Minor Project II
DYNAMIC EQUATIONS
Numerical Solution of
N-Body Gravitational Equations
EEEN30150

James Carron
15333196

May 11, 2018

Abstract

In this report a method of numerically solving for the future positions of the major bodies in our Solar System from initial conditions is presented as well as a basic method of visualising the solution.

1 Introduction

This project was to model and solve for the motion of the larger bodies in the solar system taking into account the gravitational forces between them, thus simulating the system for a period of at least three years. As suggested, the workload was broken into three sections, the modelling , the numerical solution and the presentation of the results.

While we all contributed to each section, one person was delegated the majority of the work for each one. For this project I worked with Luke Doyle-15412448 and Dominic Hoo-13443358.

Luke Doyle focused on the development of the model for the system. My role was focusing on the numerical solution of the resulting differential equations. Dominic Hoo focused on presenting the results in a human friendly format. As the majority of my contribution for this project was in the numerical solution I will discuss this in more detail.

2 Numerical Analysis

2.1 Assumptions

The first step in modelling this problem was to come up with educated assumptions to simplify the problem while minimising the error we introduce in doing so. With a problem as complex as modelling the motion of planets this was certainly necessary if we were to submit something reasonably complete before our graduation.

1. Firstly, we assumed that nothing outside of the solar system will influence the motion of the bodies in our system. This included other solar systems, galaxies and black holes. We also assumed that apart from the Sun and the 8 planets within our Solar System recognised by NASA, the rest of the bodies in the system can be neglected as they have negligible effect on our results.

2. One assumption we ruled out straight away was using the Sun as the origin for our model. This would have given rise to several issues. Firstly if we had assumed the Sun was simply not moving this would have been incorrect as Newton's third law explicitly states for every action there is an equal and opposite reaction. While its procession is minute relative to the other planets orbits it is non-negligible. Thus if we were to account for its movement we would have to shift all the other planets with each iteration.

We decided to use the Barycentre of the planets as the origin s mass and momentum is conserved (neglecting external forces).

3. We assumed that the accelerations can be modelled using Newton's basic law of gravitation and laws of motion. We decided against using Einstein's law of general relativity due to how complex it makes mapping time and because it makes the equations we have to solve a lot more complicated without adding a lot of accuracy over the time span we are considering.

4. We assumed that the values for the Astronomical Unit and Gravitational Constant. $149,597,870,700m = 1AU$ and $G = 6.67408 * 10^{-11} m^3 kg^{-1} s^{-2}$ respectively, provided by the NIST reference on constants, units and uncertainty are correct.

5. We assumed the initial conditions and body masses obtained from NASA's Jet Propulsion Lab (JPL) were accurate. This is a fair assumption as they are at the forefront of the space industry and have the necessary equipment to measure these values accurately.

6. We assumed that the masses of the respective objects stays constant. This is not

necessarily true for example mass is gained when a meteor impacts a planet and the sun is constantly expelling mass in the form of light energy (solar wind) as the result of nuclear fusion.

7. We assumed that the same solar wind's effect on the orbits of the planets is negligible. This is fair as the force exerted by solar wind is several orders of magnitude smaller then the gravitational forces.

2.2 Initial Analysis

To describe the motion of this system we must first derive a way of describing how the position of the objects within it change over time. Using the rule of superposition, which states that for all linear systems, the net response caused by two or more stimuli is the sum of the responses that would have been caused by each stimulus individually, we can treat the total gravitational force on a body as the sum of all individual gravitational forces in the system.

Thus for example we can obtain a description for the gravitational force on Earth as:

$$\vec{F}_{GEarth} = \vec{F}_{GSun} + \vec{F}_{GMercury} + \cdots + \vec{F}_{GNeptune}$$

We can also use this to consider the forces in each of the Cartesian directions $(\vec{i}, \vec{j}, \vec{k})$ separately and sum these up.

2.3 Modelling using Newton's Laws

Newton's second Law states:

$$\vec{F} = m\vec{a}$$

where F is the total external force applied to a body of mass m and a is the resulting acceleration of that body.

Newton's law of gravitation states:

$$\vec{F} = \frac{GMm}{|\vec{r}|^2}\hat{r}$$

where $\vec{F}$ is the total gravitation force between two objects 1 and 2 of mass M and m respectively with respect to the gravitational constant G and the distance between them $\vec{r}$

Substituting Newton's second law into Newton's law of gravitation we find:

$$\vec{a} = \frac{G * M}{|\vec{r}|^2}\hat{r}$$

We can then introduce the unit vector as:

$$\hat{u} = \frac{1}{|\vec{r_1}| - |\vec{r_2}|} * [x\ y\ z]$$

Thus to describe the interaction of two bodies we obtain:

$$\dot{\vec{r}} = \vec{v}, \quad \ddot{\vec{r}} = \dot{\vec{v}} = \vec{a}$$

$$\ddot{\vec{r}} = \frac{d^2}{dt^2}(\vec{r}) = \frac{G * M}{(|\vec{r_1}| - |\vec{r_2}|)^{\frac{3}{2}}} * (\vec{r_1} - \vec{r_2})$$

Thus following superposition we can describe the total acceleration components for a body using:

$$a_x = \sum_{i \neq j}^{n} G \frac{M_i}{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2)^{\frac{3}{2}}}(x_i - x_j)$$

$$a_y = \sum_{i \neq j}^{n} G \frac{M_i}{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2)^{\frac{3}{2}}}(y_i - y_j)$$

$$a_z = \sum_{i \neq j}^{n} G \frac{M_i}{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2)^{\frac{3}{2}}}(z_i - z_j)$$

2.4 Initial Conditions

The choice of exact date and time to use was somewhat arbitrary as any correct initial conditions should result in the same orbits provided we model the system correctly. We chose 01-01-18 00:00 as using the start of the year makes forecasting exact dates in the future easier and as the data we will be generating will contain values in the past we can check against these and as well as predict values in the future which is the useful outcome of this project. Mass values were obtained from JPL in kg.

