GSTDMB 2010
Dynamical Modelling for Biology and Medicine:

Introduction to MATLAB /
Qualitative and quantitative model analysis

Markus Owen

1 Getting started in UoN computer labs

Find a free computer and login using your UoN account. You should have a personal filestore, "`My Documents`", mounted as drive "`Z:`", which is independent of which computer you use to login, so you can save your work and return to it later during the course.

You should work through the examples in Sections 2 and 3 before attempting the exercises in Section 4. Files for this practical can be downloaded from:
`http://www.maths.nottingham.ac.uk/mathsforlife/gstdmb2010/prac-matlab-odes.zip`
When prompted, click "`Save`" to save the zip file to "`My Documents`", then navigate there and double-click "`prac-matlab-odes.zip`" to extract the files from the archive. This should create a folder called "`prac-matlab-odes`" containing this handout and a number of MATLAB files with a "`.m`" extension.

2 MATLAB basics

MATLAB is a contraction of *MATrix LABoratory*, and as such it is principally a tool for numerical computation with vectors and matrices. In addition, it provides a high level programming language and advanced graphics, image analysis and visualisation tools. There is also a range of toolboxes providing specialised capabilities for problems in areas such as optimization, statistics, neural networks, image processing and even systems biology.

Commands can be entered on the *Command Line* or stored as *scripts* (also known as *M-files*, simply a collection of MATLAB commands) or functions (which are passed and return *arguments*) for execution on demand.

2.1 Launching MATLAB

On University of Nottingham computer labs, you should be able to launch MATLAB via the "Start" menu:
Start > All Programs > UoN Software > School of Mathematics > Matlab > MATLAB R2008a

2.2 The command window

Once you have launched MATLAB you should see a number of windows (or one window subdivided into sections), the most important for now being the **Command Window**. This has a prompt at which you can type commands:

```
>>
```

Hitting "`Return`" will execute the command(s) entered at the prompt.

2.3 Getting help

There are two main ways to get help. You can click on **Help** on most MATLAB windows, or use the command line — to get help on any MATLAB command simply type “`help` *command*”.

2.4 Creating and assigning scalars, vectors and arrays

First a note on terminology. For our purposes:

- a **scalar** is a single number;
- a **vector** is a one-dimensional array of `n` numbers. Vectors can be row vectors (1 row, `n` columns) or column vectors (`n` rows, 1 column) ;
- an `n`×`m` **matrix** is a two-dimensional array of numbers, with `n` rows and `m` columns.

Variable names are case-sensitive, so that `a` and `A` are distinct. The following assigns values to `a` and `b`, and subsequently also adds them and assigns the answer to `c`:

```
>> a = 2

a =

     2

>> b = 3;
>> c = a + b

c =

     5
```

Note how the semicolon (`;`) suppresses the output of the assignment to the screen, which is useful and important when dealing with large arrays (you would not want to see the entries in a 100×100 matrix scrolling past).

Square brackets ([,]) are used in forming vectors and matrices. For example, vectors can be assigned in the following way:

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

x =

     1     2     3

>> y = [1,2,3]

y =

     1     2     3

>> z = [1;2;3]
```

```
z =

     1
     2
     3

>> w = [1 2 3]'

w =

     1
     2
     3
```

The first and second declarations are equivalent. `[A B]` or `[A,B]` is the horizontal concatenation of matrices `A` and `B` (where `A` and `B` must have the same number of rows), and since `1`, `2` and `3` are 1×1 matrices, `[1 2 3]` simply indicates the concatentation into a single 1×3 matrix (a row vector).

The semicolon indicates vertical concatenation (you can think of it as indicating the start of a new row), so that `[A;B]` is the vertical concatenation of matrices A and B (and A and B must have the same number of columns), and since `1`, `2` and `3` are 1×1 matrices, `[1;2;3]` simply indicates the concatentation into a single 3×1 matrix (a column vector).

To swap between row and column vectors we can use the transpose, indicated by `'`.

Given the above declarations of `x`, `y`, `z` and `w`, we can do certain operations such as addition, subtraction, etc. For example, we can add `x` and `y`, or multiply `x` by a number:

```
>> x + y

ans =

     2     4     6

>> 5*x

ans =

     5    10    15
```

Note that multiplying an array (like `x`) by a number (a scalar) simply multiplies every element by that number.

However, we cannot add `x` and `z` because they have different sizes: `x` is a 1×3 matrix (1 row and 3 columns), whereas `z` is a 3×1 matrix (3 rows and 1 column):

```
>> x + z
??? Error using ==> plus
Matrix dimensions must agree.
```

This is a common mistake to make in MATLAB.

What if we want to set up a large vector or matrix with regularly spaced entries? For example, if we want x to be a vector with the even numbers from 2 to 200, we can use the colon (:) notation.

```
>> help colon
 :  Colon.
    J:K  is the same as [J, J+1, ..., K].
    J:K  is empty if J > K.
    J:D:K  is the same as [J, J+D, ..., J+m*D] where m = fix((K-J)/D).
    J:D:K  is empty if D == 0, if D > 0 and J > K, or if D < 0 and J < K.
```

Thus we can get our even numbers with

```
>> x = 2:2:200

x =

  Columns 1 through 15

     2     4     6     8    10    12    14    16    18    20    22    24    26    28    30

  Columns 16 through 30

    32    34    36    38    40    42    44    46    48    50    52    54    56    58    60

...

  Columns 91 through 100

   182   184   186   188   190   192   194   196   198   200

>>
```

Of course we really should have used the semicolon, "x = 2:2:200;" to avoid displaying all the numbers!

To get a column vector in this way, use the transpose ('), e.g.

```
>> y = [0:3:12]'

y =

     0
     3
     6
     9
    12

>>
```

Another important operation is multiplication. There are two kinds:

- `*` : this is matrix multiplication, `A*B` only works if the number of columns of A equals the number of rows of B.
- `.*` : this is called array multiplication, `A.*B` only works if A and B have the same size (unless one of them is a scalar). Array multiplication multiplies corresponding elements, which

is extremely useful for calculating the same expression for multiple values. Suppose we want to square each of the numbers 0, 3, 6, 9, 12. We can simply do this by array multiplying y by itself. To get the cube of the same numbers, we can use the analogous array power, `.^`:

```
>> y.*y

ans =

     0
     9
    36
    81
   144

>> y.^3

ans =

           0
          27
         216
         729
        1728

>>
```

2.5 Built-in functions and constants

Most common mathematical functions are built-in: `sin`, `cos`, `log` (the natural logarithm), `log10` (base 10 logarithm), `exp` (the exponential), etc.

The constant `pi` returns the floating-point number nearest to the value of π.

2.6 Plotting

Probably the most important plotting function is "`plot`", suprisingly! "`plot(x)`", if x is a vector, simply plots the entries in the vector against their index, whereas "`plot(x,y)`" plots y against x if they are of the same size. For example, this will plot $\sin(x)$ against x:

```
>> x = 0:0.01:2*pi;
>> plot(x,sin(x))
>>
```

The plot will be on the current figure (or if there are no figure windows open, on a new figure window). This introduces a number of important commands relating to figure windows:

`figure`, `figure(h)`: Without the argument, creates a new figure window. With the argument, if a figure exists with handle `h` then that figure window is brought to the front and is made active, otherwise a new figure window is created. It is preferable to choose a figure number, and use that number to subsequently access that figure, to avoid generating new figure windows every time you execute a plot.

`clf`: Clear current figure.

`hold [on | off]`: On its own, toggles the hold property. Hold on means that subsequent plotting commands add to the existing graph, allowing you to superimpose multiple plots.

Suppose we want to plot $\cos(x)$ on the same set of axes, and to use a particular figure number:

