Chapter 8

Matlab Course

8.1 First Steps in Matlab

In this part of the book we show how to use a computer software package (such as Matlab) to simulate the various model types that were introduced in Part I, on theory. Moreover, we show how to use the models for data fitting and parameter estimation. The computer course is designed so that it can be used by students who have no computational experience at all, as well as by students who are already familiar with a computer software package (such as Maple, Mathematica, Matlab, C++, or similar).

On Unix machines, Matlab is started with the command:

```
Matlab
```

In Windows, Matlab is started by clicking the following menu items:

Start
Programs
Matlab

A large Matlab window will appear (called a workspace window). Inside this window is another, smaller window, called a Matlab command window. All Matlab commands are entered here. Note: the commands typed in this workspace cannot be saved.

8.1.1 Constants and Functions

Constants

Try the first example:

```
>> a=5

>> b=7

>> a*b
```

You should see:

```
>> a=5

a =

    5

>> b=7
```

```
b =

    7

>> a*b

ans =

      35
```

where `ans` stands for the answer. This means that there is a location in the computer which has been tagged by Matlab with the label `ans` and which now contains a value of 35.

The meaning of these lines should be apparent. The first command gives the value 5 to the constant named `a` and the second line gives the value 7 to the constant named `b`. The third line multiplies the two constants. Note that you can place a semicolon (';') after a line of code so that you do not see the output. This is useful when defining constants as above. Type

```
>> c=3.2;
```

Matlab is not sensitive about spacing. If you would like to make spaces between operations and variables, it is ok to do so.

Functions

In many instances, it may be useful to either have a particular expression stored temporarily as a function suitable for use in your Matlab session, or stored permanently for use in any Matlab session. It is important to note here that anything typed in the command window is considered as temporary code which cannot be retrieved once the session has been closed.

There are two ways that we can define a function using Matlab. The first way to define a function is to use an inline function definition; the second is to write a function m-file. We discuss both along with illustrative examples.

Functions: Inline Function Definitions

Consider the function $f(x) = a\mathbf{e}^{bx}$. To define this function as an inline function within Matlab, we write:

```
>> f=inline('5*exp(7*x)','x')
```

We have defined a function $f(x)$, where f is described as an array. The first part of the array is the expression defining our function, and the second part of the array describes all independent variables. Note that it is important to include the multiplication signs in the Matlab commands. What happens when you omit them?

Now we can evaluate f at $x = 3$ as follows:

```
>> f(3)
```

Functions of more than one variable are defined similarly. For example, $g(x, a, b) = a\mathbf{e}^{(bx)}$ is entered into Matlab as

```
>> g=inline('a*exp(b*x)','x','a','b')
```

We can evaluate g at $x = 3$, $a = 5$, and $b = 7$ as follows:

```
>> g(3,5,7)
```

In general, any inline function can be written in the following form:

```
>> g=inline('string defining expression','Var1','Var2',...)
```

You may include as many dependent variables as necessary.

Functions: m-files

Now we consider the case in which we may need to use our function at some later time. Here, we write our function in a m-file. There are two types of m-files: script files, and function files. For now, we will only focus on function files.

To make an m-file, click on **File**. Next select **New** and click on **M-File** from the pull down menu. You will be presented with the Matlab **Editor/Debugger** screen. Here, you will type your code, make changes, etc. When you are done with typing, click on **File** in the Matlab **Editor/Debugger** screen and click **Save As**. Choose a name for your file (e.g.., myfirstfunction.m) and click **Save**. Close the m-file.

Now let's reopen it! To reopen your m-file (you may need to do this at some point to make changes), type the following code in the command window:

```
>> edit myfirstfunction.m
```

Press **Enter** (or **Return**) on your keyboard. The function file you will create should have the following form:

function [output variable list]=funcname(input variable list)

If there is only one output variable, the brackets are omitted. As before, let's consider defining the function $f(x) = a\mathbf{e}^{(bx)}$. Type the following into the m-file window

```
function y = myfirstfunction(x)
a = 5
b = 7
y = a*exp(b*x)
```

Notice that we have omitted the square brackets. If you wish to make comments in your file, you may use the % key at the beginning of a line. For example, edit your m-file so that it looks like the following

```
% This is my first m-file!
function y = myfirstfunction(x)
a = 5
b = 7
y = a*exp(b*x)
```

Save and close the m-file.

Plotting Functions

We would like to plot the function f defined above. First let's plot it using the inline function:

```
>> x=0:0.01:0.5

>> plot(x,f(x))

>> grid on
```

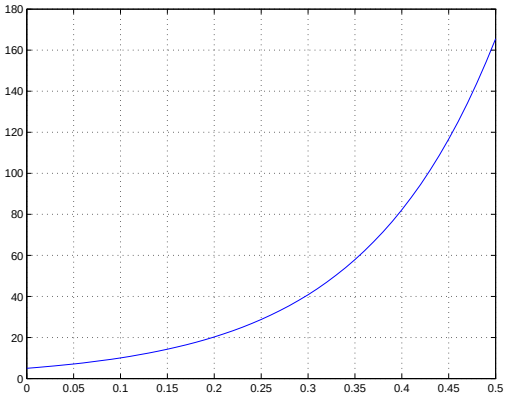

Figure 8.1.

The first argument of the `plot` command specifies the domain. It shows that we want to define our function from a value of x = 0 to a value of x = 0.5 by steps of 0.01.

Now let's plot the same function, but this time using the code from our m-file:

```
>> x=0:0.01:0.5

>> plot(x,myfirstfunction(x))

>> grid on
```

Exercise 8.1.1

(a) Define a function $h(t) = 5\sin(ct)$. Don't forget about the implied multiplications!

(b) Choose $c = 1.3$ and evaluate h at $t = 0.0$, $t = 1.0$ and $t = 3.2$.

(c) Plot the function h for $t \in [0, 5]$.

A Helpful Tip

It may be useful to place your code into an m-file, so that you can save and print you work at a convenient time. Your code should be written just as if you were using the command window. Specifically, you must start on a new line for each new operation you wish to complete. To run your m-file, type the file name (not the extension .m) into the command window.

Getting Help

The help command is the most basic way to determine the syntax and behavior of a particular function. From the Matlab command window type

```
>> help magic
```

Information is displayed directly in the command window. The command `help` by itself lists all the directories, with a description of the function category each represents.

The `lookfor` command allows you to search for functions based on a keyword. As an example, type

```
>> lookfor plot
```

Matlab searches through the first line of help text, which is known as the H1 line, for each Matlab function, and returns the H1 lines containing a specified keyword. Adding `-all` to the `lookfor` command searches the entire help entry, not just the H1 line. Try adding `-all` to the `lookfor` command.

Clearing Matlab Memory

It will be helpful to know that there are several commands that can be used to clear Matlab's internal memory, either partially or completely. The `clear` command will remove all variables from your workspace. The `clear all` command will remove all variables and functions from memory, leaving the workspace empty. The `clear name` command removes that particular m-file or variable name from the workspace (here, name is replaced by the variable or m-file name). To clear several variables at once, type `clear name1, name2,...`. Try the following:

```
>> d=1;

>> d
```

Mass (in grams)	Size (in mm)
31	140
45	160
52	180
79	200
122	220
154	240
184	260
210	280
263	300
360	320

Table 8.1. *Brown trout data*

```
d =

     1

>> clear d

>> d

???  undefined function or variable 'd'
```

8.1.2 Working with Data Sets

In most applications, it is necessary to work with experimental data. Moreover, data can be analyzed using mathematical models. To illustrate this procedure we use the following example, listing the mass and size of 10 brown trout (*Salmo trutta forma fario*) in Table 8.1.

