AEP 4380: Homework 4

Gregory Kaiser

October 9th, 2019

1 Problem Background and Solution Overview

For this assignment, I was asked to implement a generalized 4th-order Runge-Kutta method for a single time step, and then apply it to a chaotic system. While this method works for analytic systems, it is most useful in propagating forward systems which require numerical solutions.

Runge-Kutta requires some overhead work before it can be properly used. If given a linear system of order n, one must break down that system into a set of n first order equations. Once that is done, one must generate a function referred to as derivs() in Numerical Recipes[1] which calculates those first order derivatives as a function of the current state of the system (and the independent variable). This function contains all the problem-specific physics, and is passed as a function pointer to the Runge-Kutta method.

The methods that I implemented are very similar to the solution in Numerical Recipes, and I haven't done anything very fancy to change that solution other than making my variables slightly more descriptive. This verifies that I understood what is happening at each line of code, and it matches the explanation given in class. Professor Kirkland also showed us a "sneaky" way to allocate space for 5 arrays in one step and then delete them all during each call of the Runge-Kutta method, which I used.

2 Solution Description

A saga of errors plagued my initial attempts at implementing both my derivative function and my Runge-Kutta stepper method. By giving up on trying to teach myself pointers using the internet, and by waiting until after the lecture explicitly on the implementation of this method, I arrived at the solution described below.

I first created a harmonic oscillator derivatives function, which fills the derivatives array with functions of the input state array. State contains y_0 and y_1 in that order. The output array contains the derivatives $\frac{dy_0}{dt}$ and $\frac{dy_1}{dt}$, also in that order.

The Runge-Kutta method first declares k_1, k_2, k_3, k_4, and *temp* as pointers before initializing k_1 as an array five-times the value of n_eqns, which should be the size of any array describing the system. k_2 through *temp* were then initialized as locations along that array. Each was of size n_eqns. This is the "trick" that Prof. Kirkland described which allows us to call "new" and "delete" only once, since they are relatively expensive time-wise.

The derivatives are placed into k_n, which are off by a factor of h from the K_n described in Numerical Recipes. For this reason, any time a k_n value is used, it must be multiplied by a factor of h to give a valid result.

At each step, before the next call to the derivative function, *temp* is filled with the current state plus some shift dependent on the derivative call which was previously made. This follows the Runge-Kutta algorithm exactly from Numerical Recipes. I am essentially having temp hold the second argument to the $f(x, y)$ function so that k_1 through k_4 follow the following set of equations:

$$k_1 = \frac{K_1}{h} = f(x_n, y_n) \tag{1}$$

$$k_2 = \frac{K_2}{h} = f(x_n + \frac{h}{2}, y_n + \frac{hk_1}{2}) \tag{2}$$

$$k_3 = \frac{K_3}{h} = f(x_n + \frac{h}{2}, y_n + \frac{hk_2}{2}) \tag{3}$$

$$k_4 = \frac{K_4}{h} = f(x_n + h, y_n + hk_3) \tag{4}$$

where $f(x, y)$ is just $derivs(x, y, dy/dx)$. Then, the next state of the system can be calculated:

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2(k_2 + k_3) + k_4) \tag{5}$$

and then the state is updated from a driver function so that the method can be used again.[1]

By testing the harmonic oscillator first, I was able to fix errors and verify that my implementation worked (see Results and Interpretation). I was also able to add some damping to the oscillator to verify that it behaved as I expected.

Moving on to the derivatives function for the Rössler system was as simple as declaring and initializing the constants a, b, and c, and following the format of the harmonic oscillator derivative function for calculating each spot in the array.

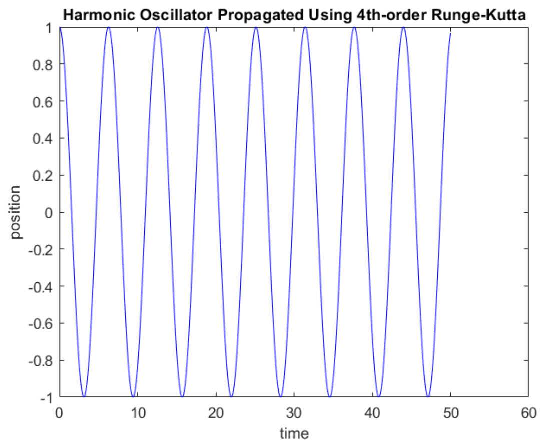
The driver setup in main() needed only to set the number of equations correctly, initialize an array of the state with some initial conditions, and declare the derivative array (it would be filled on the first call to derivs() inside ghk_rk4()). I set an initial and final time as an interval for a loop to use the Runge-Kutta method repeatedly, but left the h value as a variable for easy modification. This means that I might not land exactly on the final time specified, but I figured that h will be small enough that it won't matter a huge amount.

