TRAJECTORIES OF PLANETS AND SATELLITES AROUND A STAR SYSTEM

Name: Dyutiman Santra

Subject: Advanced Numerical Techniques (Lab)

Roll No.: 19056762015

Semester: III

University of Delhi

Acknowledgement

I, Dyutiman Santra (roll no.: 19056762015 ; Sem-III), am highly grateful to my teacher, Prof. Kirti Ranjan, who has taught me the computational tools necessary to solve advanced mathematical problems. I have my friends to thank too for they have helped me in understanding certain important concepts associated with this project. I would also like to thank my parents, without whose help and support, I couldn't have finished this project.

Contents

Topic: TRAJECTORIES OF PLANETS AND SATELLITES AROUND A STAR SYSTEM

Aim: The aim here is to simulate the planetary motions using Numerical methods and try to understand how the trajectories would change on changing some parameters.

Tools used:

1. C programing language
2. GNU Plot
3. Numerical Methods
   - Runge Kutta 4th order
   - Trapezoidal rule
   - Simpson rule
   - Gaussian Quadrature

Orbit

In physics, an orbit is the gravitationally curved trajectory of an object, such as the trajectory of a planet around a star or a natural satellite around a planet. Normally, orbit refers to a regularly repeating trajectory, although it may also refer to a non-repeating trajectory. To a close approximation, planets and satellites follow elliptic orbits, with the center of mass being orbited at a focal point of the ellipse, as described by Kepler's laws of planetary motion.

For most situations, orbital motion is adequately approximated by Newtonian mechanics, which explains gravity as a force obeying an inverse-square law.

NOTE: Albert Einstein's general theory of relativity, which accounts for gravity as due to curvature of spacetime, with orbits following geodesics, provides a more accurate calculation and understanding of the exact mechanics of orbital motion.

Numerical Methods used:

1. Trapezoidal Rule

Under this rule, the area under a curve is evaluated by dividing the total area into little trapezoids rather than rectangles.

Let f(x) be continuous on [a,b]. We partition the interval [a,b] into n equal subintervals, each of width
$\Delta x=(b-a)/n$, such that

$$a=x_0<x_1<x_2<\cdots<x_n=b.$$

The Trapezoidal Rule for approximating ${}_b\int_a f(x)dx$ is given by

$${}_b\int_a f(x)dx\approx T_n=\Delta x2[f(x_0)+2f(x_1)+2f(x_2)+\cdots+2f(x_{n-1})+f(x_n)],$$

where $\Delta x=b-an$ and $x_i=a+i\Delta x$.

As $n\to\infty$, the right-hand side of the expression approaches the definite integral ${}_b\int_a f(x)dx$.

2. Simpson Rule

Simpson’s Rule is based on the fact that given three points, we can find the equation of a quadratic through those points.

To obtain an approximation of the definite integral $\int_a^b f(x)dx$ using Simpson’s Rule, we partition the interval [a,b] into an even number n of subintervals, each of width

$$\Delta x=(b-a)/n.$$

On each pair of consecutive subintervals $[x_{i-1},x_i]$, $[x_i,x_{i+1}]$, we consider a quadratic function $y=ax^2+bx+c$ such that it passes through the points $(x_{i-1},f(x_{i-1}))$, $(x_i,f(x_i))$, $(x_{i+1},f(x_{i+1}))$.

If the function f(x) is continuous on [a,b], then

$$\int_a^b f(x)dx\approx\frac{\Delta x}{3}[f(x_0)+4f(x_1)+2f(x_2)+4f(x_3)+2f(x_4)+\cdots+4f(x_{n-1})+f(x_n)].$$

The coefficients in Simpson’s Rule have the following pattern:

1,4,2,4,2,...,4,2,4,1: (n+1 points).

3. Runge Kutta 4th order

A straightforward extension of this method for better accuracy would be to retain higher order terms in Taylor's expansion. However, the calculation of higher order derivatives that this straightforward extension requires is often cumbersome; as a result, this straightforward extension is rarely employed in practice.

Consider however the use of a step like $[\Delta y_i = y_{i+1} - y_i \approx hf(x_i, y_i)]$ to take a trial step to the mid-point of the interval. Then use the value of both x and y at that mid-point to compute the real step across the whole interval:

$$k_i = hf(x_i, y_i)$$

$$k_2 = hf(x_i + \frac{h}{2}, y_i + \frac{k_1}{2})$$

$$y_{i+1} = y_i + k_2 + O(h^3)$$

This symmetrisation cancels the first order error term, h^2, and makes the method 2nd order, i.e., the error term is proportional to h^3. This is called the *2nd order Runge-Kutta method.*

The basic philosophy of the Runge-Kutta methods is to find y_{i+1} by evaluating the derivative of y not only at the initial point (x_i, y_i), but also at the end point and some intermediate points. The increment, Δy, instead of being just $hf(x_i, y_i)$, is some kind of an average of all these derivatives. The intermediate points are so chosen that the terms of order h^2, h^3, …. in the expression for the error cancel out. So, the error is considerably reduced, being proportional to a much higher power of h.

The most widely used method in this class of methods is the 4th-order Runge-Kutta method, RK4. The error in this case is $O(h^5)$. For the solution of the differential equation

$$y' = f(x, y); \quad y(x_0) = y_0$$

$$y_{i+1} = y(x_{i+1}) = y(x_i + h) = y_i + \Delta y_i$$

Where the increment Δy_i is given by

$$\Delta y_i = \frac{k_1+2*(k_2+k_3)+k_4}{6}$$, where

$$k_1 = hf(x, y)$$

$$k_2 = hf(x + \frac{h}{2}, y + \frac{k_1}{2})$$

$$k_3 = hf(x + \frac{h}{2}, y + \frac{k_2}{2})$$

$$k_4 = hf(x + h, y + k_3)$$

The Runge-Kutta method can be extended to a system of any number of coupled 1st-order or higher order differential equations. To solve a pair of simultaneous differential equations:

$$y' = \frac{dy}{dx} = f_1(x, y, z);$$

$$z' = \frac{dz}{dx} = f_2(x, y, z);$$

$$y(x_0) = y_0 \quad z(x_0) = z_0$$

We evaluate the quantities

$$k_1 = hf_1(x_i, y_i, z_i) \quad m_1 = hf_2(x_i, y_i, z_i)$$

$$k_2 = hf_1(x_i + \frac{h}{2}, y_i + \frac{k_1}{2}, z_i + \frac{m_1}{2}) \quad m_2 = hf_2(x_i + \frac{h}{2}, y_i + \frac{k_1}{2}, z_i + \frac{m_1}{2})$$

$$k_3 = hf_1(x_i + \frac{h}{2}, y_i + \frac{k_2}{2}, z_i + \frac{m_2}{2}) \quad m_3 = hf_2(x_i + \frac{h}{2}, y_i + \frac{k_2}{2}, z_i + \frac{m_2}{2})$$

$$k_4 = hf_1(x_i + h, y_i + k_3, z_i + m_3)$$

$$m_4 = hf_2(x_i + h, y_i + k_3, z_i + m_3)$$

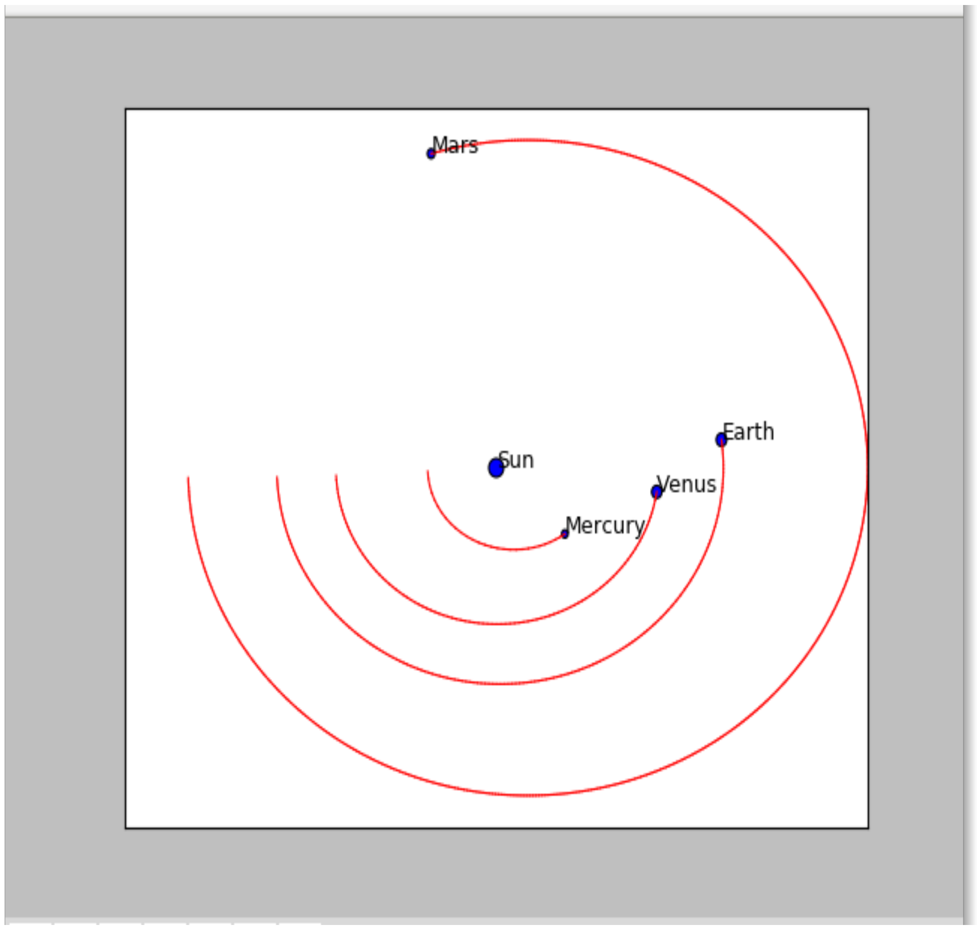
This leads to (at $x_{i+1} = x_i + \Delta h$)

$$y_{i+1} = y_i + \frac{k_1 + 2*(k_2 + k_3) + k_4}{6}$$

$$z_{i+1} = z_i + \frac{m_1 + 2*(m_2 + m_3) + m_4}{6}$$

This is the complete algorithm for RK4 method. It is very helpful in solving 2nd order differential equations.

