CHAOS WITH MATLAB

Carlo F Barenghi

February 24, 2012

Contents

Part I
MATLAB

1 Matlab basics

1.1 Ouverture

Firstly, create a *folder* (*directory*) in your Windows system of name *mas2106*. All your files and all work which you do will be in this folder. To start up Matlab, go to a university computer cluster and login to your account. From the `Start` button, select `All programs`, `Scientific software` and `Matlab2010b` (the version number may change from year to year). Matlab starts greets you with a number of windows. The most important window, the *command window*, gives access to Matlab's command line, a prompt which looks like this:

```
>>
```

(hereafter, framed text shows what appears on the computer screen).
To get a taste of Matlab, type the following commands followed by return:

```
>> x=[0:pi/100:10*pi];
>> y=x.*sin(x);
>> plot(x,y)
```

What you see is a plot of the function $y = x \sin x$ computed from $x = 0$ to $x = 10\pi$ in steps of size $\Delta x = \pi/100$. Try the above commands changing numbers and formulae (for example, try to plot another function) to see if you understand what each command does. In particular, see what happens if you remove the semi colon at the end of a line.

To get a taste of 3-dim plots, type the following commands:

```
>> [x,y]=meshgrid(-4:0.1:4, -4:0.1:4);
>> z=sin(x).*sin(y);
>> surf(x,y,z)
```

What you see is a plot of the function $z = \sin(x)\sin(y)$ for $-4 \leq x \leq 4$, $-4 \leq y \leq 4$, computed on a grid with spacing $\Delta x = 0.1$, $\Delta y = 0.1$.

1.2 Calculator

The symbols for the arithmetic operations are $+$, $-$, $*$ and $/$. The symbol ^ is used for exponents, for example 4^2=16. Type 2+3/4:

```
>> 2+3/4
ans =
    2.7500
```

Note that the result of the calculation is called `ans`. The value of `ans` is kept and can be used in a second calculation:

```
>>2*ans
ans =
    5.5000
```

1.3 Record the session

If you type the command

```
>> diary session
```

everything which appears on the screen will be saved to a file called `session` (or any other name that you want), providing you with a record of what you have done, until you type the command

```
>> diary off
```

which turns off the diary. The file `session` can be edited using using a text editor, for example *Notepad*.

1.4 Help

If you need help, click the `help` button on the toolbar. You can also type the command `help` followed by a keyword, for example `>> help plot` or `help sin`. Another option is to Google Matlab followed by a keyword: there is a huge amount of Matlab examples on the internet.

1.5 Variables

Names and values of variables are kept, and can be used over and over again in subsequent calculations (until you type the command `>> clear` to remove all variables from the workspace). Here is an example:

```
>> x=3.1
x =
    3.1000
>> y=17
y =
    17
>> z=x+y
z =
   20.1000
>> z+x
ans =
   23.2000
```

Names of variables can contain any combination of letters and numbers (do not use symbols), but must start with a letter. Avoid special names such as "pi" and "eps" which are respectively reserved for π (Greek pi) and the largest number such that $1 + \epsilon$ is not distinguishable from 1:

```
>> pi
ans =
    3.1416
>> eps
ans =
   2.2204e-16
```

Matlab uses the "e" notation for very large and very small numbers. For example:
$-1.2345e + 03 = -1.2345 \times 10^3 = -1234.5$
$-5.6789e - 01 = -5.6789 \times 10^{-1} = -0.56789.$
Matlab does all computations with about 15 significant digits.

1.6 Intrinsic functions

Matlab has many built-in functions, such as `sqrt`, `exp`, `log`, log10, the trigonometric functions `sin`, `cos`, `tan` (the argument must be in radians), and the inverse trigonometric functions `asin`, `acos`, `atan`. For example try:

```
>> cos(pi),sin(pi/2),log(exp(2))
ans =
    -1
ans =
    1
ans =
    2
```

1.7 Suppressing the output

If you do not want to see the result of an intermediate calculation, terminate the command with a semi colon:

```
>> x=1;
>> y=sqrt(4);
>> z=x+y
z =
  3
```

To keep your screen tidy, you can write more than one command on the same line:

```
>> x=1; y=3;
>> z=x+y
z =
  4.0000
```

1.8 Complex numbers

To define complex numbers use the the symbol $i = \sqrt{-1}$ in the usual way:

```
>> x=1+2i;
>> y=3+4i;
>> z=x+y
z =
  4.0000 + 6.0000i
```

2 Vectors

2.1 Row vectors

To create a row vector, enter the components one by one, separating them either by a blank or by a comma. Enclose the components in square brackets:

```
>> x=[1, sqrt(2), 3]
x =
  1.0000   1.4142   3.0000
>> y=[0.1 4 6]
y =
  0.1000   4.0000   6.0000
```

The command `>> length` returns the number of components of a vector:

```
>> length(x)
ans =
    3
```

It is easy to create a linear combination of vectors (careful: they must have the same length). Using the vectors x and y defined above, we have for example:

```
>> x+y
ans =
    1.1000   5.4142    9.0000
```

```
>>>3*x+2*y
ans =
    3.2000   12.2426   21.0000
```

A longer row vector can be built using existing row vectors:

```
>> z=[x y]
z =
    1.0000   1.4142    3.0000   0.1000   4.0000    6.0000
```

Let us change the first component of the last vector:

```
>> z(1)=0
z =
    0.1000   1.4142    3.0000   0.1000   4.0000    6.0000
```

The commands `a:b` and `a:b:c` are short cuts to create vectors of components starting from the value of a, incrementing by 1 or by the value of b, until we get to the value c (not beyond c). For example:

```
>> 5:8
ans =
    5    6    7    8
>> 0.3: 0.05: 0.5
ans=
   0.3000    0.3500   0.4000    0.4500    0.5000
>> 0.3:0.05:0.41
ans =
    0.3000    0.3500 0.4000
```

2.2 Column vectors

To create a column vector, separate the components with a semi colon or with a new line:

```
>> x=[pi; 0; sqrt(2)]
x =
  3.1415
       0
  1.4142
>> y=[0.1
     0.01
     0.001]
y =
  0.1000
  0.0100
  0.0010
```

2.3 Transpose

The transpose command (an apostrophe) turns a row vector into a column vector or viceversa:

```
>> y'
ans =
    0.1000    0.0100    0.0010
```

2.4 Scalar product of vectors

Consider a row vector $\mathbf{u}$ of components

$$\mathbf{u} = [u_1, u_2, \cdots u_N]$$

and a column vector $\mathbf{v}$ (of the same length) of components

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \cdots \\ v_N \end{bmatrix}$$

The usual scalar (or dot, or inner) product of the two vectors is a number defined as

$$\mathbf{u} \cdot \mathbf{v} = u_1 v_1 + u_2 v_2 + \cdots + u_N v_N = \sum_{j=1}^{j=N} u_j v_j$$

In Matlab we calculate it with the command `*` as in the following example:

```
>> u=[2,1,4]
u =
     2     1     4
>> v=[3;5;6]
v =
     3
     5
     6
>> z=u*v
z =
    35
```

If the second vector $\mathbf{v}$ is also a row vector, we must first transpose it to get a column vector. For example

```
>> u=[2,1,4]
u =
     2     1     4
>> v=[3,5,6]
v =
     3     5     6
>> z=u*v'
z =
    35
```

The norm of a vector can be calculated in two ways:

```
>> sqrt(u*u')
ans =
    4.5826
>> norm(u)
ans =
    4.5826
```

2.5 Hadamard product of two vectors

If **u** and **v** are two vectors of the same type (both row vectors or both column vectors) and of the same length N, their Hadamard product is the vector of components $[u_1v_1, u_2v_2, \cdots, u_Nv_N]$. In Matlab, the Hadamard product is calculated using the operator .* as in the following example:

```
>> u=[2,1,4]
u =
  2    1    4
>>v=[3,5,6]
v =
  3    5    6
>> u.*v
ans =
    6    5    24
```

Do not confuse the scalar product of vectors (which is a number) with the Hadamard product (which is a vector).

2.6 Zero and unit vectors

The command `>> ones(m,n)` creates an $m \times n$ matrix of 1's. Similarly, the command `>> zeros(m,n)` creates an $m \times n$ matrix of 0's. Therefore, a zero row vector of length 3 and a unit colum vector of length 4 are obtained by typing

```
>> x=zeros(1,3)
x =
     0     0     0
>> y=ones(4,1)
y =
     1
     1
     1
     1
```

2.7 Solved exercises

Exercise 2.1
Given two vectors [10,12,20] and [5,4,10], find the vector whose components are the ratios of the corresponding components.

