AEP 4380: Homework 5

Gregory Kaiser

October 21st, 2019

1 Problem Background and Solution Overview

This assignment asked that I numerically calculate the orbits of the Sun, Earth, Mars, and Jupiter using a variable step-size Runge-Kutta approach by directly calculating the planets' gravitational interaction. My solution is adapted from last week's assignment, the constant step-size fourth-order Runge-Kutta method, but involves a number of changes due to the added complexity of using a variable step size.

Because I wanted to be able to change my error values for a specific problem, I opted to check my error values and change *h outside* of my general Runge-Kutta stepper. In this way I can tweak my conditions for what constitutes "too small" a step without ruining my stepper function if I want to use it later for another problem.

The stepper function involves a more complicated set of constants for weighting the Runge-Kutta steps during accumulation. My main() function takes care of initialization, the error checking and variable h mentioned above, and outputting data. Two nested for loops calculate forces applied to each body by every other body in my derivatives function.

2 Solution Description

2.1 Runge-Kutta Stepper with 4th and 5th Order Solutions Using Dormand-Prince Coefficients

My stepper method follows section 17.2 in Numerical Recipes[1] which describes a Runge-Kutta method with embedded 4th and 5th order solutions. The Dormand-Prince coefficients were used to weight the values of k_n during the accumulation step. The derivatives function $f(t, y)$ is used to calculate each k_n, the predicted next value of the state array (the fifth order solution) y_{n+1}, and the fourth order solution y^*_{n+1} according to the following formulas:

$$\begin{aligned}
k_1 &= \frac{K_1}{h} = f(t_n, y_n) \\
k_2 &= \frac{K_2}{h} = f(t_n + c_2 h, y_n + a_{21} h k_1) \\
&... \\
k_6 &= \frac{K_6}{h} = f(t_n + c_6 h, y_n + a_{21} h k_1 + a_{61} h k_1 + a_{62} h k_2 + a_{63} h k_3 + a_{64} h k_4 + a_{65} h k_5) \\
k_7 &= \frac{K_7}{h} = f(t_n + h, y_{n+1})
\end{aligned} \tag{1}$$

where $f(t, y)$ is just $derivs(t, y, dy/dt)$. The next state of the system to fifth and fourth order can be calculated as:

$$\begin{aligned}
y_{n+1} &= y_n + h(b_1 k_1 + b_2 k_2 + b_3 k_3 + b_4 k_4 + b_5 k_5 + b_6 k_6) \\
y^*_{n+1} &= y_n + h(b^*_1 k_1 + b^*_2 k_2 + b^*_3 k_3 + b^*_4 k_4 + b^*_5 k_5 + b^*_6 k_6 + b^*_7 k_7)
\end{aligned} \tag{2}$$

I did not use the textbook example code, because I don't understand their code most of the time, making it impossible for me to debug. I used a similar method from the last assignment, generating a new array with all of the arrays that I need of the correct size, filling and using them, then deleting the whole array at the end of a step.

To store the Dormand-Prince coefficients, I generated separate arrays for a_{ij}, b_i, b^*_i, and c_i. By creating slightly larger arrays than necessary (a matrix in the case of a_{ij}), I was able to place the coefficients in the spot to which their subscript corresponds. Some of the values are therefore not used at all, leading to some very small space inefficiency (very worth it for debugging the stepper). Minor Note: Samuel Feibel and I worked together on generating the constants in code, since entering in all the numbers was tedious. Those values can be found either in my source code or on page 913 of Numerical Recipes.[1]

2.1.1 Testing

To make sure that this part was working, I tried to simulate the harmonic oscillator accurately, with a constant step size (just using the higher order Runge-Kutta method). I found an error in my initialization of the Dormand-Prince coefficients, which took a really long time to find! This unit testing was very much worth the time spent on it.

2.2 Variable Step Size and Error Normalization

One modification to ghk_rk4 was that ghk_rk45_dp outputs an array of error estimates corresponding to each predicted value of the next state of the system. The error array follows the following formula:

$$err_{n+1} = \frac{|y_{n+1} - y^*_{n+1}|}{|2\epsilon + y_n * \epsilon|} \tag{3}$$

Where ϵ was about 1e-6, and err_{n+1} is the normalized error for each variable and at each step.

I chose to normalize the errors as they are produced, using a scale factor similar to the one described in equation 17.2.8 of Numerical Recipes[1]. This error scaling makes sure that the errors between each variable are comparable in magnitude, since they are divided by the state of the system (a number which should be similar in magnitude to the predicted state) multiplied by a small factor (which should be a value similar to the error, if I've done my job right). Without this step, comparing the errors directly is meaningless, and won't give good information about which error is problematic.

From that array of errors, I chose the maximum in main() and decide whether or not my step size needs to change based on that value. This allows flexibility later if I want to implement other subroutines for error calculation (e.g. taking an Euler norm, as described in equation 17.2.9 of Numerical Recipes[1]).

If $err_{n+1}^{max} < 1$, that means that my next step is acceptable, and so I can send that new step to a text file and update the state with that predicted value ($y_n \leftarrow$y$_{n+1}$). Time is then propagated forwards, since that value of h was a good one.

