Assignment #1 - Lotka-Volterra Dynamical Systems

Jonathan Rainer

January 30, 2016

Question #1

Consider the simple nonlinear dynamical system (the "Lotke-Volterra predator-prey system")

$$x' = \frac{dx}{dt} = ax - bxy; \quad y' = \frac{dy}{dt} = cxy - dy$$

where x and y are the population densities of predators and prey respectively. In addition a, b, c, d are strictly positive constants.

a) i) Define a MATLAB function `lvderivs` to evaluate the time derivatives for this dynamical system, taking care to define any parameters as `global` if necessary.

The function that answers this question can be seen in Figure A.1.

ii) Verify that the origin (0,0) is a trivial fixed point of the dynamical system by evaluating `lvderivs` at this point.

To verify this we simply need to execute the function defined for the derivatives and show that the output is indeed $[0, 0]$, as Figure 1.

```
>> lvderivs([0,0])

ans =

 0     0
```

Figure 1: The evaluation of the lvderivs function at the point $[0, 0]$ as copied from the MATLAB command window

b) i) Write a MATLAB script which uses `fsolve` with your `lvderivs` function and range of random (physically reasonable) initial guesses to show that the system has a strictly positive fixed point.

The first thing that would make sense in this case would be to look at the dynamics phase portrait to see what the physically reasonable guesses might be in this case. This can be seen in Figure 2 with the code that produced the diagram shown in Figure A.2.

From this it would appear that we have at least one other fixed point than the trivial point already found and we know this to be true because of the following reasoning. If we look at the nullclines for $\frac{dx}{dt}$ we see the following:

$$0 = \frac{dx}{dt} = ax - bxy \implies 0 = x(a - by) \implies x = 0 \text{ or } y = \frac{a}{b}$$

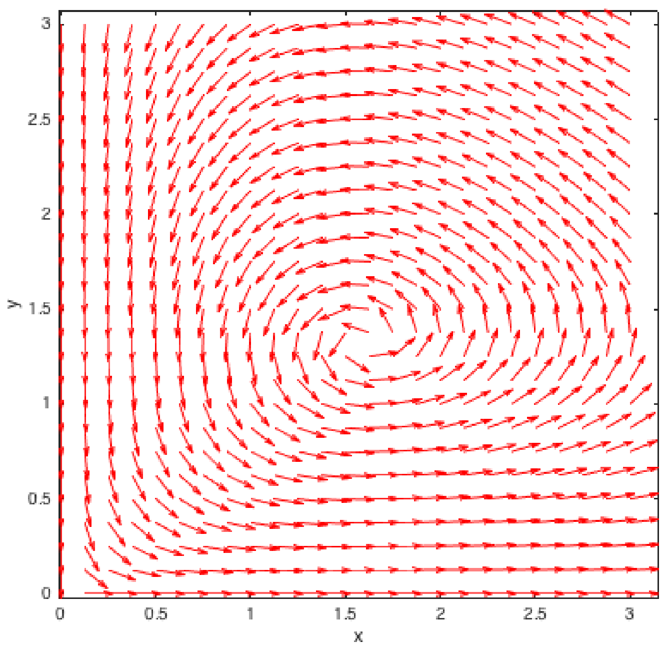

Figure 2: The phase plane for the given equations, plotted between 0 and 3. This shows that anything between 0 and 2 will probably converge to a root as we require

And then if we look at the nullclines for $\frac{dy}{dt}$ then we can also see the following:

$$0 = \frac{dy}{dt} = cxy - dy \implies 0 = y(cx - d) \implies y = 0 \text{ or } x = \frac{d}{c}$$

Now the fixed points will be the intersections of these solutions, so we'll get one from $x = 0$, $y = 0$ crossing over and then one from the non-trivial solutions crossing since all these are linear equations. So we know we're looking for two fixed points.

