John Sy
5 November 2007

Modelling in Biology

Assignment 1

Question 1: Simple numerics on the single exponential

Part 1

We first define a function, `func` as follows:

```
function xprime = func(t,x)
xprime = -(2/3)*x;
```

Then we can implement Runge-Kutta function (`ode45`) to integrate the function numerically:

```
time = [0 5];
x0=10;
[T,Y] = ode45(@func, time, x0);
plot(T,Y);
```

The analytical solution can also be derived from integrating the equation:

$$\frac{dx}{dt} = -\frac{2}{3}x$$

$$\int \frac{dx}{x} = \int -\frac{2}{3}dt$$

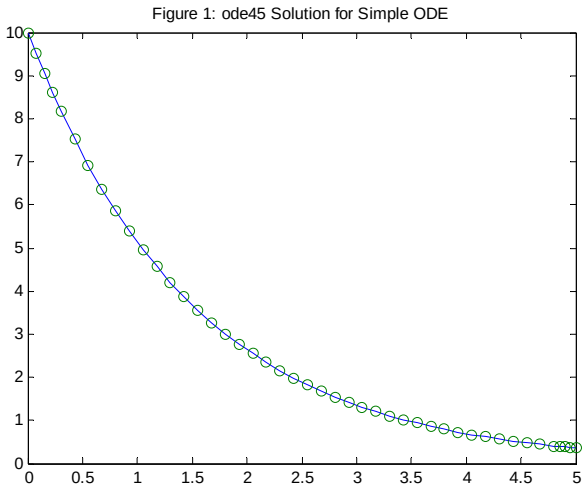
$$\ln x = -\frac{2}{3}t + C$$

$$x = x_0 e^{-\frac{2}{3}t}$$

Using the same time points derived from numerical integration with ode45, we can solve for corresponding values of x and compare the result of the Runge-Kutta method with the analytical solution.

```
Z = x0*exp(-(2/3)*T);
plot(T,Y,T,Z,'o')
title('ode45 solution for simple ODE')
```

Plotting both the analytical solution and the numerical solution on the same plot gives us an idea of how good the numerical integrator utilized by Matlab really is. The analytical solution is represented by circles while the numerical solution is represented by the solid line. As can be seen, the Runge-Kutta 4th order method of numerical integration gives satisfactory results with a cursory inspection.



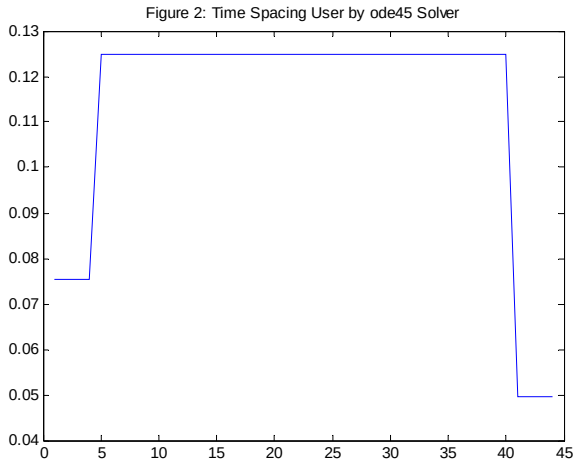
To obtain a more quantitative value of the error, we can calculate the mean squared error between the analytical solution and the numerical solution. Matlab's mse function enables a quick and easy way to do this:

```
msetest = mse(Z-Y)
```

Resulting in the mean squared error of 3.1489e-011 between the analytical and numerical solution (provided by ode45).

Part 2

To analyze more carefully the way in which ode45 solves differential equations, we can first look at the difference in time points. As is seen in the figure above, the spacing between time points is not equal, but becomes clustered towards the end.

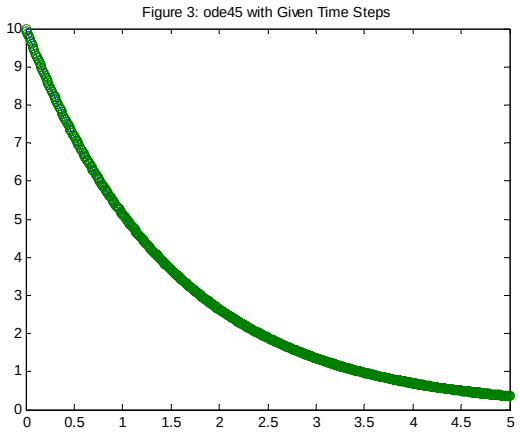


As is seen in figure 2, the solver begins with time points which are close together, then switches to time points further apart in the middle of the range, and lastly, the time points become closer together again. From this plot, we can deduce that the solver takes more time points at the beginning and towards the end of the function since it cannot rely on points beyond the limit set by the user. The Runge-Kutta method takes into account several points around the point it wants to integrate, and because of limitations of projections before the range and after the range, it must take smaller time points. By doing this, the solver saves computational time as well.

If we want equally spaced time points, we can also force the ode45 solver to take specific time points:

```
t=[0:0.01:5];
[T1,Y1] = ode45(@func, t, x0);
Z1 = x0*exp(-(2/3)*T1);
figure;
plot(T1,Y1,T1,Z1,'o')
title('Figure 3: ode45 with Given Time Steps')
```

Plotting the analytical solution as circles again superimposed on the numerical solution (line), we obtain:

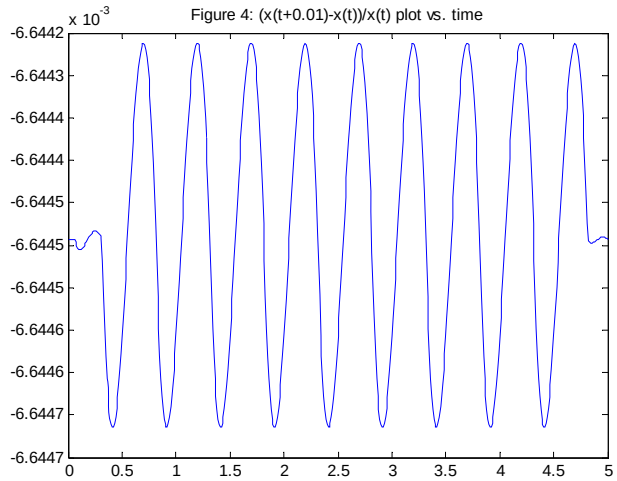


Again, there is no noticeable difference between the analytical and numerical solution, but we should calculate the mean squared error to determine a quantitative value for the error.

