ENM 502 : NUMERICAL METHODS AND MODELLING

ASSIGNMENT #4

Submitted on 17th April 2014 by :

RAMALINGAM KAILASHAM

kailr@seas.upenn.edu

Problem statement

Given the following system of non-linear ODEs,

$$\frac{dx}{dt} = (a - by)x - px^2 \quad (1)$$

$$\frac{dy}{dt} = (cx - d)y - px^2 \quad (2)$$

Where a,b,c,d,p,q are constants greater than zero.

a) The stationary/critical points of this system were found using the following commands in MATLAB. These lines were typed in the command-line window.

*[Sx, Sy] = solve ((a-b*y)*x - p*x^2==0,(c*x-d)*y -q*y^2 ==0,x,y)*

Sx =	*Sy =*
0	*0*
0	*a/p*
-d/q	*0*
*(a*c - d*p)/(b*c + p*q)*	*(b*d + a*q)/(b*c + p*q)*

We see immediately that the critical point (-d/q, 0) is not physically realistic as the population of prey cannot be a negative number.

b) Non-Linear ODE solution schemes were set up using both Explicit and Implicit Euler methods. An additional routine, TestODE.m was coded up using explicit second order Adams-Bashforth method. The errors in both the implicit and explicit Euler methods were estimated by comparing against this higher order method.

The codes have been attached as a part of Appendix A.

Using the parameter values as a=b=c=d=1 and p=q=0, the physically realistic critical points are (0,0) and (1,1). It is shown later in this paper that (0,0 is physically unstable and (1,1) is physically stable.

Choice of appropriate step-size "h" :

The eigen-values at (1,1) are purely imaginary and this implies that the phase plot of x-v/s-y has to be in the form of closed loops. (Hartman-Grobman Theorem). This also means that the explicit method is unconditionally stable for this problem. Strictly, it is neutrally stable because $Re(\lambda_i) = 0$. As discussed in class, the implicit method is unconditionally stable for a physically stable system.

At big values of h, say of the order of 10^{-1} or 10^{-2}, the phase plot showed decaying spirals. Since this does not agree with the expected result, the value of h was systematically reduced till a closed loop was observed. This was found to occur at a step size of 10^{-4}. The step size used in my program is 10^{-5}, to err on the safer side.

The Euler implicit method is accurate upto $O(h^2)$. Therefore, the tolerance of Newton's method has to be at least $O(h^2)$ so that the solution converges properly. The tolerance is varied according to step-size in the program.

As explained before, the error at an arbitrarily chosen point was estimated by comparing the solution with that obtained from the higher order method. The error was normalized by the number of points.

Figures 1 to 4 show the variance of error with h for x and y for both the solution scheme. Within scientific error limits, the graphs agree with the theoretical prediction of $O(h^2)$ scaling.

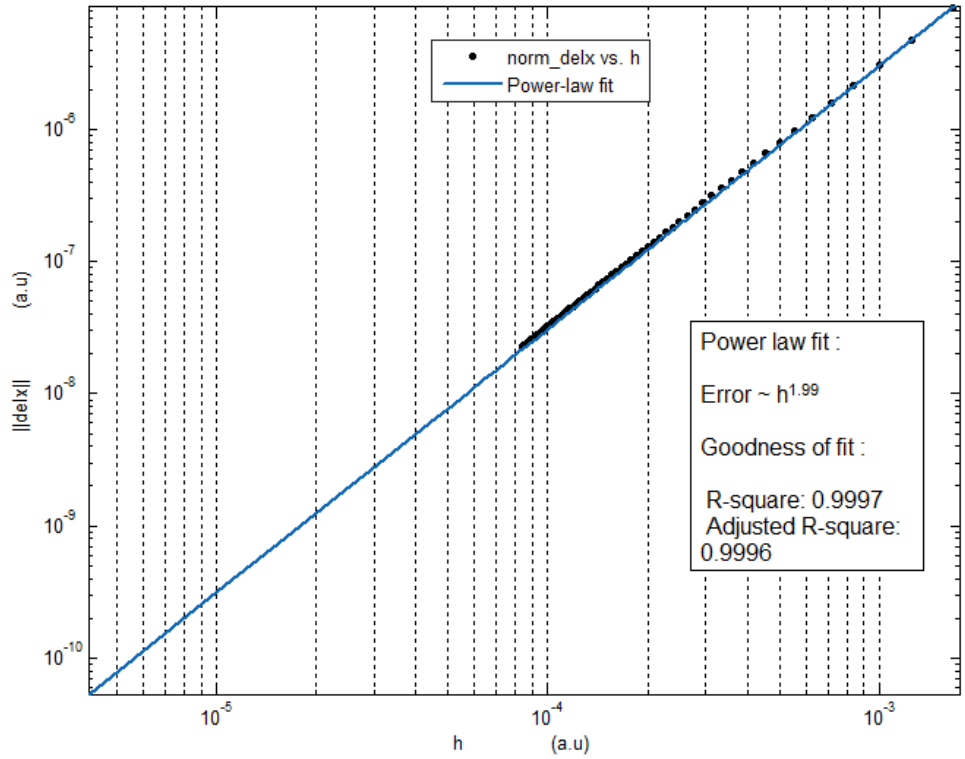

Figure 1 : Plot of error v/s step size for x calculated using Explicit Euler method

