17 Feb 05

Memorial University of Newfoundland

Physics 2820 Matlab Basics Winter 2005

The analysis and interpretation of the data and equations for ocean ecological problems requires that we use computer software. Many different software packages exist, and for many different platforms. Because this application requires a mix of graphics and analysis, we have chosen *Matlab* a software package that is available for most platforms, has a student edition, and which has been used as companion software with many different books and monographs. Readers already familiar with *Matlab* will be able to skim, or skip this section, while readers learning about computational software for the first time may want to take some time to learn how to work with this software. We will quickly cover the basics, not intending a complete or fully comprehensive introduction, moving on to the applications that are of most interest here that is the routines that will enable us to solve the independent and coupled differential equations that we must face. The examples that we will work with will be chosen from ocean ecology and will allow us to further explore some of their characteristics while at the same time learning how to apply *Matlab* in ocean ecology.

The roots of *Matlab* go back to the 1970s and a set of routines, originally written in FORTRAN, that were developed for solving matrix problems. The structure of *Matlab* still displays signs of these roots in that much of the syntax is built around vectors and matrices and the algebra for manipulating them. A reader familiar with linear algebra will likely find the syntax fairly intuitive (although how many times have you heard that about a piece of software only to be disappointed?). *Matlab* is now commercial software, although student versions, at greatly discounted prices, now exist. The original commercial product was fairly simple with an interpreter and minimal graphics. It has since grown to include 3-D graphics, signal processing, wavelets, neural net software, statistics, differential equation solvers, etc. These extra packages are added as scripts, or programs, that the user can look at and modify.

To invoke *Matlab* simply type the name and you will produce the prompt

```
>>
```

At the prompt everything is up to you. Note that *Matlab* is case sensitive so A and a are not the same thing. You also have to watch that you do not use variables that are already defined in *Matlab,* like pi, since this will make things confusing. To leave *Matlab* simply type *quit* or *leave*. There is no graphical user interface built in so it is up to the user to know what they want to do. There is a help facility that allows you to find out how to use commands, simply type the command name and you will get the first group of comment lines which normally tells you how to use the command. So to get help on the *plot* command, you type *help plot.* You can also search for commands; begin by simply typing *help help*

Note that there are two fundamental types of window in *Matlab* the command window, that you type in, and the graphics window. To clear the first window, type *clc,* to clear the second one type *clf.* To abort some process type ^*c (control c).* The first thing that we have to look at is how *Matlab* treats numbers. Note that most math is done in double precision which means that at times *Matlab* can seem slow. Matlab does however permit you to define the number of significant figures that you look at. If you type algebra statements, *Matlab*, will just echo back the result so typing

```
>> (7+4)*3/2
```

produces

```
ans=16.5000
```

Matlab puts the result into a variable labeled ans. You could have created your own variable by typing

```
>> soln=(7+4)*3/2
```

and adding a semicolon (;) at the end would suppress the echo. One could also perform a calculation by defining variables and then performing calculations with those variables

```
>>var1=12.5; var2=17.5;  % here we define the variables
>> soln=var1*(var2**3);  % here we calculate the result
```

In the first line we have put two definitions on the same line, separated by a semicolon; *Matlab* is ok with that. At the end of the line we have put a % symbol that *Matlab* reads as a comment statement. We will see later that such commands can be written together in a file and then used by as a new set of commands and thus comments will be quite useful to us. On the second line, we define our new solution where the algebra now involves multiplication of the first variable times the cube of the second one. The exponent of var2 could be 0.5, that is a non-integer, so we can use exponents to define roots. Performing standard algebra is quite easy.

Being a Matrix language, the structure is in terms of scalars, vectors and matrices. Thus

```
>> a=3.5;          b=4.0;
```

define scalar real variables. One could also define a complex number (and there are a lot of advantages to working with complex variables in *Matlab)* by writing

```
>> d=3.5 + 2.4*j
```

algebra works just like normal so that c= a + b   and d=c*a   give the expected results. In the absence of a semi-colon as a line terminator the result of the calculation will be printed. That may be a good thing to begin with but the normal mode is to add a ; to the end of every line, just to be sure to suppress unwanted things.

But what about vectors and matrices, how does *Matlab* work with them (and what are they)? Vectors and matrices do have a mathematical meaning but for us, here in *Matlab*, vectors are simply rows or columns of numbers while a matrix contains both rows and columns. We will see later that vectors and matrices provide a natural way to express the linear algebra with which we will be working but for the moment let us just explore how to work with vectors and matrices in *Matlab.* Shortly we will explore some of the mathematics that is connected to vectors and matrices.

To define a vector , use square brackets with lists of numbers that can be simply separated or separated with commas

```
>> A=[1.5 3.1  2.1]   % this produces a three element row vector
```

If we then define another such vector, say

>> *B=[1 3 4]* % and then add the two vectors together

>> *C=A+ B* % then we get a new row vector that is

C = 2.5 6.1 6.1

Note that C has three elements as well and that each element of A has been added to the element of B in the same position. This is vector addition and it is what *Matlab* wants to do by default. Of course to perform such addition, each vector must have the same number of elements. The same result could be performed with a column vector. To define the same vectors as a column vectors, then write the same definitions but now use semicolons as separators, telling *Matlab*, to skip a row before filling in new data

>> *A=[1.5; 3.1; 2.1] ; B=[1; 3; 4]; C=A+B;*

The column vector C can be converted to a row vector by writing

>> *C=C';* % this is referred to as letting C equal the transpose of C.

Multiplication of vectors is somewhat more subtle. If you try to multiply C=A*B then you will find an error. If we want to multiply the each element of A by the same element in B then we have to write

>> *C=A.*B;*

where the .* tells *Matlab* to forget matrix rules for the moment and just perform the element by element calculation. If we want matrix rules to apply then we write

>> *C=A*B';*

where the transpose operation is performed so that the number of rows of B will equal the number of columns of B. If you check this result then you will find that C is a matrix with three rows and three columns.
Another way to define a matrix builds upon the earlier syntax for vectors

>> *C=[-1, 0 , 0; 2, 1, 3; 4, 2, 3];*

This produces a matrix with three rows and three columns Here the semi-colons separate the rows of the matrix. If one needs to extend the line to define something then use an ellipsis (…) and keep on going. There is no limit to the number of line extensions that are allowed.

$$C = \begin{Bmatrix} -1 & 0 & 0 \\ 2 & 1 & 3 \\ 4 & 2 & 3 \end{Bmatrix} = \begin{Bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{Bmatrix}$$

The first index gives the row, the second index gives the column number. We can determine the size of our matrix two ways. We can type *whos* and *Matlab* will produce a list of the variables together with the size and data type. Another way is to use the *size* command

```
>> [m,n]=size(C)
```

will return the size of the matrix giving the number of rows (*m)* and columns *(n).* We will use matrix syntax regularly, in several different problems, as it is very convenient for representing groups of coupled or related equations.

