Three solutions for the Three-Body Problem: A comparison of computational methods in Orbital Mechanics

Bryan Isherwood

May 2, 2016

Abstract

In this project we numerically solve the 3-body orbital equations using three approximation methods: Velocity Verlet, Fourth order Runge-Kutta, and the Parker-Sochacki method. Using these algorithms we analyze the movements of the Sun, Earth, and Jupiter around their respective orbits. We find that Parker-Sochacki performs equivalently to Runge-Kutta in numerical precision while the Velocity Verlet algorithm cannot compete in numerical precision to either method.

1 Introduction

With the invention of Newton's calculus in the late 1600's, the problem of solving the dynamics of objects seemed to be solved. Although it became quickly apparent that there are some problems whose solutions are too difficult to solve in closed form. The presentation of the solution to this problem by Brook Taylor in 1715 opened the doors to computational mathematics and physics. If the problem could not be solved, its solution could at least be approximated. Since that time, several approximations methods appeared that allow scientists and mathematicians to preform these often tedious calculations. The velocity Verlet method, discovered many times throughout history, is a fast and simple method that repeatedly uses second order approximations to determine the trajectory of objects. The Runge-Kutta methods, invented in the early 1900's, uses successive approximations between half integer steps to create a fourth order approximation, that is error that truncates like the h^4. While both these methods provide fast and accurate solutions, they only use careful manipulation of the first order polynomial approximation of Taylor's series. While these approximations were being developed, another method became buried under the sands of time.

In the 1900's Émile Picard proposed a method of approximating functions through successive approximations using the initial conditions of Ordinary differential equations. While this method worked perfectly for many functions, it failed to properly account for quotients. The method became a footnote to existence theorems as a result. In 1988 the mathematicians G. Edgar (Ed) Parker and James Sochacki improved upon Picard's approximation, creating the so called Parker-Sochacki Theorem (PST). It removed the barrier of quotient and special function-like independent variables by introducing the concept of Polynomial Projection. By projecting all functions into the form of polynomial equations we can generate Taylor series, centered around the origin, for any function. Applying this to numerical computing allows for a fast and lightweight method of approximating analytic functions.

This paper will investigate the theory behind Picard's approximation and PST as well as its application to the orbital problem. We will then formulate an algorithm for calculating the orbits of the two and three body systems analyzed in [1]. Using this algorithm, we will compare the results to the Velocity Verlet (VV) and Fourth Order Runge-Kutta algorithms and determine which is greater in accuracy and if any is outstandingly faster than the others.

2 Theory

The theory of orbital mechanics and Newtonian gravity appears in [1] and will not be discussed here. Instead this paper will focus on the theory behind Picard Iterations and the improvement that Parker and Sochacki made upon the original method.

2.1 Picard Iteration

To know and understand Picard's iteration method is to understand the fundamental theorem of calculus (FTC) itself. To begin with, let us look at FTC:

$$\begin{aligned} F(x) &= F(0) + \int_0^x f(x)dx \\ where: \ \frac{dF}{dx} &= f(x) \end{aligned} \tag{1}$$

The meaning of this is simple: The integral of a function *f(x)* is equivalent to the anti-derivative evaluated at the end points of the domain of interest. In the guise of differential equation theory, this theory reads as: the solution of the differential equation *f(x) = F'(x)* is given by the sum of the initial condition and the anti-derivative of *f(x)*. If we use the latter definition, we can ask the question: if we cannot find our anti-derivative, how can we approximate it from eq. (1)? Picard answered this question for autonomous differential equations or those of the form:

$$F'(x) = f(x, F(x)) \tag{2}$$

and making the simple assumption of:

$$f(x, F(x)) \approx f(0, F(0)) \tag{3}$$

Using this assumption we can begin an iterative process culminating in a series of approximations:

$$\begin{aligned}
&F_0(x) = F(0)\\
&F_1(x) = F(0) + \int_0^x f(0, F_0(0))dx = F(0) + f(0, F(0))x\\
&F_2(x) = F(0) + \int_0^x f(0, F_1(0))dx = F(0) + f(0, F(0))x + \frac{f(0, F(0))x^2}{2}\\
&F_3(x) = F(0) + \int_0^x f(0, F_2(0))dx = F(0) + f(0, F(0))x + \frac{f(0, F(0))x^2}{2} + \frac{f(0, F(0))x^3}{6}\\
&\qquad\vdots\\
&F_m(x) = F(0) + \int_0^x f(0, F_{m-1}(0))dx
\end{aligned} \tag{4}$$

where m is where we have "truncated" our series. As an example, suppose *F(x)* = e^x, giving *f(x,F(x)) = F(x)*. Iterating this produces:

$$\begin{aligned}
&F_0(x) = 1\\
&F_1(x) = F(0) + \int_0^x f(0, F_0(0))dx = 1 + \int_0^x dx = 1 + x\\
&F_2(x) = F(0) + \int_0^x f(0, F_1(0))dx = 1 + \int_0^x (1 + x)dx = 1 + x + \frac{x^2}{2}\\
&F_3(x) = F(0) + \int_0^x f(0, F_2(0))dx = 1 + \int_0^x (1 + x + \frac{x^2}{2})dx = 1 + x + \frac{x^2}{2} + \frac{x^3}{6}\\
&\qquad\vdots\\
&F_m(x) = F(0) + \int_0^x f(0, F_{m-1}(0))dx = 1 + \sum_m \frac{x^m}{m!}
\end{aligned} \tag{5}$$

which is the Maclauren series for the exponential function. While this is a surprisingly simple way of determining this series, it does have the flaw that some functions will simply not allow their derivatives to be in the autonomous form (e.g. *F(x) = ln(1-x)*. At this point, many mathematicians looked at Picard's iterative method as an interesting formalism with limited use and for 80 years it remained this way. Until Dr.s Parker and Sochacki discovered a simplification method that removed these difficulties: Polynomial Projection.