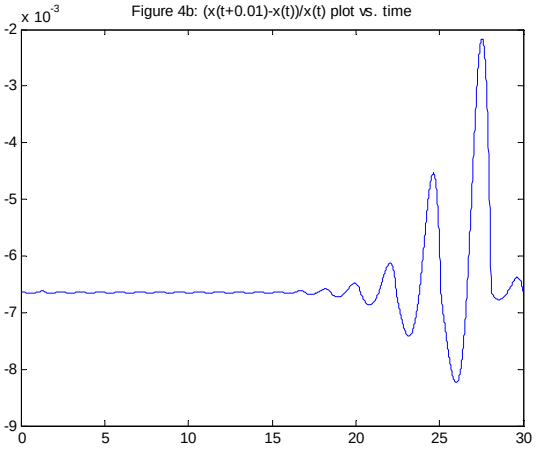
Using the mean squared error function as before yields an error of 3.4371e-011. Although we used fixed time points, there is actually a slightly greater error than with non-fixed time points, suggesting that the Runge-Kutta function corrects for time steps that would yield a higher error.

To investigate further into the mechanism of the Runge-Kutta method, we can look at the differences between two adjacent points. For each point, we can also calculate the percentage change to the next point using the numerical solution we obtained with fixed time points.

```
t2c=[0.01:0.01:5];
diffmat = diff(Y1);
foo = diffmat./Y1(1:length(Y1)-1);
figure;
plot(t2c,foo)
title('Figure 4: (x(t+0.01)-x(t))/x(t) plot vs. time')
```



This results in an interesting plot (figure 4). The solver seems to oscillate. But if we look at a longer time period, we see something quite remarkable, that as you increase the time, the oscillations become unpredictable (figure 4b).



If the time is increased even further (say t = 80), we get even more unpredictable behavior with the value reaching up to 80. Looking at the function that we are plotting carefully, one can see that it is similar to the definition of the derivative, and it indeed the derivative of the phase plane multiplied by the time step, *h*. Since we are plotting a linear differential equation in the form $\dot{x} = kx$, the limit of the value at small time, t, is just $\frac{k}{h}$. In fact this is what we see in the above equation with the graph approaching -6.67e-3 at small values of time. However, since we are doing numerical calculations and not analytical calculations, there are inherent round-off errors that tend to become multiplied over time. These round-off errors become increasingly important as the value of *x* becomes small since it represents a larger proportion of the function. Matlab compensates for this, which results in the oscillations seen in figure 4. As the error increases and subsequently the oscillations increase, Matlab takes greater steps to correct and overcompensates leading to large fluctuations seen at higher time values.

Question 2: Euler Method Implementation

The Euler method provides a simple way of generating a numerical solution to differential equations and is based on the definition of a derivative. Implementing this will allow us to compare between the Runge-Kutta method that Matlab selects and this more rudimentary method of numerical integration.

$$x(t+h) = x(t) + h[-kx(t)]$$

In Matlab this is implemented as such:

```
k = (2/3);
h = 0.01; %time step
myeuler(1) = 10;
timesteps1 = 5/h;

for t = [2:1:timesteps1+1]
    myeuler(t) = myeuler(t-1) - h*k*myeuler(t-1);
end

timevect1 = [0:h:5];
figure;
plot(timevect1,myeuler)
title('Figure 5: h = 0.01 implementation of the Euler algorithm')
```

Calculating the mean squared error gives us an error of 8.0358e-005. This is 6 orders of magnitude larger than the Runge-Kutta method.

Figure 5: h = 0.01 implementation of the Euler algorithm

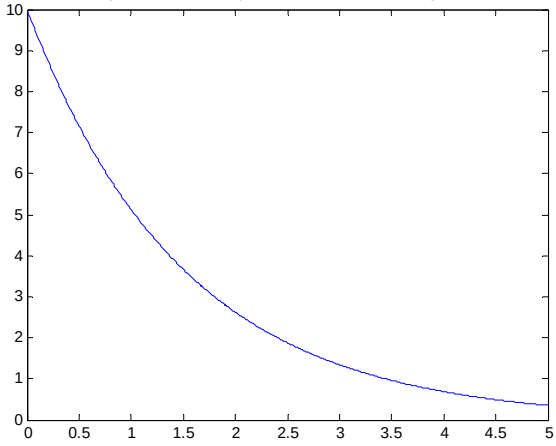
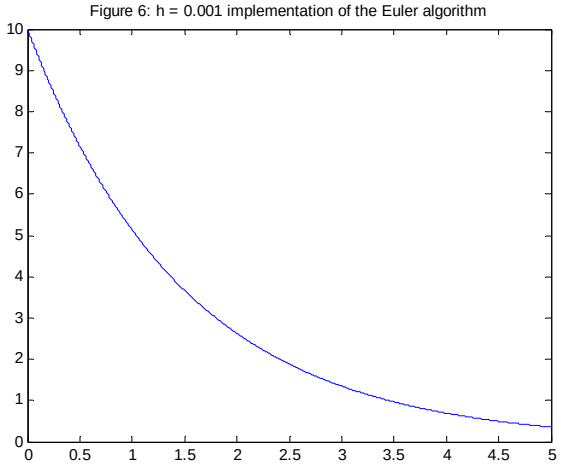


Figure 6: h = 0.001 implementation of the Euler algorithm

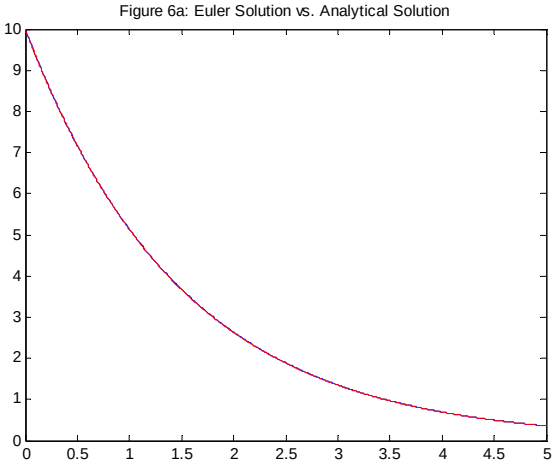


A more easy comparison comes when we try using h = 0.001 as our time step and repeating the calculation of the Euler algorithm (Figure 6).

Although there is no visible difference between the two plots (Figures 5 and 6) using ten times more points, we can calculate the mean squared error again to determine a quantitative value. The calculated mean square error is 8.0183e-007, which is 2 orders of magnitude smaller than using h = 0.01. This is surprising as one would expect that with an order of magnitude more time points, there would be a corresponding order of magnitude decrease in error.

In figure 6a, we can see that although the error is much higher than the Runge-Kutta method, our Euler method does pretty well for a first approximation.

