Chaotic Tumbling of Hyperion

Jordan Walsh - 120387836

Individual Programming Project

PY2105 - Computational Physics

University College Cork

Department of Physics

November/December 2021

Chaotic Tumbling of Hyperion

Abstract

This report investigates the chaotic tumbling of Saturn's moon 'Hyperion'. The investigation took place over a number of weeks and utilised computational techniques (Such as Euler-Cromer) and the derivation of planetary motion to simulate motion similar (but not exact) to that of Hyperion. The report also details the development cycle of programs developed in C++ used to visualise and later analyse the motion under various initial conditions. As a stretch goal of this report, a rough estimate of the Lyapunov exponent for various orbits was calculated to further verify chaos where applicable.

Introduction

Many individuals believe that a solar system is a uniformly ordered system of motion, with planets and moons of perfectly spherical shape orbiting in physical attraction to one another. For the most part they would be right. However, there a few examples where this is not the case. One of which, located right within our own solar system, is one of Saturn's 82 moons, Hyperion. Hyperion displays rotational behaviour that is far from regular. Its motion, in fact, is described as chaotic.

Hyperion is extremely small in comparison to other moons, its mass being 3×10^{-8} of Saturn's mass. This is much smaller than even our moon, who's mass is about 1% of the Earth's. The reason for Hyperion's different and locally unique behaviour is its very unusual shape and highly elliptical orbit. Most moons are approximately spherical. Hyperion, however, is shaped a lot more like an asymmetrical egg. In the second edition of J. Giordano's academic text 'Computational Physics', it is described as having qualities similar to a dumbbell; "it is roughly like an orbiting dumbbell, and anyone who has ever thrown a dumbbell, or something with a similar shape, knows that they can spin in a very erratic manner. Indeed, this is precisely what Hyperion seems to do."[1] – Hyperion tumbles chaotically. Careful astronomical observations have proved this to be the case. This actually led to an issue on the *Cassini-Huygens space-research mission*, the probe of which is more commonly known as *Cassini*. This was a large scale collaboration between NASA, the ESA and ASI that failed to reliably schedule the Cassini probe to pass by unexplored regions due to Hyperion's chaotic and unpredictable nature.

Fig 1.1 - A 3D model of Hyperion, a moon of Saturn. Credit: NASA Visualization Technology Applications and Development (VTAD) [2]

Our goal for this report will not be to perform an exact realistic simulation of Hyperion's motion. This would be a feat given the academic level at which this topic is being investigated at. Our objective instead will be to show that the motion of such an irregularly shaped moon can be chaotic.

Physical Setting

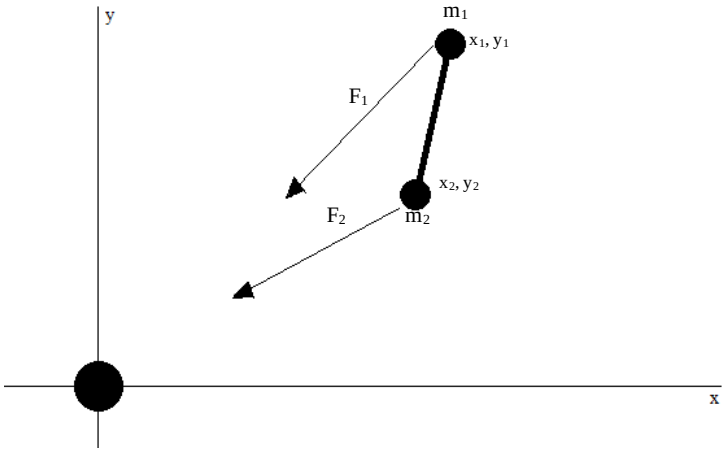


Fig 2.1

Figure 2.1 is a simple model drawn up in graphing software that shows how we will be attempting to simulate the motion of Hyperion. Our moon consists of two particles, m_1 and m_2 which are connected by a massless rigid rod, similar in principle to the "dumbbell" described by Giordano earlier. This orbits around the object seen at the origin of the graph (Saturn), which can be described as "massive" relative to the moon we see. There are two forces acting on each of the masses, the force of gravity from Saturn and the force from the rod, but since we are interested in the motion and torque about the centre of mass, the force from the rod actually has no contribution and has been disregarded from the diagram. This is a good solid base for developing our numerical model later, but first and foremost before we begin to derive any numerical algorithms, we must also develop a method for simulating planetary motion in order to deal with our *x* and *y* components. Then we may build off this to simulate our dumbbell model of Hyperion by including the rotation of the object about the axis perpendicular to the plane in the figure.

Planetary motion is much easier to describe, even numerically. In this investigation, we will be basing our 'base' orbital setting off of Kepler's Laws for planetary motion. That is,

- All planets move in elliptical orbits, with the sun at one focus.
- The line joining a planet to the Sun sweeps out equal areas in equal times.
- If *T* is the period and *a* is the semi-major axis of the orbit, then T^2/a^3 is a constant. For a circular orbit, *a* is just the radius.

It is these observations that will be used in the basis of our planetary motion simulation. In the case of Hyperion's orbit around Saturn, its motion is very obviously elliptical when analysed and viewed graphically – of course this generally is not a perfect ellipse, giving further rise to its chaotic nature. For a two-body system, all three of Kepler's laws are consequences of the fact that the gravitational force follows an inverse-square law. However, in our simulation, the moon is comprised of 2 masses fixed to one another along a rigid massless rod. This means we will deviate slightly from an inverse square dependence – thus

causing a deviation in our elliptical orbit. This is exactly what has been observed astronomically from Hyperion – Please see Figure 2.2 below.

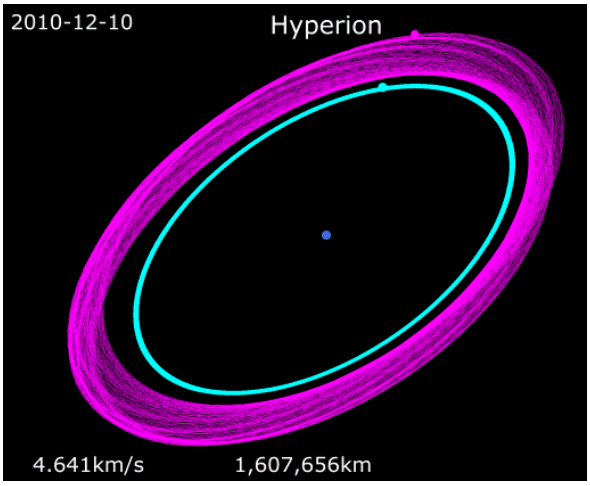
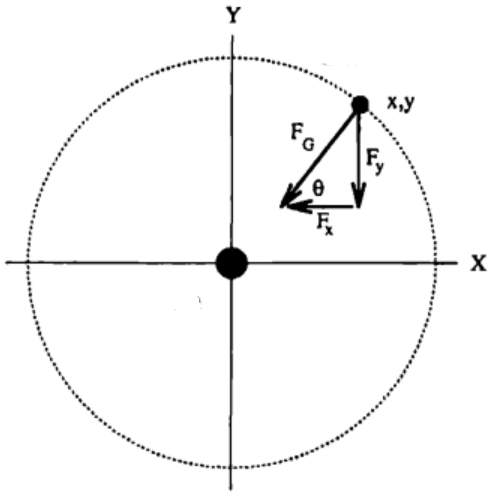


Fig 2.2 – This is a still image from an animation available for viewing on Wikipedia. It animates the observations of Hyperion over a number of years and demonstrates how the closely orbiting moon 'Titan' also has a strong part to play in Hyperion's chaotic motion. We, however, will be neglecting this in our simulation as it is out of our scope and technical ability. The animation is available for viewing in the link found in the references section. [3]

Evaluating the Physical Setting – Equations of Motion



[1] Fig 3.1 – Co-ordinate system for describing the motion of a body orbiting another

Figure 3.1 shows our hypothetical solar system. There is one planet/moon, which in our case is hypothetically Hyperion. Saturn lies at the origin. The only force in the problem, as discussed previously is gravity. According to Newton's law of Gravitation, this force is simply:

$$F_G = \frac{GM_S M_H}{r^2},$$

where M_S and M_H are the mass of Saturn and Hyperion respectively, G is the gravitational constant and r is the distance between the two bodies. In the special case of our system, this

distance will be to the centre of mass of our two particles representing Hyperion (See Figure 2.1). Re-thinking our situation into a more basic Solar-System like, our goal is to calculate the position of Hyperion as a function of time. From Newton's laws of motion, we have:

$$\frac{d^2x}{dt^2} = \frac{F_{G,x}}{M_H},$$
$$\frac{d^2y}{dt^2} = \frac{F_{G,y}}{M_H},$$

(3.1)

where $F_{G,x}$ and the corresponding force in terms of y are the cartesian components of the gravitational force. These are second order differential equations. From our diagram above (Figure 3.1) we have

$$F_{G,x} = -\frac{GM_SM_H}{r^2}cos\theta = -\frac{GM_SM_Hx}{r^3},$$

This is similar for y. The reason our forces are negative is because Hyperion is being directed towards Saturn centripetally. Continuing, our second order differential equations (3.1) have two corresponding first order differentials:

$$\frac{dv_x}{dt} = -\frac{GM_Sx}{r^3}$$
$$\frac{dx}{dt} = v_x$$
$$\frac{dv_y}{dt} = -\frac{GM_Sy}{r^3}$$
$$\frac{dy}{dt} = v_y$$

(3.2)

It is important to note that for this report it has been decided to work in Astronomical units (AU). This is mostly for display and simplification purposes. Appendix C breaks down AU into its SI correspondents and provides an in-depth explanation to the system's advantages. With this in mind, let's continue.

Converting equations of motion (3.2) into difference equations yields:

$$v_{x,i+1} = v_{x,i} - \frac{4\pi^2x_i}{r_i^3}\Delta t$$
$$x_{i+1} = x_i + v_{x,i+1}\Delta t$$
$$v_{y,i+1} = v_{y,i} - \frac{4\pi^2y_i}{r_i^3}\Delta t$$
$$y_{i+1} = y_i + v_{y,i+1}\Delta t,$$

(3.3)

where Δt is the time step and the factor of $4\pi^2$ signals that we are using astronomical units (in terms of Earth-Sun solar system: distance between the Earth and Sun = 1AU, Velocity of Earth = 2πr/(1 year) = 2π(AU/yr)), the difference of course in our case is that our solution is in terms of the radius of Saturn to Hyperion, thus, we are specifying a theoretical unit, let's call it the "Hyperion Unit" or HU. This allows us to leave the solution of GM_S as $4\pi^2$ (Solution due to circular motion), which has been done in the derivation of equations (3.3).

The set of equations (3.3) was derived using The Euler-Cromer method.

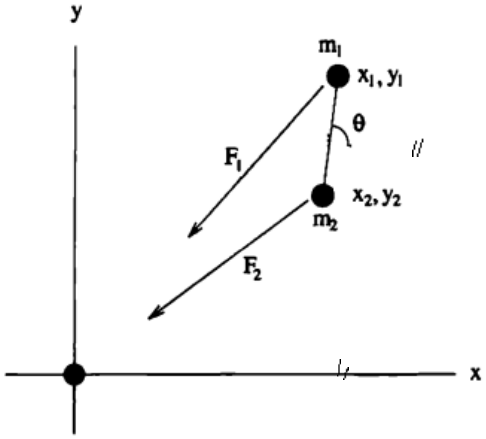
As mentioned previously, Hyperion's orbit is not circular. Let us suppose that the force law deviates slightly from an inverse-square dependence. A bit more specifically, the gravitational force is in the form

$$F_G = \frac{GM_S M_H}{r^\beta}.$$

Differing the value of β to anything other than 2 (an inverse square dependence), generally causes objects to tend not to follow circular paths. In fact, the results can be at times staggering, tracing complex petal-like paths – Elliptical orbits, as seen in the case of Hyperion.

For our solar system program, the only thing we need to change to obtain an elliptical orbit is the initial conditions in our 'difference equations', more specifically by setting our initial velocity to any value greater than 0 (but not equal to 2π) (see equations 3.3).

Taking all of this into the context of our system, we let θ be the angle that the rod makes with the x-axis and define the associated angular velocity $\omega = d\theta/dt$. Thus, we must calculate the variation of θ and ω about the centre of mass of the orbiting 'dumbbell', the general figurative motion of which, about Saturn, was discussed previously in the derivation of equations (3.3).