Now we can turn our attention to finding the fixed points, which is done by repeatedly iterating `[text]`fsolve for a variety of random starting values between $(0, 0)$ and $(2, 2)$ as we know the fixed points lie in those regions from observing the phase portrait. This obviously not infallible and relies on knowing how many fixed points there are in advance but it does provide a reasonable way of attempting to find as many as possible, and allows similarity thresholds which can be tuned so the output resembles what we see in the qualitative phase portrait. The output produced by the function can be seen below in Figure 3 with the code that produced the output shown in Figure A.3.

ii) Find the eigenvalues of the system's Jacobian matrix at this strictly positive fixed point.

This question is relatively easy because we can use the solutions we've already found from the previous part and then use another output from `fsolve` to find Jacobian as shown in Figure 4.

So now we know that the Jacobian is as follows:

$$J = \begin{bmatrix} 0 & -1.2 \\ 0.6667 & 0 \end{bmatrix}$$

```
>> eqi = equilibria_finder(100, 0.00001, 0.00001, 5, 0, 2)

eqi =

[1x2 double]
[1x2 double]

>> celldisp(eqi)

eqi{1} =

0     0

eqi{2} =

1.6000    1.3333

>> lvderivs(eqi{1})

ans =

1.0e-06 *

-0.3497    0.2331
```

Figure 3: The output from the MATLAB command window when running the defined functions, plus some code to display the fixed points. In addition we numerically verify that the points found are indeed fixed points by running them back through the derivative function.

Consequently J is diagonal so we can simply read the eigenvalues from the matrix. Thus the eigenvalues are $\lambda_1 = 0.8944i$ and $\lambda_2 = -0.8944i$

c) Using `ode45` to integrate the system forward in time, using a range of initial values with x-co-ordinate on the non-trivial y null-cline $x = \frac{d}{c}$ and using a time interval appropriate to reveal the system's dynamics.

The best way to solve this question is to use `ode45` with some of the phase plane tools we've already built up in trying to solve the previous questions. Using this we can generate several plots over increasing time spans and use that to develop an understanding of the system as a whole. So in the first instance we can use the script seen in Figure A.4 to generate the nine plots seen in Figure 5. These seem to be showing some kind of oscillatory behaviour, now if we plot the same behaviour within the phase portrait this will give us another idea of the dynamics and this can be seen in Figure 6. The code used to produce each individual plot can seen in Figure A.5, and the a script to automate seen in Figure A.6. All this together indicates that the dynamics around around the non-trivial fixed point are essentially oscillatory (i.e the fixed point is unstable) but decaying very slowly out from the fixed point itself.

d) i) Write your own MATLAB script (not a pre-defined function) which uses `lvderivs` to produce approximate solutions to the dynamical system using an explicit Euler method with a fixed time step, showing outputs for at least two different (fixed) time

```
>> eqi = equilibria_finder(100, 0.00001, 0.00001, 5, 0, 2)

eqi =

[1x2 double]
[1x2 double]

>> eqi{1}

ans =

1.6000    1.3333

>> [~,~,~,~,J] = fsolve(@lvderivs,eqi{1})

J =

-0.0000   -1.2000
0.6667    0.0000

>> eig(J)

ans =

-0.0000 + 0.8944i
-0.0000 - 0.8944i
```

Figure 4: The output from the MATLAB command window when using `fsolve` to extract the Jacobian and find its eigenvalues.

steps.

The MATLAB code that solves this problem can be seen in Figure A.7 and the plots produced can be seen below in Figure 7. The code that produced these plots can be seen in Figure A.8.

ii) Explain why these solutions differ to those of part c)

As can be seen from the Figure [??] the reason for the difference lies in the choice of time step used to estimate the solution to the equations. As the time step gets shorter the estimated solutions gets much closer to `ode45`'s solution and the reason for this is because Euler's method relies on the idea that the function itself doesn't change a great deal within the interval described by each time step. If the function is quite erratic as the solutions to the dynamical system are the changes predicted will no line up with what actually happens in the system. Add to that the fact that the method is iterative so any error added will accumulated across multiple calculations and you end up with a situation like we see in the first set of figures. It's important to choose a step size that takes into account the erratic nature of the graph and from there can give an accurate representation of the behaviour of the function to be estimated.