Importing values found from NASAs JPL into our program presented a minor problem in that to input the 6 decimal point scientific notation values for all nine planets results in a total of over 640 digits where human error can occur.This process would also have to be repeated if we wanted to obtain values for different dates.

This lead me to seek out a Python API for the JPL Ephemeris data set to programmatically import the initial conditions and convert them from the km & $kmday^{-1}$ they are provided as, to AU & AUs^{-1} respectively. AUs^{-1} was chosen because using SI units (m & ms^{-1}) causes the values to overflow when computing the radial distance and using more bits to store the values would significantly slow computation.

	Position (AU):	**Vel ($^{AU}/_{s}$):**
Sun:		1.988544e+30 kg
x:	1.80e-03	-6.77e-11
y:	5.67e-03	5.27e-11
z:	2.33e-03	2.46e-11
Mercury:		3.302e+23 kg
x:	-3.86e-01	-6.13e-08
y:	-1.54e-02	-2.78e-07
z:	3.13e-02	-1.42e-07
Venus:		4.8685e+24 kg
x:	7.29e-02	2.31e-07
y:	-6.53e-01	2.55e-08
z:	-2.98e-01	-3.17e-09
Earth:		5.97219e+24 kg
x:	-1.73e-01	-1.99e-07
y:	8.93e-01	-3.32e-08
z:	3.87e-01	-1.44e-08
Mars:		6.4185e+23 kg
x:	-1.58e+00	4.46e-08
y:	-3.63e-01	-1.30e-07
z:	-1.24e-01	-6.08e-08
Jupiter:		1.89813e+27 kg
x:	-4.26e+00	5.31e-08
y:	-3.13e+00	-5.87e-08
z:	-1.24e+00	-2.65e-08
Saturn:		5.68319e+26 kg
x:	4.79e-02	6.10e-08
y:	-9.30e+00	1.06e-09
z:	-3.84e+00	-2.19e-09
Uranus:		8.68103e+25 kg
x:	1.77e+01	-2.11e-08
y:	8.39e+00	3.51e-08
z:	3.43e+00	1.57e-08
Neptune:		1.0241e+26 kg
x:	2.87e+01	1.02e-08
y:	-7.69e+00	3.25e-08
z:	-3.86e+00	1.31e-08

Table 1: Initial Conditions

3 Numerical Solution

3.1 Initial Analysis

Now we have the differential equations we can use to model the system we have to decide on a method of solving them. A good numerical method is required to accurately and quickly perform an orbit simulation. Simple methods like the Euler method need a very small time step, and thus a large amount of computing time, to remain stable. Therefore, a more elaborate method is necessary to increase accuracy and to reduce calculation time. A fourth order Runge-Kutta method is a good compromise, as it is stable at large time steps, accurate and relatively fast.

Since the problem of orbit simulation involves a three-dimensional, second order differential equation we will have to modify the standard Runge-Kutta method to suit.

3.2 Setup

The problem arises in that in solving for the velocity the acceleration is needed and as this is dependant on the position of the body we have to update this alternately with a second Runge-Kutta for the position. As the equation is three dimensional we will have to solve for the x, y & z components simultaneously.

Thus position vector $\vec{r}$, velocity vector $\vec{v}$ and acceleration vector $\vec{a}$ of the body currently being solved for can be defined as:

$$\vec{r} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \quad \vec{v} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{pmatrix} \quad \vec{a} = \begin{pmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{pmatrix}$$

Thus our modified Runge-Kutta method can be described in order of evaluation as:

$$\begin{aligned}
\vec{k_1}_{v_{i+1}} &= \vec{a}\left(\vec{r_i}\right), \\
\vec{k_1}_{p_{i+1}} &= \vec{v_i} \\
\vec{k_2}_{v_{i+1}} &= \vec{a}\left(\vec{r_i} + \vec{k_1}_{p_{i+1}} h/2\right), \\
\vec{k_2}_{p_{i+1}} &= \vec{v_i} + \vec{k_1}_{v_{i+1}} h/2 \\
\vec{k_3}_{v_{i+1}} &= \vec{a}\left(\vec{r_i} + \vec{k_2}_{p_{i+1}} h/2\right), \\
\vec{k_3}_{p_{i+1}} &= \vec{v_i} + \vec{k_2}_{v_{i+1}} h/2 \\
\vec{k_4}_{v_{i+1}} &= \vec{a}\left(\vec{r_i} + \vec{k_3}_{p_{i+1}} h\right), \\
\vec{k_4}_{p_{i+1}} &= \vec{v_i} + \vec{k_3}_{v_{i+1}} h
\end{aligned}$$