If $err_{n+1}^{max} \ll 1$, which I define to be $err_{n+1}^{max} < .0001$, that means that I'm probably using an inefficiently-low value of h. I update h with a bigger value, corresponding to the formula:

$$h_{new} = h_{old}|\frac{err_{new}}{err_{old}}|^{1/5} \tag{4}$$

Where err_{new} is 1.

This formula comes from Numerical Recipes equation 17.2.10[1], and assumes that the error values are on the order of h^5. That means that while the previous value of h gave the error err_{old}, a new value of h might get you an error on the order of 1, our maximum scaled error tolerance.

Finally, if $err_{n+1}^{max} > 1$, that means that I took a bad step and should try again with a smaller h value. My program decreases h by a factor of five, does not update time, and does not update the state, allowing the next call to ghk_rk45_dp to be the same as the last, with only h changed.

A while loop in main() makes sure that time propagates from the specified starting and ending times. With a constant step size, time could be updated more exactly, with only one roundoff error inside each use of the Runge-Kutta stepper. However, since the step size is variable, t must accumulate based on steps taken. Hopefully, the fact that errors are now being monitored should correct for this multiple round-off error.

2.2.1 Testing

Debugging this portion was the worst part of the assignment by far. Most of my problems came from h being way too small, leading to long simulation times, or h being too large, leading to inaccuracy and diverging solutions. Even with the harmonic oscillator, debugging this took a lot of grit and printing values to the console. I found that the decision over the scaling factor for error takes more consideration and care than I anticipated. Many times, errors which should have been acceptable were not, and vice versa. I also caught an important error in my Runge-Kutta stepper, where I completely forgot to take the $b_7^* hf(x_n + h, y_{n+1})$ term into account. For some reason, this caused no issues with my harmonic oscillator test, but did cause problems during error calculation.

The harmonic oscillator output looks the same, but I did print out the value of h whenever it was increased or decreased for one run, to show that dynamic h was working correctly. If I started h pretty high, it would drop way down before stepping forward, and if I started it low it would climb steadily while outputting values.

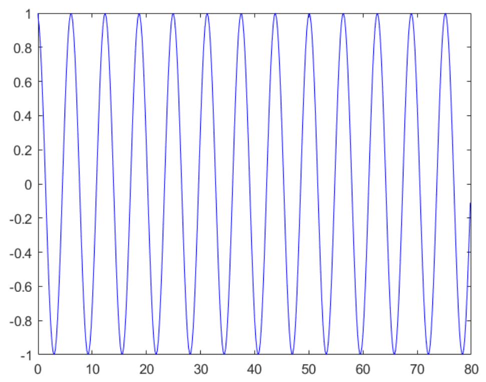
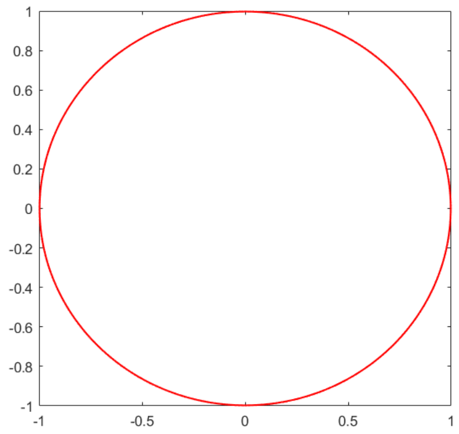

Figure 1: Harmonic Oscillator propagation with variable step-size. Left: Position as a function of time. Right: Phase Space plot of velocity versus position. This closes in a circle as it should for an undamped system.

2.3 N-Body Derivatives Function

As advised, I stored the positions and velocities of each body in sequence in a single array called *state*, allowing for expansion of the number of bodies in the future. Each degree of freedom was given a chunk of the array, and was as long as the number of bodies, creating an array of length $dof * n_{bodies}$. To easily access the values of each body, the objects had an index i and the degrees of freedom had offsets corresponding to which degree of freedom was being accessed.[2]

I tried my best to make a derivatives function that was as general as possible. By modifying the variable n_bodies, the offset values change accordingly. Then, the derivatives function steps through every body i, and each body j in an embedded for loop to calculate the interaction force between body i and body j according to the gravitational force law below:

$$\frac{d\vec{v}_i}{dt} = \frac{\vec{F}_i}{m_i} = G\sum_{j\neq i}^{n_{bodies}} m_j \frac{\vec{r}_j - \vec{r}_i}{|\vec{r}_j - \vec{r}_i|^3} \tag{5}$$

$$\frac{d\vec{r}_i}{dt} = \vec{v}_i \tag{6}$$

Equation 6 is simply updating the state of the velocities of each object in the new state array. Equation 5 is slightly more complicated:

Taking advantage of Newton's Third Law, the interaction force between i and j need only be calculated once. The force simply needs to be scaled by a given object's partner's mass to be added to the sum of the forces on that object. This intermediate value is added to the sum of the forces on object i, and subtracted from the sum of the forces on object j, since $F_{ij} = -F_{ji}$. Since I loop j from $i + 1$ to the number of bodies in the system, once the j loop is finished for a given value of i,

i's forces have fully been summed. At this point I need only multiply by the gravitational constant G to get the value I need to assign to $\frac{d\vec{v_i}}{dt}$, also known as dstate_dt[i+ovx] where ovx (or ovy) is the offset variable described above. The summing variables for object i can then be zeroed for use on the next step.