I did not use the vector classes from Numerical Recipes, but instead used simple arrays as my vectors. While I could have stepped forward all three of the systems which I used in this assignment into a single driver loop, I decided to make them three separate chunks of code. This allowed for clarity, easier debugging, and easier tuning of values at the cost of efficiency.

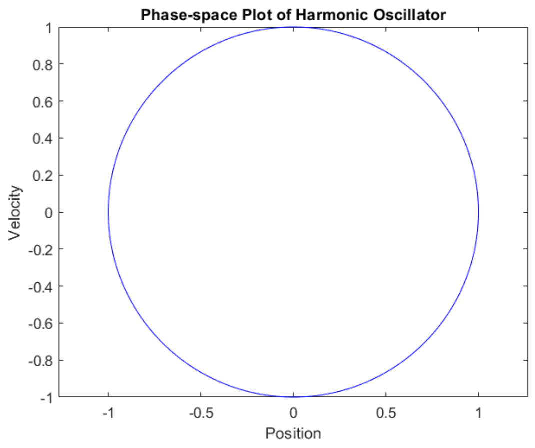
3 Results and Interpretation

The first thing to do was to test my setup using the harmonic oscillator. The plots look qualitatively correct as a good cosine wave with constant amplitude for a plot of $y_0(t)$. A circle appears in the plot of y_0 versus y_1, also known as the phase space plot of position versus velocity for this one dimensional oscillator. To verify the period, I printed out the value of t whenever the Runge-Kutta step returned a state that went from positive to negative.

An incomplete set of those numbers was: 1.57, 7.85, 14.13, 20.42... between each of those values is $T = 6.28$ which corresponds to a period of 2π. Since the analytic solution of this system is $cos(\omega t)$, this period and amplitude matches the harmonic oscillator of $\omega = 1.0$ exactly.

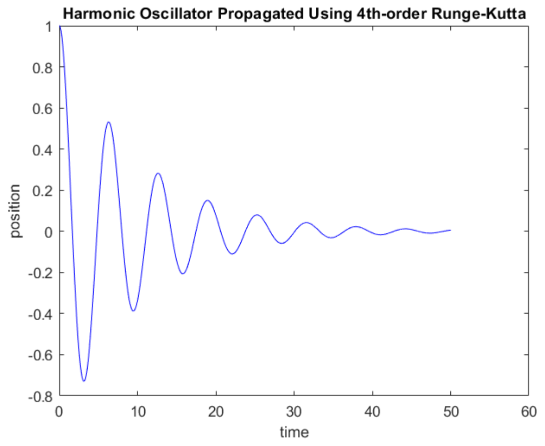


(a) Harmonic Oscillator $y_0(t)$

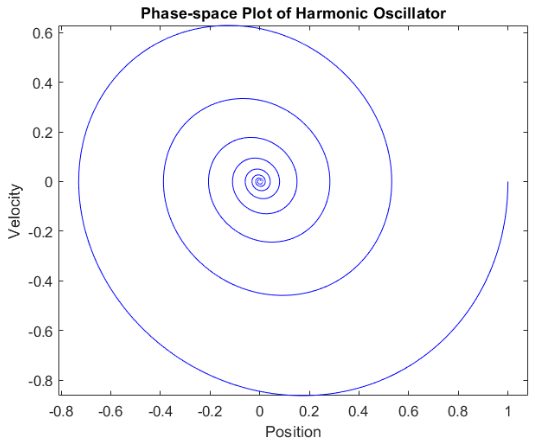


(b) Phase Space plot of the Harmonic Oscillator

By adding a damping term in the derivatives function, which subtracts $0.2y_1$ from the derivative of y_1, the system behaves as expected as well.