2.2 Parker-Sochacki Theorem

The Parker-Sochacki Theorem can be summarized as such: Picard Iteration will determine the Maclaurin Series for ANY differential equation, so long as that equation can be projected into a polynomial of some kind. This idea is best understood through example: so let us suppose we want to determine the series approximation of *ln(1-x)*. First off, we notice that:

$$f(x) = \frac{-1}{1-x} \tag{6}$$

At first all hope seems lost at using the iterative method, but suppose we now define our derivative as the function $G(x) = \frac{1}{1-x}$ giving:

$$g(x) = G'(x) = -\frac{1}{(1-x)^2} = -G(x)^2 \tag{7}$$

assuming that we use *G(x)* as the polynomial form of *f(x)* then we have created a coupled set of polynomial projections in G. Applying both of these differential equations, (6) and (7), to equation (4) will produce the Maclaurin series for both $ln(1-x)$ and $\frac{1}{1-x}$, with the initial conditions $F(0) = 0$ *and* $G(0) = 1$.

There are certain intricacies that must be pointed out before one begins this calculation though. First: The standard PST precludes the use of definite integrals with a lower bound other than *0*, although once a Macluarin series has been generated it is simple work to shift the function by composing the resulting function with *x = x'-c*, where c is a constant and x and x' share the same domain. Second: The iteration method is only accurate to the m^{th} term in the series. This is important when evaluating (7) as the squared nature of the function will generate polynomials of orders greater than *m-1*, though this problem is solved if we simply truncate the series at the m^{th} term after every iteration.

In practice, this theorem is used to determine the coefficients of the Taylor series of interest. Once the coefficients are know a simple "for" loop will evaluate the function to any desired degree of accuracy. There are some practical disadvantages to this method though. If a projection cannot be determined, then this method falls apart very quickly. There is also an exceptional amount of work necessary to set up this method when compared to the simplicity of VV and RK4 as determining projections and initial conditions is not necessarily trivial for all functions.

2.3 Projection of the Orbital Equations

In applying this method we need to find a way to project our original differential equation:

$$m_1\ddot{\mathbf{r}} = -\frac{Gm_2m_1}{r^2}\hat{\mathbf{r}} \tag{8}$$

into a polynomial form. In this equation allow G to be gravitational force constant, m_2 is the mass of the object creating a gravitational potential felt by m_1, and r is the relative distance between the two objects. If we also allow:

$$GM_{sun} = 4\pi^2\frac{AU^3}{yr^2} \tag{9}$$

then our differential equation becomes:

$$\ddot{\mathbf{r}} = -\frac{4\pi^2 m}{r^2}\hat{\mathbf{r}} \tag{10}$$

where $\boldsymbol{r}$ is of the order of astronomical units and m is the mass ratio between the planet of interest and the Sun. Creating a set of differential equations over velocity and positions gives the coupled equations of:

$$\begin{aligned}\dot{\mathbf{r}} &= \mathbf{v}\\ \dot{\mathbf{v}} &= -\frac{4\pi^2 m}{r^3}\mathbf{r}\end{aligned} \tag{11}$$

By defining the new variable $u(r) = \frac{1}{r}$ we can rewrite (11.b) as:

$$\dot{\mathbf{v}} = -\frac{4\pi^2 m}{r^3}\mathbf{r} = -4\pi^2 m u^3 \mathbf{r} \tag{12}$$

with the velocity equation now in the form of a product of polynomials, we now need to add an equation for u:

$$\dot{u} = \frac{d}{dt}(\frac{1}{r}) = -\frac{1}{r^3}\sum_{i=1}^{3} x_i v_i = -u^3 \sum_{i=1}^{3} x_i v_i \tag{13}$$

where i runs over our position coordinate space. For current purposes, this equation assumes only two body system with the significantly larger mass existing at the origin. Our system now consist of 7 coupled equations and with this system we can begin calculating our series expansion.

3 Algorithm

In order to apply the PST, it is necessary to know how each of the of the series expansion relate to one another. Considering the relationship between velocity and position, it is clear to see that:

$$r_{ij} = r_{ij0} + \sum_{l=1}^{n} r_{ijl} * t^l = r_{ij0} + \sum_{l=1}^{n} v_{i,j,l-1} * t^l \tag{14}$$

for the nth iteration of the ith coordinate of the jth planet we get the relationship:

$$r_{ijn} = \frac{v_{i,j,n-1}}{n} \tag{15}$$

The coefficient expansion for the velocity relationship is more complicated but can be shown to be:

$$v_{ijn} = \sum_{k=1}^{N_p} m_k \sum_{l=0}^{n-1} (r_{ijl} - r_{ikl}) \frac{(u_{i,j,n-l-1})^3}{n} \tag{16}$$

where $\mathrm{N_p}$ is the total number of planets in the system, j is the planet of interest and k is the index for the other planets in the system. In order to properly calculate the the cubic expansion, we break it down into two separate equations:

$$\begin{aligned} u^2_{jkn} &= \sum_{l=0}^{n} u_{jkn} u_{j,k,l-1} \\ u^3_{jkn} &= \sum_{l=0}^{n} u^2_{jkl} u_{j,k,n-1} \end{aligned} \tag{17}$$

In order to calculate A we can use:

$$A_{jkn} = \sum_{l=1}^{n} \sum_{i=1}^{3} (r_{ijl} - r_{ikl})(v_{i,j,n-l} - v_{i,k,n-l}) \tag{18}$$