When all intermediate slopes are calculated, the velocity and position vector at the next time step can be determined:

$$\vec{v}_{i+1} = \vec{v_i} + \frac{h}{6}\left(\vec{k_1}_{v_{i+1}} + 2\vec{k_2}_{v_{i+1}} + 2\vec{k_3}_{v_{i+1}} + \vec{k_4}_{v_{i+1}}\right)$$

$$\vec{r}_{i+1} = \vec{r_i} + \frac{h}{6}\left(\vec{k_1}_{r_{i+1}} + 2\vec{k_2}_{r_{i+1}} + 2\vec{k_3}_{r_{i+1}} + \vec{k_4}_{r_{i+1}}\right)$$

The obtained velocity and position can then be used as the initial conditions for the next time step. Thus, it is possible to simulate the future state of the system. However it must be noted that we are compounding error and over a long period of time the system will eventually become unstable.

4 Computation

To solve this problem and implement the Runge-Kutta algorithm we have developed I will be using the Python Programming language for its modern syntax, readability and general application.

Debug and exception handling has been removed from the code shown for the brevitys sake.

4.1 Main Function

This is the main part of the code which calls all the background functions to calculate the values. I will elaborate more on the details of each function later.

Firstly I initialise the time constants in seconds for human readability.

```
EARTH_MIN = 60.0 #60s in a min
EARTH_HR = 60*EARTH_MIN #60mins in a hr
EARTH_DAY = 24*EARTH_HR #24hrs in a day
EARTH_WK = 7*EARTH_DAY #7 days in a week
EARTH_YEAR = 365*EARTH_DAY #365 days in a year
```

Figure 1: Time Constants

Then I initialise each Body and store them all in an array for easy referencing of the whole system. Each body is initialised by passing the name of the body in lowercase and its mass, which allows it's remaining initial conditions to be called from the JPL Ephemeris database.

```
Sun     = Body("sun"    , 1.988544*(10**30))
Mercury = Body("mercury", 3.302   *(10**23))
Venus   = Body("venus"  , 48.685  *(10**23))
Earth   = Body("earth"  , 5.97219 *(10**24))
Mars    = Body("mars"   , 6.4185  *(10**23))
Jupiter = Body("jupiter", 1898.13 *(10**24))
Saturn  = Body("saturn" , 5.68319 *(10**26))
Uranus  = Body("uranus" , 86.8103 *(10**24))
Neptune = Body("neptune", 102.41  *(10**24))

SolarSys = (Sun, Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune)
```

Figure 2: Initialising the system

Then I set the options for using the Runge-Kutta N-Body Gravitational solver and pass them to the GravNBodyRK4 function. This returns the positions of the bodies at every report interval for the duration you specify using the timestep you provide. While this is processing a progress bar is printed to screen.

```
duration = 3*EARTH_YEAR
timestep = EARTH_HR
output_interval = 6*EARTH_HR

All_Pos_Vals = GravNBodyRk4(SolarSys, duration, timestep, output_interval)
```

Figure 3: Running the Runge-Kutta

The data can then be saved out using a Numpy function which stores it in a compressed file which can be imported into other programs. The details of the options used are saved in the filename for identification purposes. The file could be saved out into a standard CSV format by passing an extra function to the numpy.save function however it was convenient to use the numpy format for the purposes of this project as we were able to import the position info into Blender using its Python interface.

```
## SAVING FILE ##

# use the current time within the filename
time_now = strftime("%H:%M:%S", gmtime())
output_filename =
"MP2-d:{:.2f}yrs-h:{:.0f}s-i:{:.0f}-".format(duration/EARTH_YEAR, timestep,
output_interval) + time_now

# save out the file
np.save("Simulation_Output/Vals/" + output_filename, All_Pos_Vals)
print("Filename: {} \t- contains {} positions".format(output_filename,
len(All_Pos_Vals[0])))
```

Figure 4: Saving out the position values

4.2 Classes

To improve the readability and ease of use of my code I implemented two object classes. The Body class which can be used to represent a Planetary Body and it's properties and a Point class which can be used to represent any 3-Dimensional property.

A body is initialised by at the minimum passing it's name in lowercase and mass to the function. These are saved to internal properties of the body and the name is used by the JPL_ephemeris method I have wrote to get the initial conditions. Two optional arguments allow you to specify the Julian Date

to get the initial conditions from and a DEBUG parameter that prints out the values obtained from JPL. The default Julian Date is set to 01-01-18 00:00.

```
class Body:
    # a class for summarising the properties for a body in the system
    def __init__(self, body_name, body_mass, JulianDate = 2458119.5,
    DEBUG=False): #id_num,

        self.mass = body_mass
        self.name = body_name #used for jpl ephemeris
        self.JPL_ephemeris(JulianDate) #pull body position and vel from JPL

        if DEBUG:
            print("{}: \tMass: {} kg".format(self.name.capitalize(),self.mass))
            print("Pos_x: {: .4e} AU\tPos_y: {: .4e} AU\tPos_z: {: .2e}
            AU".format(self.pos.x, self.pos.y, self.pos.z ))
            print("Vel_x: {: .4e} AU\\s\tVel_y: {: .4e} AU\\s\tVel_z: {: .2e}
            AU\\s \n".format(self.vel.x, self.vel.y, self.vel.z ))
```