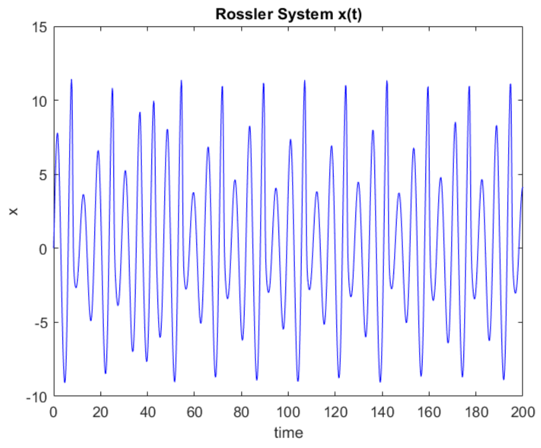


(a) Damped Harmonic Oscillator

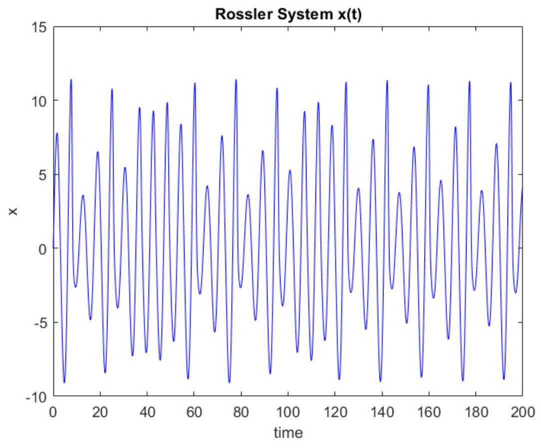


(b) Phase Space plot of the Damped Harmonic Oscillator

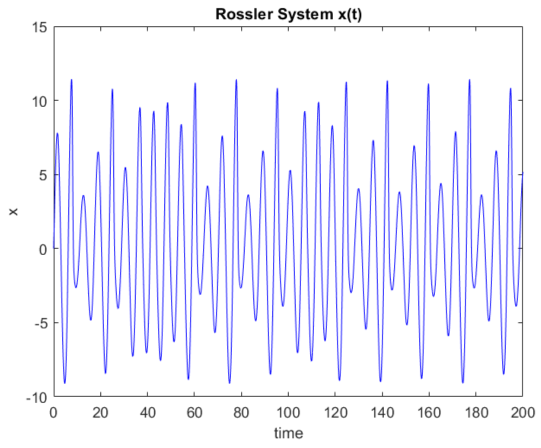
This gives me the confidence to say that my solutions to the Rössler system are also correct. To find a good h value, I first looked at $x(t)$ for an h value of about 0.1 and repeatedly changed h until I thought it looked stable:



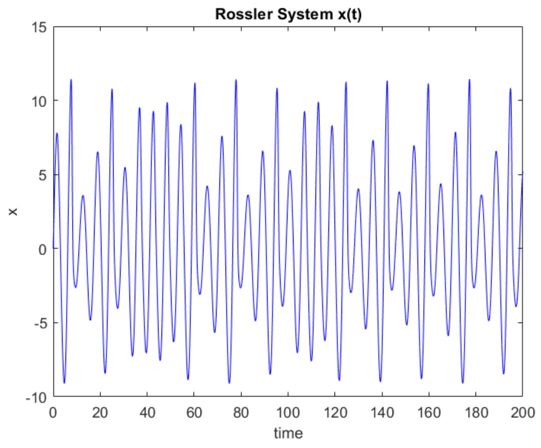
(a) h=0.2



(b) h=0.015



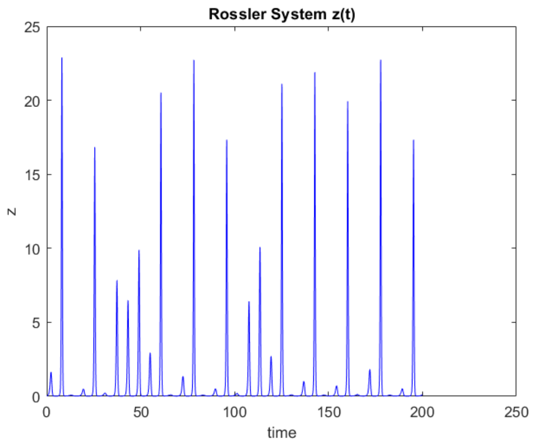
(c) h=0.008



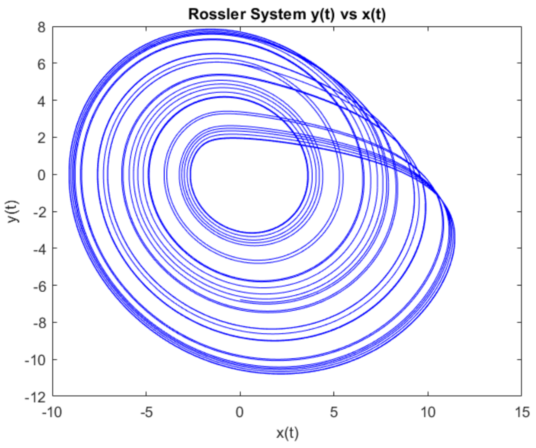
(d) h=0.002

There is a clear difference between h=0.2 and h=0.015 around the t=150 mark, but the differences become much more subtle as h decreases from there. By placing these images in the same folder and scanning through them quickly on my computer, I was able to see those differences more clearly. Since there was an imperceptible difference between h=0.008 and h=0.002, and just barely a difference between h=0.008 and h=0.015, I decided to go with h=0.008 for the rest of the assignment.

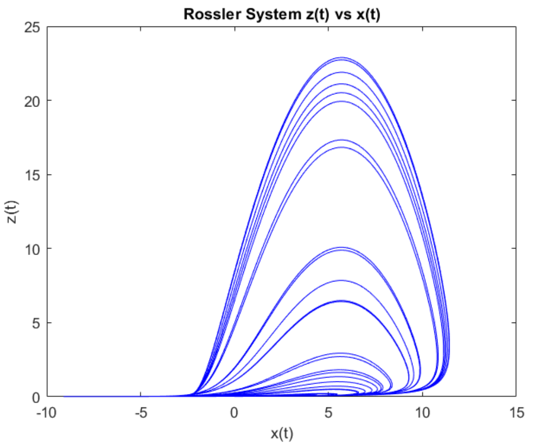
Plots of $x(t)$, $y(t)$, and $z(t)$ looked strange, but didn't jump out to me as being chaotic until I looked for periodicity and couldn't find it. The real indicator of chaos was that the phase space plots of $z(t)$ versus $x(t)$ and $x(t)$ versus $y(t)$ were not single loops. They did orbit around in a circuit, however, which is consistent with the "attractor" characteristics of some chaotic phase space plots.



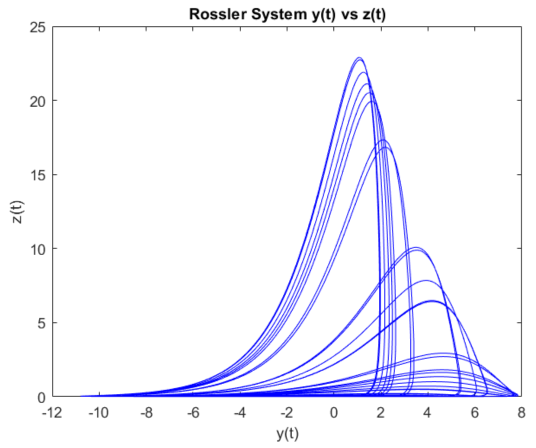
(a) z(t)



(b) y(t) versus x(t)



(c) z(t) versus x(t)



(d) y(t) versus z(t)

The three dimensional plot of $x(t)$ vs $y(t)$ vs $z(t)$ at this point was quite interesting:

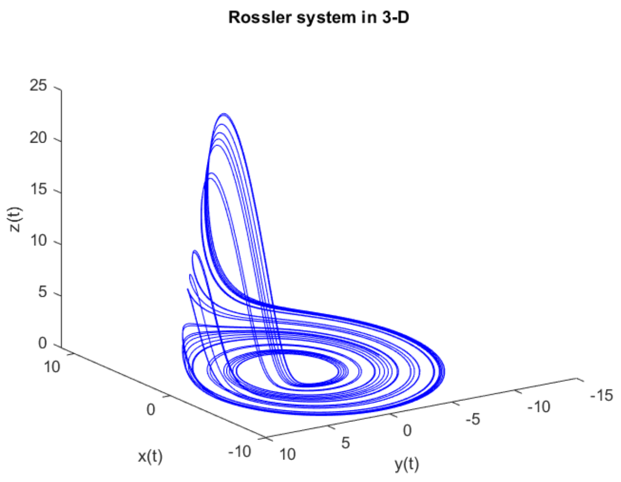
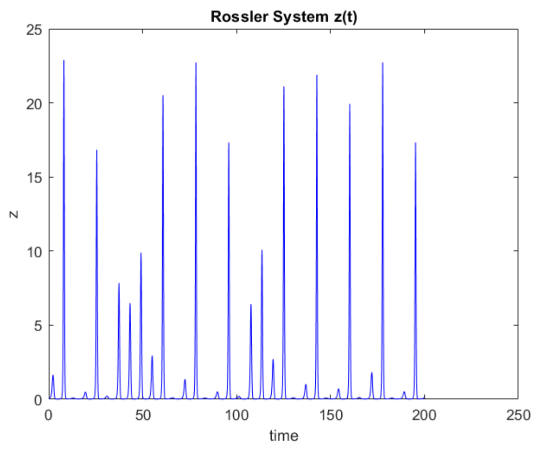


