Numerical Solutions to the Problem of the Solar System

Elizabeth Drueke

April 7, 2016

Abstract

a

1 Introduction

For as long as humans have been on the earth, we have been looking to the sky. Ancient peoples relied on the stars and planets to tell when the seasons were changing, so they would know when was a good time to plant crops for food. A little later, the stars were used by mariners as a compass, guiding them across the seas.

Today still we look towards the skies. However, we arguably know more about them. We know that our solar system is not the center of the universe, nor are we the center of our solar system. We know that our sun is a middle-sized star, and without its light and heat there would be no life on earth. We know that the rotation of the planets around the sun is dictated by Newton's Second Law, and, with this, we can calculate the trajectory of the planetary bodies around us.

Here, we present two such calculations. Using the Verlet and 4^{th}-Order Runge-Kutta (RK4) Methods, we investigate the motion of the planets around the sun. We begin by presenting the simple theory of planetary motion in Section 2. Then we discuss the Verlet and RK4 methods in Sections 2.1 and 2.2. After this, we discuss the framework and the algorithm developed particularly in this project in Section 3. Finally, results and benchmarks for the code are discussed in Section 4.

2 Theory

As mentioned in Section 1, the movements of planets are dictated by Newton's Second Law, which states

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

Thus, we have a second order differential equation:

$$m\frac{d^2\vec{x}}{dt^2} = \vec{F}$$

where $\vec{F}$ is the sum of the forces on the planet in question. In particular, the force of one planet on another is given by

$$\vec{F} = -\frac{Gm_1m_2}{r^2}\hat{r} \tag{2.1}$$

where $G = 6.67 \times 10^{-11} m^3 kg^{-1} s^{-2}$ is the gravitational constant, $m_{1,2}$ are the masses of the two planets, and r is the distance between the planets.

Now, we want to be able to use some sort of discretized version of this in order to use a computer to approximate a numerical solution to this problem. Our first step is then to look at Eq. 2.1 component-wise (ie. look at the x- and y- components separately). In particular, we should have

$$m\frac{d^2x}{dt^2} = F_x \text{ and } m\frac{d^2y}{dt^2} = F_y$$

Noting that $\vec{r} = x\hat{x} + y\hat{y}$, this gives us that

$$F_x = -\frac{Gm_1m_2x}{r^3} \text{ and } F_y = -\frac{Gm_1m_2y}{r^3}$$

Thus we have two coupled second-order differential equations:

$$\frac{d^2x}{dt^2} = -\frac{Gm_1x}{r^3} \text{ and } \frac{d^2y}{dt^2} = -\frac{Gm_1y}{r^3} \tag{2.2}$$

or, alternatively, four coupled first-order differential equations:

$$\begin{aligned} \frac{dx}{dt} &= v_x, \quad & \frac{dv_x}{dt} &= -\frac{Gm_1x}{r^3}, \\ \frac{dy}{dt} &= v_y, \quad & \frac{dv_y}{dt} &= -\frac{Gm_1y}{r^3}. \end{aligned} \tag{2.3}$$

In this analysis, we first investigate the unperturbed earth-sun system. In this case, Eq. 2.1 becomes

$$\vec{F} = \frac{GM_\odot m_E}{r^2}\hat{r} \tag{2.4}$$

where m_E is the mass of the earth and $M_\odot$ is the mass of the sun. Assuming a circular orbit, we can say that

$$a = \frac{mc^2}{r},$$

and so we have

$$\frac{mv^2}{r} = \frac{GM_\odot m_E}{r^2}$$

or

$$v^2 r = GM_\odot = 4\pi^2 AU^2 yr^{-2}.$$

Thus, for the unperturbed earth-sun system, we wish to investigate

$$\frac{d^2x}{dt^2} = -\frac{4\pi^2 x}{r^3} \text{ and } \frac{d^2y}{dt^2} = -\frac{4\pi^2 y}{r^3}. \tag{2.5}$$