One very useful way to generate sequences of numbers is to use the colon operator, thus typing

A=1:5 is equivalent to typing out a=[1, 2, 3, 4, 5];

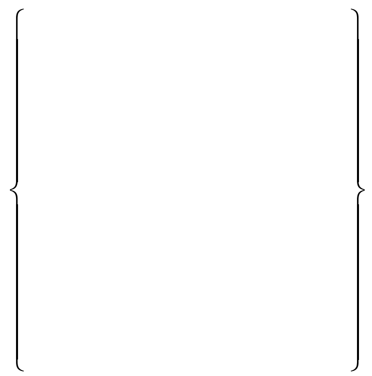
The general form of this syntax is A=A_start:A_step:A_end where A-Start is the beginning value, A_step the step size and A_final the final value. This can be very useful for generating lists of numbers in *Matlab.* :One can also change the step size between elements, making it non-integer, or negative if you like.

One can also use the colon operator to extract things from a matrix thus

```
>> D=C(:,2:3);        % extracts columns 2 & 3 from the matrix that we typed above

>> E=C(2:3,:) ;       % extracts rows 2 & 3 from the same matrix
```

One can also choose to extract specific elements from the matrix

Thus if one has a larger matrix and defines v=[1, 3, 5] then if



If this is the matrix A and we define

C=A(v,2);

Then we extract rows 1, 3 and 5 from the second column of the matrix.

One can also easily compute the transpose A.' and the complex conjugate of the transpose A'.

One can do algebra with matrices fairly easily, any multiplication with a scalar operates on all the elements of the matrix

Thus

$$C = 5 * A = 5\begin{Bmatrix} 1 & 2 \\ 3 & 4 \end{Bmatrix} = \begin{Bmatrix} 5 & 10 \\ 15 & 20 \end{Bmatrix}$$

But if you are multiplying a matrix by another matrix then there are two options, true matrix multiplication and element-by-element multiplication, and you need to be careful to clearly indicate which you want. For straight multiplication, matrix multiplication, simply write C=A*B. For element-by-element multiplication then you must write C=A.*B.

Try all this out with a few examples to see how it works and convince yourself of what *Matlab* is doing. You will discover that there are other aspects of the syntax that we have not discovered, details that are probably best learned by exploration and a certain amount of trial and error.

You can generate arrays of numbers using two commands *ones* and zeros

```
>> ones(3,3)    % will create a 3 by 3 matrix filled with ones
>> zeros(3,1)   % will create a column vector filled with zeros
```

```
>> A=zeros(3,1); B=ones(3,1);
>> C=[A B]
C =

     0     1
     0     1
     0     1
```

creates a two column and three row matrix with zeros in the first column and ones in the second column.

If you want to find out what is in a matrix or perform some simple calculations on them then you can use

```
>>min(A)   % takes the minimum in each column
>>max(A)   % takes the maximum in each column
```

```
>> [y,i]= min(A)   % gives the value and the position
```

```
>> mean (x)   % calculates the arithmetic mean
>> median(x)  % media value
>> std(x) %  calculates the standard deviation
>> var(x) % just the square of the standard deviation
```

Polynomials and roots

A polynomial of degree n has the general form of

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 = 0$$

We can define and find the roots of such functions quite easily in Matlab. There are a lot of physical problems which require the analysis and use of these functions.

If we have a polynomial of the form

$$x^3 - 2x^2 - 3x + 10 = 0$$

in Matlab would be written as

```
>> p=[1, -2, -3, 10] % or you could write p=[1 -2 -3 10]
```

and we could find the roots of this polynomial with

```
>> roots(p)
ans=

-2
2+i
2-i
```

Note that roots will only work with polynomials and so if we want to find the roots of more complex functions then we have to look further. There are also commands in Matlab for the evaluation of polynomials. So

```
>> x=1:30;
>> y=polyval(p,x)  % will evaluate this function for the values of x defined
```

We could also multiply two polynomials using the *conv* function. Thus

```
>> p=[1, -2, -3, 10]; b=[3 5 7 9];
>> c=conv(p,b)
c=

        3     -1     -12     10     11     43     90
```

and we can divide one polynomial into another

```
>> deconv(b,a)  % which will divide b by and yield a quotient q and remainder r
```

Integration

We developed several different approaches to performing numerical integration and developed a cript for integration that uses Simpson's 1/3 rule. In Matlab there are two commands to carry out integration *trapz* and *quad*

If we want to integrate the function f(x)= sinx + x^2 then we would first need to define that function in Matlab. We could integrate it using *trapz*

```
>>x=0:0.1:10;
>> y=sin(x) + x.^2;   % Here we have to define the values of the function numerically
ans=
        335.1875
```

or if we want to integrate using the quad function (more powerful and reliable) then we would

```
>> fx=inline('sin(x) + x.^2')  % here we have to define the function analytically
>>quad(fx,0,10); % to integrate from 0 to 10
ans=
        335.1724
```

Simple Graphics

Plotting in *Matlab* is fairly easy. Simple plot commands are very straightforward and there are a lot of options, most of which we will ignore here, that will permit you to generate pretty much any type of plot that you need. This structure makes it easy to learn how to do the basics very quickly without being caught up in the complexities of syntax and commands that are necessary to produce very fancy plots.

Indeed if you have two vectors x and y to plot one against the other simply type *plot(x,y)* and you are done. For example the data above, the vectors B and C can be plotted with the command *plot(A,B)* with the values for A being plotted on the x-axis and the B values on the y axis. You can add things to the plot like a title *title('title')* or labels on the x and y axis *xlabel('axis_name')* and *ylabel('axis_name')* and after that the plot is done!

