PHY 480 - Computational Physics
Modeling the Solar System

Thomas Bolden

April 4, 2016

Github Repository at https://github.com/ThomasBolden/PHY-480-Spring-2016

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Contents

Introduction

Differential equations can be used to computationally model a diverse range of systems, from harmonic oscillator potentials to astrophysical phenomena. As a result, the ability to solve differential equations is an essential skill in all fields of physics. Using our own solar system as a model, we can test the effectiveness of various methods of solving differential equations.

In this project, I will write code to simulate the nine planets and dwarf planets in our solar system using first the velocity verlet method, then the fourth order Runge-Kutta (RK4) method. I will assume the only force acting on each planet is gravity, obeying Newtonian gravity.

$$F_G = \frac{GMm}{r^2} \tag{1}$$

If then, I assume the solar system to be entirely coplanar (in xy), the following differential equations can be used.

$$\frac{d^2x}{dt^2} = \frac{F_{Gx}}{M} \tag{2}$$

$$\frac{d^2y}{dt^2} = \frac{F_{Gy}}{M} \tag{3}$$

Here, F_{Gx} and F_{Gy} are the x and y components of the gravitational force, respectively, acting on a planet of mass M.

In the report that follows, there are four sections: methods, results, conclusions, and code. In the methods section, I will outline the algorithms used to model the solar system. These include the velocity verlet and RK4. Here, I will also discuss their suitability and shortfalls in the form of truncation errors. In the section that follows (Results), I will present my findings for each part of the assignment. I will give brief explanations for each result obtained. Next, in the Conclusions section, I will generalize my results and say something on the overall effectiveness and viability of each method used to solve the differential equations. I will then say something on what should be done in the future. Finally, in the Code section, I just supply the source code used for the various parts of this project.

Methods

In order to impliment the verlet velocity and RK4 algorithms, the second-order differential equations (2) and (3) must be rewritten in terms of dimensionless variables as a set of coupled first-order differential equations. If we conider only a system with one planet (the Earth-Sun system), we have

$$F_G = \frac{GM_\odot M_E}{r^2} \tag{4}$$

with $M_\odot$ as the mass of the Sun, and M_E the mass of the Earth. Then, in the x direction, the acceleration is given by

$$\frac{d^2x}{dt^2} = \frac{F_{Gx}}{M} = \frac{GM_\odot M_E}{r^2 M_E} = \frac{GM_\odot}{r^2}.$$

Introducing polar coordinates, one gets $F_x = F\cos\theta$ and $F_y = F\sin\theta$. This can be applied to get $F_{Gx} = F\cos\theta = Fx/r$. The final form of the second order differential equations are then

$$\frac{d^2x}{dt^2} = \frac{GM_\odot}{r^3}x. \tag{5}$$

This is similar for the y direction.

$$\frac{d^2y}{dt^2} = \frac{GM_\odot}{r^3}y. \tag{6}$$

In the first order, these equations become

$$\frac{dx}{dt} = v_x \ , \ \frac{dv_x}{dt} = \frac{GM_\odot}{r^3}x \tag{7}$$

and

$$\frac{dy}{dt} = v_y \ , \ \frac{dv_y}{dt} = \frac{GM_\odot}{r^3}y. \tag{8}$$

The distances in space in terms of base SI units are quite large, so it is easier then to use astronomical units (AU) as the standard distance unit, and years (yr) as the standard time unit. From equations (4) and $F_G = M_E v^2/r$, the following relation is true

$$v^2 r = GM_\odot. \tag{9}$$

If we make subsitutions of $v = 2\pi r/\text{yr}$ for the velocity of the Earth in orbit, and $r = \text{AU}$ for the radius of Earth's orbit, we obtain

$$GM_\odot = v^2 r = \left(2\pi\frac{\text{AU}}{\text{yr}}\right)^2 \text{AU} = 4\pi^2\frac{\text{AU}^3}{\text{yr}^2}. \tag{10}$$

From this, equations (7) and (8) become

$$\frac{dx}{dt} = v_x \ , \ \frac{dv_x}{dt} = \frac{4\pi^2}{r^3}x \tag{11}$$

and

$$\frac{dy}{dt} = v_y \ , \ \frac{dv_y}{dt} = \frac{4\pi^2}{r^3}y. \tag{12}$$

With the above differential equations, the important thing now is to impliment these using both the velocity verlet and RK4 algorithms.