```
>> x = 0:0.01:2*pi;
>> figure(1)
>> clf
>> hold on
>> plot(x,sin(x),'r')
>> plot(x,cos(x),'b')
>> figure(1)
>>
```

The final "`figure(1)`" will bring the figure window to the foreground so you can actually see what you just did. You could also click on "`Window`" on any MATLAB window to see a list of active windows.

We have also introduced the line specification, a string which specifies the colour of the line, the symbols used at each data point (if any) and whether lines should be solid, dashed, etc. Type "`help plot`" to see a list of these options. You can specify additional properties using parameter/value pairs, such as
» `plot(x,sin(x),'r','linewidth',2)`
which will plot a red line twice the default thickness (which often looks better in documents).

Another way to plot functions is to use the "`fplot`" function:

```
>> fplot(@(x) sin(x), [0 2*pi])
>>
```

will plot $\sin(x)$ for x from 0 to 2π. This way you do not need to set up your own array of x values. However, it is not so easy to change the properties on the line such as its width.

2.7 Loops

For loops are easy to implement in MATLAB. They take the form
`FOR variable = expr, statement, ..., statement END`
where `expr` is usually a vector of values which do not have to be integers. Thus the loop simply assigns `variable` to take consecutive values from the vector `expr`, and evaluates whatever statements appear before `END`.

A simple example might be to plot multiple graphs of related functions. Suppose we wanted to plot exponential growth functions, e^{rt}, with a range of growth rates, r. We could use the following:

```
>> t = 0:0.01:2;
>> figure(1)
>> clf
>> hold on
>> for r=0:0.1:1, plot(t,exp(r*t),'r','linewidth',2), end
>> figure(1)
>>
```

Here the `variable`, `r` takes the values $0, 0.1, 0.2, \ldots, 1$, and for each value the plot is executed. The "`hold on`" ensures that the plots are superimposed.

The output of this code is shown in figure 1.

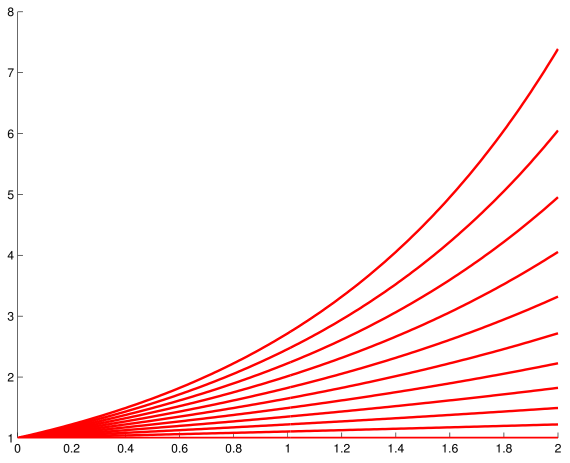

Figure 1: The output from the example loop code to plot e^{rt} for various values of r.

3 Ordinary Differential Equations in MATLAB

Here is a very simple function to simulate a first order ordinary differential equation with exponential growth:

```
function expode
%EXPODE  ODE for exponential growth, dx/dt = rx

% this is the range of time to simulate for
tspan = [0 10];
% this is the initial value of the state
x0 = 1;
% this is the value of the parameter r
r = 1;

figure(1);
clf
% solve the problem using ODE45, storing the solution in t and x
[t,x] = ode45(@f,tspan,x0);
plot(t,x)

