A project on Runge-Kutta methods and Chaotic Systems

Arjun Shivanand Kannan
ak888@cornell.edu

This LaTeX document has been formatted using the hyperref package which creates hyperlinks within the document. The viewer may click on any of the references, sections, links or figure numbers and it will link to them for easy viewing and linking within the document.

Contents

List of Figures

1 Introduction - Overview of the project

Chaotic systems are physical systems where it may happen that small differences in the initial conditions produce very great ones in the final phenomena [1]."Chaotic" is a term assigned to that class of motions in deterministic physical and mathematical systems whose time history has a sensitive dependence on initial conditions. They are bounded but not predictable. This project will extend the numerical concepts in the assignments on Numerical solutions of Ordinary Differential Equations (ODEs) to three such systems using the adaptive step size embedded Runge-Kutta 5th order method(Chapter 16 of [2]). The goal is to act as a beginning to the understanding of chaotic systems and their behaviour in the physical world.

2 Runge-Kutta methods

Runge-Kutta methods are a class of methods which judiciously use the information on the 'slope' at more than one point to extrapolate the solution to the future time step[2]. They are a set of numerical methods which are highly useful in solving systems of ordinary differential equations (ODEs) . These methods can solve a system of n ordinary differential equations by treating them as n coupled first order differential equations.

2.1 Adaptive step-size Runge-Kutta

The method we will be using in this project will be the adaptive step-size fifth-order Runge-Kutta method (Section 17.2 of [2]) which has an embedded fourth-order system to adjust the step size according to the error. The system uses equations with specialized Dormand-Prince coefficients [2] to achieve the required accuracy. It is specified as follows :

$$k_1 = hf(x_n, y_n) \tag{2.1.1}$$

$$k_2 = hf(x_n + c_2 h, y_n + a_{21} k_1) \tag{2.1.2}$$

$$\ldots \text{upto}$$

$$k_6 = hf(x_n + c_6 h, y_n + a_{61} k_1 + ... + a_{65} k_5) \tag{2.1.3}$$

$$y_{n+1} = y_n + \sum_{i=1}^{6} b_i k_i + O(h^6) \tag{2.1.4}$$

The embedded fourth-order formula is given by :

$$y^*_{n+1} = y_n + \sum_{i=1}^{6} b^*_i k_i + O(h^5) \tag{2.1.5}$$

The error estimate is :

$$\Delta \equiv y_{n+1} - y^*_{n+1} = \sum_{i=1}^{6} (b_i - b^*_i) k_i \tag{2.1.6}$$

We adjust the step size according to the error in each step. Whenever the error is lower than required, we increase the step size to gain speed over less critical areas of the curve, and whenever the error is higher than required, we decrease the step size to gain better accuracy over critical areas of the curve. To apply this method to all kinds of systems, we need a scaling method so that we are always computing the relative error irrespective of the magnitude of the parameter. For this we use a small absolute error limit for the scale ϵ_0 and define the scale as:

$$\text{Scale[i]} = |y_n(i)| + h|\frac{dy_n(i)}{dt}| + \epsilon_0 \tag{2.1.7}$$

2.2 The algorithm of automatic step size Runge-Kutta

The algorithm is as follows

1. Calculate k_1 to k_6 using the Dormand-Prince coefficients.
2. Calculate y_{n+1} and y^*_{n+1} using the values of k_1 to k_6.
3. Estimate the current relative error ϵ_{max}.
4. Find the maximum error Δ_{max}.
5. If $\epsilon_{max} > \Delta_{max}$ increase the step size and move on to the next step.
6. If $\epsilon_{max} < \Delta_{max}$ decrease the step size and repeat the current step.
7. Save y_{n+1} and set $t_{n+1} = t + h$.

The sequence y_i over i steps gives us the answer.

2.3 Initial testing

The adaptive step-size subroutine is written by us as the function `ode5()` in C++, similar to `odeint()` and `stepperdopr5` from [2]. This is tested on a system with a known solution first, so that once we obtain the right answer, we can proceed to solve for unknown systems with some confidence. In this particular case, we take the system from the homework problem (The Mathieu system) and test it in our program to make sure our subroutine is working right. The result (Figure 3.1) matches the correct result from the homework problem which gives us some confidence about the method.

3 Chaos

Chaos describes a system that is predictable in principle but unpredictable in practice. In other words, although the system follows deterministic rules, its time evolution appears random. In dynamical systems theory, the term chaos is applied to deterministic systems that are aperiodic and that exhibit sensitive dependence on initial conditions. Sensitivity means that a small change in the initial state will lead to progressively larger changes in later system states. This is illustrated in Figure 4.7.

Because initial states are seldom known exactly in real-world systems, predictability is severely limited. A bounded trajectory of a dynamical system is said to be chaotic if it has sensitive dependence on initial conditions and is not quasi-periodic. The concept of chaos has been used to explain how systems that should be subject to known laws of physics, such as weather, may be predictable in the short term but are apparently random on a longer time scale.

Before we move into our study, let us define some of the terms we will be using for our analysis.

3.1 Phase space

A phase space is a space in which all possible states of a system are represented, with each possible state of the system corresponding to one unique point in the phase space. For mechanical systems, the phase space usually consists of all possible values of position and momentum variables i.e. the cotangent space of configuration space. In plain terms it is usually a plot of the solved parameter x_i versus its derivative dx_i. Chaotic systems form distinct shapes in phase space which means that phase plots of a system are a useful tool for identifying chaos. Indeed, some of the most famous characteristic chaotic systems are identified more by their phase plots than their response plots.

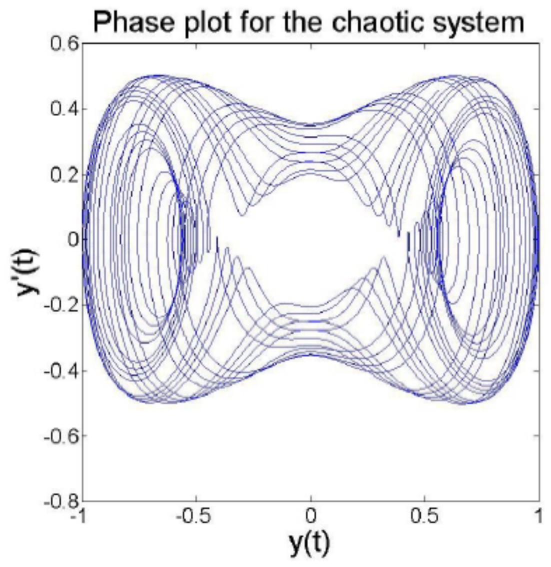

Figure 3.1: Phase space plot of the chaotic Mathieu system

3.2 Attractors

In all subsequent analysis we will be talking about the attractor of the chaotic system. An attractor is a set towards which a variable, moving according to the dictates of a dynamical system, evolves over time [1]. That is, points that get close enough to the attractor remain close even if slightly disturbed. The attractor is a region in n-dimensional space. In lay terms we may simply see it as a tendency of the chaotic system (under certain conditions) to 'pull' itself into 'repeating' a certain geometry.

3.3 Poincare maps

A Poincare map is the intersection of a periodic orbit in the state space of a continuous dynamical system with a certain lower dimensional subspace, called the Poincare section, transversal to the flow of the system[1]. It is a method to study the phenomena occurring in a chaotic system in a simpler fashion by taking an N-dimensional chaotic system and mapping it in a coordinate of $N-1$ dimensions by taking slices of the phase space or time space at certain values of one of the parameters. One such example would be to study a chaotic system with a periodic forcing function by taking slices through the phase space at every integral multiple of the period [3]. This gives us an idea of where we expect to 'catch' the system at every cycle. For a periodic system, the section will be a cluster of points at the same location whereas for a chaotic system, the section provides us with some very interesting pictures that give us a deeper look into how the system works than a phase space plot alone such as in Figure 4.5. In other cases we can choose to make a Poincare plot by choosing the inflexion points of one of the coordinates and mapping the others at those points such as in Figures 4.10 and 4.14. These pictures are an indicator of the physical phenomena occurring in the system and are a good first step towards understanding the 'flow' of the system at key points.

3.4 Fractals & Chaos

Fractals are geometric shapes that are very complex and infinitely detailed. You can zoom in on a section and it will have just as much detail as the whole fractal. They are recursively defined and small sections of them are similar to large ones. Fractals are recursively defined and infinitely detailed. They are related to chaos because they are complex systems that have definite properties. One of the most famous fractal patterns is the Lorenz strange attractor which we see in Figure 4.8 from which we can see recursive definition. This helps us get an idea of what fractals are.