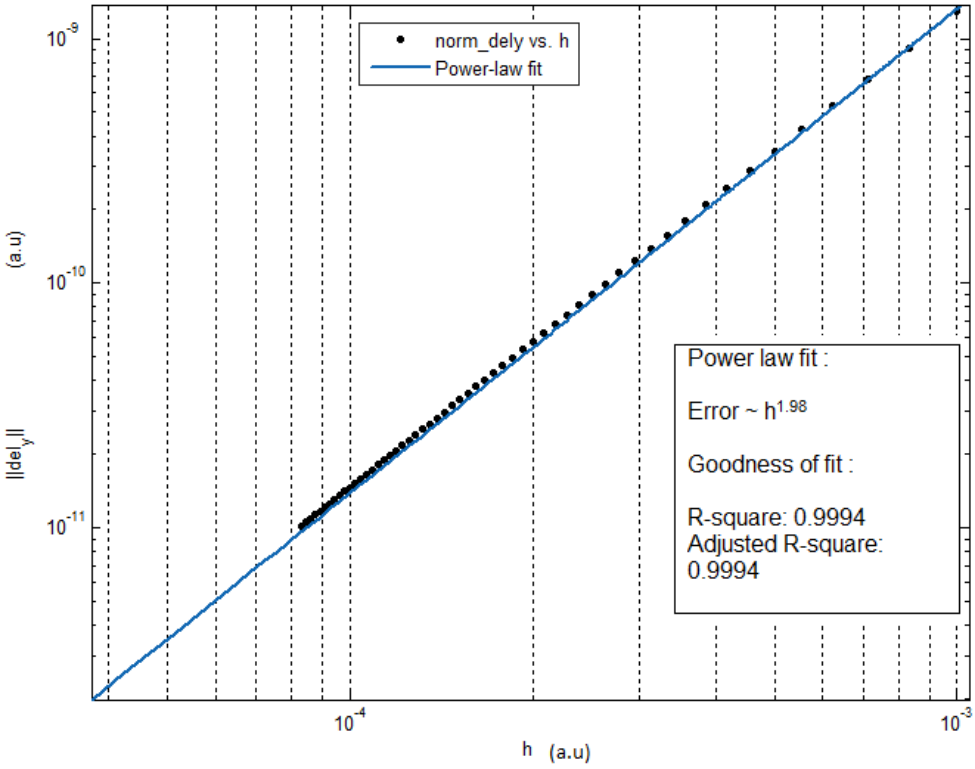

Figure 2 : Plot of error v/s step size for y calculated using Explicit Euler method

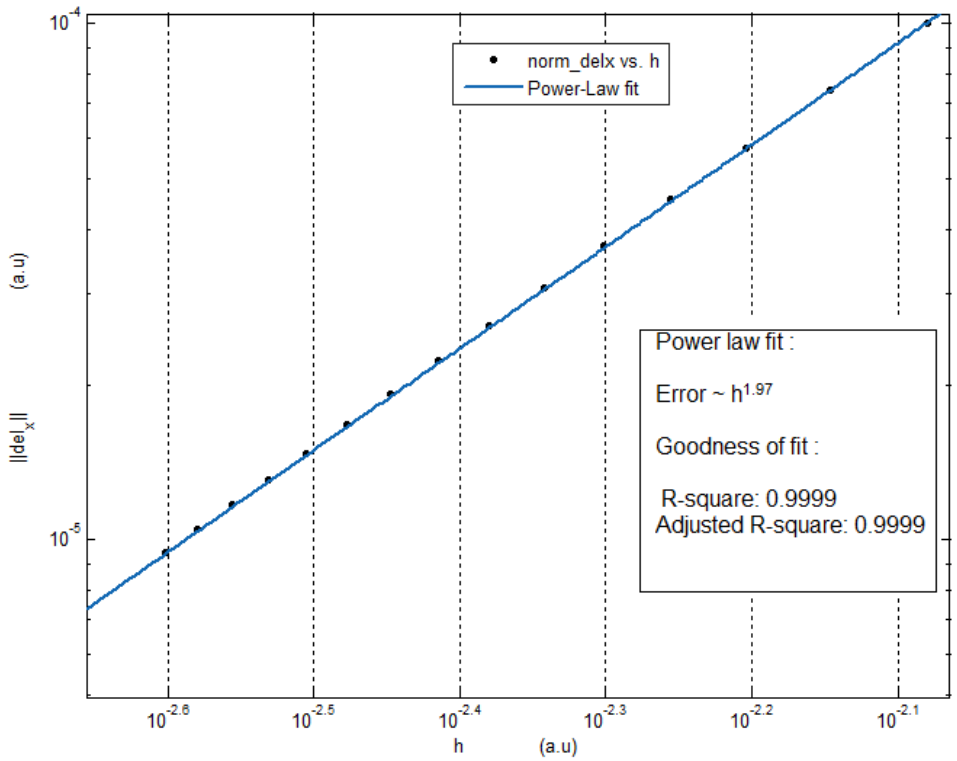

Figure 3 : Plot of error v/s step size for x calculated using Implicit Euler method