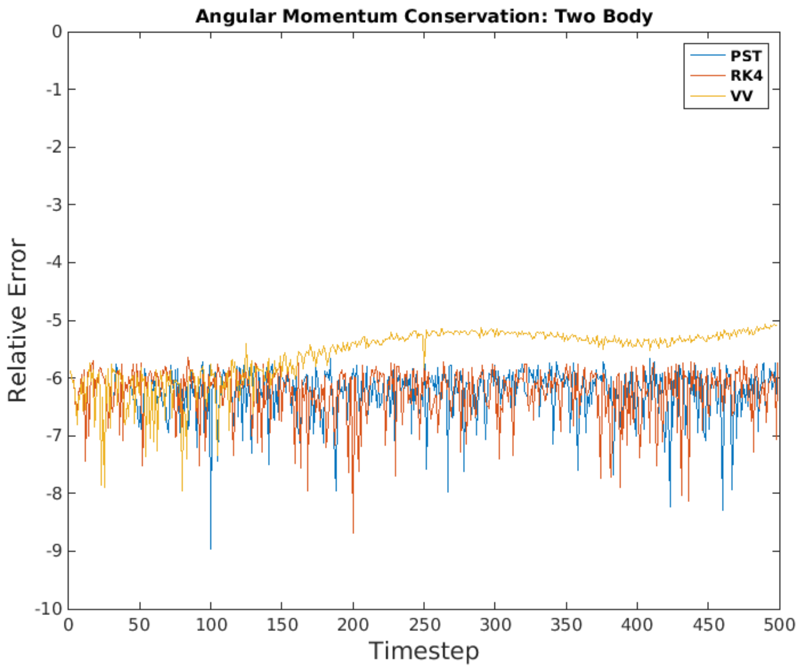

Figure 1: This plot shows the relative error of the Earth's angular momentum in a two body system using an orbital period of one year for each of the methods used: VV, RK4, and PST.

using this we can finally calculate u:

$$u_{jkn} = -\sum_{l=1}^{n-1} u_{jkl}^3 A_{j,k,n-l} \tag{19}$$

Following these calculations will generate the Maclaurin coefficients for the position and velocity vectors. If $j = k$ it is best to allow $u_{jkn} = 0$. It is then a simple process to determine the trajectory of each of the planets by the middle term in equation (14). A full description for this algorithm can be found in [2]. Each summation should be evaluated by means of a "For" loop. Attached to the end of this paper is the full program for PST, VV, and RK4 used in the project and written in C++.

4 Results

For the first part of this project, the circular orbits (Earth-Sun system) were calculated for each method provided (VV, RK4, and PST). Due to the circular nature of the solution, it was determined that the best way to calculate the error in each method was to compare the x- and y-coordinates to the cosine and sine functions respectively for one orbit around the Sun. The PST generated an 80th order polynomial for this approximation. In order to compare the accuracy

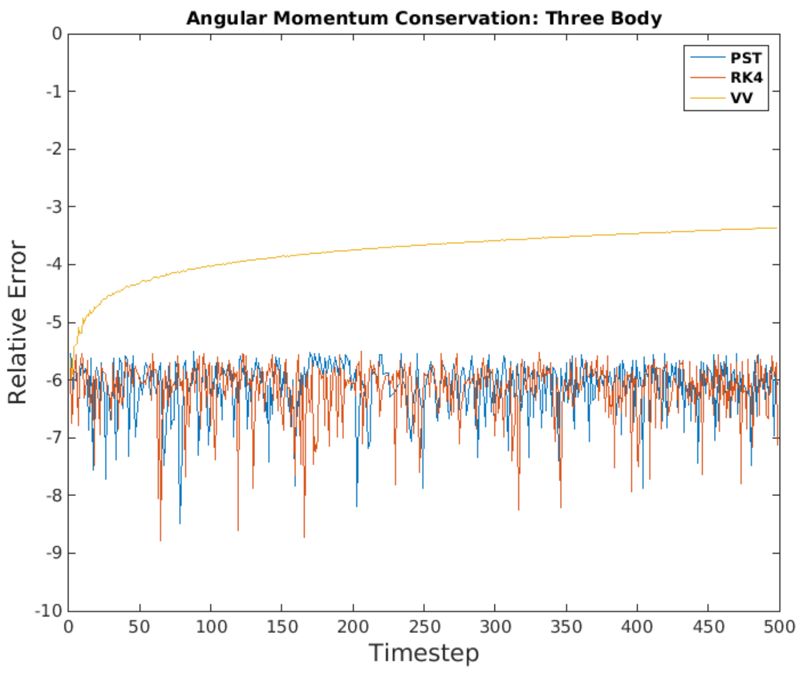

Figure 2: This plot shows the relative error of the Earth's angular momentum in a three body system using an orbital period of one year for each of the methods used: VV, RK4, and PST.

of each of these models, the magnitude of the total angular momentum of the system was calculated at each time step and compared to the magnitude of the original angular momentum of the system. To do this, the relative error in the angular momentum was calculated at each point using the formula:

$$relerror = Log_{10}(|\frac{L_{tot}(0) - L_{tot}(t)}{L_{tot}(0)}|) \tag{20}$$

where $\mathrm{L_{tot}}$ denotes the total angular momentum of the system as given by:

$$L_{tot} = \sum_{i=0}^{N_p} (x_i P_{y_i} - y_i P_{x_i}) \tag{21}$$

considering that the angular momentum only exists in the $\hat{z}$ direction. It is also note worth to mention that the simple two body system with a non-moving Sun was not done for this project. This was done to accurately compare PST to the other two methods. Because PST uses Taylor series to calculate orbital paths, it automatically generates the Taylor series for a moving sun as well. It accommodate this, the two body system now assumes a moving Sun. Figure (1) shows the relative error of the angular momentum of Earth, over a one year orbit, for each method of approximation. It was chosen to use only one year as a time constraint as comparing the models for longer than one year proved to be complicated due to how the time steps were used in each calculation. Future work will demonstrate the error as time evolves beyond one year. It is clear to see that for the evolution of this system PST and RK4 are equivalent methods, having a relative error that has a average value between -6 and -7. The Verlet algorithm on the other hand quickly diverges away from the initial value as time goes on.