4 Chaotic systems under study

4.1 The Duffing Oscillator

The Duffing oscillator (originally introduced in 1918) was introduced in relation to spatial modes of vibration of steel beams subject to external periodic forces. It is now a standard prototype of forced systems and is used to study systems like double-well potential problems such as the Holmes system(Equation 4-2.2 of [1]). The Duffing oscillator is an example of a periodically forced oscillator with a nonlinear elasticity(Equation 3-8.7 of [1]), written as :

$$\frac{d^2x}{dt^2} + \mu\frac{dx}{dt} + \alpha x + \beta x^3 \quad = \quad F\cos\omega t \tag{4.1.1}$$

The parameter μ is the damping constant which obeys $\mu \geq 0$. For $\beta > 0$ the Duffing oscillator can be interpreted as a forced oscillator with a spring. When $\alpha > 0$, this spring is called a hardening spring, and when $\alpha < 0$, it is called a softening spring although this interpretation is valid only for small x. For $\beta < 0$ the Duffing oscillator describes the dynamics of a point mass in a double well potential, and it can be regarded as a model of a periodically forced steel beam which is deflected towards two magnets (Figure 4.1).

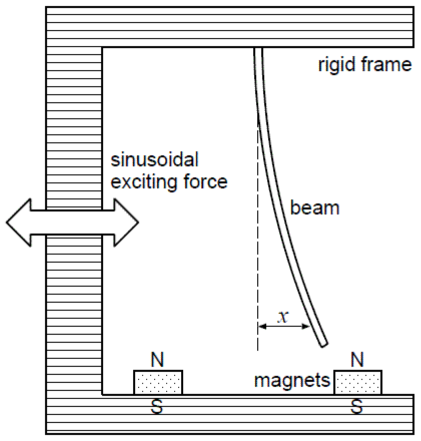

Figure 4.1: The physical interpretation of the Duffing oscillator with $\beta < 0$ (Picture courtesy Scholarpedia - `http://www.scholarpedia.org/article/Duffing_oscillator`)

4.1.1 Results for the Duffing oscillator

We can see the sensitivity of the system to the parameter β and its sign in Figures 4.3 and 4.4 where we see the number of wells doubling as β goes below zero. The Poincare section in Figure 4.5 (taken at every period of the forcing function) and the zoomed-in Poincare section (to show the attractor) in Figure 4.6 show the characteristic shape of the Duffing oscillator for a double-well potential. The Poincare sections for this system are snapshots of the phase space taken at every integral multiple of the period of the forcing function. From these plots we can understand the chaotic nature of the system as we can see that trajectories do not get repeated even over a large period of time.

[1]The parameter values α, β, μ, A and ω were chosen from the specified values for chaos from [1].

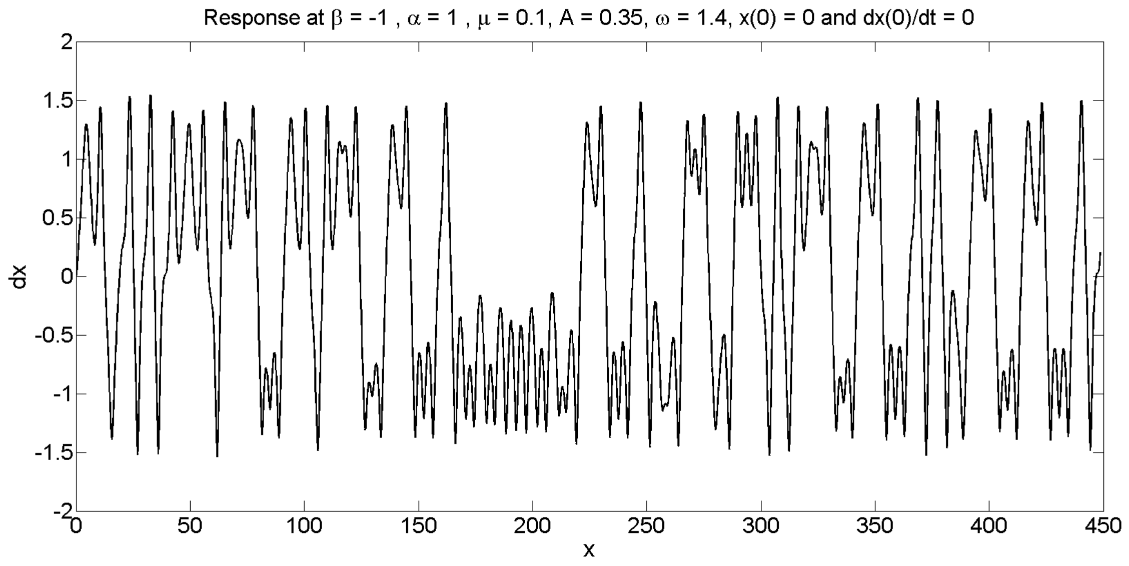

Figure 4.2: The response plot of the Duffing oscillator for the stated initial conditions

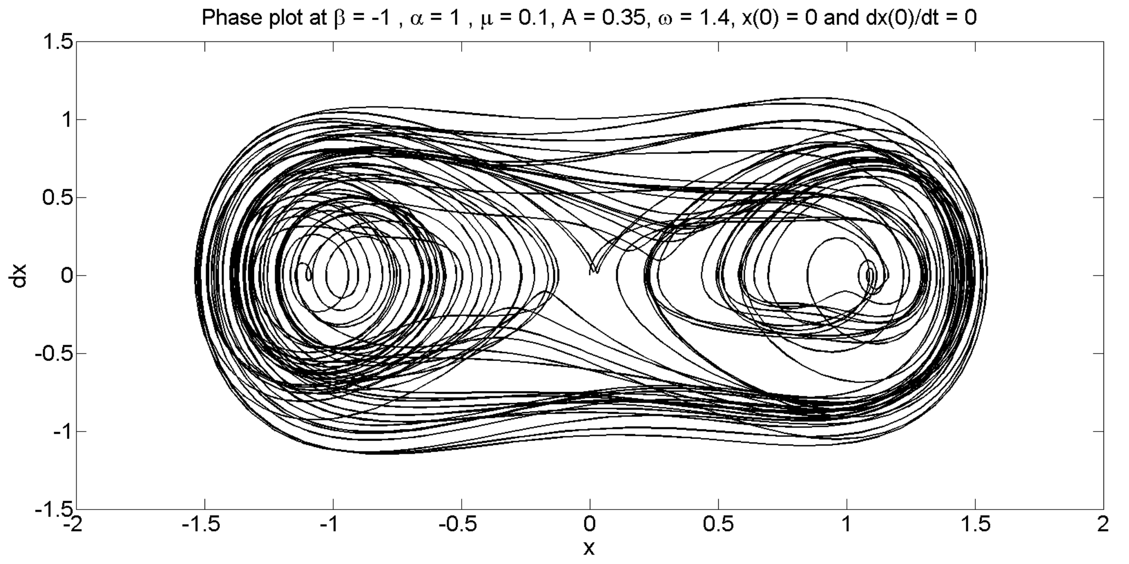

Figure 4.3: The phase plot of the Duffing oscillator for a double-well potential where $\beta < 0$

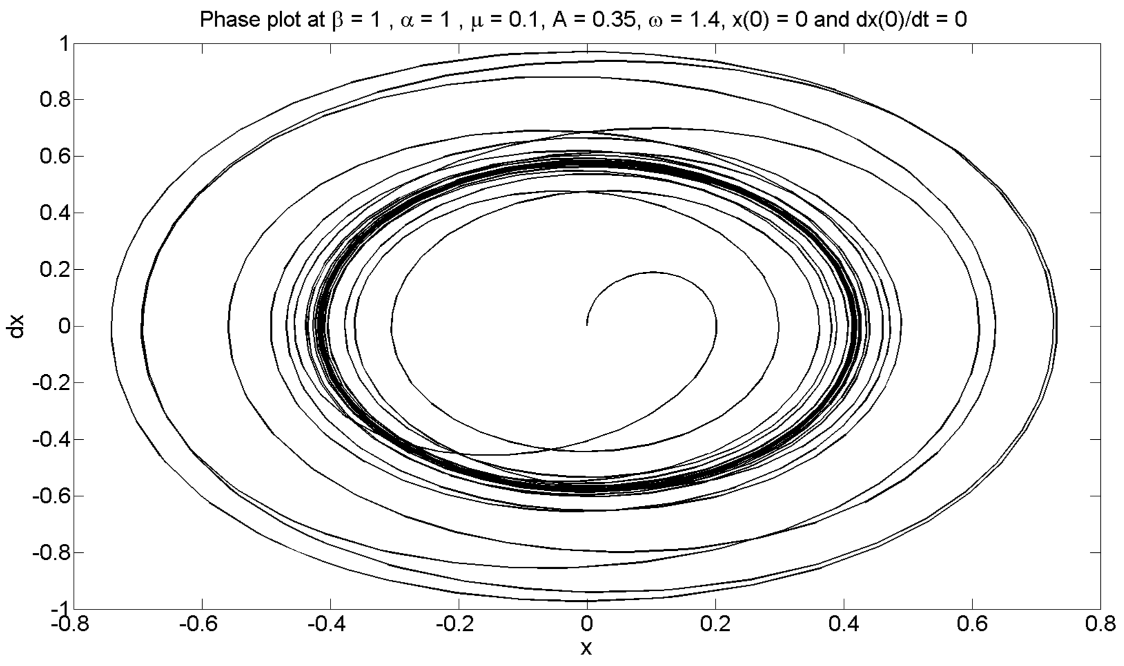

Figure 4.4: The phase plot of the Duffing oscillator for a single-well potential where $\beta > 0$