We will also want to look at adding other planets to our solar system. After all, the unperturbed earth-sun system is really to simplistic to be a reasonable approximation for how the solar system. Noting that

$$Gm_p = GM_\odot \frac{m_p}{M_\odot} = 4\pi^2 \frac{m_p}{M_\odot},$$

we have, for planet $p\prime$,

$$\begin{aligned} a_x = \frac{dv_x}{dt} = & -\frac{4\pi^2}{r_{p\prime\odot}^3}\left(x_{p\prime} - x_\odot\right) - \frac{4\pi^2}{M_\odot}\sum_{p\neq p\prime}\frac{m_p\left(x_{pprime} - x_p\right)}{r_{pp\prime}^3}, \\ a_y = \frac{dv_y}{dt} = & -\frac{4\pi^2}{r_{p\prime\odot}^3}\left(y_{p\prime} - y_\odot\right) - \frac{4\pi^2}{M_\odot}\sum_{p\neq p\prime}\frac{m_p\left(y_{p\prime} - y_p\right)}{r_{pp\prime}^3}. \end{aligned} \tag{2.6}$$

The solution of Eq. 2.5 is fairly straightforward (we are only really looking at two coupled second-order differential equations), although still not simple by any means. However, the solution to Eq. 2.6 is impossible to get by hand. The number of couple equations will be twice the number of planets in the solar system. Thus, to solve either system, it is useful to turn to numerical approximations and computer algorithms. In particular, we look into the Verlet and RK4 methods as means by which to solve the system.

2.1 Verlet Method

It is a common practice in creating computer algorithms to solve complex problems to discretize the equations in order to get something more concrete to work with. In this case, we will discretize using the Taylor Series expansion. That is, we will have

$$x\left(t+h\right) = x\left(t\right) + hx^{'}\left(t+h\right) + \frac{h^2}{2}x^{''}\left(t+h\right) + O\left(h^3\right) \tag{2.7}$$

Thus, we can say, letting $x_i = x\left(t_0 + hi\right)$, that, for planet $p\prime$ in the multi-planet system,

$$\begin{aligned} x_{i+1} &= x_i + hv_i + \frac{h^2}{2}v_i^{'} + O\left(h^3\right) \\ &= x_i + hv_i + \frac{h^2}{2}\left(-\frac{4\pi^2}{r_{p\prime\odot i}^{\ 3}}\left(x_{p\prime} - x_\odot\right)_i - \frac{4\pi^2}{M_\odot}\sum_{p\neq p\prime}\frac{m_p\left(x_{p\prime} - x_p\right)_i}{r_{pp\prime i}^{\ 3}}\right) + O\left(h^3\right). \end{aligned} \tag{2.8}$$

We can similarly discretize the velocity of planet $p\prime$ to find

$$
\begin{aligned}
v_{i+1} &= v_i + \frac{h}{2}\left(v_{i+1}^{'} + v_i^{'}\right) + O\left(h^2\right) \\
&= v_i + \frac{h}{2}\left(-4\pi^2\left(\frac{\left(x_{p\prime} - x_{\odot}\right)_i}{r_{p\prime\odot i}^{\,3}} + \frac{\left(x_{p\prime} - x_{\odot}\right)_{i+1}}{r_{p\prime\odot i+1}^{\,3}}\right)\right. \\
&\left. - \frac{4\pi^2}{M_{\odot}}\sum_{p\neq p\prime} m_p\left(\frac{\left(x_{p\prime} - x_p\right)_i}{r_{pp\prime i}^{\,3}} + \frac{\left(x_{p\prime} - x_p\right)_{i+1}}{r_{pp\prime i+1}^{\,3}}\right) + O\left(h^3\right).\right.
\end{aligned}
\tag{2.9}
$$