Solution:

```
>> u=[10,12,20]; v=[5,4,10];
>> u./v
ans =
     2    3   2
```

Exercise 2.2
Find the angle (in radians) between $\mathbf{x} = [3, 0, 0]$ and $\mathbf{y} = [2, 2, 0]$.

Solution: Since $\mathbf{x} \cdot \mathbf{y} = |\mathbf{x}||\mathbf{y}| \cos\theta$, then

```
>> x=[3 0 0]
x =
     3     0     0
>> y=[2 2 0]
y =
     2     2     0
>> theta=acos((x*y')/(norm(x)*norm(y)))
theta =
    0.7854
```

which is $\pi/4$.

Exercise 2.3
Use the Hadamard product to make a table of the values of the function $y(x) = x\cos(\pi x)$ for $x = 0$ to 1.0 in steps of 0.01.

Solution:

```
>> x=[0:0.25:1.0]';
>> y=x.*cos(pi*x)
y =
         0
    0.1768
    0.0000
   -0.5303
   -1.0000
```

3 Graphics

3.1 Simple plots

We want to plot the function $y = \sin(2\pi x)$ for $0 \leq x \leq 1$. To achieve the aim, we compute the function at a large number of points and then join the points by straight lines. Let us define a row vector $\mathbf{x}$ which consists of $N + 1$ points spaced the distance h apart. For example, let $N = 5$:

```
>> N=5; h=1/N; x=0:h:1
x =
         0    0.2000    0.4000    0.6000    0.8000    1.0000
```

Another way to create the row vector is to use the command `linspace(a,b,N)`, which generates $N + 1$ equidistance points between a and b included. For example, for $N + 1 = 6$ we have:

```
>> x=linspace(0,1,6)
x =
         0    0.2000    0.4000    0.6000    0.8000    1.0000
```

Then we compute the vector $\mathbf{y}$ of components $y_k = \sin(2\pi x_k)$ $(k = 1, 2, \cdots 6)$ and plot y_k vs x_k:

```
>> y=sin(2*pi*x); plot(x,y)
```

A windows opens with the required plot, shown in Fig. 1(top). As you can see, the plot is bad: there are too few points. An improved plot is obtained if we set $N = 100$ (suppress the output of the x-points with a semi-colon, otherwise there will be too many numbers on the screen). We also add a title and labels on the x and y axes; the result is shown in Fig. 1(bottom).

```
>> x=linspace(0,1,100);
>> y=sin(2*pi*x); plot(x,y)
>> title('y vs x')
>> xlabel('x')
>> ylabel(y')
```

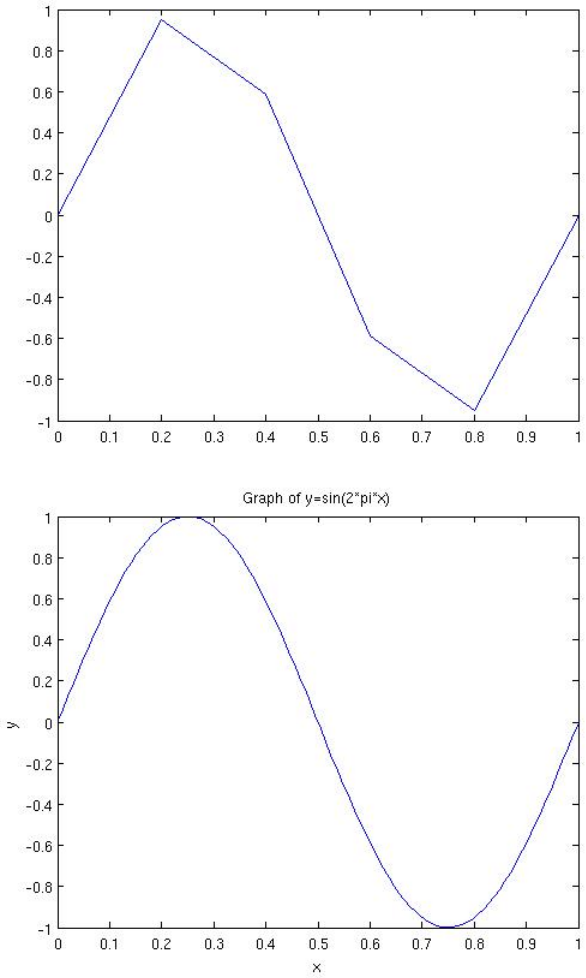

Figure 1: Top: Plot of $y = \sin(2\pi x)$ with too few $(N + 1 = 6)$ points. Bottom: Improved plot with $N + 1 = 101$ points, title and labels.

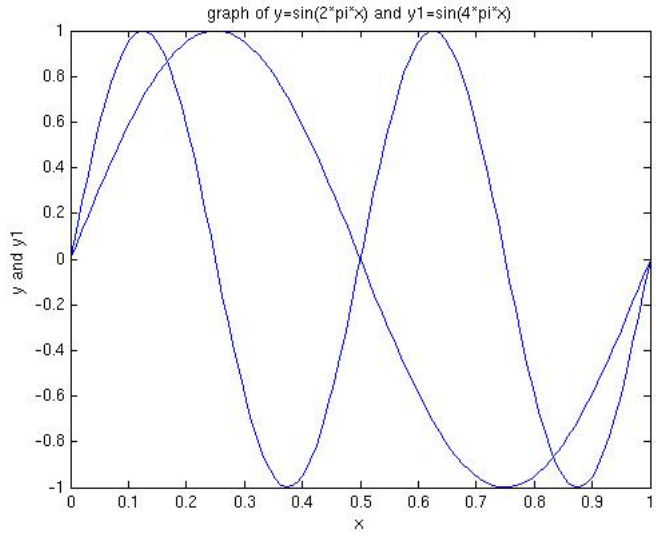

Figure 2: A curve added to the previous graph

.

3.2 The hold command

To add material to an existing graph, we use the command >> `hold on`, which retains the plot in the graphics window and lets us execute more commands on it, until we type >> `hold off`. For example, let us add the curve $y = \sin(4\pi x)$ to the existing graph of $y = \sin(2\pi x)$, and also change the title and the y-label:

```
>> hold on
>> y1=sin(4*pi*x); plot(x,y1)
>> ylabel('y and y1')
>> title('Graph of y=sin(2*pi*x) and y1=sin(4*pi*x)')
```

Fig. 2 shows the result.

3.3 Clear figure

The command `>> clf` clears the figure, emptying the graphics window.

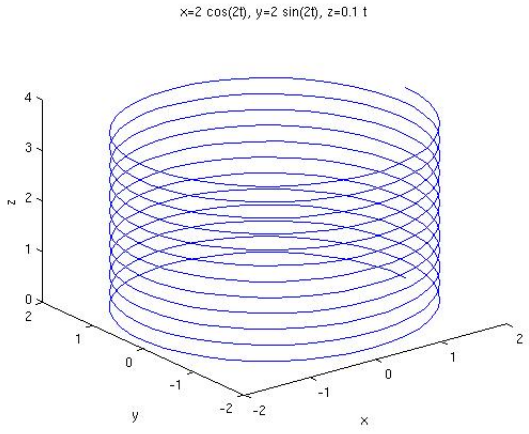

Figure 3: Helical curve $x = 2\cos(2t), y = 2\sin(2t), z = 0.1t$.

3.4 Three-dimensional plots

The following commands plot the helical curve

$$x(t) = a\cos(\omega t), \qquad y(t) = a\sin(\omega t), \qquad z(t) = bt,$$

with $a = 1$, $b = 0.1$ and $\omega = 2$, for $0 \leq t \leq 12\pi$.

```
>> a=2; b=0.1; w=2;
>> t=linspace(0,12*pi,500);
>> x=a*cos(w*t);
>> y=a*sin(w*t);
>> z=b*t;
>> comet3(x,y,z)
>> plot3(x,y,z)
>> xlabel('x'); ylabel('y'); zlabel('z');
>> title('x=2 cos(t), y=2 sin(t), z=0.1 t')
```

The outcome is shown in Fig. 3. Note that the-range has been divided in 500 points, enough to draw a smooth curve. Left-click `Tools` on the toolbar and choose `Rotate 3D` to rotate the helix and look at it from a different angle.