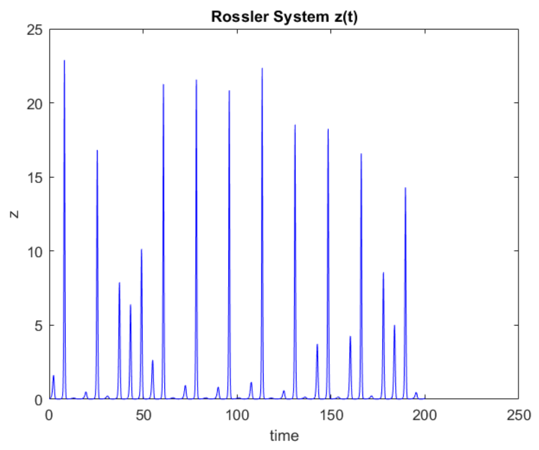
Figure 1: Three-dimensional plot of the Rössler System using plot3() in MATLAB

The physical interpretation of these results is difficult, since the system is chaotic, but I can say that my results match documentation on the Rössler system in general.[4] I could say that there is a kind of attractor in both the $z(t)$ versus $x(t)$ and the $y(t)$ versus $x(t)$ plots, since they both seem to circle around in a three-dimensional circuit.

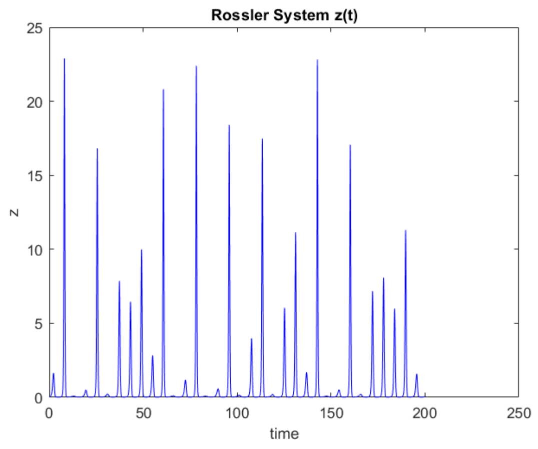
Optional:
Using the same constants a, b, and c, I changed the starting point of the system slightly:



(a) x(0)=0.0; y(0)=-6.78; z(0)=0.02



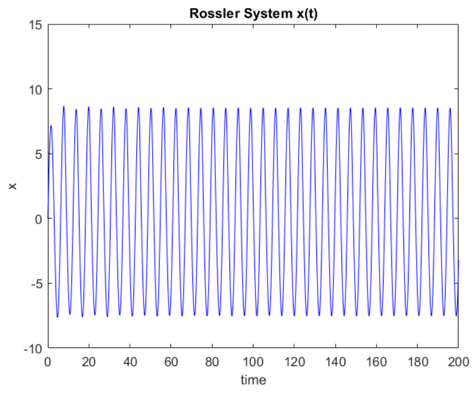
(b) x(0)=0.0; y(0)=-6.77; z(0)=0.02



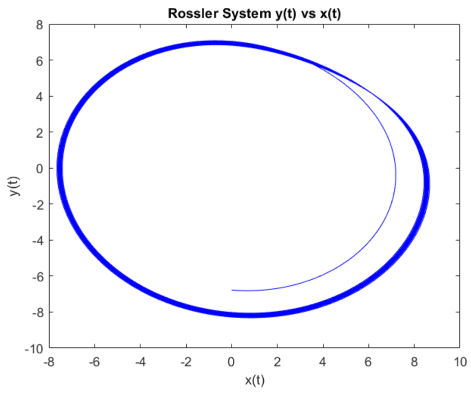
(c) x(0)=0.01; y(0)=-6.78; z(0)=0.02

It was most obvious to see the sensitivity of the system when looking at plots of $z(t)$. Sensitivity to these small changes to initial conditions is also a hallmark of chaotic systems.

