Project 3

Jane Kim
Physics 480: Computational Physics
(Dated: May 6, 2018)

Summary, numbers.

I. INTRODUCTION

motivation, explain structure of report.

II. THEORY

The physics background required for this project is not very extensive, as the only force involved is gravity. To start simply, consider two celestial bodies with masses M_1 and M_2 at the locations (x_1, y_1) and (x_2, y_2) on the x-y plane. We can obtain two coupled differential equations for the motion of M_2

$$\begin{aligned}\frac{d^2x_2}{dt^2} &= \frac{GM_1(x_1 - x_2)}{((x_1 - x_2)^2 + (y_1 - y_2)^2)^{3/2}},\\ \frac{d^2y_2}{dt^2} &= \frac{GM_1(y_1 - y_2)}{((x_1 - x_2)^2 + (y_1 - y_2)^2)^{3/2}},\end{aligned} \tag{1}$$

using Newton's second law. Now add more planets with masses $M_3, M_4, ..., M_n$ into the system. Then to find the equations of motion for the j^{th} planet, we need to sum over the interactions between M_j and all the other M_k's:

$$\begin{aligned}\frac{d^2x_j}{dt^2} &= \sum_{\substack{k=1\\k\neq j}}^{n} \frac{GM_k(x_k - x_j)}{((x_k - x_j)^2 + (y_k - y_j)^2)^{3/2}},\\ \frac{d^2y_j}{dt^2} &= \sum_{\substack{k=1\\k\neq j}}^{n} \frac{GM_k(y_k - y_j)}{((x_k - x_j)^2 + (y_k - y_j)^2)^{3/2}}.\end{aligned} \tag{2}$$

This can easily be extended into three dimensions

$$\begin{aligned}\frac{d^2x_j}{dt^2} &= \sum_{\substack{k=1\\k\neq j}}^{n} \frac{GM_k(x_k - x_j)}{r_{j,k}^3},\\ \frac{d^2y_j}{dt^2} &= \sum_{\substack{k=1\\k\neq j}}^{n} \frac{GM_k(y_k - y_j)}{r_{j,k}^3},\\ \frac{d^2z_j}{dt^2} &= \sum_{\substack{k=1\\k\neq j}}^{n} \frac{GM_k(z_k - z_j)}{r_{j,k}^3},\\ r_{j,k} &= \sqrt{(x_k - x_j)^2 + (y_k - y_j)^2 + (z_k - z_j)^2},\end{aligned} \tag{3}$$

but we will focus on the two dimensional system for simplicity.

Using convenient units can simplify these calculations considerably. We express distances in astronomical units (AU) and times in years (yr). For a simple Earth-Sun system, we can approximate Earth's orbit to be circular around a fixed Sun. Then the centripetal force on Earth is equal to the gravitational force due to the sun. If the Sun has mass M_1 and the Earth has mass M_2, then we have that

$$\frac{M_2v_2^2}{r_{1,2}} = \frac{GM_1M_2}{r_{1,2}^2} \implies GM_1 = v_2^2 r_{1,2}. \tag{4}$$

Based on our choice of units, the distance between the Sun and Earth $r_{1,2}$ is just 1 AU. The speed of Earth v_2 is given by

$$v_2 = \frac{2\pi r_{1,2}}{T} = 2\pi\ \frac{AU}{\text{yr}}, \tag{5}$$

so we have that

$$GM_1 = 4\pi^2\ \frac{AU^3}{\text{yr}^2}. \tag{6}$$

This quantity is precalculated and stored as `prefactor`.

Define the mass ratio $m_k = M_k/M_1$ for each $k = 1, 2, ..., n$ celestial body in the system. Then (2) becomes

$$\begin{aligned}\frac{d^2x_j}{dt^2} &= \sum_{\substack{k=1\\k\neq j}}^{n} \frac{4\pi m_k(x_k - x_j)}{((x_k - x_j)^2 + (y_k - y_j)^2)^{3/2}},\\ \frac{d^2y_j}{dt^2} &= \sum_{\substack{k=1\\k\neq j}}^{n} \frac{4\pi m_k(y_k - y_j)}{((x_k - x_j)^2 + (y_k - y_j)^2)^{3/2}}.\end{aligned} \tag{7}$$

Discretizing these equations allow us to map the orbits of all the planets in the system.