Figure 6a: Euler Solution vs. Analytical Solution



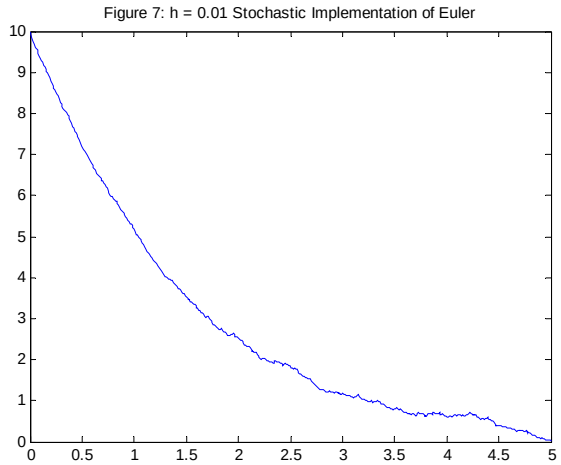
Part 2

To simulate noise in the environment, we can include a stochastic term in our differential equation where σ represents the amplitude of the random noise process dW .

$$dx = -kxdt + \sigma dW$$

In Matlab, we can use the rand function to generate random numbers and the stochastic differential equation can be solved also using the Euler algorithm in an adaptation to the code presented in the previous section.

```
k = (2/3);
h3 = 0.01;
mystoeul(1) = 10;
sig = 0.2;
timesteps3 = 5/h3;

for t = [2:1:timesteps3+1];
    mystoeul(t) = mystoeul(t-1) - h3*k*mystoeul(t-1) + sig*sqrt(h3)*randn;
end

timevect3 = [0:h3:5];
figure;
plot(timevect3,mystoeul)
title('Figure 7: h = 0.01 Stochastic Implementation of Euler')
```



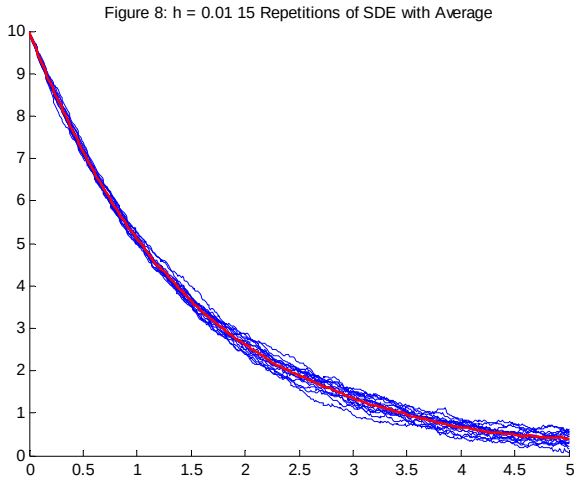
With each run of the program, the stochastic implementation generates a new graph and reflects the unpredictable nature of stochastic differential equations. One would expect that the average of an infinite number of runs of the algorithm will result in the deterministic solution. This exact method is used in image processing to reduce the noise level in an image by averaging many photographs of the same thing.

If we plot 15 runs of the solution to the SDE and take an average (Figure 8), we can see that the average (red line) approaches the deterministic solution.

```
storevals = zeros(15,5/h3);
mystoeul1(1) = 10;
storevals(:,1) = mystoeul1(1);

figure;
for n = 1:1:15
    for t = [2:1:timesteps3+1];
        mystoeul1(t) = mystoeul1(t-1) - h3*k*mystoeul1(t-1) +
sig*sqrt(h3)*randn;
        storevals(n,t) = mystoeul1(t);
    end
end

for n = 1:1:15
    hold on;
    plot(timevect3,storevals(n,:))
end
title('Figure 8: h = 0.01 15 Repetitions of SDE with Average')

for i = 1:1:timesteps3+1
    sumvals(i) = sum(storevals(:,i)); %sums values
end

avgvals = sumvals./15; %calculates the average values
```

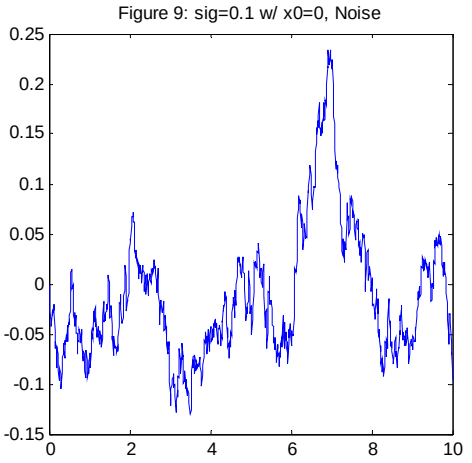
```
plot(timevect3,avgvals,'red','LineWidth',2)
hold off;
```

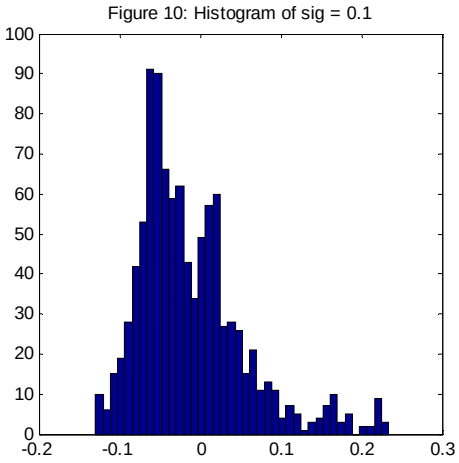
One interesting thing to note about Figure 8 is that there seems to be less noise at the beginning when x is large. This is in part due to accumulation of noise throughout the run as $x(t+h)$ value is dependent upon $x(t)$. The actual noise itself is generated from a Gaussian distribution (if using `randn`) or from a uniform distribution (if using `rand`).

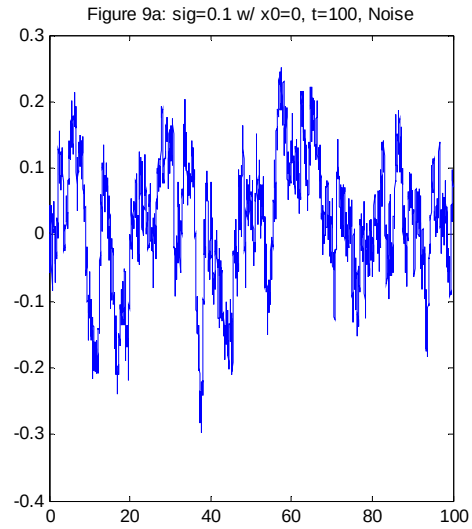


To get a more accurate picture of what is happening with the noise, we can effectively just plot the noise over time and generate a histogram of values. We can also calculate the mean and the standard deviation of the noise generated by the function `randn`.

```
k = (2/3);
step = 0.01;
mysto(1) = 0;
sig = 0.1;
timeval = 10;
tstep = timeval/step;

for t = [2:1:tstep+1];
    mysto(t) = mysto(t-1) - step*k*mysto(t-1) + sig*sqrt(step)*randn;
end

tvect = [0:step:timeval];
figure;
subplot(1,2,1), plot(tvect,mysto)
title('Figure 9: sig=0.1 w/ x0=0, Noise')
subplot(1,2,2), hist(mysto,40)
title('Figure 10: Histogram of sig = 0.1')

mean02 = mean(mysto)
std02 = std(mysto)
```