3.5 Colour and linetype

A third argument added to the plot command controls the colour (first character) and the line type (second character) according to the table below. For example, the command >> `plot(x,y,'rx')` marks the data points with a red cross.

y	yellow	.	point
r	red	o	circle
b	blue	+	plus
g	green	-	solid
c	cyan	x	cross
m	magenta	:	dotted
k	black	-.	dash-dotted
w	white	–	dashed

Compare what you get with the different options:

```
>> t=linspace(0,1,100);
>> x=sin(2*pi*t);
>> plot(t,x)
>> plot(t,x,'ro')
>> plot(t,x,'bo')
>> plot(t,x,'-.')
```

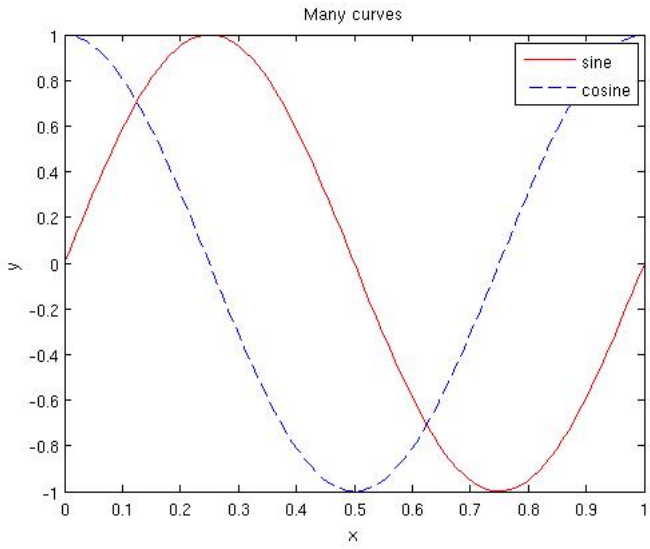

Figure 4: Multiple curves.

3.6 Multiple curves

If we want to plot more than one curve on the same graph, we proceed as in the following example, where the sine function is plotted as a solid red line and the cosine function as a dashed blue line, see Fig. 4. Note that we have also added a legend.

```
>> x=linspace(0,1,100);
>> y1=sin(2*pi*x)
>> y2=cos(2*pi*x)
>> plot(x,y1,'r-',x,y2,'b--')
>> legend('sine','cosine')
>> xlabel('x')
>> ylabel('y')
```

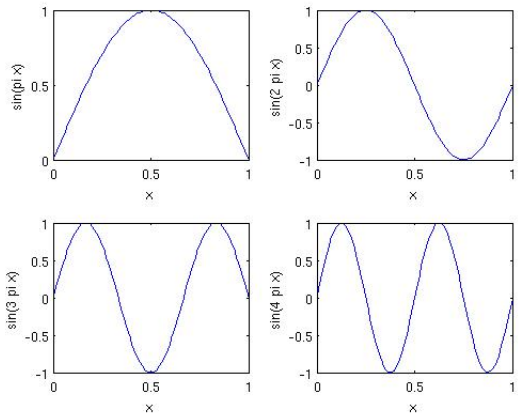

Figure 5: Plot with subplots.

3.7 Subplots

Another strategy is to split the graphics windows into an array of $m \times n$ smaller windows, each containing one graph. The windows are numbered row by row starting from the top left. In the following example we take $m = 2$, $n = 2$ and make plots of $sin(\pi x)$, $\sin(2\pi x)$, $\sin(3\pi x)$ and $\sin(4\pi x)$ for $0 \leq x \leq 1$; the resulting multiple plot is shown in Fig. 5.

```
>> x=linspace(0,1,100);
>> y1=sin(pi*x);
>> y2=sin(2*pi*x);
>> y3=sin(3*pi*x);
>> y4=sin(4*pi*x);
>> subplot(221), plot(x,y1)
>> xlabel('x'),ylabel('sin(pi x)')
>> subplot(222), plot(x,y2)
>> xlabel('x'),ylabel('sin(2 pi x)')
>> subplot(223), plot(x,y3)
>> xlabel('x'),ylabel('sin(3 pi x)')
>> subplot(224), plot(x,y4)
>> xlabel('x'),ylabel('sin(4 pi x)')
```

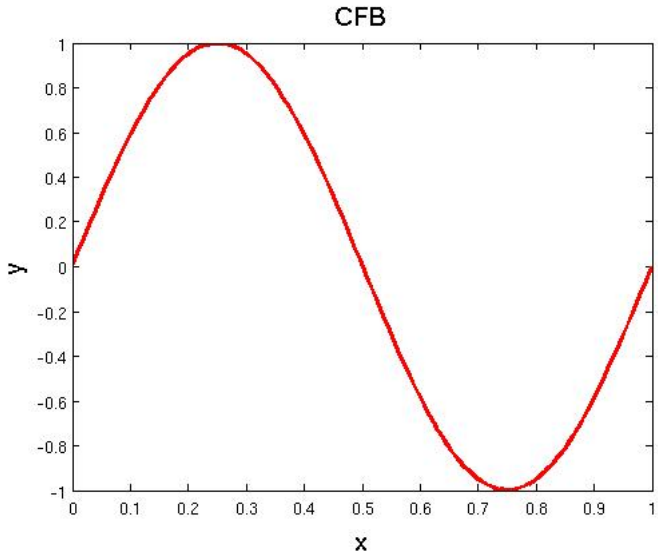

Figure 6: Controlling line thickness and symbol size.

3.8 Linewidth and symbol size

The thickness of the curves and the size of symbols can be controlled. For example the following commands produce the plot shown in Fig. 6:

```
>> x=linspace(0,1,100);
>> y=sin(2*pi*x);
>> plot(x,y1,'r-','linewidth',3)
>> title('CFB','fontsize',16)
>> xlabel('x','fontsize',16)
>> ylabel('y','fontsize',16)
```

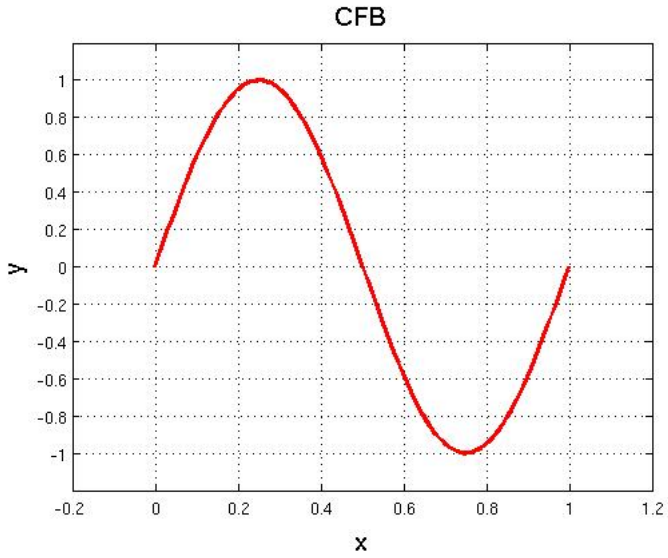

Figure 7: Controlling the limits of the axes and adding a grid.

3.9 Axes limits, grid

To control the limits of the axes and to add a grid to Fig. 6 we use the command `>> axis` and `>> grid`; for example, if we type

```
>> axis([-0.2 1.2 -1.2 1.2]), grid
```

we obtain Fig. 7.

3.10 Zoom, print, save

To see details of a graph, position the mouse where you want and type the command >> `zoom`. Then clicking the left (right) mouse button will zoom in (out). The command >> `zoom off` will switch off the zoom.

To print the graph, select `Print` on the window's toolbar. To save the graph into a file, select `Save as ...`; different formats are available.