Figure 5: Body Class

The point class is a very simple implementation of a 3D property. It is used for properties such as the positions, velocities, accelerations and K-coefficients. In hindsight I should have implemented this using a NumPy matrix as it would have been better solution with the only downside being that you couldn't then reference the x, y, z values by name.

```
class Point:
    # a class for describing a value in 3D [x, y, z]

    def __init__(self, x,y,z):
        self.x = x
        self.y = y
        self.z = z

    #### OVERLOADING OPERATORS ####
    # I probably should change this to use Numpy arrays instead as they have
    all the same functionality and more
    def __add__(self,other):
        x = self.x + other.x
        y = self.y + other.y
        z = self.z + other.z
        return Point(x,y,z)

    def __mul__(self,other):
        x = self.x * other
        y = self.y * other
        z = self.z * other
        return Point(x,y,z)

    def __div__(self, other):
        x = self.x / other
        y = self.y / other
        z = self.z / other
        return Point(x,y,z)

    def __str__(self):
        return "({: .4e},{: .4e},{: .4e})".format(self.x,self.y,self.z)
```

Figure 6: Point Class

4.3 Runge-Kutta Algorithm

Here the fourth order Runge-Kutta algorithm we developed in the previous section is applied to the system we pass to it within Bodies. At each time step within the duration of the simulation each body is evaluated. The initial conditions are called from the bodies position and velocity history properties last entry. The K coefficients are then calculated in 3 dimensions simultaneously.

Once all the K coefficents have been calculated and the next value for the position and velocity is calculated. These new values are stored in the respective bodies position and velocity history respectively. Once the next position for each body in the system has been calculated it is applied and the next time step is calculated.

Once the whole duration has been calculated the system is passed to a function that converts the position histories into a array of arrays of the form $[\,[[x_{sun_1}, y_{sun_1}, z_{sun_1}], [x_{sun_2}, y_{sun_2} \ldots z_{uranus_n}]]\,]$ where the index of the position histories is the same as the index of the bodies passed into the function. This function also removes values that do not match multiples of the output interval that is passed to the function.

This I understand to be a sub optimal solution that uses more memory to store all the values during calculation compared to discarding the ones we will not be saving out as it runs.

```
def GravNBodyRk4(Bodies, duration, h, output_t):
    """Takes a system of bodies and uses 4th order runge kutta and
    newtons gravitational laws to estimate the position and
    velocities at the next timestep"""

    number_steps = int(duration//h)

    #iterate through all timesteps
    for i in range(number_steps):
        progressbar(i,number_steps-1)

        #iterate through all bodies
        for Body in Bodies:

            #use the previous timesteps values for the new intial conditions
            v0 = Body.vel_hist[-1]
            p0 = Body.pos_hist[-1]
```

```
        #Initialise empty values
        k1_v = k2_v = k3_v = k4_v = Point(0,0,0)
        k1_p = k2_p = k3_p = k4_p = Point(0,0,0)

        #Calculate K coefficients
        k1_v = NBody_grav_accel(Bodies, Body, p0)
        k1_p = v0

        k2_v = NBody_grav_accel(Bodies, Body, p0 + k1_p*(h/2.0))
        k2_p = v0 + k1_v*(h/2.0)

        k3_v = NBody_grav_accel(Bodies, Body, p0 + k2_p*(h/2.0))
        k3_p = v0 + k2_v*(h/2.0)

        k4_v = NBody_grav_accel(Bodies, Body, p0 + k2_p*(h/2.0))
        k4_p = v0 + k1_v*h

        #Calculate the new values
        v1 = v0 + (k1_v + k2_v*2 + k3_v*3 + k4_v)*(h/6.0)
        p1 = p0 + (k1_p + k2_p*2 + k3_p*3 + k4_p)*(h/6.0)

        Body.vel_next = v1
        Body.pos_next = p1

    for Body in Bodies: #update the values once all the new values have be calculated
        Body.vel_hist.append(Body.vel_next)
        Body.pos_hist.append(Body.pos_next)

output_interval = report_interval/h

All_Pos_Vals = pntarr_to_cartesian(Bodies, output_interval)

return All_Pos_Vals
```

Figure 7: Runge Kutta Function

4.3.1 N-Body Gravitational Acceleration Function

This function is called when each K-value for the velocity is calculated. It does the summation of all the accelerations due to the bodies in the system and returns this. It uses a tmp variable to reduce repeated calculations.

```
# returns the total gravitation acceleration on a body due to all other bodies
in a system
def NBody_grav_accel(Bodies, target_Body, target_pos):

    AU = 149597900000
    G_const = 6.67408e-11 / (AU**3) #m3 kg-1 s-2

    total_acc = Point(0,0,0) #initialise value to 0

    for Body in Bodies: #calc accel due to all other bodies using superposition

        if Body == target_Body: #don't calculate grav accel wrt self
            continue

        #radial distance between the two bodies
        r = sqrt((Body.pos.x - target_pos.x)**2 + (Body.pos.y -
        target_pos.y)**2 + (Body.pos.z- target_pos.z)**2)
        tmp = G_const * Body.mass / (r**3) #store temporary value to reduce
        repeated calculations

        total_acc.x += tmp * (Body.pos.x - target_pos.x)
        total_acc.y += tmp * (Body.pos.y - target_pos.y)
        total_acc.z += tmp * (Body.pos.z - target_pos.z)

    return total_acc
```

