AEP4380 HW5
Three-Body Problem

Dan Girshovich

Mar 7, 2013

1 Overview

This document details the results of a C++ program which calculates the two-dimensional trajectories of three bodies acting under gravity. The system chosen contains the Earth, the Moon, and a fake second moon acting classically. That is, the differential equations that dictate the motion of the bodies are found using Newton's second law and Newton's law of universal gravitation. These are numerically solved with the fifth-order Dormand-Prince method described in section 17.2.3 of Numerical Recipes [1]. Notably, with this method the step size is constantly adjusted based on an estimated error in an effort to minimize computations.

2 Equations of Motion

Each body's motion is defined by the initial conditions:

	Earth	Moon	Moon-2
x	0.0	0.0	-6.0e8
y	0.0	3.84e8	4.05e8
v_x	-12.593	1019.0	500.0
v_y	0.0	0.0	50.0

and Newton's Laws:

$$\frac{d^2\vec{x}_i}{dt^2} = \frac{\vec{F}_i}{m} = G\sum_{j\neq i} m_j \frac{\vec{x}_j - \vec{x}_i}{|\vec{x}_j - \vec{x}_i|^3}$$

Where $\vec{x}_i$ and m_i the position vector and mass of the i^{th} body, respectively, and G is the gravitational constant. As usual, this equation must be rewritten as a set of coupled first-order ODEs in order to use a numerical solver like Dormand-Prince:

$$\frac{d\vec{v}_i}{dt} = \frac{\vec{F}_i}{m} = G\sum_{j\neq i} m_j \frac{\vec{x}_j - \vec{x}_i}{|\vec{x}_j - \vec{x}_i|^3}$$

$$\frac{d\vec{x}_i}{dt} = v_i$$

This notation implies there $2 * N * D$ coupled first-order ODEs, where N is the number of bodies and D is the number of dimensions. In this case N = 3 and D = 2, so there are a total of 12 equations to solve in parallel. Notice that 12 initial conditions were already provided, as required.

3 The Dormand-Prince Method

3.1 Overview

The Dormand-Prince method numerically solves these coupled ODEs within a requested bound of local error. To do this, it uses a fifth-order Runge-Kutta step that generates an approximation for y_{n+1} from y_n. Using the same function evaluations from this step, but with different weight coefficients, an embedded fourth-order solution y^*_{n+1} is also produced. These two estimates are then used to approximate the error for every step in each ODE as:

$$\Delta = |y_{n+1} - y^*_{n+1}|.$$

Using Δ and the user specified error tolerances (absolute and relative), the method automatically updates the step size to minimize computation while staying within the error bound.

Importantly, very little additional computation is required per step to estimate the error compared to an ordinary fifth-order Runge-Kutta method. This is because no extra function evaluations (assumed to be relatively computationally expensive) are performed to estimate the error. Instead, different weights are applied to each function to produce an embedded low-order solution. This, then, provides the necessary information to approximate error. The weights used in this implementation were given by Dormand and Prince, hence the name of the method.

The total computation is actually greatly reduced (at least for the three-body problem given) because of the variable step size. The amount by which the step size varies in this problem is discussed in a later section.

3.2 Implementation

The implementation combines the `Odeint` and `StepperDopr5` objects from Chapter 17 of Numerical Recipes [1] with the `rhs_grav` object in the attached C++ code. The core of the implementation is in the following section of `gen_traj_data`:

```
rhs_grav derivs(m, nbodies);
Doub rtol = 1e-8, abtol = 0.0, hinit = 1.0, hmin = 0.0;
Odeint<StepperDopr5<rhs_grav> > ode(ic, 0, secs, abtol, rtol, hinit, hmin,
                                    out, derivs);
ode.integrate();
```

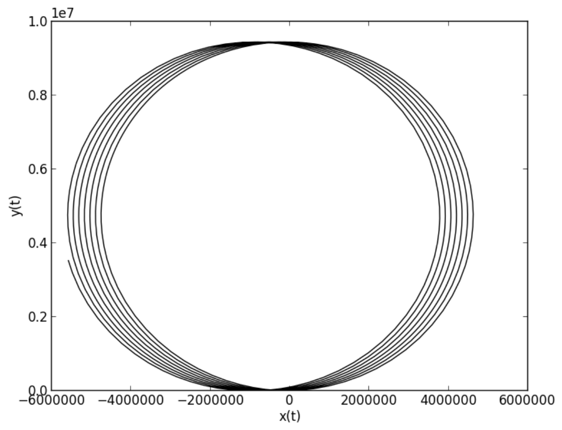
The `rhs_grav` object overrides the () operator, so its instances can act like the `derivs` function expected by the stepper. However, because it is an object, parameters which are not passed to `derivs` by the stepper can be passed to the constructor first. Using this method, information like the number of bodies in the system and the mass of each one is baked into the `derivs` "function". This way, the generic stepper implementation does not need to be modified to solve the differential equations for the three-body system.