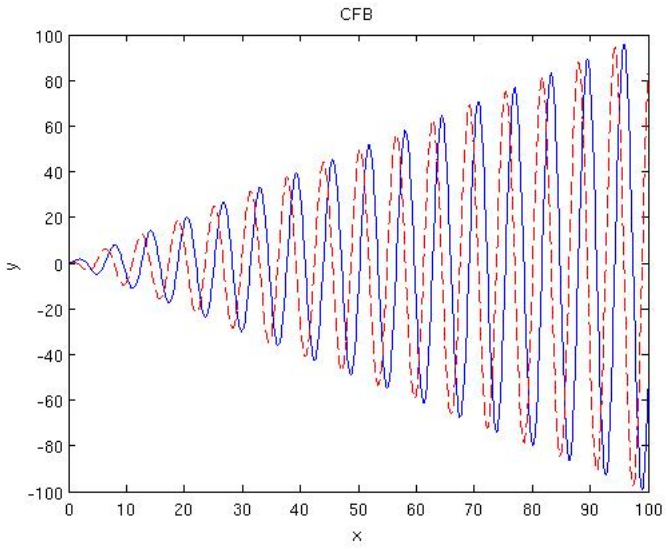

Figure 8: Solution to Exercise 3.1

3.11 Solved exercises

Exercise 3.1
Make a graph of the two functions $y_1(x) = x \sin x$ and $y_2(x) = x \cos x$, evaluating the values at 1000 points from $x = 0$ to $x = 100$. The axes must be limited by $0 \leq x \leq 100$, $0 \leq y \leq 100$. The first function must be a solid blue line, the second a dashed red line. The title should contain your name. The labels of the horizontal and vertical axis should be x and y respectively.

Solution: The Matlab commands are:

```
>> x=linspace(0,100,1000);
>> y1=x.*sin(x);
>> y2=x.*cos(x);
>> plot(x,y1,'b-',x,y2,'r--')
>> title('CFB'); xlabel('x'); ylabel('y');
```

Fig. 8 shows the result.

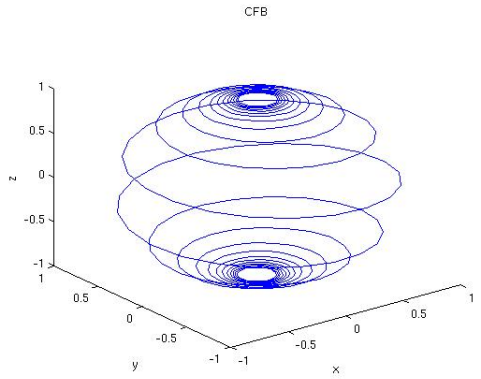

Figure 9: Solution to Exercise 3.2

Exercise 3.2
Make a three-dimensional graph of the parametric curve

$$x(t) = \frac{\cos(\omega t)}{\sqrt{1+a^2t^2}}, y(t) = \frac{\sin(\omega t)}{\sqrt{1+a^2t^2}}, z(t) = \frac{-at}{\sqrt{1+a^2t^2}},$$

for $a = 0.2$ and $\omega = 2$ and $-12 \leq t \leq 12$.
Solution: The Matlab commands are:

```
a=0.2; w=2;
t=linspace(-12*pi,12*pi,500);
x=cos(w*t)./sqrt(1+a^2*t.^2);
y=sin(w*t)./sqrt(1+a^2*t.^2);
z=-a*t./sqrt(1+a^2*t.^2);
comet3(x,y,z)
plot3(x,y,z)
xlabel('x'); ylabel('y'); zlabel('z'); title('CFB')
```

The result is shown in Fig. 9.

4 Loops and if

4.1 For loop

A *for loop* is used to repeat a command a number of times. It has the form

```
>> for n = n1:n2:n3
   [commands to be executed]
   end
```

where the counter n takes its values from a given row vector. Here are some examples:
>> `for n=1:4` means that $n = 1, 2, 3, 4$;
>> `for n=0:2:8` means that $n = 0, 2, 4, 6, 8$;
>> `for n=[3 12 5 20]` means that $n = 3, 12, 5, 20$.

Let us use a for loop to find the sum of the first 10 integers:

$$y = \sum_{k=1}^{k=10} k,$$

The commands are:

```
>> y=0;
>> for k=1:10
   y=y+k;
   end
>> y
y =
    55
```

Note that at the beginning the variable y is initilised to zero. At the first iteration (k=1) we add 1 to y (getting y=1); at the second iteration (k=2) we add 2 to y (getting y=3); etc. When we exit the loop after k=10 y contains the required answer, the number 55.

4.2 If statement

The `if` command can have various forms:

```
>> if [test]
   [command to be executed if test is true]
   end
```

or

```
>> if [test1]
   [command] to be executed if test1 is true]
   else
   [command to be executed otherwise]
   end
```

The test can be whether a number is bigger or smaller than another number. The syntax is

`x == y`	Is x equal to y ?
$x \sim= y$	Is x not equal to y?
$x > y$	Is x greater than y?
$x < y$	Is x less than y?
$x >= y$	Is x greater than or equal to y?
$x <= y$	Is x less than or equal to y?

The following is an example:

```
>> a=1; b=2;
>> if (b>=a)
   c=b
   else
   c=a
   end
c =
    2
```

4.3 While loop

The general form of the *while loop* is

```
>> while [test]
   [commands to be executed]
   end
```

for example, let us use a *while loop* to find the greatest number n such that the sum $y = 1^2 + 2^2 + \cdots + n^2$ is less than 100.

The commands are the following:

```
>> n=1; y=1;
>> while y+(n+1) < 100
   n=n+1; y=y+n;
   end
>> [n,y]
ans =
     13   91
```

Note that at each iteration we add 1 to n and n to y; we also test that if we add another number we do not exceed 100. We result is $n = 13$ because $y = 1 + 2 + 3 + \cdots + 13 = 91$: if we added 14, the sum would exceed 100.

4.4 Solved exercises

Exercise 4.1
The Fibonacci numbers are defined by the recursion relation

$$f_n = f_{n-1} + f_{n-2}$$

with initial conditions $f_1 = 1$ and $f_2 = 2$. Use a `for loop` to find the fist ten Fibonacci numbers.

Solution: The commands are:

```
>> f(1)=1; f(2)=2;
>> for n=3:10
   f(n)=f(n-1)+f(n-2);
   end
>> f
f =
     1     2     3     5     8    13    21    34    55    89
```

Exercise 4.2
Use a *while loop* to find the root of the equation $x = \cos x$. The idea is to start from a guess x_1, say $x_1 = 1$, and iterate the sequence of values $x_{n+1} = \cos x_n$ for $n = 2, 3, \cdots$ until the difference beteen two successive values, $d = |x_{n+1} - x_n|$, is smaller than, say, 0.0001.

Solution: The commands are:

```
>> xnew=1;
>> d=1;
>> n=1;
>> while d>0.0001 & n< 1000
   n=n+1;
   xold=xnew;
   xnew=cos(xold);
   d=abs(xnew-xold);
   end
>> [n,xnew,xold]
ans =
   23.0000    0.7391    0.7391    0.0001
```

Note that at each iteration we test that d does not become smaller than 0.0001. We also test that n does not exceed a large number, say 1000, to stop the loop if for unexpected reasons it fails to converge. We conclude that after 23 iterations we converge to the root $x = 0.7391$, which is thus the solution of the equation $x = \cos x$.

5 Script files

5.1 Where is my file ?

Make sure that you are working in the folder *mas2106*. To check in which folder you are, at Matlab's prompt type the command

```
>> pwd
```

where `pwd` means "print working directory". If you are in the home folder rather than in the *mas2106* folder, you need to move down into *mas2106*: type

```
>> cd mas2106
```

where `cd` means "change directory". To move up one level in the directory tree, type

```
>>  cd ..
```

where the double dot means "the folder above" in the directory tree.

5.2 M-files

Script files, also known as *m-files*, are text files with the extension *.m* which contain Matlab commands. When choosing names of m-files, avoid symbols or names which are built-in Matlab functions. Script files are created using an editor, for example *Notepad*. In a script file, any text which follows the % symbol is ignored: the purpose of such text is to include comments which explain the commands, reminding you (or anybody else using your m-file) of the aim of the script file.

For example, let us write a Matlab program which asks for a number x, and computes the number $y = x + 10$.