Lists (Vector arrays)

Each experimental measurement consists of two numbers, mass and size. We will define two lists to save these measurements, one for the mass and one for size. A list in Matlab is described using a vector array. We begin by defining the list of masses:

```
>> mass=[31 45 52 79 122 154 184 210 263 360]

mass =

      31 45 52 79 122 154 184 210 263 360
```

We can extract specific elements from the vector array by specifying their position. For example:

```
>> mass(5)

ans =

    122
```

We continue by defining the list of sizes. Of course, we could use a command similar to the one above. But it should be obvious that the size of the fish increases by 20 from fish to fish with the size of the i-th fish given by $120 + i * 20$. We can take advantage of this observation, and increment each value of the vector in the following way:

```
>> size=[140:20:320]

size =

      140  160  180  200  220  240  260  280  300  320
```

Here we have defined the start number, the value of the increment, and the last number. Note that both mass and size have the same number of entries. This will be important when it comes time to plot this data.

Plotting With Lists (Vectors Arrays)

We now use the vector arrays that we have just defined to show the brown trout data graphically. To show the data graphically, we again use the `plot` command, but now with a different set of arguments than before:

```
>> plot(mass,size,'*')
```

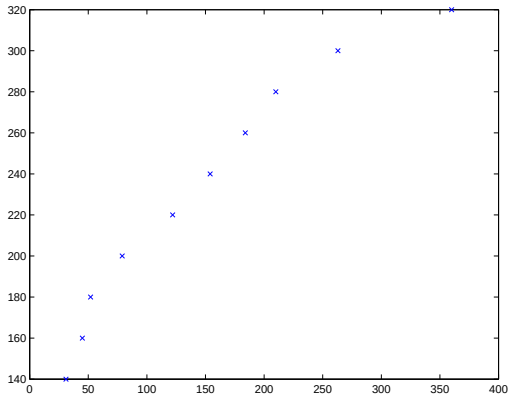

Figure 8.2.

What happens when you remove the * ?

Exercise 8.1.2 Research the `plot` command, and remake the plot, this time labeling the axes and adding a title. Try first by using the pull down options from the figure display window and then by using the command window. It is always good practice to enclose your labels in single quotes, that is, use '`mass`' and '`size`'.

Data Transformations

The above plot supports the idea of a power law of the form

$$\text{size} = a\,\text{mass}^b$$

to describe the brown trout data. We would like to find a and b (the fitting parameters). If we take the logarithm of the above formula,

$$\ln(\text{size}) = \ln(a) + b * \ln(\text{mass}),$$

then ln(size) is a linear function of ln(mass), and a and b can be found easily from the y-intercept and the slope of the function. We first transform the brown trout data to a logarithmic scale:

```
>>log_m=log(mass)

log_m =

     3.433987204 3.806662490 3.951243719 ...5.886104031
```

If you pass a vector to a predefined math function (`log` in our case), it will return a vector of the same size, and each entry is found by performing the specified operation on the cooresponding entry of the original vector. The new vector `log_m` has the same number of entries as the vector `mass`. Similarly,

```
>>log_s=log(size);
```

Here, we use the semicolon (';') to finish the last command. The semicolon means that the result of this command will not be printed on the screen. If at anytime we are interested in seeing the value of the vector array `log_s` we can simply type

```
>>log_s

log_s =

     4.941642423 5.075173815 5.192956851 ...5.768320996
```

In the next section, we use the vector arrays containing the transformed data to determine the values of the fitting parameters a and b by linear regression.

8.1.3 Linear Regression

We expect the transformed data to have a linear relationship. Let's have a look:

```
>> plot(log_m,log_s)
```

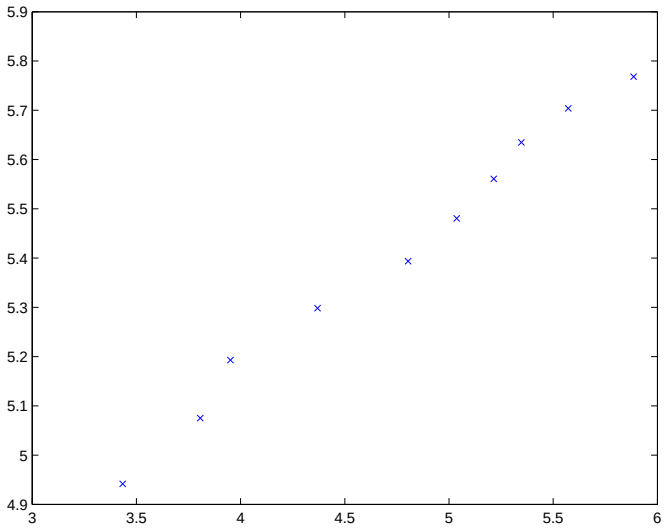

Figure 8.3.

From the graph, it appears that indeed there is a linear relationship between the mass and the size of the fish. We now try to find the regression line, which is the straight line that best fits the data. To learn more about regression, see Section 7.2 and reference [44] of the text.

Fitting with the least squares: `polyfit` and `polyval`

Statistical functions are available in Matlab. Unlike Maple, they are activated right away, so we do not have to worry about including the appropriate library packages.

We use the `polyfit` command to find the regression line. We first assume that the fit is linear. That is, fit the function $y = ax + b$ to our data.

```
>> coefficients1=polyfit(log_m,log_s,1)

ans =

   0.3351    3.8130
```

The "`polyfit(x,y,n)`" command finds the coefficients of a polynomial p(x) of degree n (note that above we choose n=1 as we expect a linear relationship) that fits the data in the least squares sense. This means that the sum of the squares of

the pointwise distance between the polynomial and the data should be minimized. Here, the parameter $a = 0.3351$, and $b = 3.8130$

We could also try a quadratic fit (we fit function $y = ax^2 + bx + c$). If we assume that there is a quadratic relation, then we write:

```
>> coefficients2=polyfit(log_m,log_s,2)
```

As you can see, the coefficient of x^2 is very small (a = -0.0099) compared to the other coefficients. This indicates that a linear fit is sufficient. We return to the comparison of model fits to data in Section 7.3 of the text.

Checking the fit

Now we would like to plot the regression line. We need to convert the results of the fit to a function. We do this using the `polyval` command.

```
>> newy=polyval(coefficients1,log_m)
```

Here we have generated a new value of y (an estimate) based on the coefficients found with polyfit. Now let's plot the fitted function together with the brown trout data:

```
>> plot(log_m,log_s,'*',log_m,newy,'-')
```

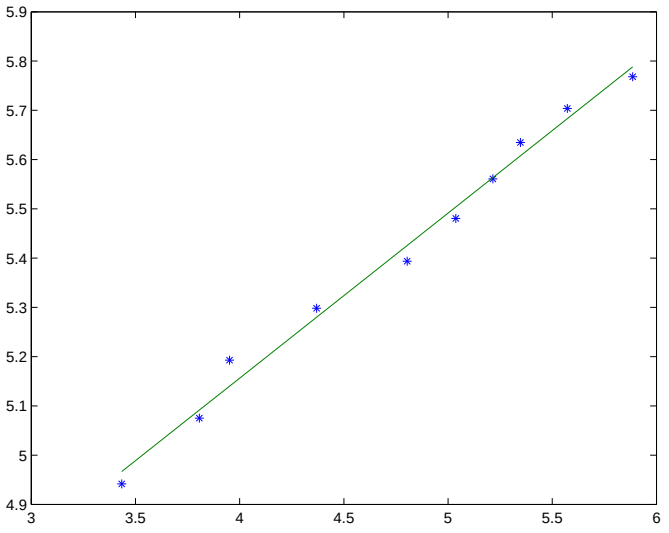

Figure 8.4.