We can now add in Jupiter to the same system. For this part of the project, please refer to [2] in order to see how the initial conditions for the system were determined. In this case the results are quite interesting.

It is clear to see that for this system, PST and RK4 once again tie for computational accuracy. While there are large variations in the relative error, it's average value stays constant around -6 to -7. VV on the other hand, have errors that grow as time evolves. VV has a fast rise that slows down for large times. While sub-second time calculations could not be taken, it is important to know that each calculation still completed in under one second and used 500 iteration points.

5 Discussion

All three approximation methods preformed adequately for each of their respective intervals. In regards to accuracy, VV preformed the worst with RK4 and PST performing the best. In figures (1) and (2), PST and RK4 have the same relative error form, while VV has a more stable curve that quickly diverges away from the initial value. The error in each of these calculations is also determined from different numerical sources. VV and RK4 require a defined time step, whose value then determines how large the relative error becomes. In principle, the smaller the time step, the more accurate the calculation, up to a limit. PST on the other hand does not require small time steps in order to achieve accuracy. It only requires that more and more terms in the Taylor series be present. When a specific coordinate needs to be calculated, one must simply apply the time of interest to the generated series calculation. This means that the accuracy of PST is primarily dependent on the numerical precision of each term generated. For this calculation, it was found that numerical precision was still non-zero at the 80th iteration. Although, due to the final output being limited to six digits of accuracy, it is likely that the entire series did not add more precision to the calculation.

6 Conclusion

While fast, the Verlet algorithm does not hold up in precision calculations. It's relative error grows over time and is heavily dependent on the time step size. Runge-Kutta and Parker-Sochacki performed on equal footing. For the test provided, neither method completed the calculation with objectively better precision or time. In the future, better timing calculations will be preformed on each method and an expansion beyond the one-year time interval will be accomplished.

7 References

[1]B. Isherwood. Comparison of the Orbits of Heavenly bodies using Velocity Verlet and Fourth Order Runge-Kutta Computational Methods

[2] J.W. Rudmin. Application of the Parker-Sochacki Method to Celestial Mechanics. Department of Physics, James Madison University, March 1998

[3] M. Hjort-Jensen. Computational physics, lecture notes spring 2016. Department of Physics, Michigan State University, 2016.