I was also able to easily change the values of a, b, and c easily, to see what would happen:

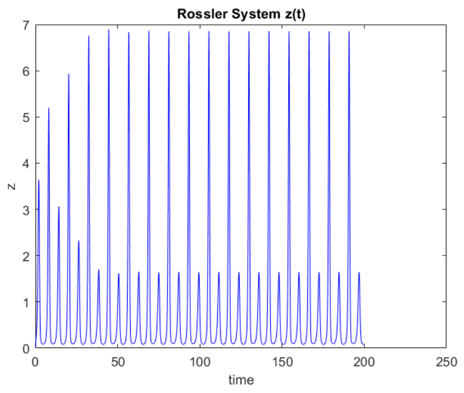


(a) x(t) for a=0.1; b=0.2; c=5.7

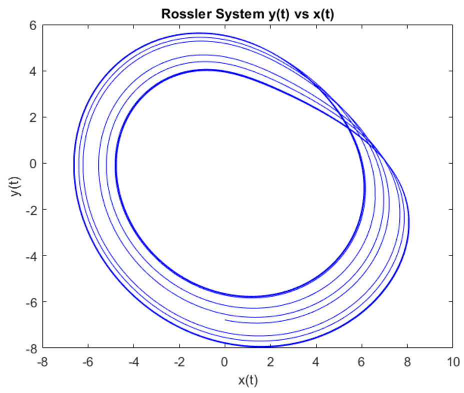


(b) x(t) versus y(t) for a=0.1; b=0.2; c=5.7

Changing a to 0.1 produced a system which became periodic and therefore not chaotic very rapidly.

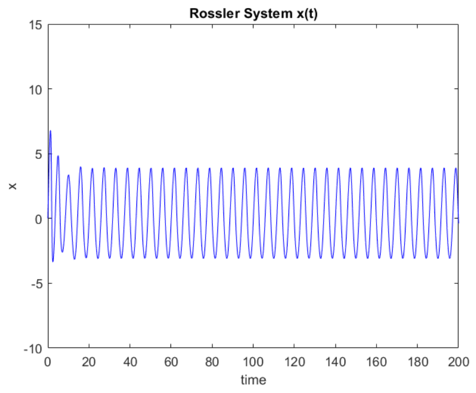


(a) x(t) for a=0.2; b=1; c=5.7

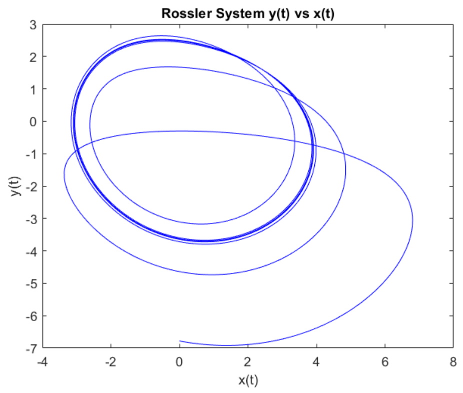


(b) x(t) versus y(t) for a=0.2; b=1; c=5.7

By changing b to 1.0 I observed a kind of period doubling which I also recognize from chaotic systems in AEP 3330.



(a) x(t) for a=0.2; b=0.2; c=2.0



(b) x(t) versus y(t) for a=0.2; b=0.2; c=2.0

It is clear that using c=2.0 makes this system behave periodically and non-chaotically after a short period of time.

Just for fun, I decided to make the Lorenz system because I've seen the "butterfly" before but haven't had the opportunity to plot it myself. I had to take cues from Wikipedia for values of the constants that are used (I just looked at the values used to get the butterfly image I want), but I took the form of the derivatives function from Edward Lorenz's 1963 paper on Deterministic Nonperiodic Flow.[2][3]