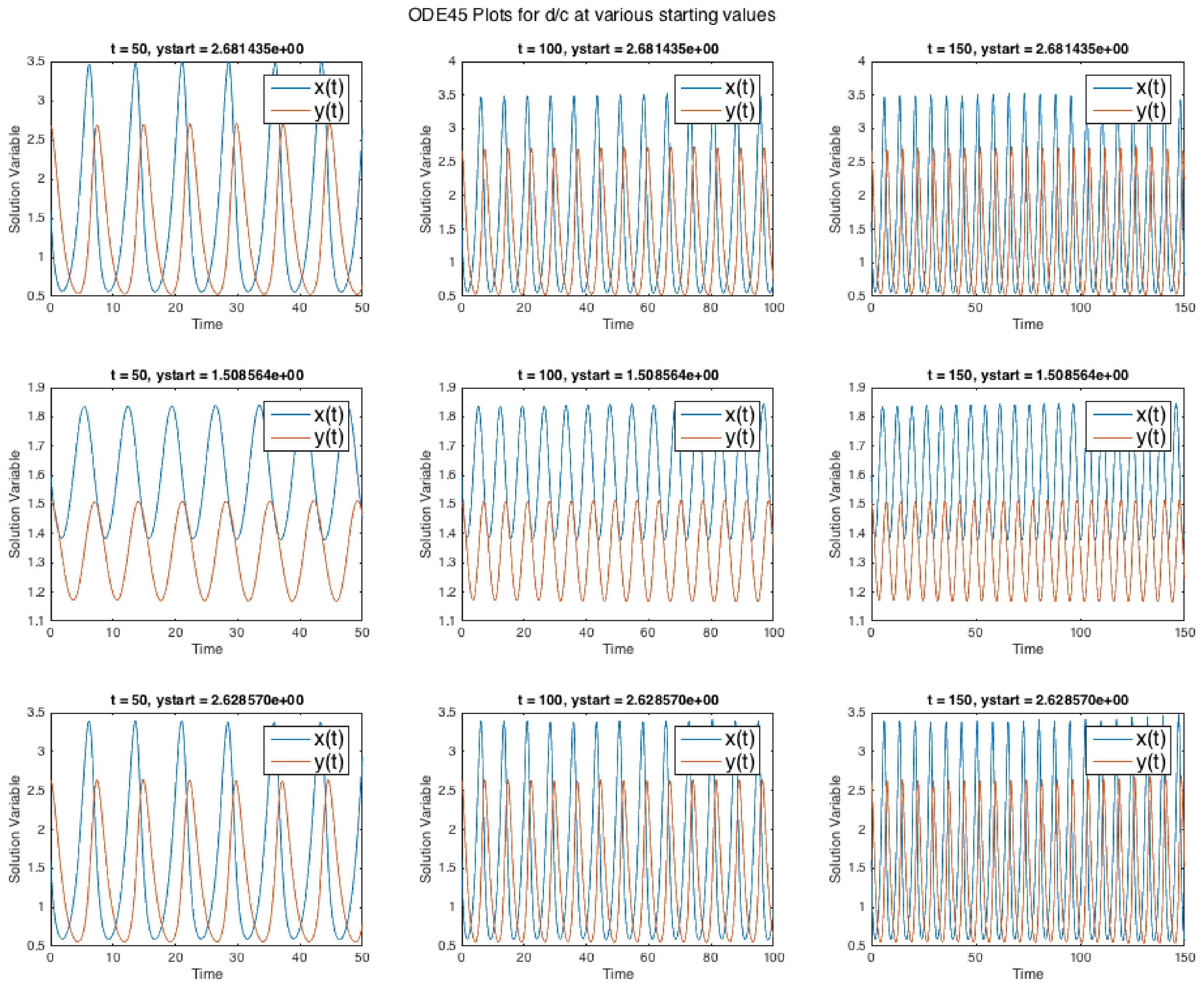

Figure 5: A selection of 9 plots done for different starting values of y over increasing time spans. The behaviour shown seems to be in some sense oscillatory as we'll see later in the phase portraits