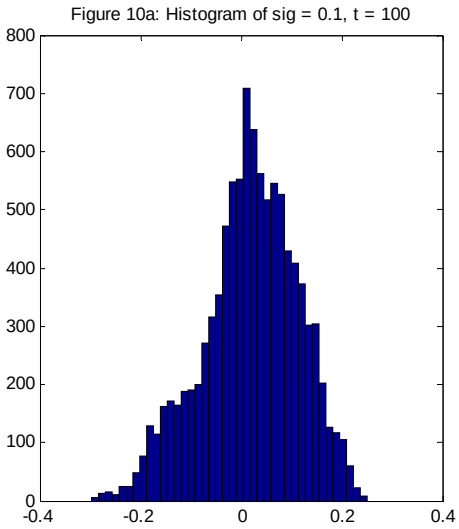
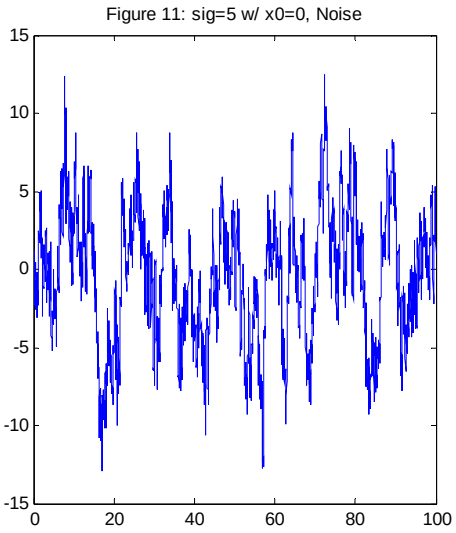
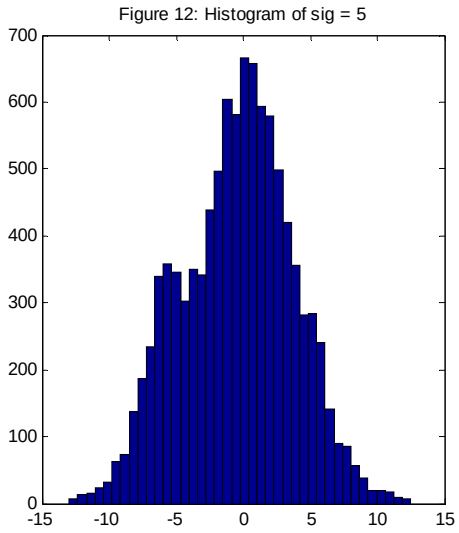


Figure 9 shows just the pure noise when sigma is set to 0.1 and the histogram is shown in Figure 10. The mean value of the noise was -0.0119 with standard deviation of 0.0665. With the histogram in Figure 10, it is much easier to see that the mean is still centered around zero with standard deviation of approximately 0.1. Increasing the sampling time in figures 9a and 10a, the mean remains close to zero (0.0191), but the standard deviation gets closer to 0.1 (0.0947). This is yet another method which is used in reducing noise in practical applications such as image processing. The noise is measured for a long period of time to generate the mean level and the standard deviation. This value is then subtracted from the signal value to recover the original level of the signal.

When we increase sigma to 5, we obtain the results shown in figures 11 and 12.





The mean value for sigma = 5 was -0.4445 and the standard deviation was 4.1589. Again, these results are expected but once again show the unpredictability of a stochastic differential equation. The differences in width of Figure 10 and 12 are related to the difference in standard deviation and the sigma value that we used. Running the noise for an extended period of time will result in the normal distribution of values with the given standard deviation used. Since we used 5 for sigma in Figure 12 and only 0.1 for Figure 10, we expect the width of Figure 12 to be much larger.

Question 3: Monte Carlo Algorithms

Part 1

```
1     clear;
2     i=0;
3     for N=[100 1000 5e3 1e4 5e4 1e5 5e5];
4         i=i+1;
5         pp=rand(N,2);
6         P=4*mean((pp(:,1).^2+pp(:,2).^2)<=1);
7         A(i,:)=[N P];
8     end
```

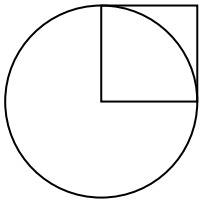
This program generates N number of random points (line 5) and implements a test in line 6. The average value of all the numbers is saved as the variable P and the array A stores all the values of P for given values of N (as defined in line 3). Using the `rand` function generates random numbers from 0 to 1 selected from a uniform distribution so that all values are equally likely to show up. On the other hand, if the function had used `randn` to generate numbers, this selects values from a normal distribution with mean 0 and standard deviation of 1.

The test that the program implements is taking the sum of the squares of the two numbers generated and seeing if it is less than one. This program is equivalent to making two columns, labelling one x and the other y, and seeing if each row satisfies the equation $x^2 + y^2 \leq 1$. If they do, that point is assigned the value 1 and if the condition is not satisfied, then the point is

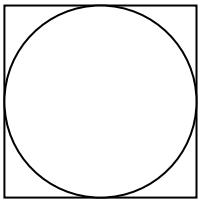
assigned a value 0. Line 6 of the program takes the average of all the 1's and 0's and multiplies it by 4.

If we look closely at the equation that is being used to test the two points, we notice that it takes the form of a circle. Indeed, this program is equivalent to generating N number of points and seeing whether or not they fall inside a circle of unit radius or not. As the value of N increases, coverage of this circle increases. However, for the purposes of Matlab, we test only ¼ of the circle since we cannot generate negative values using rand. To correct for this, we just multiply by 4 (line 6). In the limit, as N approaches infinity, then the average taken in line 6 approaches the percentage of the quarter of the circle covering a unit square. The fraction covered, ie the probability that it hits inside the circle (and also the area of the sector since we are dealing with a circle of unit radius and a unit square) is $\pi/4$. Multiplying this by 4 gives the area of the entire circle, π, the limit that P approaches.

The diagram below represents what is happening in the code and what it translates to in real terms.