That's looking very good! Finally, we will check the fit with the original data set on the non-logarithmic scale. To do so, we need to transform `newy` to the non-logarithmic scale, which we accomplish with the following steps:

```
g=exp(newy)
```

Now let's see how the function $g(x)$ fits the original brown trout data on the non-logarithmic scale:

```
>> plot(mass,size,'*',mass,g,'-')
```

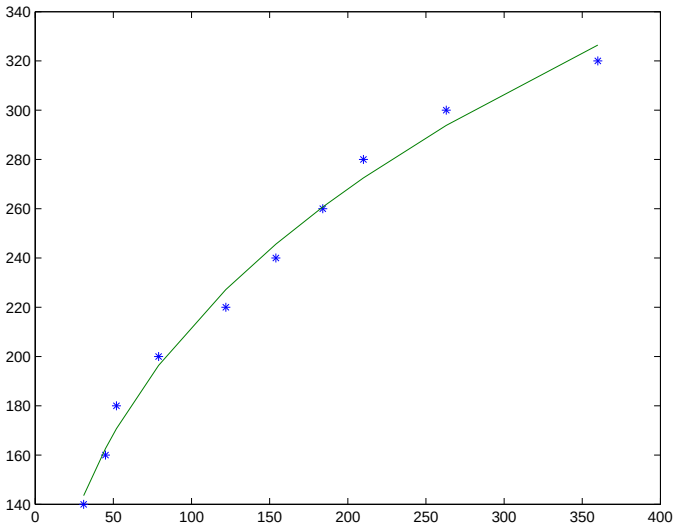

Figure 8.5.

This also is looking very good!

Exercise 8.1.3: World population 1850-1997 The world population from 1850 to 1997 is given in Table 8.2.

Year	Size of population (in Millions)
1850	1200
1940	2249
1950	2509
1960	3010
1970	3700
1985	4800
1997	5848.7

Table 8.2. *World population data*

(a) Define lists containing the data.

(b) Plot the data (world population versus year).

(c) Transform the data set for the world population to a logarithmic scale, and show your results in a graph.

(d) Fit the data of the world population. Show the fit on both the logarithmic and on the non-logarithmic scale. What type of function did you use to fit the data?

(e) Try fitting the data directly with a quadratic and a cubic function. Which fit do you think is best?

8.2 Discrete Dynamical Systems: The Ricker Model

We briefly discussed the Ricker model in Section 2.2.4, and saw that use of the model is appropriate for describing populations with non-overlapping generations. We determined the fixed points of the model, as well as their stability, and alluded to the fact that the Ricker model can exhibit complex dynamics, such as cycles and chaos, for certain choices of the model parameters. We will now use Matlab to conduct a thorough investigation of the dynamics of this model.

We begin with simplifying equation (2.24) by letting $a = e^r$, $b = r/k$, and $\overline{x}_n = bx_n$. After dropping the overbars, we obtain the simplified Ricker model,

$$x_{n+1} = ax_n e^{-x_n}. \tag{8.1}$$

Of interest for the remainder of this section is the behaviour of (8.1) and its dependence on the value of the model parameter a. Although the restriction $r > 0$ for the original Ricker model implies $a > 1$, we will study (8.1) in more generality, and allow $a > 0$.

Let x^* be a fixed point of (8.1), that is, x^* satisfies

$$x^* = f(x^*) = ax^* e^{-x^*}.$$

Although it is easy to solve this equation for x^* by hand, we will use Matlab so that we can learn the "solve" command. (Note: use of this command requires the Matlab 'Symbolic Math Package.' If this package is not installed on your version of Matlab, then skip to the next paragraph.)

```
>> f = 'a*x*exp(-x)-x'
>> roots=solve(f,'x')

roots=
0
-log(1/a)
```

The arguments of the `solve` command are, first, the equation and, second, the quantity we want to find. In this case, we obtain two fixed points: $x_1^* = 0$ and $x_2^* = \ln(a)$. The trivial fixed point $x_1^* = 0$ describes a population which is extinct. Note that this fixed point exists for all values of the model parameter a. The nontrivial fixed point, $x_2^* = \ln(a)$, exists only for $a > 1$ (this is consistent with our earlier observation that $r > 0$ implied $a > 1$).

To determine the stability of the fixed points, we need

$$f'(x) = ae^{-x}[1 - x],$$

so that

$$\begin{aligned} f'(x_1^*) &= f'(0) = a, \\ f'(x_2^*) &= f'(\ln(a)) = 1 - \ln(a). \end{aligned}$$

Exercise 8.2.1

(a) At what value of a does the stability of the trivial fixed point, $x_1^* = 0$, change?

(b) Plot the function $g(a) = 1 - \ln(a)$. For which values of a is $|g(a)| < 1$ ($|g(a)| > 1$)? That is, when is the nontrivial fixed point stable (unstable)?

(c) Sketch (by hand) a partial bifurcation diagram for the simplified Ricker model, (8.1), as we did in Section 2.2.3 for the rescaled logistic map (see Figure 2.7).

In terms of the simplified Ricker model, we say that there are bifurcations at $a = 1$ and $a = e^2$. The bifurcation is $a = 1$ is called a *transcritical bifurcation*. We defer discussion of the bifurcation at $a = e^2$ to later.

We are interested in plotting solution trajectories for various values of the model parameter a. For a given value of a, suppose that we wish to iterate the map 20 times, and plot the iterates as a function of the iteration number. This means that we need to create a list with 21 coordinates of the form $[i, x_i]$ (the 20 iterates plus the initial condition). Since we need to keep track of the current iterate to create the next, writing an appropriate recursive command can be a bit tricky. The easiest way to create the list of coordinates might be to use a `for` statement to build up the list recursively. We take a small detour now to learn about these this command.

The `for` statement

The `for` statement is a type of repetition statement. It provides the ability to execute a command or a sequence of commands repeatedly. The sequence of the commands to be executed repeatedly is listed between a `for` command and an `end`. The remainder of the statement specifies the number of times that the sequence of commands needs to be executed. Before executing the command below, note that we have stretched the statement over 3 lines. Press **shift** and **enter** to get to the next line without executing the command! Try the following now:

```
>> for i=0:2:20,
i
end
```

The variable `i` counts the number of times the sequence of commands is executed. Here, Matlab is instructed to start with `i=0`, increase `i` by 2 at the end of each repetition, and terminate the repetition as soon as `i>20`. Did the command give the result you expected?

Adding and replacing elements within vector arrays

Define the following list of elements:

```
>> L1 = [ 1 [1 2] [1 2 3] 2 ]


L1 =
   1 1 2 1 2 3 2
```

Note that it is not necessary to include the square brackets inside your array, as Matlab gets rid of them. Also, if an array is ever too long to fit on one line, type 3 dots (...) and continue the vector on the next line (using the **shift** + **enter** trick). Now let's try to insert elements in this list. For example, to insert `[33,12,-3]` into `L1` (at the end) type

```
>> L1(8) = 33;
>> L1(9) = 12;
>> L1(10) = -3;
```

Notice that the L1 vector had 7 entries. Since Matlab allocates memory for all variables on the fly, this allows you to increase the size of a vector by assigning a value to an element that has not been previously used (i.e., elements 8, 9, and 10).