[1] Fig 3.2 – Co-ordinate system for describing the motion of Hyperion orbiting Saturn, fully labelled.

To do this, we will use, once again, the Euler-Cromer method. Thus, the equation of motion for ω is required. This can be obtained from the torque of the moon. In the discussion of the physical setting, it was discussed that there are technically two forces acting on the masses, that of the rod and the force of gravity from Saturn. We concluded that the force from the rod can be neglected, as it does not contribute since we are interested only in the systems motion. The gravitational force on m_1 can be written as

$$\overrightarrow{F_1} = -\frac{GM_S m_1}{r_1^2}(x_1\hat{\imath} + y_1\hat{\jmath}),$$

where *i* and *j* are unit vectors in the x and y directions and r_1 is the distance from Saturn to m_1. A similar equation can be concluded for vector $\overrightarrow{F_2}$ respectively. The co-ordinates of the centre of mass are (x_c, y_c), this is such that $(x_1 - x_c)\boldsymbol{i} + (y_1 - y_c)\boldsymbol{j}$ is the vector from the centre of mass to m_1. Thus, the torque is as follows:

$$\overrightarrow{\tau_1} = [(x_1 - x_c)\hat{\imath} + (y_1 - y_c)\hat{\jmath}] \times \overrightarrow{F_1}$$

Once again, the expression for $\overrightarrow{\tau_2}$ is similar for $\overrightarrow{F_2}$ respectively. The total torque on the moon is then simply $\overrightarrow{\tau_1} + \overrightarrow{\tau_2}$, relating then to ω by the following:

$$\frac{d\vec{\omega}}{dt} = \frac{\overrightarrow{\tau_1} + \overrightarrow{\tau_2},}{I},$$

where $I = m_1 |r_1|^2 + m_2 |r_2|^2$ is the moment of inertia. Finally, we achieve the following after combining it all together and completing the algebra:

$$\frac{d\omega}{dt} \approx -\frac{3GM_S}{r_c^5}(x_c sin\theta - y_c cos\theta)\,(x_c cos\theta + y_c sin\theta)$$

where r_c is the distance from the centre of mass to Saturn. (3.4)

Once again, using the Euler-Cromer method, we can define the difference equation of ω.

$$\omega_{i+1} = \omega_i - \frac{12\pi^2(x_{c,i} sin\theta_i - y_{c,i} cos\theta_i)(x_{c,i} cos\theta_i + y_{c,i} sin\theta_i)}{r_i^5}\Delta t$$

(3.5)

Corresponding to this, we also require a change in θ based on these conditions. Thus:

$$\theta_{i+1} = \theta_i + \omega_{i+1}\Delta t$$

(3.6)

This analysis provides us with a complete set of difference equations to model Hyperion's chaotic nature.

The Program - Producing Results

In this report, the system was modelled using C++ (Compiled with g++) and visualised using GnuPlot. This came with a lot of limitations and minor procedural annoyances, all of which will be discussed throughout the remainder of the report.

Please note: for the purpose of presentation outside of the academic context of this document, the code displayed is not formatted correctly. If you wish to run the code, please see the attached files submitted with this document. It is better commented and correctly formatted.

C++ Source code (21/09/21) (ChaoticTumbling.cxx (edited)):

```
#include <iostream>
#include <string>
#include <stdio.h>
#include <unistd.h>
#include <math.h>
#include <vector>
#include <stdlib.h>
#include <stdarg.h>
#include <assert.h>
#include "gnuplot.cxx"
using namespace std;

const double PI = 3.141592653589;

int main()
{
	//Solar System
	double dt = 0.001;		//time step - Hyperion years
	double T = 10;			//total time - Hyperion years
	int N = T/dt;			//number of steps - not points

	double x[N+1], y[N+1];		//array of x co-ordinates
					//
	double vx[N+1], vy[N+1];	//velocity values
					//
	double r[N+1];			//radius of orbit (this varies in elliptical
					motion)
					//	all arrays have N+1 components as N
					represents "steps", thus requiring N+1 "points" to
					graph
	double theta[N+1];		//angle "theta" for position in orbit
					//
	double angVelocity[N+1];	//angular velocity
					//
	double time[N+1];		//array for storing values of t - Hyperion Years
					//
		time[0] = 0;		//inital time, *this cannot be modified

	//other initial conditions
	r[0] = 1;			//initial radius - Hyperion unit(s)

	x[0] = r[0];			//starting position is initial length of radius on
					the x axis - Hyperion Unit(s)
	y[0] = 0;			// ^
	vx[0] = 0;			//cannot be modified unless inital position is on
					the y-axis
	vy[0] = 2*PI;			//Since we are starting on the x-axis, vy is our
					inital velocity
```

```
	theta[0] = 0;			//Inital value of theta - radians
	angVelocity[0] = 0;			//inital angular velocity - radians /
						Hyperion Year

	//iteration
	int i = 0;
	while(i <= N){

		//time
		time[i+1] = (i+1)*dt;

		//velocity components
		vx[i+1] = vx[i] - ((4*PI*PI*x[i])/(r[i]*r[i]*r[i]))*dt;
		vy[i+1] = vy[i] - ((4*PI*PI*y[i])/(r[i]*r[i]*r[i]))*dt;

		//cartesian co-ordinates
		x[i+1] = x[i] + (vx[i+1]*dt);
		y[i+1] = y[i] + (vy[i+1]*dt);

		//angular velocity
		angVelocity[i+1] = angVelocity[i] - (((12*PI*PI*((x[i]*sin(theta[i]))-
(y[i]*cos(theta[i])))*((x[i]*cos(theta[i]))+(y[i]*sin(theta[i])))))/(r[i]*r[i]*r[i]*r[
i]*r[i]))*dt;

		//theta
		theta[i+1] = theta[i] + angVelocity[i+1]*dt;

		//theta reset
		if(theta[i+1] > PI){
			theta[i+1] = -PI;
		}
		if(theta[i+1] < -PI){
			theta[i+1] = PI;
		}

		//radius
		r[i+1] = sqrt((x[i+1]*x[i+1]) + (y[i+1]*y[i+1]));
		//cout << r[i] << "\n";


		i++;

	}

	gnuplot_one_function ("Hyperion orbiting Saturn", "lines", "x", "y", x, y,
	N+1);
	gnuplot_one_function ("Theta Vs. Time", "lines", "Time / Hyperion Years",
	"Theta", time, theta, N+1);
	gnuplot_one_function ("Angular Velocity Vs. Time", "lines", "Time / Hyperion
	Years", "AngVelocity", time, angVelocity, N+1);

	return 0;
}
```

Please turn over for results

***Results Produced*:**

Circular Orbit

The following (Figures 4.1 – 4.3) represent the tumbling of Hyperion assuming a circular orbit. The vertical jumps seen in the value of 'Theta' (θ) in Figure 4.2 are due to the resetting of the angle from $-\pi$ to $+\pi$. We also assumed a radius of 1. As discussed previously, this corresponds to 1 HU, where the unit of time corresponds to the period of Hyperion's orbit (1 Hyperion Year). The time step used for these calculations was 0.001 'Hyperion Years'.

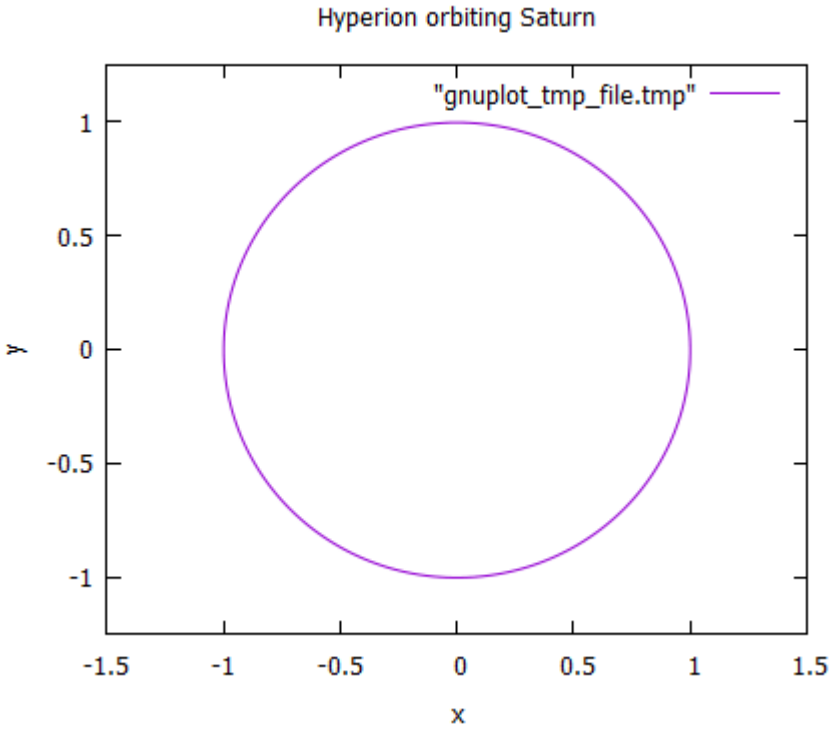


Fig 4.1 – Orbit of Hyperion ($v_0 = 2\pi$)

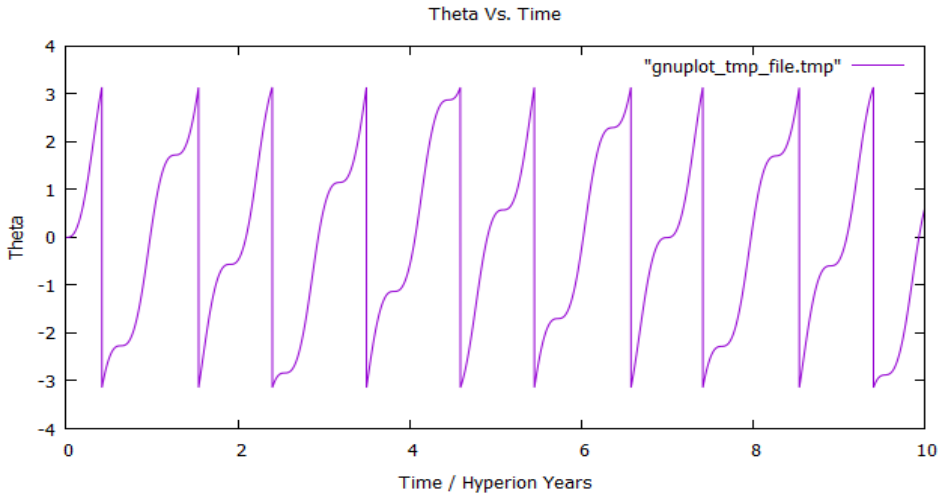


Fig 4.2 – Theta with respect to time (circular orbit)