III. METHOD

A. Forward Euler

The Euler method is a quick and simple algorithm for approximating the solution of first-order differential equations with the form $y' = f(y, t)$ and initial conditions $y(0) = y_0$. First, choose a time step h and let $t_n = t_0 + nh$. Then we have that

$$y(t_{n+1}) = y(t_n + h) = y(t_n) + hy'(t_n) + O(h^2). \tag{8}$$

If we substitute the differential equation and let $y_n \approx y(t_n)$ be the approximate numerical solution, we obtain

$$y_{n+1} \approx y_n + hf(y_n, t_n). \tag{9}$$

Since our equations are second-order, we apply this method twice to $y' = f(y,t)$ and $y'' = g(y',t)$ with initial conditions $y(0) = y_0$ and $y'(0) = v_0$.

B. Velocity Verlet

The velocity Verlet method is an algorithm for solving second-order differential equations with the form $y''(t) = f(y,t)$ with initial conditions $y(0) = y_0$ and $y'(0) = v_0$. Using the same notation as before, we have that

$$y_{n+1} \approx y_n + h = y_n + hy'_n + \frac{1}{2}h^2 f(y_n, t_n). \tag{10}$$

We can use Euler's method to then approximate $y'(t_{n+1})$ with a better guess for its derivative, namely,

$$y'_{n+1} \approx y'_n + \frac{1}{2}h(f(y_n, t_n) + f(y_{n+1}, t_{n+1})). \tag{11}$$

IV. IMPLEMENTATION

A class called `CPlanet` was created to hold the name, mass ratio, and initial conditions of each planet. This class also includes member functions to change the mass ratio or the initial conditions.

Another class `CSolarSystem` contained a list of CPlanet objects called `planet_list_`. The total number of planets in the system is stored in an integer `planets_`. Planets may be added to the system by using the following function:

```
// solar_system.cpp
void CSolarSystem::add(CPlanet NewPlanet){

   planets_ += 1;
   planet_list_.push_back(NewPlanet);

   // add one column for each body in the solar
       system
   for(int i = 0; i < N_+1; ++i){

      x_[i].resize(planets_);
      y_[i].resize(planets_);
      vx_[i].resize(planets_);
      vy_[i].resize(planets_);

      if(dim_ == 3){
         z_[i].resize(planets_);
         vz_[i].resize(planets_);
      }
   }
}
```

The `CSolarSystem` member functions `solve_euler` and `solve_vv` use the forward Euler and velocity Verlet methods, respectively, to calculate the trajectories of all the planets during the time window $[t_0, t_f]$. Thus to employ these functions properly, we designate an integer `N_` for the number of time steps between t_0 and t_f and let the time step `h_` be given by

$$h_- = \frac{t_f - t_0}{N_-}. \tag{12}$$

Each point in time $t_i = t_0 + i\mathtt{h\_}$, with $i = 0, 1, 2, ..., N$, was stored in a `CSolarSystem` member vector. Since the force on one planet requires the positions of all the other planets in the system, the calculations of the trajectories were stored in several matrices. The matrices `x_` and `y_` contain the positions in the $x-$ and $y-$ directions, while the matrices `vx_` and `vy_` contain the velocities. The columns in the matrices represent the planets in the system, so the matrices must be resized when new planets are added (see previous code excerpt). There are $N + 1$ rows in each matrix to store the calculations for each time step. Storing the computations in this way allows for easy extraction of the distances between planets.

Notice that the names of the class members are always followed by an underscore. For example, the name of a planet is stored in a string `name_`. This notation helps us keep track of the elements that can be changed by the use of a member function. The `CSolarSystem` functions `solve_euler` and `solve_vv` call another function named `get_acceleration` which calculates the acceleration of the j^{th} planet due to the other planets in the system during the i^{th} time step.

```
// solar_system.cpp
void CSolarSystem::get_acceleration(int i, int j,
    double& ax, double& ay, double& az){

   double m, r, r3;

   ax = 0.0;
   ay = 0.0;
   az = 0.0;
   for(int k = 0; k < planets_; k++){
      if(k != j) {
         m = planet_list_[k].m_;
         r = distance(i, j, k);
         r3 = r*r*r;
         ax += prefactor*m*(x_[i][k]-x_[i][j])/r3;
         ay += prefactor*m*(y_[i][k]-y_[i][j])/r3;
         if(dim_ == 3) az +=
             prefactor*m*(z_[i][k]-z_[i][j])/r3;
      }
   }
}
```

In addition, the `distance` function calculates the distance between the j^{th} and k^{th} planets during the i^{th} time step:

```
//solar_system.cpp
double CSolarSystem::distance(int i, int j, int k){

   double x, y, z = 0;

   x = x_[i][j]-x_[i][k];
   y = y_[i][j]-y_[i][k];
   if(dim_ == 3) z = z_[i][j]-z_[i][k];

   return sqrt(x*x+y*y+z*z);
}
```