You can just as easily change an element in the list. For example, to change the first element to a 2, rather than a 1, type

```
>> L1(1) = 2;
```

Recursive definitions for vectors. Using the `for` command

To create a a pair of lists recursively, we use a `for` command to insert elements into each list one at a time. For example, we write our first list as:

```
>> for i=1:5,
L1(i) = i
end


L1 =
     1
L1 =
     1 2
L1 =
     1 2 3
L1 =
     1 2 3 4
L1 =
     1 2 3 4 5
```

Now let's define a second list.

```
>> for i=1:5,
L2(i) = i*i
end


L1 =
     1
L1 =
     1 2
L1 =
     1 2 4
L1 =
     1 2 4 16
L1 =
     1 2 4 16 25
```

Note that our `for`-loop repeatedly redefines the *same* variables, `L1` and `L2`, recursively.

Plotting a trajectory

We now return to the Ricker model, and use the new Matlab commands to create a list of coordinates of a trajectory, let's say for $a = 0.8$ (from the linear stability analysis, we know that the trivial fixed point $x_1^* = 0$ is stable in this case, and the nontrivial fixed point $x_2^* = \ln(a)$ does not exist yet).

We begin by specifying the value of the model parameter a, and setting the initial condition x_0 (the zeroth iterate):

```
>> a = 0.8;
>> iter(1) = 1.0;
```

Note that here we have placed our initial condition in the first place in a vector array (`iter(1)`). There is no such thing as a zeroth position in a an array when using Matlab (i.e., `iter(0)` cannot be defined). Next, we create a pair of lists corresponding to the coordinate points x and y respectively. We use the iteration number `i-1` as the x-coordinate and the corresponding iterate `iter` as the y-coordinate. The list corresponding to the x coordinates is quite simple to generate using the recursive properties from the previous section.

```
>> for i=1:21,
X(i) = i-1
end
```

Note that we have assigned a value of `i-1` to each element in our `X` array. We do this because we want to start at the x-coordinate 0. Now, to create the corresponding y list, we make use of the Ricker Model. Again, let's define our function using the inline method.

```
>> RM = inline('0.8*x.*exp(-x)','x')
```

Note that we have placed a '.*' between the first `x` and the exponential function. This implies that we are performing pointwise vector multiplication. A single '*' implies scalar multiplication. Before defining the second list, let's initialize the first y-coordinate in our vector array `Y` as the first iterate. Type

```
>> Y(1)=iter(1)
```

Now, to recursively define the second list type

```
>> for i=1:20,
Y(i+1) = RM(iter(i))
iter(i+1) = Y(i+1)
end
```

Finally, we plot the list of coordinates:

```
>> X = 0:20
>> plot(X,Y,'*')
```

As you can see, with $a = 0.8$, the population dies out, at least with the initial condition $x_0 = 1.0$. The numerical result is consistent with the results of our linear stability analysis. You can check that the population dies out with other initial

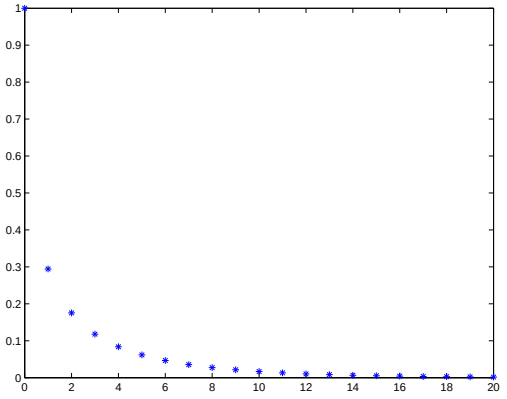

Figure 8.6.

conditions as well, by making an appropriate change on the line specifying the initial condition, and pressing the return key a few times to re-execute that line and the following lines.

8.2.1 Procedures (Functions) in Matlab: m-files

We're interested in seeing the behaviour of the model for different values of the parameter a. We can continue making appropriate changes in the lines we already have on the screen. Instead of changing the value of a over and over, it is more elegant to define a function which plots the trajectory for a given value of a. Here's how we define such a function. Open an m-file and type the following (note that Matlab ignores lines that begin with "%", so we can use these to include comments in our Matlab program):

```
    % defining a recursive function in an m-file

    function y=plot_traj(a)
    RM=inline('a*x.*exp(-x)','a','x')
    % Note that we are using an inline function.  Sometimes it's easier
to do this.

    % Collecting list of x-coordinates
    for i = 1:31,
    X(i)=i-1
    end

    % Collecting list of y-coordinates
```

```
for i = 1:30,
Y(i+1)=RM(a,iter(i))
iter(i+1)=Y(i+1)
end

y = plot(X,Y,'*')
```

Now, save your m-file (as plot_traj.m) and close it. Type the following into the command window:

```
>> plot_traj(0.8)
>> plot_traj(1.0)
>> plot_traj(5.0)
>> plot_traj(8.0)
>> plot_traj(13.0)
>> plot_traj(14.5)
>> plot_traj(20.0)
```

You should observe that the qualitative behaviour of the Ricker model changes drastically when the value of the parameter a is changed, as should be expected from the results of the linear stability analysis. Before we investigate interesting types of behaviour, we will dissect the above Matlab code.

1. The keyword "function" indicates that we are going to define a function.
2. The name of the function, follows the "=". In parentheses, we give a list of input parameters, which are separated by commas.
3. Next, we define local variables. The values for these variables are only known to this particular procedure and they cannot be used outside of this procedure.
4. Then the Matlab commands follow, which define the action of the function.
5. The "end" command ends the definition of a function.

Exercise 8.2.2 As our function stands right now, the initial condition of the iteration is set by the programmer (you) within the function. Modify the procedure so that the initial condition for the iteration also can be specified by the user. You should test your function with a variety of initial conditions.

Exercise 8.2.3 The "signum", which is thought of the "sign" of a number, can be defined as:

$$\operatorname{sgn}(x) := \begin{cases} -1, & \text{if } x < 0, \\ 0, & \text{if } x = 0, \\ 1, & \text{if } x > 0. \end{cases}$$

Write a Matlab function to define the signum function in this way. Note that Matlabs own name for the signum function is "`sign`", so you must give it a different name.

8.2.2 Feigenbaum Diagram and Bifurcation Analysis

We would like to understand the changes in the qualitative behaviour of the Ricker model. We will focus on the steady-state behaviour of the model. We ask the following question. For a given value of the model parameter a, what is the steady-state behaviour of the model? We use the numerical capabilities of Matlab to help us answer this question, and create a Feigenbaum diagram. (also known as an orbital bifurcation diagram), as we did for the rescaled logistic equation in Section 2.2.3 (see Figure 2.13).

To accomplish this, we do the following. For each value of a of interest, we ask Matlab to iterate the model a large number of times so that we can be sure that we have reached steady state. Then we throw out most of the iterations, and only save the last few. Finally, we plot the iterations that we kept (a is on the x-axis, and the value of the population at steady state is on the y-axis). If the model converges to a fixed point for a particular value of a, then the points for that value of a will all be plotted on top of each other. If the model converges to an orbit of period 2, then there will be two distinct points for that value of a, and so on. You will create the Feigenbaum diagram for the Ricker model in two steps via the following two exercises.

Exercise 8.2.4 Define a function which iterates the Ricker model a total of 600 times for a particular value of a. The arguments of the function should be the parameter a and the initial condition x_0. Your procedure should return a list (for example, `mylist`) that contains the coordinates for the points that will appear in the Feigenbaum diagram. Note that you do not need to create coordinates for the first 500 iterates, only for the last 100 iterates. Hint: Each coordinate in the list should be of the form `[a,iter]`, not `[i,iter]`. Make sure your program returns the values in the list.