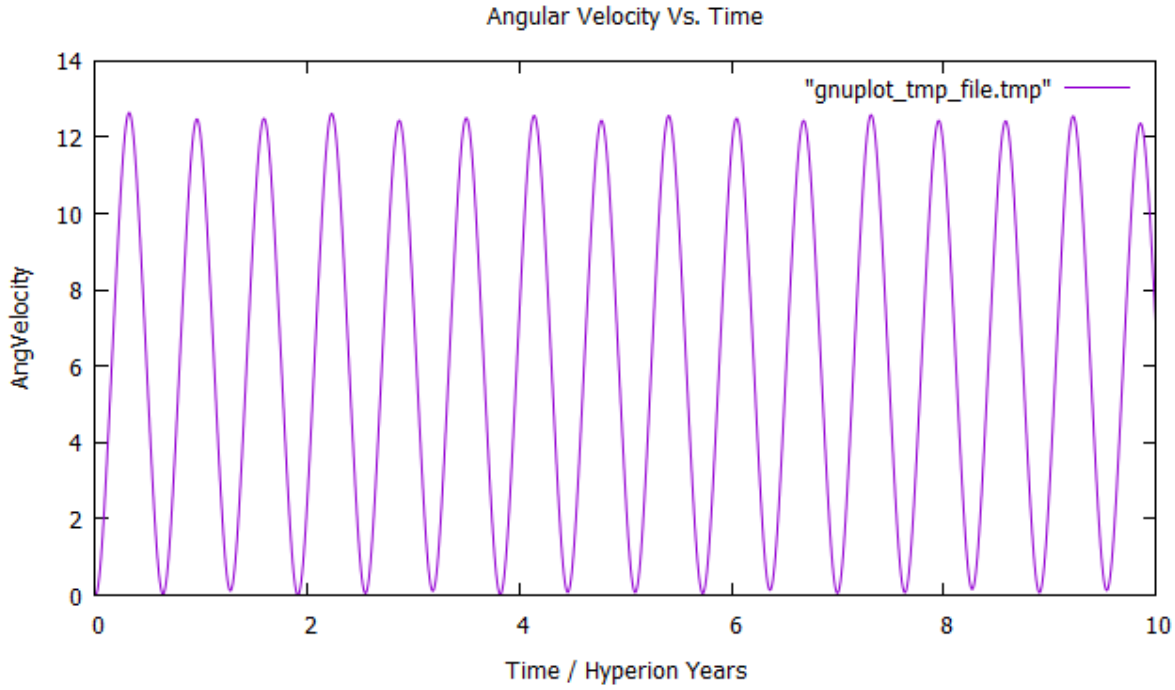


Fig 4.3 – Angular Velocity (ω) with respect to time

Reflecting on these results; it is very clear that the motion is not chaotic with a circular orbit. While obtaining these graphs, our first issue was encountered. As mentioned previously, we used a time-step ("dt" in the source code) of 0.001 Hyperion Years. Ideally, this could have been a smaller value. However, due to how GnuPlot and C++ interact, the values must be sent to GnuPlot via an array. This became an issue within the program due to the number of arrays that were required. Upon researching this, even very powerful machines struggle with this issue. There simply was not enough memory being allocated to the many arrays with massive component numbers (just over 10,000 each for the results seen above). This was an issue purely with C++, whose maximum allocation of a single array lies at 65,536 bytes.[4] Upon extensive testing it was found that no more than ~50,000 components could be input into any array (depending on the data type). A workaround to this would have been to write the data into a file and then send the data directly to GnuPlot via other means (or by using the GnuPlot terminal manually). Unfortunately, this is well out of the scope of this document and would have taken longer than the amount of time allocated to completing this report. Thus, a time step of 0.001 Hyperion Years produced the best results that could be obtained reliably (this remained the case for the elliptical orbits seen below).

Elliptical Orbit

The following (Figures 4.4 – 4.6) represent the tumbling of Hyperion assuming an elliptical orbit. The parameters and initial conditions discussed previously in the circular orbit remain constant. However, to obtain an elliptical orbit our initial velocity (v_{y0}) was initialised to 5 HU (per Hyperion Year) and our value of r did not remain constant (for obvious reasons), this was taken care of in the code by calculating the next length of r during every iteration (more specifically: by finding the absolute or 'norm' value of the vector using its component arrays of $x[\ ]$ and $y[\ ]$).

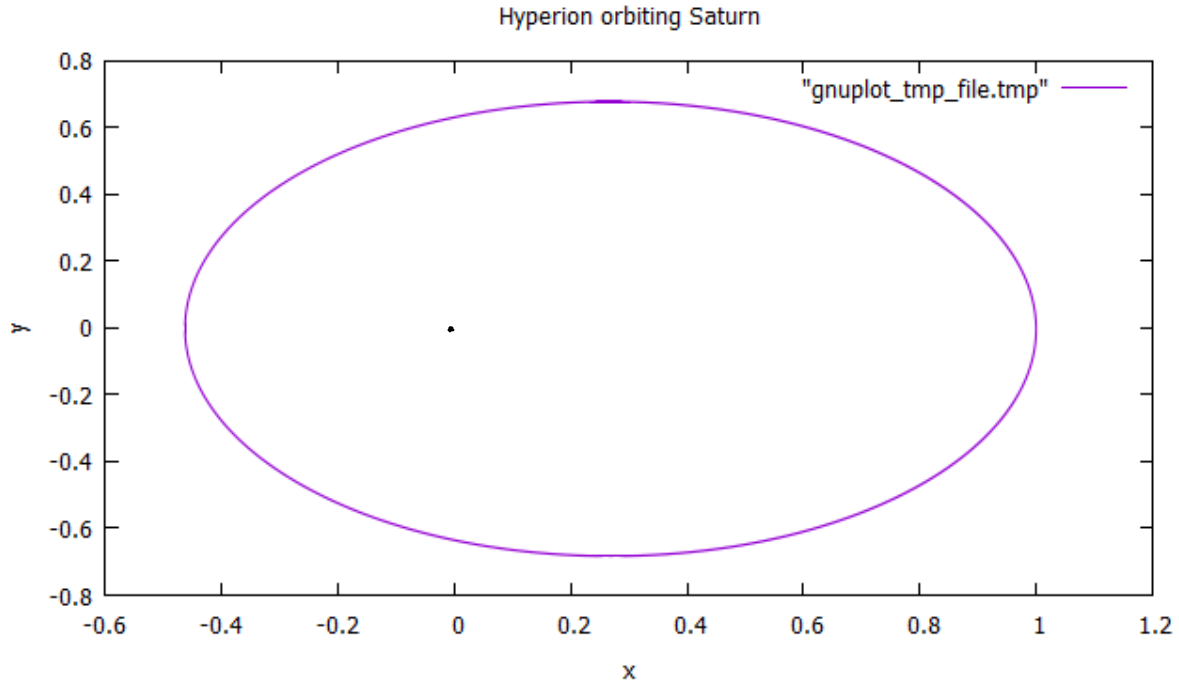


Fig 4.4 – Orbit of Hyperion ($v_0 = 5$)

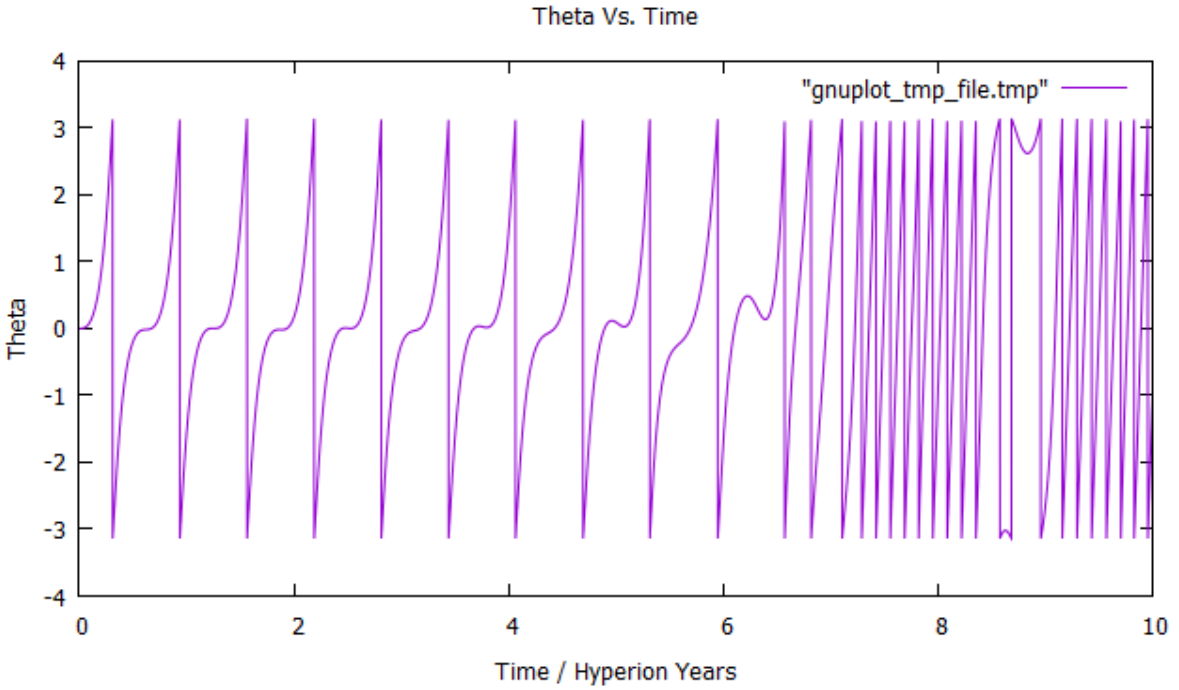


Fig 4.5 – Theta with respect to time assuming an elliptical orbit – chaotic.

Please turn over.

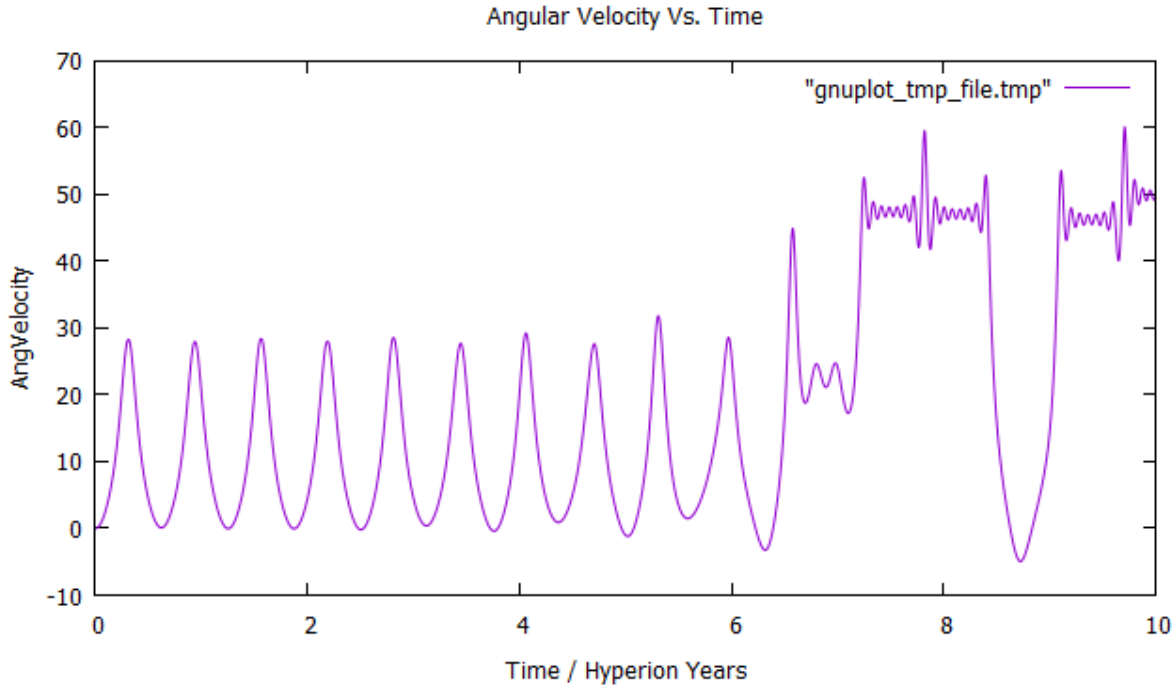


Fig 4.6 – Angular Velocity, ω, with respect to time assuming an elliptical orbit – chaotic.

The results for the elliptical orbit are a far cry from those seen for the circular. The tumbling is now chaotic. It is important to note that this model of Hyperion (as stressed previously) is a highly simplified version of the real thing. Our goal for this report was to prove that the chaotic nature of motion displayed by Hyperion is possible. The results above (Fig 4.4 – 4.6) demonstrate this very well. The motion displayed and suggested by the data in Figures 4.5 and 4.6 show the motion is complicated and erratic after a period of about 5 Hyperion Years. This was intriguing at first when considering that the initial velocity of the moon in orbit was 5 HU/year. However, after extensive testing there was little correlation to be found between these. Altering the initial velocity to a value other than 5 resulted in motion that was immediately chaotic *often* with an unstable elliptical orbit (traced a petal like pattern) – this will be discussed and studied further in the next section.

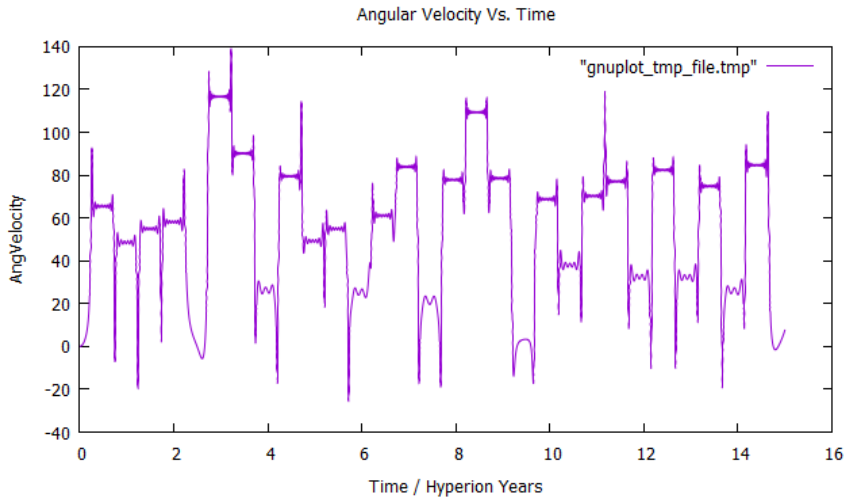


Fig 4.7 – Angular velocity with respect to time, initial velocity v_{y0} = 4 HU/year – immediately chaotic.