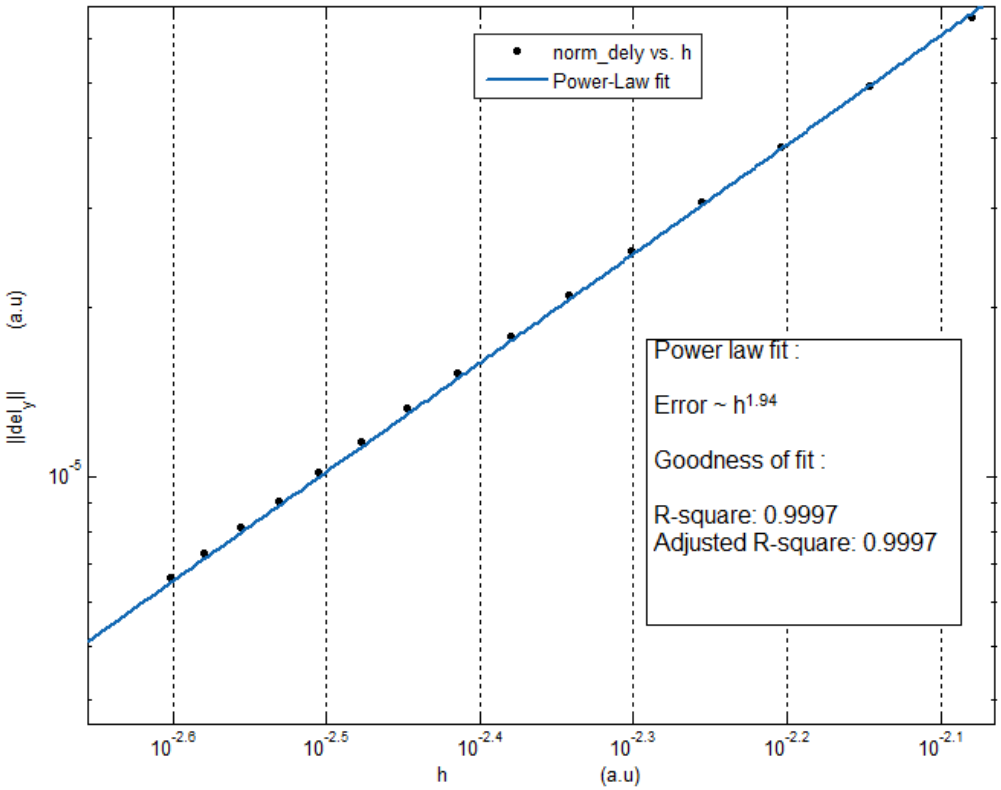

Figure 4 : Plot of error v/s step size for y calculated using Implicit Euler method

It was rigorously tested to confirm that the solutions obtained for x and y by both the Explicit and Implicit methods were the same. Given below is a plot of x and y versus time for the initial condition of $x_0 = 5$, $y_0 = 5$. Similar plots for different initial conditions have been attached as a part of Appendix B.