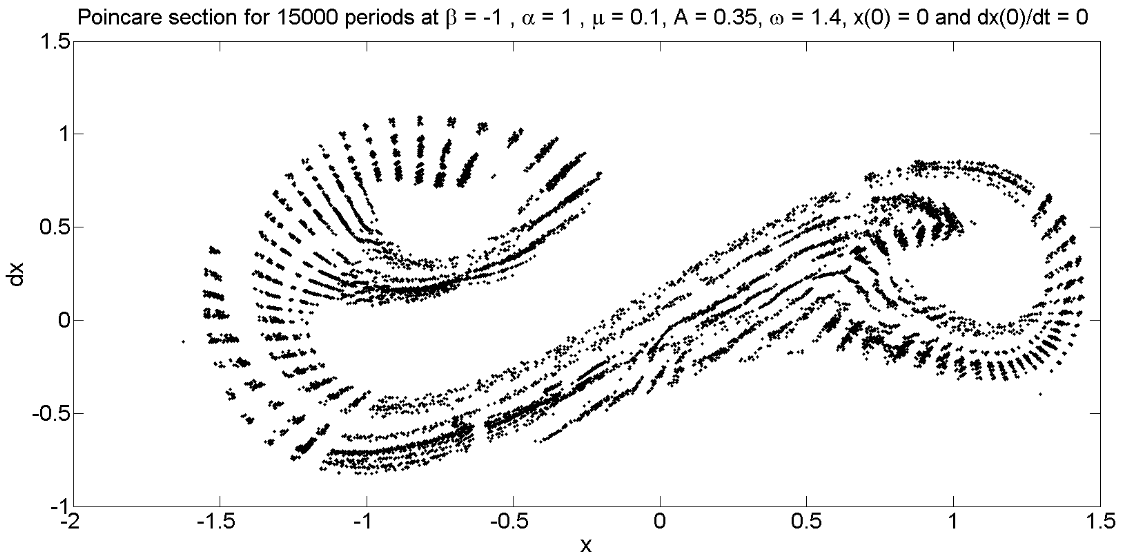

Figure 4.5: The Poincare plot of the Duffing oscillator for the stated initial conditions

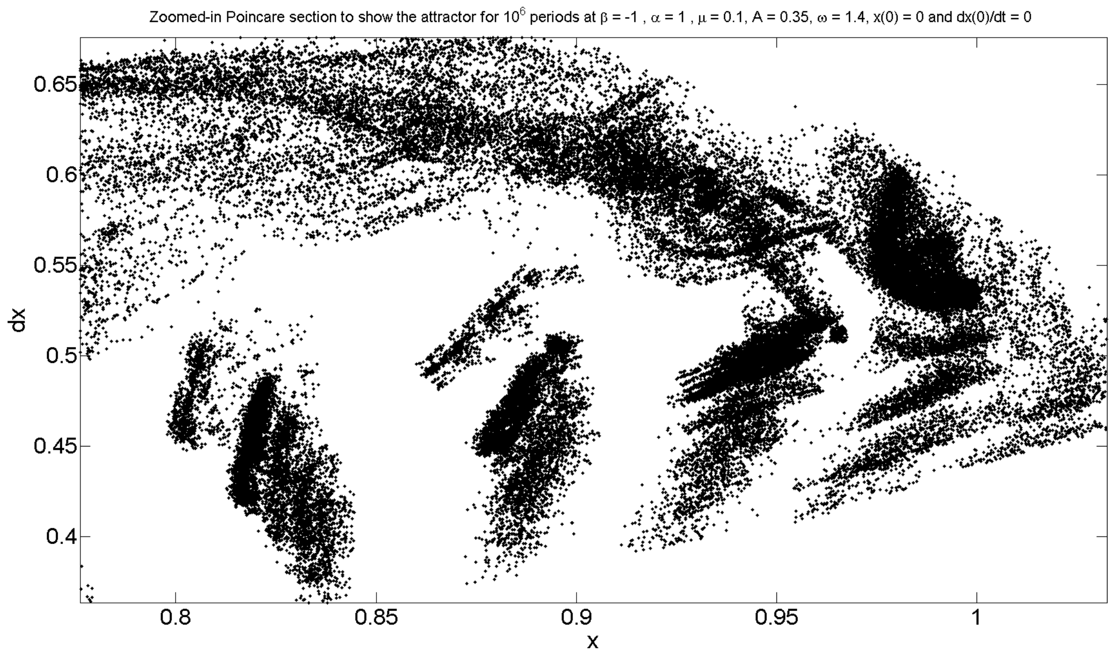

Figure 4.6: The zoomed-in Poincare plot of the Duffing oscillator to show the attractor (in this particular case, we see the similarities in geometry between $0.9 \leq x \leq 1$

4.2 The Lorenz System

The Lorenz system is a mathematical model of convection with three state variables(Equation 1-3.9 of [1]) x,y and z where x is proportional to the amplitude of the fluid velocity circulation in the fluid ring and y and z measure the distribution of temperature around the ring. The chaotic system is described as follows :

$$\frac{\partial x}{\partial t} = \sigma(y-x) \tag{4.2.1}$$

$$\frac{\partial y}{\partial t} = Rx - y - xz \tag{4.2.2}$$

$$\frac{\partial z}{\partial t} = xy - Bz \tag{4.2.3}$$

This system depends on the parameter values of σ,R and B which specify the characteristics of air flow(as specified in the text of [1]). Lorenz modeled the location of a particle moving subject to atmospheric forces and obtained a system of ordinary differential equations. When he solved the system numerically, he found that his particle moved wildly and apparently randomly. After a while, though, he found that while the momentary behaviour of the particle was chaotic, the general pattern of an attractor appeared. In his case, the pattern was the butterfly shaped attractor now known as the Lorenz attractor. We first plot the different response for varying values of R to see the sensitivity of the system. We then study this attractor with the parameters $R = 28$, $B = 8/3$ and $\sigma = 10$ to gain some insight into the physical implications of this system with initial conditions (x,y,z) = (0,1,0). The Poincare maps are made by taking points of inflexion of y where it changes sign and mapping the x-z coordinates at those points [3].

4.2.1 Results for the Lorenz system

We can see the sensitivity of the system in Figure 4.7. We then proceed to study the characteristic ‘butterfly’ shapes of the chaotic parameters (and the conceptualization of the fractal) in Figures 4.8 and 4.9. The Poincare plots are illustrated in Figure 4.10 where we can see the response of the system at the inflexion of y. One of the characteristics of the Lorenz system is that it does not form the filled-in butterfly attractor for values of R less than 24.28 [3], as is evident in Figure 4.11.