In the Earth-Sun system, the center of mass is within the radius of the Sun so it is a fair approximation to simply keep the Sun fixed at (0,0). However, for most other situations, it is more appropriate to use the center of mass frame and let the Sun have a momentum such that the total momentum of the system is zero. Since this requires changing the initial conditions of the system, the `CSolarSystem` class has the option to "switch on" the center of mass frame. It also has the option to add a third dimension to the system. The changes to the initial conditions are performed in the following function:

```
//solar_system.cpp
void CSolarSystem::initialize(){

   // initial conditions of planets (including sun)
   for(int j = 0; j < planets_; j++){

      x_[0][j] = planet_list_[j].x0_[0];
      y_[0][j] = planet_list_[j].x0_[1];
      vx_[0][j] = planet_list_[j].v0_[0];
      vy_[0][j] = planet_list_[j].v0_[1];

      if(dim_ == 3){
         z_[0][j] = planet_list_[j].x0_[2];
         vz_[0][j] = planet_list_[j].v0_[2];
      }
   }

   // shift initial positions if center-of-mass
       option is chosen
   // change initial velocity of sun so that total
       momentum = 0
   CM_[0] = 0.0;
   CM_[1] = 0.0;
   CM_[2] = 0.0;
   if(CM_frame_){

      // find center-of-mass position
      double m, xsum = 0.0, ysum = 0.0, zsum = 0.0;
      for(int j = 0; j < planets_; j++){

         m = planet_list_[j].m_;
         xsum += x_[0][j];
         ysum += y_[0][j];
         CM_[0] += m*x_[0][j];
         CM_[1] += m*y_[0][j];

         if(dim_ == 3){
            zsum += z_[0][j];
            CM_[2] += m*z_[0][j];
         }
      }
      if(xsum != 0) CM_[0] = CM_[0]/xsum;
      if(ysum != 0) CM_[1] = CM_[1]/ysum;
      if(zsum != 0) CM_[2] = CM_[2]/zsum;

      cout << "CM = (" << CM_[0] << ", " << CM_[1]
          << ", " << CM_[2] << ")\n" << endl;

      // shift
      for(int j = 0; j < planets_; j++){
         x_[0][j] -= CM_[0];
         y_[0][j] -= CM_[1];
         if(dim_ == 3) z_[0][j] -= CM_[2];
      }

      // zero out momentum
      double px = 0.0, py = 0.0, pz = 0.0;
      for(int j = 1; j < planets_; j++){
         m = planet_list_[j].m_;
         px += m*vx_[0][j];
         py += m*vy_[0][j];
         if(dim_ == 3) pz += m*vz_[0][j];
      }
      m = planet_list_[0].m_;
      vx_[0][0] = -px/m;
      vy_[0][0] = -py/m;
      if(dim_ == 3) vz_[0][0] = -pz/m;
   }
}
```

After the `initialize` function is called, the system is ready to solve. We can use either `solve_euler` or `solve_vv` to fill out the matrices `x_`, `y_`, `vx_`, `vy_` (also, `z_` and `vz_` if `dim_` =3). The forward Euler method was written as

```
if(CM_frame_) j0 = 0;
else{
   j0 = 1;

   // fix the sun at (0,0)
   x_[i][0] = 0.0;
   y_[i][0] = 0.0;
   vx_[i][0] = 0.0;
   vy_[i][0] = 0.0;
   if(dim_ == 3){
      z_[i][0] = 0.0;
      vz_[i][0] = 0.0;
   }
}

// calculate orbits
for(int j = j0; j < planets_; j++){

   x_[i][j] = x_[i-1][j] + h_*vx_[i-1][j];
   y_[i][j] = y_[i-1][j] + h_*vy_[i-1][j];
   if(dim_ == 3) z_[i][j] = z_[i-1][j] +
       h_*vz_[i-1][j];
   get_acceleration(i-1, j, a[0], a[1], a[2]);
   vx_[i][j] = vx_[i-1][j] + h_*a[0];
   vy_[i][j] = vy_[i-1][j] + h_*a[1];
```

```
   if(dim_ == 3) vz_[i][j] = vz_[i-1][j] + h_*a[2];
}
```