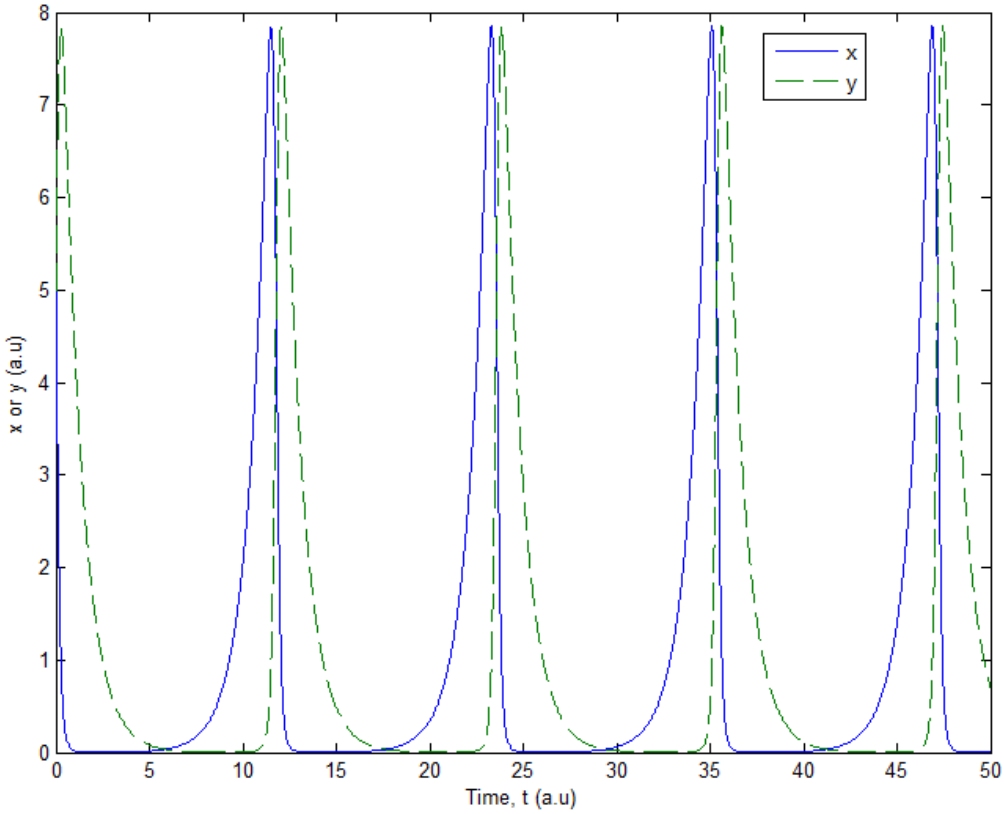

Figure 5 :Plot showing the variation of x and y as a function of time for the initial condition (5,5)

Note : There are about 10^5 data points in both x(t) and y(t), these have been plotted as lines for better clarity.

It is seen that the population of both the prey and predator oscillate with time. This is to be expected, as in the absence of competition and overcrowding effects (p=q=0), x and y pass very near to 0 but never die out.

Physically, this means that the predators thrive when there is plenty of prey but soon consume all their food and die out. As the predator population is low, the prey will grow again. This cycle of growth and death continues.

As (1,1) is the stable stationary point for the system, if we use it as the initial condition, the system doesn't evolve in time. Mathematically, the eigen-values at this point are purely imaginary and cannot contribute to evolution in time.

c) Phase Plots

The phase plots for the given system for different initial conditions have been plotted in a single graph.

Interpreting the graph: (7,5) means that the initial value for x is 7 and the initial value for y is 5. Same holds for the other loops.

Note: As there are about 10^5 data points in each loop, they can't be seen distinctly.

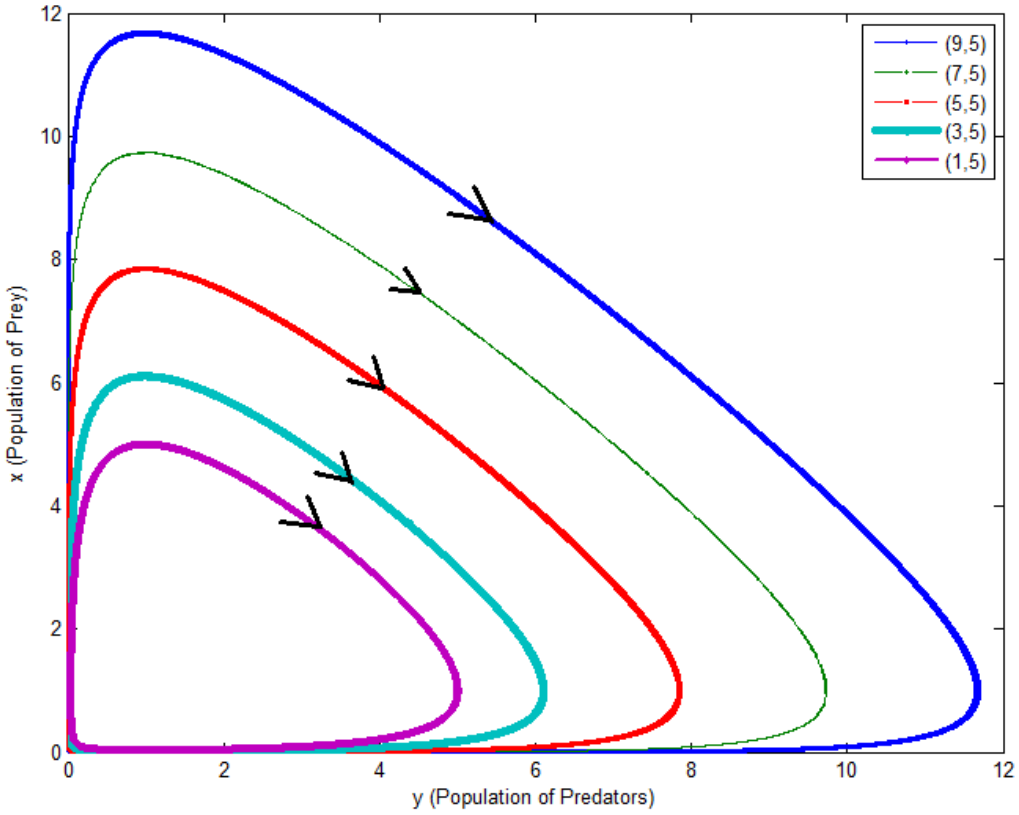

Figure 6 : Phase plot for different initial conditions

The arrows indicate the direction of positive time evolution for each loop. As the initial population of predators is kept constant and the initial prey population is increased, the system takes longer times to complete a full cycle.