Also, note that only a relative tolerance is used. The discussion on page 914 of Numerical Recipes [1] mentions that for sets of equations whose dependent variables vary greatly in magnitude, using only relative error is a good strategy. For the three body problem, some of the equations give positions (order $10^8 m$) and others give velocities (order $10^3 \frac{m}{s}$). So, for this choice of units, using purely a relative error bound is a good approach.

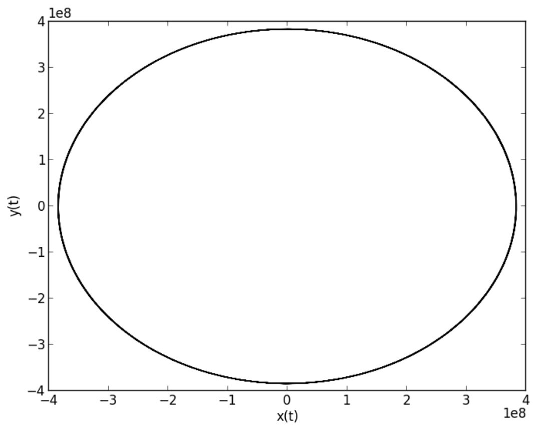
4 Results

4.1 Test System

The familiar Earth-Moon system was used as a test case for the code. This system was calculated by setting the mass of the second moon equal to 0. The calculated trajectories for 200 days are shown below.



(a) Earth



(b) Moon

Figure 1: Trajectories of the Earth and Moon in the two-body system.

As expected the two bodies orbited their center of mass in nearly circular trajectories. Both orbits slightly drift towards the negative x direction, but this is only visible for the Earth due to the scales shown. Looking at the initial conditions, there is a net negative v_x component of the center of mass, which explains the drift. Also, the data for the moon's trajectory shows constant period between 27 and 28 days, which confirms that the total integration error is reasonable over this time scale.

4.2 Three-body system

The full three-body system has a second moon with 0.2 the mass of the real moon. The calculated trajectories for 200 days are shown below. As expected both moons orbited the much more massive earth.

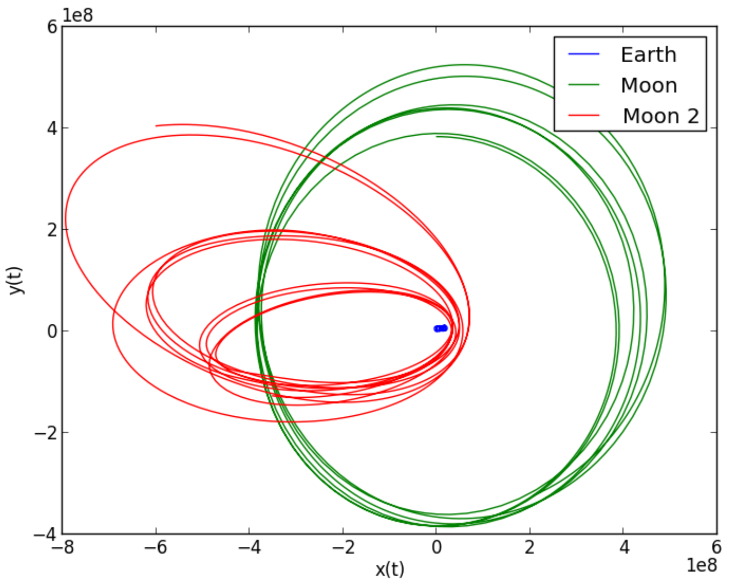


Figure 2: Trajectories in the three-body system.