Using *Notepad*, we create the file *simple.it* which contains the following lines:

```
% simple.m
% given x, computes y=x+10
x=input('Enter x = ');
y=x+10
```

We save the file and leave *Notepad*. We check that the file is in the directory *mas2106*. (to see which files are present in the directory where you are working, type `what`). Then, at Matlab's prompt, we execute the script file by typing `simple`. If we input $x = 3$ when asked, we get the following result on the screen:

```
>> simple
Enter x = 3
y =
    13
```

5.3 Solved exercises

Exercise 5.1
Write a Matlab program which computes the sum of the first N integers:

$$y = \sum_{k=1}^{k=N} k$$

where N is input by the user.
Solution: We have already seen how to solve this problem by typing directly Matlab's commands. Here we do it with a script file. Using *Notepad*, we create the following file called *sumN.m*:

```
% sum.mN
% to sum the first N numbers
N=input('Enter N = ')
y=0;
for k=1:N;
    y=y+k;
end
text='The sum of the first N number is ';
fprintf('%s \t %d \n',text,y)
```

We save the file and exit *Notepad*. To execute the m-file, at Matlab's prompt we type `sumN`:

```
>> sumN
Enter N = 5
N =
     5
The sum of the first N number is    15
```

6 Controlling the output

The `>> fprintf` command is used to control the output and write numbers or text to the screen or to a file. Here we are concerned only about writing to screen; the syntax is

`fprintp(format,variables)`

where `format` is a text string that determines the appearance of the output, and `variables` is a comma-separated list of variables to be displayed according to `format`. The format codes are:

%s	string
%d	integer
%f	floating point
%e	scientific
%n	insert new line
%t	insert tab

In the following example we define three variables a, b and c (respectively integer, real, and text) and explore writing them in different ways:

```
>> a=5; b=2^7; c='moon';
>> fprintf('%d\n',a)
5
>> fprintf('%f\n',b)
128.000000
>> fprintf('%e\n',b)
1.280000e+02
>> fprintf('%d\n',b)
128
>> fprintf('%s\n',c)
moon
>> fprintf('%d %f %s \n',a,b,c)
5 128.000000 moon
>> fprintf('%d \t %f \t %s \n',a,b,c)
5   128.000000   moon
```

Part II
CHAOS

7 The logistic map

The *logistic map*

$$x_{n+1} = r x_n (1 - x_n),$$

is used in biology to model how a population x_n (of birds or insects for example) changes with the seasons (time is represented by the integer numbers n); here x_1 is the initial condition and r is a parameter.

The equation can be solved with a pocket calculator, at least in principle. For example, suppose that $r = 2.8$ and $x_1 = 0.4$ and that we want to predict the population up to season $n = 20$. We time-step from x_1 to x_2 to x_3 etc in the following way:

$x_1 = 0.4$ at $n = 1$;
$x_2 = 2.8 \times 0.4 \times (1 - 0.4) = 0.6720$ at $n = 2$;
$x_3 = 2.8 \times 0.6720 \times (1 - 0.6720) = 0.6172$ at $n = 3$;
etc

Clearly a computer allows us to proceed much faster than a pocket calculator.

7.1 Steady solution

To solve the logistic map using Matlab we use the following commands:

```
>> nn=20; r=2.8; x(1)=0.4;
>> for n=1:nn-1
   x(n+1)=r*x(n)*(1-x(n));
   end;
>> plot(1:nn,x)
```

Note that the `for loop` goes from $n = 1$ to $n = nn - 1$ so that the lengths of the two vectors which are plotted (the vector with the integers from 1 to nn and the vector with the real numbers x_1, x_2, $\cdots$ x_{nn}) is the same, nn. The result is shown in Fig. 10. Note also that the solution of the logistic map is a discrete set of points x_n, not a continuos line; the straight segments which join the points x_n in Fig. 10 are only a guide to the eye.

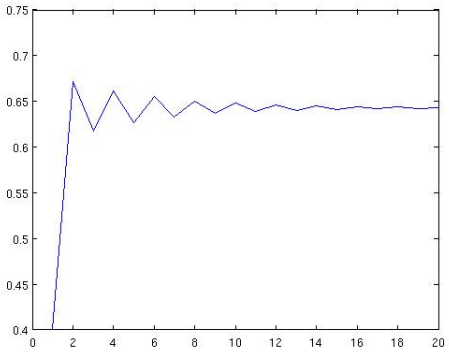

Figure 10: Solution x_n of the logistic map vs $n = 1, 2, \cdots, 20$ for $r = 2.8$ and initial condition $x = 0.4$ at $n = 1$.

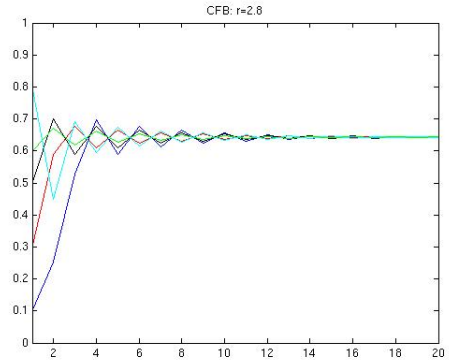

Figure 11: Solutions of the logistic map for $r = 2.8$ and various initial conditions: $x_1 = 0.1$, 0.3, 0.5, 0.6 and 0.8. Note that they tend to the same value

Fig. 11 shows that, for large n, the solution x_n is the same, no matter what is the initial condition. To produce this figure, which contains more than one plot, we use the `>> hold on` command, as in the example below:

```
>> nn=20; r=2.8; x(1)=0.1;
>> for n=1:nn-1
   x(n+1)=r*x(n)*(1-x(n));
   end
>> plot(1:nn,x,'b'); title('CFB: r=2.8'); axis([1 20 0 1]);
>> hold on
>> x(1)=0.3;
>> for n=1:nn-1
   x(n+1)=r*x(n)*(1-x(n));
   end
>> plot(1:nn,x,'r')
```

A quicker way to proceed is to use *Notepad* to create the following script file `r28.m`, which time-steps three different solutions $x_{m,n}$ $(m = 1, \cdots 5)$ corresponding to three different initial conditions $x_{m,1}$ up to time $n = 20$:

```
nn=20; r=2.8;
x(1,1)=0.1; x(2,1)=0.3; x(3,1)=0.5;
for k=1:5
    for n=1:nn-1
        x(k,n+1)=r*x(k,n)*(1.0-x(k,n));
    end
end
plot(1:nn,x(1,1:nn),1:nn,x(2,1:nn),1:nn,x(3,1:nn)
title('CFB: r=2.8'); axis([1 20 0 1]);
```

7.2 Period-doubling transition to chaos

Computer experiments reveal that, for $r < 3$, the solution x_n of the logistic map is steady. If $r > 3$, x_n oscillates between two values (irrespectively of the initial condition) with period $T = 2$, see Fig. 12. This kind of oscillation in which x_n repeats every two steps is called a *period-2 cycle*.

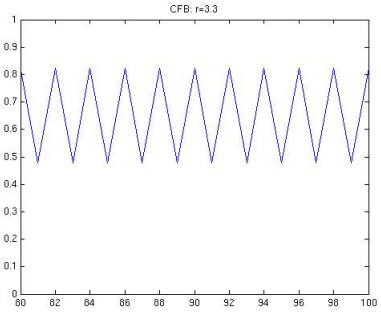

Figure 12: Period-2 solution of the logistic equation for $r = 3.3$, plotted for $80 \leq n \leq 100$, after the initial transient

.

A further increases of r, for example to $r = 3.5$, brings the solution into a regime where x_n repeats after $T = 4$, as shown in Fig. 13. This is called a *period-4 cycle*.

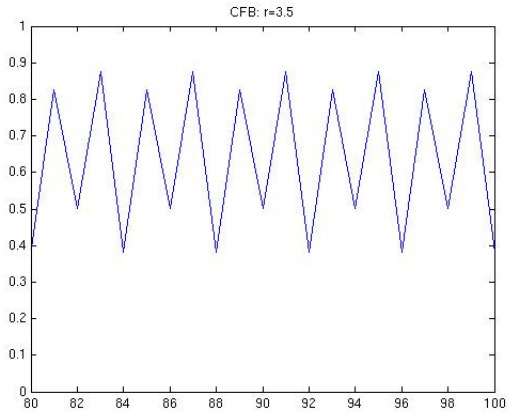

Figure 13: Period-4 solution of the logistic equation for $r = 3.5$, plotted for $80 \leq n \leq 120$, after the initial transient.

Further smaller increases of r induce more period-doubling transitions, until, if r exceeds the critical value $r_c = 3.569946$, the period T becomes infinite. This means that the values x_n never repeat exactly, and depend on the precise initial condition. We call this regime *chaos*.