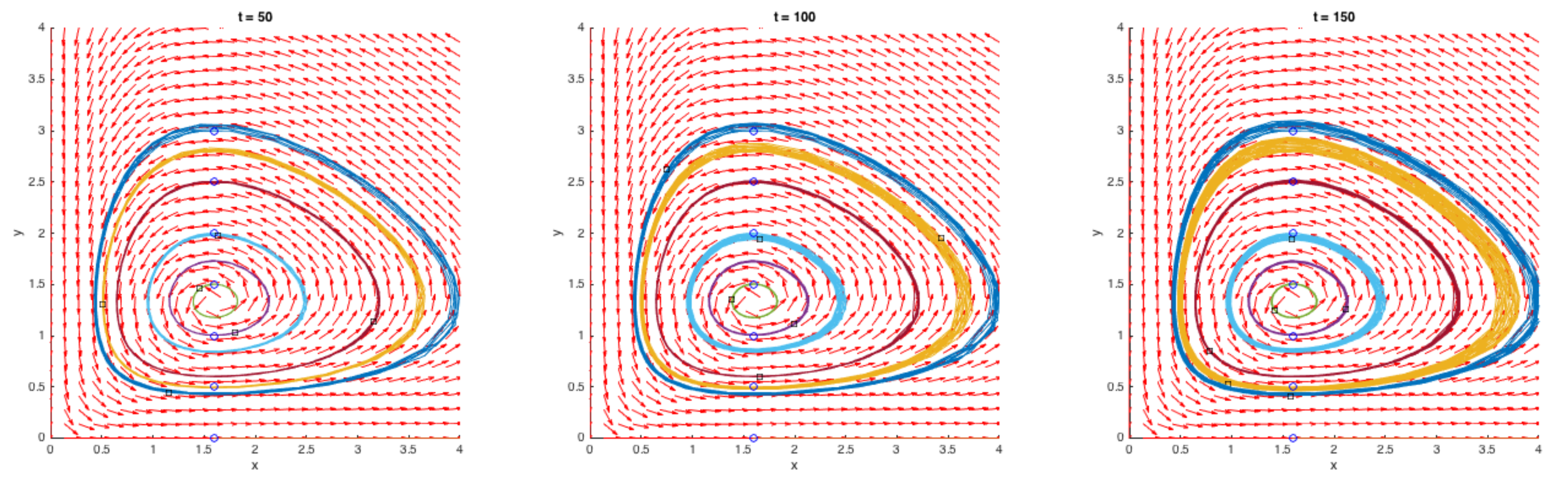

Figure 6: A selection of 3 plots done over the same starting y-values but with increasing time spent, the result is very similar to what we observe plotting the time series against time. An oscillatory behaviour around the non-trivial fixed point.