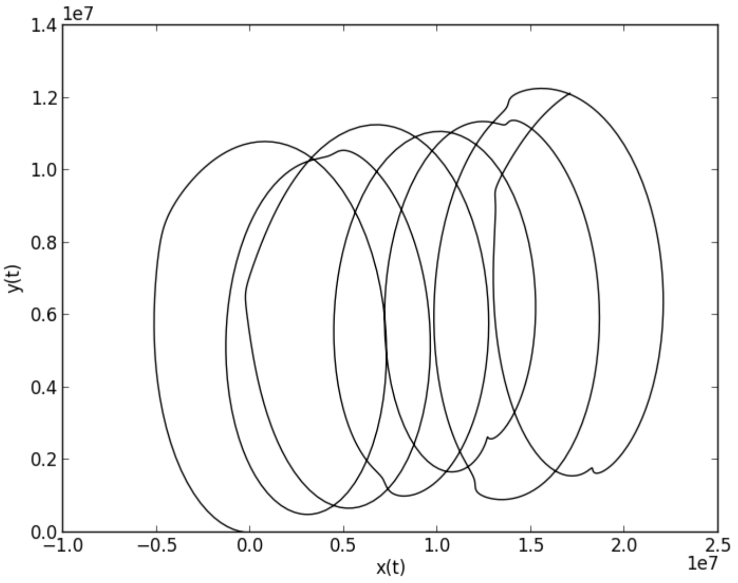
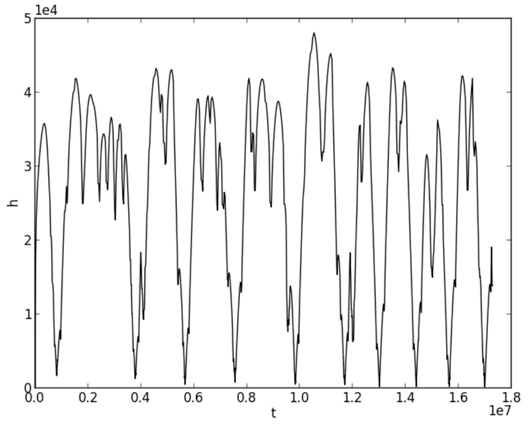
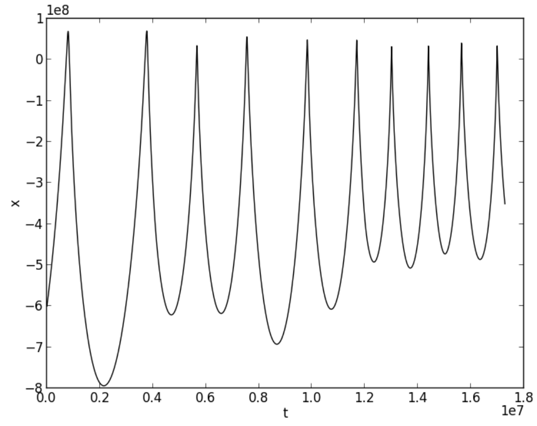


Figure 3: The expanded view of the Earth shows its relatively small motion.

The average step size used by the stepper to produce these trajectories is on the order of 10^4, but the minimum is on the order of 10^2. This minimum represents the step size that would be required by a fixed step size method for it to achieve the same accuracy. So, the variable step size method is more efficient by two orders of magnitude. Of course, this is a very rough estimate and is it only valid for this system. The periodic dips to very small step size are caused by the large force experienced by the second moon as it swings around the Earth. During these times, the second moon's velocity diverges from the linear approximation more than usual. This requires the step size to shrink for the error bound to be maintained. Both the step size variation and the x position of the second moon are shown below.



(a) Step size



(b) x position of second moon

The spikes on both plots neatly align, as expected. Both the plot of the x position of the second moon and the diagram of the full trajectories suggest that the second moon's orbit is stabilizing over time.