Varying Initial Conditions – Finding $\Delta\theta$

In order to study the behaviour of our model for different initial conditions a complete re-structuring of our program was required to accommodate even more arrays. A goal that was set during the development of this program was to allow a *maximum* time-step of 0.001 HU/year with the intention of keeping the time-step consistent with the results seen in the section prior ('The Program - Producing Results').

In order to get around this memory limitation the program instead had to be structured such that run-time and efficiency were of very little concern. The goal at this point was simply to produce results. The program was restructured such that the model of Hyperion was now a function or "method" that could be called and given different initial conditions as specified with editing of the code. This too came with its limitations. With more time dedicated to development, it is not out of the question that this program could be optimised further. The biggest flaw in it's final state is that for every iteration in the main function for calculating 'dtheta[ ]', the 'hyperion( )' function also has to be iterated within that iteration. The only reason for this is due to the fact that C++ cannot return entire arrays. This results in very long run times. However, the program works without issue. This could be fixed using pointers. In fact, this was attempted during it's development, but the program is limited once again by the language's limited memory allocation. The function that simulates Hyperion simply requires far too many arrays to avoid these long load times. This attempt can be found in Appendix A for reference. The code used to produce our results can be found below.

As for $\Delta\theta$ itself, it's value can be found at any given point in time between varying orbits by finding the absolute value of their numerical difference at that time (this generally is in radians). This is the equivalent of saying the following:

$$\Delta\theta \equiv \sqrt{(\theta_1 - \theta_2)^2}.$$

Please note: for the purpose of presentation outside of the academic context of this document, the code displayed is not formatted correctly. If you wish to run the code, please see the attached files submitted with this document. It is better commented and correctly formatted.

C++ Source code (21/11/21) (ChaoticTumblingDTHETA.cxx (edited)):

```
#include <iostream>
#include <string>
#include <stdio.h>
#include <unistd.h>
#include <math.h>
#include <vector>
#include <stdlib.h>
#include <stdarg.h>
#include <assert.h>
#include "gnuplot.cxx"
using namespace std;
```

```
const double PI = 3.141592653589;
const double Ti = 10;
const double dti = 0.001;
const double vy0i = 5;

double hyperion(double q, int j){ //copied from main in previous program, only what
                                  needs to be run. did this to save memory - but not
                                  time unfortunately
                                  //q is the initial value for theta, j is the itera-
                                  tion which is called at the end - unfortuantely the
                                  only viable to do this. -Too many arrays.
        //Solar System
        int N;                    //number of points
        double T, dt;             //time and step size
        dt = dti;                 //Hyperion years
        T = Ti;                   //Hyperion years
        N = T/dt;                 //number of steps - not points

        double x[N+1], y[N+1];          //array of x co-ordinates
        double vx[N+1], vy[N+1];        //velocity values
        double r[N+1];                  //radius of orbit (this varies in elliptical
                                        motion)
                                        //all arrays have N+1 components as N repre-
                                              sents "steps", thus requiring N+1
                                              "points" to graph
        double theta[N+1];              //angle "theta" for position in orbit
        double angVelocity[N+1];        //angular velocity

        //other initial conditions
        r[0] = 1;
        x[0] = r[0];
        y[0] = 0;
        vx[0] = 0;
        vy[0] = vy0i;
        theta[0] = q;                   //Inital value of theta - radians
        angVelocity[0] = 0;             //inital angular velocity

        //iteration
        int i = 0;
        while(i <= N){

                //time
                //time[i+1] = (i+1)*dt;

                //velocity components
                vx[i+1] = vx[i] - ((4*PI*PI*x[i])/(r[i]*r[i]*r[i]))*dt;
                vy[i+1] = vy[i] - ((4*PI*PI*y[i])/(r[i]*r[i]*r[i]))*dt;

                //cartesian co-ordinates
                x[i+1] = x[i] + (vx[i+1]*dt);
                y[i+1] = y[i] + (vy[i+1]*dt);

                //angular velocity
                angVelocity[i+1] = angVelocity[i] - (((12*PI*PI*((x[i]*sin(theta[i]))-
(y[i]*cos(theta[i])))*((x[i]*cos(theta[i]))+(y[i]*sin(theta[i]))))/(r[i]*r[i]*r[i]*r[
i]*r[i]))*dt;

                //theta
                theta[i+1] = theta[i] + angVelocity[i+1]*dt;

                //radius
```

```
            r[i+1] = sqrt((x[i+1]*x[i+1]) + (y[i+1]*y[i+1]));
            //cout << r[i] << "\n";


            i++;

        }

        return theta[j];

}

int main()
{

        int N;              //number of points
        double T, dt;       //time and step size
        dt = dti;           //Hyperion years     //defined by a constant
        T = Ti;             //Hyperion years     //defined by a constant
        N = T/dt;           //number of steps - not points

        double time[N+1];                //array for storing values of t - HU
            time[0] = 0;                 //inital time, *this cannot be modified

        double PhaseDiff = 0.01;    //radians

        //iteration
        int i = 0;

        double theta1[N+1], theta2[N+1], dtheta[N+1];
            dtheta[0] = PhaseDiff;

        while(i <= N){

            theta1[i+1] = hyperion(0, i+1);
            theta2[i+1] = hyperion(PhaseDiff, i+1);

            //time array
            time[i+1] = (i+1)*dt;

            dtheta[i+1] = sqrt((theta1[i+1]-theta2[i+1])*(theta1[i+1]-
            theta2[i+1])));

            //if(dtheta[i+1] >= PI){
                //dtheta[i+1] -= 2*PI;
            //}
            //if(dtheta[i+1] <= -PI){
                //dtheta[i+1] += 2*PI;
            //}

            i++;

        }

        gnuplot_one_function ("Change in Theta Vs. Time", "lines", "Time / Hyperion
        Years", "Change in Theta / radians", time, dtheta, N+1);


        return 0;
}
```

Please turn over for results

***Results Produced*:**

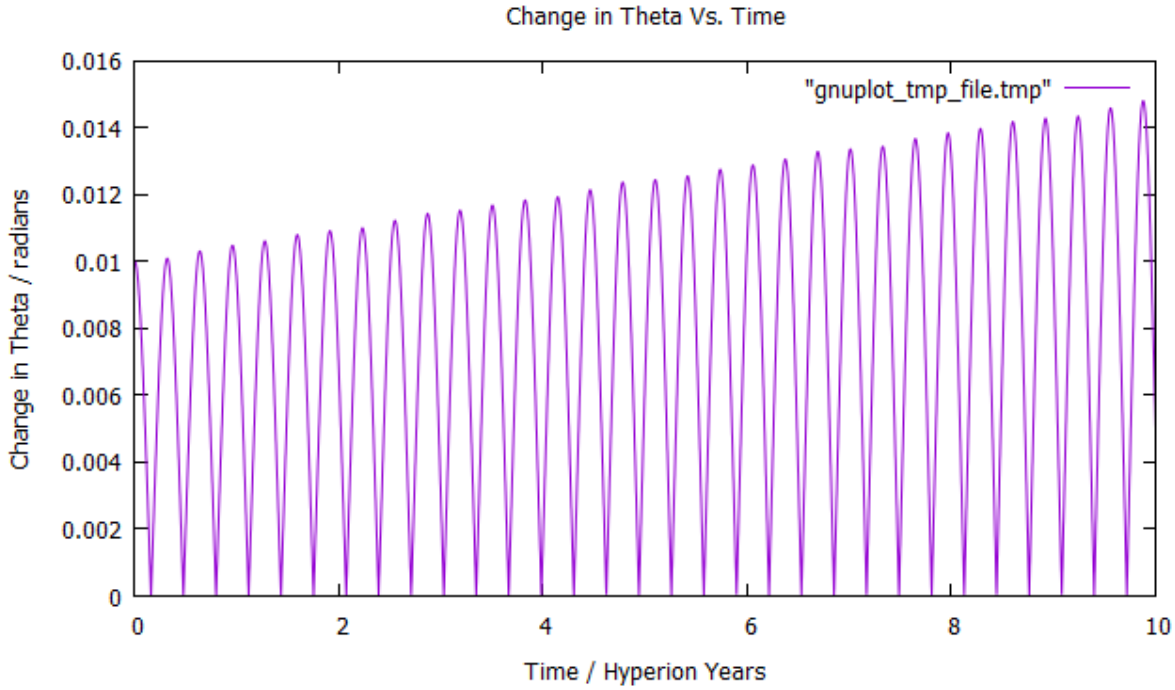


Fig 5.1 – Change in Theta with respect to time – stable circular orbit

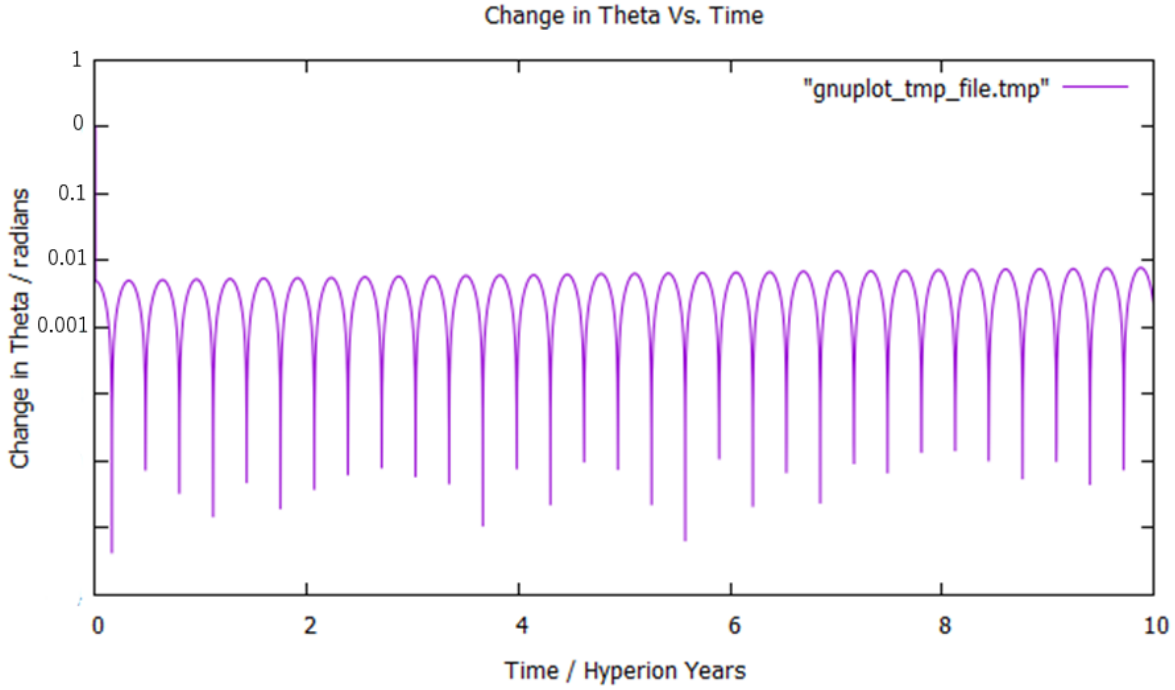


Fig 5.1.1 – Change in Theta with respect to time - Change in theta is linear, logarithmic deviation is null.

