Homework 5: Three Body Problem

In this homework, we first practice the implementation of the fifth order Runge-Kutta method with automatic step sizing as outlined in *Numerical Recipes* [1], and then proceed to use it in order to observe the behavior of a hypothetical 3-body problem in which another moon is added to Earth's orbit. First, we start off with the 5th/4th order solution as outlined in section 17.2 of *Numerical Recipes*:

$$\begin{aligned} y_{n+1} &= y_n + b_1 k_1 + b_2 k_2 + b_3 k_3 + b_4 k_4 + b_5 k_5 + b_6 k_6 + O(h)^6 \\ y^*_{n+1} &= y_n + b^*_1 k_1 + b^*_2 k_2 + b^*_3 k_3 + b^*_4 k_4 + b^*_5 k_5 + b^*_6 k_6 + b^*_7 y_{n+1} O(h)^5 \\ k_1 &= h f(x_n, y_n) \\ k_2 &= h f(x_n + c_2 h, y_n + a_{21} k_1 \\ &... \\ k_6 &= h f(x_n + c_6 h, y_n + a_{61} k_1 + ... + a_{65} k_5) \end{aligned} \tag{1}$$

In order to implement this method, I started by allocating enough space for each of the 8 arrays that are used later on in the method (k1-k7 and temp), each of which have a length of neqns, which represents the number of equations in the system. When calculating each k, I use temp to store the value that is evaluated by the system, and then store those values in k. Since every value in the array of each k depends on the values of the array of the previous k, it is necessary to do a complete for-loop for each of them in order to later evaluate y_{n+1}. Finally, once the routine returns all of the values, I then delete k1 in order to allow the program to dynamically allocate the next set of k values.

The auto-step size Runge-Kutta routine departs from the regular Runge-Kutta routine at this point, since once I have acquired y_{n+1}, I then use this value, which corresponds to a 5th order approximation, in order to calculate the 4th order approximation, y^*_{n+1}. Finding the maximum of such error, and then using an appropriate scale to find what *Numerical Recipes* [1] refers to as the "worst offender", or the value for which the two approximations differ the most. We then take this value to reevaluate an appropriate h to be used in the next iteration of the Runge-Kutta, as specified by the equations:

$$\begin{aligned} h_0 &= h_1 |\frac{err_0}{err_1}|^{1/5} \\ err_1 &= max \frac{|y_{n+1} - y^*_{n+1}|}{scale} \end{aligned} \tag{2}$$

where err_0 is the tolerable error, and err_1 is the maximum value of the difference between the two methods divided by a scale, as shown in the equation above. Using this subroutine, I then plug in the system of equations, along with its initial conditions. As a test case, my first system of differential equations was that of a sinusoidal function:

$$\frac{d^2 y}{dt^2} + \omega^2 y = 0 \tag{3}$$

Setting the initial conditions to $y(t = 0) = 1$ and $y(t = 0) = 0$, we expect this set of equations to yield a cosine function after using it as an argument in the ODE solver. The figure below matches this prediction (amplitude = 1, period = 2π):

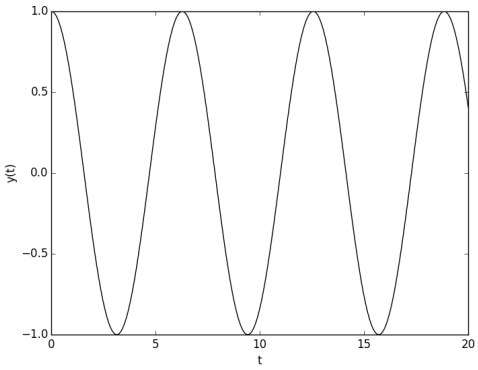

Figure 1: Cosine Function evaluated by ODE autostep solver

After establishing that the ODE solver was working, I then moved on to the Three-Body system, as defined by the equations:

$$\begin{aligned} \frac{dx_i}{dt} &= v_i \\ \frac{dv_i}{dt} &= G\sum_{j\neq i} m_j \frac{x_j - x_i}{|x_j - x_i|^3} \end{aligned} \tag{4}$$

where G is the gravitational constant, m_j is the mass of object j, and the summation of j≠i represents the acceleration contribution of each object j on object i. Using the initial conditions specified in the homework, minus the influence of the second moon, with a starting h step size of 1000, and a tolerance of 10E-08, I was able to generate the following graph for the predicted orbit of the Earth and Moon:

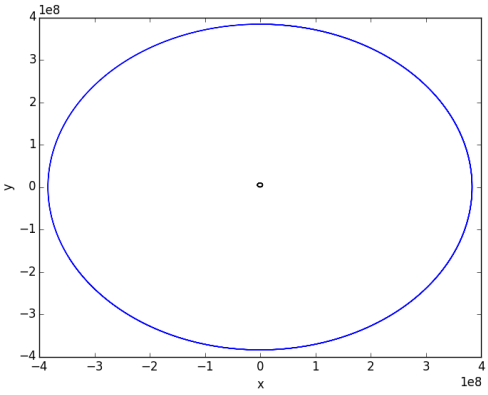

Figure 2: Initial Orbit Attempt. The blue line is the Moon and the black line is the Earth

The stability of these orbits demonstrate an agreement with the normal behavior of the Earth and Moon, indicating that the method is working. In the following table, I list the behavior of h as the method goes along:

h (seconds)	t (days)
10408.7	0
2847.3	25
10855.4	50
10412.2	75
11224.8	100
11029.6	125
10415.4	150
11344.9	175
10922.9	200

Figure 3: Observing the progression of h with t

In the region from about 20 days to 30 days in the orbit, the step size of the function drops to around 3000 seconds, noting that the auto-step size method had difficulty navigating this region, and thus narrowed down the step size to accurately explore it. Thus, if we were to do a regular Runge-Kutta method, we would have to choose a step size of around h = 2000 in order to ensure the disruption of this region did not influence the rest of the behavior of the system; a step size which is 5 times less than the stable values of about h = 10000 that the rest of the system exhibited. This in turn allows us to save a lot of time, by progressing through the problem quickly, without sacrificing stability and accuracy.

After completing this example, I then went on to the Three-Body system, introducing a third moon whose mass was about half that of Earth's actual moon.

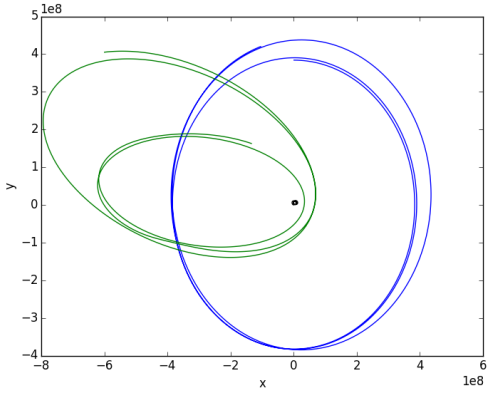

Figure 4: Three Body Orbit. The blue line is the Moon, the black line is the Earth, and the green line is a hypothetical third Moon

This graph demonstrates a notably different behavior from the two body figure in the previous problem. Unlike the stable orbit of the Earth and the Moon, it appears that the second moon is starting to wrap in closer and closer to the Earth. In fact, zooming in closer to the Earth, we see a very notable drift in the positive x-direction:

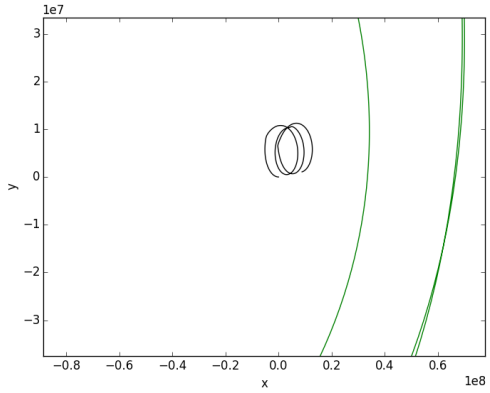

Figure 5: Earth Drift as a result of the second moon

Taking the time span to be even greater, I even found that a collision between the second moon and the Earth would occur within about 500 days. This indicates that the initial conditions on the second moon made it so that it did not have enough energy in its orbit to maintain its elliptical path about the Earth.