In summary we have the following *period-doubling sequence*:

- At $r = 3$ period 2 is born
- At $r = 3.449$ period 4 is born
- Ar $r = 3.54409$ period 8 is born
- At $r = 3.5644$ period 16 is born
- At $r = 568759$ period 32 is born
- $\cdots$
- At $r = r_c = 3.569946$ period ∞ (chaos) is born

Fig. 14 shows two chaotic solutions at $r = 3.9$: the blue line corresponds to the initial condition $x_1 = 0.1$, the red line to $x_1 = 0.100001$. The two initial conditions differ by one part in one million only; the corresponding solutions cannot be distinguished if $n < 20$, but for $n > 25$ they are very different. This extreme sensitivity on the initial conditions is the hallmark of chaos.

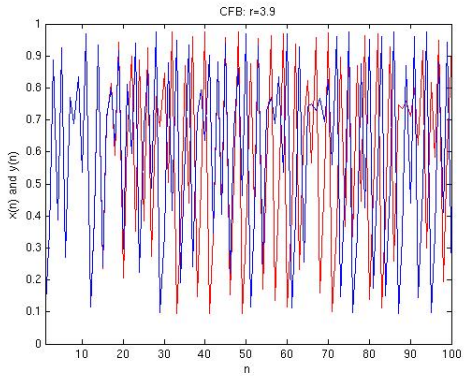

Figure 14: Solutions x_n and y_n of the logistic map for $r = 3.9$ corresponding to the initial conditions $x_1 = 0.1$ and $y_1 = 0.100001$. Note that initially x_n and y_n are essentially the same, but at later times they are very different.

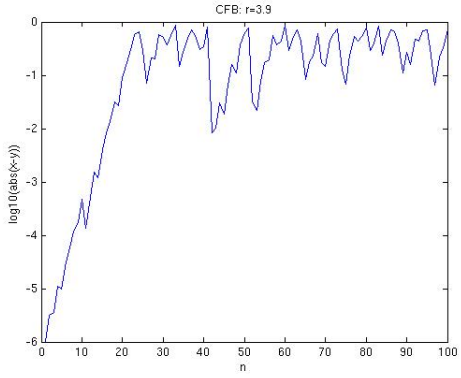

Figure 15: Semilog plot of the separation $d_n = |x_n - y_n|$ of the two solutions x_n and y_n plotted in Fig. 14.

The difference $d_n = |x_n - y_n|$ between the solution x_n (which corresponds to the initial condition $x_1 = 0.1$) and the solution y_n (which corresponds to the initial condition $y_1 = 0.100001$) is plotted vs n in Fig. 15. The logscale highlights the initially rapid (exponential) increase of d_n. Fig. 14 is created using the following script file:

```
nn=100; r=3.9;
x(1)=0.1;
y(1)=0.100001;
d(1)=abs(x(1)-y(1));
for n=1:nn-1
       x(n+1)=r*x(n)*(1.0-x(n));
\texnoindent
       y(n+1)=r*y(n)*(1.0-y(n));
       d(n+1)=abs(x(n+1)-y(n+1));
end
plot(1:nn,x(1:nn),'r-',1:nn,y(1:nn),'b-')
title('CFB: r=3.9')
axis([1 100 0 1]);
xlabel('n')
ylabel('x(n) and y(n)')
```

Fig. 15 is then obtained by typing:

```
>> plot(1:nn,log10(d(1:nn)))
>> ylabel('log10(abs(x-y))')
>> xlabel('n')
>> title('CFB: r=3.9')
```

Fig. 16 plots the long-term values of x_n (that is to say the values past any initial transient) against the parameter r. The transitions from steady solution to period-2, from period-2 to period-4, from period-4 to period-8 are clearly visible. Further period-doubling transitions take place in a narrow range of r-space, just before $r_c = 3.569946$, and are not visible on this scale. Fig. 17 zooms into the previous figure to show more details: period-16 is visible at the right:

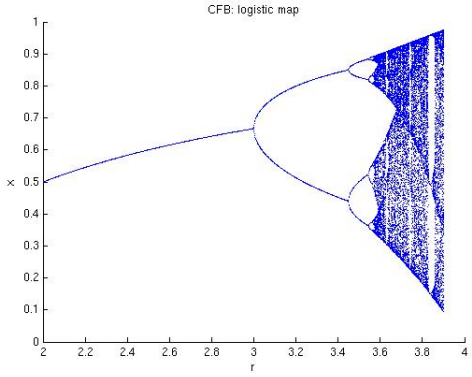

Figure 16: Long-term values x_n corresponding to r.

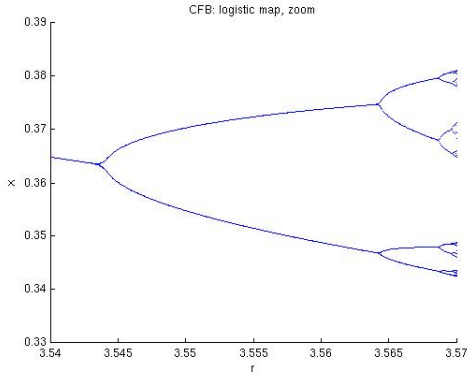

Figure 17: Detail of Fig. 16.

The script file which produces Fig. 16 is

```
nn=1000;
r=linspace(2.0,3.9,1000)
hold on
for k=1:length(r);
   x(1)=0.1;
   for n=1:nn-1
       x(n+1)=r(k)*x(n)*(1.0-x(n));
       if(n>0.9*nn)
          plot(r(k),x(n),'b.','MarkerSize',1)
       end
   end
end
title('CFB: logistic map'); xlabel('r'); ylabel('x');
```

8 Numerical solution of differential equations

8.1 Euler's method

Consider the following differential equation for $x(t)$

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

where t is time and the function f is assigned. Given the initial condition $x(0)$ at time $t = 0$, we want to find the solution $x(t)$ at later times.

We discretize the time span of interest into a large number of small time steps t_n ($n = 1, 2, \cdots$) of size $h \ll 1$. Let $x_n = x(t_n)$ be the approximate (numerical) solution to the differential equation at time t_n. We need a rule to time-step the solution, starting from the initial value, to values at later times.

A simple recursion formula is derived in the following way. Integrate $dx/dt = f$ between time t_n and time t_{n+1}:

$$\int_{t_n}^{t_{n+1}} \frac{dx}{dt} dt = \int_{t_n}^{t_{n+1}} f(t, x) dt,$$

The definite integral at the left-hand-side can be evaluated exactly and we have $x_{n+1} - x_n = \int_{t_n}^{t_{n+1}} f(t, x)dt$. The right-hand-side is the area under the curve $f(t, x(t))$ between t_n and t_{n+1}. Since $t_{n+1} - t_n = h \ll 1$, this area is approximately equal to the area of the rectangle of base h and height $f_n = f(t_n, x_n)$ (see Fig. 18), hence $x_{n+1} - x_n \approx hf_n$. We obtain **Euler's method**:

$$x_{n+1} = x_n + hf_n.$$

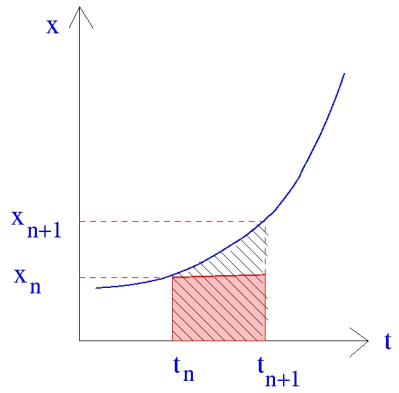

Figure 18: The pink area is the approximation to the true area, which is indicated by the diagonal lines.

8.2 Euler's method with Matlab

We want to solve the equation $dx/dt = -xt$ with initial condition $x(0) = 1$ at $t = 0$. We apply Euler's formula and obtain $x_{n+1} = x_n + hf_n = x_n + h(-x_n t_n)$. Using *Notepad*, we create the following script file called `euler0.m`:

```
% To solve dx/dt=-xt  with Euler method
clear all;                    % Clear all variables
nn=10;                        % Number of time steps
h=0.1;                        % Time step
x(1)=1;                       % Initial x
t(1)=0;                       % Initial t
for n=1:nn-1                  % Time loop
    fun=-x(n)*t(n);           % Get RHS at old time
    x(n+1)=x(n)+h*fun;        % Get new x
    t(n+1)=t(n)+h;            % get new t
end
plot(t,x)                     % Plot x vs t
title('CFB'); xlabel('t'), ylabel('x')
```

At Matlab's prompt we type `>> euler0` and obtain the graph shown in fig. 19: The last value computed (at $n = 11$) is the solution $x = 0.6282$ at $t = 1$. The exact solution of the equation is $x(t) = x(0)e^{-t^2/2}$, hence $x(1) = e^{-0.5} = 0.6065$. So the relative difference between the approximate (numerical) solution and the exact (analytical) solution is $(0.6282 - 0.6065)/0.6065 \approx 6\%$. If we reduce the time step h the relative error is less, that is to say our numerical solution is more accurate.