to account for the options for a third dimension and the center of mass frame. The velocity Verlet method, on the other hand, uses the same `j0` as above, but the loop to calculate the orbits is

```
// calculate orbits
for(int j = j0; j < planets_; j++){

   get_acceleration(i-1, j, a1[0], a1[1], a1[2]);
   x_[i][j] = x_[i-1][j] + h_*vx_[i-1][j] +
       0.5*h_*h_*a1[0];
   y_[i][j] = y_[i-1][j] + h_*vy_[i-1][j] +
       0.5*h_*h_*a1[1];
   if(dim_ == 3) z_[i][j] = z_[i-1][j] +
       h_*vz_[i-1][j] + 0.5*h_*h_*a1[2];

   get_acceleration(i, j, a2[0], a2[1], a2[2]);
   vx_[i][j] = vx_[i-1][j] + 0.5*h_*(a1[0]+a2[0]);
   vy_[i][j] = vy_[i-1][j] + 0.5*h_*(a1[1]+a2[1]);
   if(dim_ == 3) vz_[i][j] = vz_[i-1][j] +
       0.5*h_*(a1[2]+a2[2]);
}
```

Then the trajectories are written to file using `write_orbits`. In this function, the number of points that are printed is limited so that the files for the full solar system are managable to plot.

Since there is no external torque acting on the system the angular momentum of each planet should be conserved. The angular momentum of the j^{th} planet at the i^{th} time step is calculated in

```
//solar_system.cpp
void CSolarSystem::get_angmomentum(int i, int j,
    double& L){

   double M, v, r;

   M = planet_list_[j].m_;
   v = velocity(i, j);
   r = distance(i, j, 0);

   L = M*v*r;
}
```

Likewise, the energy must also stay conserved because the system is isolated. The energy of the j^{th} planet at the i^{th} time step is given by

```
//solar_system.cpp
void CSolarSystem::get_energy(int i, int j, double&
    E){

   double M, m, v, r;

   // kinetic energy
   M = planet_list_[j].m_;
   v = velocity(i,j);
   E = 0.5*M*v*v;

   // potential energy
   for(int k = 0; k < planets_; k++){
      if(k != j){
         m = planet_list_[k].m_;
         r = distance(i, j, k);
         E += -prefactor*M*m/r;
      }
   }
}
```

V. TESTS

Four test cases were generated to examine the properties of the program. In all four cases, the velocity Verlet method was used to solve the simple Sun-Earth system. Earth was initially placed at $(x, y) = (1AU, 0)$ and given an initial velocity of $2\pi\frac{AU}{yr}$ in the positive y-direction.

A. Conservation of Energy

To test the conservation of energy, we first chose a small enough step size `h_`$= 10^{-4}$yr $= 52.6$ min so that our calculations were very stable. The initial energy of the system was calculated using `get_energy` and compared to the energy of Earth at every 100 time steps during one full orbit.

B. Conservation of Angular Momentum

The conservation of angular momentum was tested in a similar way. The function `get_angmomentum` returned the initial angular momentum of earth, which was then compared to the angular momentum at later times.

C. Stability of Earth's Orbit

To check that Earth's orbit remain stable over the course of 1000 years, we required that the distance between the Sun and the Earth was 1 AU at serveral different times. The step sized used in this case was the same as before.

D. Escape Velocity of Earth

If Earth's initial velocity is too large, it can fall out of the Sun's orbit. The theoretical escape velocity of Earth, or the minimum speed needed to escape the Sun's orbit, is given by

$$v_e = \sqrt{\frac{2GM_1}{r_{1,2}}} = \sqrt{2(4\pi^2)} = \sqrt{2}2\pi \approx 8.886\ \frac{AU}{yr}. \quad (13)$$

The escape velocity was found numerically by employing a variation of the bisection algorithm. We first defined the window for the escape velocity as [min,max]=[2π,3π] then used the following algorithm to narrow the window around the escape velocity