Consider a simple example

```
X=-4.0:0.05:4.0              % create the data for the x axis
Y1=exp(-0.5*X).*sin(5*X); %  y data will be an exponentially decaying sinusoidal oscillations
figure(1);                   %  you can have more than one figure window, call this one number one
plot(X,Y1);                  % do the plotting here
xlabel('time'); ylabel('temperature');  % label the x and y axes
hold on;                     % we can overlay additional plots, first hold the plot window
Y2=exp(-0.5*X).*cos(5*X); % define a cosine curve this time
plot(X,Y2);                  % here we do the plotting
grid; gtext('My best plot');  % overlay a grid and put on a text label;
hold off                     % we have done with this plot and so can turn off the hold
```

You can have more than one plot window open at one time using the figure command and so you can have *figure(1), figure(2)* and so on. The active figure will be the last one that you have identified with the *figure* command.

It is also possible to generate multiple plots on a single page. The easiest way to do this is with the subplot command. The syntax for this command is *subplot(mnp)* where *m* by *n* defines the number of rows and columns of plots, thus 3 by 2 gives three rows of plots and two columns for a total of six plots; *subplot(321)* would work with the first of the 12 and subplot(326) with the sixth. To avoid ambiguity with many plots, it is best to separate *(m,n,p)* with commas.

You can also define functions and use them. Note that the functions are pulled from your own directory first, before *Matlab* looks for its own functions, so be careful not to redefine a function like sin or cos. *Matlab* should warn you if you use an inappropriate function name but it may forget and you could cause problems.

```
function v=f101(x)
v=sin(x.^3);
```

and a routine to plot this function would be

```
x=2:0.04:4;   % define the data
y=f101(x);    % invoke the function
plot(x,y);                              % plot and then label the function
xlabel('x stuff'); ylabel('y stuff');
figure(2)                               % let's see what a different approach looks like
fplot('f101',[2,4],10);                 %  use the fplot function defining and plotting the data
xlabel('x stuff'); ylabel('y stuff');
```

There are other ways in which you can present your plots as we have, linear-linear, and various log plots *semilogx(x,y), semilogy(x,y), loglog(x,y)* to change the scaling. One can also change the line styles, the default is solid but one can choose dashes - , dots . by simply typing *plot(x,y,'-')* There are also a number of specific built in plotting features that we have not touched upon, error bars, histograms, stemplots, rose plots and many others. A more complete summary of *Matlab* graphics features can be found in the manuals that come with the software or one of the introductory texts that have been written that make extensive use of the graphics (e.g. Etter 1993; Lindfield and Penny 1995; Biran and Breiner 1999).

Ultimately one can change all the parameters in plotting by simply adjusting the 'handle' of the graphics. We will not explore that here although the manuals provide good guidance on that. To explore that type

```
>> h=plot(x,y)
>> get(h)
```

Each of the handles that exist in *Matlab* can be adjusted with a set command and so you have enormous power to adjust the plot to your liking but there are so many possibilities that you can in practice get lost in playing with the graphics rather than plotting and interpreting your data.

Plotting in two and three dimensions.

Visualizing data in one dimension, e.g. time series of population abundance or temperature, is generally quite straightforward. The graphical issues do become more complicated, however, as we move to higher dimensions. Plotting the surface temperature of a lake is not difficult in principle, but what if the data are not uniformly located about the lake and how do we represent a time series? With just this simple problem of representing temperature on a surface we have opened questions of interpolation schemes, a problem for which long books have been written (e.g. Cressie 1991), while the time series issue leads us into much more sophisticated forms of data analysis or computer movies. Once we move into three-dimensional data such as the temperature structure through the lake, then relatively sophisticated graphics packages are required and great care and effort is required to produce useful and interesting results. *Matlab* is quite capable of all these tasks although for complicated three-dimensional data software packages specifically designed for such problems will produce much better results. Here we shall focus on two-dimensional data, how to contour and plot data like the lake sea surface temperature, and show how to do some simple three-dimensional plotting.

In two and three dimensions, we have a few choices in plotting data; we can plot points, we can use colour maps, we can contour lines, we can use vector arrows (for velocity data) or we can plot trajectory lines in two or three dimensions. We will not cover all the possibilities but will look at the most commonly faced problems, ones which will come up again later in the text.

We begin with some data vectors in x and y

```
>> x=-100: 5: 100;
>> y=x;
```

which we need as a matrix that is created by

```
>> [X,Y]=meshgrid(x,y);
```

This result produces two matrices of equal size for X and for Y that will cover the complete domain for which we can now calculate a third variable z. we will plot a simple hyperbolic parabaloid, a saddle function that has some interesting spatial function.

```
>> z=X.*X _ Y.*Y;
```

There are several different choices for plotting these data that are three dimensional in x, y and z.

```
>> plot3(X,Y,z);          %  plot the lines which by default will change colour
>> mesh(X,Y,z);           %  plot a coloured mesh surface
>> surf(X,Y,z);           %   a mesh surface but now with the colours filled
>> surfl(X,Y,z);          %  a filled coloured mesh surface with directed lighting
>> contour (X,Y,z);       % a contour plot of the surface
>> image(x,y,z);         % a simple 2-D colour fill representation of z (note (x,y) vectors)
>> imagesc(x,y,z);        % same as image but now scaled
```

For the colour plots a scale bar can be obtained by typing *colorbar* after invoking the plotting command. Labels can be produced on the contour plots by using *clabel* after the contour command

```
>> h=contour(X,Y,z);      clabel(h);
```

It is possible to define the contours that will be labeled but for complicated contour plots the labels often get in the way and it may be necessary to locate the labels manually. The *plot3* command will produce a trajectory in space that is most easily illustrated by the following example

```
>> x=0:0.04:40;
>> plot3(sin(x),cos(x),x);
```

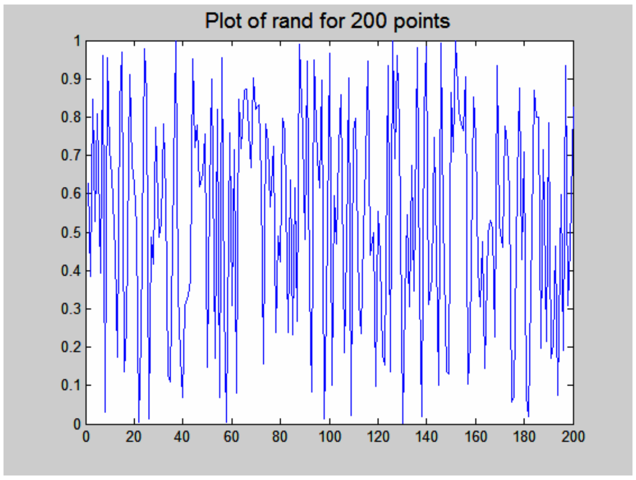
Plotting the trajectory in the phase plane can be particularly illuminating, either a two dimensional line or as a three-dimensional representation as here. Such three dimensional plots are a neat way to explore the relation between variables such as position, velocity and force.