Figure 8: Initialising the system

5 Visualising Data

Before I proceeded with further analysis I needed to check we were getting sensical data out. Hence I plotted the positions of the planets in the xy plane which gave me:

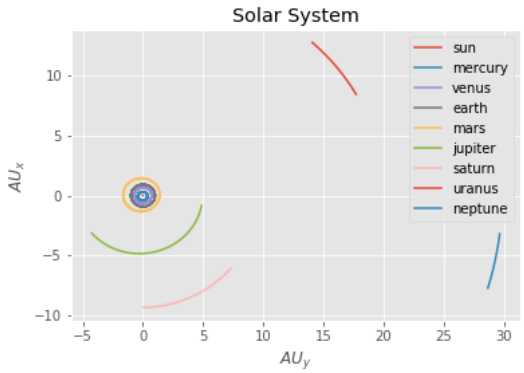

Figure 9: Basic Orbit Visualisation

Here you can see the planets positions are calculating correctly.

Looking closer at the orbits of Earth and Mercury reveal their periods are correctly being approximated to once per year and just over four times per year respectively.

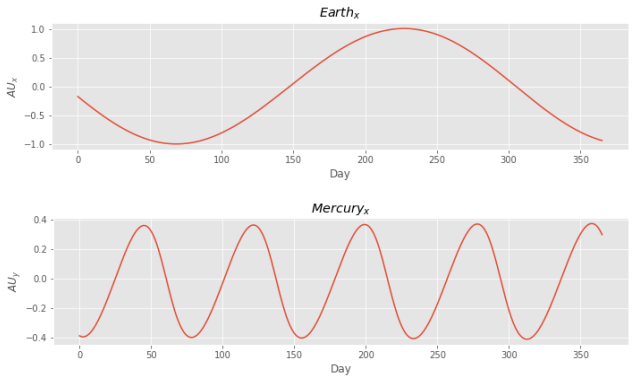

Figure 10: Period Visualisation

6 Time Step Analysis

To choose the appropriate time-step with which to evaluate the longer simulations we firstly want to look at the impact on shorter simulations. This allows us to look at the trade off between accuracy and computational expense. The position error of each body was calculated using the formula:

$$\frac{\sqrt{(x_s - x_a)^2 + (y_s - y_a)^2 + (z_s - z_a)^2}}{\sqrt{{x_a}^2 + {y_a}^2 + {z_a}^2}}$$

where the actual position is denoted $\vec{p}_a = [x_a, y_a, z_a]$ and our simulated position is denoted $\vec{p}_s = [x_s, y_s, z_s]$

Using the JPL Ephemeris data we can pull what the exact positions should be at different points in time according to NASA. So far as our accuracy is concerned if we can emulate their results that would be satisfactory.

Looking at our results we can see that longer time steps are less accurate, this is due to the fact that the force experienced by a body due to another body is proportional to the square of the distance between them. This makes the acceleration very sensitive to close encounters and approximating these often means we underestimate the acceleration at that time step. This is evident in the position of Mercury. The compounding of the error is also evident especially looking at Venus.

Time step (hrs)	**Avg Error after 1 Yr**	**Compute Time (s)**
0.25	0.33 %	119.84
1.00	0.37 %	31.10
6.00	0.56 %	5.29
24.00	0.55 %	1.24

Table 2: Time step tradeoff comparison

The time taken to compute the solutions is the next measure I will evaluate. This is directly correlated with the time-step used and the processing power of the computer it is ran on. Therefore with a supercomputer the results would be different.

Following from this Analysis I decided to use a 1hr time-step as it's a good trade off between the simulations accuracy and the time taken to compute the solution.