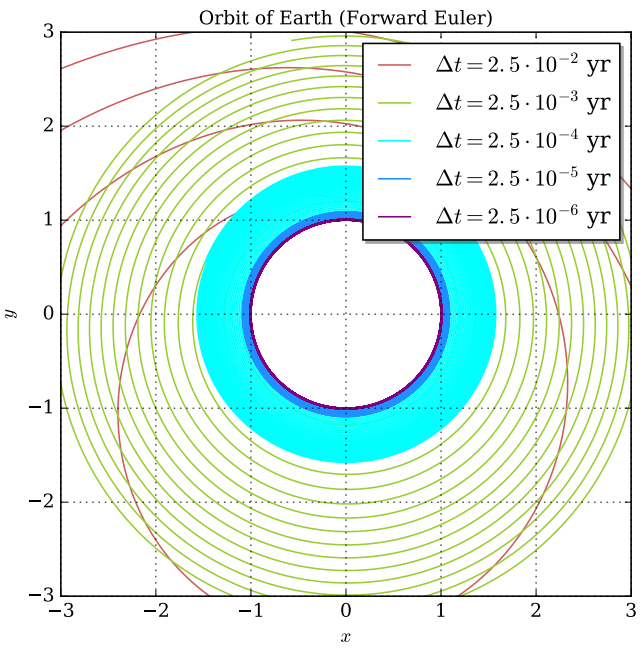
```
while(max-min > 1.0E-5){

   v0 = 0.5*(min+max);
   CPlanet sun("Sun", 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
       0.0);
   CPlanet earth("Earth", 3.0E-6, 1.0, 0.0, 0.0,
       0.0, v0, 0.0);
   CSolarSystem test(N, 2, 0, 1, false);
   test.add(sun);
   test.add(earth);
   test.initialize();
   test.solve_vv();
   test.get_energy(N, 1, E);
   if(E < 0) min = v0;
   if(E > 0) max = v0;
}
```

A time step of h_= 10^{-5}yr = 5.26 provided enough accuracy to correctly find the escape velocity up to 5 places after the decimal point.

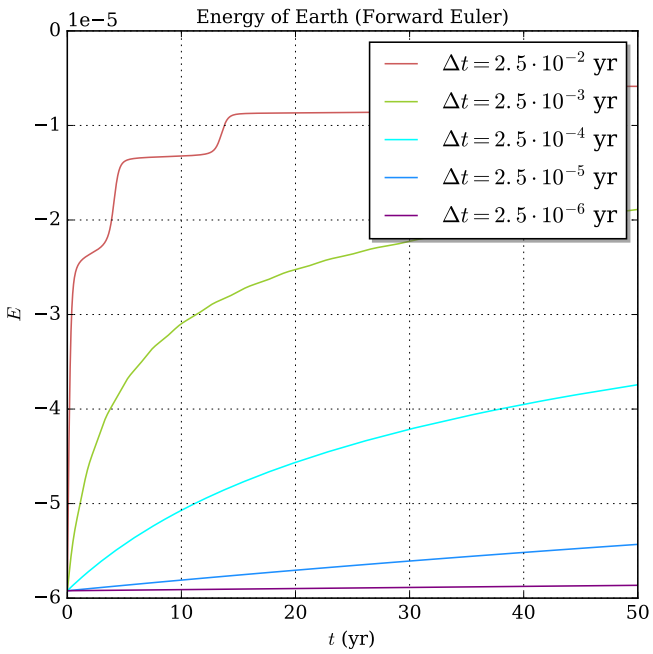
VI. RESULTS

All four tests, consisting of 10204 assertions, were passed.