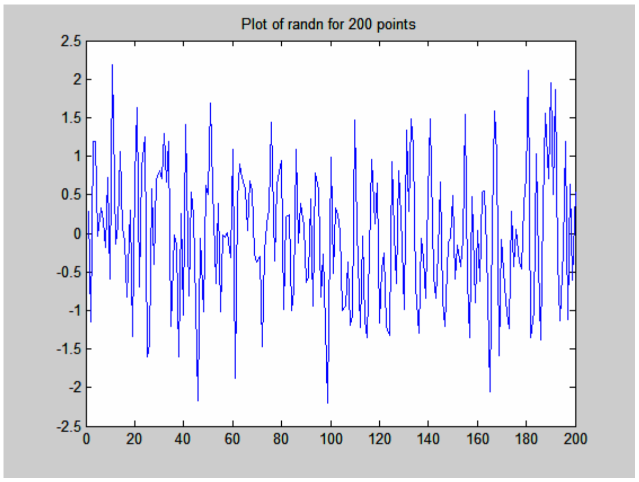
In order to plot points it is necessary that they be on an even grid. In the examples above the x and y points were created on an even grid so it was not a problem as will generally be the case where we can use synthetic computer data. There are however often cases when spatial or time series data are not evenly space. For simple plots, it may be acceptable to leave them as unevenly spaced points and just plot them as points, e.g. time series points of temperature where the sampling time is uneven or there are gaps in the data. But for two and three-dimensional data such an approach will in general be unsatisfactory and it will be necessary to interpolate the data from their irregular locations to an evenly spaced grid. If we can do this in two or three dimensions then we will also be able to do it in one dimension and so we will have a quite general tool that will be useful in other circumstances. As mentioned above, interpolation of irregularly spaced data can be a quite difficult mathematical problem, one that we will only touch upon. Our goal here is to present some simple routines in *Matlab* that will work with simple problems. For difficult spatial mapping problems the reader will have to look elsewhere (Cressie 1991, Daley 1991; Emery and Thomson 1999) . Transferring data from random locations to an even grid is an art but if we are not too fussy then it can be fairly easy.

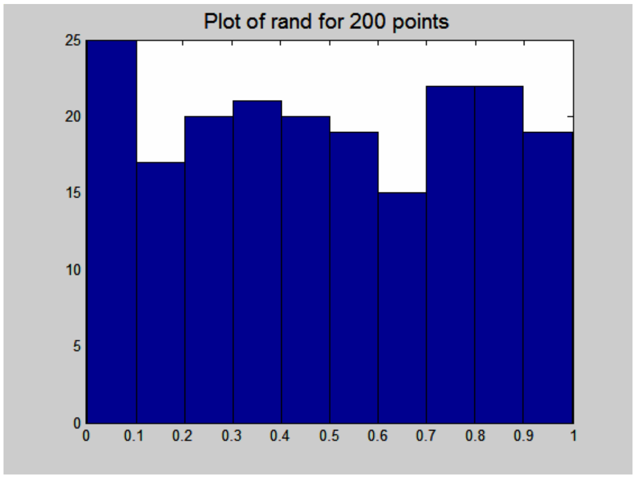
There are two random number generators in Matlab - *rand* and *randn*

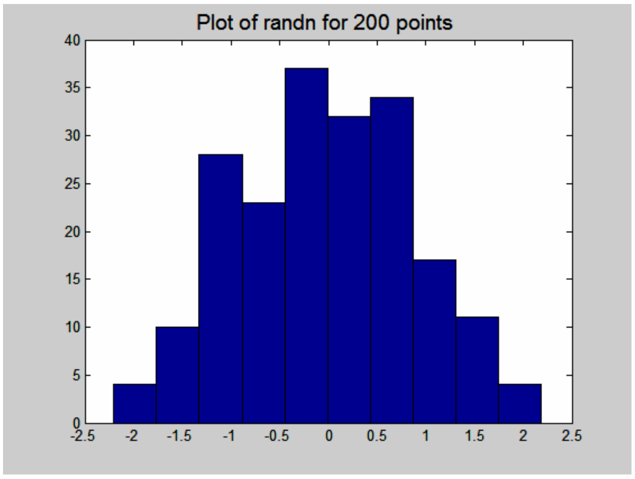
```
>> rand(1,3)
ans =

 0.9501     0.2311   0.6068

>> randn(1,3)
ans =

0.4326   -1.6656   0.1253
```

Every time that you invoke one of these commands you will get a different result. Note that rand produces random numbers between 0 and 1. The other function, *randn*, generates random numbers between + and – infinity centered about 0. You can see the difference by making very many numbers and then plotting the result:









The above plots show that *rand* produces numbers that are evenly distributed between 0 and 1. The other function for generating random numbers, *randn,* produces random numbers with a normal (Gaussian) distribution centered about 0.