8 Appendix

```
#include <iostream>
#include <string>
#include <cmath>
#include <fstream>
#include <cstdlib>
#include <new>
#include <vector>
#include "planet.h"
#include "lib.h"
#include "time.h"


using namespace std;


 void RK4(){

    double **R,**X,**Y,**Xhalf,**Yhalf,**Rhalf;
    double **Vxhalf,**Vyhalf,**Vx,**Vy;
   // double *vx,*vy,*x,*y,*r;
    //double *vxhalf,*vyhalf,*xhalf,*yhalf,*rhalf;
    double
        gravnowx,gravnowy,gravnextx,gravnexty,gravhalfx,gravhalfy,tmax,t0,h,h2,pi,gravnow,gravnext,t
    double *kx1,*kx2,*kx3,*kx4,*kvx1,*kvx2,*kvx3,*kvx4;
    double *ky1,*ky2,*ky3,*ky4,*kvy1,*kvy2,*kvy3,*kvy4;
    int i,j,n,k,m;
    t0 = 0.0;
    tmax = 1.0;
    n = 500;
    h = (tmax - t0)/(n+1);
    h2 = h*h;
    pi = acos(-1.0);
    clock_t start, finish;
  /*  r = new double[n];
    kvx1 = new double[n];
    kvy1 = new double[n];
    kx1 = new double[n];
    ky1 = new double[n];

    rhalf = new double[n];
    vxhalf = new double[n];
    vyhalf = new double[n];
    xhalf = new double[n];
    yhalf = new double[n];*/


       kvx1 = new double[3];
       kvy1 = new double[3];
       kx1 = new double[3];
       ky1 = new double[3];
       kvx2 = new double[3];
       kvy2 = new double[3];
       kx2 = new double[3];
       ky2 = new double[3];
       kvx3 = new double[3];
       kvy3 = new double[3];
       kx3 = new double[3];
       ky3 = new double[3];
       kvx4 = new double[3];
       kvy4 = new double[3];
       kx4 = new double[3];
       ky4 = new double[3];

    R = (double **) matrix(n,3,sizeof(double));
    X = (double **) matrix(n,3,sizeof(double));
    Y = (double **) matrix(n,3,sizeof(double));
    Rhalf = (double **) matrix(n,3,sizeof(double));
    Xhalf = (double **) matrix(n,3,sizeof(double));
    Yhalf = (double **) matrix(n,3,sizeof(double));

    Vx = (double **) matrix(n,3,sizeof(double));
    Vy = (double **) matrix(n,3,sizeof(double));
    Vxhalf = (double **) matrix(n,3,sizeof(double));
    Vyhalf = (double **) matrix(n,3,sizeof(double));



 //   planet Earth(0.000003,-9.687087657503362E-01,
     -2.304478999525212E-01,0.0,3.715293823075145E-03*365.0,-1.679555302972549E-02*365.0,0.0);
 //   planet
     Sun(1.0,0.00510182404,0.0008987727027,0.0,-0.0004829298902,-0.002481983714,0.0);
     //(mass,x,y,z,vx,vy,vz)
 //   planet
     Jupiter(.000954,-5.344777687381148,.9428344302147893,0.0,-1.398575965353750E-03*365.0,-7.075019
    //planet all_planets[3] = {Sun,Earth,Jupiter};
planet Earth(0.000003,1.0000,0,0,0,2*pi,0);
planet Sun(1.0,0,0,0,0,0,0);
//planet CM(0,0,0,0,0,0,0);
//planet Jupiter(0.000954,5.2,0,0,0,-2*pi/sqrt(5.2),0);
planet all_planets[2] = {Sun,Earth};//,Jupiter};
m = 2;


/* x[0] = 1.0;
vy[0] = 2*pi+2.5885754;
```

```
vx[0] = 0.0;
y[0] = 0.0;*/

for(i=0;i<m;i++){
X[0][i] = all_planets[i].position[0];
Y[0][i] = all_planets[i].position[1];
Vx[0][i] = all_planets[i].velocity[0];
Vy[0][i] = all_planets[i].velocity[1];

}
start = clock();
    //Runge-kutta--------------------------------------------------------------------------------
for(i=0;i<n-1;i++){

   //First
    for(j=0;j<m;j++){
    R[i][j] = sqrt(X[i][j]*X[i][j] + Y[i][j]*Y[i][j]);
    gravnowx = 0.0;
    gravnowy = 0.0;
       for(k=0;k<m;k++){

        if(k!=j){

           gravnowx +=
               4*pi*pi*(X[i][j]-X[i][k])*all_planets[k].mass/pow(pow(X[i][j]-X[i][k],2.0)+pow(Y[i][j]
           gravnowy +=
               4*pi*pi*(Y[i][j]-Y[i][k])*all_planets[k].mass/pow(pow(X[i][j]-X[i][k],2.0)+pow(Y[i][j]
           else{
           gravnowx+= 0.0;
           gravnowy+= 0.0;
}          }



    gravnow = 4.0*pi*pi/(R[i][j]*R[i][j]*R[i][j]);


    kvx1[j]= gravnowx;
    kvy1[j] = gravnowy;
    Xhalf[i][j] = X[i][j] + h/2.0*Vx[i][j];
    Yhalf[i][j] = Y[i][j] + h/2.0*Vy[i][j];
    Vxhalf[i][j] = Vx[i][j] - kvx1[j]*h/2.0;
    Vyhalf[i][j] = Vy[i][j] - kvy1[j]*h/2.0;
   // cout << kvx1[j] << endl;
    //test = gravnow*X[i][j];
    //cout << test << " " << kvx1[1] << endl;
}
    //Second

    for(j=0;j<m;j++){
    R[i][j] = sqrt(X[i][j]*X[i][j] + Y[i][j]*Y[i][j]);
    gravhalfx = 0.0;
    gravhalfy = 0.0;
       for(k=0;k<m;k++){
        if(k!=j){

           gravhalfx +=
               4*pi*pi*(Xhalf[i][j]-Xhalf[i][k])*all_planets[k].mass/pow(pow(Xhalf[i][j]-Xhalf[i][k],
           gravhalfy +=
               4*pi*pi*(Yhalf[i][j]-Yhalf[i][k])*all_planets[k].mass/pow(pow(Xhalf[i][j]-Xhalf[i][k],
           else{
           gravhalfx+= 0.0;
           gravhalfy+= 0.0;
}           }

        Rhalf[i][j] = sqrt(Xhalf[i][j]*Xhalf[i][j]
            +Yhalf[i][j]*Yhalf[i][j] );
        gravhalf = 4.0*pi*pi/(Rhalf[i][j]*Rhalf[i][j]*Rhalf[i][j]);
        kvx2[j] = gravhalfx;
        kvy2[j] = gravhalfy;