Appendix A

ExpODE.m

```
%********************** INPUT PARAMETERS *******************************%
% n  - Number of points at which the solution needs to be evaluated     %
% x0 - Initial condition for x                                          %
% y0 - Initial condition for y                                          %
% T  - Timespan over which solutions need to be evaluated              %
%  a,b,c,d,p,q - Constants as defined in the problem statement.         %
%                                                                       %
%********************** OUTPUT VALUES **********************************%
%
% solx - The solution x(t) calculated using explicit Euler method      %
% soly - The solution y(t) calculated using explicit Euler method      %
% time - Points on the time axis at which solutions have been          %
%        calculated                                                     %
%                                                                       %
%***********************************************************************

function[solx,soly,time] = ExpODE(n,x0,y0,T,a,b,c,d,p,q)

solx(1) = x0;
soly(1) = y0;
time(1) = 0;
h=T/n;
for i=1:n
    fx(i) = (a-b*soly(i))*solx(i) - p*(solx(i)^2);
    fy(i) = (c*solx(i)-d)*soly(i) - q*(soly(i)^2);
    time(i+1) = i*h;
    solx(i+1) = solx(i) + h*fx(i);
    soly(i+1) = soly(i) + h*fy(i);
end
end
```

ImpODE.m

```
%********************** INPUT PARAMETERS *******************************%
% n  - Number of points at which the solution needs to be evaluated     %
% x0 - Initial condition for x                                          %
% y0 - Initial condition for y                                          %
% tspan  - Timespan over which solutions need to be evaluated          %
% a,b,c,d,p,q - Constants as defined in the problem statement.         %
%                                                                       %
%********************** OUTPUT VALUES **********************************%
%                                                                       %
% solx - The solution x(t) calculated using implicit Euler method      %
% soly - The solution y(t) calculated using implicit Euler method      %
% time - Points on the time axis at which solutions have been          %
%        calculated                                                     %
%                                                                       %
%***********************************************************************



function [solx,soly,t] = ImpODE(n,x0,y0,tspan,a,b,c,d,p,q)
```

```
syms x;
syms y;
dx_dt = (a-b*y)*x - p*x^2;
dy_dt = (c*x-d)*y - q*y^2;

solx(1) = x0;
soly(1) = y0;

h=tspan/n;
t(1) = 0;
tol = (h^2); % Tolerance for Newton's iteration set as h^2.
for i=1:n

    %Generating initial guesses using Predictor-corrector
    fx = func_eval(dx_dt,solx(i),soly(i));
    fy = func_eval(dy_dt,solx(i),soly(i));

    x_init = solx(i) + h*fx;
    y_init = soly(i) + h*fy;

    t(i+1) = i*h;
    [solx(i+1),soly(i+1)] =
newton(tol,n,x_init,y_init,solx(i),soly(i),h,a,b,c,d,p,q);

end
end
```

newton.m

```
%************************** INPUT PARAMETERS ***************************%
%                                                                       %
% tol    - Tolerance value. Depends on the step-size,h                  %
% x_init - Initial guess for x at the given point                       %
% y_init - Initial guess for y at the given point                       %
% x_prev - converged value of x at the previous point                   %
% y_prev - converged value of y at the previous point                   %
% h      - step size in time                                            %
% a,b,c,d,p,q - Constants as defined in the problem statement           %
%                                                                       %
%************************** OUTPUT VALUES ******************************%
%                                                                       %
% solx - converged value of x at the given point                        %
% soly - converged value of y at the given point                        %
%                                                                       %
%***********************************************************************%

function [solx,soly] = newton(tol,x_init,y_init,x_prev,y_prev,h,a,b,c,d,p,q)

norm_x=1;
norm_y = 1;

while(norm_x>tol && norm_y > tol)
[Jac_u,Resid_u] = Jac_Res(x_init,y_init,x_prev,y_prev,h, a,b,c,d,p,q);
del_u_calc = Jac_u\Resid_u;
```

```
del_x = (del_u_calc(1));
del_y = (del_u_calc(2));
norm_x = norm(del_x,2);
norm_y = norm(del_y,2);
x_init = x_init + del_x;
y_init = y_init + del_y;
end

solx = x_init;  %this x_init contains the converged value of x at point (n+1)
soly = y_init;  %this y_init contains the converged value of y at point (n+1)

end
```