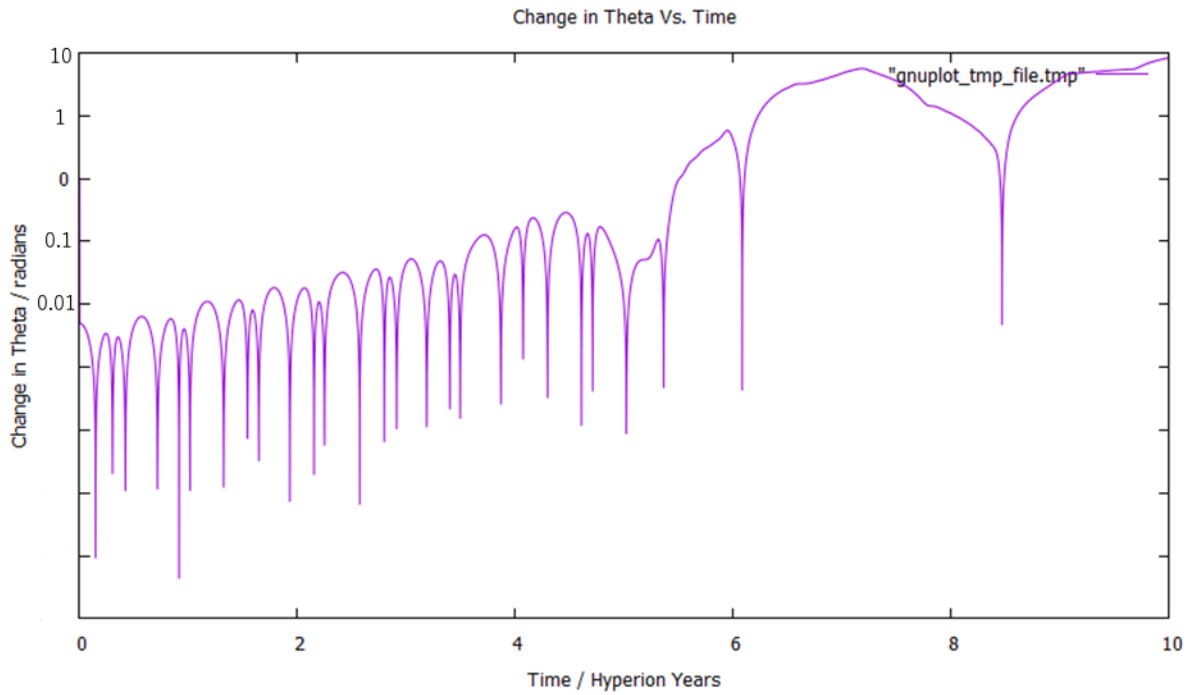


Fig 5.2 – Change in Theta with respect to time – chaotic elliptical orbit – for both sets of orbits shown through figures 5.1 and 5.2, we continue to see chaos present for only elliptical orbits.

The Lyapunov Exponent

"In mathematics, the Lyapunov exponent or Lyapunov characteristic exponent of a dynamical system is a quantity that characterizes the rate of separation of infinitesimally close trajectories." [5] In other words, the Lyapunov exponent is the rate of the exponential separation with time of initially close trajectories. It is an essential tool in studying chaos. **A complete and accurate numerical solution of the Lyapunov exponent for each variation of rotation is not our goal.** This would merit another entire report to itself. Thus, our calculated exponents should only verify what we already know from theory about the orbit in question: whether it is chaotic or not. The absolute value of the estimated Lyapunov exponent has little significance to the conclusions of this report. It's sign, however, will. Generally, if the maximum Lyapunov exponent of a system under particular initial conditions (remember that one of the characteristics of a chaotic system is it's sensitivity to initial conditions) is positive, the system is expected to be chaotic. The relation can be briefly expressed by the following:

$$\Delta\theta \approx e^{\gamma t}$$

where γ denotes the Lyapunov exponent. (6.1)

We have seen previously that Hyperion's orbit, in our simulation, is chaotic only when it's trace is elliptical. From classical mechanics we know that elliptical orbits are far more complex in their trace and calculation in terms of the cartesian plane than that of a circular. To understand and qualitatively show how the Lyapunov exponent relates to our orbit, consider the following:

In the case of our hypothetical orbit of Hyperion around Saturn (or any elliptical orbit in terms of initial velocity, v_0) we can conclude that while

$$v_{y0} < 2\pi: \qquad v_{y0} = v_{min}$$

$$\text{with } a = \frac{1}{1+e}, \text{ and so } e = 1 - (\frac{v}{2\pi})^2.$$

Similarly, while

$$v_{y0} > 2\pi: \qquad v_{y0} = v_{max}$$

$$\text{with } a = \frac{1}{1-e}, \text{ and so } e = (\frac{v}{2\pi})^2 - 1$$

where *a* is the size semi-major axis, as seen in Kepler's 3rd Law of planetary motion, and *e* is the eccentricity of the orbit. Notice that there is no solution of *e*, the eccentricity of the orbit, for a circular orbit. The eccentricity of a circular orbit is null.

These relations were concluded from equation 4.11 in Chapter 4 of 'Computational Physics' by J. Giordano and Nakanishi, which concluded from analysis of Kepler's 2nd Law and the stability of planetary orbits under inverse square (and higher order) laws that

$$v_{max} = \sqrt{GM_O}\sqrt{\frac{(1+e)}{a(1-e)}\left(1+\frac{M_P}{M_O}\right)}$$

$$v_{min} = \sqrt{GM_O}\sqrt{\frac{(1-e)}{a(1+e)}(1+\frac{M_P}{M_O})}$$

Where M_O refers to origin mass (i.e. the mass of the sun in terms of a two body Earth-Sun system) and M_P is the mass of the orbiting body/planet. The full derivation from the text itself is available for viewing in Appendix B.

Applying what we know of the origin of the values of eccentricity, *e*, and the semi major axis, *a*, numerically; we can now use our program from the previous section to investigate these under various initial conditions and visualize how the Lyapunov exponent changes with respect to these.

In terms of the program, little has changed computationally or methodically speaking involving Hyperion. We had to implement a way to calculate our values of *e* and *a* for the respective orbit in simulation for qualitative analysis. Implementing this was not a problem. However, implementing a method to find the Lyapunov exponent proved to be a difficult task. Since our exponent is only a rough estimate of its actual value, a lot of the work can be done quantitatively. Still, it was crucial to produce a number in order to show whether or not an orbit is chaotic – more precisely determined by it's sign. Let's look at Figure 5.2 once again (See below as Figure 5.2.1). There is a rough estimate of the slope (qualitatively average values) that can be made from looking at this graph. Since this graph is in a logarithmic scale, this slope is actually curved and its value increases exponentially with time. This is how we will be calculating our Lyapunov exponent, by find the gradient of a linear slope in the logarithmic plane; this will correspond to the exponential component in the general cartesian (non-logarithmic).

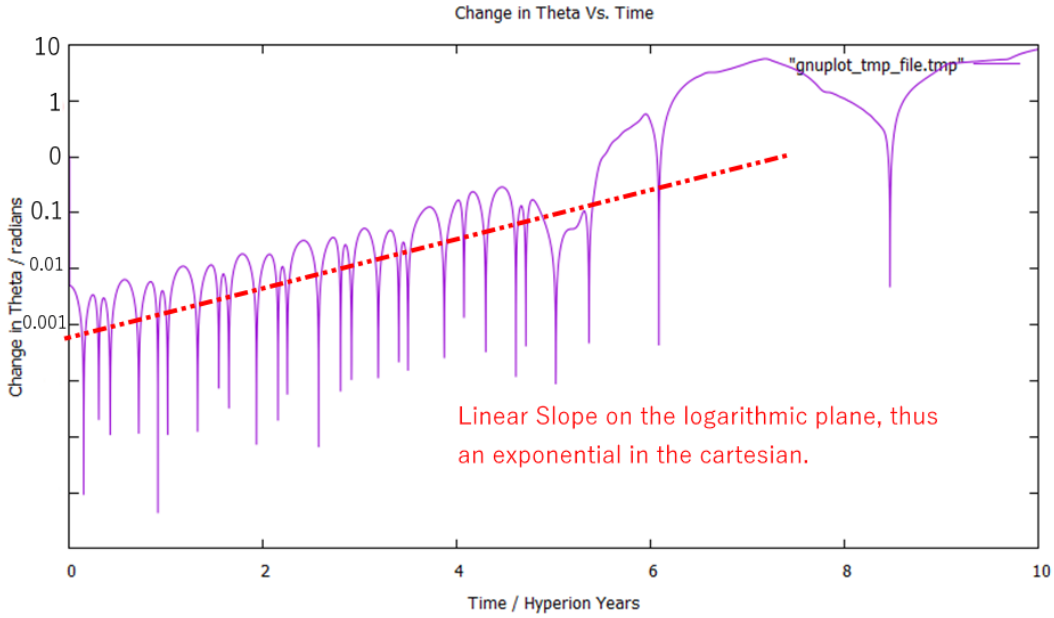


Fig 5.2.1 – Change in Theta with respect to time – demonstrating how the Lyapunov can be determined qualitatively.

This was done in the program by simply taking the vertical value of the logarithmic trace at equal intervals of its course (or seemingly random) in a general cartesian setting. We cannot take values for every step as doing so will decrease the accuracy of our Lyapunov (to elaborate briefly, the values of the *gradient* will overall cancel each other out when summed due to how to the graphs produce their trace). The reason for this is due to the fact that we are finding the gradient between 1 selected point and the previously selected with respect to time, then summing all gradients and dividing by their cardinal number to find the average gradient – Producing a rough value for our Lyapunov. The final source code of this report can be found below.

Finally, it is very important to note a final issue encountered with using GnuPlot and C++ before displaying the results. The integration of GnuPlot into C++ is fantastic for quick calculations and graphs as we have shown previously. However, there is a very clear limitation regarding graph axes and scales. It was not possible to display the graphs shown on a locally logarithmic scale. For previous sections, the images produced by GnuPlot were digitally altered to display the results on a logarithmic scale. However, the shear size of the operation for this section of the report would have been an unwarranted use of time. Thus, the scales of the graph are displayed as was produced, however the trace (graphed line) of the deviation of 'dtheta' ($\Delta\theta$) is accurate logarithmically (where on the axes: 2 = 10, 1 = 1, 0 = 0, -1 = 0.1, and -2 = 0.01 so on respectively).

Please turn over for preview of source code

Please note: for the purpose of presentation outside of the academic context of this document, the code displayed is not formatted correctly. If you wish to run the code, please see the attached files submitted with this document. It is better commented and correctly formatted.