Mathematical Structure of the problem

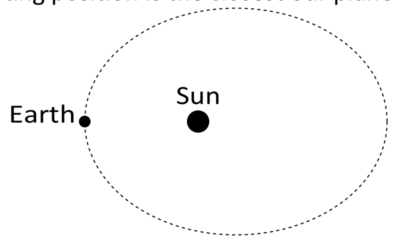


Generating planet trajectories in C

The main part of the project is to write a C program which computes the orbits of objects in space, interacting via gravity. Because of the complexity of the code, we will divide it in various functions, where each function has its own unique task. It is important that we solve each task at a time and validate the functionality that we have, before proceeding.

Earth moving around the Sun

The first version of the code will only simulate one single planet rotating around a fixed sun. Our simulations take place in 2D and we place our sun at the origin (0,0) of our coordinate system. Our planet, Earth, will be placed at (-147095000000, 0). This starting position is the closest our planet gets to our sun during a year.



To simulate how Earth moves around the sun we need to solve the equations of motion of the planet. In general, these equations are 2nd-order ordinary differential equations, one for each coordinate:

$$\ddot{x} = \frac{F_x}{m}$$
$$\ddot{y} = \frac{F_y}{m}$$

where F_x is the force acting on the planet in x-direction and F_y is the force acting in the y-direction. m is the mass of the planet. In order to solve the equations, they are usually transformed into two 1st-order ordinary differential equations for each component:

$$\dot{x} = v_x$$
$$\dot{y} = v_y$$
$$\dot{v_x} = \frac{F_x}{m}$$
$$\dot{v_y} = \frac{F_y}{m}$$

where v_x is the velocity in x-direction, and v_y the velocity in y-direction of the object.

These types of equations can be solved numerically by a computer program. For this we discretize and integrate them. In essence how this works is taking the current position (x_n,y_n) and velocity ($v_{x,n}$,$v_{y,n}$) of our planet and compute new values for each of these after a given time step Δt. We call these new values (x_{n+1},y_{n+1}) and ($v_{x,n+1}$,$v_{y,n+1}$). As time step Δt we choose one 1 Earth day, which is 86400 seconds.

What our program will be doing is compute a new position and velocity of the planet after each day, as illustrated in the following picture:

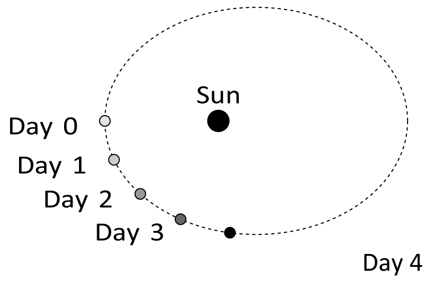


Figure: After each day we compute a new point of the planet's orbit

For this we need a formula to compute the new positions and velocities based on their current values and the time step:

$$x_{n+1} = x_n + \Delta t(\dots) \; y_{n+1} = y_n + \Delta t(\dots)$$

$$v_{x,n+1} = v_{x,n} + \Delta t(\dots) \; v_{y,n+1} = v_{y,n} + \Delta t(\dots)$$

Gravitational force between the two masses

The force of one mass M of an object at location (x_M, y_M) acting on another object of mass m at location (x,y) is given by the following formulas in x- and y-direction.

$$\Delta x = x_M - x$$
$$\Delta y = y_M - y$$
$$F_x = \frac{GMm}{\Delta x^2 + \Delta y^2}\frac{\Delta x}{\sqrt{\Delta x^2 + \Delta y^2}}$$
$$F_y = \frac{GMm}{\Delta x^2 + \Delta y^2}\frac{\Delta y}{\sqrt{\Delta x^2 + \Delta y^2}}$$

where G is the gravitational constant:

$$G = 6.67408 \cdot 10^{-11}$$

With this formula and Newton's law $F = ma$, we can compute the amount of acceleration contributed by a mass M to a body at position (x,y):

$$a_x = \frac{GM}{\Delta x^2 + \Delta y^2}\frac{\Delta x}{\sqrt{\Delta x^2 + \Delta y^2}}$$
$$a_y = \frac{GM}{\Delta x^2 + \Delta y^2}\frac{\Delta y}{\sqrt{\Delta x^2 + \Delta y^2}}$$

For our simplest example, where the sun is located at the orgin ($x_M = 0$, $y_M = 0$), the formula reduces to:

$$a_x = \frac{GM_{\text{Sun}}}{x^2 + y^2}\frac{-x}{x^2 + y^2} \quad \text{p}$$

$$\frac{\text{Sun}}{x^2 + y^2}\frac{}{\sqrt{x^2 + y^2}} \qquad GM \quad -y \; a_y =$$

Note that in either case, a_x and a_y are functions of x and y:

$$a_x = a_x(x,y)$$

$$a_y = a_y(x,y)$$

So, depending where we are relative to the body with mass M, we experience a different acceleration due to the gravitational force.

Integrating a single time step

Now, we will write the code necessary to integrate one single time step and compute the new position and velocity of our planet. The goal is to find the new location of Earth after one day. The step method will now be implemented.

Numeric integration using 4th order Runge-Kutta method

The numeric integration of the new position and velocity is the heart of our orbit simulation program. It should be implemented in a step member function. The method takes two parameters: the time step dt and the body which creates forces. **I have assumed that the only body doing this is the sun.**

Because we are using such a large time step, we have to write the integration using a very sophisticated method called a Runge-Kutta integrator of 4th order. We use the following scheme to translate the formulas into code:

$$x_{n+1} = x_n + \frac{\Delta t}{6}\left(k_{1,x} + 2k_{2,x} + 2k_{3,x} + k_{4,x}\right)$$
$$y_{n+1} = y_n + \frac{\Delta t}{6}\left(k_{1,y} + 2k_{2,y} + 2k_{3,y} + k_{4,y}\right)$$

$$v_{x,n+1} = v_{x,n} + \frac{\Delta t}{6}\left(k_{1,v_x} + 2k_{2,v_x} + 2k_{3,v_x} + k_{4,v_x}\right)$$
$$v_{y,n+1} = v_{y,n} + \frac{\Delta t}{6}\left(k_{1,v_y} + 2k_{2,v_y} + 2k_{3,v_y} + k_{4,v_y}\right)$$

As we can see, the new position (x_{n+1},y_{n+1}) and velocities ($v_{x,n+1}$,$v_{y,n+1}$) are dependent on the time step Δt, the current position (x_n,y_n) and velocity ($v_{x,n}$,$v_{y,n}$), as well as 16 parameters ($k_{1,x}$, $k_{2,x}$, $k_{3,x}$, $k_{4,x}$, $k_{1,y}$, $k_{2,y}$, $k_{3,y}$, $k_{4,y}$, k_{1,v_x}, k_{2,v_x}, k_{3,v_x},k_{4,v_x}, k_{1,v_y}, k_{2,v_y}, k_{3,v_y}, k_{4,v_y}).

The tricky part is computing these parameters, but all we have to do is follow the next steps and combine them to the above equation to get the new positions and velocities.

1. Computing $k_{1,x}$ and $k_{1,y}$:

$$k_{1,x} = v_{x,n} \quad k_{1,y} = v_{y,n}$$

2. Computing k_{1,v_x} and k_{1,v_y}:

$$k_{1,v_x} = a_x(x_n,y_n) \quad k_{1,v_y} = a_y(x_n,y_n)$$

NOTE: All following steps which evaluate a_x and a_y do the same, but use different x and y values.

3. Computing $k_{2,x}$ and $k_{2,y}$:

$$k_{2,x} = v_{x,n} + \frac{\Delta t}{2} k_{1,v_x}$$
$$k_{2,y} = v_{y,n} + \frac{\Delta t}{2} k_{1,v_y}$$

4. Computing k_{2,v_x} and k_{2,v_y}:

$$k_{2,v_x} = a_x(x_n + \frac{\Delta t}{2} k_{1,x}, y_n + \frac{\Delta t}{2} k_{1,y})$$
$$k_{2,v_y} = a_y(x_n + \frac{\Delta t}{2} k_{1,x}, y_n + \frac{\Delta t}{2} k_{1,y})$$

5. Computing $k_{3,x}$ and $k_{3,y}$:

$$k_{3,x} = v_{x,n} + \frac{\Delta t}{2} k_{2,v_x}$$
$$k_{3,y} = v_{y,n} + \frac{\Delta t}{2} k_{2,v_y}$$

6. Computing k_{3,v_x} and k_{3,v_y}:

$$k_{3,v_x} = a_x(x_n + \frac{\Delta t}{2} k_{2,x}, y_n + \frac{\Delta t}{2} k_{2,y})$$
$$k_{3,v_y} = a_y(x_n + \frac{\Delta t}{2} k_{2,x}, y_n + \frac{\Delta t}{2} k_{2,y})$$

7. Computing $k_{4,x}$ and $k_{4,y}$:

$$k_{4,x} = v_{x,n} + \Delta t \cdot k_{3,v_x}$$
$$k_{4,y} = v_{y,n} + \Delta t \cdot k_{3,v_y}$$

8. Computing k_{4,v_x} and k_{4,v_y}:

$$k_{4,v_x} = a_x(x_n + \Delta t \cdot k_{3,x}, y_n + \Delta t \cdot k_{3,y})$$
$$k_{4,v_y} = a_y(x_n + \Delta t \cdot k_{3,x}, y_n + \Delta t \cdot k_{3,y})$$

- Once the function has been implemented, we verify the program.
- Next, we simulate 365 days and generate our desired results and trajectories.
- We move on to do the same for a satellite around a planet (moon around earth) and add another planet (mars).
- We also calculate the central potential energy and the effective potential energy.
- The next part is to change the parameters like mass and radii of the system and plot the varying trajectories.