Takes ¼ of the circle first because of the limitations on rand



Multiplying by 4 translates to a full circle

Part 2

```
clear; Q=0; Nfinal=1e6;
for N=1:Nfinal;
    pp=rand(1,2);
    Q=Q+(1./(1+pp(1,1).^2)>=pp(1,2));
end
P1=Q/Nfinal
```

To find the limiting value of P1 in the program above, we can use a similar method to what was used in Part 1. If we look closely at the program, we can see that we are trying to find the area under the curve of the equation $y = \frac{1}{1+x^2}$ in the interval [0,1] by again "throwing darts" at the plot and finding the mean of the values of 0s and 1s. Actual integration of the equation will yield the limiting value of arctan(1) which is approximately 0.7854. The value of P1 generated by the function above is 0.7851, which approximates the limiting value very well.

```
clear; Q=0; Nfinal=1e6;
for N=1:Nfinal;
    pp=rand(1,2);
    Q=Q+(pp(1,1)./(1+pp(1,1).^2)>=pp(1,2));
end
P2=Q/Nfinal
```

In the program above, we are looking at integrating the equation $y = \frac{x}{1+x^2}$ in the interval [0,1]. The limiting value when integrating the equation is $0.5\ln 2$, which is approximately 0.3466. The value of P2 generated by the function above is 0.3469, which also approximates the limiting value very well.

However, there are some limitations to integration by using a Monte Carlo algorithm. First, the method coded above only allows for functions that are defined within the unit box from 0 to 1. Otherwise, all the points in the box would satisfy the condition and the final value will be 1. We can correct for this by increasing our box to a suitable size, say 2 x 2 if the $f(0)$ = 1.5, or we can scale down our function to fit within the range, just remembering to rescale it when we report the value of the integration. If we change the range, we must also multiply our final value by the area of our box, since what we are effectively doing in the algorithm is sampling the entire box and calculating the percentage of the box which satisfies the conditions set about in the function.

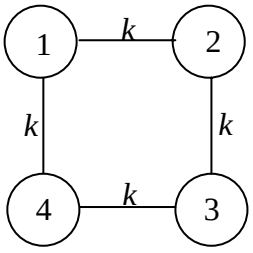
Theoretically, this integration method can be extended to n-dimensions, so long as one keeps track of the exact calculation that is being done, or else it is easy to misinterpret the results. Furthermore, the other limitation is that initial conditions have to be known in order to be able to set an appropriate area/volume through which to integrate. Setting the range too small will give an erroneous result, but setting the range too large could be costly in terms of computational time.

Question 4: Normal modes and systems of coupled harmonic (linear) oscillators

Consider the set of differential equations:

$$\begin{aligned}
\dot{x}_1 &= -k(x_1 - x_2) - k(x_1 - x_4) \\
\dot{x}_2 &= -k(x_2 - x_3) - k(x_2 - x_1) \\
\dot{x}_3 &= -k(x_3 - x_4) - k(x_3 - x_2) \\
\dot{x}_4 &= -k(x_4 - x_3) - k(x_4 - x_1), k = 1
\end{aligned}$$

Upon careful inspection of each, we can determine that these equations represent a system of 4 masses attached to each other by springs of equal spring constant (k = 1). Furthermore, we can deduce that the motion of mass 1 (as governed by the first equation) is connected to masses 2 and 4 and the velocity is related to the length of the "bond" separating them. Going through each equation gives a clearer picture of the system we are trying to model as shown below.



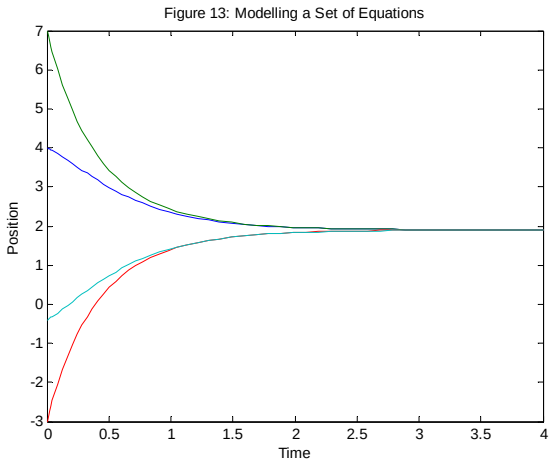
With the help of Matlab, we can easily integrate our set of differential equations to come up with a numerical solution. We first define a function `mydysys` to store the equations.

```
function dx = mydysys(t,x)
dx = zeros(4,1);
dx(1) = -(x(1)-x(2)) - (x(1)-x(4));
dx(2) = -(x(2)-x(3)) - (x(2)-x(1));
dx(3) = -(x(3)-x(4)) - (x(3)-x(2));
dx(4) = -(x(4)-x(3)) - (x(4)-x(1));
```

And is called by the main program with the given initial conditions:

```
iniconds = [4 7 -3 -0.4];
time = [0 4];
[T,X] = ode45(@mydysys,time,iniconds);

figure;
plot(T,X)
title('Figure 13: Modelling a Set of Equations')
```

This results in the plot shown in Figure 13.



In the plot, we can see that although the four particles begin at different positions, they all end up in the same position over time. In fact, the sum of the all four coordinates remain the same over time, suggesting that the center of mass remains constant over time. With the given initial conditions, the masses are moved away from their equilibrium value and released. After a given time, they return to their square shape having a fixed length between them.

To get a better idea of what is happening, we can look at the eigenvalues and eigenvectors of the set of equations. We first transform our set of equations into the form $\dot{x} = Ax$.

$$\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{pmatrix} = k \begin{pmatrix} -2 & 1 & 0 & 1 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 1 & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}$$

Matlab then allows an easy method of getting the eigenvalues and eigenvectors of our 4th dimensional problem.

```
A = [-2 1 0 1;1 -2 1 0;0 1 -2 1;1 0 1 -2];
[V,D] = eig(A)
```

The eigenvectors:

```
V =

    0.5000   -0.0000    0.7071   -0.5000
   -0.5000    0.7071    0.0000   -0.5000
    0.5000    0.0000   -0.7071   -0.5000
   -0.5000   -0.7071         0   -0.5000
```

The corresponding eigenvalues in a diagonalized matrix:

```
D =

   -4.0000         0         0         0
         0   -2.0000         0         0
         0         0   -2.0000         0
         0         0         0    0.0000
```

As is seen, the maximum eigenvalue is 0 and the corresponding eigenvector is [-0.5, -0.5, -0.5, -0.5].

If we look more to the general solution derived from solving the system of differential equations, we may be able to gain a greater insight as to what the eigenvalues and eigenvectors mean in terms of the system. Given our eigenvalues, λ, and our eigenvectors, η, the general solution to a linear system of differential equations is displayed below.

$$\vec{x}(t) = \sum_{i=1}^{n} \vec{\eta}_i c_i e^{\lambda_i t}$$

For our system, this results in the equation below, with the values of *c* being constants depending on the initial values of the system.