C++ Source code (29/11/21) (ChaoticTumblingFinal_lyapunov.cxx (edited)):

```
#include <iostream>
#include <string>
#include <stdio.h>
#include <unistd.h>
#include <math.h>
#include <vector>
#include <stdlib.h>
#include <stdarg.h>
#include <assert.h>
#include "gnuplot.cxx"
#include <fstream>
#include <sstream>

using namespace std;

const double PI = 3.141592653589;

//consistently changed variables - for ease of access.
const double Ti = 10;
const double dti = 0.0005;
const double vy0i = 10;
const double PhaseDiff_i = 0.01;

double hyperion(double q, int j){

	//Solar System
	int N;				//number of points
	double T, dt;		//time and step size
	dt = dti;			//Hyperion years
	T = Ti;				//Hyperion years
	N = T/dt;			//number of steps - not points

	double x[N+1], y[N+1];			//array of x co-ordinates
	double vx[N+1], vy[N+1];	//velocity values

	double r[N+1];		//radius of orbit (this varies in elliptical motion)

	double theta[N+1];				//angle "theta" for position in orbit
	double angVelocity[N+1];		//angular velocity


	//other initial conditions
	r[0] = 1;
	x[0] = r[0];
	y[0] = 0;
	vx[0] = 0;
	vy[0] = vy0i;
	theta[0] = q;
	angVelocity[0] = 0;

	//iteration
	int i = 0;
	while(i <= N){
```

```
		//velocity components
		vx[i+1] = vx[i] - ((4*PI*PI*x[i])/(r[i]*r[i]*r[i]))*dt;
		vy[i+1] = vy[i] - ((4*PI*PI*y[i])/(r[i]*r[i]*r[i]))*dt;

		//cartesian co-ordinates
		x[i+1] = x[i] + (vx[i+1]*dt);
		y[i+1] = y[i] + (vy[i+1]*dt);

		//angular velocity
		angVelocity[i+1] = angVelocity[i] - (((12*PI*PI*((x[i]*sin(theta[i]))-
(y[i]*cos(theta[i])))*((x[i]*cos(theta[i]))+(y[i]*sin(theta[i])))))/(r[i]*r[i]*r[i]*r[
i]*r[i]))*dt;

		//theta
		theta[i+1] = theta[i] + angVelocity[i+1]*dt;
		//no need to bound theta. We want to measure its difference now.

		//radius
		r[i+1] = sqrt((x[i+1]*x[i+1]) + (y[i+1]*y[i+1]));

		i++;

	}

	return theta[j];

}

int main()
{

	int N;			//number of points
	double T, dt;		//time and step size
	dt = dti;		//Hyperion years
	T = Ti;			//Hyperion years
	N = T/dt;		//number of steps - not points

	//for Lyapunov Analysis:
	double e;		//ecentricty
	double a;		//semi-major axis

	if(vy0i < 2*PI){
		e = 1 -((vy0i/(2*PI))*(vy0i/(2*PI)));	//eccentricity of the system
		a = 1/(1+e);				//semi-major axis
	}else if(vy0i > 2*PI){
		e = (vy0i/(2*PI))*(vy0i/(2*PI)) - 1;
		a = 1/(1-e);
	}
	else if(vy0i == 2*PI){				//Circular orbit
		e = 0;
		a = 1;
	}

	double time[N+1];
		time[0] = 0;

	double PhaseDiff = PhaseDiff_i;	//radians

	//iteration
	double theta1[N+1], theta2[N+1], dtheta[N+1];
		dtheta[0] = PhaseDiff;
			double dtheta_actual[N+1];
```

```
	double Lyapunov[300];
	double LyapunovGRAD[300];

	int i = 0;
	int j = 0;
	double w[10000];
	while(i <= N){

		theta1[i+1] = hyperion(0, i+1);
		theta2[i+1] = hyperion(PhaseDiff, i+1);

		//time array
		time[i+1] = (i+1)*dt;

		//assigning dtheta into the array
		dtheta[i+1] = log(sqrt((theta1[i+1]-theta2[i+1])*(theta1[i+1]-
		theta2[i+1])));
		dtheta_actual[i+1] = sqrt((theta1[i+1]-theta2[i+1])*(theta1[i+1]-
		theta2[i+1]));

		//Lyapunov components
		if(i%100 == 0){
			w[j] = time[i+1];		//w[] stores the times of points taken
			Lyapunov[j] = dtheta[i+1];

			if(j > 0){
				LyapunovGRAD[j] = (Lyapunov[j]-Lyapunov[j-1])/(w[j]-w[j-
				1]);
				//cout << "\nLyapunovGRAD: " << LyapunovGRAD[j];
			}
			//cout << "\nLyapunov: " << Lyapunov[j];
			j++;
		}

		i++;
	}

	//Calculating Lyapunov approximation for qualitative analysis
	double LyapunovGRADsum = 0;
	for(int w = 0; w <= j; w++){
		LyapunovGRADsum += LyapunovGRAD[w];
	}
	double LyapunovGRADfinal = LyapunovGRADsum/j;

	cout << "\nrough estimation of Lyapunov: " << LyapunovGRADfinal;
	cout << "\ne = " << e;
	cout << "\na = " << a;
	cout << "\nj = " << j;

	gnuplot_one_function ("Change in Theta Vs. Time", "lines", "Time / Hyperion
	Years", "Change in Theta / radians", time, dtheta, N+1);
	gnuplot_one_function ("Change in Theta Vs. Time", "lines", "Time / Hyperion
	Years", "Change in Theta / radians", time, dtheta_actual, N+1);


	return 0;
}
```

Please turn over for results

***Results Produced*:**

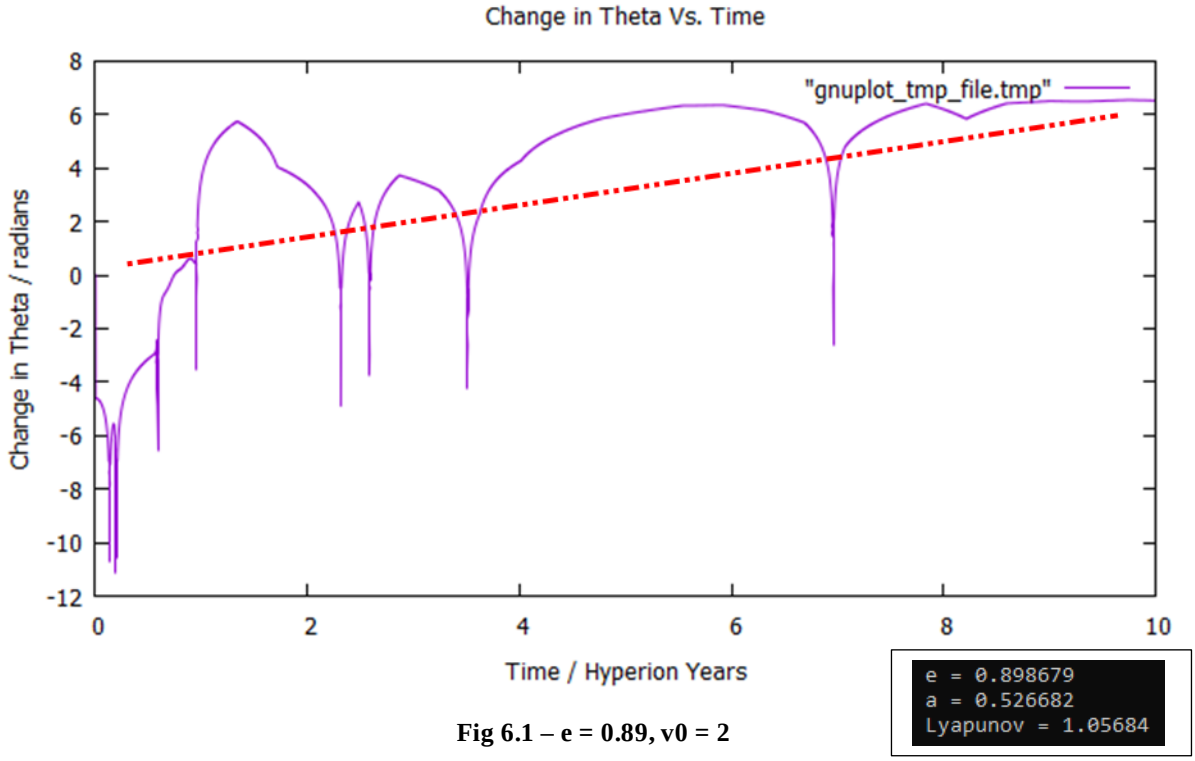


Fig 6.1 – e = 0.89, v0 = 2

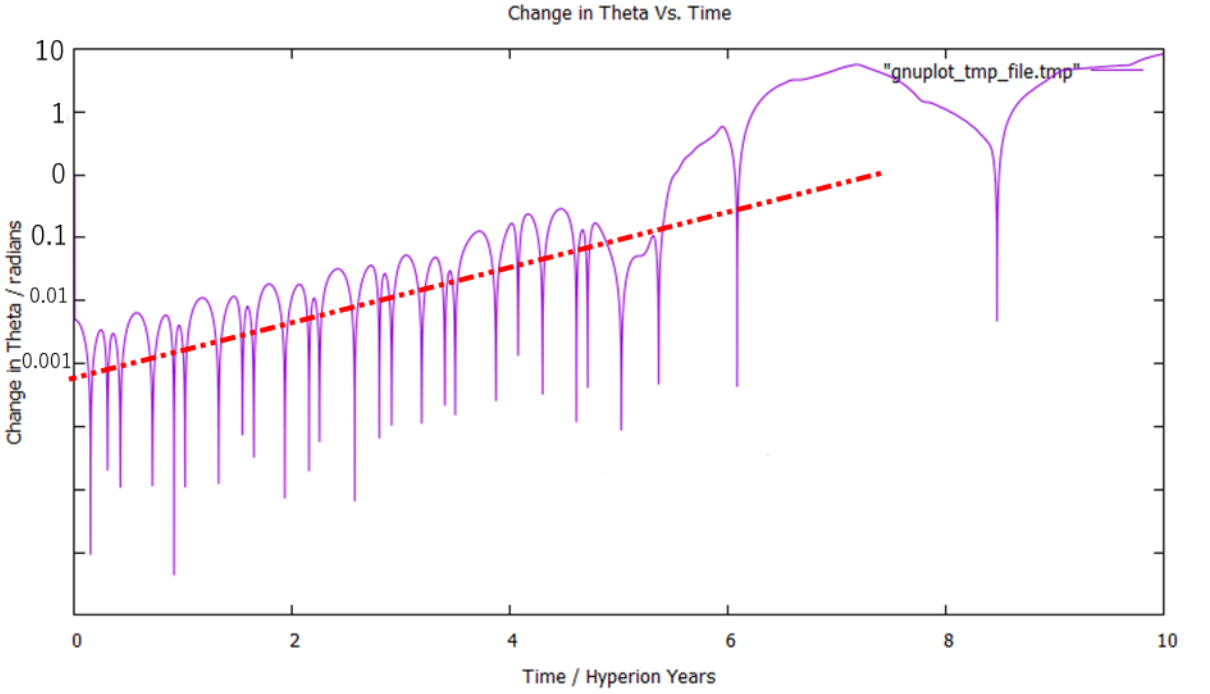


Fig 6.2 – e = 0.367, v0 = 5

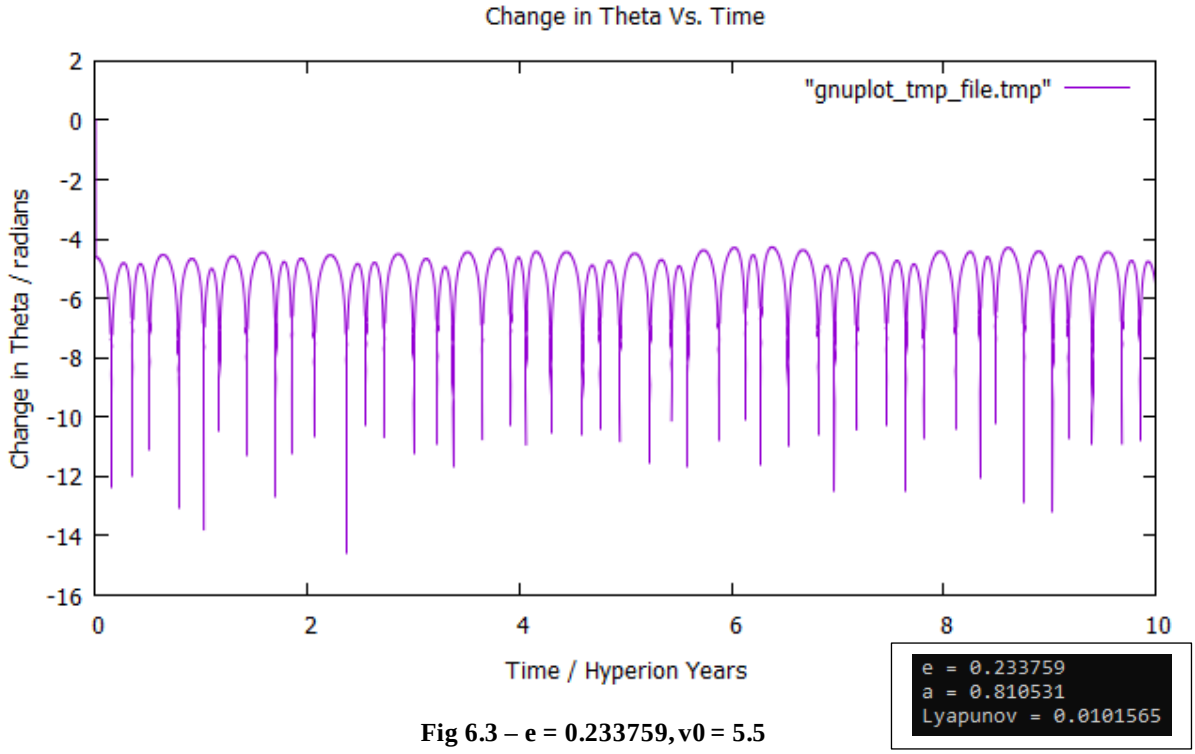


Fig 6.3 – e = 0.233759, v0 = 5.5

Notice here how close the Lyapunov is to 0, and qualitatively how small the deviation is in Figure 6.3. This demonstrates the properties of the chaotic orbit and Lyapunov very nicely. Also notice how we are steadily approaching an initial velocity of 2π in context of the next Figures.