Plots

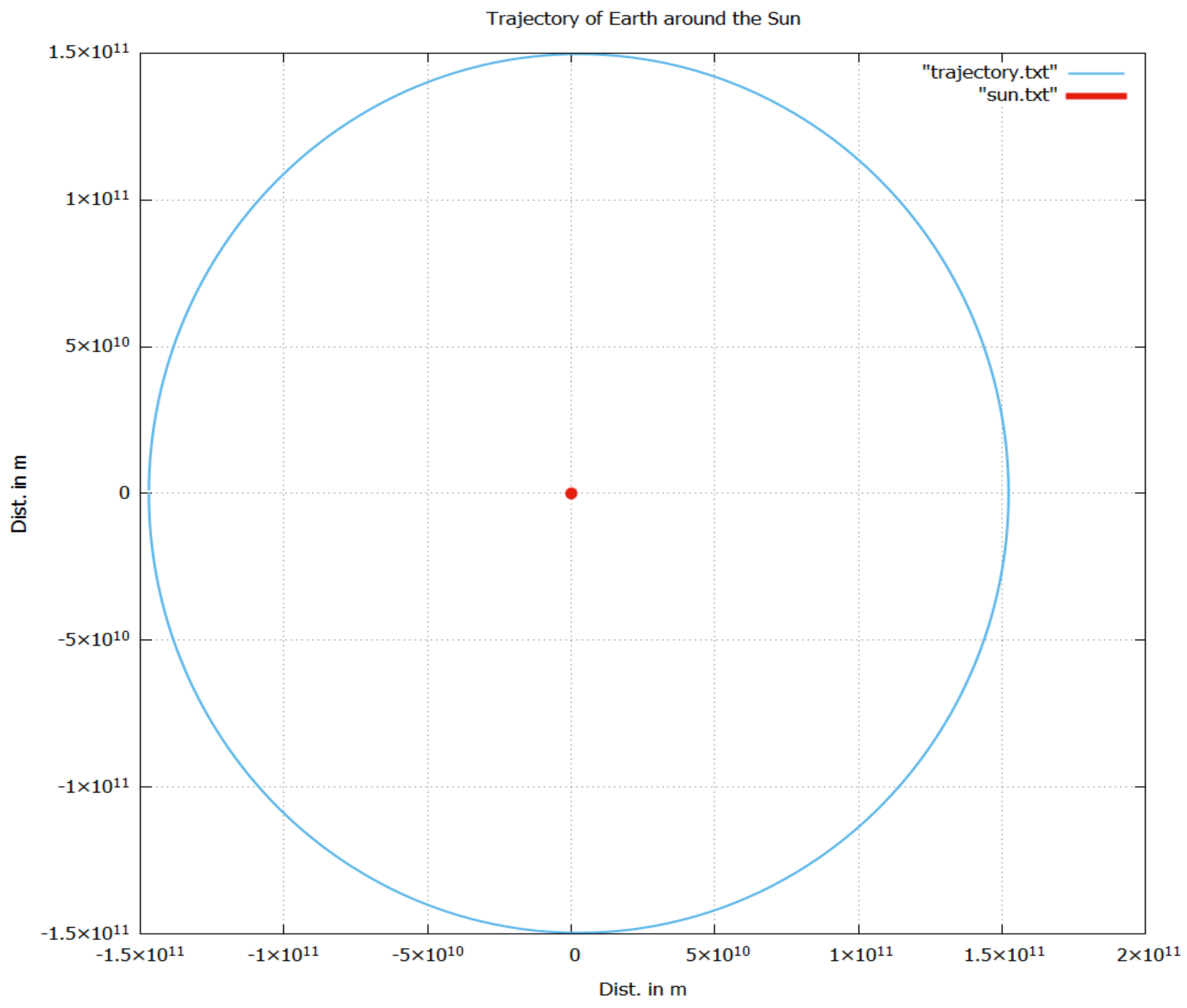
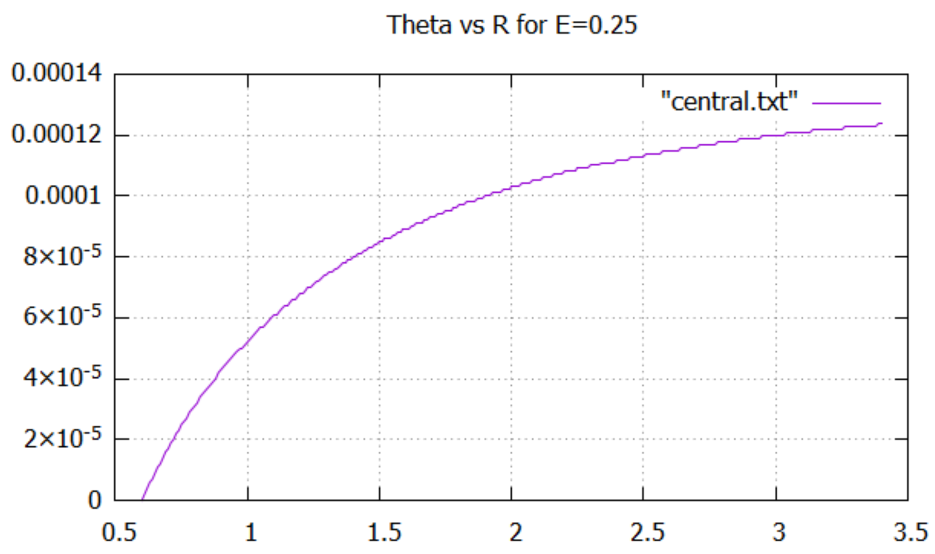
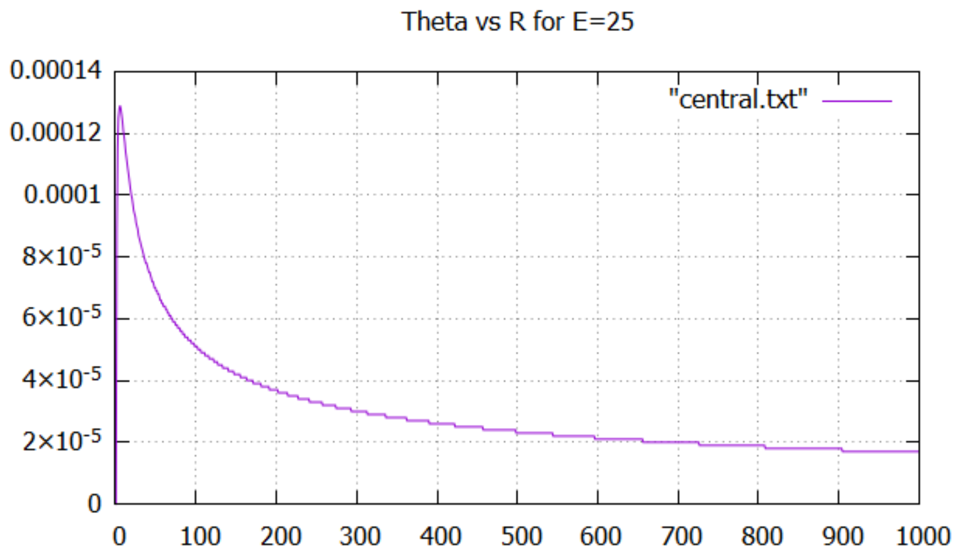


Figure 1: The plot shows the trajectory of Earth, our green planet, around the big, beautiful and hot sun.



$$\theta(R) = \int_{r_0}^{R} \frac{dr}{r^2\left[\left(\frac{2mE}{l^2}\right) - \left(\frac{2mV(r)}{l^2}\right) - \frac{1}{r^2}\right]^{\frac{1}{2}}}$$



