPositions(1,i) = strtod(bodies(xPosition,i))
    initialPositions(2,i) = strtod(bodies(yPosition,i))
end

for i=1:numBodies

    //apply Given's rotations here.

```

```

//for the angle, we have to get the radians from
//the degrees input into the body config files.
bodysRotationAngle = ( ...
    strtod(bodies(rotationAngleXYPlane,i)) * %pi)/180 ;
GivensMatrix = [
    cos(bodysRotationAngle) -sin(bodysRotationAngle)
    sin(bodysRotationAngle) cos(bodysRotationAngle) ];

//need to find the parent's xy values, so we can
//subtract from the childs initial conditions.
//we need to do this so we make sure that we're
//rotating about the parent, and not anything else.
//example: in the moon's case, we want to rotate about
//the earth, not the sun.
originalXPositionChild = strtod(bodies(xPosition,i))
originalYPositionChild = strtod(bodies(yPosition,i))
originalXPositionParent = 0
originalYPositionParent = 0
for j=1:numBodies
    if bodies(parentBody,i) == bodies(name,j) then
        originalXPositionParent = initialPositions(1,j);
        originalYPositionParent = initialPositions(2,j);
    end
end
adjustedXPosition = originalXPositionChild - ...
    originalXPositionParent;
adjustedYPosition = originalYPositionChild - ...
    originalYPositionParent;

originalPosition = [ adjustedXPosition
    adjustedYPosition ];

originalVelocity = [ strtod(bodies(xVelocity,i))
    strtod(bodies(yVelocity,i)) ];

```



```

adjustedPosition = GivensMatrix * originalPosition
adjustedVelocity = GivensMatrix * originalVelocity

adjustedPosition = [ adjustedPosition(1,1) + ...
                    originalXPositionParent
                    adjustedPosition(2,1) + ...
                    originalYPositionParent ]

xPosDifference = adjustedPosition(1,1) - ...
                originalXPositionChild
yPosDifference = adjustedPosition(2,1) - ...
                originalYPositionChild

bodies(xPosition , i) = string(adjustedPosition(1,1))
bodies(yPosition , i) = string(adjustedPosition(2,1))
bodies(xVelocity , i) = string(adjustedVelocity(1,1))
bodies(yVelocity , i) = string(adjustedVelocity(2,1))

for k=1:numBodies
    if bodies(name,i) == bodies(parentBody,k) then
        //we move the smaller (k).

        //first , we have to move the child body the same
        //amount as the parent body moved, while not
        //rotating it .
        //Then, if the child body itself needs rotated ,
        //then it will be handled in the previous for
        //loop , not this one.

        posX = strtod(bodies(xPosition ,k));
        posY = strtod(bodies(yPosition ,k));

```

```

    adjPosx = posX + xPosDifference;
    adjPosy = posY + yPosDifference;

    bodies(xPosition,k) = string(adjPosx);
    bodies(yPosition,k) = string(adjPosy);
  end
end
end

//this is outside the initial for loop to make sure that
//all the rotated velocities have been
//calculated prior to adding parent velocities to the
//child.
for i=1:numBodies
//add the velocity of the parent body to the child body.
  for j=1:numBodies

    //ex if the moon's (i) parent body == the earth(j)
    if bodies(parentBody,i) == bodies(name,j) then

      velXI = strtod(bodies(xVelocity,i));
      velXJ = strtod(bodies(xVelocity,j));
      velYI = strtod(bodies(yVelocity,i));
      velYJ = strtod(bodies(yVelocity,j));

      theXSum = velXI + velXJ;
      theYSum = velYI + velYJ;

      bodies(xVelocity,i) = string(theXSum);
      bodies(yVelocity,i) = string(theYSum);
    end
  end
end
end

```

