

Introduction to Octave

Dr. P.J.G. Long
Department of Engineering
University of Cambridge

Based on the Tutorial Guide to
Matlab written by Dr. Paul Smith

September 2005

This document provides an introduction to computing using Octave. It will teach you how to use Octave to perform calculations, plot graphs, and write simple programs.

The close compatibility of the open-source Octave¹ package with MATLAB², which is heavily used in industry and academia, gives the user the opportunity to learn the syntax and power of both packages where funding and licence restrictions prevent the use of commercial packages.

To maintain the ideal of learning both Octave and Matlab from this tutorial, the differences between Octave and Matlab have been highlighted and details of any modifications etc. required to run a function/program with Matlab described in footnotes. In a number of cases additional functions have had to be written or startup options set, these are included as default on the MDP³ distribution and documented in an appendix.

Draft version 0.1.0, please email any errors to mdp-support@eng.cam.ac.uk

¹www.octave.org

²MATLAB® The Mathworks, www.mathworks.com

³Multidisciplinary Design Project

Contents

1	Introduction	4
1.1	What is OCTAVE?	4
1.2	What OCTAVE is not	4
1.3	Who uses OCTAVE?	4
1.4	Why not use a ‘normal’ highlevel language, e.g. C++	4
2	Simple calculations	5
2.1	Starting OCTAVE	5
2.2	OCTAVE as a calculator	5
2.3	Built-in functions	6
3	The Octave environment	6
3.1	Named variables	8
3.2	Numbers and formatting	9
3.3	Number representation and accuracy	10
3.4	Loading and saving data	10
3.5	Repeating previous commands	11
3.6	Getting help	11
3.7	Cancelling a command	13
3.8	Semicolons and hiding answers	13
4	Arrays and vectors	13
4.1	Building vectors	13
4.2	The colon notation	14
4.3	Displaying large vectors and matrices	14
4.4	Vector creation functions	15
4.5	Extracting elements from a vector	15
4.6	Vector maths	16
5	Plotting graphs	17
5.1	Improving the presentation	18
5.2	Multiple graphs	19
5.3	Multiple figures	20
5.4	Manual scaling	21
5.5	Saving and printing figures	22
6	Octave programming I: Script files	23
6.1	Creating and editing a script	23
6.2	Running and debugging scripts	24
6.3	Remembering previous scripts	24
7	Control statements	25
7.1	if...else selection	25
7.2	switch selection	26
7.3	for loops	27
7.4	while loops	28

7.5	Accuracy and precision	28
8	Octave programming II: Functions	29
8.1	Example 1: Sine in degrees	30
8.2	Creating and using functions	30
8.3	Example 2: Unit step	31
9	Matrices and vectors	33
9.1	Matrix multiplication	34
9.2	The transpose operator	35
9.3	Matrix creation functions	35
9.4	Building composite matrices	37
9.5	Matrices as tables	37
9.6	Extracting bits of matrices	37
10	Basic matrix functions	38
11	Solving $Ax = b$	40
11.1	Solution when A is invertible	40
11.2	Gaussian elimination and LU factorisation	41
11.3	Matrix division and the slash operator	41
11.4	Singular matrices and <code>rank</code>	41
11.5	Ill-conditioning	43
11.6	Over-determined systems: Least squares	44
11.7	Example: Triangulation	44
12	More graphs	45
12.1	Putting several graphs in one window	45
12.2	3D plots	46
12.3	Changing the viewpoint	46
12.4	Plotting surfaces	47
12.5	Images and Movies	48
13	Eigenvectors and the Singular Value Decomposition	49
13.1	The <code>eig</code> function	49
13.2	The Singular Value Decomposition	50
13.3	Approximating matrices: Changing rank	51
13.4	The <code>svd</code> function	51
13.5	Economy SVD	52
14	Complex numbers	53
14.1	Plotting complex numbers	54
14.2	Finding roots of polynomials	54
15	Appendix - Setup conditions	55
16	Further reading	55
17	Acknowledgements	56

1 Introduction

1.1 What is Octave?

OCTAVE is an open-source interactive software system for numerical computations and graphics. It is particularly designed for matrix computations: solving simultaneous equations, computing eigenvectors and eigenvalues and so on. In many real-world engineering problems the data can be expressed as matrices and vectors, and boil down to these forms of solution. In addition, OCTAVE can display data in a variety of different ways, and it also has its own programming language which allows the system to be extended. It can be thought of as a very powerful, programmable, graphical calculator. OCTAVE makes it easy to solve a wide range of numerical problems, allowing you to spend more time experimenting and thinking about the wider problem.