2.3.1 Testing

Testing for this function was performed by initializing the four body system described below. See the testing section there for details.

2.4 Four-Body System Setup

Using the N-body derivative function, I generated the state array necessary for the four-body simulation that I was asked to perform in main(). I decided to order every array starting from the center of the solar system: Sun, Earth, Mars, Jupiter. The masses of each body were stored in an array at the top of the program for reference inside the derivatives function.

Other than the different starting and ending times, initial h values, and error parameters, propagation of the system follows the harmonic oscillator setup exactly. Section 2.2 describes the variable step-size algorithm.

2.4.1 Testing

At first, my results were very confusing, and I started by looking for errors inside my Runge-Kutta propagation method and my h value-changing logic. However, I found that my implementation of the derivative function was flawed. I forgot to zero the sum of the forces for a given object after that time step completed. This led to some interesting errors, where the sum of forces over several steps was accumulating. This makes sense of my error graphs, where the planets shoot towards one another, and then past each other into interstellar space.

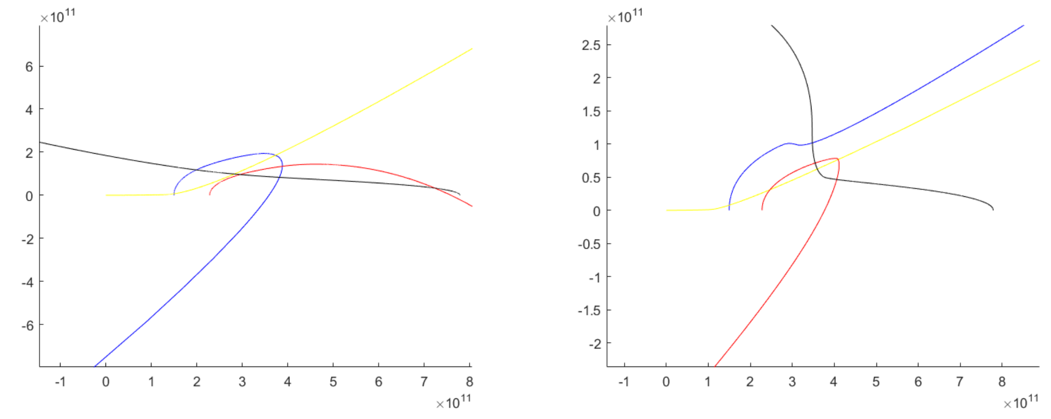

Figure 2: My first and second attempts at the four body simulator, zoomed close to starting points. The bodies flew far away from these positions at high t value. My apologies for using yellow to denote the Sun's trajectory. Sun: Yellow, Earth: Blue, Mars: Red, Jupiter: Black.

3 Results and Interpretation

3.1 Stable Orbits

The initial conditions for stable orbits were given to me, and they produced closed circular orbits. These make sense given that our current solar system is stable over long periods of time, especially over a period of 25 years, which was the simulated time for my testing. Since Jupiter's orbit is roughly 12 years[3], it also made sense that I saw two black circle traces on the plot.

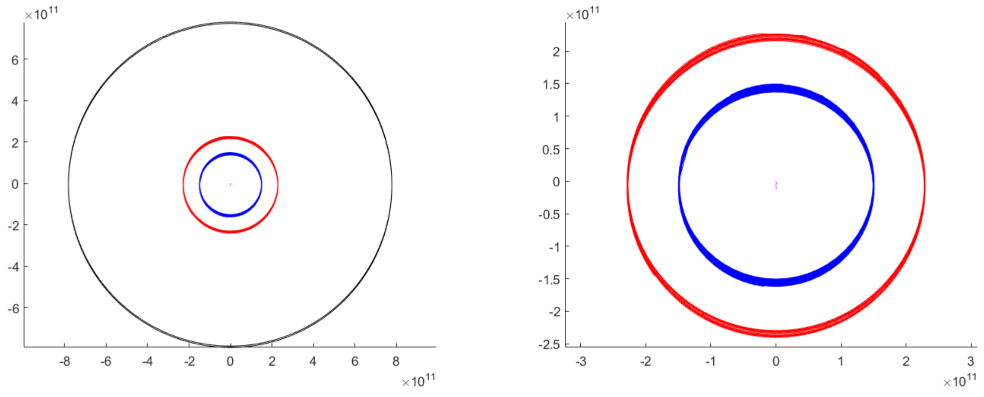

Figure 3: The Sun, Earth, Mars, and Jupiter system propagated for 25 years. Left: full view. Right: zoomed to Earth, Mars, Sun system. Notice the thicker lines at high and low y values, caused by the y-velocity drift (from net y momentum due to initial conditions discussed below). Sun: Magenta, Earth: Blue, Mars: Red, Jupiter: Black.