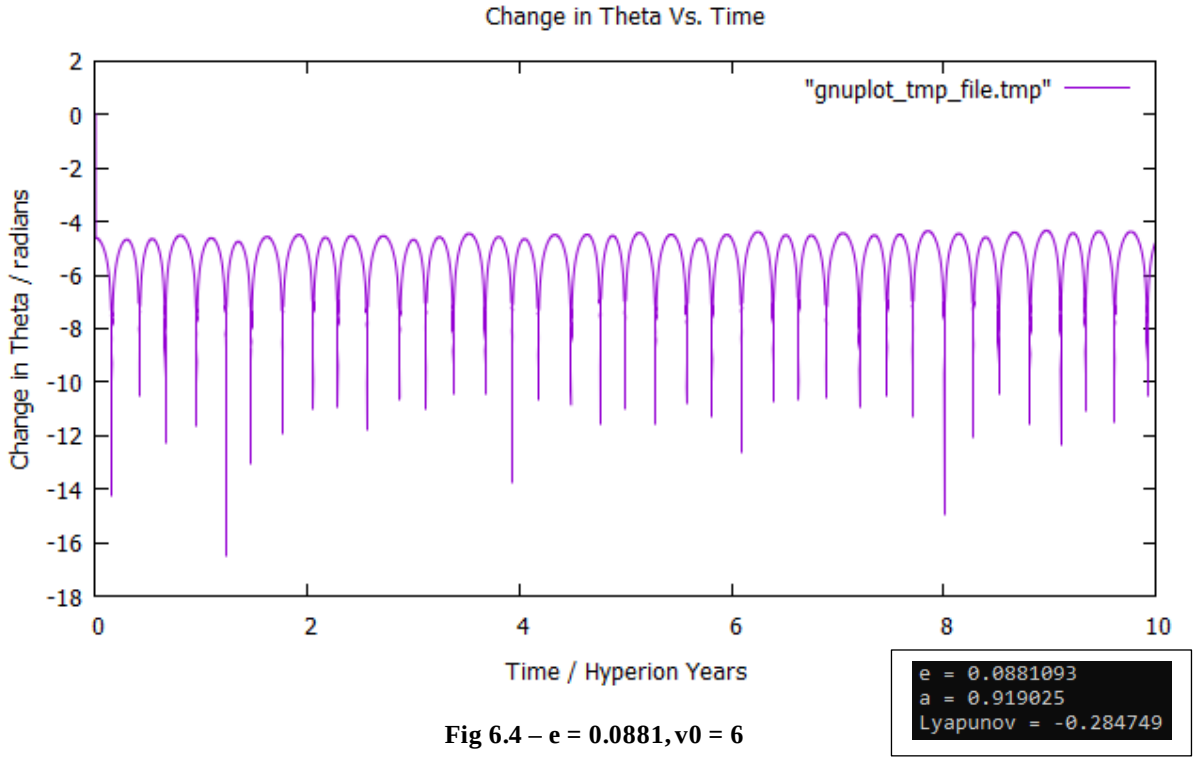


Fig 6.4 – e = 0.0881, v0 = 6

This is where our method in finding the Lyapunov falls apart slightly. In theory this orbit should be chaotic – but as you can see the orbit shows qualities not unlike what we have seen for circular orbits. Notice how close the eccentricity is to 0. Computational errors occurred in calculating the Lyapunov due to how small the qualitative deviation is. A "finer" Lyapunov theoretically greater than 0 is

possible if one were to calculate a value of 'LyapunovGRAD' for larger (or independantly chosen) timesteps (or also by leaving the program run even longer), but this, ultimately, would be impractical and raises more questions than it answers with regards to data manipulation.

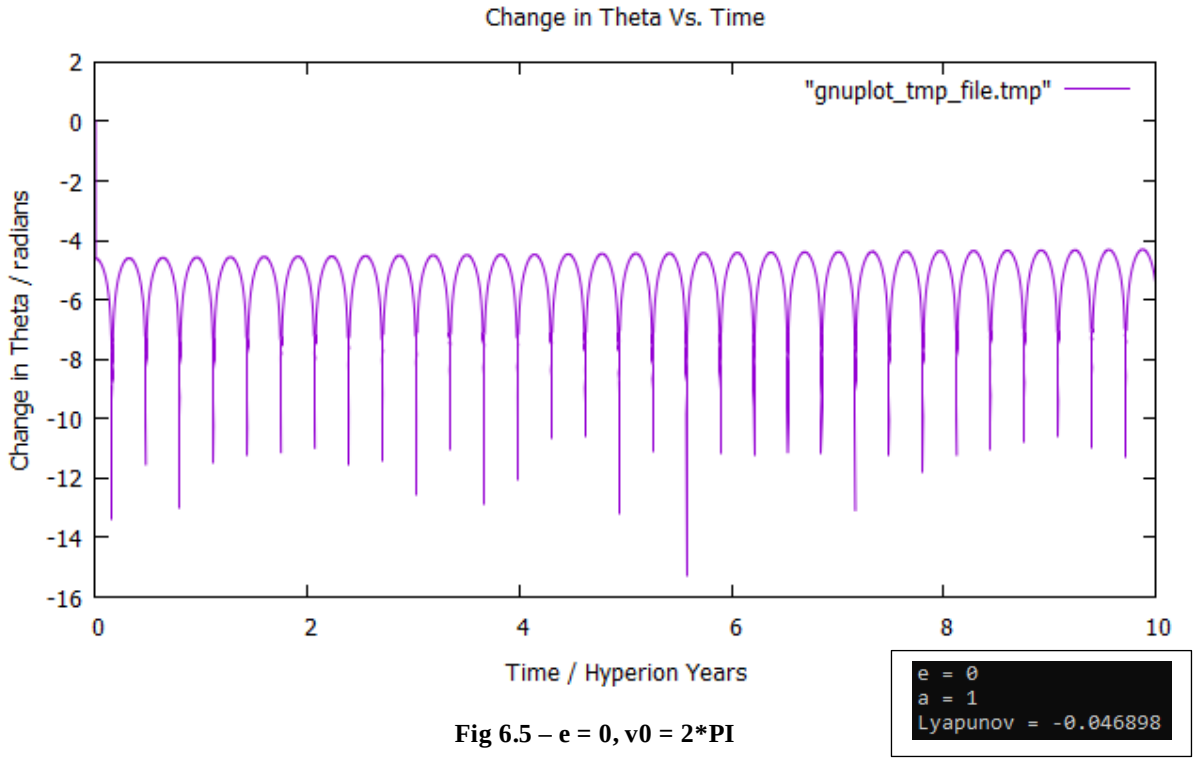


Fig 6.5 – e = 0, v0 = 2*PI

As expected, the Lyapunov is negative in a stable circular orbit.

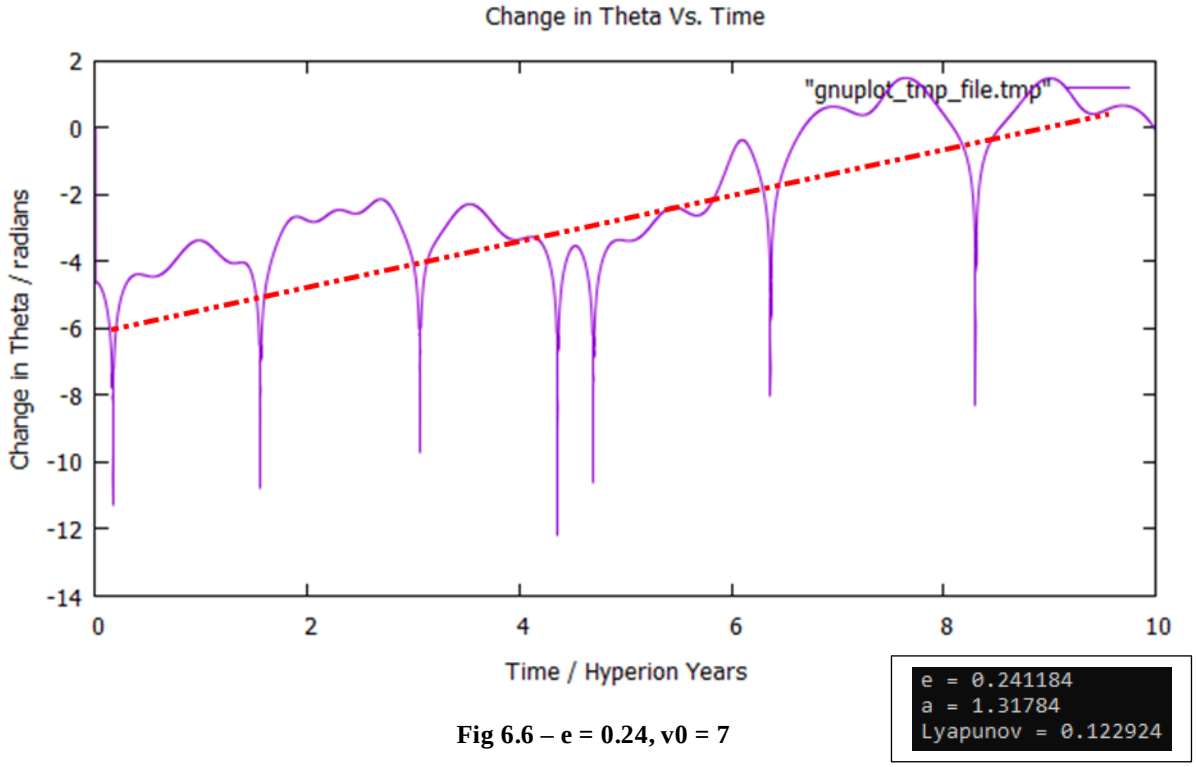
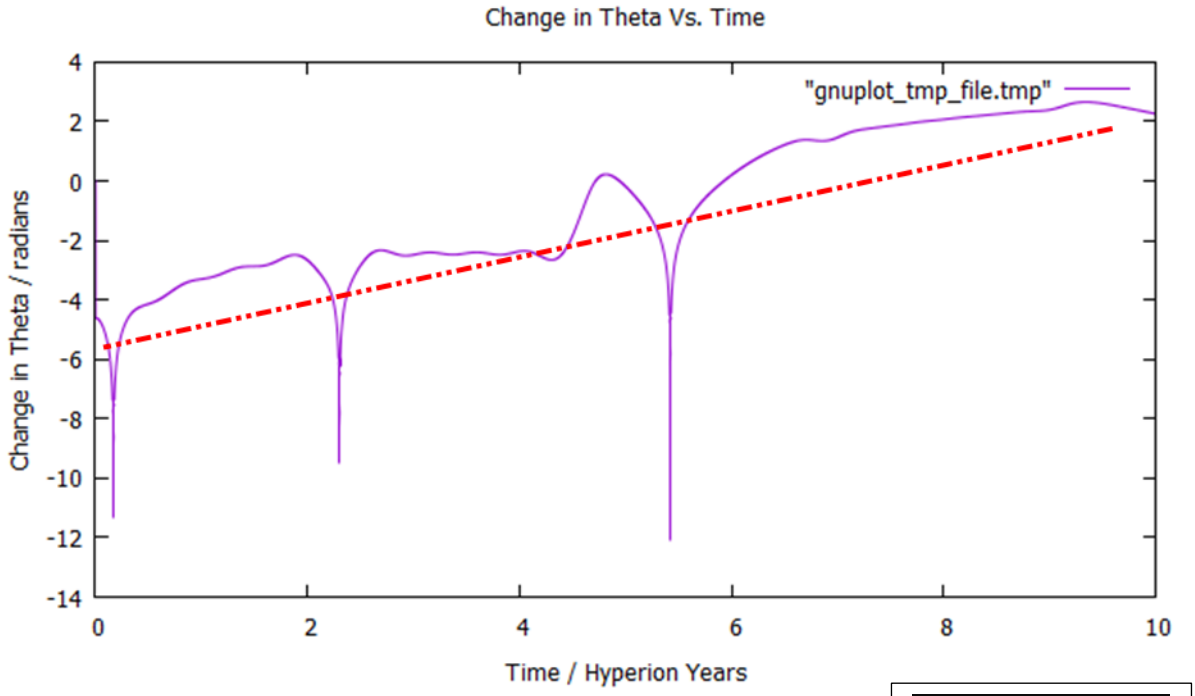


Fig 6.6 – e = 0.24, v0 = 7

Surpassing an initial velocity of 2π has caused our semi major axis to exceed 1HU.




```
e = 0.424829
a = 1.73861
Lyapunov = 0.440313
```

Fig 6.9 – e = 0.42, v0 = 7.5

Discussion/Conclusion

The aim of this report was to show that a moon, irregularly shaped like that of Hyperion, can tumble chaotically. I feel that this has been sufficiently shown. In fact, I am beyond satisfied with the results produced within this report. However, with hindsight, were I to tackle this assignment again, I would certainly use higher-level software to visualise the data with greater sufficiency and a higher-level programming language for computation such as C# or Python.