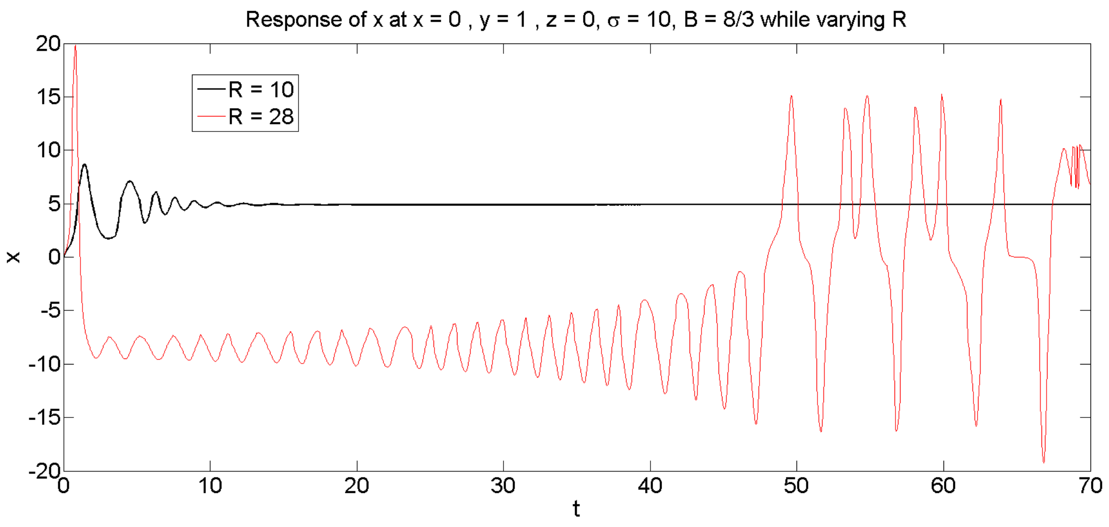

Figure 4.7: The response plots (x vs. t) for the Lorenz system for varying R

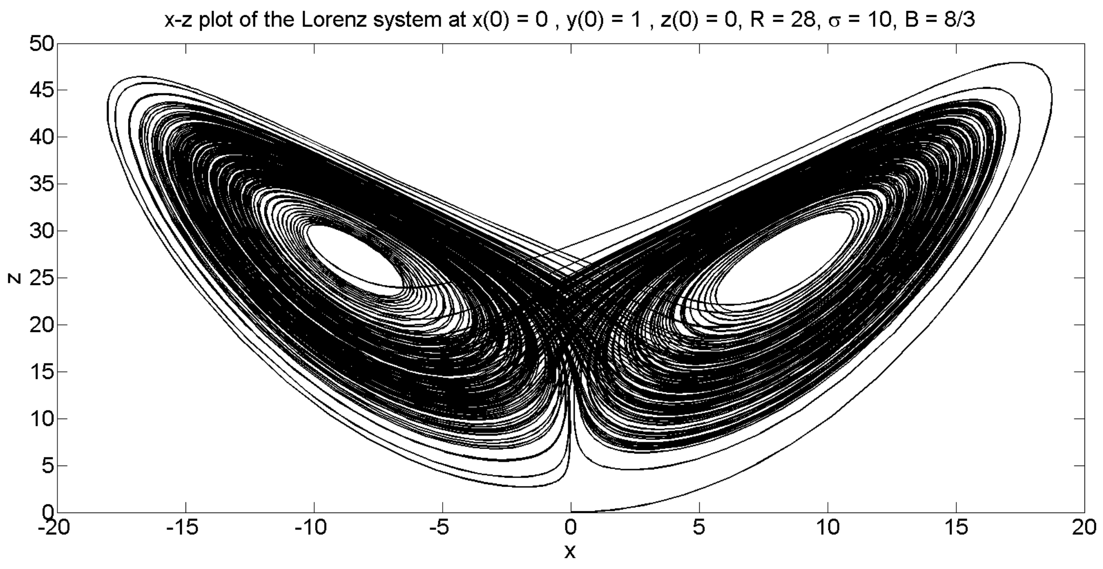

Figure 4.8: The characteristic Lorenz ’butterfly’ in two dimensions (x - z) plot

[2]The parameter values R = 28, B = 8/3 and σ = 10 and initial conditions (x,y,z) = (0,1,0) were chosen from the specified values for chaos from [3].

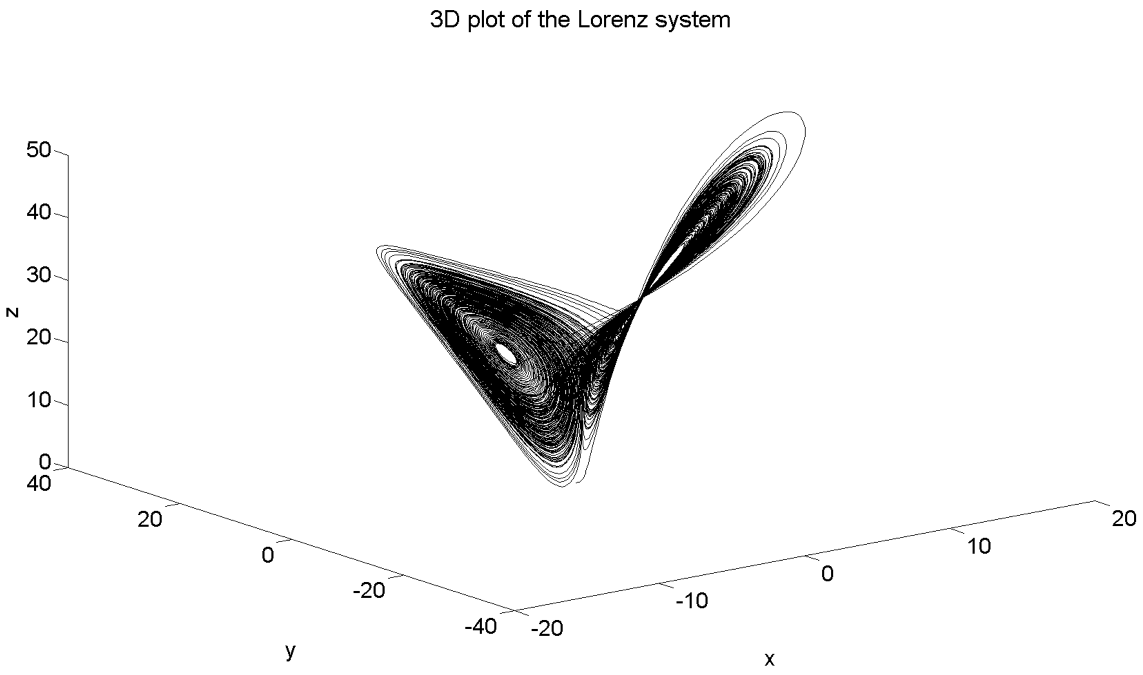

Figure 4.9: The characteristic Lorenz 'butterfly'