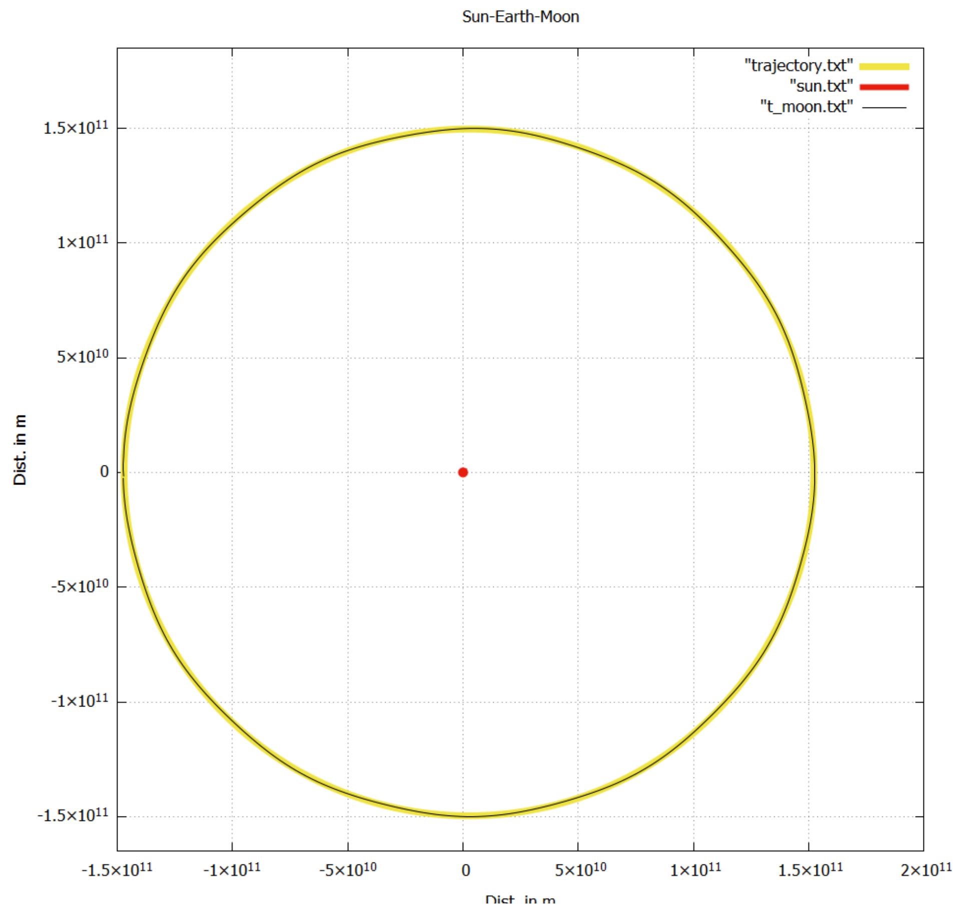


Figure 2: The above plot shows the trajectory of both Earth and Moon around the sun. The trajectory of the moon is seen to slightly vary along the trajectory of earth, owing to the fact that the moon itself revolves around the earth.

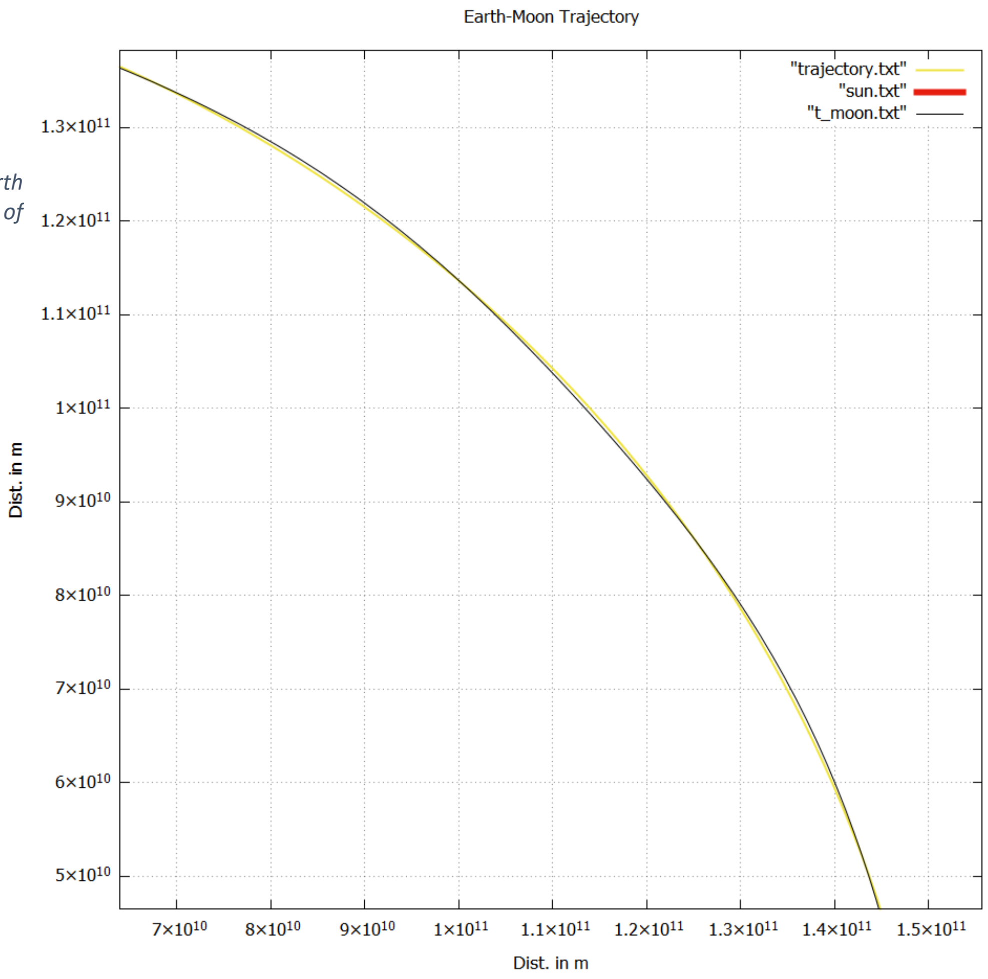


Figure 3: The plot shows the trajectories of earth and moon, focussing on the slight variation of moon's trajectory.

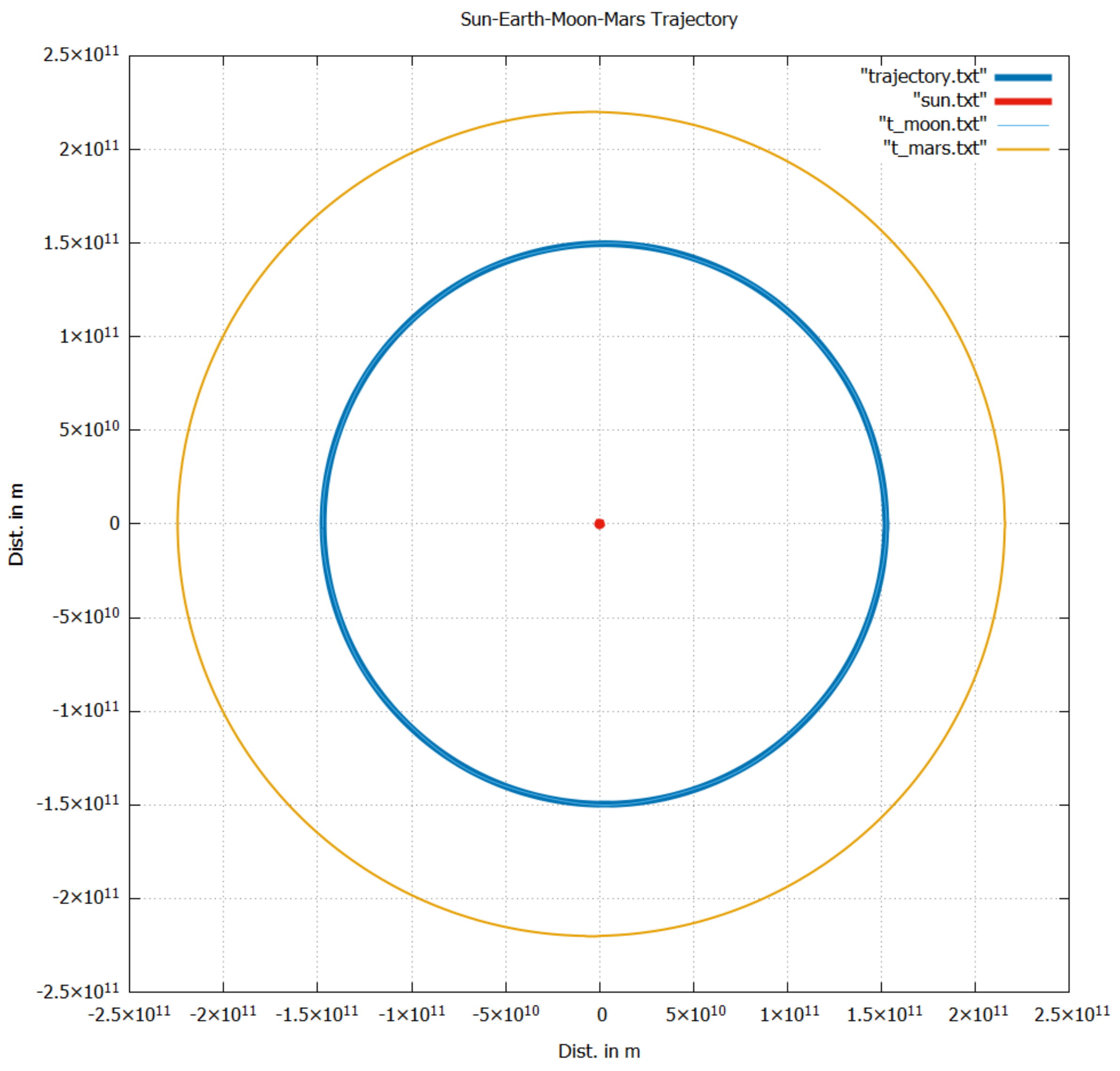


Figure 4: The above plot shows the orbits of Earth and Mars around the sun.