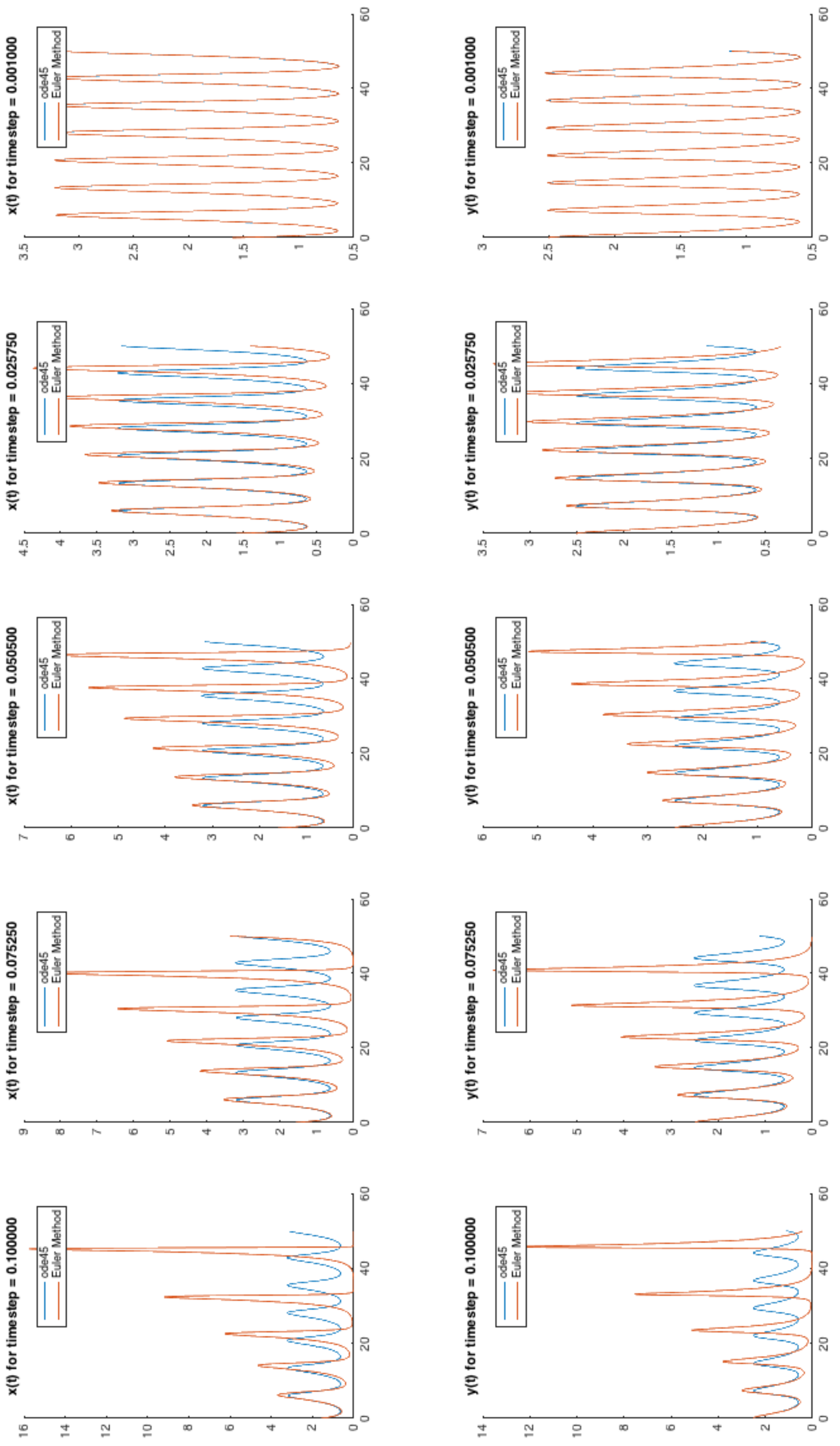

Figure 7: A selection of plots showing the effect of decreasing the step size in the Euler method using `ode45` as a comparison.

Question #2

The Lotka-Volterra model in question 1 has been criticised because it does not have a linearly stable strictly positive equilibrium.

a) i) Modify the model so that, at least for some set of biological meaningful parameter values, it has a linearly stable strictly positive equilibrium.

The model will be modified to limit the growth of the prey population (more is explained in the next part). Suffice to say that the modification is to replace the exponential growth term in the current model with the logistic equation thus:

$$x' = \frac{dx}{dt} = ax\left(1 - \frac{x}{K}\right) - bxy; \quad y' = \frac{dy}{dt} = cxy - dy$$

Where K is the carrying capacity of the population of prey.

ii) Explain what the modification represents biologically.

Biologically this means that you're getting rid of the odd assumption that in the absence of a predator species the prey will grow without limit. With this modification it takes into account the fact that the environment within which the prey live can only support up to a certain amount of prey and so saturates the first term at this carrying capacity K. In other words if the population grows beyond the limit set by K then the rate of change evaluates to a negative amount so the population cannot grow beyond the size of K. This introduces a linearly stable fixed point which points to the co-existence of both species under this model.

b) Show either algebraically or numerically that there is a linearly stable strictly positive equilibrium in your revised model.