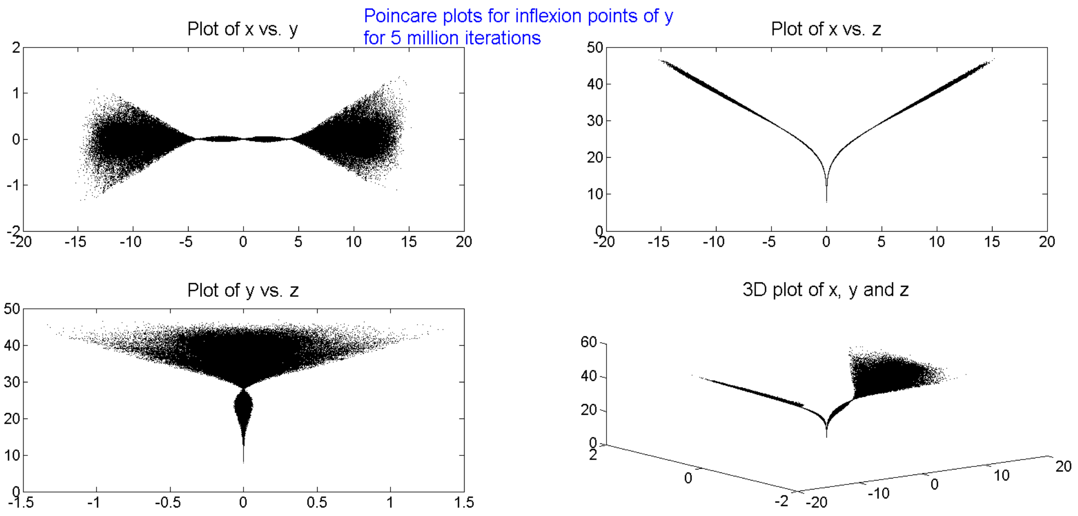

Figure 4.10: Poincare plots for the Lorenz system at the inflexions of y

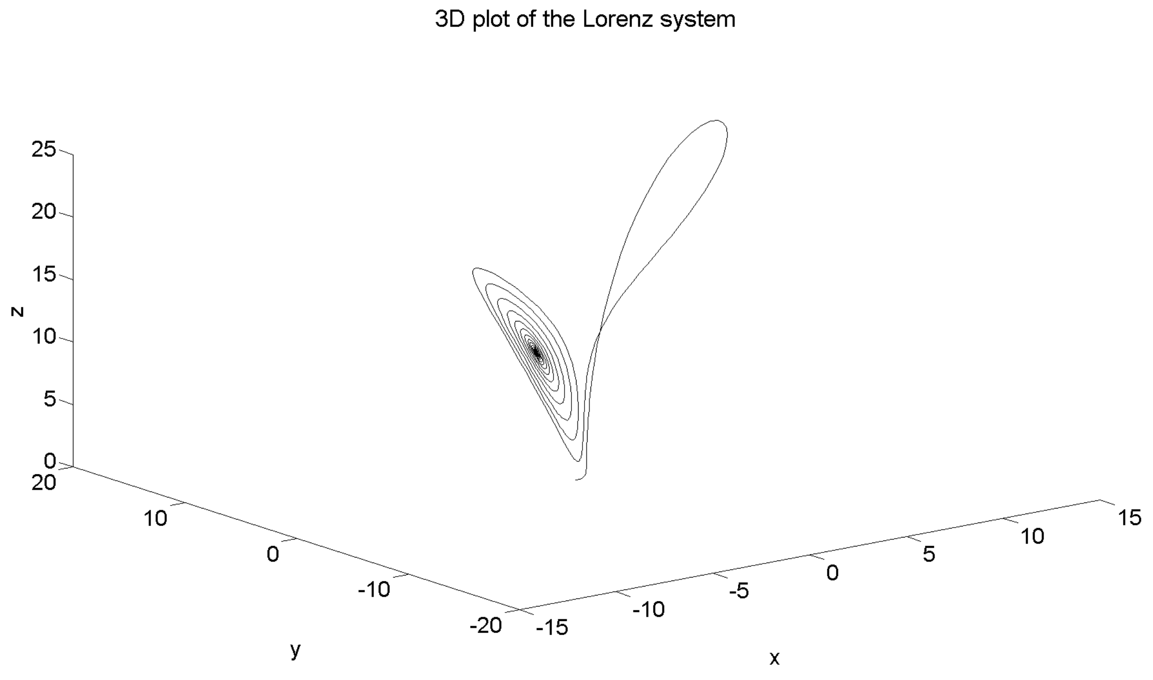

Figure 4.11: The 3D plot of the Lorenz system with $R = 15$

4.3 The Rossler System

The Rossler system a system of three non-linear ordinary differential equations. These equations define a continuous-time dynamical system that exhibits chaotic dynamics. An orbit within the attractor follows an outward spiral close to the plane around an unstable fixed point. In the time domain, it becomes apparent that although each variable is oscillating within a fixed range of values, the oscillations are chaotic. This attractor has some similarities to the Lorenz attractor, but is simpler. Otto Rossler designed the Rossler attractor in 1976, but the original theoretical equations were later found to be useful in modeling equilibrium in chemical reactions [1]. The series does not form limit cycles nor does it ever reach a steady state. Instead it is an example of deterministic chaos. As with other chaotic systems the Rossler system is sensitive to the initial conditions, two initial states no matter how close will diverge, usually sooner rather than later. We use the initial conditions $(x,y,z) = (0,0,0)$ to make a model of this system and study it.

The standard Rossler system [1] is :

$$\frac{\partial x}{\partial t} = -y - z \tag{4.3.1}$$

$$\frac{\partial y}{\partial t} = x + ay \tag{4.3.2}$$

$$\frac{\partial z}{\partial t} = b + z(x - c) \tag{4.3.3}$$

4.3.1 Results for the Rossler system

The characteristic response of the system is observed in Figure 4.12, the characteristic 3D plot analyzed in Figure 4.13 and the Poincare plot (we follow the same methodology as that in the Lorenz system) in Figure 4.14 which shows that the 'spike' in the response occurs at transition points of y. What is interesting about this system is that although the individual responses seem to be quasi-periodic, they form a chaotic system when combined together. As we increase the value of c, we can see a more filled-in chaotic attractor (Figure 4.15) and if we decrease the value of a below zero, the attractor converges to a centrally fixed point (Figure 4.16).

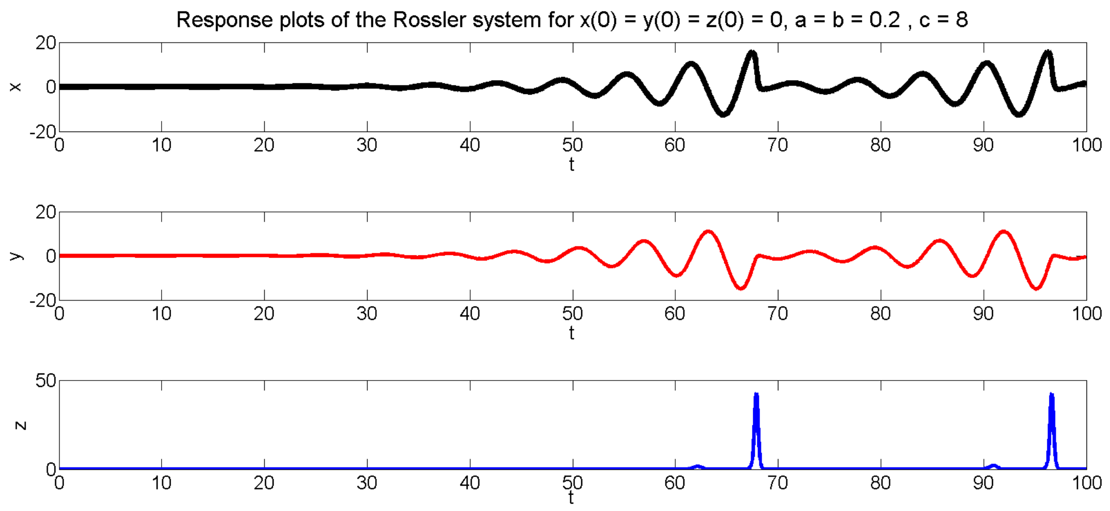

Figure 4.12: The response plots for the Rossler system

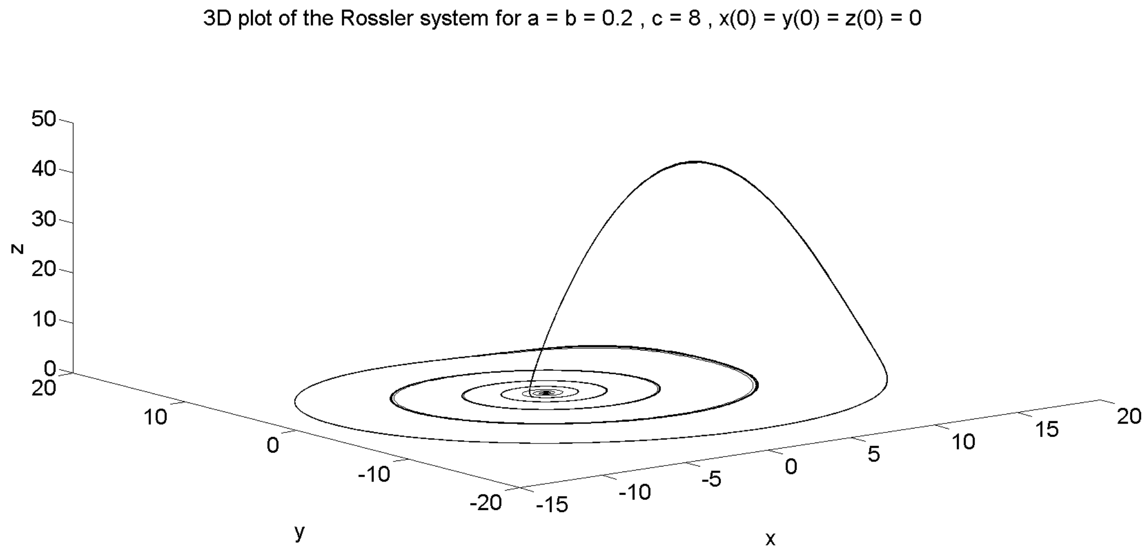

Figure 4.13: The 3D plot of the Rossler system

[3]The parameter values $a = 0.2$, $b = 0.2$ and $c = 8$ and initial conditions $(x,y,z) = (0,0,0)$ were chosen from the specified values for chaos from [1].