We can use these random number generators to produce some three dimensional data (x,y,z)

```
>> x=rand(100,1)*200 - 100;
>> y=rand(100,1)*200 - 100 ;
>> z=x.*^2 – y.*^2;
```

The vectors *x* and *y* are randomly distributed points having values ranging from –100 to 100. To see their distribution in space use *plot(x,y, 'x')* which will produce an xy plot showing the locations of all the data locations. We cannot yet plot the z function because its values are irregularly spaced and most of the functions that we used before to plot the same function, when the data were regularly spaced, required that regular spacing. In order to plot these data we will have to generate a regular grid that we will use to overlay onto our irregular spaced data points. We begin with a set of regular points from –100 to 100 and then use the *meshgrid* command again to generate a matrix of evenly spaced xy points. We then use the command *griddata* to fit a surface to the data. There are three possible ways to interpolate the data: linear (the default), cubic and nearest neighbour.

```
>> x_reg=-100:1:100;              % generate a regular spaced x series
>> [xi,yi]=meshgrid(x_reg,x_reg);        % make a matrix mesh of regularly spaced points
>> zi=griddata(x,y,z,xi,yi);      % grid the irregularly spaced data onto the mesh
>> mesh(xi,yi,zi);                % plot a mesh surface
>> hold on;                       % hold the plot so that we can overlay the points
>> plot3(x,y,z,'ok');             % now overlay the original points as black  circles
```

Three other interpolation functions exist in *Matlab* that provide similar results; *interp1, interp2* and *interp3.* They provide one-, two- and three-dimensional interpolation with somewhat different syntax for the input data. These interpolation routines require that the data be monotonic, that is they must always increase in value, although they can be at irregularly spaced intervals.

Solving linear equations

So far, we have explored a number of different ecological equations, and while we have looked at the solutions to the more simple equations, we have not yet developed any general approaches that will allow us to solve these equations to look directly at the solution. We have techniques for exploring the characteristics of the solutions, e.g. phase plane analysis and dimensional analysis, but we clearly need some tools for generating solutions. Some of the simpler problems do have analytical solutions, solution in equation form, that we can find but most problems, even some that are relatively simple ones do not have analytical solutions and it is only through numerical analysis and the application of computers that we solve these problems.

In this section, we will develop techniques for solving ordinary differential equations. Such equations may have multiple variables, for example predators (*P*) and prey (*H),* but there is only one independent variable. Time is the independent variable in this predator-prey problem. Thus when we write the differential equations the differentials are in time alone and our final solution will be written for the different variables in terms of time alone. We will see later (in Chapter 5) that most physical problems generate partial differential equations, that depend on time and space. Most ecological problems are of course ultimately related to both space and time, and as we shall see have to be written as partial differential equations, but we choose to begin our approach to differential equations with the simple ordinary form.

We begin with the simplest set of equations, coupled linear equations, and see how we can solve them using *Matlab.* Consider the following set of equations

$$\begin{aligned} 10x_1 - 7x_2 + 0x_3 &= 7 \\ 0x_1 + 2.5x_2 + 5x_3 &= 2.5 \\ 0x_1 + 0x_2 + 6.2x_3 &= 6.2 \end{aligned}$$

These are three linear equations in three unknowns (x_1, x_2, x_3). It is linear because none of the independent variables are multiplied by each other or appear with an exponential other then one. Nonlinear equations are almost always more difficult to solve.
We could write the above equations as

$$\begin{pmatrix} \mathbf{10} & \mathbf{-7} & \mathbf{0} \\ \mathbf{0} & \mathbf{2.5} & \mathbf{5} \\ \mathbf{0} & \mathbf{0} & \mathbf{6.2} \end{pmatrix} \begin{pmatrix} \boldsymbol{x_1} \\ \boldsymbol{x_2} \\ \boldsymbol{x_3} \end{pmatrix} = \begin{pmatrix} \mathbf{7} \\ \mathbf{2.5} \\ \mathbf{6.2} \end{pmatrix}$$

which we can write in our generalized matrix notation as

$$\boldsymbol{Ax = b}$$

Where our matrix is **A**, **x** is the column vector containing our variables (**x_1, x_2, x_3**) and **b** is a column vector containing the constants on the right (**7, 2.5, 6.2**). The solution to this is obtained by multiplying this equation by the inverse of the matrix **A**

$$A^{-1}Ax = A^{-1}b$$

The inverse of a matrix by itself just gives us the identity matrix and writing the inverse times **b** as the matrix divide into **b** (which is like dividing by A) then we obtain

$$x = A \backslash b$$

Note that this is **b** divided by **A** and that it matters whether the A or the b comes first in *Matlab*, try it. It does matter whether one writes **A\b** or **A/b** and so one must be careful. The solution to the above problem can therefore quickly be found in *Matlab*, once the matrices and vectors are defined by simply typing

```
>> x=A\b
```

with the result that **x_1=0**, **x_2=-1** and **x_3=1**. This approach will of course only work when the number of equations is equal to the number of unknowns. Note that not all sets of equations are solvable, the information may be contradictory. If there are fewer equations than unknowns, then we have an *underdetermined* set and can only find partial solutions. If there are more equations than unknowns then the system is said to be *overdetermined* and we must resort to numerical methods. The most common numerical approach is the least squares technique, which may be linear or non-linear. Linear least squares looks mathematically quite similar to our direct approach. The goal is to make the square of the difference between the 'fit' and the observations as small as possible, hence the name least squares. For fitting a straight line to data it is easy to show graphically what we mean by this approach (see Figure 5). Our goal is to minimize the distance between the sum of the squared distance between the fitted line and the observation points.

We work with a simple straight line problem to illustrate the least squares approach.

$$y = a + bx$$

where **a** and **b** are constants and we consider that we have *m* observations or trials each of which has some error associated with it. Our goal is to find the **a** and **b** that minimizes the total of the errors $\mathbf{e_i}$

$$y_i = a + bx_i + e_i \quad i = 1,2,....m$$

Normally it is the $\mathbf{y_i}$ that we measure and we assume that the error is all in the $\mathbf{y_i}$ and that there is no error in the $\mathbf{x_i}$ at all

$$y_i^{obs} = (y_i - e_i) = a + bx_i \quad i = 1,2,....m$$

And our goal is to find the coefficients that provide the best fit t o the observations, which we explicitly accept contain some error. We seek to minimize the sum of the squared errors

$$\mathbf{Error} = \sum_{\mathrm{i}} \mathrm{e}_{\mathrm{i}}^2 = \sum_i (y_i - a - bx_i)^2$$

If we write our matrices and vectors as

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \quad A = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_m \end{bmatrix} \quad X = \begin{bmatrix} a \\ b \end{bmatrix}$$

Then one can show (see Lindfield and Penny 1995) that the solution to the coefficients is given by

$$X = (A'A)^{-1} A'Y$$

which is written in *Matlab* as

*>> x=inv(A'*A)*A'*Y*

but we can write this much more simply in *Matlab* as

>> x=A/Y

with the same result. As an example, let us generate a test case using the following function to generate our data and adding some noise to make the result a little more interesting. Note that the function is linear even though we have an exponential in **x**. The equations must be linear in the coefficients for the linear least squares technique to work. Either the equations must be linear from the beginning or it must be possible to transform them into a linear form.

$$y = a + be^x$$

We set the coefficients as **a=1.5** and **b=2.5**. One can generate some test data and then solve with some noise added so that we don't make it too boring

```
>> x=1.1:0.1:11.1;                      % define the x data
>> y=1.5 + 2.5*exp(x) + rand(10,1)/10;  % calculate our observations with some noise
>> A=[exp(X) ones(101,1)]';             % setup the A matrix based upon our equation
>> Coef=A\y                             % find the coefficients allowing Matlab to echo
```

You can play with this script to explore how the result depends upon the amount of noise that is added. As you might expect, the quality of the result degrades as the noise is increased in amplitude.