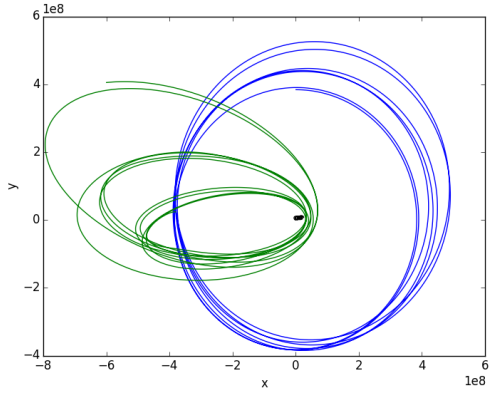

Figure 6: The Second Moon collapsing into the Earth at t = 503 days

The h-behavior in this Three Body system is very different from that of the two body system. Most notably, right near the time of the collision, the h-step size goes down rapidly, as the autostep system struggles to cope with the enormous gravitational force the two celestial bodies exert on one another as they get closer and closer. During the regions while they maintain some semblance of an orbit however, the h values are comparable, if not sometimes greater than that of the two body problem. However, they fluctuate to lower values more often, at the times when the second body becomes close to the Earth, causing the autostep function to refine its step size to better analyze these regions.

h (seconds)	t (days)
10394.2	0
5956.77	25
10871.6	50
11342.5	75
6675.57	100
12472.7	125
4771.71	150
10991.1	175
4220.13	200
12919.3	300
6742.11	400
242.408	500

Figure 7: Observing the progression of h with t

Optional 1: Since it was apparent that given the initial conditions of this problem, the second moon did not have enough energy to maintain its orbit, I decided to change the initial conditions to see if I could maintain its orbit for at least 1000 days. By increasing its inital (x,y) velocity to (800, 500) m/s, I was able to make it maintain a suitable orbit for 1000 days without causing a collision. However, the earth in this system still began to drift, now with an additional y velocity, as shown in the graph below:

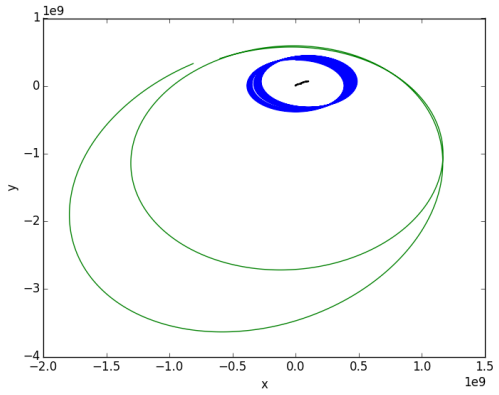

Figure 8: "Fixed" Three Body Problem. No collision at t = 1000 days, though Earth still drifts, now with a small +y component.

This behavior indicates just how precarious the orbits of a system can be, and how extraordinary it is that we managed to end up in such a stable gravitational system.