VII. CONCLUSION



(a) A subfigure

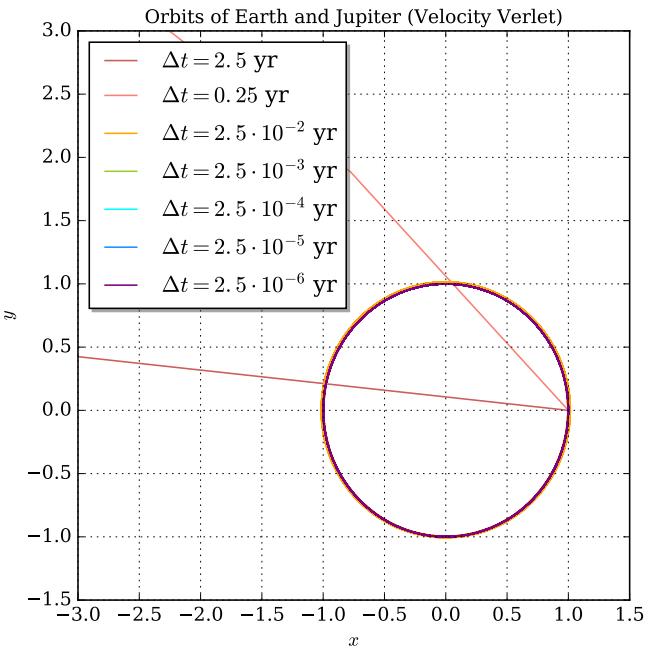


(b) A subfigure

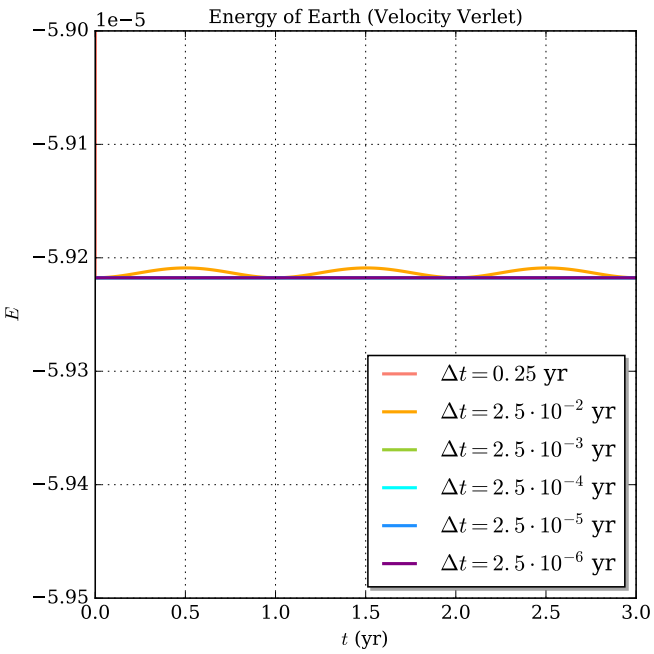
FIG. 1: A figure with two subfigures

---

[1] M. Hjorth-Jensen. "Computation Physics, Lecture Notes Fall 2015". University of Oslo. August 2015.