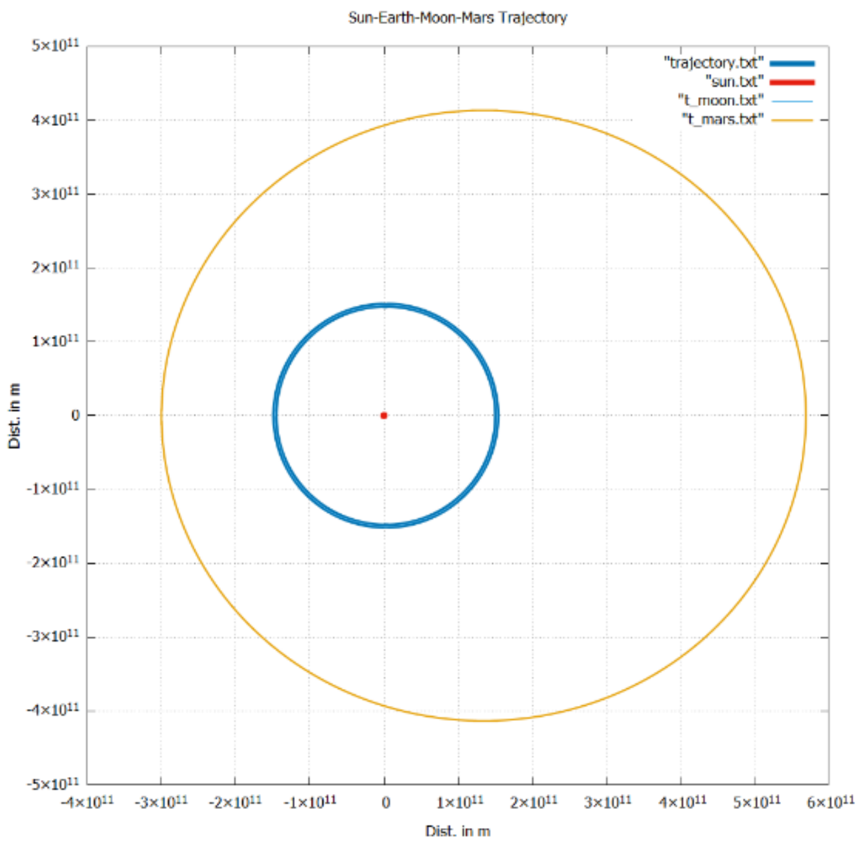


Figure 5: Trajectory change of Mars due to increased initial radius.

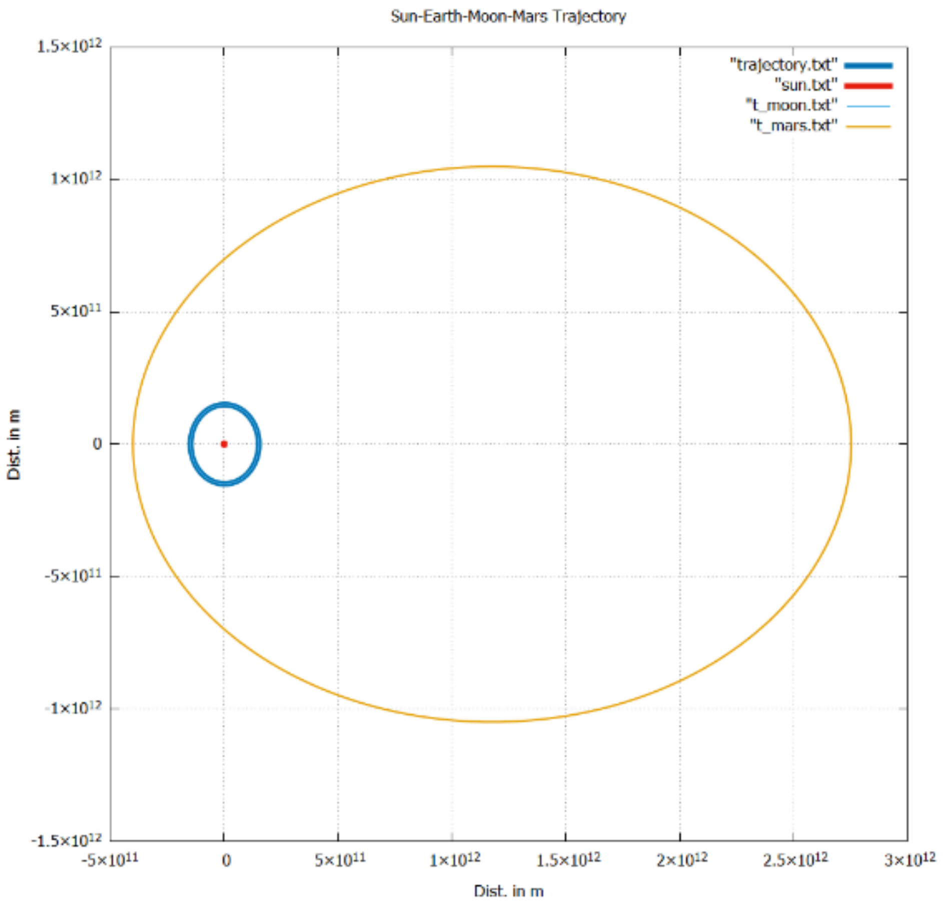


Figure 6: Trajectory change of Mars on further increasing initial radius.

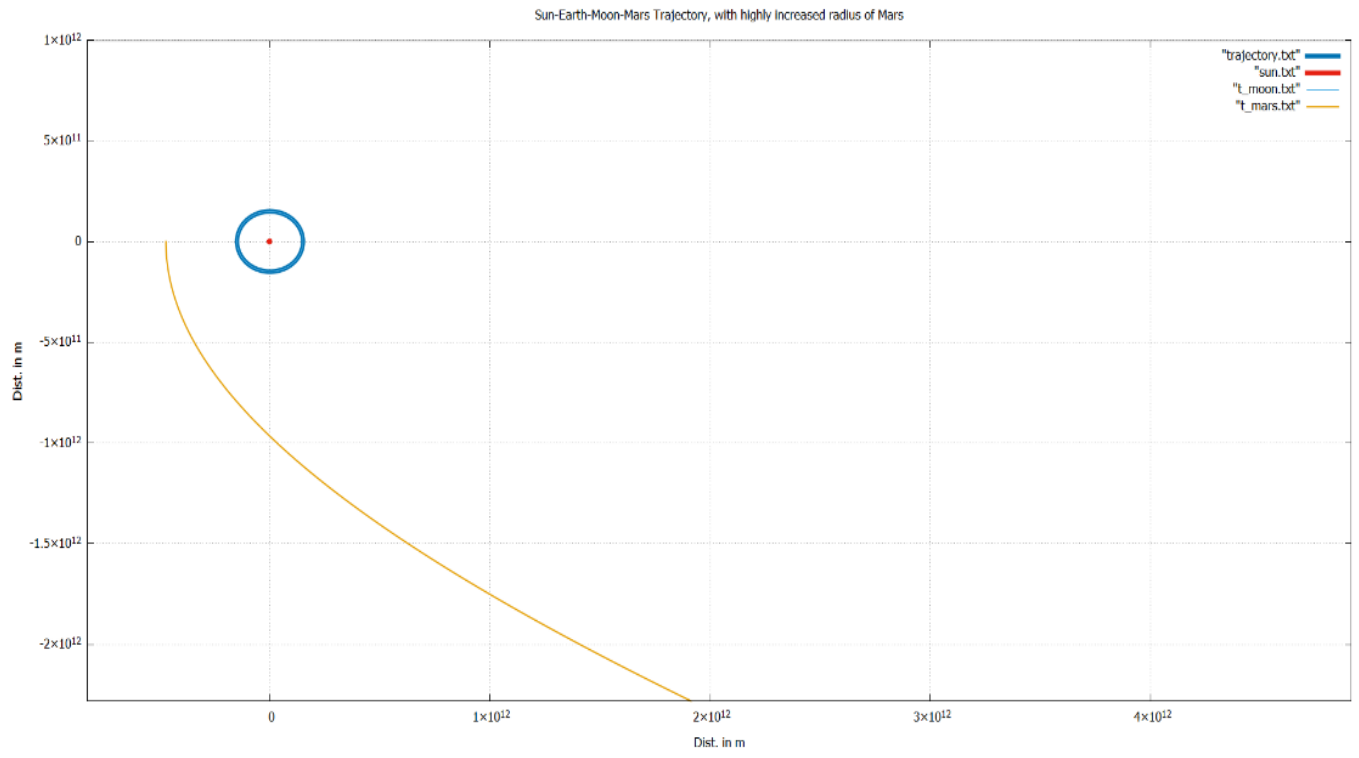


Figure 7: Mars breaks off from the gravitational pull of Sun at a certain increased value of radius.