Jac_Res.m

```
%************************* INPUT PARAMETERS ***************************%
%                                                                       %
% x_init - Initial guess for x at the given point                       %
% y_init - Initial guess for y at the given point                       %
% x_prev - converged value of x at the previous point                   %
% y_prev - converged value of y at the previous point                   %
% h      - step size in time                                            %
% a,b,c,d,p,q - Constants as defined in the problem statement           %
%                                                                       %
%************************* OUTPUT VALUES ******************************%
%                                                                       %
% J -    Jacobian matrix calculated at (x_init,y_init)                  %
% R -    Residual vector calculated at (x_init,y_init)                  %
%                                                                       %
%***********************************************************************%

function[J,R] = Jac_Res(x_init,y_init,x_prev,y_prev,h,a,b,c,d,p,q)

R1 = x_init - x_prev - h*((a-b*y_init)*x_init - p*x_init^2); %eqn for dx_dt
R2 = y_init - y_prev -h*((c*x_init-d)*y_init - q*y_init^2);  %eqn for dy_dt

J1x = 1 - h* ( (a-b*y_init) - 2*p*x_init);
J1y = -h*(-b*x_init);

J2x = -h*(c*y_init);
J2y = 1 - h*( (c*x_init-d) - 2*q*y_init);

J(1,1) = J1x;
J(1,2) = J1y;
J(2,1) = J2x;
J(2,2) = J2y;

R = -1*cat(1,R1,R2);

end
```

TestODE.m

```
%********************** INPUT PARAMETERS *******************************%
% n  - Number of points at which the solution needs to be evaluated     %
% x0 - Initial condition for x                                          %
% y0 - Initial condition for y                                          %
% T  - Timespan over which solutions need to be evaluated              %
% a,b,c,d,p,q - Constants as defined in the problem statement.         %
%                                                                       %
%********************** OUTPUT VALUES **********************************%
%                                                                       %
% normx - Absolute value of the difference between x(at a point)       %
%         calculated using the two methods, normalised by the          %
%         number of points                                             %
% normy - Absolute value of the difference between y(at a point)       %
%         calculated using the two methods, normalised by the          %
%         number of points                                             %
%***********************************************************************

function[normx,normy] = TestODE(n,x0,y0,T,a,b,c,d,p,q)

solx = zeros(n+1,1);
soly = zeros(n+1,1);
time = zeros(n+1,1);

solx(1) = x0;
soly(1) = y0;
time(1) = 0;
h=T/n;

fx(1) = (a-b*soly(1))*solx(1) - p*(solx(1)^2);
fy(1) = (c*solx(1)-d)*soly(1) - q*(soly(1)^2);

%using Euler's method to calculate the solution at the first step
solx(2) = solx(1) + h*fx(1);
soly(2) = soly(1) + h*fy(1);

fx(2) = (a-b*soly(2))*solx(2) - p*(solx(2)^2);
fy(2) = (c*solx(2)-d)*soly(2) - q*(soly(2)^2);


solx(3) = solx(2) + (h/2)*(3*fx(2)-fx(1));
soly(3) = soly(2) + (h/2)*(3*fy(2)-fy(1));

for i=3:n
    fx(i) = (a-b*soly(i))*solx(i) - p*(solx(i)^2);
    fy(i) = (c*solx(i)-d)*soly(i) - q*(soly(i)^2);
    time(i+1) = i*h;
    %using Explicit 2nd order Adams-Bashforth for error analysis.
    solx(i+1) = solx(i) + (h/2)*(3*fx(i)-fx(i-1));
    soly(i+1) = soly(i) + (h/2)*(3*fy(i)-fy(i-1));
end
```

```
[solxE,solyE,time] = ImpODE(n,x0,y0,T,a,b,c,d,p,q);

%this line can be changed to ExpODE to calculate error with respect to
%Explicit Euler.

%evaluating errors at the midway point along the time axis.
normx = norm (solx(ceil(n/2)) -solxE(ceil(n/2)))/n;
normy = norm (soly(ceil(n/2)) -solyE(ceil(n/2)))/n;

end
```