At first I thought that the lines were too jagged, and that my h values must be too high. However, I trust that my h value is being tuned to give optimal calculation time while maintaining my specified error margin. Errors on the scale of the solar system can be "large" as interpreted by me, as a human, but once you zoom out to the right scale it makes sense that these perturbations are small compared to the full orbit.

3.1.1 Sun's Motion and Net Momentum

I allowed for the center-of-mass motion of the Sun, which produced the following trace over a 25 year span:

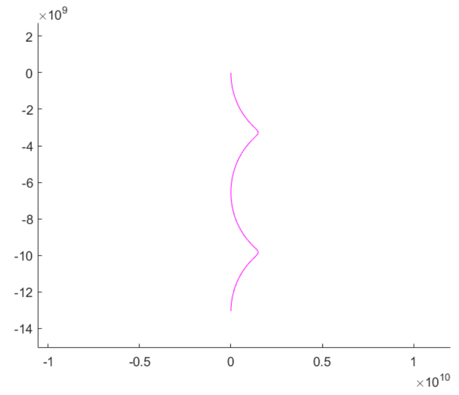

Figure 4: The Sun's movement during the 25 year period. Notice a small wobble that drifts downwards from (0,0).

It is also clear that the orbits of each planet are slowly shifting towards towards the negative y direction, following the Sun's motion, as they should be. This implies that there is net y-momentum downwards given the initial conditions of the system. Through a simple calculation using the constants and initial conditions given:

$$\vec{v}_{sys,y,0} = \frac{1}{m_{sys}}(m_S\vec{v}_{S,y,0} + m_E\vec{v}_{E,y,0} + m_M\vec{v}_{M,y,0} + m_J\vec{v}_{J,y,0}) \approx -17.42\frac{m}{s} \tag{7}$$

Where m_{sys} is the sum of all the body masses. By inspection, this matches what I observe in terms of the shift downwards, since the Sun seems to be drifting about 13e9 meters in 25 years, corresponding to a system velocity of $\approx -16.5\frac{m}{s}$. This also makes sense since the center of mass of the system essentially follows the Sun.

The two main "bumps" in the motion of the Sun must correspond to it's most significant gravitational partner: Jupiter. It orbits twice in 25 years, and hence there are two main bumps in the Sun's trajectory.

3.2 Jupiter Shifted and Accelerated

After lowering the initial orbital radius of Jupiter by a factor of 4.8, and increasing its velocity in the y direction by $\sqrt{4.8}$ to keep a relatively circular orbit, the following trajectories are produced:

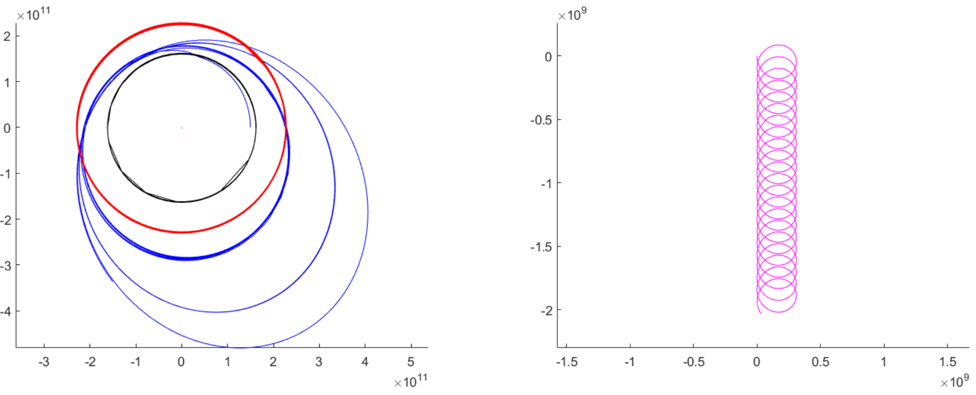

Figure 5: Jupiter's orbit radius and speed changed, causing instability of Earth's orbit within 25 years. Right: Lower orbital period of Jupiter causes faster (and pretty) changes in the Sun's trajectory. Sun: Magenta, Earth: Blue, Mars: Red, Jupiter: Black.

It is clear that Earth gets launched from its normal circular orbit into an increasingly elliptical one. Plotting for longer times gets messy, but shows more instability in Earth's orbital trajectory. Fortunately, it stays in some kind of orbit instead of being ejected from the system, but who knows what kinds of things happen in reality with such a crazy path around the Sun.

This makes some sense energetically, since larger objects should tend to orbit further from a host star for given angular momentum. If they orbit too close to smaller bodies, they'll tend to capture objects to mess with their orbits. I know that Jupiter, even at its stable orbit now, has such a strong gravitational effect on other smaller bodies that it collect asteroids along its path, and has protected us from heavy bombardment on Earth. It makes sense that changing that very important piece of mass messes with other planets' stability as well.

If a stable set of initial conditions is necessary to give a planet long enough time to develop life, then of course we should expect to have such a system as our own. In essence, we aren't lucky, since this setup is inevitable if we exist to observe it.