Using the machinery we've already established we can fairly easily show that the equilibrium required does indeed exist. In the first instance if we form the equations for the $\dot{x}$ null-clines we see the following:

$$0 = ax\left(1 - \frac{x}{K}\right) - bxy \implies x\left(a - \frac{ax}{K} - by\right) = 0 \implies x = 0 \quad \text{or}$$
$$\left(a - \frac{ax}{K} - by\right) = 0 \implies bKy = aK - ax \implies y = \frac{a}{b} - \frac{ax}{bK} = \frac{a}{b}\left(1 - \frac{x}{K}\right)$$

and similarly for the $\dot{y}$ null-clines:

$$0 = cxy - dy = y(cx - d) \implies y = 0 \quad \text{or} \quad x = \frac{d}{c}$$

Now the fixed points are at $(0, 0)$ and $\left(\frac{d}{c}, \frac{a}{b}\left(1 - \frac{d}{Kc}\right)\right)$

Now the Jacobian is the following

$$J(x,y) = \begin{bmatrix} a(1-x) - \dfrac{2ax}{K} - by & -bx \\ cy & cx - d \end{bmatrix}$$

Now the first thing to spot is if we evaluate this at the non-trivial fixed point the bottom right entry will be 0. Which means that the trace will only recieve a contribution from the upper left term. Now if we evaluate this we see the following:

$$a\left(1 - \frac{d}{c}\right) - \frac{2ad}{cK} - a\left(1 - \frac{d}{Kc}\right) = a - \frac{ad}{c} - \frac{2ad}{cK} - a + \frac{ad}{cK} = -\frac{ad}{c}\left(1 + \frac{1}{K}\right)$$

Now this term will always be negative due to the restrictions on the constants we can choose. So we'll always have a strictly negative trace. Now considering the determinant we know that because the bottom right entry is 0 the determinant Δ is equal to $bcxy$. Now since x and y are both positive as this system models physical phenomena this term is positive so we have a positive determinant and a negative trace which leads to the fixed point is stable for any choice of parameter values. This can be seen from the plot included below in Figure 8 which shows several trajectories converging onto the non-trivial stable fixed point, plotted using the same tools as the other phase plots but with a new derivative function.

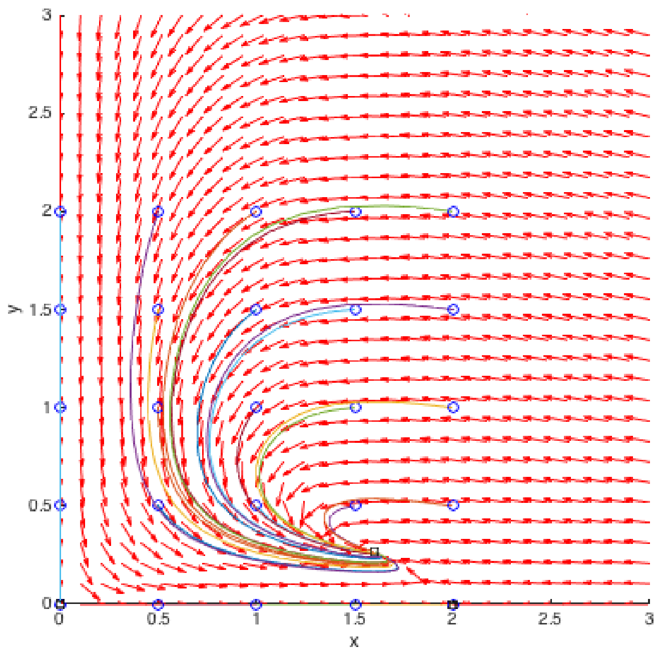

Figure 8: A plot for the new dynamical system with chosen parameter values, showing that all the trajectories converge to the stable fixed point, indicating a co-existence of the two species.

c) i) Using a numerical method, calculate the elasticity of the linearly stable strictly positive equilibrium point, for some chosen set of parameter values.

ii) If possible, confirm your result numerically.

Appendices

Appendix A

Programming Code

```
function F = lvderivs(co_ords)
%LVDERIVS Calculate a vector of the derivate of the two functions in
%our Lotka-Volterra Dynamical System.

    % Set up constants and define as global
    global a b c d
    a = 1.0;
    b = 0.75;
    c = 0.5;
    d = 0.8;

    % Factor out the recurring product to save space
    xy = co_ords(1)*co_ords(2);

    % Calculate derivatives
    dxdt = a*co_ords(1) - b*xy;
    dydt = c*xy - d*co_ords(2);

    % Arrange derivatives into the output vector
    F = [dxdt, dydt];
end
```