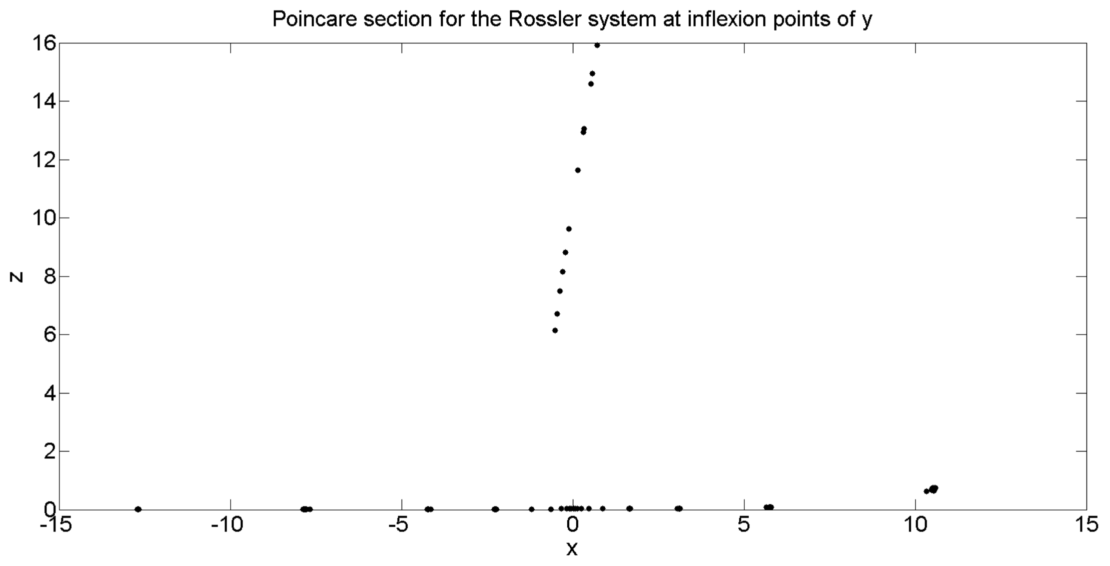

Figure 4.14: Poincare plot for the Rossler system at the inflexions of y

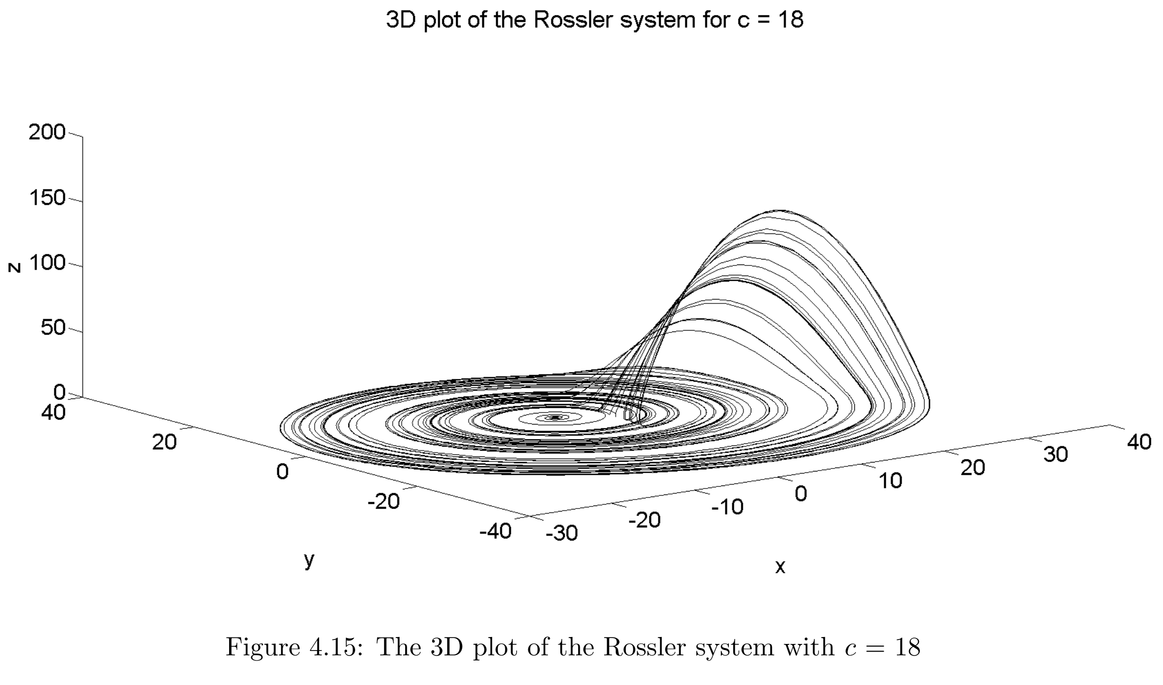

Figure 4.15: The 3D plot of the Rossler system with $c = 18$

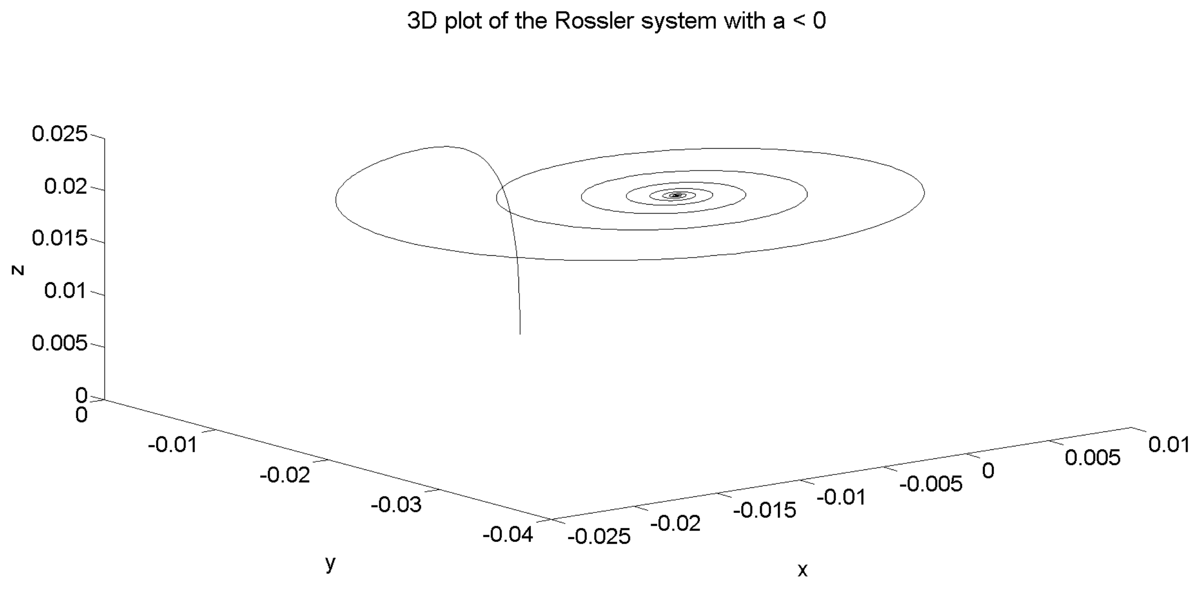

Figure 4.16: The 3D plot of the Rossler system with $a = -0.2$

5 Conclusions

- In all three systems we observe the sensitivity to initial conditions and parameters that is the characteristic of chaotic systems. We could very easily keep varying the parameters endlessly to keep getting different responses, as we can see from the systems we have observed.
- We can clearly observe the sensitivity of the Duffing Oscillator to the parameter β which defines the geometry of the potential well, and the complexity of the zoomed-in Poincare plot which shows us the deeply chaotic nature of the system.
- The Lorenz system is noticeably chaotic and the complexity of the system is seen in the observed fractal pattern. This makes it easier for us to imagine how the complexity of a system of this nature makes it an accepted model for physically relevant systems such as weather.
- The Rossler system shows us that the analysis of chaotic systems requires the understanding of the overall picture, especially concepts like Poincare sections and 3D plots, since its chaotic nature is not clearly apparent simply from the seemingly non-chaotic response plots and instead requires both a three-dimensional plot as well as a Poincare section to fully illustrate its chaotic nature.

Thus, we can begin to understand the manifestation of chaos in the physical world, how useful the computer is to solve physical problems which cannot be solved otherwise using methods such as the adaptive step-size Runge-Kutta and how we can, in turn, use those results to interpret the physical phenomena occurring in the system.

References

[1] Francis C. Moon *Chaotic and Fractal Dynamics - An Introduction for Applied Scientists and Engineers* Wiley (2004)

[2] 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)

[3] H. J. Korsch and H.J. Jodl *Chaos, A Program Collection for the PC, 2nd edit.,* SpringerVerlag, (1999),

A Source Code Listing

A.1 The Duffing Oscillator

```
/* Program to run RK5 on a Duffing oscillator
 * File:   duffing.cpp
 * Author: Arjun
 *Run on an Intel Core i7 machine
 * gcc 4.6.1 compiler, NetBeans IDE
 */


#include<cstdio>
#include<cmath>
#include<conio.h>
#include<cstdlib>
/*----All commented out printfs were used for debugging purposes----*/
/*--------------------------------RK5------------------------------*/
double ode5(double yold[], double ynew[], double &h, double t,
                //&h - Call by reference - Increments h in main
        int numberofequations, void frhs(double[], double, double[])) {
    int i;
    double *k1, *k2, *k3, *k4, *k5, *k6, *temp, *ynewstar; //*yerror;  //pointers to arrays

    //Values from Numerical Recipes

    double c2 = 0.2, c3 = 0.3, c4 = 0.8, c5 = 8.0 / 9.0, //Constants
            a21 = 0.2, //Coefficients
            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, a73 = 500.0 / 1113.0, a74 = 125.0 / 192.0,
            a75 = -2187.0 / 6784.0, a76 = 11.0 / 84.0;
    /*Error value coefficients*/
    double e1 = 71.0 / 57600.0, e3 = -71.0 / 16695.0, e4 = 71.0 / 1920.0,
            e5 = -17253.0 / 339200.0, e6 = 22.0 / 525.0, e7 = -1.0 / 40.0;
    double *error, *scale; //Scale now
    double hmin = 1.0e-10;
    //Minimum size to avoid getting stuck in an infinite loop

    double tolerable_error = 1.0e-7; //Error limit
    double error_min = 1.0e-6; //Error limit for scale
    double max_error_i; //To find maximum error
    k1 = new double[10 * numberofequations];
    //integer*numberofequations where the integer=# of arrays being used

    if (NULL == k1) {
```