Representing Differential Equations

The solution techniques that we have looked at so far have been to linear equations, not to differential equations. What happens when we add a derivative term to the equation? How will we solve it then? The basic idea of the solution is quite simple. We remove the differential term by integrating the entire equation. Since integration is reverse differentiation this will have the effect of removing the differential terms. We will see that this is fairly easy to implement for simple problems but even with ordinary differential equations the numerical approach can be quite subtle. One thing to keep in mind as we develop these numerical approaches is the fundamental limitations of numerical solutions. The numerical representation is always an approximation to the exact equations. The quality of that approximation depends upon the character of the problem and our skill but it will always be an approximation. Thus our solutions will be imperfect and sometimes, if we are careless, or the problem is particularly difficult, the solution may be quite poor. Pointing out these concerns at the beginning is not intended to cause you to avoid numerical solutions but it is important to pay attention to the numerical details and not to blindly assume that the numerical result is correct or that any approach will work. As with lots of problems, there are a thousand wrong ways to solve a problem and only a few correct ones.

Differential equations provide us with a model description of a biological or physical system. For the most part, we shall consider what are referred to as initial value problems, that is we have an equation which describes the dynamics of our problems and we have some initial values that determine the starting location for the trajectory. Thus we might know the numbers of predators and prey on a specific date and we want to predict their future numbers. There are analytical solutions to many simple problems but for most of the *real* problems that are of interest, these analytical solutions are inadequate. Today even for some of the simpler problems it is simply easier to apply a numerical technique to solve the problem. But how do we do that?? What are the basics of the approach?

First we must consider how represent differential equations. The computer does not solve the differential equations by looking at the analytical terms, at least not in the approach that we are developing here. So as a first step, the analytical terms, such as the gradient, or derivative of a function f(x) which is *df/dx* must be represented. There are three standard options available: (i) Forward differencing in which we approximate the slope by stepping a distance *h* forward (ii) Backward distance in which we approximate the slope by stepping a distance *h* backward and (iii) Centered differencing in which we average over the forward and backward difference. Numerically these are written as

Forward Difference

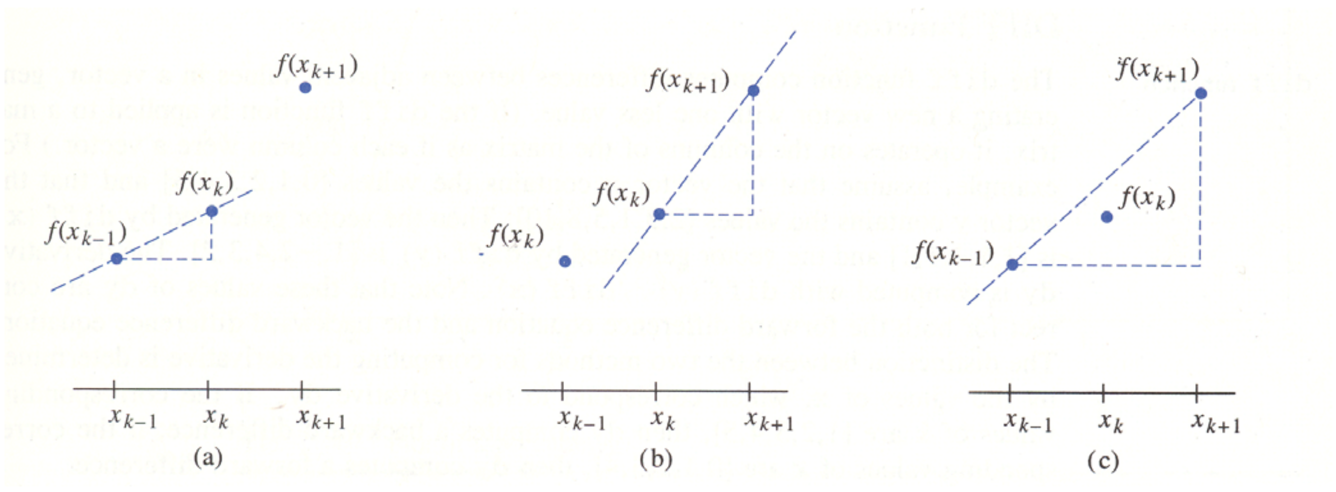
$$\frac{df_0}{dx} \approx \frac{f_1 - f_0}{h}$$

Backward Difference

$$\frac{df_0}{dx} \approx \frac{f_0 - f_{-1}}{h}$$

Centered Difference

$$\frac{df_0}{dx} \approx \frac{f_1 - f_{-1}}{2h}$$



Graphical representation of (a) forward, (b) backward and (c) centered differencing schemes which are used to represent the firs derivative, or the slope, of the function *f(x)*.

It should be clear that each of these approaches involves an approximation to the actual slope at the point of interest. We can improve the accuracy of the representation by making *h* smaller but the smaller that we make *h* the more work that the computer will be asked to do and there will be a point where the computer is not able to solve our problem because it is taking too long taking detailed calculations. Our general goal in formulating the numerical representation of our problem is to make it accurate but we also need to consider efficiency. So there will often be a tradeoff between the two. Here in the world of numerical computing you do not always get to have your cake and eat it too.