Exercise 8.2.5

(a) Define a function which collects and plots all coordinates for the Feigenbaum diagram for values of a from 0 to 10 in steps of 1. (The easiest way to do this is to expand on the function you used in the previous exercise).

Hint: you may want to store values for a in a matrix A, where each row in the matrix corresponds to a list of constant a values, and the number of columns corresponds to the iteration number. Matrix arrays are defined in a similar fashion as vector arrays. As an example, typing A = [1 2; 3 4] creates a 2 x 2 matrix A with elements 1 2 in the first row and elements 3 4 in the second row. Also, A(i,j) corresponds to the element in the i^{th} row and the j^{th} column of matrix A. From the example given, typing A(1,2) should give back an answer of 2. It may also be helpful to note that, if you would like to appeal to the entire first row of elements in the matrix, type $A(1,:)$.

(b) If you're confident in your results from the previous exercise, repeat the exercise with smaller steps of a to obtain more details in the diagram (be careful not to make the steps too small though, or you will have to wait a long time

to view the result).

(c) Recall that the linear stability analysis of Exercise 8.2.1 predicted a *transcritical bifurcation* at $a = 1$ and another bifurcation at $a = e^2$. How are the bifurcations manifested in the Feigenbaum diagram(s) you produced? What kind of bifurcation occurs at $a = e^2$?

The diagram you produced in the previous exercise shows a typical route to chaos, namely the period-doubling route. For small values of a, we find one stable fixed point. As a increases, the fixed point loses its stability at a *period-doubling bifurcation*, and we obtain a stable orbit of period 2 instead. As a increases further, the period is doubled again to 4 and further to 8 and so on. We will analyze this process in more detail below.

Recall that a 2-cycle is defined by the values u and v with

$$u = f(v), \qquad v = f(u).$$

If we apply f twice, we get

$$u = f(f(u)), \qquad v = f(f(v)).$$

This suggests that we should consider the second-iterate function $f(f(.))$ instead of $f(.)$, since fixed points of this function correspond to a 2-cycle.

We will find the iterates visited during the 2-cycle for $a = 8$, and verify that they correspond to the fixed points of the second-iterate function. Before defining the second-iterate function, let's define the original function.

```
>> RM = inline('8*x.*exp(-x)','x')
```

Now, define the second-iterate function as follows:

```
>> RM2 = @(x) RM(RM(x))
```

Note that we have defined our new function in a new way (not using the inline command). This new method is used to create what are called 'handles' on functions, so that Matlab does not 'forget' what the original function was when defining a new function in terms of the original. This tool is very useful when defining any sort of composite function. It is also a useful tool when you want Matlab to remember a parameter. For example, consider the function $f(x) = a\sin(x)$. To create a function handle to this function, where $a = 0.9$, type

```
>> a = 0.9;

>> f =@ (x) sin(a*x)

>> f(0)

>> f(pi/2)
```

Try defining the same function using the inline method. Notice that you will receive an error when you try to evaluate the function.

Exercise 8.2.6

(a) Estimate the values of the iterates visited during the 2-cycle observed for $a = 8$ from the bifurcation diagram.

(b) Obtain accurate values of these iterates by using your `plot_traj` procedure.

(c) Plot $f(f(x))$ for $a = 8$, together with the diagonal line $y = x$, and verify that two of the fixed points of the second-iterate function correspond to this same 2-cycle.

In the above exercise, you should have found that the higher iterate of the 2-cycle lies between 2.6 and 3. We can solve for this iterate, which is one of the fixed points of the second-iterate map, as follows:

```
>> fsolve(@(x)RM2(x)-x,3)

solution =

    2.7726
```

Note that `fsolve` finds only one solution at a time.

Exercise 8.2.7 Find the other nonzero solution we expected, and verify that these two nonzero solutions together correspond to the 2-cycle at $a = 8$. (verify that $f(u) = v$ and $f(v) = u$, where u and v are the functions found).

8.3 Stochastic Models with Matlab

Matlab has several features for simulating stochastic processes. In particular, it has random number generators.

To get Matlab to uniformly choose a random number between 0 and 1, we use the `rand` command. In the command window, type

```
>> rand(1)
```

Now try doing this again.

```
>> rand(1)
```

You should notice that you get two different random numbers. You can try this over and over to get as many random numbers as you would like. (Note: the 1 in the argument of the rand command means that you want the computer to return a *scalar* random variable.)

Now let's try something a little more complicated. We consider an individual jumping equal-sized distances along a horizontal line. At each time step, the individual must make a decision to jump to the right or the left. Suppose that, at each

time step, the probability an individual jumps to the right is `R` and the probability the individual jumps to the left is `L = 1 - R`. Furthermore, we assume that the event of jumping to the right is given by a Bernoulli random variable with probability $p =$ `R`. Recall that a Bernoulli random variable is a discrete random variable that takes on the value 1 with probability p and the value 0 with probability $1 - p$.

Exercise 8.3.1 To make life more interesting, let's consider the Bernoulli random variable to be a special case of the Binomial random variable. What is the special case, and why does this work? (Hint: follow the few steps of code below and generate some binomial random numbers)

Try the following (here R = 0.5):

```
>> for i = 1:20,
rnum(i) = binornd(1,0.5)
end
```

The first argument in the `binornd` command tells the computer to generate a Bernoulli random variable (i.e., a Binomial random variable that can only have the values zero and one). The second argument in `binornd` indicates the 'probability of success' in the Binomial random variable (i.e., the probability of having the Bernoulli random variable take the value one).

You should see that our function `rnum` randomly returns a 1 with probability `R = 0.5` and a 0 otherwise, as desired. Now, if we can make a transformation so that we either obtain a 1 with probability R and a -1 otherwise, then we can use the result to keep track of the location of the individual. We will call the transformed function `jump`. It is defined as follows:

```
jump = inline('2*binornd(1,0.5) - 1','x')
```

You will notice that `jump` is a function that requires you to evaluate at a particular value of x. However, this value can be considered a dummy variable, since the function doesn't actually depend in it. Therefor, you can let x be whatever you want when you evaluate this function. (For example, it is just as likely that `jump(1)` will give you the same answer as `jump(500)`). Try testing it out for yourself!

Exercise 8.3.2 Explain the transformation, and evaluate the `jump` function a few times. Did it work?

Next we will define a way to simulate a sequence of jumps at successive time steps. In an m-file type the following:

```
loc = 5;
numjumps = 10;

for i = 1:numjumps
loc = loc +(2*binornd(1,0.5) - 1)
end
```

Save the m-file as **location.m** and run it. By recursive evaluation of `loc`, we have simulated a random walk made by an individual. Here, we let the individual begin at location 5, and then we let the individual make 10 jumps.

Exercise 8.3.3 Let's visualize the random walk:

(a) Create a function called `plot_sim` that plots location versus time for an individual who makes successive random jumps. Each jump is one unit to the right with probability `R`, and otherwise one unit to the left. The arguments of `plot_sim` should be `R`, the probability of jumping one unit to the right, `a`, the initial location of the individual, and `numjumps`, the number of jumps the individual makes. Of course, you should incorporate the ideas from the m-file defined above.

(b) Determine the effect of varying the value of `R`. Run your function several times, keeping the initial location constant at `a=0`, and the number of jumps at `numjumps=25`, but varying the value of `R`. Explain the results (it is a good idea to run the simulation several times for each value of `R`).