Source Code Listing

```
/*
AEP 4380 Homework 5

Three Body Problem

Jack Newman October 14, 2015

Last Edited: October 23, 2015

OS: Windows 8

Intel(R) Core(TM) i7-4810MQ CPU @ 2.80 GHz

MinGW C++ compiler, GCC 4.8.1

Note: Comments are either inline or below the lines they describe
*/

#include <cstdlib> // importing plain c
#include <cmath> // math library

#include <iostream> // stream IO
#include <fstream> // stream file IO
#include <iomanip> // to format the output

using namespace std;

double w = 1.0;
bool collision;
double const msd2 = 7.46496E9, msd = 86400;
double const M[] = {5.976E24, 0.0123*5.976E24, 0.0123*5.976E24*0.2};//0.0213*5.976E24*0.2}; // mass constants
double const Tm = 648, G = 6.6726E-11; // orbital period moon, gravitational const
double const R[] = {6378000.0, 3476000.0, 0.5*3476000.0}; // radius constants in m
double const c2=0.2,c3=0.3,c4=0.8,c5=8.0/9.0,a21=0.2,a31=3.0/40.0,
a32=9.0/40.0,a41=44.0/45.0,a42=-56.0/15.0,a43=32.0/9.0,a51=19372.0/6561.0,
a52=-25360.0/2187.0,a53=64448.0/6561.0,a54=-212.0/729.0,a61=9017.0/3168.0,
a62=-355.0/33.0,a63=46732.0/5247.0,a64=49.0/176.0,a65=-5103.0/18656.0,
a71=35.0/384.0, a72=0.0, a73=500.0/1113.0,a74=125.0/192.0,a75=-2187.0/6784.0,
a76=11.0/84.0,e1=5179.0/57600.0,e2=0.0,e3=7571.0/16695.0,e4=393.0/640.0,
e5=-92097.0/339200.0,e6=187.0/2100.0,e7=1.0/40.0;
// establishing the constants and parameters used later in the program
const int N1 = 3; // number of objects
const int N = 2*2*N1;
//const int N = 2*2*N1; // number of ODE
//evaluate the ODE right hand side
//change this for each different ODE


void myrhs( double y[], double t, double f[]){
int istep, j;//right hand side of the Rossier system.
//bool collision = false;
double d, dx, dy;
//cout << "call myrhs()\n" << endl;
for(istep=0;istep<N1; istep++){
//y[i] x pos. of ith object
//y[i+N1] y pos. of ith object
//y[i+2*N1] vx velocity of ith object vxo
//y[i+3*N1] vy velocity of ith object
f[istep] = y[istep+2*N1]; // dx/dt
```

```
f[istep + N1] = y[istep+3*N1]; // dy/dt
f[istep+2*N1] = 0;
f[istep+3*N1] = 0;
for(j=0; j<N1; j++){ // + dy^2
if( j != istep ){
dx = (y[j]-y[istep]);
dy = (y[j+N1]-y[istep+N1]);
d = sqrt(dx*dx + dy*dy);
f[istep+2*N1] = f[istep+2*N1] + G*M[j]*dx/(d*d*d); // dvx/dt
f[istep+3*N1] = f[istep+3*N1] + G*M[j]*dy/(d*d*d); //dvy/dt
//cout << "y[4] =" << y[4] << endl;
if (d < (R[istep] + R[j])){
collision = true; //register if there is a collision
}
}
}
}
return;
}

void myrhs1( double y[], double t, double f[]){
//right hand side of the simple harmonic oscillator
f[1] = -w*w*y[0]; // f[1] = dy_1/dt
f[0] = y[1]; // f[0] = dy_o/dt
return;
}



void ode45(double yold[], double ynew[], double ynew2[], double yerr[], double h,
 double t, int neqns, void myrhs(double[], double, double[])){
// ODE solver using 5th/4th Order Runge-Kutta AutoStep Method
// yold[] is the array which stores the initial y-values of the equation
// ynew[] stores the new y-values once the code has completed
// h is the step size
// t is the initial t value put into the ODE solver
// neqns is the number of equations that are needed to describe the system
// myrhs are the equations which define the system

// ODE solver using 4th Order Runge-Kutta
int i; // represents a counter for the number of equations
double *k1, *k2, *k3, *k4, *k5, *k6, *k7, *temp;
//initializing the 8 arrays needed for the 5th order method

k1 = new double[8*neqns];
// dynamically allocate 8*neqns memory locations for further use
if(NULL == k1){ //check that array is allocated
cout << "Cannot allocate k1 in ode4" << endl;
exit(0);
}
k2 = k1 + neqns;
k3 = k2 + neqns;
k4 = k3 + neqns;
k5 = k4 + neqns;
k6 = k5 + neqns;
k7 = k6 + neqns;
temp = k7 + neqns;

myrhs(yold, t, k1);
```

```
// first, evaluate the rhs equations at t, and then store the values in k1

for(i = 0; i < neqns; i++){
temp[i] = yold[i] + a21*h*k1[i];
}
myrhs(temp, t + c2*h, k2);
for(i = 0; i < neqns; i++){
temp[i] = yold[i] + h*(a31*k1[i] + a32*k2[i]);

}
myrhs(temp, t + c3*h, k3);
for(i = 0; i < neqns; i++){
temp[i] = yold[i] + h*(a41*k1[i] + a42*k2[i] + a43*k3[i]);
}
myrhs(temp, t + c4*h, k4);
for(i = 0; i < neqns; i++){
temp[i] = yold[i] + h*(a51*k1[i] + a52*k2[i] + a53*k3[i] + a54*k4[i]);
}