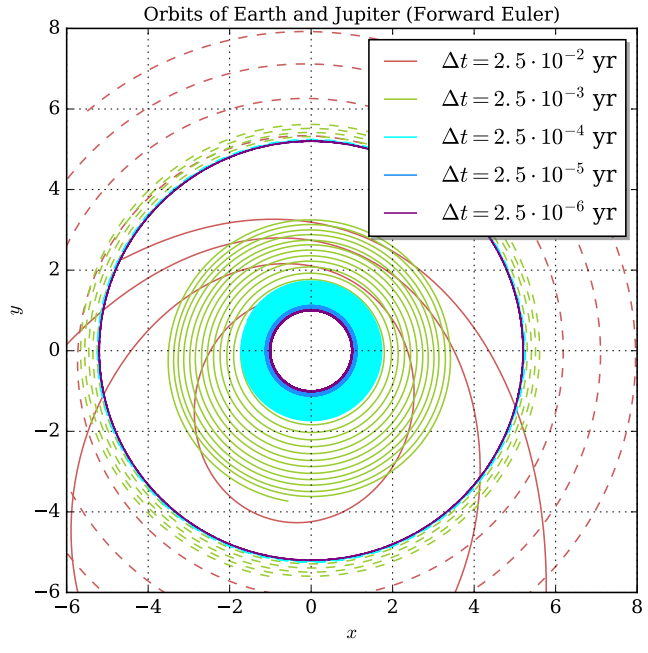


(a) A subfigure

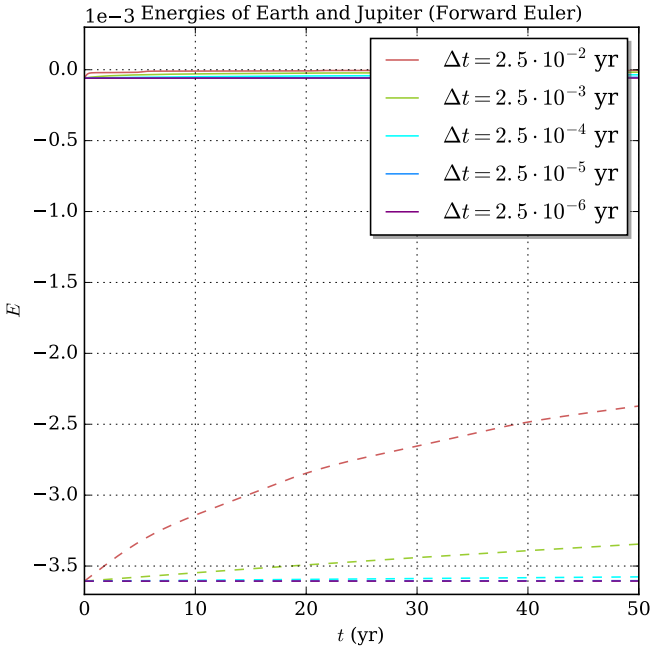


(b) A subfigure

FIG. 2: A figure with two subfigures

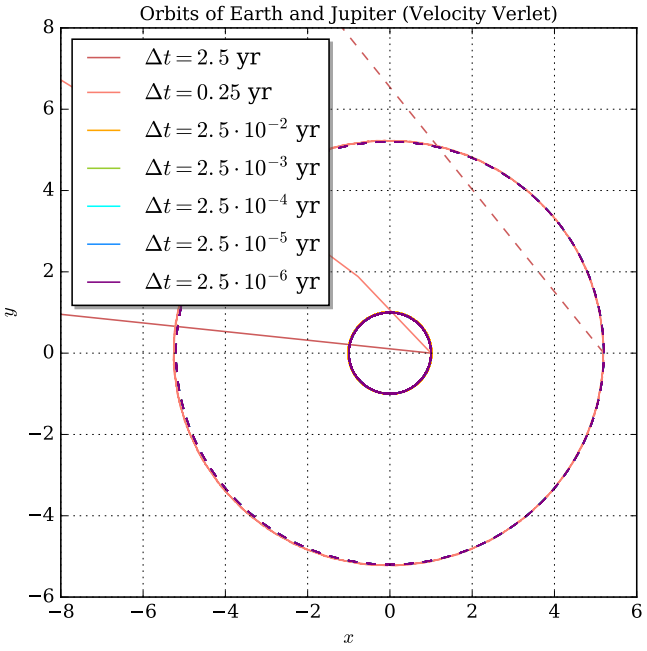


(a) A subfigure

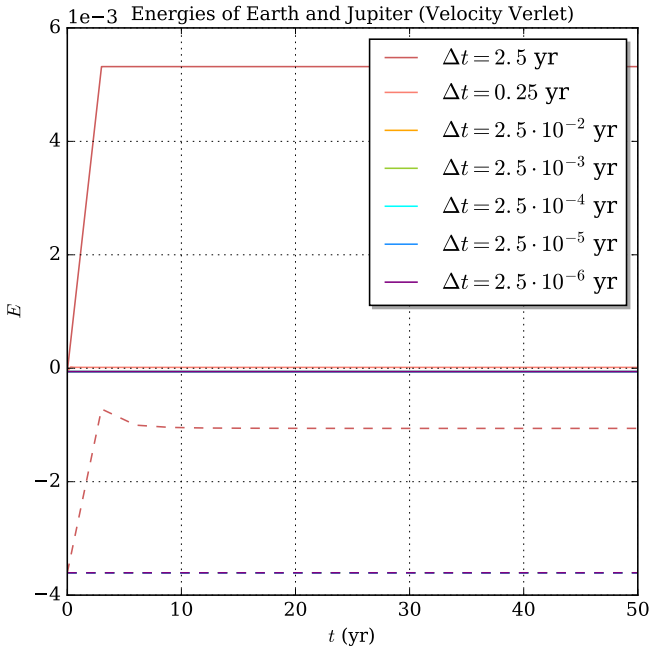


(b) A subfigure

FIG. 3: A figure with two subfigures

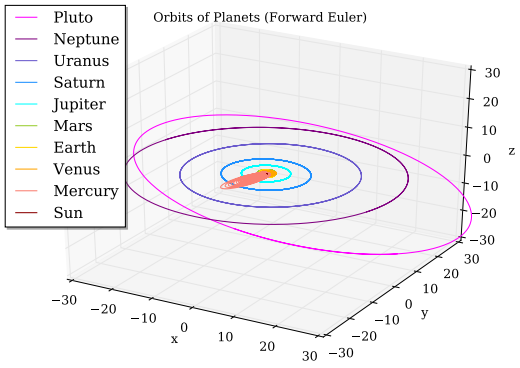


(a) A subfigure

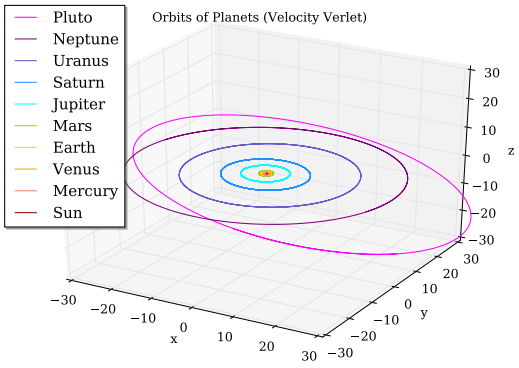