func_eval.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function accepts f, a function of x and y, and the x and y values at %
% which the value of the function has to be evaluated.                       %
% The function returns the value of f at the specified values.               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [sol] = func_eval(expression,a,b)
syms x;
syms y;
sol = eval(subs(expression,{x,y},{a,b}));
end
```

Appendix B

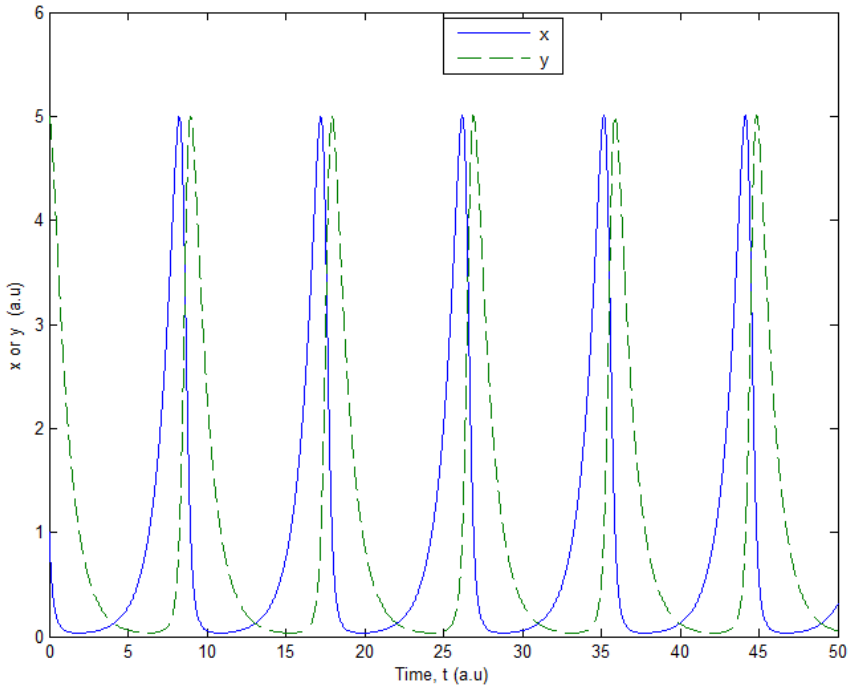