We won't always need to generate a graph of the random walk. For example, we will define a procedure in an m-file that returns the location of the individual after `numjumps` steps directly, and try it out for `numjumps=25` (Note that we are now starting at position 0). In a new m-file type the following:

```
loc = 0;
numjumps = 25;

for i = 1:numjumps
loc = loc+(2*binornd(1,0.5)-1);
end
finalloc = loc
```

Save your m-file as **finallocation.m** and run it. Now we will rework our code so that we can evaluate the above procedure many times so as to empirically deduce the probabilities associated with different endpoints after a fixed number of jumps (25 is a nice number to start with), always starting at the same location. We evaluate the procedure 500 times, and save the results in an array called `finalloc`. Update the m-file **finallocation.m** to look like the following:

```
numjumps = 25;
M=500;

for j = 1:M
loc = 0;

for i = 1:numjumps
loc = loc+(2*binornd(1,0.5)-1);
end
```

```
finalloc(j)=loc;

end
finalloc
```

Save your m-file and run it. To generate a histogram of the results, we use the `hist` command. In the command window type

```
>> i=-500:500

>> hist(finalloc,i)
```

Exercise 8.3.4 In class, we learned that if the random walk is unbiased (`R=0.5`), the distribution of individuals after a large number of time steps should be approximated by a Gaussian distribution with mean $\mu = 0$ and variance σ^2 equal to the number of time steps (or the number of jumps, `numjumps`). We will test the theory here.

(a) Use appropriate Matlab commands to evaluate the mean and variance of `finalloc` and compare the results with theory.

(b) Create a new plot of the appropriate Gaussian (using the theoretical values for the mean and variance) and assign this plot to the variable `graph2`.

(c) Plot `graph1` and `graph2` on the same set of axes. How good is the approximation?

(d) Repeat the above with values of `numjumps` larger and smaller than 25, and compare.

Exercise 8.3.6 Theory also tells us that the *mean squared displacement* of individuals undergoing the random walk processes should increase linearly with time if the walk is unbiased (`R=0.5`). Let $x_i(t)$ be the location of individual i at time t. Then test the theoretical prediction by using Matlab to plot $(\sum_{i=1}^{M}(x_i(t)^2))/M$ versus t for $t = 1, 2, 3, 4, 5$. Test your code with a small value for `M`. When your code works, you will want to use a relatively large value for `M`. What is the slope of the line? Explain the value of the slope using the theory discussed in Chapter 5.

8.4 Ordinary Differential Equations: Applications to an Epidemic Model and a Predator-Prey Model

In a previous section, we studied a model that describes population growth in terms of discrete time intervals. However, in some cases it is important to know the state of the system at any time. This can be achieved using differential equations. In this section, we focus on ordinary differential equations. Recall that an autonomous ordinary differential equation can be written as

$$\frac{d}{dt}x(t) = f(x(t)),$$

where tR, xR^n, and $f : R^n \to R^n$. The left-hand side, $dx(t)/dt$, is the rate of change of the state variable $x(t)$ and the right-hand side, $f(x(t))$, summarizes all factors which cause a change in $x(t)$ (for example, birth, death, creation, removal, etc). We will investigate two specific models, one describing the time course of an infection in a population, and the second describing a simple predator-prey system.

8.4.1 The SIR Model of Kermack and McKendrick

To obtain the basic epidemic model of Kermack and McKendrick, we split the population into a class S of susceptible individuals, a class I of infective individuals, and a class R of recovered or deceased individuals. First, we consider the transition from class S to class I. Not every encounter between a susceptible and an infective individual leads to infection of the susceptible. We consider a small time step Δt, and we introduce the parameter β, which measures the average number of effective contacts per unit time per infective individual (an effective contact is one in which the infection is transmitted from an infective to a susceptible individual). When an infection is successful, the newly infected individual is removed from the class of susceptibles, and added to the class of infectives. Thus, in the small time step Δt, the change in the number of susceptible individuals, ΔS, is

$$\Delta S = -\beta \Delta t S I,$$

and the change in the number of infective individuals, ΔI, is

$$\Delta I = \beta \Delta t S I.$$

Dividing both sides of the equations by Δt and taking the limit as $\Delta t \to 0$ gives

$$\begin{aligned} \dot{S} &= -\beta S I, \\ \dot{I} &= \beta S I. \end{aligned}$$

Next, we consider the transition from class I to class R. Depending on the disease, infectives either recover (here, we assume that individuals recover with permanent immunity to the disease), or they die. Both cases lead to the same model. We assume that the rate of recovery is α. That is, in a time step of Δt, the number of individuals that undergo the transition from class I to class R is $\alpha \Delta t I$. After the limiting process, the full model then reads

$$\begin{aligned} \dot{S} &= -\beta S I, \\ \dot{I} &= \beta S I - \alpha I, \\ \dot{R} &= \alpha I. \end{aligned}$$

Note that R is decoupled from the rest of the system (once a solution for I is known, R is known as well, and R does not feed back onto the equations for S and I). So it is sufficient to study the first two equations of the model. We won't be able to

find an explicit solution, giving S, I, and R as functions of time t. A little later, we will learn how to find solutions numerically. Before we do that, we will look for solutions in the form $I(S)$, that is, I as a function of S. We have

$$\frac{dI}{dS} = \frac{dI}{dt}\frac{dt}{dS} = \frac{dI}{dt}\left(\frac{dS}{dt}\right)^{-1} = \frac{\beta SI - \alpha I}{-\beta SI} = -1 + \frac{\alpha}{\beta S}.$$

We can solve this equation by hand via separation of variables, but we choose here to use Matlab. First, we tell Matlab about the differential equation.

We begin by defining a function called $odefun$ (in an m-file) as the right-hand side of the differential equation as follows:

```
function Iprime = odefun(S,I)

Iprime= -1 + a/(b*S)
```

Note that we have changed our variables α and β, to a and b respectively (for ease in programming). Now, save your m-file as **odefun.m** and close it. Next we will solve our ODE by setting up the differential equation using one of Matlabs ODE solvers. Before doing this, type `help ode45`. You will find the syntax for ode45 is:

```
[T,Y] = ODE45('ODEFUN',TSPAN,Y0);
```

where T is the independent variable(s) and Y is the dependent variable(s). Tspan = [T0 T1] is the range of the T you would like to solve over, where T0 is the initial T, and T1 is the final T. Y0 is the initial condition, and ODEFUN is the name of the ode function you saved in your m-file (odefun.m).

Before solving the ODE, go back to your m-file and specify values for a and b. Specifically, edit your file to look like the following:

```
function Iprime = odefun(S,I)

a = 0.04

b = 0.0002

Iprime= -1 + a/(b*S)
```

Now, in the command window type

```
>>[S,I] = ODE45('odefun',[1,1000],1);
```

you should notice a list of values in the command window. These values correspond to the derivative of the function for the prescribed range of S. To view the plot of the derivative, type,

```
>> plot(S,I)
```

Congratulations. You have plotted a trajectory in the $S - I$ phase plane that goes through the point $(S, I) = (1, 1)$. What direction should the arrow be on the trajectory (HINT: does s increase or decrease with time?).

Next, we discuss a second method for solving derivatives using the **symbolic math toolbox**. Note that is the previous method we described a way of solving for the ODES by specifying values for our parameters a and b, as well as an initial condition. Here, we make use of the `dsolve` command. This will allow us to solve for the ODE in terms of unspecified parameters.

```
>>I = dsolve('DI = -1 + a/(b*S)','S')
```

If you would like, you can also include the condition that the trajectory go through $(S, I) = (1, 1)$ by typing:

```
>> I = dsolve('DI = -1 + a/(b*S)', 'I(1)=1' ,'S')
```

Exercise 8.4.1

Let I_0 denote a number of newly infected individuals in an otherwise susceptible population S_0.