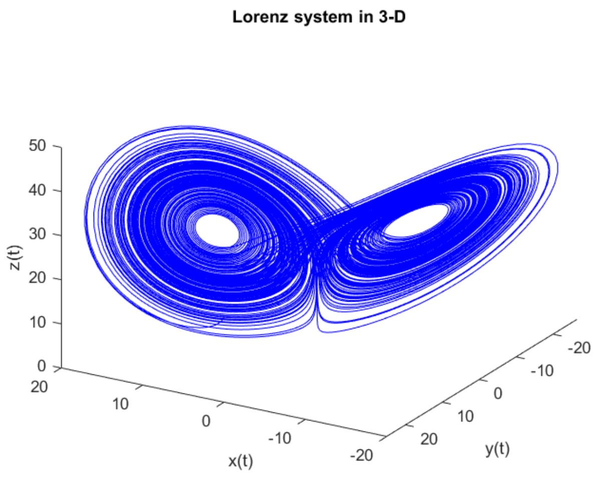


Figure 2: Three-dimensional plot of the Lorenz System using plot3() in MATLAB

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] E. N. Lorenz, "Deterministic Nonperiodic Flow", J. Atmos. Sci. 20, 1963, p.130.

[3] Wikipedia, "Lorenz System", https://en.wikipedia.org/wiki/Lorenz_system

[4] C. Letellier and O. E. Rossler "Rossler attractor", http://www.scholarpedia.org/article/Rossler_attractor