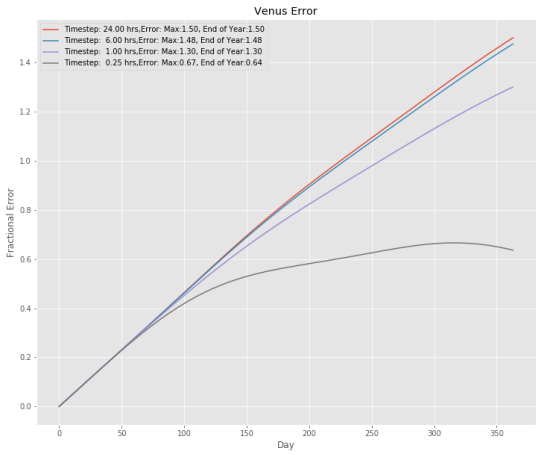

Figure 11: Venus Position Error

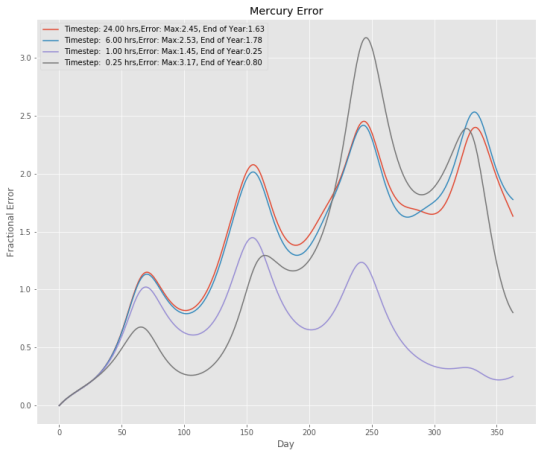

Figure 12: Mercury Position Error

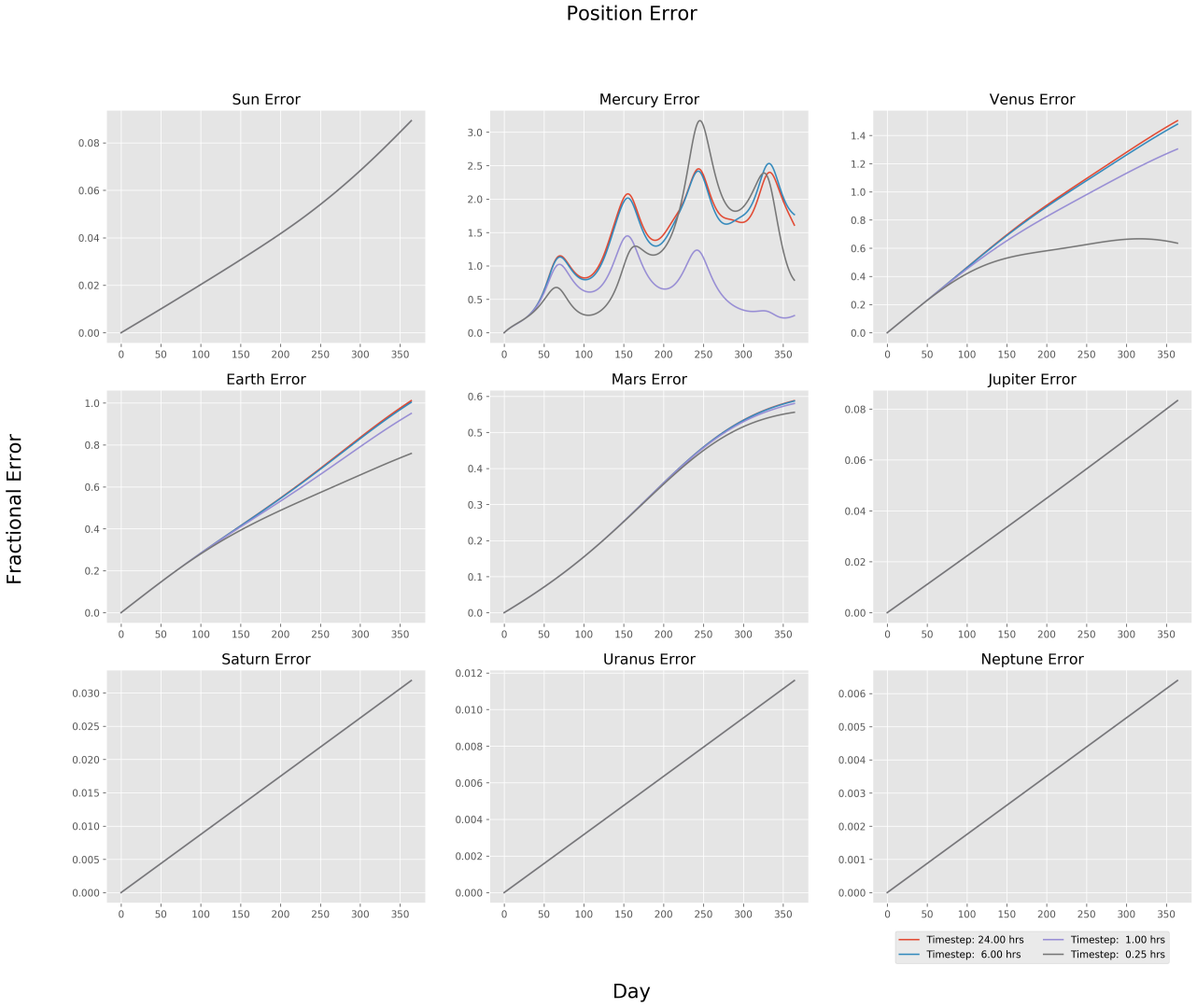

Figure 13: Positional Error

7 Visualisation

Good Visualisation is one of the core aspects of scientific learning. It allows experts to see their results with greater clarity as well as individuals with little scientific training a chance to understand and appreciate science & technology.

This part of the project was predominantly done by Dominic Hoo. I will go over an outline of the work he done. For the rendering portion of this project we will be using Blender for it's high quality path tracing render engine and its Python interface.