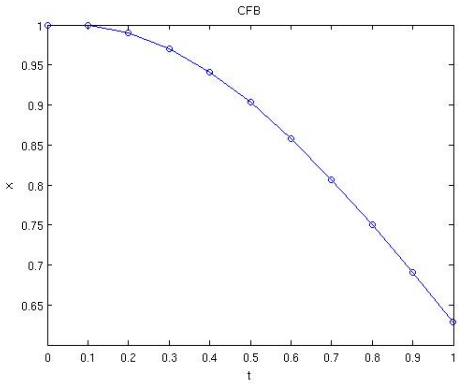

Figure 19: Numerical solution of $dx/dt = -xt$ with initial condition $x(0) = 1$; the time step is $h = 0.1$, hence 11 values (marked by the small circles) cover the range from $t = 0$ to $t = 1$.

8.3 Euler's method for a system of equations

It is easy to generalise what we have done to a system of differential equations. Consider the 2-dimensional dynamical system

$$\frac{dx_1}{dt} = x_2, \qquad \frac{dx_2}{dt} = -x_1 - 0.2x_2,$$

Create the following script file called `euler1.m`:

```
% To solve the system
% dx1/dt=x2
% dx2/dt=-x1-0.2*x2
% with Euler's method
clear all                       % Clear all variables
nn=1000;                        % Number of time steps
h=0.05;                         % Time step
x1(1)=1.0;                      % Initial x1
x2(1)=0.0;                      % Initial x2
t(1)=0;                         % Initial t
for n=1:nn-1                    % Time loop
    fun1=x2(n);                 % Get RHS of 1st eq at old time
    fun2=-x1(n)-0.2*x2(n);      % Get RHS of 2nd eq at old time
    x1(n+1)=x1(n)+h*fun1;       % Get new x1
    x2(n+1)=x2(n)+h*fun2;       % Get new x2
    t(n+1)=t(n)+h;              % get new t
end
plot(t,x1)                      % Plot x1 vs t
title('CFB'); xlabel('t'), ylabel('x1')
```

At Matlab's prompt, type `euler1`, and obtain the graph shown in Fig. 20, which shows x_1 vs t. The corresponding plot of x_2 vs x_1, shown in Fig. 21, is generated in a similar way.

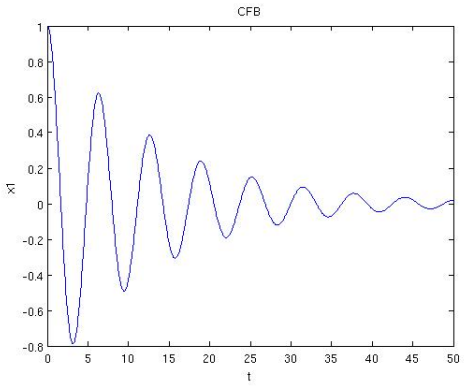

Figure 20: Numerical solution of $dx_1/dt = x_2$, $dx_2/dt = -x_1 - x_2/5$ with initial condition $x_1(0) = 1$, $x_2 = 0.$; the time step is $h = 0.05$. Plot of x_1 vs t.

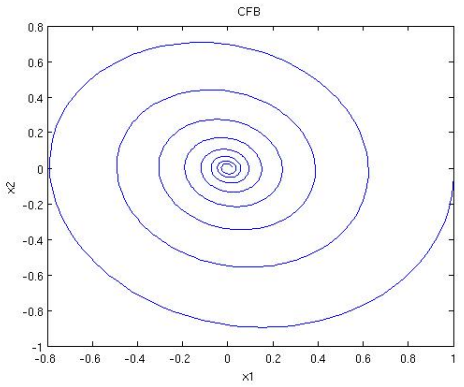

Figure 21: Plot of x_2 vs x_1 corresponding to Fig. 20.

9 The Lorenz equations

The Lorenz equations for $x(t)$, $y(t)$ and $z(t)$ are

$$\frac{dx}{dt} = \sigma(y - x), \qquad \frac{dy}{dt} = rx - y - xz, \qquad \frac{dz}{dt} = xy - bz,$$

where σ, r and b are parameters. We choose $\sigma = 10$, $b = 8/3$ and $r = 28$ and let the initial conditions be $x(0) = y(0) = z(0) = 0.1$ at $t = 0$. We choose time step be $h = 0.001$ and total number of steps be $nn = 10000$. Using *Notepad*, we create the following script file `eulerlorenz.m`:

```
% Aim: to solve the Lorenz equations
% dx/dt=sigma*(y-x); dy/dt=-x*z+r*x-y; dz/dt=x*y-b*z
sig=10.0; b=8/3; r=20;           % Parameters
t(1)=0.0;                        % Initial t
x(1)=0.1;i y(1)=0.1; z(1)=0.1;   % Initial x,y,z
dt=0.005;                        % Time step
nn=10000;                        % Number of time steps
for k=1:nn                       % Time loop
fx=sig*(y(k)-x(k));              % RHS of x equation
fy=-x(k)*z(k)+r*x(k)-y(k);       % RHS of y equation
fz=x(k)*y(k)-b*z(k);             % RHS of z equation
x(k+1)=x(k)+dt*fx;               % Find new x
y(k+1)=y(k)+dt*fy;               % Find new y
z(k+1)=z(k)+dt*fz;               % Find new z
t(k+1)=t(k)+dt;                  % Find new t
end                              % Close time loop
plot(t,x)                        % Plot x vs t
title('CFB')                     % Title
xlabel('t'); ylabel('x');        % Label axes
```

At Matlab's prompt we type `eulerlorenz` and obtain the graph shown in Fig. 22. If we type `plot3(x,y,z)` we obtain a three-dimensional plot, see Fig. 23. which we can rotate byy clicking the button on the toolbar of the graphics window.

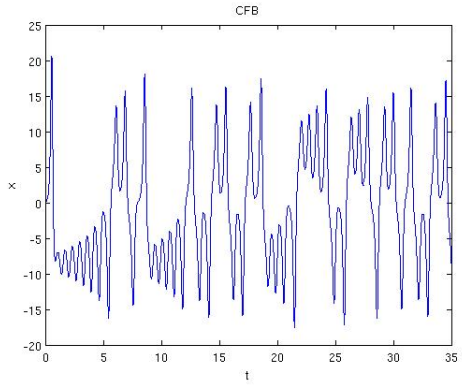

Figure 22: Solution of the Lorenz equations for $\sigma = 10$, $b = 8/3$, $r = 28$, initial condition $x(0) = y(0) = z(0) = 0.1$, using Euler's method with time step $h = 0.005$ for 7000 steps.

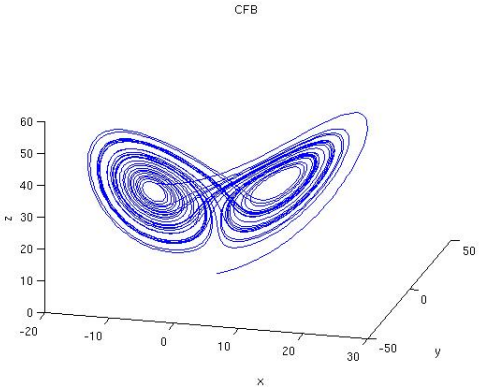

Figure 23: Three-dimensional plot of $x(t)$, $y(t)$ and $z(t)$ corresponding to Fig. 22.

10 Using ode45

Euler's method is simple but not very accurate. Matlab has better built-in functions to solve differential equations. The most popular is *ode45*. For example, let us solve the differential equation $dx/dt = x - x^2$. First we use *Notepad* to create the following file tuto1.m which contains the right hand side of the equation:

```
function dx=tuto1(t,x)
dx=x-x^2
```

Secondly, we create the following file called `maintuto1.m`:

```
% To solve the equation dx/dt=x-x^2; It uses the file tuto1.m
t=[0 10];                              % Time span
xinit=[0.1];                           % Initial condition
[t,x]=ode45(@tuto1,t,xinit);           % Integrate equation
plot(t,x)                              % Plot solution
title('CFB'); xlabel('t'); ylabel('x(t)');
```

`maintuto1.m` sets the time span of integration and the initial condition, calls *ode45* using the function `tuto1.m`, and plots the solution. In general, the call to *ode45* has the form

```
[t,x]=ode45(@fname, tspan, xinit, options)
```

where `fname` is the name of the function (the m-file which contains the right hand side of the equation), `tspan` is a vector which defines the beginning and the end limits of integration, `xinit` is a vector of initial conditions, and `options` is not used here.