5 Source Code

```
/*
AEP 4380 HW #5
Dan Girshovich
3/7/13
Calculates the positions of three bodies acting under gravity.
Compile with: g++ -std=c++0x -O2 -o gen_data hw5.cpp
Tested on Mac OSX 10.8.2 with a Intel Core 2 Duo
*/

#include <cstdio>
#include <cstdlib>
#include <cmath>
#include "nr3.h"
#include "stepper.h"
#include "odeint.h"
#include "stepperdopr5.h"

// type / func defs
void gen_traj_data(Doub, const char *);
void alert_if_collision(Int, Int, Doub, Doub);
FILE *open_file_w(const char *);

const Doub G = 6.6726e-11; // Nm^2/kg^s (Gravitational constant)

const Doub Me = 5.976e24; // kg (Mass of the Earth)
const Doub Mm = 0.0123 * Me; // kg (Mass of the Moon)
const Doub Mm2 = 0.2 * Mm; // kg (Mass of the second Moon)

const Doub Re = 6.387e6; // m (Radius of the Earth)
const Doub Rm = 3.476e6; // m (Radius of the Earth)
const Doub Rm2 = 0.5 * Rm; // m (Radius of the Earth)

int main()
{
    gen_traj_data(200, "out.dat");
}

// functor for the 'derivs' calculation in the n body problem
struct rhs_grav
{
    Int n; // number of bodies
    VecDoub m; // for the n masses
    rhs_grav(VecDoub mm, Int nn) : m(mm), n(nn) {}
    Int x_index(Int k) { return k; }
    Int y_index(Int k) { return k + n; }
    Int vx_index(Int k) { return k + 2 * n; }
    Int vy_index(Int k) { return k + 3 * n; }
    void set_vals(Doub &x, Doub &y, Doub &vx, Doub &vy, Int k, VecDoub &w)
    {
        x = w[x_index(k)];
        y = w[y_index(k)];
        vx = w[vx_index(k)];
        vy = w[vy_index(k)];
    }
    // w should contain {x1,..., xn, y1,..., yn, vx1,..., vxn, vy1,..., vyn}
    void operator() (const Doub t, VecDoub &w, VecDoub &dwdt)
    {
        Doub xi, xj, yi, yj, vxi, vxj, vyi, vyj;
        for (Int i = 0; i < n; i++)
        {
            set_vals(xi, yi, vxi, vyi, i, w);
            Doub sumx = 0, sumy = 0;
            for (Int j = 0; j < n; j++)
            {
                if (i != j)
                {
                    set_vals(xj, yj, vxj, vyj, j, w);
                    Doub d = pow(pow(xi - xj, 2) + pow(yi - yj, 2), 1.5);
                    alert_if_collision(i, j, d, t);
                    sumx += m[j] * (xj - xi) / d;
                    sumy += m[j] * (yj - yi) / d;
                }
            }
            dwdt[x_index(i)] = vxi;
            dwdt[y_index(i)] = vyi;
            dwdt[vx_index(i)] = G * sumx;
            dwdt[vy_index(i)] = G * sumy;
        }
    }
};

// sets the initial positions and velocities for the 3 bodies
void set_ics(VecDoub &ic)
{
    // initial x values
    ic[0] = 0.0;
    ic[1] = 0.0;
    ic[2] = -6.0e8;
    //initial y values
    ic[3] = 0.0;
    ic[4] = 3.84e8;
    ic[5] = 4.05e8;
    // initial vx values
    ic[6] = -12.593;
    ic[7] = 1019.0;
    ic[8] = 500.0;
    // initial vy values
    ic[9] = 0.0;
    ic[10] = 0.0;
    ic[11] = 50.0;
}

// uses odeint with rhs_grav to generate the positions of the three bodies
void gen_traj_data(Doub days, const char *fname)
{
    Doub secs = days * 24 * 60 * 60;
    Int nbodies = 3;
    Output out(0);
    VecDoub m(3);
    // masses of Earth, Moon, and Moon2
    m[0] = Me, m[1] = Mm, m[2] = Mm2;
    VecDoub ic(nbodies * 4); // there are 2 * 2 * N initial conditions
    set_ics(ic);
    rhs_grav derivs(m, nbodies);
    Doub rtol = 1e-8, abtol = 0.0, hinit = 1.0, hmin = 0.0;
    Odeint<StepperDopr5<rhs_grav> > ode(ic, 0, secs, abtol, rtol, hinit, hmin,
                                        out, derivs);
    ode.integrate();
    FILE *fp = open_file_w(fname);
    // print calculated positions of every body for each time step
    for (Int i = 0; i < out.count; i++)
    {
        fprintf(fp, "%f␣%f␣%f␣%f␣%f␣%f␣%f␣\n",
                out.xsave[i], out.ysave[0][i], out.ysave[1][i], out.ysave[2][i],
                out.ysave[3][i], out.ysave[4][i], out.ysave[5][i]);
    }
    fclose(fp);
}

// debugging tool for detecting collisions
void alert_if_collision(Int i, Int j, Doub d, Doub t)
{
    VecDoub r(3);
    r[0] = Re, r[1] = Rm, r[2] = Rm2;
    if (r[i] + r[j] > d)
    {
        printf("Collision:␣%f␣%d␣%d␣\n", t, i, j);
        exit(1);
    }
}

// opens a new file for output
FILE *open_file_w(const char *fname)
{
    FILE *fp;
    fp = fopen(fname, "w+");
    if (NULL == fp)
    {
        printf("cannot␣open␣file\n");
        return (EXIT_SUCCESS);
    }
    return fp;
}
```

References

[1] W.H. Press, S.A Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipies, The Art of Scientific Computing*. Camb. Univ. Press, 3rd Edition, 2007.