(a) Determine the constant $C1$ such that $I(S_0) = I_0$.

(b) Create a Matlab function for $I(S)$ with the appropriate constant of integration.

(c) Choose some values for the parameters a, b, I_0, and S_0, and plot the function $I(S)$ (e.g., choose $a = 0.04$, $b = 0.0002$, $I_0 = 10$, $S_0 = 990$). What is an appropriate domain for your graph (think about the maximum and minimum number of susceptible individuals during the infection)? Does the infection progress as you expected?

We would really like to find solutions of the model as a function of time. As mentioned before, we will not be able to do so explicitly. Solving a system of ODES explicitly in Matlab is much more simple then solving the system numerically. For example, consider the system

$$\frac{dx}{dt} = 0.3x - y$$
$$\frac{dy}{dt} = 0.4y.$$

To solve for this system with initial conditions $y(0) = 0$ and $x(0) = 1$, using the `dsolve` command, type:

```
>> [x,y] = dsolve('Dy=0.4*y,Dx=0.3*x-y','y(0)=0','x(0)=1','t')
```

You should get back the answer

```
x =

      exp((3*t/10))

y =
```

```
0
```

To solve for a solution numerically, we must go back to our original method for solving odes and use one of Matlabs ODE solvers. Let's open a new m-file and create a function to numerically solve and plot the system of ODES that we are interested in. Your m-file should look like the following.

```
function dy = odefun2(t,y)

a = 0.04

b = 0.0002

dy = [-y(1).*y(2)*b; y(1).*y(2)*b-y(2)*a];
```

Here, we use the variable `dy` to denote a 2-dimensional vector, whose first component, `y(1)`, corresponds to the differential equation for for S, and whose second component, `y(2)`, corresponds to the differential equation for I.

Save your m-file as **odefun2** and close it. Now we will find and plot an implicit solution for the system. In the command window, type

```
>> [t,y]=ode45('odefun2',[0,100],[990,10]);

>> plot(t,y(:,1))
```

Here, we have solved for the system with initial conditions $S(0) = 990$, and $I(0) = 10$ (recall that variable S corresponds to `y(1)`, and variable I corresponds to `y(2)`). Note that this is only the plot for S vs t. To get the plot for I vs t type

```
>> plot(t,y(:,2))
```

To view both plots together type

```
>> plot(t,y(:,1),t,y(:,2))
```

Similarly, to plot the phase portrait, S vs I, type

```
>> plot(y(:,1),y(:,2))
```

Exercise 8.4.2

Experiment with different values of the model parameters and initial conditions. Try several cases, and verify that there is an epidemic outbreak when $R_0 = S_0 b/a > 1$ and no outbreak when $R_0 < 1$.

8.4.2 A Predator-Prey Model

We denote the size of the prey population at time t by $x(t)$ and the size of the predator population by $y(t)$. We assume that in the absence of a predator the prey population approaches its carrying capacity as modelled by the logistic law (Verhulst's growth model),

$$\dot{x} = ax(1 - x/K),$$

where $a > 0$ is the per capita growth rate and $K > 0$ is the carrying capacity. We also assume that the predator population cannot survive without prey, and model this with an exponential decay equation,

$$\dot{y} = -dy.$$

For each encounter of a predator with a prey there is a certain probability that the prey will be eaten. We apply the law of mass action model and represent removal of prey by predators as $-bxy$, where $b > 0$ is a rate constant. Several successful hunts by the predator will result in the production of offspring. This is modelled with a term cxy, where $c > 0$ is a reproduction rate. In general, $b \neq c$ (why?). We obtain the following predator-prey model:

$$\begin{aligned} \dot{x} &= ax\left(1 - \frac{x}{K}\right) - bxy \\ \dot{y} &= cxy - dy. \end{aligned}$$

We nondimensionalize this model by letting $\tau = at, \kappa = (cK)/d, g = d/a, u = (cx)/d$, and $v = (by)/a$, to get

$$\begin{aligned} \dot{u} &= u\left(1 - \frac{u}{\kappa}\right) - uv, \\ \dot{v} &= g(u - 1)v. \end{aligned}$$

The steady states of the system satisfy the following algebraic system:

$$\begin{aligned} u\left(1 - \frac{u}{\kappa}\right) - uv &= 0, \\ g(u - 1)v &= 0. \end{aligned}$$

We solve for the steady states with the `solve` command, for the specific case $\kappa = 2$ and $g = 1$:

```
>> eqn=solve('u*(1-u/2)-u*v=0', '1*(u-1)*v=0', 'u','v')


eqn=
    u:  [3x1 sym]
    v:  [3x1 sym]
```

To view the solution type,

```
>> [eqn.u,eqn.v]
```

You should get three steady states, [0,0],[2,0], and [1,1/2]. We wish to determine the stability of each of the steady states. For this, we need the Jacobian matrix of the right-hand side of the system of equations, and it is

$$Df(u,v) := \begin{pmatrix} 1 - 2\frac{u}{\kappa} - v & -u \\ gv & g(u-1) \end{pmatrix}.$$

We need to evaluate the Jacobian matrix at each of the three steady states and then determine its eigenvalues to deduce their stability.

The `jacobian` command gives us exactly what we are looking for (again, we include the actual values of κ and g.):

```
>> syms u v

>> desys = [u*(1-u/2)-u*v, 1*(u-1)*v];

>> w = [u,v];

>> jack=jacobian(desys, w);

>> pretty(jack)
```

$$\begin{bmatrix} 1 - u - v & -u \\ v & u - 1 \end{bmatrix}$$

The first command, `syms`, defines the variables you are going to be using. The last command, `pretty`, puts the matrix in a nice form.

We define the three matrices `m1`, `m2`, and `m3` to be the Jacobian matrix evaluated at the three steady states, respectively. The first steady state corresponds to the point (0,0). Here, we type:

```
>> m1 = subs(jack,[u,v],[0,0])
```

Let's do the same for the next two matrices.

```
>> m2 = subs(jack,[u,v],[2,0])

>> m3 = subs(jack,[u,v],[1,0.5])
```

Next, we proceed to find the eigenvalues with the `eig` command:

```
>> ev1 = eig(m1)

>> ev2 = eig(m2)

>> ev3 = eig(m3)
```

Exercise 8.4.3 Verify that the steady states $(0,0)$ and $(2,0)$ are saddles, and that $(1,0.5)$ is a stable spiral.

Note that if you also are interested in eigenvectors for each eigenvalue, then you can type, for example

```
>> [V,e]=eig(m3)
```

This will give you two matrices. The columns of the first matrix correspond to the eigenvectors, and the entries of the second matrix correspond to the eigenvalues.

We can go further and visualize the phase portrait. For this, we first need to solve the system of differential equations. Since an explicit solution cannot be found, we will again need to rely on one of Matlab's ode solvers. In an m-file type:

```
function dw = odefun3(t,w)

g=1

k=2

dw=[w(1)*(1-w(1)/k)-w(1)*w(2);g*(w(1)-1)*w(2)];
```

Save your m-file. Now, in the command window type:

```
>> [t,w1]=ode45('odefun3',[0,100],[2,0.8])


>> [t,w2]=ode45('odefun3',[0,100],[0.5,0.8])
```

We now graph the phase portrait defined by the above differential equations corresponding to the two different initial conditions described above.

```
>> plot(w1(:,1),w1(:,2),w2(:,1),w2(:,2))
```

Try including many more initial conditions! The initial condition for the first trajectory, corresponding to the variable $w1$, is [2,0.8], and the initial condition corresponding to the second variable $w2$, is [0.5,0.8] (Note: the labels for variables $w1$ and $w2$ are arbitrary. You can call them whatever you want!).