3.3 Changing Initial Conditions

As a bonus I tried playing around with a few initial conditions to see what would happen. Most of the time, scaling the orbits a bit didn't change much over 25 years, as long as I kept the orbit of Jupiter outside of the orbit of Earth and Mars.

I slowly started shifting Jupiter's initial y position, in order to induce an elliptical orbit, which was successful:

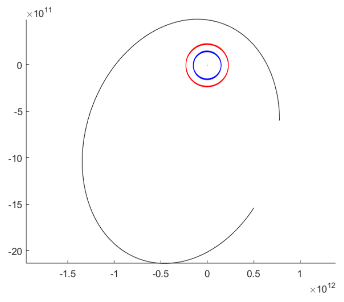

Figure 6: Changing Jupiter's initial y position gives it a more elliptical orbit without creating instability over 25 years.

I then shifted the initial x position of Jupiter closer to Mars's orbit of around 350e9 meters, producing an even more eccentric orbit:

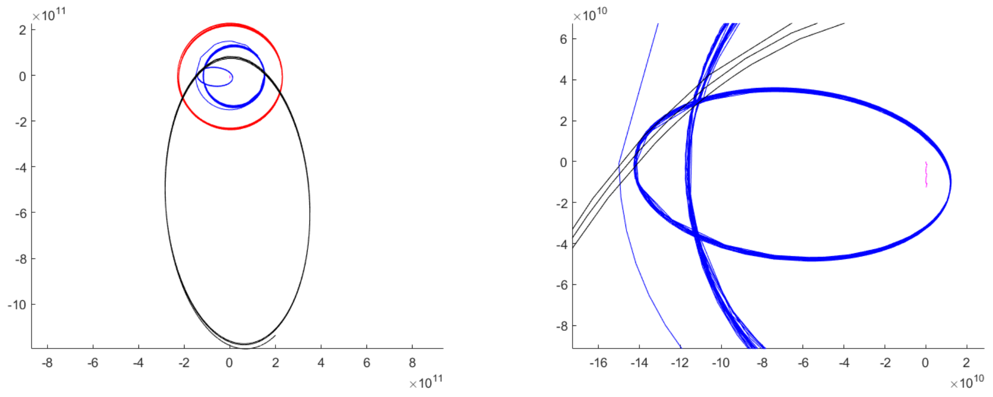

Figure 7: Left: Highly elliptical orbit of Jupiter with shifted x and y positions. Right: The smaller elliptical orbit that the Earth was flung into when Jupiter came by for a visit.

I particularly like how this set of constants has Jupiter "grab" Earth on one of its fly-bys and kick it into a highly elliptical orbit around the Sun. The Earth sits there pretty consistently while Jupiter is on its long trip out to the rim.

The possibilities are endless, but using Jupiter's initial conditions, due to its mass, seems to be the best way to get different results each time.

3.4 Potential Extensions

Because these reports and plots are static depictions of a system that propagates in time, I also took it upon myself to try to animate the trajectories of the planets. This requires some tricky math with the pause() function in MATLAB, since the time steps change throughout the data. It wasn't much of a help for debugging the system, since I did this after finishing the project, but it was certainly helpful with my intuition about how fast the planets orbit relative to one another. It was also better to show to friends!

I began looking into ways of scaling this project up a bit, and found that on the National Aeronautics and Space Administration's (NASA) Jet Propulsion Laboratory (JPL) website, something called ephemeris data is stored and published.[4] This could be used to create a full simulation of many of the bodies in our solar system at once. Of course, this requires that I have the patience to enter all of the necessary constants. Maybe over a school break this would be a fun thing to do.

Even further, Prof. Kirkland mentioned in class the idea of simulating entire galaxies of bodies and then colliding them. This would perhaps be even easier than entering in a bunch of data about our solar system, since the positions and velocities of each object could be somehow procedurally generated. This could also be used to simulate the formation of the solar system itself, by allowing small objects to blend into a single more massive object under some conditions.

This particular method, the variable step size Runge-Kutta solver, seems very useful in my future endeavors in any kind of situation where a model of a dynamic system is required. As long as I can specify the components of a system accurately, it seems that I can propagate it forward arbitrarily far. While initially entering this class I thought that simulating something like a double pendulum would be a culminating project, due to its chaotic nature. Now, I feel that simulating such a system is almost trivial!

References

[1] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes, The Art of Scientific Computing.* (3rd edit.), Cambridge Univ. Press, 2007, (ISBN 978-0-521-88068- 8, QA297 .N866 2007)

[2] Professor Kirkland. *Three or More Body Problem.* Homework Assignment Number 5 AEP 4380. https://courses.cit.cornell.edu/aep4380/secure/hw05f19.pdf

[3] Dr. David R. Williams, *Jupiter Fact Sheet.* NASA Goddard Space Flight Center Greenbelt, MD 20771, Last Updated: 18 July 2018, DRW, https://nssdc.gsfc.nasa.gov/planetary/factsheet/jupiterfact.html

[4] Ryan S. Park, Alan B. Chamberlin, *HORIZONS Web-Interface* Jet Propulsion Laboratory Solar System Dynamics, https://ssd.jpl.nasa.gov/horizons.cgi