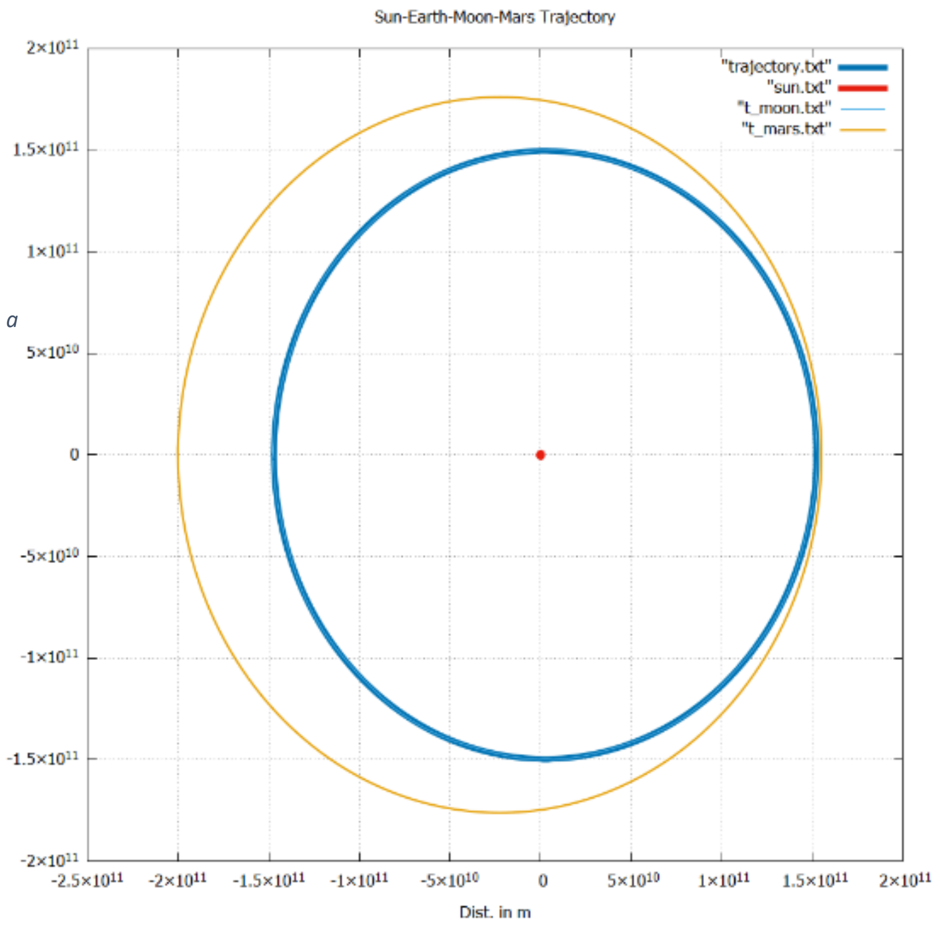


Figure 8: Trajectory change of Mars due to a decreased initial radius.

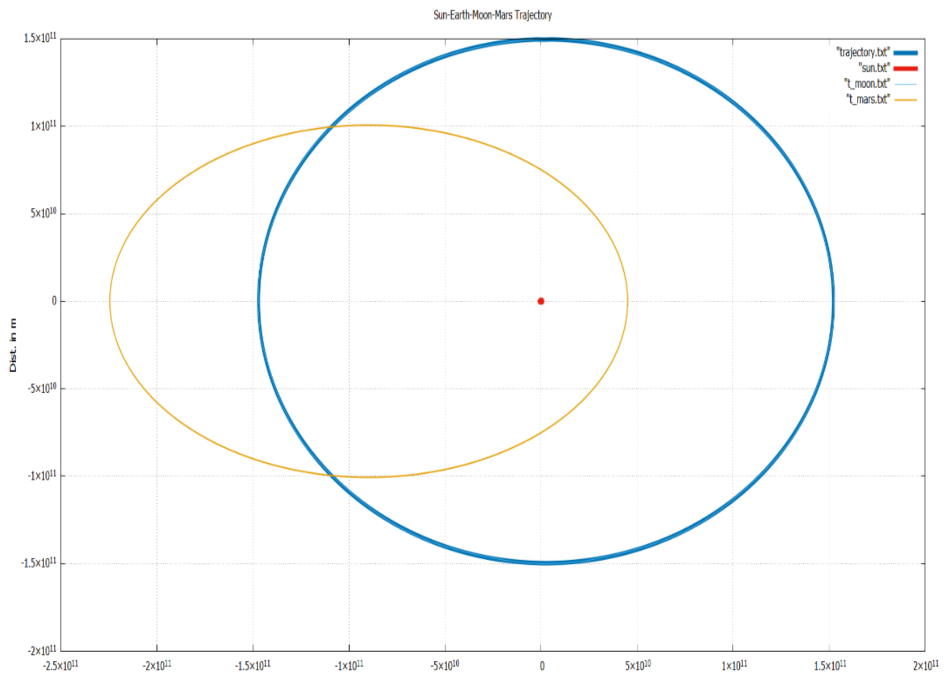


Figure 9: Trajectory change of Mars due to decreased velocity of the planet.

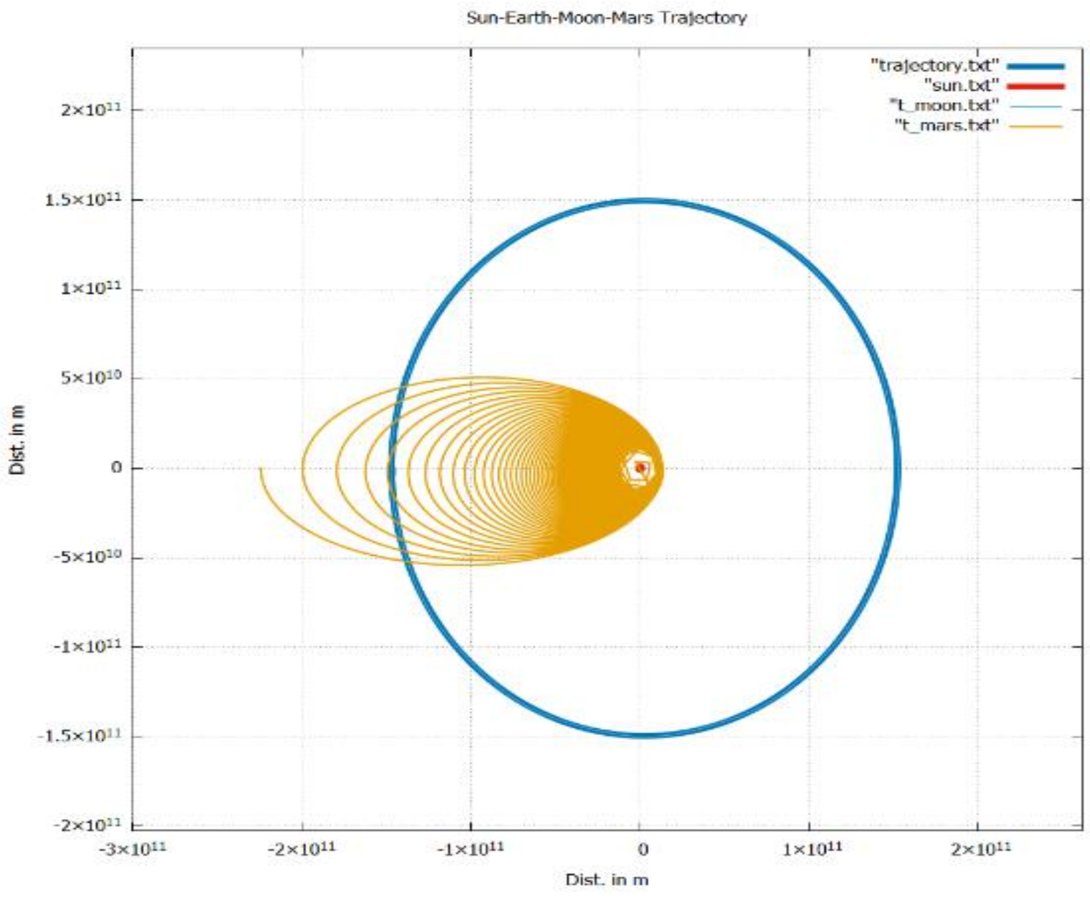


Figure 10: Velocity of Mars is decreased and at a certain value, the planet collapses into the sun.

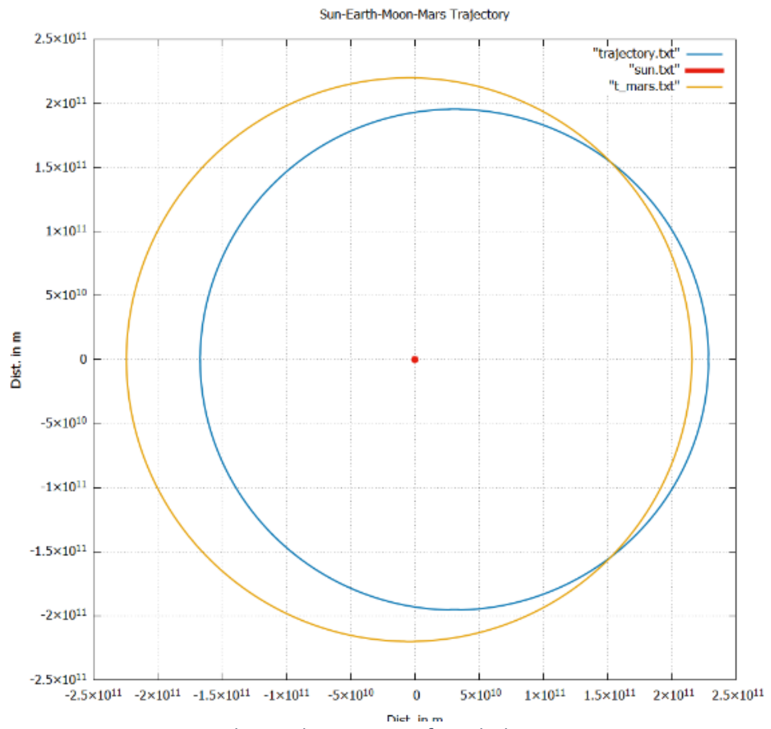


Figure 11: Changed trajectory of earth due to slightly decreased radius of orbit.