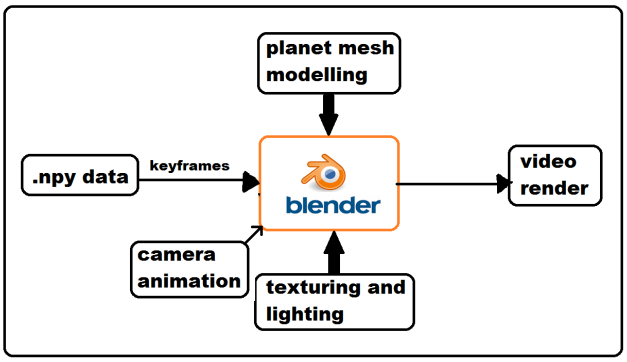

Figure 14: Blender Dataflow

While it is possible to call every user interface command from the Python interface this was not needed and cumbersome. Therefore I will not list the exact commands used and instead go over the steps taken.

Representing a scene billions of kilometers with the bodies so relatively small becomes a balancing act between accuracy, aesthetics and visibility. Therefore the distances were scaled down to make the visualisation more interesting.

7.1 Movement

Initially we ensured it was possible to programmatically move spheres. We achieved this using random data. Later I provided Dominic with positions along perfectly circular orbits which he could use to develop the visualisation in parallel to me developing the numerical solution. These positions were read in from Numpy files as single arrays and extracted into individual body coordinate sets. These were then used to set position keyframes. Interpolation between these points was disabled to ensure only simulated data was presented.

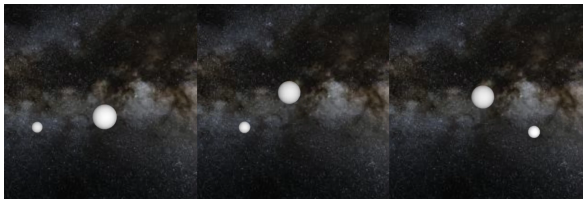

Figure 15: Initial animation test frames

7.2 Body Appearances

To make the simple sphere's we were using better represent the bodies they were portraying, images were mapped onto the sphere's. 8K UV maps were used,mapping a rectangular image onto a sphere would result in differing resolutions at different longitudes.

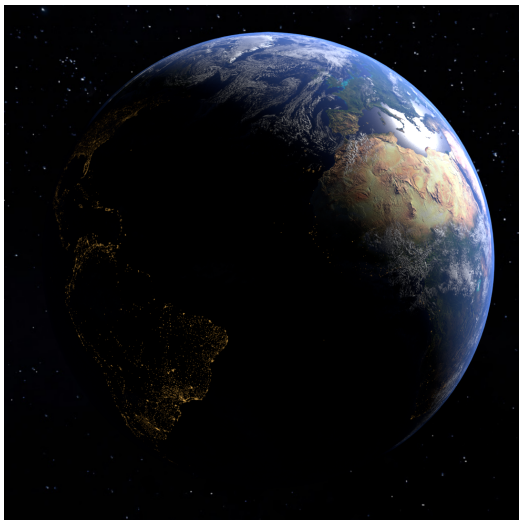

Figure 16: Final Representation of Earth

Texture maps were also used to mimicking terrain on each planet. To make the dark side of the Earth body more realistic we added lights to mimic the city street lights. These were not animated to turn off during the day as they are massively outshone when they are on the bright side.

As the sun cant be seen with any detail with the human eye, a NASA image capture of the Sun with filters applied was used.

7.3 Camera Movement

To add cinematic effect to the scene and draw the viewers attention to particular details of the simulation camera movements were introduced focusing on different parts. This allowed closeups of each of the planets as well as a better visualisation of how they move around each other.

7.4 Background

To mimic the galaxies around the bodies we are simulating we introduced a 8K sky-box background image of the Milky Way was used. As it is an image not a body no reflections or shadows are cast.

Figure 17: MilkyWay image

7.5 Lighting

To light the scene realistically a point light source was placed in the centre of the sun to mimic the light it emits. To prevent the sun's surface blocking the emission it was set as a transparent surface.

To render high quality visuals a lighting algorithm called Path Tracing was used. In contrast to Gouraud or Blinn-Phong, Path Tracing simulates the properties of light ray travel most accurately. This enables each image rendered to look photo-realistic, however it must be noted that it is not comparable to professional renderers like Keyshot, Cinema4d, Rhino3d, V-Ray and Pixar Renderman which are used by VFX studios in million dollar budget films.

7.6 Results

As the rendered video demonstrates this visualisation makes it far easier to understand the huge quantities of data that underlie the relative movement of the larger objects of our Solar System. The video brings the data into a realm of understanding that anyone can access.

The final quality is above average when compared to MATLAB output with the realistic lighting and shadows cast upon close representations of the large bodies in our Solar System. The final video also contains multiple fly by's of the planets and overview shots of the whole system and it's movement.

8 Conclusion

Concluding this project the author is satisfied he has presented methods sufficient for solving the three dimensional second order differential equations that model the movement of the large bodies of the Solar System. This was achieved and visualised to a high quality for a period of three years with a estimated error of below 2% at the end of the simulation.