[4]The C++ code was written using the gcc 4.6.1 compiler and the NetBeans 7.1.1 IDE, and all resulting data was plotted using Matlab.

```
        printf("Cannot allocate k1 in ode5() \n");
        return (0);
    }
    k2 = k1 + numberofequations;
    k3 = k2 + numberofequations;
    k4 = k3 + numberofequations;
    k5 = k4 + numberofequations;
    k6 = k5 + numberofequations;
    temp = k6 + numberofequations;
    ynewstar = temp + numberofequations; //For the embedded 4th order
    error = ynewstar + numberofequations; //For the error array
    scale = error + numberofequations; //For the scale array
    frhs(yold, t, k1); // Get the RHS
    //printf("\n Calling frhs (yold,t,k1) \n");


repeat_current_step: //goto label
    //printf("h = %f\n",h);
    double h_remember = h;
    for (i = 0; i < numberofequations; i++)//Step 1
    {
        temp[i] = yold[i] + a21 * h * k1[i];
    }
    frhs(temp, t + c2*h, k2); //Step 2
    //printf("\n Calling frhs (yold,t+c2*h,k2) \n");
    for (i = 0; i < numberofequations; i++) {
        temp[i] = yold[i] + h * (a31 * k1[i] + a32 * k2[i]);
    }
    //printf("\n Calling frhs (yold,t+c3*h,k3) \n");
    frhs(temp, t + c3*h, k3); //Step 3
    for (i = 0; i < numberofequations; i++) {
        temp[i] = yold[i] + h * (a41 * k1[i] + a42 * k2[i] + a43 * k3[i]);
    }
    //printf("\n Calling frhs (yold,t+c4*h,k4) \n");
    frhs(temp, t + c4*h, k4); //Step 4
    for (i = 0; i < numberofequations; i++) {
        temp[i] = yold[i] + h * (a51 * k1[i] + a52 * k2[i] +
                a53 * k3[i] + a54 * k4[i]);
    }
    //printf("\n Calling frhs (yold,t+c5*h,k5) \n");
    frhs(temp, t + c5*h, k5); //Step 5
    for (i = 0; i < numberofequations; i++) {
        temp[i] = yold[i] + h * (a61 * k1[i] + a62 * k2[i] + a63 * k3[i]
                            + a64 * k4[i] + a65 * k5[i]);
    }
    //printf("\n Calling frhs (yold,t+c6*h,k6) \n");
    frhs(temp, t + h, k6); //Step 6
    for (i = 0; i < numberofequations; i++) {
        ynew[i] = yold[i] + h * (a71 * k1[i] + a73 * k3[i] + a74 * k4[i]
                            + a75 * k5[i] + a76 * k6[i]);
    }

    //printf("\n Final \n");
    frhs(ynew, t + h, ynewstar); //Final

    for (i = 0; i < numberofequations; i++)//to set up the scale
```

```
        scale[i] = fabs(yold[i]) + fabs(h * k1[i]) + fabs(error_min);

    for (i = 0; i < numberofequations; i++)//Estimate error
    {
        error[i] = (h * (e1 * k1[i] + e3 * k3[i] + e4 * k4[i]
                + e5 * k5[i] + e6 * k6[i] + e7 * ynewstar[i])) / scale[i];
        //printf("Error(%d) = %g\n",i,error[i]);
    }

    max_error_i = fabs(error[0]);
    for (i = 0; i < numberofequations; i++) {
        if (fabs(error[i]) > max_error_i) {
            max_error_i = fabs(error[i]);
        }
    }//Finding the maximum error

    if (max_error_i > tolerable_error) {
        h = h / 5;

        if (h < hmin) exit(0);

        goto repeat_current_step;
    }

    if (max_error_i < tolerable_error) {
        h = h * (pow(tolerable_error / max_error_i, 0.2));
    }
    //printf("h value in ode5 = %f\n",h);

    //getch();
    //exit(0);

    delete[]k1; //Free memory
    return (h_remember);
}//end ode5
//end rk5
const int N = 2; /*For Duffing oscillator, SHM and Mathieu system*/

//evaluate the ODE RHS
//Change for each problem
/*Old systems for testing*/
void myrhs1(double yold[], double t, double f[])//Harmonic oscillator
{
    f[1] = -yold[0]; //Harmonic oscillator with omega = 1 //f[1] = y'(t)
    f[0] = yold[1]; //f(0) = y(t)
}

void myrhs(double yold[], double t, double f[])
//Chaotic system - Mathieu equation
{
    f[1] = -(cos(2 * t) * yold[0]);
    f[0] = yold[1];
}

/*--------------Duffing Oscillator-------------*/
void duffing(double yold[], double t, double f[]) {
    /*Parameters :
     * alpha    -       a
     * beta     -       b
     * damping coefficient      -       mu
     * mass     -       m
     * periodicity coefficient  -       A
     * frequency        -       w       (omega)
     */


    double b = 1, a = 1, mu = 0.1, A = 0.35, w = 1.4, m = 1;
    f[1] = (-b * yold[0] - a * yold[0] * yold[0] * yold[0]
                - mu * yold[1] + A * cos(w * t)) / m;

    f[0] = yold[1]; //f(0) = y(t)
}

//Main program

int main() {
    int istep; //nstep ;
    double yold[N], ynew[N];
    double t, t_initialize = 0.0; //initial value of t
    double tmax;
    double h = 1; //step size
    int n_period = 100;
    double h_increment;

    FILE *fp, *fp1;
    t = t_initialize;
    double pi = 4 * atan(1);
    double w = 1.4;
    tmax = 2 * pi * n_period / w;

    yold[0] = 0; //y(0)=1
    yold[1] = 0; //y'(0)=0
    /* To open a new output file and give it a title */
    fp = fopen("Duffing.dat", "w+");
    if (NULL == fp) {
        printf(" Unable to open the file \n");
        return ( 0);

    }
    fp1 = fopen("PoincareDuffing.dat", "w+");
    if (NULL == fp1) {
        printf(" Unable to open the file \n");
        return ( 0);

    }
    //printf("\n Starting process \n\n");
    int time = 0;

    for (istep = 0; t < tmax; istep++) {

        //printf("istep = %d \n",istep);
        h_increment = ode5(yold, ynew, h, t, N, duffing);
        //Uses call by reference

        //printf("t,ynew[0],ynew[1]=%f,%f,%f\n",t,ynew[0],ynew[1]);
        fprintf(fp, "%f %f %f\n", t, ynew[0], ynew[1]);

       for (int i = 0; i < N; i++)
       {
            yold[i] = ynew[i];
       }

        //Print out ynew and copy ynew-->yold
        t = t + h_increment;
        //printf("h in main = %f\n",h);
    }

    /*For the Poincare section
     * Set up a loop with tmin = 0 and tmax = one period ( 1*T)
     * Perform RK5 and get the values
     * Loop it such that tmin = 1*T and tmax = 2T
     * Perform RK5 and get the values
     * Keep doing this for as many time periods as needed
     */


    double tbound = 2 * pi / w;
    /*Set the initial conditions again*/
    t = t_initialize;
    yold[0] = 0; //y(0)=1
    yold[1] = 1; //y'(0)=0
    int count = 0;

    while (t < tmax) {
        for (istep = 0; t < tbound; istep++)
        {
            //printf("istep = %d \n",istep);
            h_increment = ode5(yold, ynew, h, t, N, duffing);
            //printf("t,ynew[0],ynew[1]=%f,%f,%f\n",t,ynew[0],ynew[1]);

            for (int i = 0; i < N; i++)
            {
                yold[i] = ynew[i];
            }

            t = t + h_increment;

        }
        fprintf(fp1, "%f %f %f\n", t, ynew[0], ynew[1]);
        //printf("%d\t%f\n",count,tbound);
        count = count + 1;
        tbound = tbound + 2 * pi / w;
    }
    return (0);
    //end
}
```