```

//initial condition matrix setup.
z0 = zeros(max(size(bodies)) * numBodies);
for i=1:numBodies
    z0(xPosition + (numAttributes*(i-1)) ) = strtod(bodies(xPosition , i));
    z0(yPosition + (numAttributes*(i-1))) = strtod(bodies(yPosition , i));
    z0(xVelocity + (numAttributes*(i-1))) = strtod(bodies(xVelocity , i));
    z0(yVelocity + (numAttributes*(i-1))) = strtod(bodies(yVelocity , i));
    z0(radius + (numAttributes*(i-1)))     = strtod(bodies(radius , i));
    z0(mass + (numAttributes*(i-1)))       = strtod(bodies(mass , i));
    z0(g + (numAttributes*(i-1)))          = strtod(bodies(g , i));
    z0(rotationAngleXYPlane + (numAttributes*(i-1))) = ...
        strtod(bodies(rotationAngleXYPlane , i));
end

z0

// in order to start the bodies at positions other than x=aphelion , y=0,
// we can perform a Givens rotation to specify an angled offset of that.
// We'll need to extract the x and y positions of each body, and multiply
// that by the Givens rotation matrix with the angle specified for the
// body, in order to get the correct rotated positions. We'd have to do
// this not only for the x and y positions , but also for the velocities
// of the respective bodies , and it would have to happen right here.vv

// the tic() toc() sequence is used for benchmarking purposes , to see how
// different things affect the calculation.
tic()
y = ode("rkf", z0, t0, t, gravFunc);
timeForCalc = toc()

fileCalc = getcwd() + '\Benchmarks\currentBenchmark.txt';
calc = mopen(fileCalc , 'a+');

```

```

mfprintf(calc, 'Time to calculate= %f \n', timeForCalc);
mclose(calc)

[m,n] = size(y);

for i=1:numBodies
    x1 = zeros(2,n);
    y1 = zeros(2,n);

    x1(1,:) = y(xPosition + (numAttributes*(i-1)),:);
    x1(2,:) = y(yPosition + (numAttributes*(i-1)),:);
    y1(1,:) = y(xVelocity + (numAttributes*(i-1)),:);
    y1(2,:) = y(yVelocity + (numAttributes*(i-1)),:);

    fileN = getcwd() + '\ ' + getBodyCoordinatesDir() + '\ ' + ...
        bodies(name,i) + '.txt';
    u = file('open', fileN, 'unknown')

    for j=1:n
        fprintf(u, '%f %f', x1(1,j), x1(2,j));
    end
    file('close', u)
end