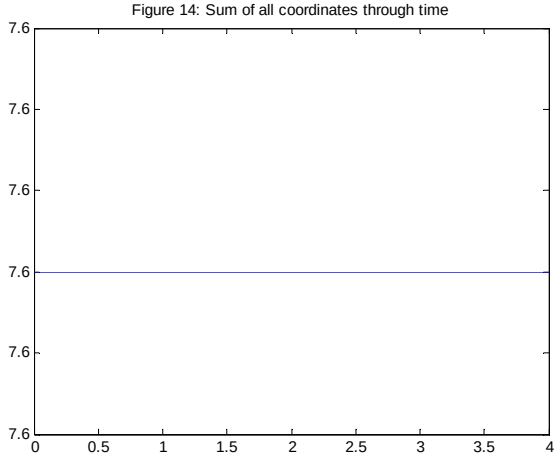
$$\vec{x}(t) = \begin{pmatrix} -0.5 \\ -0.5 \\ -0.5 \\ -0.5 \end{pmatrix} c_1 + \begin{pmatrix} 0.71 \\ 0 \\ -0.71 \\ 0 \end{pmatrix} c_2 e^{-2t} + \begin{pmatrix} 0 \\ 0.71 \\ 0 \\ -0.71 \end{pmatrix} c_3 e^{-2t} + \begin{pmatrix} 0.5 \\ -0.5 \\ 0.5 \\ -0.5 \end{pmatrix} c_4 e^{-4t}$$

We can see that as time approaches infinity, we are left with the eigenvector corresponding to the eigenvalue of 0. What that equation tells us is that the position vector for each is constant and in the same direction, so all the masses in our system are stable. If we consider other values of t, then we realize that they die off, resulting in the same position given by the eigenvector corresponding to the eigenvalue of 0.

We also must remember that we are in the overdamped limit, which will help us to understand the other modes corresponding to the other eigenvalues, -2 and -4. For one of the -2 eigenvalues, we see that this corresponds to perturbing masses 1 and 3 or 2 and 4 at the same time. The bond between all the masses will correct for this perturbation and the system returns to the position defined by the 0 eigenvalue. In effect, we are only stretching two

bonds out of 4, so this would theoretically require half as much energy to excite, hence half the eigenvalue. The -4 eigenvector corresponds to perturbing 1 and 2 in opposite directions as well as 3 and 4 in opposite directions, so this represents the stretching of all the bonds, requiring the most amount of energy.

Figure 14: Sum of all coordinates through time



So we can see, that in these types of systems, the eigenvalues correspond to something similar to vibrational modes of the system where 0 represents no energy input required and increasingly negative values requiring the need for more energy to excite that particular mode.

In figure 14, we can see that if we sum all the coordinates in time, they remain constant (7.6), keeping in line with the fact that the center of mass remains constant at all times.

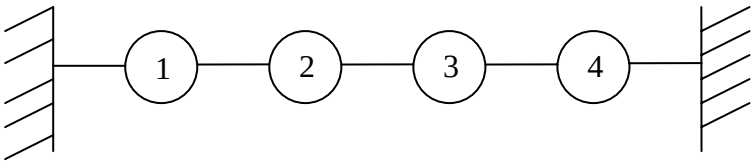
To have perhaps a better understanding of the system, we can imagine our coordinate system in terms of the square upon which our masses lie. The position of the masses are determined by the initial conditions and the eigenvectors will determine the modes of the system. Moving in the negative direction would refer to moving counter-clockwise along the square coordinate system while moving in the positive direction would refer to moving clockwise along the coordinate system. With the eigenvector relating to eigenvalue of 0, all of the masses are moving in the same direction, and thus the entire square is rotating (the translational energy level). With the other eigenvalues and eigenvectors, we are perturbing the system and causing extension or compression of the bonds/springs, which requires more energy, and thus has a lower eigenvalue than our maximum eigenvalue of 0.

Part 2

We consider a new set of differential equations as shown below.

$$\dot{x}_1 = -k(x_1 - x_2) - kx_1$$
$$\dot{x}_2 = -k(x_2 - x_3) - k(x_2 - x_1)$$
$$\dot{x}_3 = -k(x_3 - x_4) - k(x_3 - x_2)$$
$$\dot{x}_4 = -k(x_4 - x_3) - kx_4, k = 1$$

Now, we can see that the first and fourth mass is connected to both another mass and a fixed wall, as opposed to two masses in the first problem. A diagram of the physical system that we are trying to model is shown below.



We can again obtain the eigenvalues and eigenvectors of the system of differential equations in Matlab.

First, the transformed system in matrix notation:

$$\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{pmatrix} = k \begin{pmatrix} -2 & 1 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}$$

```
B = [-2 1 0 0;1 -2 1 0;0 1 -2 1;0 0 1 -2];
[V1 D1] = eig(B)
```

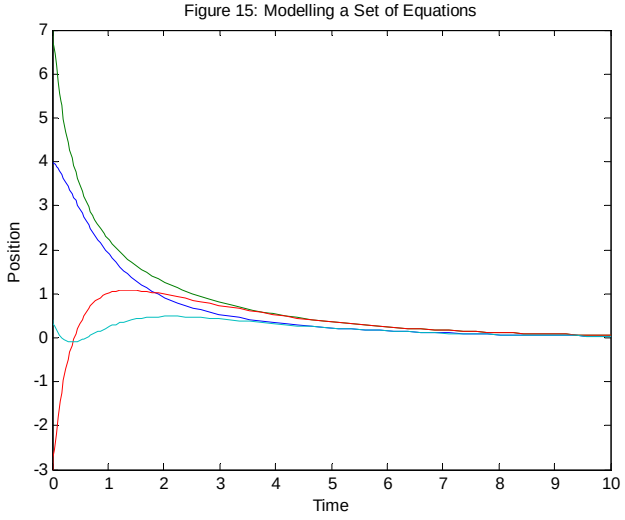
The eigenvalues are:

```
D1 =

   -3.6180         0         0         0
         0   -2.6180         0         0
         0         0   -1.3820         0
         0         0         0   -0.3820
```

And the corresponding eigenvectors are:

```
V1 =

    0.3717   -0.6015   -0.6015   -0.3717
   -0.6015    0.3717   -0.3717   -0.6015
    0.6015    0.3717    0.3717   -0.6015
   -0.3717   -0.6015    0.6015   -0.3717
```

Figure 15: Modelling a Set of Equations



Here, the maximum eigenvalue is not 0, but is -0.3820. Since the maximum eigenvalue is negative, we can assume that any perturbation in the system will result in the system dying down to zero. And indeed, as shown in figure 15, the center of mass returns to zero, given the initial conditions set in the previous example.

Since the entire system is fixed to a wall, it makes sense that the solution of the differential equations will result in the system returning to its equilibrium state. For our first example, the center of mass could be translated and the system was not fixed, but for here, the system cannot be moved and when perturbed, its motion will return it back to the equilibrium position.