(b) A subfigure

FIG. 4: A figure with two subfigures

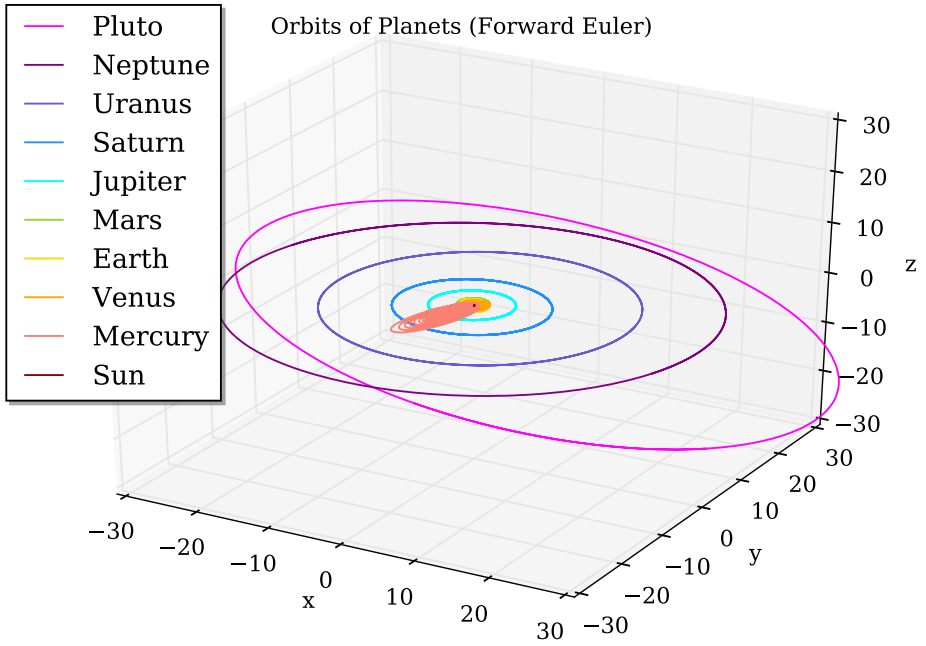


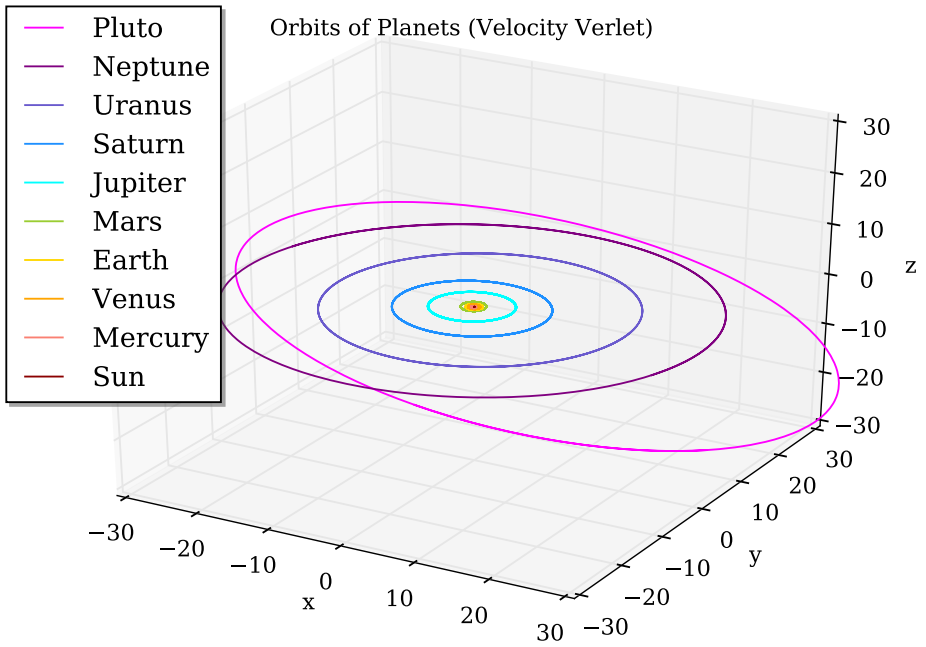
(a) A subfigure



(b) A subfigure

FIG. 5: A figure with two subfigures


Orbits of Planets (Forward Euler)
Pluto
Neptune
Uranus
Saturn
Jupiter
Mars
Earth
Venus
Mercury
Sun
x
y
z


Orbits of Planets (Velocity Verlet)
Pluto
Neptune
Uranus
Saturn
Jupiter
Mars
Earth
Venus
Mercury
Sun
x
y
z