The Velocity Verlet Algorithm

The velocity verlet algorithm uses Taylor expansions to rewrite the folloing velocity formula

$$v_{xi} = \frac{x_{i+1} - x_{i-1}}{2h} + O(h^2). \tag{13}$$

In this formula, the velocity has an error growing as $O(h^2)$. Using Taylor expansions on the position and velocity, the following formulae can be obtained:

$$\begin{aligned} x_{i+1} &= x_i + hx_i' + \frac{h^2}{2}x_i'' + O(h^3) \\ v_{i+1} &= v_i + hx_i' + \frac{h^2}{2}x_i'' + O(h^3) \\ &= v_i + \frac{h}{2}(v_{i+1}' + v_i') + O(h^3) \end{aligned}$$

The final equations are then

$$x_{i+1} = x_i + hv_i + \frac{h^2}{2}v'_i + O(h^3) \tag{14}$$

and

$$v_{i+1} = x_i + \frac{h}{2}(v'_{i+1} + v'_i) + O(h^3) \tag{15}$$

There are a couple of important things to notice about these two equations. First, they both have truncation errors of the third degree, growing as $O(h^3)$. Also, in the second equation, v_{i+1} is dependent on x_{i+1}, meaning $x(t_{i+1})$ must be calculated before $v(t_{i+1})$. Below is the pseudocode algorithm to impliment this.

Algorithm 1 Verlet Velocity

function Verlet(x, y, v_x, v_y)
  for $i < i_{\text{max}}$, $i++$ **do**
    $r_i = \sqrt{x_i^2 + y_i^2}$
    $v'_{x,i} = 4\pi^2 x_i / r_i^3$
    $v'_{y,i} = 4\pi^2 x_i / r_i^3$
    $x_{i+1} = x_i + hv_{x,i} + \frac{h^2}{2}v'_{x,i}$
    $y_{i+1} = y_i + hv_{y,i} + \frac{h^2}{2}v'_{y,i}$
    $r_{i+1} = \sqrt{x_{i+1}^2 + y_{i+1}^2}$
    $v'_{x,i+1} = 4\pi^2 x_{i+1} / r_{i+1}^3$
    $v'_{y,i+1} = 4\pi^2 y_{i+1} / r_{i+1}^3$
    $v_{x,i+1} = v_{x,i} + \frac{h}{2}(v'_{x,i+1} + v'_{x,i})$
    $v_{y,i+1} = v_{y,i} + \frac{h}{2}(v'_{y,i+1} + v'_{y,i})$
    $x_{i+1} \to x_i$
    $y_{i+1} \to y_i$
    $v_{x,i+1} \to v_{x,i}$
    $v_{y,i+1} \to v_{y,i}$
  end for
end function

Fourth Order Runge-Kutta Method

Like the verlet velocity algorithm, this method uses Taylor expansions. However, it also uses numberical integral approximations based on Simpson's rule. If we let $dy/dt = \int f(t, y)\,\mathrm{d}t0$, we can make the following estimations to lead to a final estimation of $y(t_i + h)$.