If we were to repeat the calculation of summing all of the coordinates, the result will not be the same. The system will begin at a certain point (the sum of the initial conditions), and will exponentially decay to zero since the system is fixed and cannot move

Question 5: A Second Order System

We consider the differential equation below.

$$\frac{d^2y}{dt^2} + \eta\frac{dy}{dt} + y = 0$$

We can write this as a system of linear first order differential equations by using the substitution, $x_2 = \frac{dy}{dt}$. By definition, $\dot{x}_2 = \frac{d^2y}{dt^2}$, and if we let $x_1 = y$, this results in the system as shown below.

$$\dot{x}_1 = x_2$$
$$\dot{x}_2 = \frac{d^2y}{dt^2} = -\eta\frac{dy}{dt} - y = -\eta x_2 - x_1$$

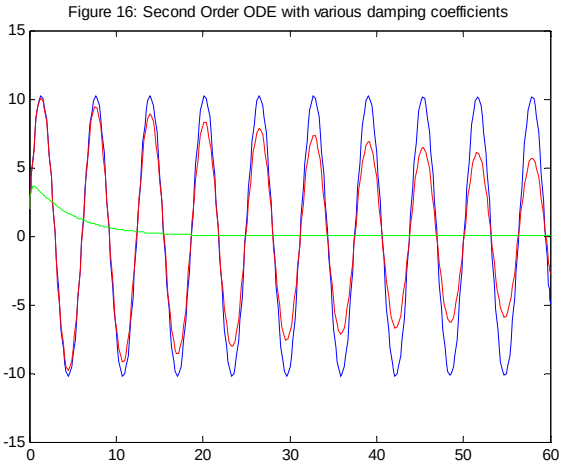
We can put this into Matlab as a function as shown below.

```
function dx = seode(t,x)
n = 0;
dx = zeros(2,1);
dx(1) = x(2);
dx(2) = -n*x(2)-x(1);
```

And using ode45 to numerically solve the system of equations for three different values of η (0, 0.02, and 5)

```
t = [0 60];
iniconds = [2 10];
[T,Y] = ode45(@seode, t, iniconds);
figure;
plot(T,Y(:,1))
hold on;
[T1,Y1] = ode45(@seode02, t, iniconds);
plot(T1,Y1(:,1),'red')
[T2,Y2] = ode45(@seode5, t, iniconds);
plot(T2,Y2(:,1),'green')
title('Second Order ODE with various damping coefficients')
hold off;
```

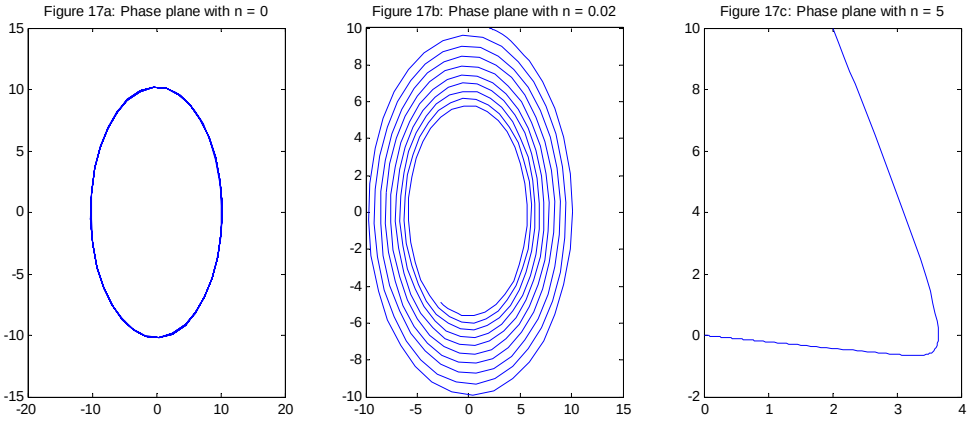
Results in the following plot (figure 16)



With $\eta = 0$, then the solution is purely harmonic with no damping. With η = 0.02, you begin to see some damping, but as the constant is still small, damping remains minimal. With η = 5, we are in the overdamped regime and no oscillations occur.

To get a better idea of the trajectories of the three cases, we can explore the results on a phase plane by plotting the differential equation itself without having to integrate.

```
figure;
subplot(1,3,1), plot(Y(:,1),Y(:,2))
title('Phase plane with n = 0')
subplot(1,3,2), plot(Y1(:,1), Y1(:,2))
title('Phase plane with n = 0.02')
subplot(1,3,3), plot(Y2(:,1), Y2(:,2))
title('Phase plane with n = 5')
```



In figure 17a, we see that if η = 0, then we get a circle (closed line) meaning that the solution is periodic. With η not equal to zero, then we don't get a closed line solution, and as is seen, the solution is no longer periodic and will decay to zero. We can analyze this system again using eigenvalues and eigvectors as we did in question 4, and we might gain some more insight into the mechanics of the problem.

We can write the system of equations in the form $\dot{x} = Ax$ as shown below.

$$\vec{\dot{x}} = \begin{pmatrix} 0 & 1 \\ -1 & -\eta \end{pmatrix} \vec{x}$$

As the eigenvalues and eigenvectors depend on the value of η, we can first test by setting it equal to zero.

The eigenvalues are:

```
D =

         0 + 1.0000i         0
         0                   0 - 1.0000i
```

And the corresponding eigenvectors are:

```
V =

    0.7071              0.7071
         0 + 0.7071i         0 - 0.7071i
```

The eigenvalues are purely imaginary and thus, the analytical solution to the system of equations is a combination of imaginary exponentials (or a combination of sine waves). If we change the value of η to be 0.02, the underdamped case, then we get the following eigenvalues and eigenvectors.

Eigenvalues:

```
D =

   -0.0100 + 0.9999i         0
         0             -0.0100 - 0.9999i
```

And the corresponding eigenvectors:

```
V =

    0.7071              0.7071
   -0.0071 + 0.7071i   -0.0071 - 0.7071i
```

Here, we are no long dealing with purely imaginary eigenvalues, and so our solution is a combination of both decay and sinusoidal terms, resulting in the oscillations seen in figure 16. In the overdamped case where η = 5, let's see what happens to the eigenvalues. Since we are not expecting any oscillations, our eigenvalues should be non-imaginary.

Eigenvalues:

```
D =

    -0.2087          0
          0    -4.7913
```

And the corresponding eigenvectors:

```
V =

     0.9789    -0.2043
    -0.2043     0.9789
```

So, the hypothesis was correct in that in the overdamped case, the imaginary term disappears and we are left with exponentials that decay to zero without any oscillations. Theoretically, the switch over value of η from underdamped to overdamped is when ζ = 1 (ie the critically damped case). If we recall the general formula for harmonic oscillations and the definition of ζ, we can derive at which value of η the critically damped case will occur.

$$m\ddot{x} + c\dot{x} + kx = 0$$

$$\zeta = \frac{c}{2\sqrt{km}}$$

In our case, both m and k are 1 and the c is equal to η. This leads to the relation $\zeta = \frac{\eta}{2}$. Thus, for critical damping to occur (ζ = 1), then η must equal 2. Let us check to see if our solution is correct by considering the eigenvalues and eigenvectors of this system.