Thirdly, at Matlab's prompt we type `maintuto1` and obtain the graph shown in Fig. 24.

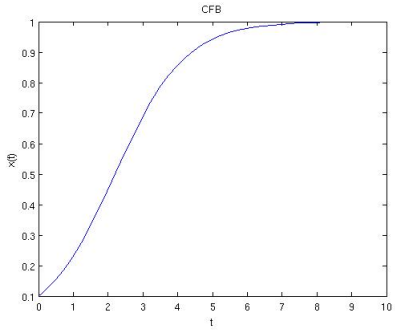

Figure 24: Solution of $dx/dt = x - x^2$ using Matlab's *ode45*.

Let is use `ode45` to solve the Lorenz equations. First, using *Notepad*, we create the following file *lorenz.m*:

```
function dx=lorenz(t,x)
% Parameters
sigma=10; r=28; b=8/3;
%Right hand sides
dx1=sigma*(x(2)-x(1));
dx2=r*x(1)-x(2)-x(1)*x(3);
dx3=x(1)*x(2)-b*x(3);
%Put togther the RHS vector
dx=[dx1;dx2;dx3];
```

Secondly, again using *Notepad*, we create the following file *mainlorenz.m*:

```
% To solve the Lorenz equations. It uses the file lorenz.m
clear all                          % Clear all variables
t=[0 50];                          % Time window
xinit=[0.1;0.1;0.1];               % Initial condition
[t,x]=ode45(@lorenz,t,xinit);      % Integrate in time
plot3(x(:,1),x(:,2),x(:,3))        % Plot solution
title('CFB'); xlabel('x'); ylabel('y'); zlabel('z');
```

Thirdly, at Matlab's prompt, we type *mainlorenz* and obtain a 3-dimensional plot of $x(t)$, $y(t)$ and $z(t)$.
It is apparent that *ode45*, besides being more accurate than Euler's method, is faster to use.

10.1 Solved exercises

Exercise 10.1
Use *ode45* to solve the system of differential equations

$$\frac{dx_1}{dt} = x_2, \qquad \frac{dx_2}{dt} = -x_1,$$

for $0 < t < 10$ given the initial condition $x_1(0) = 1$, $x_2(0) = 0$. (the solution is at the end of the section).
Solution: Firstly, using *Notepad*, create the following file `tuto2.m`:

```
function dx=tuto2(t,x)
dx1=x(2);
dx2=-x(1);
dx=[dx1;dx2];
```

Secondly, again with *Notepad*, create the following file `maintuto2.m`:

```
% To solve the system of equations dx1/dt=x2; dx2/dt=-x11
% It uses the file tuto2.m
clear all                                % Clear all
t=[0 10];                                % Time span
xinit=[1;0];                             % Initial condition
[t,x]=ode45(@tuto2,t,xinit);             % Integrate equations
plot(t,x(:,1))                           % Plot x1 vs t
title('CFB'); xlabel('t'); ylabel('x1');
%plot(t,x(:,2))                          % Plot x2 vs t
%title('CFB'); xlabel('t'); ylabel('x2');
%plot(x(:,1),x(:,2))                     % Plot x2 vs x1
%title('CFB'); xlabel('x1'); ylabel('x2)');
```

Thirdly, at Matlab's prompt type `>> maintuto2` and obtain the graph shown in Fig. 25. By using the lines at the bottom of `maintuto2` which at the moment are blanked out, we can generate the graphs shown in Fig. 26 and 27.

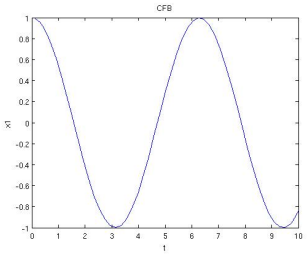

Figure 25: Solution of $dx_1/dt = x_s$, $dx_2/dt = -x_2$ with initial condition $x_1(0) = 1$, $x_2(0) = 1$ using Matlab's *ode45*. Plot of x_1 vs t.

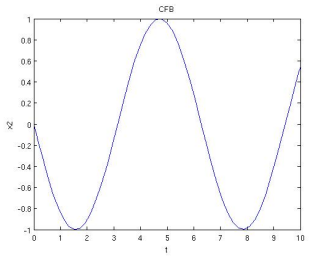

Figure 26: Plot of x_2 vs t corresponding to Fig. 25.

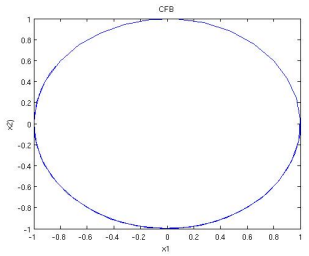

Figure 27: Plot of x_2 vs x_1 corresponding to Fig. 25.

Exercise 10.2
Solve the Lorenz system using *ode45* for $\sigma = 10$, $r = 28$, $b = 8/3$, first with initial condition $x(0) = 0.1$, $y(0) = 1.2$, $z(0) = 1.2$, then for perturbed initial condition $x(0) = 0.101$, $y(0) = 1.2$, $z(0) = 1.2$. In both cases the tipe span should be from $t = 0$ to $t = 50$. Plot $x$vs t corresponding to the two diffeent initial conditions on the same graph. Then plot $\log_{10}(d(t))$ vs t, where $d(t)$ is the distance between the two trajectories at time t, that is $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$.

Solution:
The following script file

```
% To solve the Lorenz equations and show the butterfly effect
% It uses the file lorenz.m
clear all                          % Clear all variables
t=[0 50];                          % Time window
x1init=[0.1;1.2;1.2];              % Initial condition for x1
[t,x1]=ode45(@lorenz,t,x1init);    % Find first trajectory x1
x2init=[0.101;1.2;1.2];            % Initial condition for x2
[t,x2]=ode45(@lorenz,t,x2init);    % Find second trajectory x2

nn=length(t);                      % Number of time steps used
for n=1:nn                         % Loop to find the separation d
    dx=x2(n,1)-x1(n,1);            % between x1 and x2
    dy=x2(n,2)-x1(n,2);
    dz=x2(n,3)-x1(n,3);
    d(n)=sqrt(dx*dx+dy*dy+dz*dz);
end
% Plot x1 and x2 vs t
%plot(t,x1(:,1),'b')
%hold on
%plot(t,x2(:,1),'r')
%title('CFB'); xlabel('t'); ylabel('x1 and x2');

% Plot log(d) vs t
plot(t,log10(d))                   % Plot log(d) vs time
title('CFB'); xlabel('t'); ylabel('log10(d)');
```

is used to produce the graphs shown in Figs. 28 and 29.

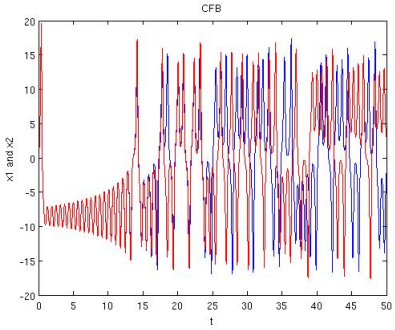

Figure 28: Solution of the Lorenz equation for $\sigma = 10$, $r = 28$, $b = 8/3$ and initial condition $x(0) = 0.1$, $y(0) = 1.2$, $z(0) = 1.2$ (in blue), and then with perturbed initial condition $x(0) = 0.101$, $y(0) = 1.2$, $z(0) = 1.2$ (in red), obtained using Matlab's *ode45* function.

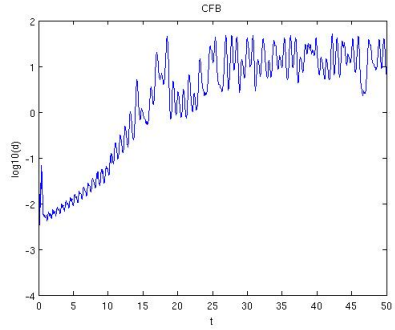

Figure 29: Separation $d(t)$ vs t between the two solutions shown in Fig. 28.