Source Code

```
/*  AEP 4380 Assignment #5
    Three or More Body Problem

    Run on Windows core i7 with gcc version 8.2.0 (MinGW.org GCC-8.2.0-3)

    Gregory Kaiser October 21st 2019

*/
#include <cstdlib>
#include <cmath>
#include <iostream>
#include <fstream>
#include <iomanip>

using namespace std; //makes writing code easier

double w=1.0, much_less = .001,
max_error=1.0, epsilon=1e-6;//"...or whatever" -Numerical Recipes page 914
double a_tol = epsilon,
r_tol = epsilon; //absolute and relative tolerance for scaling error values
int i,j; //loop variables

//----------------physical constants-------------------
#define BIG_G 6.67408e-11 //(Nm^2)/kg^2
const double radii[4] = {6.96e8, 6.378e6, 3.3934e6, 7.1398e7}; //S,E,M,J
const double masses[4] = {1.9891e30, 5.9742e24, 0.64191e24, 1898.8e24}; //S,E,M,J
//---------------------------------------------------

//-----------------Dormand-Prince Coefficients--------
//The arrays are of larger size for convenient access to the correct index, with some small
//space sacrificecollaborated with Sam Feibel in the production of these values from the
//book, since it was annoying enough to plug in as is
const double dpc[8] = {0, 0.0, 1.0/5.0, 3.0/10.0, 4.0/5.0, 8.0/9.0 , 1.0, 1.0};
const double dpb[8] =
{0, 35.0/384.0, 0.0, 500.0/1113.0, 125.0/192.0, -2187.0/6784.0, 11.0/84.0, 0.0};
const double dpbs[8] =
{0, 5179.0/57600.0, 0.0, 7571.0/16695.0, 393.0/640.0, -92097.0/339200.0, 187.0/2100.0, 1.0/40.0};
const double dpa[8][7] = {
{0, 0,              0,                0,              0,             0,                  0          },
{0, 0,              0,                0,              0,             0,                  0          },
{0, 1.0/5.0,        0,                0,              0,             0,                  0          },
{0, 3.0/40.0,       9.0/40.0,         0,              0,             0,                  0          },
{0, 44.0/45.0,      -56.0/15.0,       32.0/9.0,       0,             0,                  0          },
{0, 19372.0/6561.0, -25360.0/2187.0,  64448.0/6561.0, -212.0/729.0, 0,                  0          },
{0, 9017.0/3168.0,  -355.0/33.0,      46732.0/5247.0, 49.0/176.0,    -5103.0/18656.0,    0          },
{0, 35.0/384.0,     0.0,              500.0/1113.0,   125.0/192.0,   -2187.0/6784.0,     11.0/84.0 },
};
//---------------------------------------------------

//harmonic oscillator derivatives function---------------------------------
//Calculates the derivative of each first order function for propagation
//is fed the current state and time and fills the derivatives array
void harm_osc_derivs(double t, double state[], double dstate_dt[]){
    dstate_dt[0] = state[1]; //dy_0/dt = y1
    //below: dy_1/dt = -w*w*y_0
    dstate_dt[1] = -w*w*state[0];//-.2*state[1];//includes a damping term used to verify solution
```

```
}//end harm osc derivs
//-------------------------------------------------------------------------

//--------------------------------------------------------------------
//variables that get recycled for calculations
int n_bodies=4;
double force_sum_x[4];
double force_sum_y[4];
double dist_abs, disp_x, disp_y, force_inter_x, force_inter_y;
int ox=0, oy=4, ovx=8, ovy=12;
//4 body problem derivatives function for sun earth mars jupiter
//Calculates the derivative of each first order function for propagation
//is fed the current state and time and fills the derivatives array
void four_body_derivs(double t, double state[], double dstate_dt[]){
    ox=0;oy=n_bodies;ovx=2*n_bodies;ovy=3*n_bodies;
    for(i=0;i<n_bodies;i++){
        for(j=i+1;j<n_bodies;j++){//loop only over interactions that haven't been calculated,ie i+1-->n
            //calculate the interaction force between i, j
            disp_x = state[j+ox]-state[i+ox];//fabs(state[j+ox]-state[i+ox]);
            disp_y = state[j+oy]-state[i+oy];//fabs(state[j+oy]-state[i+oy]);
            dist_abs = sqrt((disp_x*disp_x)+(disp_y*disp_y));//distance between i and j

            //The force interaction without mass scaling between i and j is
            force_inter_x = (disp_x/(dist_abs*dist_abs*dist_abs));
            force_inter_y = (disp_y/(dist_abs*dist_abs*dist_abs));

            //add the force to i (x,y) scaled by mass of obj j,
            //subtract from j (x,y) scaled by moss of obj i, since Newton's third
            force_sum_x[i] += masses[j]*force_inter_x;
            force_sum_y[i] += masses[j]*force_inter_y;
            force_sum_x[j] -= masses[i]*force_inter_x;
            force_sum_y[j] -= masses[i]*force_inter_y;
        }
        //once the forces are calculated for i, you can fill state with i's variables