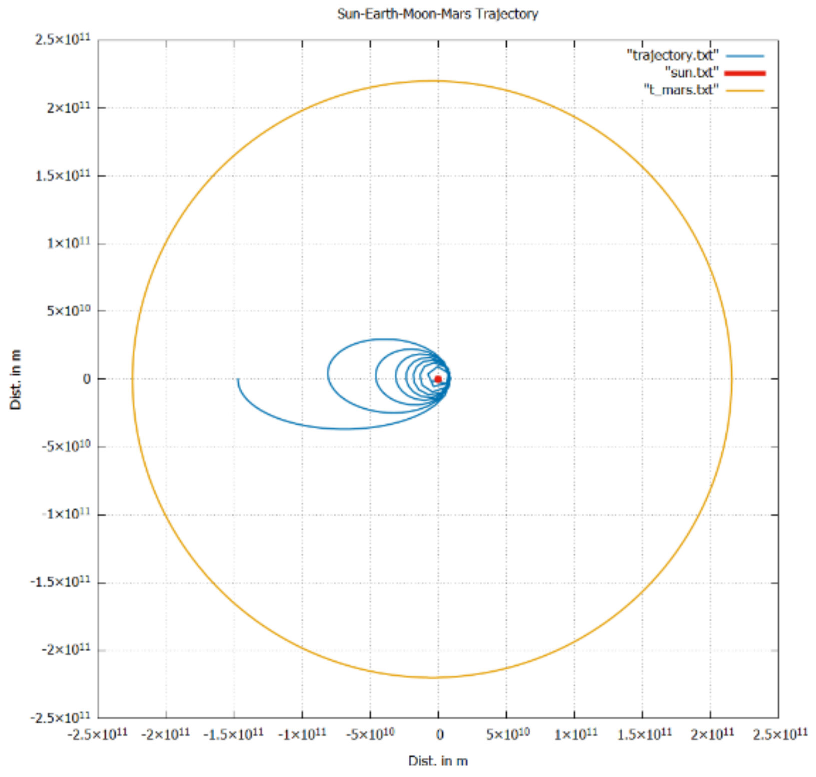


Figure 12: The Earth collapses into the Sun due to decreased orbital velocity

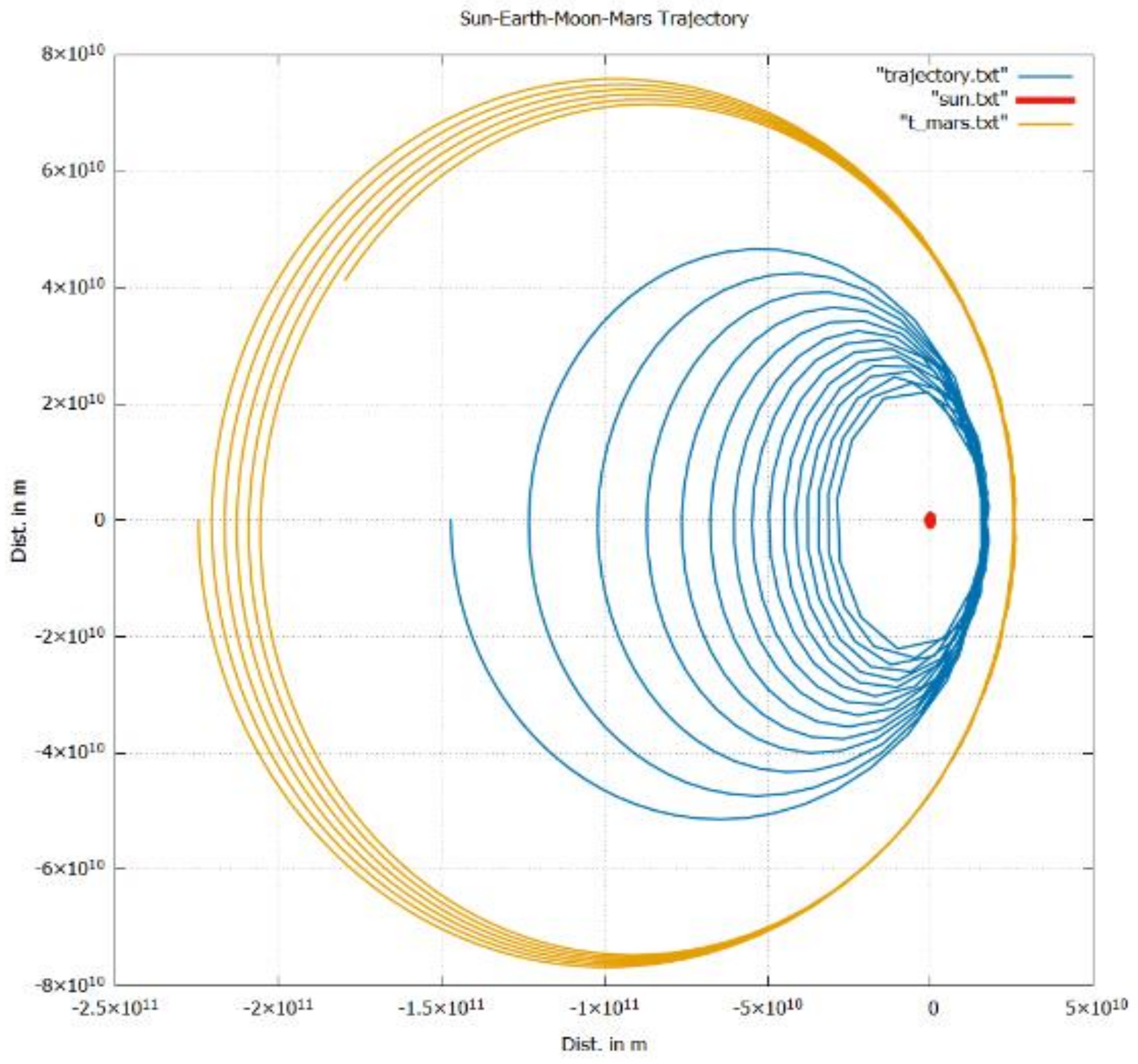


Figure 13: Both Mars and Earth collapses into the sun as the mass of the Sun is increased (God forbid).