One step in guiding our selection of a numerical representation is to consider the error associated with each representation. One can evaluate the error of these different representations (and there are many more) by considering their expansions in terms of a Taylor series (see for example Lindfield and Penny 1995). The first two approaches, which are quite similar and only differ in that one steps forward while the other steps backward, both have the same error that is given by

$$\cong \frac{-h\ f_i^{''}}{2}$$

The centered difference scheme has a quite different looking error

$$\cong \frac{-h^2 f_i^{'''}}{6}$$

Note that the first error depends on the second derivative, $f^{''}(x)$; the second term depends on the third derivative, $f^{'''}(x)$. The other difference between the two terms is that the forward and backward schemes are first order in *h* while the centered scheme is second order in *h.* This difference is very important since it dramatically effects the error of our result as we shrink *h.* If we decrease *h* by 10 then the errors for the first sechemes decrease by 10 but if we do the same for the centered scheme then the error decreases by 100, a dramatic improvement. We don't know how the second and third derivatives compare, although we hope that the third is in general smaller than the second, but since *h* is a small quantity then as *h* gets smaller then we should in general find that centered differencing is better then forward or backward differencing. It may still be that at the start of a calculation that we must begin with forward differencing, this is particularly true of time derivatives, since this is the only way that we can get things moving.

There are of course more complicated schemes and it is natural to ask why we do not try them, should not the errors continue to get smaller as we go to higher order schemes?

$$\left.\begin{aligned}\frac{df}{dx} &\approx \frac{-f_{i+2}+4f_{i+1}-3f_i}{2h} \\[2ex] \frac{df}{dx} &\approx \frac{3f_i-4f_{i-1}+f_{i-2}}{2h}\end{aligned}\right\}$$

Both these schemes can be shown to represent reasonable expansions of the function, however for each of them the error is not really so different from the centered scheme above

$$\frac{h^2 f_i'''}{3}$$

Thus more complicated schemes may not be much more accurate than simple schemes and offer us small reductions in the error but at the expense of substantially increased complexity. Elegance and simplicity in numerical representation are desirable both because they are usually associated with numerical efficiency and also because they make numerical implementation more straightforward. It is usually straightforward to build up the software code for a numerical model, step by step, but the final result can be quite complex and difficult to work with. Simplicity of design can be a definite advantage in working with and debugging such software. It is unfortunate but nearly inevitable that debugging will be necessary at some stage. There is, of course, an entire literature on this subject. It is enough for us to note that (i) that we need to represent analytical functions and (ii) that there are often tradeoffs in our choices.

Solving differential equations

Having learned how to numerically represent the terms in the differential equations, we now have to move towards actually solving these equations. We will begin with some very simple intuitive approaches that will be easy to understand and to implement. It will not be hard to generalize the simple approaches to show how more robust techniques can be developed. The basic idea of solving these differential equations is to integrate them to find the functional solutions. The equations themselves provide the differential representation. Our goal is to remove the differential terms and find the functional representation. Thus if we have a simple differential equation that can be generally written as

$$\frac{dy}{dt} = f(y,t)$$

then our goal here is to find *y(t)* which involves integrating both sides of the above equation. While the idea is straightforward the approach to implementing it is not. As with the numerical representation of the terms, there is no general approach that will provide us with an error free result. Each set of equations will have its own characteristics and we will need to be careful to make sure that our approach is working properly for the problem at hand.

The first technique that we will look at is called Euler's method. All the others take a similar form so we shall understand the basics if we see how to use this method. We begin by writing the forward difference scheme

$$\frac{dy}{dt} \approx \frac{y_{n+1} - y_n}{h}$$

This equation can now be rewritten to find the value of *y* at a future time *n+1*

$$y_{n+1} = h\frac{dy}{dt} + y_n \qquad \text{where} \qquad \frac{dy}{dt} = f(y,t)$$

The parameter *h* is the time step size, something we can adjust, the time derivative of *y* is given by the original function *f(y,t)*, defined by our original equation. We will require knowledge of *y* at some earlier time *n*, written here as y_n, in order to step forward in time. Thus if we know the value of *y* at time *t=0* then we can find y for *t=1, t=2, t=3* and so on

$$\left.\begin{aligned} y_1 &= y_0 + hy_0' = y_0 + hf(y_0, t_0) \\ y_2 &= y_1 + hf(y_1, t_1) \\ y_3 &= y_2 + hf(y_2, t_2) \end{aligned}\right\}$$

And so it is a serially sequential operation starting at step one and just stepping along

A Taylor series expansion can be used to show that the error for this solution is in fact quite large and is given by

$$y'(t_0)h$$

In general, we would like to see the error to be above order one in *h* so that the error can decline very quickly as *h* decreases in size. This poor error for the Euler solution approach is a sign that the technique may not be all that robust, as we shall see shortly.

Let us explore the characteristics of this solution by looking at a specific example, one for which we know the solution. Let us consider the equation

$$\frac{dy}{dt} = -0.1(y - 10)$$

The exact solution of this equation is

$$y = 90e^{-0.1t} + 10$$

So when we develop our numerical solution we will be able to compare it with the exact solution. This will not be possible in general but it is a good idea to test an algorithm the first time you take it for a drive just to be sure that it is doing what it claims to be capable of and also to be sure that you have set it up properly.

```
% Example of Euler's method solution
finish=50;              % the number of steps to move forward
start=0;                % the start location for our solution
step=1.0;               % the step size
steps=(finish - start)/step + 1;          % given the first three parameters then the number of steps
y=100; t=0;             % y=100 at time t=0, we need a first guess, here we take it from the equation
yvals=y;                % storing the first value of y
tvals=start;            % storing the first value of time
for i=2:steps           % step forward the Euler solution from 2 to the number of steps
   y1=y+step*(-0.1*(y-10));   % increment the  next value of y using the Euler formula
   t1=t+step;           % increment the time in the loop
   tvals=[tvals, t1]; yvals=[yvals,y1];% store the new value of time and y
   t=t1; y=y1;          % reset the time and y in the loop and prepare for the next step
end                     % end of the loop
t=0:50;                 % create a vector of time for the analytical equation
clf;                    % clear the screen
plot(t,90*exp(-0.1*t)+10);   % plot of the exact solution (with the equation inside the plot command)
hold on;                % freeze the graph to prepare for the Euler solution to be plotted
grid;                   % lay on a grid to make it easier to compare the two solutions
plot(tvals,yvals,'r:')  % now  plot  the approximate Euler solution, admire the result!
```

We can explore the sensitivity of the solution by changing the step size and the plotting the result. As the step size increases then the two curves look quite different. As the step size decreases then they come closer together. The calculation is so simple here that changing the step size does not much influence the computation time. As we shall see the problems do not have to become much more complicated before changing the step size will indeed have a noticeable effect hence decreasing the step size will not always be a sensible approach to improving the solution.