        dstate_dt[i+ox] = state[i+ovx]; //dx/dt = vx
        dstate_dt[i+oy] = state[i+ovy]; //dy/dt = vy
        dstate_dt[i+ovx] = BIG_G*force_sum_x[i]; //dvx/dt = ax
        dstate_dt[i+ovy] = BIG_G*force_sum_y[i]; //dvy/dt = ay

        //done calculating the forces for i, so clear the sum variable for the next step
        force_sum_x[i]=0;
        force_sum_y[i]=0;
    }
}//end derivs four body
//-------------------------------------------------------------------

//General Purpose Runge-Kutta
//5th order with embedded 4th order solution using Dormand-Price parameters
//Is fed the state, time, h step, and a derivative function
//which contains all of the problem specific information.
void ghk_rk45_dp(double *state, double *new_state, double *error, double t, double h, int n,
    void (*derivs)(double, double[], double[])){
    double *k1,*k2,*k3,*k4,*k5,*k6,*k7,*temp,*new_state_star;
    //clever trick that Prof. Kirkland used in class to initialize all the arrays at once with "new".
    k1 = new double[n*9];
    k2 = k1 + n;
    k3 = k2 + n;
    k4 = k3 + n;
    k5 = k4 + n;
    k6 = k5 + n;
    k7 = k6 + n;
    temp = k7 + n;
    new_state_star = temp + n;
    //------------------------------
    derivs(t, state, k1); //information about the starting point

    for (i=0;i<n;i++){
        //fills temp with first step from starting point
        temp[i] = state[i]+dpa[2][1]*h*k1[i];
    }
    derivs(t+dpc[2]*h, temp, k2);//step two

    for (i=0;i<n;i++){
        //fills temp with next step
        temp[i] = state[i]+h*(dpa[3][1]*k1[i]+dpa[3][2]*k2[i]);
    }
    derivs(t+dpc[3]*h, temp, k3);//step three

    for (i=0;i<n;i++){
        //fills temp with next step
        temp[i] = state[i]+h*(dpa[4][1]*k1[i]+dpa[4][2]*k2[i]+dpa[4][3]*k3[i]);
    }
    derivs(t+dpc[4]*h, temp, k4);//step four

    for (i=0;i<n;i++){
        //fills temp with next step
        temp[i] = state[i]+h*(dpa[5][1]*k1[i]+dpa[5][2]*k2[i]+dpa[5][3]*k3[i]+dpa[5][4]*k4[i]);
    }
    derivs(t+dpc[5]*h, temp, k5);//step five

    for (i=0;i<n;i++){
        //fills temp with next step
        temp[i] = state[i]+h*(dpa[6][1]*k1[i]+dpa[6][2]*k2[i]+dpa[6][3]*k3[i]+dpa[6][4]*k4[i]+
        dpa[6][5]*k5[i]);
    }
    derivs(t+dpc[6]*h, temp, k6);//step six

    //assign new state <-- y_n+1, and assign temp <-- y*_n+1 for error calculation
    for (i=0;i<n;i++){
        //fills temp with next step
        new_state[i] = state[i]+h*(dpb[1]*k1[i]+dpb[2]*k2[i]+dpb[3]*k3[i]+dpb[4]*k4[i]+
        dpb[5]*k5[i]+dpb[6]*k6[i]);
    }
    derivs(t+dpc[7]*h, new_state, k7);//last step for last term in y*

    for (i=0;i<n;i++){
        //fourth order solution
        new_state_star[i] = state[i]+h*(dpbs[1]*k1[i]+dpbs[2]*k2[i]+dpbs[3]*k3[i]+dpbs[4]*k4[i]+
        dpbs[5]*k5[i]+dpbs[6]*k6[i]+dpbs[7]*k7[i]);
        //error calculation and normalization
        error[i] = fabs(new_state[i] - new_state_star[i])/(fabs(a_tol)+fabs(r_tol*state[i])+epsilon);
    }

    //------------------------------
    delete []k1; //clear space since this method is used repeatedly for small step sizes
    return;
}//end rk45_dp
```

```
int main(){

    ofstream fp; //output file for table
    ofstream fp2;
    fp.precision(6);
    fp2.precision(6);

    fp.open("hw5_1.dat");
    if(fp.fail()){
        cout<<"cannotopenfile"<<endl;
        return(EXIT_SUCCESS); //defined by standard library
    }
    fp2.open("hw5_2.dat");
    if(fp2.fail()){
        cout<<"cannotopenfile"<<endl;
        return(EXIT_SUCCESS); //defined by standard library
    }

    //**********************HARMONIC OSC***********************************
    //---------------Initializing the harmonic oscillator as a test case-------------
    n_bodies = 1;
    int n_dof = 2; //1 harmonic oscillator, with a position and a velocity
    int n_state = n_bodies*n_dof; //the number of equations in the full state array
    double harm_state[n_state] = {1.0,0.0};//initial conditions
    double harm_new_state[n_state];
    double harm_error[n_state];
    double max_scaled_error;
    //much_less=.001;//just for harm osc
    //max_error=1; //just for the harm osc
    //------------------------------------------------------------------------------
    /*
    To use Dormand Prince method, I use ghk_rk4_dp as a stepper, but only update the state if that
    trial step is accepted (error is "low enough"). I also therefore only print if the state is
    updated, making sure that I can re-do a step until the error is acceptable.
    */
    double t_init=0.0,t_final=80.0, t=t_init, h=5; //h will of course change, but needs an initial value
    //since h is no longer constant, I will have to update time incrementally, which means there