    % ---------------------------------------------------------------
    function dxdt = f(t,x)
    dxdt = r*x;
    end
end
```

ode45 is one of a suite of ODE solvers in MATLAB. It should be fine for most of the systems you look at. On the command line, "help ode45" includes "See also" information with links to other MATLAB ODE solvers. Similar information can be found by clicking "Help > Product Help" and then searching for ode45.

This code is saved as "expode.m" in the zip archive for this practical at:
http://www.maths.nottingham.ac.uk/mathsforlife/gstdmb2010/prac-matlab-odes.zip

Alternatively, to write a new m-file, click the blank sheet of paper in the toolbar, or click File->New->M-File. Type in the above code into a new M-File and save it as expode.m.

Either way, once you have the file expode.m, you can either hit F5 while in the M-File editor window, or you can type expode in the command window, provided the current working directory is the same as where you have saved the function.

The output of this function is shown in figure 2. You might want to enhance this by adding

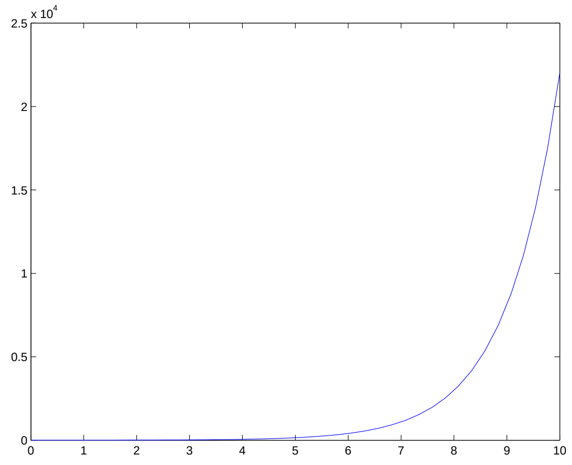

Figure 2: The output from the above example code expode.m.

axis labels, etc. Try adding these lines after "ode45(@f,tspan,x0);":

```
xlabel('time, t','fontsize',16);
ylabel('solution, x(t)','fontsize',16);
title('Output of expode.m, simulation of dx/dt=rx','fontsize',16)
```

You can also see the solution for a different initial condition, or a different choice of parameter. For example, try adding these lines after "title('...'":