myrhs(temp, t + c5*h, k5);
for(i = 0; i < neqns; i++){
temp[i] = yold[i] + h*(a61*k1[i] + a62*k2[i] + a63*k3[i] + a64*k4[i] + a65*k5[i]);
}
myrhs(temp, t + h, k6);
for(i = 0; i < neqns; i++){
ynew[i] = yold[i] + h*(a71*k1[i] + a72*k2[i] + a73*k3[i] +
a74*k4[i] + a75*k5[i] + a76*k6[i]);

// Use the calculated k-values to find the new ynew values.
}
myrhs(ynew, t + h, k7);
for(i = 0; i < neqns; i++){
ynew2[i] = yold[i] + h*(e1*k1[i] + e2*k2[i] + e3*k3[i] +
e4*k4[i] + e5*k5[i] + e6*k6[i] + e7*k7[i]);
//calculate ynew and k-values to find ynew2, the 4th order solution
}

for(i = 0; i < neqns; i++){
yerr[i] = abs(ynew[i]-ynew2[i]);
//find the difference between the 5th and 4th order solution
}
// The above steps represent the 5th/4th Order Runge-Kutta formulas as
// described in Numerical Recipes

delete[] k1; // clear the memory so that new values can be stored.
return;
}//end ode45

double maxi(double *arr, int neqns){
double scale[] = {6378, 3E8, 3E8, 6378, 3E8, 3E8, 10, 500, 500, 10, 500, 500};
//scale used to create a reasonable relative error for each of the values
int i;
double max = arr[0]/scale[0];//scale the difference
for(i = 1; i < neqns; i++){
        if(max < arr[i]/scale[i]){
         max = arr[i]/scale[i];
        }//sort through the arrays and get the maximum error/scale
    }
    return max;
}
```

```
int main()
{
ofstream fp;

fp.open( "example5odea.dat" ); //open new file for graph
if( fp.fail() ) { // or fp.bad()
cout << "cannot open file" << endl;
// catches fp.open failure, and returns appropriate error
return( EXIT_SUCCESS );
}

int i; // counter
double yold[N], ynew[N], yerr[N], ynew2[N];
double tmin = 0, h=1000.0, tol = .00000001, tmax=200.0*msd*5, t, max;
//set the initial values of the equation

yold[0] = 0.0; yold[1] = 0.0; yold[2] = -6.0E8;
yold[3] = 0.0; yold[4] = 3.84E8; yold[5] = 4.05E8;
yold[6] = -12.593; yold[7] = 1019.000; yold[8] = 800.0000;
yold[9] = 0.0; yold[10] = 0.0; yold[11] = 500.0000;
// initialize yold[]
t = tmin;
while(t<tmax){//until we reach tmax
ode45( yold, ynew, ynew2, yerr, h, t,
N, myrhs);
max = maxi(yerr, N);
//cout << yold[2] << endl;
//exit(0);
cout << "max = " << max << endl;
//implement the ODE solver at each step

if(max < tol){ //step succeeded
cout << "STEP SUCCESS" << endl;;
     h = h*pow(tol/(max), 0.2);
     //if the step works, adjust h according to the ratio of tol to max
cout << "h =" << h << endl;
     t = t + h;
cout << "t =" << t << endl;
     for(i = 0; i < N; i++){
yold[i] = ynew[i];
//Set values of yold to ynew, iterating to the next step

fp << h << setw(15) << t/msd << setw(15) << ynew[0] << setw(15) <<
ynew[1] << setw(15) << ynew[2] << setw(15) << ynew[3] <<
setw(15) << ynew[4] << setw(15) << ynew[5] << setw(15) <<
ynew[6] << setw(15) << ynew[7] << setw(15) <<
ynew[8] << setw(15) << ynew[9] << setw(15) <<
ynew[10] << setw(15) << ynew[11] <<
endl;
//store values in array for plotting and tables
}
     }

     else{// if maximum error is greater than the tolerance
cout << "STEP FAILURE, max = " << max << endl;
     h /= 5.0; //reduce the step size of h
```

```
     if(abs(h) < 1E-20){
     cout << "h too small, abort program" << endl;
     return( EXIT_SUCCESS );
     //if h is trending to 0, abort the program, and return error message
}
}

if(collision == true){//if the radius of the objects come within d of each other
cout << "collision occurred at t =" << t/msd << endl;
fp.close();
exit(0);
//print error message, indicating the time of the collision and exit.

}
cout << setw(15) << t/msd << setw(15) << ynew[9] << setw(15) << ynew[10] <<
setw(15) << ynew[11] << endl;
}
fp.close();

return(EXIT_SUCCESS);
}
```

hw5plotN.py

```
# uses numpy, scipy, matplotlib packages
from pylab import *
from openpyxl import *
from mpl_toolkits.mplot3d import Axes3D

# read data file x and two y values
data1 = loadtxt( "example5odea.dat", 'float' )
fig = figure()
plot(data1[:,2], data1[:,5], 'k-')#(x,y) vs t of Earth is the Black line
plot(data1[:,3], data1[:,6], 'b-')#(x,y) vs t of the Moon is the Blue line
plot(data1[:,4], data1[:,7], 'g-')#(x,y) vs t of the second line is the Green line
xlabel("x")
ylabel("y")

show()
```

References

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