A.2 The Lorenz and Rossler systems

```
/* Lorenz and Rossler systems
 * File:   lorenz.cpp
 * Author: Arjun
 *
 * Created on May 1, 2012, 8:45 PM
 */
#include<cstdio>
#include<cmath>
#include<conio.h>
#include<cstdlib>
/*----All commented out printfs were used for debugging purposes----*/
/*-------------------------------RK5-----------------------------*/
double ode5(double yold[], double ynew[], double &h, double t,
        //&h - Call by reference - Increments h in main
        int numberofequations, void frhs(double[], double, double[])) {
    int i;
    double *k1, *k2, *k3, *k4, *k5, *k6, *temp, *ynewstar;
    //*yerror;  //pointers to arrays
    //Values from Numerical Recipes
    double c2 = 0.2, c3 = 0.3, c4 = 0.8, c5 = 8.0 / 9.0, //Constants
            a21 = 0.2, //Coefficients
            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, a73 = 500.0 / 1113.0, a74 = 125.0 / 192.0,
            a75 = -2187.0 / 6784.0, a76 = 11.0 / 84.0;

    /*Error value coefficients*/
    double e1 = 71.0 / 57600.0, e3 = -71.0 / 16695.0, e4 = 71.0 / 1920.0,
            e5 = -17253.0 / 339200.0, e6 = 22.0 / 525.0, e7 = -1.0 / 40.0;
    double *error, *scale; //Scale now
    double hmin = 1.0e-10; //Minimum size to avoid infinite loop
    double tolerable_error = 1.0e-6; //Error limit
    double error_min = 1.0e-2; //Error limit for scale
    double max_error_i; //To find maximum error
    k1 = new double[10 * numberofequations];
    //integer*numberofequations where the integer=# of arrays being used
    if (NULL == k1) {
        printf("Cannot allocate k1 in ode5() \n");
        return (0);
    }
    k2 = k1 + numberofequations;
    k3 = k2 + numberofequations;
    k4 = k3 + numberofequations;
    k5 = k4 + numberofequations;
    k6 = k5 + numberofequations;
    temp = k6 + numberofequations;
    ynewstar = temp + numberofequations; //For the embedded 4th order
    error = ynewstar + numberofequations; //For the error array
    scale = error + numberofequations; //For the scale array
    frhs(yold, t, k1); // Get the RHS
    //printf("\n Calling frhs (yold,t,k1) \n");


repeat_current_step: //goto label
    //printf("h = %f\n",h);
    double h_remember = h;
    for (i = 0; i < numberofequations; i++)//Step 1
    {
        temp[i] = yold[i] + a21 * h * k1[i];
    }
    frhs(temp, t + c2*h, k2); //Step 2
    //printf("\n Calling frhs (yold,t+c2*h,k2) \n");
    for (i = 0; i < numberofequations; i++) {
        temp[i] = yold[i] + h * (a31 * k1[i] + a32 * k2[i]);
    }
    //printf("\n Calling frhs (yold,t+c3*h,k3) \n");
    frhs(temp, t + c3*h, k3); //Step 3
    for (i = 0; i < numberofequations; i++) {
        temp[i] = yold[i] + h * (a41 * k1[i] + a42 * k2[i] + a43 * k3[i]);
    }
    //printf("\n Calling frhs (yold,t+c4*h,k4) \n");
    frhs(temp, t + c4*h, k4); //Step 4
    for (i = 0; i < numberofequations; i++) {
        temp[i] = yold[i] + h * (a51 * k1[i] + a52 * k2[i]
                + a53 * k3[i] + a54 * k4[i]);
    }
    //printf("\n Calling frhs (yold,t+c5*h,k5) \n");
    frhs(temp, t + c5*h, k5); //Step 5
    for (i = 0; i < numberofequations; i++) {
        temp[i] = yold[i] + h * (a61 * k1[i] + a62 * k2[i]
                        + a63 * k3[i] + a64 * k4[i] + a65 * k5[i]);
    }
    //printf("\n Calling frhs (yold,t+c6*h,k6) \n");
    frhs(temp, t + h, k6); //Step 6
    for (i = 0; i < numberofequations; i++) {
        ynew[i] = yold[i] + h * (a71 * k1[i] + a73 * k3[i]
                + a74 * k4[i] + a75 * k5[i] + a76 * k6[i]);
    }

    //printf("\n Final \n");
    frhs(ynew, t + h, ynewstar); //Final

    for (i = 0; i < numberofequations; i++)//to set up the scale
        scale[i] = fabs(yold[i]) + fabs(h * k1[i]) + fabs(error_min);

    for (i = 0; i < numberofequations; i++)//Estimate error
    {
        error[i] = (h * (e1 * k1[i] + e3 * k3[i] + e4 * k4[i] +
            e5 * k5[i] + e6 * k6[i] + e7 * ynewstar[i])) / scale[i];
        //printf("Error(%d) = %g\n",i,error[i]);
    }

    max_error_i = fabs(error[0]);
    for (i = 0; i < numberofequations; i++) {
        if (fabs(error[i]) > max_error_i) {
            max_error_i = fabs(error[i]);
        }
    }//Finding the maximum error

    if (max_error_i > tolerable_error) {
        h = h / 5;

        if (h < hmin) exit(0);

        goto repeat_current_step;
    }

    if (max_error_i < tolerable_error) {
        h = h * (pow(tolerable_error / max_error_i, 0.2));
    }
    //printf("h value in ode5 = %f\n",h);

    //getch();
    //exit(0);

    delete[]k1; //Free memory
    return (h_remember);
}//end ode5
//end rk5
const int N = 3; /*For Lorenz system*/

void lorenz(double x[], double t, double dx[]) {
    double R = 28, B = 8.0 / 3.0, Sigma = 10.0;

    dx[0] = Sigma * (-x[0] + x[1]);
    dx[1] = R * x[0] - x[1] - x[0] * x[2];
    dx[2] = -B * x[2] + x[0] * x[1];
}

void rossler(double x[], double t, double dx[]) {
    double a = 0.2, b = 0.2, c = 10.0;

    dx[0] = -x[1] - x[2];
    dx[1] = x[0] + a * x[1];
    dx[2] = b + x[0] * x[2] - c * x[2];
}
//Main program
int main() {
    int istep; //nstep ;
    double x[N], dx[N];
    double x1[N], dx1[N];

    //To check for sign change in y for Poincare section
    double flagx[N], flagt, flagdx[N];



    /* x[N] -->  x , y , z
     * dx[N] -->  dx/dt , dy/dt , dz/dt
     */
```

```

double t, t_initialize = 0.0; //initial value of t
double tmax;
double h = 0.5; //step size

double h_increment;

FILE *fp, *fp1, *fp2;
t = t_initialize;

tmax = 5e2;

/*Lorenz plot*/
x[0] = 0.0; //x0
x[1] = 1.0; //y0
x[2] = 0.0; //z0

/*Rossler plot*/
x1[0] = 0.0; //x0
x1[1] = 0.0; //y0
x1[2] = 0.0; //z0



/* To open a new output file and give it a title */
fp = fopen("Lorenz.dat", "w+");
if (NULL == fp) {
    printf(" Unable to open the file \n");
    return ( 0);

}

fp1 = fopen("PoincareLorenz.dat", "w+");
if (NULL == fp1) {
    printf(" Unable to open the file \n");
    return ( 0);
}

fp2 = fopen("Rossler.dat", "w+");
if (NULL == fp2) {
    printf(" Unable to open the file \n");
    return ( 0);

}
//printf("\n Starting process \n\n");
for (istep = 0; t < tmax; istep++) {

    for (int i = 0; i < N; i++) {
        flagx[i] = x[i];
        flagdx[i] = dx[i]; //Change to x1,dx1 for Rossler
        flagt = t;
    }
    //printf("istep = %d \n",istep);
    h_increment = ode5(x, dx, h, t, N, lorenz); //Uses call by reference
    //printf("%f\t%f\t%f\t%f\t%f\t%f\t%f\n",
    //       t , x[0], x[1], x[2],dx[0] , dx[1] , dx[2]);
```

```
        fprintf(fp, "%f\t%f\t%f\t%f\t%f\t%f\t%f\n",
                        t, x[0], x[1], x[2], dx[0], dx[1], dx[2]);





        h_increment = ode5(x1, dx1, h, t, N, rossler); //Uses call by reference
        fprintf(fp2, "%f\t%f\t%f\t%f\t%f\t%f\t%f\n",
                        t, x1[0], x1[1], x1[2], dx1[0], dx1[1], dx1[2]);



        for (int i = 0; i < N; i++) {
            x[i] = dx[i];
            x1[i] = dx1[i];
        }

        /*For Poincare section where y changes sign (crosses zero)*/
        if (((flagx[1] < 0) && (x[1] > 0)) || ((flagx[1] > 0) && (x[1] < 0)))
        {
            //Change to x1,dx1 for Rossler
            fprintf(fp1,"%f\t%f\t%f\t%f\t%f\t%f\t%f\n",
                    0.5 * (t + h_increment + flagt),
                    0.5 * (flagx[0] + x[0]),
                    0.5 * (flagx[1] + x[1]), 0.5 * (flagx[2] + x[2]),
                    0.5 * (flagdx[0] + dx[0]), 0.5 * (flagdx[1] + dx[1]),
                    0.5 * (flagdx[2] + dx[2]));
        }
        //Print out ynew and copy ynew-->yold
        t = t + h_increment;
        //printf("h in main = %f\n",h);
    }
    return (0);
    //end
}
```

[5] Document compiled using LaTeXand TeXnicCenter.