Let us look back now to consider how we have developed this solution. We will see that that the Euler approach is not generally useful but the approach that we have taken is one that will work even for more sophisticated algorithms. What is the general approach here?

(1) Specify the initial conditions – we need the starting guess
(2) Choose a time step – in this case this is where we influence the solution the most
(3) Calculate the present value of the function – here the algorithm does its work
(4) Calculate a new value of the function – the second stage of the algorithm and then
(5) Go back to (3) until done

The limitations of the Euler approach may not be apparent from the example that we have worked with above. It is important to test any new approach very carefully because while a solution may work with one type of problem it may break down in other situations. But how accurate are these solutions and when do we know that we have the *right* solution. The best lesson is to be careful because all these techniques break down somewhere, so you have to be careful.

If we have an equation

$$\frac{dy}{dt} = Ky$$

then the Euler solution is of the form

$$y_{n+1} = y_n + hKy_n$$

The Euler solution repeatedly applies this solution and if you work through the algebra then you can show that the solution has the general form

$$y_{n+1} = (1 + hK\ )^{n+1} y_0$$

And if h is small then this approaches the value

$$e^{Kt}$$

Runge-Kutta Techniques

The true workhorses of the differential equation world are a class of techniques named after their founders Runge-Kutta (rung-uh cut-ah). The basic idea is pretty much the same as that of the Euler approach but with some more thought about how to keep the forward step closer to the function so that the integration is more precise and more stable. There are many different versions of this basic approach although the basic idea behind the implementation is always common. More sophisticated algorithms seek improved efficiency or are targeted towards specific types of differential equations. Differential equations can have different general characteristics that will determine how they should be dealt with. Analytical solution techniques are tailored to the type of the equation and so too with the numerical solution approaches. We will not consider all aspects of the Runge-Kutta approach, since there are so many, but will point out the general strategy

The starting point of this formula is the Euler equation

$$y_{n+1} = y_n + hf(y_n, t_n)$$

The solution is stepped forward y_{n+1} based upon the previous value y_n and the step size h times the function $f(y_n,t_n)$ which is evaluated at the departure point (y_n,t_n). We have already seen how this works. The trick in the Runge-Kutta approach is to evaluate the function f at a slightly different point. Stepping forward makes sense but in the Euler approach we are trying to step forward based on the information at the starting point and this can get us into trouble. If we pick up some additional information on the way, as we move forward, then we correct for some of the errors that we might carry with us from the starting point. The first step is to write

$$y_{n+1} = y_n + hf(y_{n+1/2}, t_{n+1/2})$$

Where we now write the forward step in terms of the previous value, just as before but now we evaluate *f* at a point $(y_{n+1/2}, t_{n+1/2})$ that is somewhere between the starting position and the next step and is given by

$$y_{n+1/2} = y_n + \frac{1}{2}hf(y_n, t_n)$$

This scheme is referred to as first order because it involves one 'correction' to the slope evaluation that is used to step the Runge-Kutta scheme forward. Even this very simple form of the Runge-Kutta approach is an improvement over the Euler method as we can quickly see by looking at an example of approximating the function

$$y = e^{-t}$$

Let us consider Euler's method at different time steps

Euler	**Runge-Kutta**
$x(.1) = 1 + 0.1(-1) = 0.9$	$x(0.1) = 1 + 0.1(-1) = 0.9$
$x(.2) = 0.9 + 0.1(-0.9) = 0.81$	$x(0.2) = 0.9 + 0.2(-0.9) = 0.82$
$x(.3) = 0.81 + 0.1(-0.81) = 0.729$	$x(0.3) = 0.82 + 0.1(-0.82) = 0.738$
$x(.4) = 0.729 + 0.1(-0.729) = 0.6561$	$x(0.4) = 0.82 + 0.2(-0.738) = 0.6724$

where the exact solution is

$$e^{-0.4} = 0.6703$$

After just a few steps, at the same step size, the Runge-Kutta solution is substantially better then the Euler solution. The Runge-Kutta approach is better and better behaved than the Euler approach and will work with most differential equations, pretty much all the ones that we will face here in this text.

We shall not consider the specific details of the Runge-Kutta implementation but will write down the second and the fourth order approaches, simply as examples. The higher order approaches simply add additional intermediate steps to further correct the forward integration. In general, the solution improves with additional terms but there is of course a computational price to be paid and ultimately there is a tradeoff between improved estimation and numerical efficiency. The most commonly used scheme, one that provides the optimal tradeoff, is the fourth order scheme which is truly the workhorse of differential equation solvers. There are race horses, and other types of 'four-legged' solvers but the fourth scheme gets the job done without acting up too much.

The second order scheme is

$$y_{n+1} = y_n + h/2(f(y_n, t_n) + f(y_{n+1}, t_{n+1})$$

where

$$y_{n+1} = y_n + hf(y_n, t_n)$$

and the fourth order Runge-Kutta is

$$y_{n+1} = y_n + \frac{h}{6}[f_1 + 2f_2 + 2f_3 + f_4]$$

where

$$f_1 = f(y,t)$$

$$f_2 = f(y + \frac{h}{2}f_1, t + h/2)$$

$$f_3 = f(y + \frac{h}{2}f_2, t + h/2)$$

$$f_4 = f(y + \frac{h}{2}f_3, t + h/2)$$

As an exercise it is instructive to write your own solver, at least once, as it illustrates how straightforward the idea really is. In the Matlab solver only takes a few lines of coding and a homemade solver will work with nearly all of the differential equations that we shall see. The solvers that we will use in *Matlab* are more sophisticated than the typical homemade ones. They try to be efficient and check to ensure that they are working properly. In an effort to be efficient, they adjust the time step, essentially lengthening it if the function is not changing quickly. The improved efficiency is balanced by the time it takes to ensure that the algorithm is working and as a result the *Matlab* routines are not necessarily superfast although they are in general well-behaved.