Figure A.1: The definition of the `lvderivs` function is relatively simple it just sets up the parameters, factors out some recurring parts and then computes the derivatives.

```
function [] = phase_portrait(deriv_func, x_range, y_range, num_points)
%PHASE_PORTRAIT Plot the phase portrait for a given derivative function.
    % Generate a mesh grid of points so that we have data to feed into
    % derivative function.
    [x,y] = meshgrid(linspace(x_range(1), x_range(2), num_points), ...
                        linspace(y_range(1), y_range(2), num_points));
    % Preallocate arrays to hold the values of dx/dt and dy/dt resp. at
    % each x,y position.
    u = zeros(size(x));
    v = zeros(size(x));
    for i = 1:numel(x)
        derivs = deriv_func(0, [x(i), y(i)]);
        u(i) = derivs(1);
        v(i) = derivs(2);
    end
    for i = 1:numel(x)
        vec_mod = sqrt(u(i)^2 + v(i)^2);
        u(i) = u(i)/vec_mod;
        v(i) = v(i)/vec_mod;
    end
    quiver(x,y,u,v,'r');
    figure(gcf)
    xlabel('x')
    ylabel('y')
    axis tight equal;
end
```

Figure A.2: The code to generate the phase portrait displayed in Figure 2. The code simply generates the required vectors and then computes the rate of change at each point and uses the quiver command to automate all the plotting.

```
function equilibria = equilibria_finder(iterations, sim_threshold, ...
round_threshold, rounded_dec_places, rand_min, rand_max)
%EQUILIBRIA_FINDER Will find the fixed points of a given dynamical system.
%   Runs fsolve over a number of iterations to try and discover as
%   many points of the dynamical system as possible.

    % Set up a structure to store the output equilibrium
    equilibria = {};
    % Perform fsolve, the specified number of times, checking for
    % duplicates each time.
    for i=1:iterations
        % Turn off display even if a solution is found, we don't care about
        % help dialogs.
        options = optimset('Display','Off');
        % Find a result of the equation
        eq_temp = fsolve(@lvderivs, ...
            (rand_max - rand_min).*rand(1,2) + rand_min, options);
        % Check if the fractional part of what's been found is
        % large enough to merit consideration
        if abs(eq_temp - fix(eq_temp)) < round_threshold
            % If not then round it to the specified number of decimal
            % places
            eq_temp = round(eq_temp, rounded_dec_places);
        end
        % Add in the new solution to the current set of equilibria
        equilibria = duplicate_checker(sim_threshold, equilibria, eq_temp);
    end
end

function equilibria_new = duplicate_checker(sim_threshold, equilibria, ...
candidate)
% DUPLICATE_CHECKER Looks at the given candidate to test if it's already
% present within the set of found equilibria
    % Iterate through the equilibria we already have
    for i=1:size(equilibria, 1)
        % Compare each one to the candidate in turn, and see if it's lower
        % than the similarity threshold (turned into a vector due to
        % MatLab's syntax).
        if abs(equilibria{i, 1} - candidate) < sim_threshold
            % If it is then there is no need to add a new equilibria
            equilibria_new = equilibria;
            return
        end
    end
    % If no similar element is found then add in the candidate.
    equilibria_new = [equilibria ; candidate];
end
```

Figure A.3: This code simplifies repeatedly running `fsolve` but also allows us to check for duplicates when the repeated runs are aggregated. The secondary function allows for this by checking whether what exists is within a similarity threshold and if so ignoring it as a new fixed point. This allows us to account for the situation where you get several roots occuring within ~~a very small interval. This option can be removed by setting the threshold to 0 of course.~~

```
clf
suptitle(sprintf('ODE45 Plots for d/c at various starting values\n\n'));
for p = 1:3
    start_val = rand_range(1,3,1);
    for k = 1:3
        subplot(3,3,(p-1)*3 + k)
        [t,y] = ode45(@lvderivs_time, 0:0.1:k*50, [d/c,start_val]);
        plot(t,y)
        legend({'x(t)','y(t)'},'FontSize', 16);
        xlabel('Time', 'FontSize', 16);
        ylabel('Solution Variable', 'FontSize', 16);
        title(sprintf('t = %d, ystart = %d', k*50, start_val), ...
            'FontSize', 16);
    end
end
```