The Source Code

```
#include <math.h>
#include <stdio.h>
#define Msun 1.989e30
#define Rsun 695700000.0
#define G 6.67408e-11
#define Mearth 5.972e24
#define Rearth 6378000.0


double ax(double x, double y) //accceleration of earth in x direction
{
double r=x*x+y*y;
double r1=r*sqrt(r);
return (G*Msun*(-1.0)*x)/r1;
}
double ay(double x, double y) //acceleration of earth in y direction
{
double r=x*x+y*y;
double r1=r*sqrt(r);
return G*Msun*(-1.0)*y/r1;
}


double axm(double x, double y) //acceleration of moon in x direction
{
double r=x*x+y*y;
double r1=r*sqrt(r);
return G*Mearth*(-1.0)*x/r1;

}
double aym(double x, double y) //acceleration of moon in y direction
{
double r=x*x+y*y;
double r1=r*sqrt(r);
return G*Mearth*(-1.0)*y/r1;
}

//________________________________________________________________________

double f(double x)
{
return (-1.0)*G*Msun*Mearth/x;
}
double simpson13(double func(double x),double a,double b,double n)
{
double x,h,I,S=0;
int i;
h=fabs(b-a)/n;
for(i=1;i<n;++i)
{
           x=a+i*h;
           if(i%2==0)
           S=S+2*func(x);
else
           S=S+4*func(x);
}
I=(h/3)*(func(a)+func(b)+S);
return I;
}
double trapezoidal(double func(double x),double a,double b,int n)
{
```

```
int i;
double x,h,I,S=0;
h=fabs(b-a)/n;
for(i=1;i<n;++i)
{
            x=a+i*h;
            S=S+func(x);
}
I=(h/2)*(func(a)+func(b)+2*S);
return I;
}

//________________________________________________________________________

double fn(double r,double E) //function of central potential
{
double l=Mearth*Rearth*30300.0; //angular momentum
//printf("%f          %f          %f\n",l,E,r);
return (1/(r*r*sqrt(2*Mearth*E/l+2*Mearth*G*Msun/(l*r)-1/(r*r))));
}
double gauss(double f(double r,double E),double r,double E,double a, double b) //gaussian quadrature
{
double x1,x2;
x1=((b-a)/2.0)*(1/1.73)+((b+a)/2);
x2=((b-a)/2.0)*(-1/1.73)+((b+a)/2);
return (b-a)/a*(f(x1,E)+f(x2,E));
}

//_____________________________________________________________________

void central() //evaluating the integral to get R vs theta graph
{
FILE*fp=NULL;
fp=fopen("central.txt","w");
double E,r,r0,rm;
printf("Enter the value of E:");
scanf("%lf",&E);
printf("Enter the lower and upper limit r0 & rm:\n");
scanf("%lf%lf",&r0,&rm);
for (r=r0;r<=rm;r=r+1.0) // varaying R from r0 to rm
{
            printf("%lf\t%lf\n",r,gauss(fn,r,E,r0,r));
            fprintf(fp,"%lf\t%lf\n",r,gauss(fn,r,E,r0,r));
}
}

//________________________________________________________________________


double energy (double x,double y,double vx, double vy)
{
double r=sqrt(x*x+y*y);
return 0.5*(vx*vx+vy*vy)-G*Msun/r;
}

//________________________________________________________________________

double rk()
{
FILE*fp=NULL;
fp=fopen("trajectory.txt","w");
            FILE*fpm=NULL;
            fpm=fopen("t_moon.txt","w");
            FILE*fpe=NULL;
            fpe=fopen("energy.txt","w");
double xm=-384400000.0;
double ym=0.0;
double vxm=0.0;
double vym=-1022.0;
double k1xm,k2xm,k3xm,k4xm,k1ym,k2ym,k3ym,k4ym,k1vxm,k2vxm,k3vxm,k4vxm,k1vym,k2vym,k3vym,k4vym;

double x=-147095000000.0;
double y=0.0;
double vx=0.0;
double vy=-30300.0;
double h =86400.0;
int m=0;
double k1x,k2x,k3x,k4x,k1y,k2y,k3y,k4y,k1vx,k2vx,k3vx,k4vx,k1vy,k2vy,k3vy,k4vy;
for(int n=0;n<=365;n++)
{
            fprintf(fp,"%f        %f\n",x,y);
            {
                        k1x=vx;
                        k1y=vy;
                        k1vx=ax(x,y);
                        k1vy=ay(x,y);

                        k2x=vx+k1vx*h/2.0;
                k2y=vy+k1vy*h/2.0;
                        k2vx=ax(x+k1x*h/2.0,y+k1y*h/2.0);
                k2vy=ay(x+k1x*h/2.0,y+k1y*h/2.0);

                        k3x=vx+k2vx*h/2.0;
                k3y=vy+k2vy*h/2.0;
                k3vx=ax(x+k2x*h/2.0,y+k2y*h/2.0);
                k3vy=ay(x+k2x*h/2.0,y+k2y*h/2.0);

                        k4x=vx+k3vx*h;
                k4y=vy+k3vy*h;
                k4vx=ax(x+k3x*h,y+k3y*h);
                k4vy=ay(x+k3x*h,y+k3y*h);

                x=x+(h/6.0)*(k1x+2.0*k2x+2.0*k3x+k4x);
                y=y+(h/6.0)*(k1y+2.0*k2y+2.0*k3y+k4y);
                vx=vx+(h/6.0)*(k1vx+2.0*k2vx+2.0*k3vx+k4vx);
                vy=vy+(h/6.0)*(k1vy+2.0*k2vy+2.0*k3vy+k4vy);

                        //__________________________________________________


                        fprintf(fpm,"%f       %f\n",xm+x,ym+y);
                        k1xm=vxm;
                        k1ym=vym;
                        k1vxm=axm(xm,ym);
                        k1vym=aym(xm,ym);

                        k2xm=vxm+k1vxm*h/2.0;
                        k2ym=vym+k1vym*h/2.0;
                        k2vxm=axm(xm+k1xm*h/2.0,ym+k1ym*h/2.0);
                        k2vym=aym(xm+k1xm*h/2.0,ym+k1ym*h/2.0);

                        k3xm=vxm+k2vxm*h/2.0;
                        k3ym=vym+k2vym*h/2.0;
                        k3vxm=axm(xm+k2xm*h/2.0,ym+k2ym*h/2.0);
                        k3vym=aym(xm+k2xm*h/2.0,ym+k2ym*h/2.0);
                        k4xm=vxm+k3vxm*h;
                        k4ym=vym+k3vym*h;
                        k4vxm=axm(xm+k3xm*h,ym+k3ym*h);
                        k4vym=aym(xm+k3xm*h,ym+k3ym*h);

                    xm=xm+(h/6.0)*(k1xm+2.0*k2xm+2.0*k3xm+k4xm);
                    ym=ym+(h/6.0)*(k1ym+2.0*k2ym+2.0*k3ym+k4ym);
                    vxm=vxm+(h/6.0)*(k1vxm+2.0*k2vxm+2.0*k3vxm+k4vxm);
                    vym=vym+(h/6.0)*(k1vym+2.0*k2vym+2.0*k3vym+k4vym);
            }
            //n++;
            double e=energy(x,y,vx,vy);
            //printf("%d          %f\n",n,e);
            fprintf(fpe,"%d       %f\n",n,e);
            //fprintf(fp2,"%f     %d\n",y,n);
            //fprintf(fp3,"%f     %d\n",vx,n);
            //fprintf(fp4,"%f     %d\n",vy,n);
}//while(n<365);
}

//__________________________________________________________________________

double rk_mars()
{
FILE*fp=NULL;
            fp=fopen("t_mars.txt","w");
double x=-224490000000.0;
double y=0.0;
double vx=0.0;
double vy=-24070.0;
//double a[4];
//int n=0;
double h =88620.0;
double k1x,k2x,k3x,k4x,k1y,k2y,k3y,k4y,k1vx,k2vx,k3vx,k4vx,k1vy,k2vy,k3vy,k4vy;
for(int n=0;n<=687;n++)
{
fprintf(fp,"%f        %f\n",x,y);
k1x=vx;
k1y=vy;
k1vx=ax(x,y);
            k1vy=ay(x,y);

k2x=vx+k1vx*h/2.0;
            k2y=vy+k1vy*h/2.0;
k2vx=ax(x+k1x*h/2.0,y+k1y*h/2.0);
            k2vy=ay(x+k1x*h/2.0,y+k1y*h/2.0);

k3x=vx+k2vx*h/2.0;
            k3y=vy+k2vy*h/2.0;
            k3vx=ax(x+k2x*h/2.0,y+k2y*h/2.0);
            k3vy=ay(x+k2x*h/2.0,y+k2y*h/2.0);

k4x=vx+k3vx*h;
            k4y=vy+k3vy*h;
            k4vx=ax(x+k3x*h,y+k3y*h);
            k4vy=ay(x+k3x*h,y+k3y*h);

            x=x+(h/6.0)*(k1x+2.0*k2x+2.0*k3x+k4x);
            y=y+(h/6.0)*(k1y+2.0*k2y+2.0*k3y+k4y);
            vx=vx+(h/6.0)*(k1vx+2.0*k2vx+2.0*k3vx+k4vx);
            vy=vy+(h/6.0)*(k1vy+2.0*k2vy+2.0*k3vy+k4vy);
}
}
```

```
//_______________________________________________________________________

void sun()
{
FILE*fp=NULL;
           fp=fopen("sun.txt","w");
for(int i=1;i<=365;i++)
{
           double x=Rsun*cos(i);
           double y=Rsun*sin(i);
           //printf("%f          %f\n",x,y);
           fprintf(fp,"%f        %f\n",x,y);
}
}


//========================================================================

int main()
{
rk();
sun();
rk_mars();
central();
int n=2;
double a=sqrt(146000000000),b=sqrt(152000000000),I,A;
//printf("%f %f",a,b);
do
{
           I=A;
           n=n+2;
           A=simpson13(f,a,b,n);
}while(fabs(A-I)>=0.00001);
printf("\nThe energy integral using Simpson's Rule is  : %lf\n",A);
n=2;
do
{
           I=A;
           n++;
           A=trapezoidal(f,a,b,n);
}while(fabs(A-I)>=0.00001);
printf("The energy integral using Trapezoidal Rule is: %lf",A);
}



/*
OUTPUT

The energy integral using Simpson's Rule is  :
-159639247768960310000000000000000000000000000.000000
The energy integral using Trapezoidal Rule is:
-159639247770203150000000000000000000000000000.000000
--------------------------------
Process exited after 0.2448 seconds with return value 0
Press any key to continue . . .
*/
```

Functions used

Name	**Return Type**	**Function**
ax()	*double*	computing acceleration of earth in x direction
ay()	*double*	computing acceleration of earth in y direction
axm()	*double*	computing acceleration of moon in x direction
aym()	*double*	computing acceleration of moon in y direction
f()	*double*	containing the function to integrate
simpson13()	*double*	applying Simpson's 1/3rd rule
trapezoidal()	*double*	applying Trapezoidal rule
fn()	*double*	containing the central potential function
gauss()	*double*	applying gaussian quadrature
central()	*void*	evaluating the integral stored in fn() to obtain R vs Theta graph
energy()	*double*	calculating the energy of the system
rk()	*double*	applying RK4 method of evaluation to obtain the trajectories of earth and also the trajectory of moon associated with that of earth
rk_mars()	*double*	applying RK4 method to compute the trajectory of mars around the sun
sun()	*void*	plotting the sun
main()	*int*	the main function, to call all the other functions

Inference

The program can thus compute the trajectories of the planets and their respective satellites and help us understand the nature of their orbits. We have also computed those trajectories by varying:

- The radius of planets
- The orbital velocities of the planets
- The mass of the sun itself

This provides us with a better understanding of how planetary motions might work.

We have also evaluated the total energy of the system, as well as the potential of it.

Discussion

While the project provides us with trajectories which are quite good, it however is restricted to a special condition of the orbits being spherical.

Also, the energy calculated has discrepancies in it. The workings of the Numerical Methods are correct but there could lie some fallacies in its implementation to the associated formulae. The range we are dealing with in our problem is in millions of kilometers and yet, results are yield in range of meters, which is wrong! Critically analyzing and further research on the topic might help in providing a feasible solution.

Another aspect which I have not incorporated into this project is the fact that in reality, the planetary systems are not two-body systems, but are affected by forces of all the other bodies surrounding it. In the small solar system which I have created, consisting of the sun, the earth, the moon and the mars only, the two planets do not interact with each other gravitationally. The incorporation, however, is an easy one. The first of the few steps would be to use the mass of all the involved bodies instead of just the sun, in our calculation of components of acceleration (in *ax()* function and *ay()* function). That would result in slight change in the trajectory plot in the event that the earth and the mars came too close to each other.

Bibliography

- *www.google.com*
- *en.wikipedia.org*
- *www.stackoverflow.com*
- *Classical Mechanics, by Herbert Goldstein*
- *An Introduction to Mechanics, by Kleppner & Kolenkow*
- *Classical Mechanics and Properties of Matter, by A.B. Gupta*

Thank You