        kx2[j] = Vxhalf[i][j];
        ky2[j] = Vyhalf[i][j];
        //test = gravhalf*Yhalf[i][1];
        //cout << test << " " << kvy2[1] << endl;
}
    //Third
    for(j=0;j<m;j++){
        Xhalf[i][j] = X[i][j] + h/2.0*kx2[j];
        Yhalf[i][j] = Y[i][j] + h/2.0*ky2[j];
        Vxhalf[i][j] = Vx[i][j] - h/2.0*kvx2[j];
        Vyhalf[i][j] = Vy[i][j] - h/2.0*kvy2[j];}

    //Fourth

    for(j=0;j<m;j++){
    Rhalf[i][j] = sqrt(Xhalf[i][j]*Xhalf[i][j] +
        Yhalf[i][j]*Yhalf[i][j]);
    gravhalfx = 0.0;
    gravhalfy = 0.0;
       for(k=0;k<m;k++){

        if(k!=j){

           gravhalfx +=
               4*pi*pi*(Xhalf[i][j]-Xhalf[i][k])*all_planets[k].mass/pow(pow(Xhalf[i][j]-Xhalf[i][k],
           gravhalfy +=
               4*pi*pi*(Yhalf[i][j]-Yhalf[i][k])*all_planets[k].mass/pow(pow(Xhalf[i][j]-Xhalf[i][k],
           else{
           gravhalfx+= 0.0;
           gravhalfy+= 0.0;
}           }
```

```
        gravhalf = 4.0*pi*pi/(Rhalf[i][1]*Rhalf[i][1]*Rhalf[i][1]);
        kvx3[j] = gravhalfx;
        kvy3[j] = gravhalfy;
        kx3[j] = Vxhalf[i][j];
        ky3[j] = Vyhalf[i][j];
        //test = gravhalf*X[i][1];
        //cout << test << " " << kvx3[1] << endl;


    }


    //Fifth

    for(j=0;j<m;j++){
    Xhalf[i][j] = X[i][j] + h*kx3[j];
        Yhalf[i][j] = Y[i][j] + h*ky3[j];
        Vxhalf[i][j] = Vx[i][j] - h*kvx3[j];
        Vyhalf[i][j] = Vy[i][j] - h*kvy3[j];
        Rhalf[i][j] = sqrt(Xhalf[i][j]*Xhalf[i][j]
            +Yhalf[i][j]*Yhalf[i][j] );
               gravhalf =
                   4.0*pi*pi/(Rhalf[i][j]*Rhalf[i][j]*Rhalf[i][j]);


               gravhalfx = 0.0;
               gravhalfy = 0.0;
                  for(k=0;k<m;k++){

                   if(k!=j){

                      gravhalfx +=
                          4*pi*pi*(Xhalf[i][j]-Xhalf[i][k])*all_planets[k].mass/pow(pow(Xhalf[i][j]-X
                      gravhalfy +=
                          4*pi*pi*(Yhalf[i][j]-Yhalf[i][k])*all_planets[k].mass/pow(pow(Xhalf[i][j]-X
                      else{
                      gravhalfx+= 0.0;
                      gravhalfy+= 0.0;
            }          }




               kvx4[j] = gravhalfx;
               kvy4[j] = gravhalfy;
               kx4[j] = Vxhalf[i][j];
               ky4[j] = Vyhalf[i][j];
               //test = gravhalf*X[i][j];
               //cout <<test << " " << kvx4[1] << endl;
}

    //Sixth
    for(j=0;j<m;j++){
            X[i+1][j] = X[i][j] + h/6.0*(Vx[i][j] +2.0*(kx2[j]+kx3[j]) +
                kx4[j]);
            Y[i+1][j] = Y[i][j] + h/6.0*(Vy[i][j] +2.0*(ky2[j]+ky3[j]) +
                ky4[j]);
            Vx[i+1][j] = Vx[i][j] - h/6.0*(kvx1[j] +2.0*(kvx2[j]+kvx3[j]) +
                kvx4[j]);
            Vy[i+1][j] = Vy[i][j] - h/6.0*(kvy1[j] +2.0*(kvy2[j]+kvy3[j]) +
                kvy4[j]);
            R[i+1][j] = sqrt(X[i+1][j]*X[i+1][j] + Y[i+1][j]*Y[i+1][j]);

}

    //cout << r[i] << endl;

//cout << R[i][1] << endl;

}
finish = clock();
    ofstream myfile;
        myfile.open("RK4circ.txt");
    for(i = 0; i<n;i++){
   myfile << X[i][1] << " " << Y[i][1] << " "<< Vx[i][1] << " " <<
        Vy[i][1] << endl;

      //"  " << i << endl;
    }
    cout << ((finish - start)/CLOCKS_PER_SEC) << endl;
    myfile.close();
//cout << r[n-1] << " " << x[n-1] << " " << y[n-1] << " " << vx[n-1] <<
     " " << vy[n-1] - 2*pi << endl;

    //-----------------------------------------------------------------------------------------

    delete [] kx1;
       delete [] kx2;
       delete [] kx3;
       delete [] kx4;
    delete [] ky1;
       delete [] ky2;
       delete [] ky3;
       delete [] ky4;
    delete [] kvx1;
       delete [] kvx2;
       delete [] kvx3;
       delete [] kvx4;
    delete [] kvy1;
       delete [] kvy2;
       delete [] kvy3;
       delete [] kvy4;
    free_matrix((void **) X);
    free_matrix((void **) Y);

    free_matrix((void **) R);
    free_matrix((void **) Rhalf);
    free_matrix((void **) Yhalf);
    free_matrix((void **) Xhalf);
    free_matrix((void **) Vyhalf);
    free_matrix((void **) Vxhalf);
    free_matrix((void **) Vx);
    free_matrix((void **) Vy);
return;




}

void VV(){
    double **R,**X,**Y,**Xhalf,**Yhalf,**Rhalf;
    double **Vxhalf,**Vyhalf,**Vx,**Vy;
   // double *vx,*vy,*x,*y,*r;
    //double *vxhalf,*vyhalf,*xhalf,*yhalf,*rhalf;
    double
        gravnowx,gravnowy,gravnextx,gravnexty,gravhalfx,gravhalfy,tmax,t0,h,h2,pi,gravnow,gravnext,t
    double *kx1,*kx2,*kx3,*kx4,*kvx1,*kvx2,*kvx3,*kvx4;
    double *ky1,*ky2,*ky3,*ky4,*kvy1,*kvy2,*kvy3,*kvy4;
    int i,j,n,k,m;
    t0 = 0.0;
    tmax = 1.0;
    n = 500;
    h = (tmax - t0)/(n+1);
    h2 = h*h;