First the eigenvalues:

```
D =

    -1     0
     0    -1
```

And the corresponding eigenvectors:

```
V =

    0.7071   -0.7071
   -0.7071    0.7071
```

Indeed, we see that we no longer get an imaginary eigenvalue meaning that there are no oscillations. To check to see if η = 2 really is the critical damping point, we can go back to first principals of finding eigenvalues. If we find the determinant of $A - \lambda(I)$, where $A = \begin{pmatrix} 0 & 1 \\ -1 & -\eta \end{pmatrix}$ and set it equal to zero (step to find the eigenvalues), then we are left with the equation $\lambda^2 + \eta\lambda + 1 = 0$ whose solutions are $\frac{-\eta \pm \sqrt{\eta^2 - 4}}{2}$. From here, it is easy to see that η = 2 is the point at which the solutions, and thus the eigenvectors, will turn from being non-imaginary to having an imaginary component. Thus, this is the value at which one expects the system to switch from one behavior to the other. Figure 18 shows the result of critical damping and figure 19 displays the corresponding phase plane.

Figure 18: Critical Damping

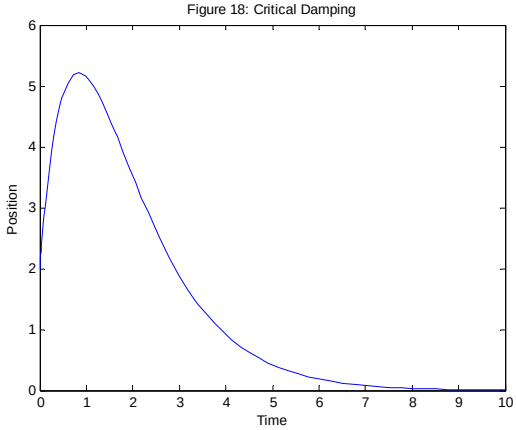
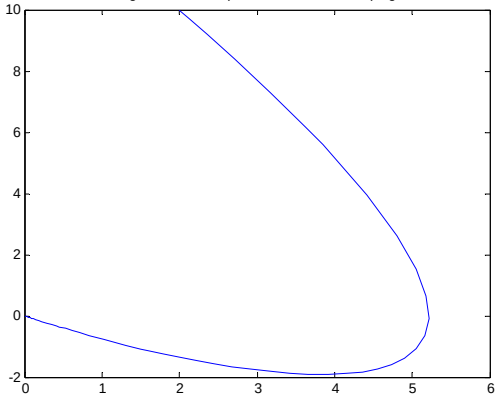


Figure 19: Phase plane with Critical Damping



```
%Question 2 - An algorithmic implementation of the Euler Method
%euler.m
%John Sy

clc; close all; clear all;

%Part 1a

k = (2/3);
h = 0.01; %time step
myeuler(1) = 10;
timesteps1 = 5/h;

for t = [2:1:timesteps1+1]
    myeuler(t) = myeuler(t-1) - h*k*myeuler(t-1);
end

timevect1 = [0:h:5];
figure;
plot(timevect1,myeuler)
title('Figure 5: h = 0.01 implementation of the Euler algorithm')

anasoln1 = myeuler(1)*exp(-(2/3)*timevect1);
mse21a = mse(anasoln1 - myeuler)

%Part 1b

k = (2/3);
h2 = 0.001; %time step
myeuler1(1) = 10;
timesteps2 = 5/h2;

for t = [2:1:timesteps2+1]
    myeuler1(t) = myeuler1(t-1) - h2*k*myeuler1(t-1);
end

timevect2 = [0 : h2 : 5];
figure;
plot(timevect2,myeuler1)
title('Figure 6: h = 0.001 implementation of the Euler algorithm')

anasoln2 = myeuler1(1)*exp(-(2/3)*timevect2);
mse21b = mse(anasoln2 - myeuler1)
```

```
%Assignemnt 1 Question 2 Part 2
%eulerSDE.m
%John Sy

%Part 2

clc; close all;

k = (2/3);
h3 = 0.01;
mystoeul(1) = 10;
sig = 0.2;
timesteps3 = 5/h3;

for t = [2:1:timesteps3+1];
    mystoeul(t) = mystoeul(t-1) - h3*k*mystoeul(t-1) + sig*sqrt(h3)*randn;
end

timevect3 = [0:h3:5];
figure;
plot(timevect3,mystoeul)
title('Figure 7: h = 0.01 Stochastic Implementation of Euler')

%Part 2b

storevals = zeros(15,5/h3); %create array
mystoeul1(1) = 10;
storevals(:,1) = mystoeul1(1);

figure;
for n = 1:1:15
    for t = [2:1:timesteps3+1];
        mystoeul1(t) = mystoeul1(t-1) - h3*k*mystoeul1(t-1) + sig*sqrt(h3)*randn;
        storevals(n,t) = mystoeul1(t); %store values into array
    end
    hold on;
    plot(timevect3,storevals(n,:))
end
title('Figure 8: h = 0.01 15 Repetitions of SDE with Average')

for i = 1:1:timesteps3+1
    meanvals(i) = mean(storevals(:,i)); %averages values
end

plot(timevect3,meanvals,'red','LineWidth',2)
hold off;

%Part 2c - for time values greater than 5

k = (2/3);
step = 0.01;
mysto(1) = 0;
sig = 0.1;
timeval = 100;
tstep = timeval/step;
```

```
for t = [2:1:tstep+1];
    mysto(t) = mysto(t-1) - step*k*mysto(t-1) + sig*sqrt(step)*randn;
end

tvect = [0:step:timeval];
figure;
subplot(1,2,1), plot(tvect,mysto)
title('Figure 9a: sig=0.1 w/ x0=0, t=100, Noise')
subplot(1,2,2), hist(mysto,40)
title('Figure 10a: Histogram of sig = 0.1, t = 100')

mean02 = mean(mysto)
std02 = std(mysto)

sig2 = 5;
mysto1(1) = 0;
for t = [2:1:tstep+1];
    mysto1(t) = mysto1(t-1) - step*k*mysto1(t-1) + sig2*sqrt(step)*randn;
end

tvect = [0:step:timeval];
figure;
subplot(1,2,1), plot(tvect,mysto1)
title('Figure 11: sig=5 w/ x0=0, Noise')
subplot(1,2,2), hist(mysto1,40)
title('Figure 12: Histogram of sig = 5')

mean5 = mean(mysto1)
std5 = std(mysto1)
```