```
hold on
x0 = 2;
[t,x] = ode45(@f,tspan,x0);
plot(t,x,'r')
```

This will superimpose in red the simulation for a starting x-value of 2. As expected the new curve should be twice the first one.

4 Exercises

The above should be enough to get you started with plotting functions and simulating differential equations in MATLAB. The following exercises will build on this so you can use MATLAB and graphical techniques to analyse models of the kind discussed in the lectures.

We will study the case of

$$\frac{dx}{dt} = P(x) - \delta x$$

with various forms for $P(x)$. This simple model represents transcriptional autoregulation, where $P(x)$ characterises the effect of an mRNA on its own rate of transcription, and we assume a constant rate of mRNA degradation.

4.1 Positive feedback: $P(x) = \frac{Ax}{h+x}$

Open the file "`saturating.m`". This contains a function of the same name which will:

Figure 1: Plot $P(x)$ with $A = 1$ and $h = 1$ and x ranging from 0 to 2.
Superimpose on the same plot δx with $\delta = 0.5$.
For these parameters, there should be intersections at $x = 0$ and $x = 1$.

Figure 2: Plot $\frac{dx}{dt} = P(x) - \delta x$ for these parameter values. In order to more clearly see where the curve crosses the horizontal axis, we have used "`grid on`" to superimpose a grid based on the axes tick-marks.

Figure 3: Simulate this system for a range of initial conditions

Run the function as described above (using F5 or typing "`saturating`" on the command line and then "`Return`"), then consider the following questions:

1. Do the plots in Figure 1 agree with the graphical analysis and steady states predicted in the lecture?
2. Figure 2 is essentially the phase line diagram, without the direction of trajectories marked. On paper, sketch the phase-line diagram, including steady states and their stability, and representative solutions (i.e. sketch x as a function of t for different initial conditions $x(0)$).
3. Do the simulations of Figure 3 agree with your sketched representative solutions?
4. What happens as you increase δ? Identify any bifurcations.

You can save your plots if you like in a number of ways: via the "`File`" menu on a Figure window you can select "`Save`" and select from a number of formats (eps, jpeg, png, bmp, and others). You can also print to a file using the command line "`print`" command, e.g. "`print -depsc test.eps`" will produce a colour figure as encapsulated postscript.

4.2 Positive feedback: $P(x) = \frac{Ax^n}{h^n + x^n}$

When $n = 1$ this is the same as the previous case. Now we will see that this Hill function form with $n > 1$ has a different shape which raises additional possibilities including **bistability**. Here, and in the following, it is best to save your modified functions under different names.

Start by saving "`saturating.m`" as a new file "`hill.m`". Then modify this file to allow a Hill function with arbitary n. This should simply require modifying the subfunction "`P(x)`" and setting the parameter `n` along with the other parameters. Initially set $n = 1$ and run `hill.m` to check that it gives the same results as the previous exercise when $n = 1$.

1. Now set $n = 2$, $A = 1$, $h = 0.5$ and $\delta = 0.9$, and run the function `hill.m`. What do you notice about the shape of the function $P(x)$ compared to that with $n = 1$? Are there any intersections of $P(x)$ with δx? What does this mean?

2. Use the plot of $dx/dt = P(x) - \delta x$ to sketch the phase-line diagram and representative solutions for these parameter values. How many stable steady states are there? If there are more than one, what choice of initial values `x0` give solutions going to each steady state?
   NOTE: only one steady state is plotted so far, but we could find and plot more by varying the initial guess passed to "`fzero`", replacing

   ```
   xss = fzero(@dxdt,1);
   ```

   with, for example,

   ```
   xis = [0 0.5 1];
   for j=1:length(xis)
       xss(j) = fzero(@dxdt,xis(j));
   end
   ```

3. Do the simulated solutions generated in Figure 3 agree with your qualitative analysis?
   NOTE: the superimposed solutions for multiple initial values `x0` show how important initial conditions can be when a system is bistable.

4. What happens as you increase δ? Identify any bifurcations. If you increase δ just beyond the bifurcation, you should find that solutions starting from large x take a long time to reach the remaining steady state (you may have to increase the maximum time in `tspan` - e.g. try "`tspan=[0 100]`" with $\delta = 1.01$).

5. What happens as you decrease δ?

6. Try increasing n. What happens to the shape of the function $P(x)$? Does this change the qualitative nature of the system?

4.3 Positive feedback with constitutive transcription: $P(x) = \frac{Ax^n}{h^n+x^n} + B$

When $B = 0$ this is of course the same as the previous example 4.2. We know that with δ sufficiently large we can lose the upper steady states. Here we see that increasing B can regain those states, but also as we increase B further we lose the *lower* two steady states.

Start by saving "`hill.m`" as a new file "`hillbasal.m`". Then modify this file to add the basal transcription term to $P(x)$. This should simply require modifying the subfunction "`P(x)`" and setting the parameter `B` along with the other parameters.

1. Now set $n = 4$, $A = 1$, $h = 0.5$ and $\delta = 1.2$. Run `hillbasal.m` for B ranging from 0 to 0.3. How do the intersections change as B varies?
   NOTE: You could try wrapping the main plotting and ODE routines in a loop which successively picks values of B and superimposes the plots on the same axes.

2. Use the plot of $dx/dt = P(x) - \delta x$ to sketch the phase-line diagram and representative solutions for these parameter values. How many stable steady states are there? How does this depend on B? What choice of initial values `x0` gives solutions going to each steady state?
   HINT: you should find *bifurcations* at $B \approx 0.04$ and $B \approx 0.245$.

3. Do the simulated solutions generated in Figure 3 agree with your qualitative analysis? See the figures above.

   For the larger values of B, As you increase B just beyond the second bifurcation at $B \approx 0.245$, you should find that all solutions converge to a steady state with a high TF level, but that solutions starting from small x take a long time to reach that remaining steady state (you may have to increase the maximum time in `tspan`).

4. What are the biological implications of sigmoidal feedback plus basal transcription?