    pi = acos(-1.0);
  /*  r = new double[n];
    kvx1 = new double[n];
    kvy1 = new double[n];
    kx1 = new double[n];
    ky1 = new double[n];

    rhalf = new double[n];
    vxhalf = new double[n];
    vyhalf = new double[n];
    xhalf = new double[n];
    yhalf = new double[n];*/
clock_t start, finish;

       kvx1 = new double[3];
       kvy1 = new double[3];
       kx1 = new double[3];
       ky1 = new double[3];
       kvx2 = new double[3];
       kvy2 = new double[3];
       kx2 = new double[3];
       ky2 = new double[3];
       kvx3 = new double[3];
       kvy3 = new double[3];
       kx3 = new double[3];
       ky3 = new double[3];
       kvx4 = new double[3];
       kvy4 = new double[3];
       kx4 = new double[3];
       ky4 = new double[3];

    R = (double **) matrix(n,3,sizeof(double));
    X = (double **) matrix(n,3,sizeof(double));
    Y = (double **) matrix(n,3,sizeof(double));
    Rhalf = (double **) matrix(n,3,sizeof(double));
    Xhalf = (double **) matrix(n,3,sizeof(double));
    Yhalf = (double **) matrix(n,3,sizeof(double));

    Vx = (double **) matrix(n,3,sizeof(double));
    Vy = (double **) matrix(n,3,sizeof(double));
    Vxhalf = (double **) matrix(n,3,sizeof(double));
    Vyhalf = (double **) matrix(n,3,sizeof(double));




   // planet Earth(0.000003,-9.687087657503362E-01,
       -2.304478999525212E-01,0.0,3.715293823075145E-03*365.0,-1.679555302972549E-02*365.0,0.0);
  //   planet
      Sun(1.0,0.00510182404,0.0008987727027,0.0,-0.0004829298902,-0.002481983714,0.0);
      //(mass,x,y,z,vx,vy,vz)
  //  planet
       Jupiter(.000954,-5.344777687381148,.9428344302147893,0.0,-1.398575965353750E-03*365.0,-7.07501
    //planet all_planets[3] = {Sun,Earth,Jupiter};
planet Earth(0.000003,1.0000,0,0,0,2*pi,0);
planet Sun(1.0,0,0,0,0,0,0);
//planet CM(0,0,0,0,0,0,0);
//planet Jupiter(0.000954,5.2,0,0,0,-2*pi/sqrt(5.2),0);
planet all_planets[2] = {Sun,Earth};//,Jupiter};
m = 2;


/*x[0] = 1.0;
vy[0] = 2*pi+2.5885754;
vx[0] = 0.0;
y[0] = 0.0;*/

for(i=0;i<m;i++){
X[0][i] = all_planets[i].position[0];
Y[0][i] = all_planets[i].position[1];
Vx[0][i] = all_planets[i].velocity[0];
Vy[0][i] = all_planets[i].velocity[1];

}



    //Verlet-----------------------------------------------------------------------------------------
start = clock();
```

```
   for(i = 0;i<n-1;i++){
      for(j = 0;j<m;j++){
        //  cout << R[0][1] << endl;
       R[i][j] = sqrt(X[i][j]*X[i][j] + Y[i][j]*Y[i][j]);
       gravnow = 4.0*pi*pi/pow(R[i][j],3);
       kx1[j] = 0.0;
       ky1[j] = 0.0;
          for(k=0;k<m;k++){

           if(k!=j){

              kx1[j] +=
                  4*pi*pi*(X[i][j]-X[i][k])*all_planets[k].mass/pow(pow(X[i][j]-X[i][k],2.0)+pow(Y[i
              ky1[j] +=
                  4*pi*pi*(Y[i][j]-Y[i][k])*all_planets[k].mass/pow(pow(X[i][j]-X[i][k],2.0)+pow(Y[i
              else{
              kx1[j]+= 0.0;
              ky1[j]+= 0.0;
}          }

        //  cout << gravnow*X[i][1] << " " << kx1[1] << endl;
    X[i+1][j] = X[i][j] + h*Vx[i][j] - h2/2.0*kx1[j];
    Y[i+1][j] = Y[i][j] + h*Vy[i][j] - h2/2.0*ky1[j];
 //   cout << kx1[j] << endl;
       }
    //all_planets[j].position[0] = X[i+1][j];
    //all_planets[];
      for(j=0;j<m;j++){
          kx2[j] = 0.0;
          ky2[j] = 0.0;
    R[i+1][j] = sqrt(X[i+1][j]*X[i+1][j] + Y[i+1][j]*Y[i+1][j]);
    gravnext = 4.0*pi*pi/(R[i+1][j]*R[i+1][j]*R[i+1][j]);
    for(k=0;k<m;k++){
     if(k!=j){
        //cout<<X[i+1][k] << endl;
       kx2[j] +=
           4*pi*pi*(X[i+1][j]-X[i+1][k])*all_planets[k].mass/pow(sqrt(pow(X[i+1][j]-X[i+1][k],2.0)+p
       ky2[j] +=
           4*pi*pi*(Y[i+1][j]-X[i+1][k])*all_planets[k].mass/pow(sqrt(pow(X[i+1][j]-X[i+1][k],2.0)+p
       //cout << gravnextx << " " << j << " " << k << endl;
     }else{
       kx2[j]+= 0.0;
       ky2[j]+= 0.0;
}          }

    Vx[i+1][j] = Vx[i][j] -h/2.0*(kx2[j] + kx1[j]);
    test =Vy[i][j] - h/2.0*(gravnext*Y[i+1][1]+gravnow*Y[i][1]);
    Vy[i+1][j] = Vy[i][j] -h/2.0*(ky2[j] + ky1[j]);

   // cout << Vy[i+1][1] << " "<< test << endl;
}
}
   finish = clock();
    ofstream myfile;
```

```
        myfile.open("VVcirc.txt");
    for(i = 0; i<n;i++){
     myfile << X[i][1] << " " << Y[i][1] << " "<< Vx[i][1] << " " <<
         Vy[i][1] << endl;
     // "  " << i << endl;
    }
    //cout << finish-start << endl;
    cout << ((finish - start)/CLOCKS_PER_SEC) << endl;
    myfile.close();
//-----------------------------------------------------------------------------------------------------

return;
}



void PST(){
    int m,n,N,k,j,l,i,m1,m1L,mL,j1;
clock_t start, finish;
    double a,b,pi = acos(-1.0),nt,c,**d,e,f,g;
   // d = (double **) matrix(N,N,sizeof(double));