    //will be roundoff error at each step, hopefully accounted for by our error estimator

    //print the first value for simplicity later
    fp<<harm_state[0]<<setw(15)<<harm_state[1]<<setw(15)<<t<<endl;
    while(t<t_final){
        //take a step with rk45_dp
        ghk_rk45_dp(harm_state, harm_new_state, harm_error, t, h, n_state, harm_osc_derivs);
        //find the maximum inside the error array
        max_scaled_error = 0.0;
        for(i=0;i<n_state;i++){
            if(harm_error[i]>max_scaled_error){
                max_scaled_error = harm_error[i];
            }
        }
        //so now max_scaled_error holds the largest error value
        if(max_scaled_error<=max_error){//error is acceptable
            //print the new state
            fp<<harm_new_state[0]<<setw(15)<<harm_new_state[1]<<setw(15)<<t<<endl;
            //update the state with the accepted step
            for (i=0;i<n_state;i++){
                harm_state[i] = harm_new_state[i];
            }
            //update the time with the used h value
            t = t + h;
            if(max_scaled_error<much_less*max_error){
                //special case, if the error is super small, keep the step but make h bigger
                h = h*pow(max_error/max_scaled_error,0.2); //h<--hopt for new step
            }
        }
        else{//error is too big! don't change state or time, just try again with a smaller h value
                h = h*0.2;
        }
    }//end harm osc simulation while

    //***********************4BODY SIMULATOR*********************************
    //--------------HW problem specific initialization of state----------
    n_bodies = 4; n_dof = 4; //4 planets, with xy positions and velocities
    n_state = n_bodies*n_dof; //the number of equations in the full state array
    double fb_state[n_state];
    fb_state[0] = 0.0;          // Sx
    fb_state[1] = 149.598e9;    // Ex
    fb_state[2] = 228.0e9;      // Mx
    fb_state[3] = 778.298e9;    // Jx
    fb_state[4] = 0.0;          // Sy
    fb_state[5] = 0.0;          // Ey
    fb_state[6] = 0.0;          // My
    fb_state[7] = 0.0;          // Jy
    fb_state[8] = 0.0;          // Svx
    fb_state[9] = 0.0;          // Evx
    fb_state[10] = 0.0;         // Mvx
    fb_state[11] = 0.0;         // Jvx
    fb_state[12] = -3.0e1;      // Svy
    fb_state[13] = 2.9786e4;    // Evy
    fb_state[14] = 2.4127e4;    // Mvy
    fb_state[15] = 1.30588e4;   // Jvy
    double fb_new_state[n_state];
    double fb_error[n_state];
    max_error=1;
    much_less = .0001; max_error=1.0; epsilon=1e-4;
    //changed these values to play with 4body separate from harm osc values
    a_tol = epsilon; r_tol = epsilon; //absolute and relative tolerance for scaling error values
    //--------------------------------------------------------------------
    t_init=0.0; t_final=75e7; t=t_init; h=1;
    //setup same as harm osc, 75e7=number of seconds in 25 years
    //print the first value for simplicity later
    for(i=0;i<n_bodies;i++){
        //print x and y of each body in sequence
        fp2<<fb_state[i]<<setw(15)<<fb_state[i+oy]<<setw(15);
    }
    fp2<<t<<endl;

    while(t<t_final){
        //take a step with rk45_dp
        ghk_rk45_dp(fb_state, fb_new_state, fb_error, t, h, n_state, four_body_derivs);
        //find the maximum inside the error array
        max_scaled_error = 0.0;
        for(i=0;i<n_state;i++){
            if(fb_error[i]>max_scaled_error){
                max_scaled_error = fb_error[i];
            }
        }
        //so now max_scaled_error holds the largest error value
        if(max_scaled_error<=max_error){//error is acceptable
            //print the new state
            //format is Sx,Sy,Ex,Ey,Mx,My,Jx,Jy
            for(i=0;i<n_bodies;i++){
                //print x and y of each body in sequence
                fp2<<fb_new_state[i]<<setw(15)<<fb_new_state[i+oy]<<setw(15);
            }
            fp2<<t<<endl;
            //update the state with the accepted step
            for (i=0;i<n_state;i++){
                fb_state[i] = fb_new_state[i];
            }
            //update the time with the used h value
            t = t + h;
            if(max_scaled_error<much_less*max_error){
                //special case, if the error is super small, keep the step but make h bigger
                h = h*pow(max_error/max_scaled_error,0.2); //h<--hopt for new step
            }
        }
        else{//error is too big! don't change state or time, just try again with a smaller h value
                h = h*0.2;
        }
    }//end 4 body simulation while
    fp.close();
    fp2.close();
    return(EXIT_SUCCESS);
} //end main
```