exit

```

---

APPENDIX B  
INSTALLATION

The following programs/extensions will be needed in order to obtain a completely working copy of the code:

1. Tortoise SVN (1.5.3 Build 13783), or equivalent.

<http://tortoisesvn.net/downloads>

Google Code uses subversion for their repository, so any compatible client will do.

2. OpenGL files. These must be in your C:\WINDOWS\System(32) folder or equivalent for your particular operating system:

- opengl32.dll (<http://www.dll-files.com/dllindex/dll-files.shtml?opengl32>)
- glu32.dll (<http://www.dll-files.com/dllindex/dll-files.shtml?glu32>)
- glut32.dll (<http://www.dll-files.com/dllindex/dll-files.shtml?glut32>)

If they aren't already there, visit the corresponding links to download. These may vary depending on what type of system you're using. Now would be a good time to mention [www.opengl.org](http://www.opengl.org). If you have any questions regarding the OpenGL library, that's a great place to start. This link:

<http://www.xmission.com/~nate/glut.html>

has anything and everything you need to know about glut (else it will direct you someplace that does). The NeHe tutorials are always good too for beginners and gurus alike: <http://nehe.gamedev.net/>

3. Python v. 2.5.2

<http://www.python.org/download/>

Set environment variables to the python installation directory (C:\Python25), and also the scripts directory within that (C:\Python25\scripts).

All the extensions you'll need for Python:

- PyOpenGL v. 3.0.0.065

<http://sourceforge.net/projects/pyopengl/>

- SetupTools After you download PyOpenGL, read its documentation, and it'll take you exactly where you need to go for the SetupTools. Make sure your environment variables are set!

- WxPython v. 2.8

<http://www.wxpython.org/download.php>

- Config v. 0.3.7

[http://www.red-dove.com/python\\_config.html](http://www.red-dove.com/python_config.html) from Red-Dove.

- Python Imaging Library(PIL) v. 1.1.6

<http://www.pythonware.com/products/pil/>

#### 4. Wingware IDE v. 3.1.3-1 Personal

<http://www.wingware.com/downloads/wingide-personal>

#### 5. Scilab 5.0.1

<http://www.scilab.org/>

Set environment variable to where scilex.exe is located within the main Scilab program file folder.

Now that you've got everything setup, pick a place for your repository to reside within your file system, right click on the folder, and use Tortoise SVN to check out the project to that folder. The code can be found here:

<http://code.google.com/p/extensiblesimulationofplanetsandcomets/>, and on the accompanying CD. You will have to consult Google for your username and password credentials.

You will have to change a few files in order for everything to work properly, and these include a few Scilab files:

- BodyConfig.sci
- ESPCConfig.sci
- RungeKutta2D.sci

You have to change the paths at the beginning of the aforementioned files to the appropriate directory where this project will reside in order for them to be able to read and communicate with each other.



## REFERENCES

- [1] J. S. Bagla. Cosmological N-Body simulation: Techniques, Scope and Status. *Current Science*, 88:1088–1100, April 2005.
- [2] Jacques Crovisier and Thérèse Encrenaz. *Comet Science, The Study of Remnants from the Birth of the Solar System*. Cambridge University Press, 2000.
- [3] Encyclopædia Britannica. "Lagrange, Joseph-Louis, comte de l'Empire.". *Encyclopædia Britannica Online.*, October 2007.
- [4] Encyclopædia Britannica. "Lagrangian Point.". *Encyclopædia Britannica Online.*, October 2007.
- [5] Encyclopædia Britannica. "Poincaré, Henri.". *Encyclopædia Britannica Online.*, October 2007.
- [6] D. J. D. Earn and J. A. Sellwood. The Optimal N-Body Method for Stability Studies of Galaxies. *The Astrophysical Journal*, 451:533–+, October 1995.
- [7] A.P. French. *Newtonian Mechanics*. W W Norton and Company, Inc. New York, 1971.
- [8] Michael T. Heath. *Scientific Computing, An Introductory Survey*. McGraw-Hill, second edition, 2002.

- [9] Rhonda Martens. *Kepler's Philosophy and the New Astronomy*. 2000.
- [10] Robert J. Beichner Raymond A. Serway. *Physics For Scientists and Engineers with Modern Physics*. Saunders College Publishing, fifth edition, 2000.
- [11] D. C. Richardson, T. Quinn, J. Stadel, and G. Lake. Direct Large-Scale N-Body Simulations of Planetesimal Dynamics. *Icarus*, 143:45–59, January 2000.
- [12] Maurice D. Weir Ross L. Finney and Frank R. Giordano. *Thomas' Calculus*. Greg Tobin, 2001.
- [13] Dr. Keith Evan Schubert. *Keith On... Numerical Analysis*. Web, <http://csci.csusb.edu/schubert/pubs/KeithOnNumerical.pdf>.
- [14] J. G. Stadel. *Cosmological N-body simulations and their analysis*. PhD thesis, AA(University of Washington), 2001.
- [15] Reginald J. Stephenson. *Mechanics and Properties of Matter*. John Wiley and Sons, Inc., 1969.
- [16] Hubble Space Telescope Comet Team and NASA. Color hubble image of multiple comet impacts on jupiter, July 1994.
- [17] Q.-D. Wang. The global solution of the n-body problem. *Celestial Mechanics and Dynamical Astronomy*, 50:73–88, 1991.
- [18] Dr. Bernd Wirsing. Supercomputer simulations explain the formation of galaxies and quasars in the universe, June 2005.