Figure B1 : Plot showing the variation of x and y as a function of time for the initial condition (1,5)

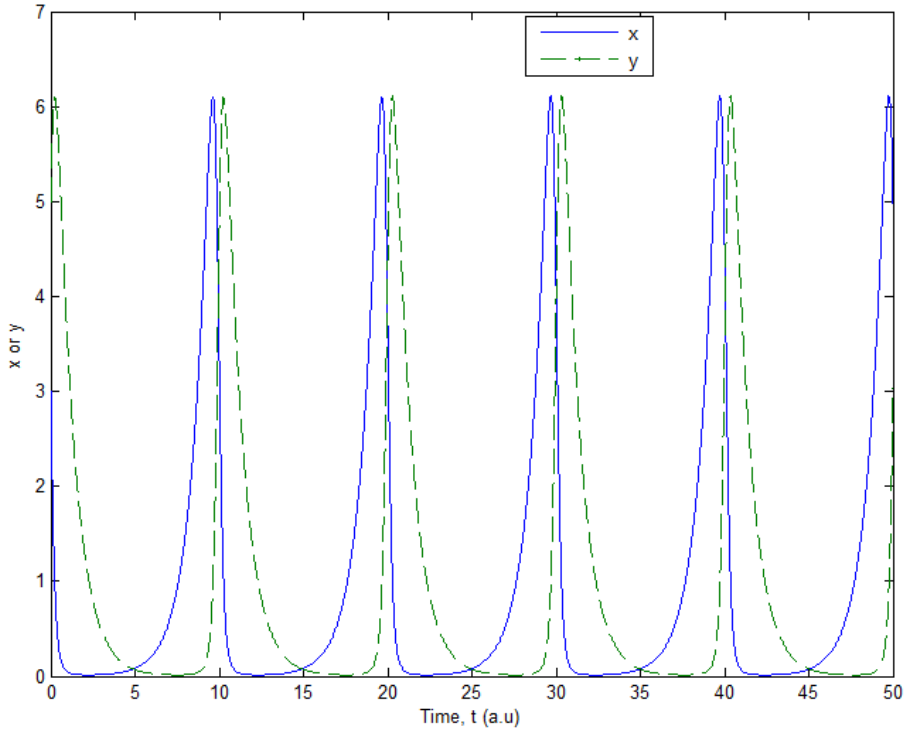

Figure B2 : Plot showing the variation of x and y as a function of time for the initial condition (3,5)

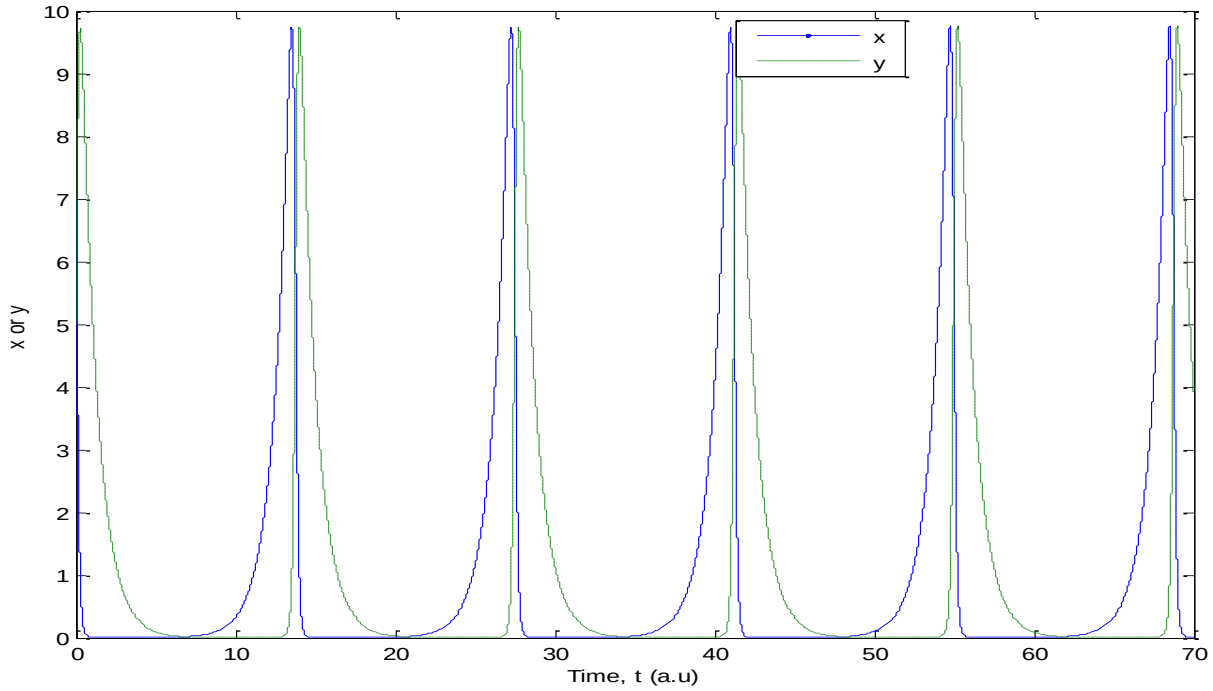

Figure B3 : Plot showing the variation of x and y as a function of time for the initial condition (7,5)

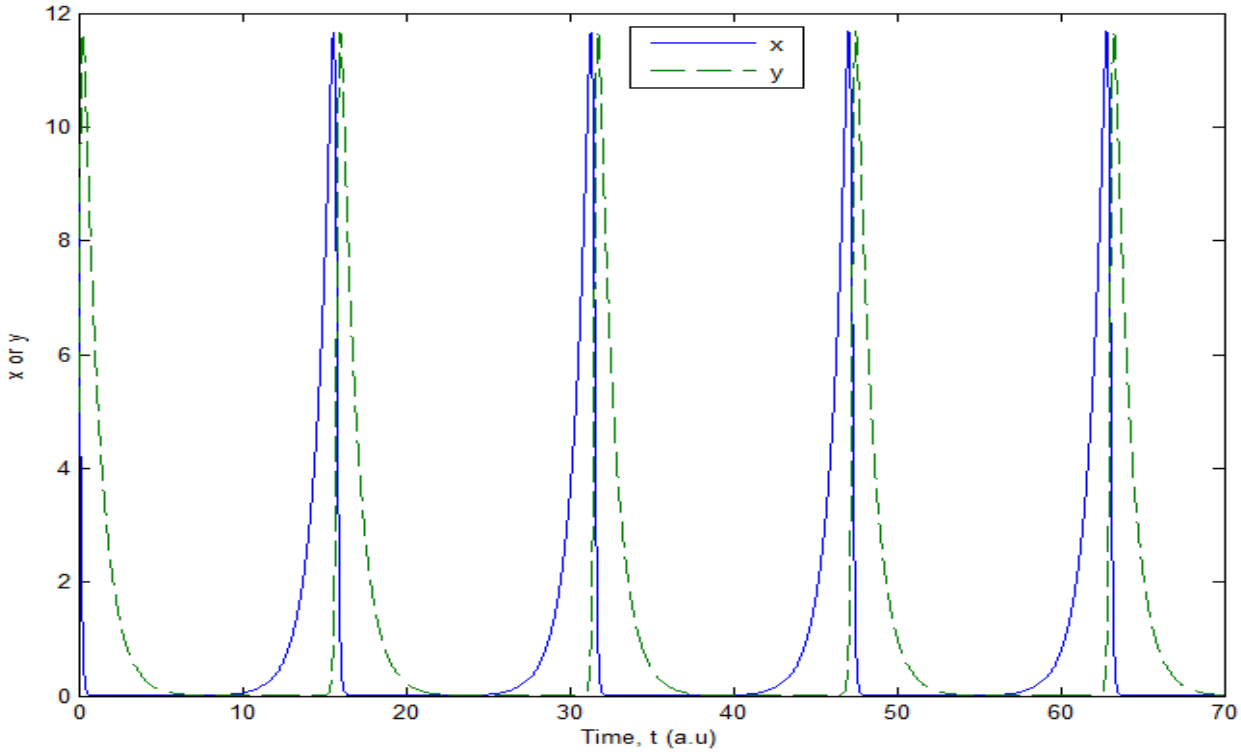

Figure B4 : Plot showing the variation of x and y as a function of time for the initial condition (9,5)