$$k_1 = hf(t_i, y_i) \tag{16}$$

$$k_2 = hf(t_i + \frac{h}{2}, y_i + \frac{k_1}{2}) \tag{17}$$

$$k_3 = hf(t_i + \frac{h}{2}, y_i + \frac{k_2}{2}) \tag{18}$$

$$k_4 = hf(t_i + h, y_i + k_3) \tag{19}$$

Then for the next iteration

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{20}$$

Algorithm 2 Runge-Kutta 4

function RK4(t, y, f)
  for $t < t_{\text{max}}, t++$ **do**
    $k_1 = f(y, t)$
    $y_1 = y + k_1 \frac{h}{2}$
    $k_2 = f(y_1, t + \frac{h}{2})$
    $y_2 = y + k_2 \frac{h}{2}$
    $k_3 = f(y_2, t + \frac{h}{2})$
    $y_3 = y + k_3 h$
    $k_4 = f(y_3, t + \frac{h}{2})$
    $y \leftarrow y + (k_1 + 2k_2 + 2k_3 + k_4)/6$
  end for
end function

The fourth-order Runge-Kutta algorithm gets its name from the global truncation error that grows as $O(h^4)$.

Results

The Earth-Sun System

.

Critical Velocities

.

The Eath-Jupiter-Sun System

.

The Solar System

.

The Perihelion Procession of Mercury - Optional

.

.

Conclusions

The first task was to.

Code

../Code/aster.h

```h
#ifndef ASTER_H
#define ASTER_H
#define _USE_MATH_DEFINES
#include <cmath>
#include <vector>

class body
{
public:
    // Properties
    double mass;
    double position[2];
    double velocity[2];
    double potentialE;
    double kineticE;

    // Initializers
    body();
    body(double M, double x, double y, double vx, double vy);

    // Functions
    double distance(body neighbor);
    double GravitationalForce(body neighbor, double G);
    double Acceleration(body neighbor, double G);
    double KineticEnergy();
    double PotentialEnergy(body &neighbor, double G, double epsilon);

};

#endif // ASTER_H
```

../Code/aster.cpp

```cpp
/*
    aster class based on the planet class by M. Hjorth-Jensen (2016)
*/

#include "aster.h"

/*
aster::planet()
{
    mass = 1.;
    position[0] = 1.;
    position[1] = 0.;
```

```
13      position[2] = 0.;
14      velocity[0] = 0.;
15      velocity[1] = 0.;
16      velocity[2] = 0.;
17      potential = 0.;
18      kinetic = 0.;
19  }
20  */
21
22  aster::body(double M, double x, double y, double vx, double vy)
23  {
24      mass = M;
25      position[0] = x;
26      position[1] = y;
27      velocity[0] = vx;
28      velocity[1] = vy;
29      potentialE = 0.; // is this line necessary ??
30      kineticE = 0.;
31  }
32
33  double aster::distance(body neighbor)
34  {
35      double x1,y1,x2,y2,dx,dy,r;
36
37      x1 = this->position[0];
38      y1 = this->position[1];
39
40      x2 = neighbor.position[0];
41      y2 = neighbor.position[1];
42
43      dx = x1-x2;
44      dy = y1-y2;
45
46      r = sqrt(dx*dx + dy*dy);
47
48      return r;
49  }
50
51  double aster::GravitationalForce(body neighbor, double G);
52  {
53      double r = this->distance(neighbor);
54      if(r!=0.0) return G*this->mass*neighbor.mass/(r*r);
55      else return 0;
56  }
57
58  double aster::Acceleration(body neighbor, double G);
59  {
60      double r = this->distance(neighbor);
61      if(r!=0) return this->GravitationalForce(neighbor,G)/(this->mass*r);
62      else return 0;
63  }
64
65  double aster::KineticEnergy();
66  {
```

```
67      double velocity2 = (this->velocity[0]*this->velocity[0]) + (this->velocity[1]*this->vel
68      return 0.5*this->mass*velocity2;
69  }
70
71  double aster::PotentialEnergy(body &neighbor, double G, double epsilon);
72  {
73      if(epsilon==0.0) return -G*this->mass*neighbor.mass/this->distance(neighbor);
74      else return (G*this->mass*neighbor.mass/epsilon)*(atan(this->distance(neighbor)/epsilo
75  }
```

References

[1] M. Hjorth-Jensen, *Computational Physics*, University of Oslo (2015).