Figure A.4: This code automates the plotting of several ODE45 outputs, over multiple time spans and multiple initial starting values of y, the x-coordinate already being fixed to lie on the given null-cline.

```
function [] = phase_portrait_trajectories(deriv_func, ...
    x_range, y_range, num_points, x_vals, y_vals, time_span)
%PHASE_PORTRAIT_TRAJECTORIES Summary of this function goes here
    hold on
    phase_portrait(deriv_func, x_range, y_range, num_points);
    for i = x_vals
        for j = y_vals
            [~, sols] = ode45(deriv_func,[0,time_span], [i;j]);
            plot(sols(:,1),sols(:,2));
            plot(sols(1,1),sols(1,2),'bo') % starting point
            plot(sols(end,1),sols(end,2),'ks') % ending point
        end
    end
    axis([x_range, y_range]);
    hold off
end
```

Figure A.5: This code automates the plotting of a phase portrait with the associated trajectories, it also uses ODE45 and marks the beginning and end of the trajectories as it appears on the diagram.

```
clf
for k = 1:3
    subplot(1,3, k)
    phase_portrait_trajectories(@lvderivs_time, [0,4],[0,4],30, ...
        d/c, 0:0.5:3 ,k*50)
    title(sprintf('t = %d', k*50), 'FontSize', 16);
end
```

Figure A.6: This code simply automates the plotting of multiple phase portraits using the `subplot` commands.

```
function sols = euler_method(deriv_function, step_size, time_limit, ...
    y_start)
%EULER_METHOD Uses the Euler method to provide approximate solutions to the
%dynamical system specified in deriv_function.
    global d c;
    iteration_number = ceil(time_limit/step_size);
    sols = zeros(iteration_number,3);
    sols(1,:) = [0,d/c,y_start];
    for i=2:iteration_number
        derivs = deriv_function(sols(i-1,2:3));
        sols(i,:) = [sols(i-1)+step_size, sols(i-1,2) + ...
            step_size*derivs(1), sols(i-1,3) + step_size*derivs(2)];
    end
end
```

Figure A.7: This code is an implementation of the Euler method which first sets the starting values in a matrix and then iterates finding the derivative and predicting the next point in the function.

```
clf
[t,y] = ode45(@lvderivs_time,0:0.0001:50, [d/c,2.5]);
grid_x_size = 5;
for i=1:grid_x_size
    time_steps = linspace(0.1,0.001,grid_x_size);
    subplot(2,grid_x_size,i);
    sols = euler_method(@lvderivs, time_steps(i), 50, 2.5);
    hold on
    plot(t,y(:,1))
    plot(sols(:,1),sols(:,2))
    title(sprintf('x(t) for timestep = %f', time_steps(i)));
    legend('ode45','Euler Method')
    hold off
    subplot(2,grid_x_size,i+grid_x_size)
    hold on
    plot(t,y(:,2))
    plot(sols(:,1),sols(:,3))
    legend('ode45','Euler Method')
    title(sprintf('y(t) for timestep = %f', time_steps(i)));
    hold off
end
```

Figure A.8: This code automates plotting the Euler approximation against the `ode45` output to see how they compare for decreasing step size.

```
function F = new_lvderivs(co_ords)
%NEW_LVDERIVS Calculate a vector of the derivate of the two functions in
%our Lotka-Volterra Dynamical System with a change made to allow for
%non-exponential growth in the prey population.

    % Set up constants and define as global
    a = 1.0;
    b = 0.75;
    c = 0.5;
    d = 0.8;
    K = 2;

    % Factor out the recurring product to save space
    xy = co_ords(1)*co_ords(2);

    % Calculate derivatives
    dxdt = a*co_ords(1)*(1-co_ords(1)/K) - b*xy;
    dydt = c*xy - d*co_ords(2);

    % Arrange derivatives into the output vector
    F = [dxdt, dydt];
end
```

Figure A.9: This code calculates the new derivative function with the added carrying capacity scaling to introduce the linearly stable fixed point.