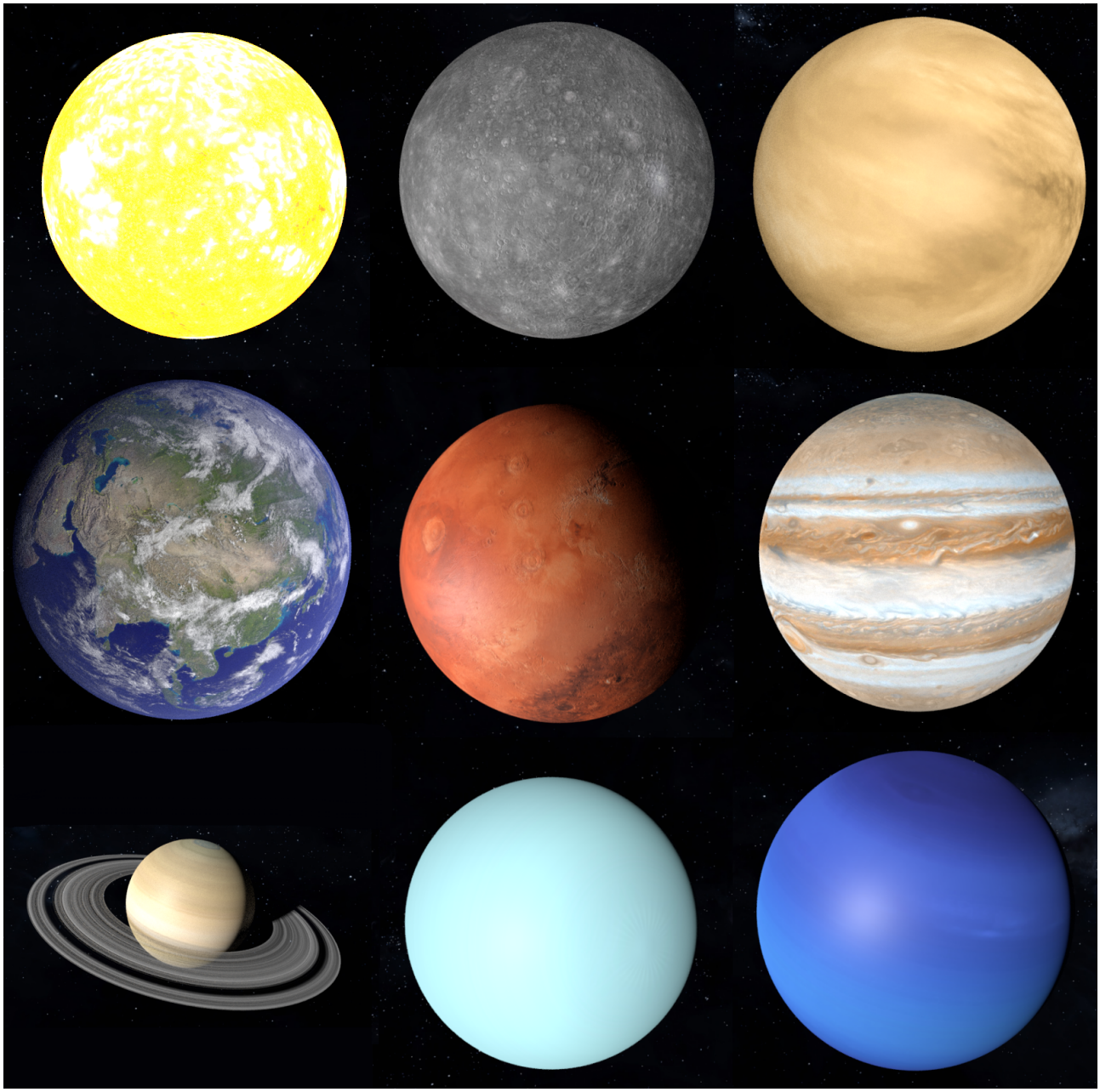

Figure 18: All Body Appearances

9 Other Code

```
def JPL_ephemeris(self, JulianDate = 2458119.5):

    """ Takes the name of the body and queries the JPL ephemeris data.
    Returns the position and velocity data at a given JulianDate in m/s

    Requires:
        import de421 #DE421 (February 2008) – 27 MB covering years 1900
        through 2050
        from jplephem import Ephemeris
        eph = Ephemeris(de421)

    See here for more info: https://pypi.org/project/jplephem/1.2/

    Default Julian Date: 01-01-2018"""

    """positions are returned in kilometers along the three axes of the
    ICRF
    (a more precise reference frame than J2000 but oriented in the same
    direction)
    Velocities are returned as kilometers per day."""
    km_to_AU = 1000.0/149597900000
    km_to_m = 1000 # number of ms in a km
    day_to_s = 1.0/(60*60*24)

    km_pday_to_m_ps = km_to_m * day_to_s
    km_pday_to_AU_ps = km_to_AU * day_to_s

    #Earth does not have its own entry, instead it is calculated relatively
    if self.name == "earth":
        barycenter_pos, vel = eph.position_and_velocity("earthmoon",
        JulianDate)
        moonvector = eph.position('moon', JulianDate)
        pos = barycenter_pos - moonvector * eph.earth_share

    else:
        pos, vel = eph.position_and_velocity(self.name, JulianDate)

    #casts strings to floats and converts km to AU and km/day to AU/s
    respectively
    pos = [float(val)*km_to_AU for val in pos]
    vel = [float(val)*km_pday_to_AU_ps for val in vel]

    #unpack and set vectors
    self.pos = Point(*pos)
    self.vel = Point(*vel)

    #save initial condition for planet
    self.pos_hist = [self.pos]
    self.vel_hist = [self.vel]
```

Figure 19: Initialising the system