8.5 Partial Differential Equations: An Age-Structured Model

In this section, we consider the age-structured model developed in Section 4.2. We let $u(t, a)$ denote the density of females with age a at time t.

To derive an evolution equation for $u(t, a)$, we consider the population after a small time increment Δt. The change in the number of individuals between the ages of a and $a + \Delta a$ is given by

$$u(t + \Delta t, a) - u(t, a) = u(t, a - \Delta a) - u(t, a) - \mu(a)u(t, a)\Delta t.$$

The first term on the right-hand side of the equation represents the number of females progressing *from* the previous age class, the second term represents the number of females progressing *to* the next age class, and the third term represents the number of females that die, with $\mu(a)$ being the age-dependent death rate. Dividing the equation by Δt, and applying the usual limiting process, we obtain

$$\frac{\partial u(t,a)}{\partial t} = -\frac{\partial u(t,a)}{\partial a} - \mu(a)u(t,a),$$

or

$$\frac{du(t,a)}{dt} = \frac{\partial u(t,a)}{\partial t} + \frac{\partial u(t,a)}{\partial a} = -\mu(a)u(t,a).$$

Note that since age and time progress at the same rate, we have $\Delta a = \Delta t$ and $da/dt = 1$.

You may recognize the above equation as a transport equation or convection equation. The partial derivative with respect to a is the transport term, and represents the contribution to the change in $u(t,a)$ from females getting older (and the velocity with which females age is 1).

We have a first-order partial differential equation, and we need two conditions to complete the model. In particular, we need boundary conditions at $a = 0$ and $t = 0$.

The distribution $u(0,a)$ represents the initial age distribution and can be any nonnegative function. The distribution $u(t,0)$ represents the newborns, and it is determined by the biology as follows:

$$u(t,0) = \int_0^\infty b(a)u(t,a)\ da,$$

where $b(a)$ is the age-dependent reproduction rate. Thus, the integral represents the total number of newborns at time t.

Exercise 8.5.1 For most populations, newborn individuals are not immediately capable of reproduction. So it is natural to expect the lower limit of integration to be a number bigger than 0. Similarly, for some species, the female population stops reproduction after a certain age. Discuss the limits of integration in this context. Why can we integrate from 0 to ∞?

Matlab is capable of handling partial differential equations, as it has built in PDE solvers. However, to solve this partial differential equation, we are going to discretize our model, and use Matlab to give us a numerical approximation to the solution. This method will allow us to get a better understanding of how these types of solvers work.

For ease of discussion, assume discrete time steps of one year, and let

$u_i^j :=$ number of individuals with age i at the beginning of year j.

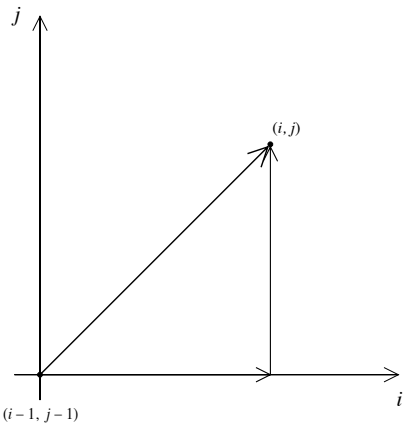

Figure 8.7. *Schematic for the discrete approximations of the time and age partial derivatives.*

We choose the discrete versions of the two derivatives in our model according to the schematic shown in the Figure (8.7):

$$\frac{\partial u(t,a)}{\partial t} = u_i^j - u_i^{j-1},$$
$$\frac{\partial u(t,a)}{\partial a} = u_i^{j-1} - u_{i-1}^{j-1},$$

so that the discrete version of our model becomes

$$u_i^j = u_{i-1}^{j-1} - \mu_i u_{i-1}^{j-1} \text{ for } i \geq 1$$
$$u_0^j = \sum_{i=0}^{n} b_i u_i^{j-1}.$$

Note that we have accounted for a maximum age of n (for human populations, we can safely take $n = 100$).

Defining arrays: a refresher

We are going to develop a numerical simulation of the discrete model. For programming purposes, it will be convenient to use arrays. Recall that, to create the array V = [0, 1, 4, 9, 16, 25], we type

```
>> for i = 1:6,

V(i) = (i-1)*(i-1)
```

```
end
```

For the discrete age-structured model, the age classes run from 0 to n, and so arrays with indices that run from 1 to $n+1$ will be most convenient (note that all arrays in Matlab start with the index 1, not 0). Just as before, you can use the **shift + enter** command to get to a new line when using the command window, so as not to evaluate before you finish writing the code for the loop.

Creating the simulation for the age-structured model

For our numerical simulation, we begin with a population that is equally distributed over all age classes. We set up an array for u_i^j, and assign a population of 1 to each age class (obviously, we cannot have a fractional number of individuals, so you should scale these numbers, that is, $u_i^j = 1$ means 1 million individuals in the corresponding age class, for example). First, open an m-file and type the following

```
function y = evolve(N)
```

Here, we have called our function "evolve". The variable `N` will be used as input later (when we run our simulation) and corresponds to the total number of evolutions we wish to run our simulation. Each evolution corresponds to 1 year. We now define the initial population in each age class i by typing (in the same m-file)

```
for i=1:101
pop(i)=1;
end
```

To initialize a loop that evaluates the population distribution over a given number of evolutions j type

```
for j=1:N
```

(Note that this loop will not be closed until the very end of the m-file). Next, we set up an array containing the birth rate for each of the age classes. For the time being, we assume a birth rate of 7% between the ages of 20 and 35. Again, in the same m-file write

```
for i=1:101
birth(i)=0;
end

birthrate = 0.07
for i=21:36
birth(i)=birthrate;
end
```

Similarly, we set up an array containing the survival rate for each of the age classes. We assume a death rate of 1% for each age class, equivalent to a survival rate of 99%:

```
survivalrate = 0.99
for i=1:101
survival(i)=survivalrate;
end
```

To let our population evolve over time, we define a loop that calculates the population in year $j+1$ (`newpop`) from a population in year j (`pop`) (note that we start with the population distribution denoted by the array variable `pop`, and use it to calculate a new population 1 year later). First type

```
for i=2:101
newpop(i)=survival(i)*pop(i-1);
end
```

Next, to determine the number of newborn (age class 1), type

```
for i=1:101
A(i)=birth(i)*pop(i);
end
newpop(1)=sum(A);
```

Here, we have made use of the `sum` command. This command is used to sum together all elements in an array. Now, we return the new population by typing

```
for i=1:101
pop(i)=newpop(i);
end
```

Now we close the loop corresponding to the year j.

```
end
```

Finally, we write our final line of code that will be used to plot the number of individuals in each age class i of a population and year `N`:

```
i=1:101
plot(i,pop,'*')
xlabel('Age Class')
ylabel('Density')
```

Save your m-file as **evolve.m** and close it.

Running the simulation

Now we have defined all that we need: an array to contain the number of individuals in each age class in a population, arrays for the birth and survival rates, loops that update the number of individuals in each age class, and a final line of code to plot the result. Let's start the simulation.

First, we plot the initial population. Since N=1, in the command window type

```
>> evolve(1)
```

Now we plot the population after one year:

```
>> evolve(2)
```

We can let the population progress several years by typing the following

```
>> evolve(10)
```

Try letting the population evolve 50 years. Try 200 years!

Exercise 8.5.2 What do you observe? Describe what happens to the population as time progresses.

Exercise 8.5.3 Modify the birth and death rates, and study the behaviour of the population over time. Describe this behaviour in words.