   // double u1,u2,u3,A,xco,vco;
    // planet Earth(0.000003,-9.687087657503362E-01,
        -2.304478999525212E-01,0.0,3.715293823075145E-03*365.0,-1.679555302972549E-02*365.0,0.0);
    // planet
        Sun(1.0,0.00510182404,0.0008987727027,0.0,-0.0004829298902,-0.002481983714,0.0);
        //(mass,x,y,z,vx,vy,vz)
   //  planet
       Jupiter(.000954,-5.344777687381148,.9428344302147893,0.0,-1.398575965353750E-03*365.0,-7.0750
     //planet all_planets[3] = {Sun,Earth,Jupiter};
 planet Earth(0.000003,1.0000,0,0,0,2*pi,0);
 planet Sun(1.0,0,0,0,0,0,0);
 //planet CM(0,0,0,0,0,0,0);
 //planet Jupiter(0.000954,5.2,0,0,0,-2*pi/sqrt(5.2),0);
 N=2;
 planet all_planets[N] = {Sun,Earth};//,Jupiter};
// N=3;
 nt=80;
 d = (double **) matrix(N,N,sizeof(double));



 vector<vector<vector<double> > > u1;
 vector<vector<vector<double> > > u2;
 vector<vector<vector<double> > > u3;
 vector<vector<vector<double> > > A;
 vector<vector<vector<double> > > xco;
 vector<vector<vector<double> > > vco;

 // Set up sizes. (HEIGHT x WIDTH)
 u1.resize(N);
 u2.resize(N);
 u3.resize(N);
 A.resize(N);
 xco.resize(N);
 vco.resize(N);
 for (int i = 0; i < N; ++i) {
   u1[i].resize(N);
   u2[i].resize(N);
   u3[i].resize(N);
   A[i].resize(N);
   xco[i].resize(N);
   vco[i].resize(N);
   for (int j = 0; j < N; ++j){
     u1[i][j].resize(nt);
     u2[i][j].resize(nt);
     u3[i][j].resize(nt);
     A[i][j].resize(nt);
     xco[i][j].resize(nt);
     vco[i][j].resize(nt);
 }
    }
for(j=0;j<N;j++){
for(i=0;i<2;i++){
 xco[i][j][0] = all_planets[j].position[i];
 vco[i][j][0] = all_planets[j].velocity[i];
}
}
for(j=0;j<N;j++){
   for(k = 0;k<N;k++){
     for(i = 0;i<2;i++){
      d[j][k] += pow(xco[i][j][0]-xco[i][k][0],2);
      A[j][k][0] =
           (xco[i][j][0]-xco[i][k][0])*(vco[i][j][0]-vco[i][k][0]);
     }

}
}

for(j=0;j<N;j++){
for(i=0;i<N;i++){
    if( i == j)
        b = 0.0;
   else{
    b =1.0/d[j][i];}
 u1[j][i][0] = b;
 u2[j][i][0] = b*b;
 u3[j][i][0] = b*b*b;
 u1[j][i][0] = b;
 u2[j][j][0] = b*b;
 u3[j][i][0] = b*b*b;
}
}
//cout << "test" << endl;
start = clock();
    for(m=1;m<nt;m++){ // m may need to equal zero, run program and check
    m1 = m-1;
   // cout << m << endl;
```

```
for(j=0;j<N;j++){
    for(i = 0;i<2;i++){
        xco[i][j][m] = vco[i][j][m1]/m;
      //   cout << xco[i][j][m] << " "<< i << " " << j << " " << m
             <<endl;
        a = 0;
        for(k = 0;k<N;k++){
            b = 0;
            for(l = 0; l<=m1;l++){ //run program to double check
                 bounds
                m1L = m1- l;
                b += (xco[i][k][l] - xco[i][j][l])*u3[j][k][m1L];
        }
            a += 4.0*pi*pi*b*all_planets[k].mass/m;
        }
        vco[i][j][m] = a;

    }

    j1 = j-1;
    for(k=0;k<j1;k++){
        a = 0;
        for(l = 0;l<m1;l++){
            m1L = m1-l;
            a -= u3[j][k][l]*A[j][k][m1L];
        }
        u1[j][k][m] = a/m;
        u1[k][j][m] = a/m;
        a = 0;
        for(l = 0;l<m1;l++){
            mL = m-l;
            a += u1[j][k][l]*u1[j][k][mL];
        }
        u2[j][k][l] = a;
        u2[k][j][l] = a;
        a = 0;
        b = 0;
        for(l = 0;l<m;l++){
            mL = m-l;
            b+= u2[j][k][l]*u1[j][k][mL];
            for (i=0;i<2;i++){
                a+=(xco[i][j][l]-xco[i][k][l])*(vco[i][j][mL]-vco[i][k][mL]);
            }
        }
        A[j][k][m] = a;
        A[k][j][m] = a;
        u3[j][k][m] = b;
        u3[k][j][m] = b;
    }
    A[j][j][m] = 0;
    u1[j][j][m] = 0;
    u2[j][j][m] = 0;
    u3[j][j][m] = 0;
}
```

```
    }
   // finish = clock();
  //  cout << xco[0][1][2]<< endl;

//cout << "test" << endl;
  //  cout << xco[0][1][0]+
      xco[0][1][2]+xco[0][1][4]+xco[0][1][6]+xco[0][1][8]+xco[0][1][10]+xco[0][1][12]+xco[0][1][14]
    //       +xco[0][1][16]+xco[0][1][18]+xco[0][1][20]+xco[0][1][22]<<
        endl;

    ofstream myfile;
    myfile.open("PSTcirc.txt");
    for(j=0;j<500;j++){
        a = 0.0;
        c = 0.0;
        e = 0.0;
        f = 0.0;
        g=0.0;
    for(i=0;i<nt;i++){
        //a = 2*i;
        c += xco[0][1][i]*pow(j/501.0,i);
        e += xco[1][1][i]*pow(j/501.0,i);
        f += vco[0][1][i]*pow(j/501.0,i);
        g += vco[1][1][i]*pow(j/501.0,i);
        cout  << c << endl;
    }
    myfile << c << " " << e << " " << f << " " << g << endl;
    }finish = clock();
    //cout << (finish-start)/CLOCKS_PER_SEC << endl;
    myfile << ((finish - start)/CLOCKS_PER_SEC) << endl;
myfile.close();


return;

}





int main()
{

  //  cout << X[0][1] << endl;
    int i;
    for(i=0;i<10;i++){
        cout << i << endl;
RK4();

VV();
PST();
}
```

```
    return 0;
}
```