In the case of the unperturbed earth-sun system, Eqs. 2.8 and 2.9 simplify to

$$
x_{i+1} = x_i + hv_i + \frac{h^2}{2}\left(-\frac{4\pi^2}{r_{p\prime\odot i}^{\,3}}\left(x_{p\prime} - x_{\odot}\right)_i\right) + O\left(h^3\right)
$$

and

$$
v_{i+1} = v_i + \frac{h}{2}\left(-4\pi^2\left(\frac{\left(x_{p\prime} - x_{\odot}\right)_i}{r_{p\prime\odot i}^{\,3}} + \frac{\left(x_{p\prime} - x_{\odot}\right)_{i+1}}{r_{p\prime\odot i+1}^{\,3}}\right)\right) + O\left(h^3\right).
$$

Together, Eqs. 2.8 and 2.9 make up what is known as the Verlet method **cite lecture notes**. With the introduction of the velocity Verlet method, this method is self-starting.

2.2 Fourth-Order Runge-Kutta

The RK4 method is a bit more precise than the Verlet method discussed in Section 2.1. It is based on the observation that, for

$$
\frac{dy}{dt} = f\left(t, y\right),
$$

we can say

$$
y\left(t\right) = \int f\left(t, y\right) dt.
$$

Discretizing, this yields

$$
y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f\left(t, y\right) dt.
$$

Letting $y_{i+1/2} = y\left(t_i + h/2\right)$, and using the midpoint formula for the integral, we find

$$
\int_{t_i}^{t_{i+1}} f\left(t, y\right) dt \approx hf\left(t_{i+1/2}, y_{i+1/2}\right) + O\left(h^3\right).
$$

Thus, we have

$$
y_{i+1} = y_i + hf\left(t_{i+1/2}, y_{i+1/2}\right) + O\left(h^3\right).
$$

However, it is clear that, in order to use this method, we must have some idea of what $y_{i+1/2}$ is. To get this quantity, we use Euler's method to approximate it:

$$y_{i+1/2} \approx y_i + \frac{h}{2} f\left(t_i, y_i\right).$$

This leads us to the 2^{nd}-Order Runge-Kutta Method, or RK2, which says that, for

$$\begin{aligned} k_1 &= hf\left(t_i, y_i\right) \\ k_2 &= hf\left(t_{i+1/2}, y_i + k_1/2\right), \end{aligned} \tag{2.10}$$

we have

$$y_{i+1} \approx y_i + k_2 + O\left(h^2\right). \tag{2.11}$$

We can go through another similar sequence of steps to get to RK4, culminating in the following definitions:

$$\begin{aligned} k_1 &= hf\left(t_i, y_i\right) \\ k_2 &= hf\left(t_i + h/2, y_i + k_1/2\right), \\ k_3 &= hf\left(t_i + h/2, y_i + k_2/2\right), \\ k_4 &= hf\left(t_i + h, y_i + k_3\right), \\ y_{i+1} &\approx y_i + (1/6)\left(k_1 + 2k_2 + 2k_3 + k_4\right) + O\left(h^4\right). \end{aligned} \tag{2.12}$$

3 The Algorithm

The code developed for this project is written in C++ using the ROOT **cite root** framework for plotting. Code from **project 1 cite** and **project 2 cite** was reused, particularly the thevec and themat classes defined in classes.C. However, several new classes and functions were designed specifically for the problem of solving the solar system.

In header file `odesolvers.h` we define the `Verlet` and `RK4` functions, each of which return two `thevec`s, one of the discretized position solution and the other of the discretized velocity solution.