That said, I thoroughly enjoyed writing and programming the elements of this report in C++. While I initially felt I had little to learn from taking on this assignment, I find now that I have a broader understanding of not only C++, but also basic software integration into code outside Microsoft's .NET framework (C#), the career possibilities in Computational Physics, and a joyfully niched understanding of Hyperion and it's motion.

While it is true that the simulations within this report show little resemblance to the actual motion of Hyperion, I do hope to tackle this topic again at some point in the future to improve on what I have produced here. A better and fully featured analytical calculation of the Lyapunov exponent within the realms of Linear Algebra (for the project in its current structure), and a more accurate trace of Hyperion's actual motion, which may take into account surrounding orbits (Such as that of the nearby moon Titan), would be a goals I would like to achieve. Though for now, I believe these goals are realistically out of the scope of my ability.

Finally, I would like to thank Dr. Andreas Ruschhaupt for aiding my development process significantly in clarifying particular mathematical concepts that I initially found difficult to understand, namely that of Lyapunov exponents and their relation to chaotic orbits.

References

[1] 'Computational Physics', J. Giordano, 2nd Edition, Chapter 4

[2] NASA Visualization Technology Applications and Development (VTAD), Source: https://solarsystem.nasa.gov/moons/saturn-moons/hyperion/in-depth/

[3] 'Animation of Hyperion's orbit', Wikipedia, source: https://en.wikipedia.org/wiki/Hyperion_(moon)#/media/File:Animation_of_Hyperion_orbit_around_Saturn.gif

[4] According to HCL Software, source: https://help.hcltechsw.com/dom_designer/9.0.1/appdev/H_STR_ARRTOOBIG.html#:~:text=maximum%20allowable%20size.-,The%20maximum%20allowable%20array%20size%20is%2065%2C536%20bytes%20(64K).,of%20each%20element%20in%20bytes).

[5] 'Lyapunov Exponent', Wikipedia, source: https://en.wikipedia.org/wiki/Lyapunov_exponent

[6] 'Astronomical Unit', Wikipedia, source: https://en.wikipedia.org/wiki/Astronomical_unit

Appendix A

Alternate Program (Dysfunctional) for the simulation of $\Delta\theta$ with respect to time using pointers in attempt to reduce load times.

Please note: for the purpose of presentation outside of the academic context of this document, the code displayed is not formatted correctly.

C++ Source code (21/11/21) (ChaoticTumblingLypunov2.cxx (edited)):

```
#include <iostream>
#include <string>
#include <stdio.h>
#include <unistd.h>
#include <math.h>
#include <vector>
#include <stdlib.h>
#include <stdarg.h>
#include <assert.h>
#include "gnuplot.cxx"
using namespace std;

const double PI = 3.141592653589;
const double Ti = 10;
const double dti = 0.001;
const double vy0i = 5;
const double PhaseDiff_i = 0.01;

double *hyperion(double q){

	//Solar System
	int N;			//number of points
	double dt, T;		//time and step size
	T = Ti;
	dt = dti;		//Hyperion years;
	N = T/dt;

	double x[N+1], y[N+1];		//array of x co-ordinates
	double vx[N+1], vy[N+1];	//velocity values
	double r[N+1];
	double *theta = new double[N+1];	//angle "theta" for position in orbit
	double angVelocity[N+1];	//angular velocity


	//other initial conditions
	r[0] = 1;
	x[0] = r[0];
	y[0] = 0;
	vx[0] = 0;
	vy[0] = vy0i;
	theta[0] = q;			//Inital value of theta - radians
	angVelocity[0] = 0;		//inital angular velocity - radians / Hyperion Year

	//iteration
	int i = 0;
	while(i <= N){

		//time
		//time[i+1] = (i+1)*dt;

		//velocity components
```

```
                vx[i+1] = vx[i] - ((4*PI*PI*x[i])/(r[i]*r[i]*r[i]))*dt;
                vy[i+1] = vy[i] - ((4*PI*PI*y[i])/(r[i]*r[i]*r[i]))*dt;

                //cartesian co-ordinates
                x[i+1] = x[i] + (vx[i+1]*dt);
                y[i+1] = y[i] + (vy[i+1]*dt);

                //angular velocity
                angVelocity[i+1] = angVelocity[i] - (((12*PI*PI*((x[i]*sin(theta[i]))-
(y[i]*cos(theta[i])))*((x[i]*cos(theta[i]))+(y[i]*sin(theta[i])))))/(r[i]*r[i]*r[i]*r[
i]*r[i]))*dt;

                //theta
                theta[i+1] = theta[i] + angVelocity[i+1]*dt;

                //radius
                r[i+1] = sqrt((x[i+1]*x[i+1]) + (y[i+1]*y[i+1]));

                i++;
        }

        return theta;
}

int main()
{

        int N;                  //number of points
        double T, dt;           //time and step size
        dt = dti;               //Hyperion years
        T = Ti;                 //Hyperion years
        N = T/dt;               //number of steps - not points

        double time[N+1];
                time[0] = 0;

        double PhaseDiff = PhaseDiff_i;
        double theta1[N+1], theta2[N+1], dtheta[N+1];
                dtheta[0] = PhaseDiff;

        //pointers
        double *p0;
        double *p1;
        p0 = hyperion(0);
        p1 = hyperion(PhaseDiff_i);

        //iteration
        int i = 0;
        while(i <= N){
                theta1[i] = p0[i];
                theta2[i] = p1[i];

                //time array
                time[i] = i*dt;
                dtheta[i] = abs((theta1[i]-theta2[i]));

                i++;
        }
        gnuplot_one_function ("Change in Theta Vs. Time", "lines", "Time / Hyperion
        Years", "Change in Theta / radians", time, dtheta, N+1);
        return 0;
}
```

Appendix B [1]

We first note that for a two-body system, all three of Kepler's laws are consequences of the fact that the gravitational force follows an inverse-square law, (4.1). The details of such a calculation can be found in, e.g., Marion and Thornton (1995), but we will sketch some of the analytic results here to lay the ground for further numerical investigations.

We consider a two-body system in which the interaction force depends only on the separation r. The relative motion in this system can be studied as if it were a one-body system; i.e., as if one of the bodies is at rest, while the other orbits about it. The moving body in this equivalent system has a mass equal to the so-called reduced mass $\mu \equiv m_1 m_2/(m_1 + m_2)$ where m_1 and m_2 are the masses of the two original bodies. The position of the equivalent body is given by the relative displacement $\vec{r} \equiv \vec{r}_2 - \vec{r}_1$ of the original two bodies. Of course, if $m_1 \gg m_2$, such as if body 1 were our Sun and body 2 one of its planets, $\mu \approx m_2$ and $\vec{r}$ would be the position of the planet relative to an almost stationary Sun.

The orbital trajectory for a body of reduced mass μ is given in polar coordinates by

$$\frac{d^2}{d\theta^2}\left(\frac{1}{r}\right) + \frac{1}{r} = -\frac{\mu r^2}{L^2} F(r) , \tag{4.8}$$

where $L = \mu r^2 \dot{\theta}$ is the angular momentum and $F(r)$ is the force acting on the body. For the solar system with Sun's mass M_S and a planet's mass M_P, the force is given by $F(r) = -GM_S M_P/r^2$ The angular momentum L is conserved since the system is invariant under rotation.

Since $F(r)$ has the inverse square form (i.e., $F(r) \propto 1/r^2$), (4.8) can be readily solved. For our solar system case, the solution can be expressed as

$$\frac{1}{r} = \left(\frac{\mu G M_S M_P}{L^2}\right)[1 - e\cos(\theta + \theta_0)] , \tag{4.9}$$

or, choosing $\theta_0 = 0$ (which defines the axes of the ellipse),

$$r = \left(\frac{L^2}{\mu G M_S M_P}\right)\frac{1}{1 - e\cos\theta} . \tag{4.10}$$

This result does not give us the position of the planet as a function of time. Rather, it gives the shape of the orbital trajectory, $r(\theta)$. Equation 4.10 describes a *conic section*; it is a circle if $e = 0$, an ellipse if $0 \leq e < 1$, a parabola if $e = 1$, and a hyperbola if $e > 1$, with a focus at the origin. The value of e is known as the eccentricity. Thus, any closed orbit in a two-body system with an inverse square interaction must be elliptical with the location of one of the bodies at a focus, as in Kepler's first law. Kepler's second law amounts to the conservation of angular momentum L, a fact that we have already used.

Equation (4.10) is useful in calculating quantities such as the planet's closest approach (called the *perihelion*) at distance $r_{\min} = a(1-e)$, the farthest point (called the *aphelion*) at $r_{\max} = a(1+e)$, and the corresponding speeds $v_{\max}$ and $v_{\min}$ (see also Figure 4.3) These quantities often provide handy starting points in numerical calculations, e.g., as initial conditions. Note that the length of the semimajor axis of the ellipse, a, is given by $a = L^2/[\mu G M_S M_P (1-e^2)]$ Also, since the angular momentum $L = \sqrt{\mu G M_S M_P a(1-e^2)}$ is conserved, we may set this equal to $\mu r_{\min} v_{\max}$ and to $\mu r_{\max} v_{\min}$. This leads to

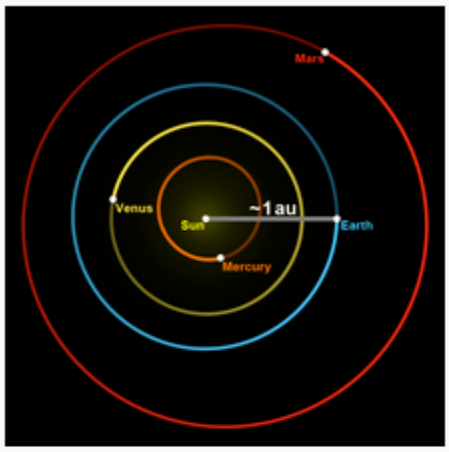
$$v_{\max} = \sqrt{GM_S}\sqrt{\frac{(1+e)}{a(1-e)}\left(1+\frac{M_P}{M_S}\right)} \tag{4.11}$$

$$v_{\min} = \sqrt{GM_S}\sqrt{\frac{(1-e)}{a(1+e)}\left(1+\frac{M_P}{M_S}\right)}\,.$$

Appendix C [6]

The astronomical unit (symbol: au, or au or AU) is a unit of length, roughly the distance from Earth to the Sun and equal to about 150 million kilometres (93 million miles) or ~8 light minutes.

Astronomical unit



The grey line indicates the Earth–Sun distance, which on average is about 1 astronomical unit.

General information

Unit system	Astronomical system of units (Accepted for use with the SI)
Unit of	length
Symbol	au or AU or AU

Conversions

1 au or AU or AU *in* ...	... *is equal to* ...
metric (SI) units	$1.495\,978\,707 \times 10^{11}$ m
imperial & US units	9.2956×10^{7} mi
astronomical units	4.8481×10^{-6} pc 1.5813×10^{-5} ly

The metre is defined to be a unit of proper length, but the SI definition does not specify the metric tensor to be used in determining it. Indeed, the International Committee for Weights and Measures (CIPM) notes that "its definition applies only within a spatial extent sufficiently small that the effects of the non-uniformity of the gravitational field can be ignored". As such, the metre is undefined for the purposes of measuring distances within the Solar System. The 1976 definition of the astronomical unit was incomplete because it did not specify the frame of reference in which time is to be measured, but proved practical for the calculation of ephemerides: a fuller definition that is consistent with general relativity was proposed, and "vigorous debate" ensued until August 2012 when the IAU adopted the current definition of 1 astronomical unit = 149597870700 metres.

The astronomical unit is typically used for stellar system scale distances, such as the size of a protostellar disk or the heliocentric distance of an asteroid, whereas other units are used for other distances in astronomy. The astronomical unit is too small to be convenient for interstellar distances, where the parsec and light-year are widely used. The parsec (parallax arcsecond) is defined in terms of the astronomical unit, being the distance of an object with a parallax of 1″. The light-year is often used in popular works, but is not an approved non-SI unit and is rarely used by professional astronomers.

** When simulating a numerical model of the Solar System, the astronomical unit provides an appropriate scale that minimizes (overflow, underflow and truncation) errors in floating point calculations.