Source Code

```
/*  AEP 4380 Assignment #4
    Ordinary Differential Equations and Chaotic Systems:
    Implementation of a 4th order Runge-Kutta method.
    Simulated the Rossler System, similar to the Lorenz system.

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

    Gregory Kaiser October 9th 2019

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

using namespace std; //makes writing code easier
int i,j; //just iteration variables
const double a=.2,b=.2,c=5.7,w=1.0; //the constants that I will use in generating the derivative functions
//const double a=.1,b=.2,c=5.7,w=1.0; //a different set of constants for changing a
//const double a=.2,b=1,c=5.7,w=1.0; //a different set of constants for changing b
//const double a=.2,b=.2,c=2,w=1.0; //a different set of constants for changing c

double tol = 1e-6; //a tolerance for checking the period of the harmonic oscillator

//constants for the lorenz system were pulled from wikipedia, just for convenience
double sigma = 10.0, r = 28.0, b2=8.0/3.0;

//Calculates the derivative of each first order function for propagation
//being fed the current state, and fills the derivatives array
void hw_derivs(double t, double state[], double dstate_dt[]) {
    dstate_dt[0] = -state[1]-state[2];//the derivative of x during moment state dxdt=-y-z
    dstate_dt[1] = state[0]+a*state[1];//the derivative of y during moment state dydt = x+ay
    dstate_dt[2] = (b+state[2]*(state[0]-c));//the derivative of z during moment state dzdt = b+z(x-c)
}

//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
}

//Calculates the derivative of each first order function for propagation
//is fed the current state and time and fills the derivatives array
//this system setup is described in Edward Lorenz's 1963 paper
void lorenz_derivs(double t, double state[], double dstate_dt[]){
    dstate_dt[0] = sigma*(-state[0]+state[1]);//the derivative of x during moment state dxdt=-sigma*x-_sigma*
    dstate_dt[1] = -state[0]*state[2]+r*state[0]-state[1];//the derivative of y during moment state dydt = -s
    dstate_dt[2] = state[0]*state[1]-b2*state[2];//the derivative of z during moment state dzdt = x*y-b*z
}

//General Purpose 4th order Runge-Kutta method with t as the only independent variable
//and state containing n_eqns (a number of) dependent variables. Is fed the state, time, h step, and a deriva
//which contains all of the problem specific information
```

```
void ghk_rk4(double *state, double *new_state, double t, double h, int n_eqns, void (*derivs)(double, double[
    double *k1,*k2,*k3,*k4,*temp;
    k1 = new double[n_eqns*5]; //clever trick that Prof. Kirkland used in class to initialize all the arrays
    k2 = k1 + n_eqns;
    k3 = k2 + n_eqns;
    k4 = k3 + n_eqns;
    temp = k4 + n_eqns;
    derivs(t, state, k1); //information about the starting point

    for (i=0;i<n_eqns;i++){
        //fills temp with first step from starting point
        temp[i] = state[i]+0.5*h*k1[i];
    }
    derivs(t+h*0.5, temp, k2); //use temp to generate a set of predicted derivatives

    for (i=0;i<n_eqns;i++){
        //fills temp with second step from starting point
        temp[i] = state[i]+0.5*h*k2[i];
    }
    derivs(t+h*0.5, temp, k3); //use temp to generate another set of predicted derivatives

    for (i=0;i<n_eqns;i++){
        //fills temp with third step from starting point
        temp[i] = state[i]+h*k3[i];
    }
    derivs(t+h, temp, k4); //final step of predicted derivatives

    for (i=0;i<n_eqns;i++){
        //accumulate all of the derivatives with proper weights to have a prediction
        //with error O(h^5)
        new_state[i] = state[i]+(h/6.0)*(k1[i]+2.0*(k2[i]+k3[i])+k4[i]);
    }
    delete []k1; //clear space since this method is used repeatedly for small step sizes
    return;
}

int main(){

    ofstream fp; //output file for testing
    fp.precision(5);

    fp.open("hw4_1.dat");
    if(fp.fail()){
        cout<<"cannotopenfile"<<endl;
        return(EXIT_SUCCESS);
    }

    ofstream fp2; //output file for the hw function
    fp2.precision(5);

    fp2.open("hw4_2.dat");
    if(fp2.fail()){
        cout<<"cannotopenfile"<<endl;
        return(EXIT_SUCCESS);
    }

    ofstream fp3; //output file for the hw function
    fp3.precision(5);
```

```
fp3.open("hw4_3.dat");
if(fp3.fail()){
    cout<<"cannotopenfile"<<endl;
    return(EXIT_SUCCESS);
}
//HW Assignment
//-------------------------HARMONIC OSC----------------------------------
//test with the harmonic oscillator
int neqn = 2, istep, nstep=30000;
//The state vector contains the initial conditions
double harm_state[neqn]= {1.0,0.0}; //in this case, y_0=1.0, and y_1=0.0
double harm_new_state[neqn]; //initially empty new state
double t_init=0.0,t_final=50.0, t, h=.01;
for (istep=0;t<=t_final;istep++){
    t=t_init+istep*h;
    //output current value stored in state before updating
    fp<<harm_state[0]<<setw(15)<<harm_state[1]<<setw(15)<<t<<endl;
    //fill new state and update old
    ghk_rk4(harm_state, harm_new_state, t, h, neqn, harm_osc_derivs);
    if (harm_new_state[0]<=0&&harm_state[0]>0){
        //prints out the zeroes (or close enough) so that I can
        //verify the period at a glance to the terminal
        cout<<t<<endl;
    }
    for (i=0;i<neqn;i++){
        harm_state[i] = harm_new_state[i];
    }
}
//--------------------------THE ROSSLER SYSTEM--------------------------------------------
//now test the actual homework function
neqn = 3;
double hw_state[neqn]= {0.00,-6.78,.02}; //initial conditions {x,y,z}
//double hw_state[neqn]= {0.0,-6.77,.02}; //initial conditions {x,y,z}
//double hw_state[neqn]= {0.01,-6.78,.02}; //initial conditions {x,y,z}
double hw_new_state[neqn];
t_init=0.0,t_final=200.0,t=t_init,h=.008;
for (istep=0;t<=t_final;istep++){
    t=t_init+istep*h;
    //output current value stored in state before updating
    fp2<<hw_state[0]<<setw(15)<<hw_state[1]<<setw(15)<<hw_state[2]<<setw(15)<<t<<endl;
    //fill new state and update old
    ghk_rk4(hw_state, hw_new_state, t, h, neqn, hw_derivs);
    for (i=0;i<neqn;i++){
        hw_state[i] = hw_new_state[i];
    }
}
//--------------------------THE LORENZ SYSTEM---------------------------------------------
//just for fun
neqn = 3;
double lorenz_state[neqn]= {10,3,4}; //initial conditions {x,y,z}
double lorenz_new_state[neqn];
t_init=0.0,t_final=200.0,t=t_init,h=.008;
for (istep=0;t<=t_final;istep++){
    t=t_init+istep*h;
    //output current value stored in state before updating
    fp3<<lorenz_state[0]<<setw(15)<<lorenz_state[1]<<setw(15)<<lorenz_state[2]<<setw(15)<<t<<endl;
    //fill new state and update old
    ghk_rk4(lorenz_state, lorenz_new_state, t, h, neqn, lorenz_derivs);
    for (i=0;i<neqn;i++){
```

```
                lorenz_state[i] = lorenz_new_state[i];
            }
        }

        fp.close();
        fp2.close();
        fp3.close();
        return(EXIT_SUCCESS);
    } //end main
```