- The `Verlet` method begins by determining a step size `h` according to certain inputs (the initial time `t0` and final time `tf`, as well as the number of steps `nsteps`) as

  ```
  h = (1.0*tf-1.0*t0)/(1.0*nsteps);
  ```

  and then sets the first available position point to be

  ```
  pos[0] = x[0];
  pos[1] = pos[0]+h*v0
  ```

  for some given initial position `x0`. It then iterates over integers less than `nsteps` in order to completely solve the position vector:

```
for(int i=2;i<nsteps+1;i++){
     pos[i] = 2*pos[i-1]-pos[i-2]-a*pow(h,2)*pos[i-1]/r;
}
```

where `a` is the $4\pi^2$ factor. It then solves for the velocity vector by setting

```
vel[0] = v0;
```

for some given initial velocity `v0` and then iterating again over integers less than `nsteps` as follows:

```
for(int i=1;i<nsteps;i++){
     vel[i] = vel[i-1]+(h/2)*a*pos[i]/pow(r,3)-a*pos[i-1]/pow(r,3);
}
```

where `r` is the averate distance between the planet and the sun.

- The `RK4` algorithm computes the step size `h` and initializes the position and velocity vectors as the `Verlet` algorithm does, and then iterates over integers less than `nsteps` as follows:

```
for(int i=0;i<nsteps;i++){
     double k1p = h*vel[i-1];
     double k1v = h*a*pos[i-1]/pow(r,3);

     double k2p = h*vel([i-1]+k1v/2);
     double k2v = -h*a*(pos[i-1]+k1p/2)/pow(r,3);

     double k3p = h*(vel[i-1]+k2v/2);
     double k3v = -h*a*(pos[i-1]+k2p/2)/pow(r,3);

     double k4p = h*(vel[i-1]+k3v);
     double k4v = -h*a*(pos[i-1]+k3p)/pow(r,3);

     pos[i] = pos[i-1]+(1/6)*(k1p+2*k2p+2*k3p+k4p);
     vel[i] = vel[i-1]+(1/6)*(k1v+2*k2v+2*k3v+k4v);
}
```

These algorithms clearly are only designed to solve one direction (x or y) at a time, and are designed only to handle the earth-sun system with the earth in an assumed circular orbit. The multi-body problem was solved by a similar code in `solar_system.C`.

We also, for this project, found it prudent to define two new classes: `planet` and `solar_system`. The `planet` class is declared and defined in `planets.h` and `planets.C`. Each object of type `planet` has associated it with it a `mass` (`double` of mass in kilograms),

`dist_sun` (`double` of the average distance between the planet and the sun, `name` (`string` of the name of the planet), `acc` (`double` of the acceleration the planet would have in a pure, idealized sun-planet system with a circular orbit), `v0` (`double` of the average velocity of the planet around the sun), and eight `thevec` which are used to hold the positions and velocities in the x- and y-directions after they are solved for by the `RK4` and `Verlet` algorithms. These last objects, however, are not directly used by the `planet` class. It only became obvious after the class had been designed, during the designing stage of the `solar_system` class, that it would be convenient for the `planet` objects to have these components. Thus they are used solely by the `solar_system` class as discussed below.

The `planet` class has several constructors as well as functions:

- `print()`, which returns a string one might use to print the planet information to screen,
- `kinetic()`, which calculates the kinetic energy of the planet from a velocity which is taken as an argument,
- `potential()`, which calculates the potential energy of the planet from a distance from the sun which is taken as an argument,
- `ang_mom()`, which calculates the angular momentum of the planet from a distance from the sun and a velocity, which are both taken as arguments.

These algorithms are all fairly straightforward and do not merit discussion here.

The `solar_system` class is really the meat of the code. Defined and declared in `solar_system.h` and `solar_system.C`, this class was designed to solve the multi-body problem in full. Each object of type `solar_system` has associated with it `planets` (`vector<planet*>` of the planets to be included in the solar system), `nsteps` (`int` of the number of steps to be used by the Verlet and RK4 algorithms discussed below), `tf` (`double` of the final time to be used by the RK4 and Verlet algorithms), and `originx` and `originy` (`double` which describe the x- and y-components of the position of the origin to be used in the calculations). This class also has several constructors, as well as the following functions:

- `Add()`, which adds a planet, taken as an argument, to the solar system
- `Solve_Verlet()`, which solves the multi-body solar system using the Verlet algorithm. In particular, after calculating the step size as

  ```
  double h = tf/nsteps;
  ```

  and defining the $-4\pi^2$ factor as `fact` and the mass of the sun as `msun`, this algorithm creates a dynamic array of the planets in `planets` and initializes their position and velocity `thevec` objects by using their `v0` and `dist_sun` and the initial x-velocity and y-positions, respectively. Then, there is a `for` loop which iterates over integers `i` which are less than `nsteps`. For each `i`, it goes calculates the both the position at i for every planet in the solar system:

```
for(unsigned int it=0;it<planets.size();it++){

   planet myplan = theplanets[it];

   double vx_prev = myplan.velocitiesx_v[i-1];
   double vy_prev = myplan.velocitiesy_v[i-1];
   double x_prev = myplan.positionsx_v[i-1];
   double y_prev = myplan.positionsy_v[i-1];

   double r_prev = sqrt(pow(x_prev-originx,2)
                           +pow(y_prev-originy,2));

   double x = x_prev+h*vx_prev;
   double y = y_prev+h*vy_prev;

   for(unsigned int m = 0;m<planets.size();m++){

      planet plan = theplanets[m];

      if(myplan != plan){

         double xp_prev = plan.positionsx_v[i-1];
         double yp_prev = plan.positionsy_v[i-1];

         double rp_prev = sqrt(pow(x_prev-xp_prev,2)
                                    +pow(y_prev-yp_prev,2));

         x += (pow(h,2)/2)*fact*plan.mass
                  *(x_prev-xp_prev)/(msun*pow(rp_prev,3));
         y += (pow(h,2)/2)*fact*plan.mass
                  *(y_prev-yp_prev)/(msun*pow(rp_prev,3));

      }
   }

theplanets[it].positionsx_v[i]=x;
theplanets[it].positionsy_v[i]=y;
```

Then, the algorithm solves for the velocity at i for each of the planets:

```
for(unsigned int it = 0;it<planets.size();it++){

   planet myplan = theplanets[it];

   double vx_prev = myplan.velocitiesx_v[i-1];
```

```
      double vy_prev = myplan.velocitiesy_v[i-1];
      double x_prev = myplan.positionsx_v[i-1];
      double y_prev = myplan.positionsy_v[i-1];

      double r_prev = sqrt(pow(x_prev-originx,2)+pow(y_prev-originy,2));

      double x = myplan.positionsx_v[i];
      double y = myplan.positionsy_v[i];
      double newr = sqrt(pow(x-originx,2)+pow(y-originy,2));

      double vx = vx_prev;
      double vy = vy_prev;

      for(unsigned int m = 0;m<planets.size();m++){

         planet plan = theplanets[m];

         if(myplan != plan){

            double xp_prev = plan.positionsx_v[i-1];
            double yp_prev = plan.positionsy_v[i-1];

            double xp_cur = plan.positionsx_v[i];
            double yp_cur = plan.positionsy_v[i];

            double rp_cur = sqrt(pow(x-xp_cur,2)+pow(y-yp_cur,2));
            double rp_prev = sqrt(pow(x_prev-xp_prev,2)
                                  +pow(y_prev-yp_prev,2));

            vx += ((h/2)*fact*plan.mass/msun)*((x-xp_cur)/pow(rp_cur,3)
                      +(x_prev-xp_prev)/pow(rp_prev,3));
            vy += ((h/2)*fact*plan.mass/msun)*((y-yp_cur)/pow(rp_cur,3)
                      +(y_prev-yp_prev)/pow(rp_prev,3));

         }
      }

      theplanets[it].velocitiesx_v.point[i] = vx;
      theplanets[it].velocitiesy_v.point[i] = vy;

   }
```

Then, the algorithm redefines the planets in the `planets` vector so that their position and velocity Verlet `thevecs` are filled appropriately.

- `Solve`$_{RK4}()$ *uses the RK4 algorithm to solve the multi − body problem.*

4 Results and Benchmarks

5 Conclusions

6 Bibliography

1. a