Octave was originally developed as a companion software to a undergraduate course book on chemical reactor design⁴. It is currently being developed under the leadership of Dr. J.W. Eaton and released under the GNU General Public Licence. Octave's usefulness is enhanced in that it is mostly syntax compatible with MATLAB which is commonly used in industry and academia.

1.2 What Octave is not

OCTAVE is designed to solve mathematical problems *numerically*, that is by calculating values in the computer's memory. This means that it can't always give an exact solution to a problem, and it should not be confused with programs such as Mathematica or Maple, which give *symbolic* solutions by doing the algebraic manipulation. This does not make it better or worse—it is used for different tasks. Most real mathematical problems (particularly engineering ones!) do not have neat symbolic solutions.

1.3 Who uses Octave?

OCTAVE and MATLAB are widely used by engineers and scientists, in both industry and academia for performing numerical computations, and for developing and testing mathematical algorithms. For example, NASA use it to develop spacecraft docking systems; Jaguar Racing use it to display and analyse data transmitted from their Formula 1 cars; Sheffield University use it to develop software to recognise cancerous cells. It makes it very easy to write mathematical programs quickly, and display data in a wide range of different ways.

1.4 Why not use a 'normal' highlevel language, e.g. C++

C++ and other industry-standard programming languages are normally designed for writing general-purpose software. However, solutions to mathematical problems take time to program using C++, and the language does not natively support many mathematical concepts, or displaying graphics. OCTAVE is specially designed to solve these kind of problems, perform calculations, and display the results. Even people who will ultimately be writing

⁴www.octave.org/history.html

software in languages like C++ sometimes begin by prototyping any mathematical parts using OCTAVE, as that allows them to test the algorithms quickly.

OCTAVE is available on the MDP Resource CD and can be downloaded from www.octave.org if required.

2 Simple calculations

2.1 Starting Octave

If not already running start Octave, (see **start** → **Programs** → **Octave** on the MDP CD.) or type in a xterm window

```
octave
```

After a pause, a logo will briefly pop up in another window, and the terminal will display the header similar to this:

```
GNU Octave, version 2.1.57 (i386-pc-linux-gnu).
Copyright (C) 2004 John W. Eaton.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. For details, type 'warranty'.
```

Additional information about Octave is available at <http://www.octave.org>.

Please contribute if you find this software useful.

For more information, visit <http://www.octave.org/help-wanted.html>

Report bugs to <bug-octave@bevo.che.wisc.edu> (but first, please read <http://www.octave.org/bugs.html> to learn how to write a helpful report).

```
octave:1>
```

and you are now in the OCTAVE environment. The `octave:1>` is the OCTAVE prompt, asking you to type in a command.

If you want to leave OCTAVE at any point, type `quit` at the prompt.

2.2 Octave as a calculator

The simplest way to use OCTAVE is just to type mathematical commands at the prompt, like a normal calculator. All of the usual arithmetic expressions are recognised. For example, type

```
octave:##> 2+2
```

at the prompt and press return, and you should see

```
ans = 4
```

The basic arithmetic operators are `+` `-` `*` `/`, and `^` is used to mean ‘to the power of’ (e.g. $2^3=8$). Brackets `()` can also be used. The order *precedence* is the same usual i.e. brackets

are evaluated first, then powers, then multiplication and division, and finally addition and subtraction. Try a few examples.

2.3 Built-in functions

As well as the basic operators, OCTAVE provides all of the usual mathematical functions, and a selection of these can be seen in Table 1. These functions are invoked as in C++ with the name of the function and then the function *argument* (or arguments) in ordinary brackets `()`, for example⁵

```
octave:##> exp(1)
```

```
ans = 2.7183
```

Here is a longer expression: to calculate $1.2 \sin(40^\circ + \ln(2.4^2))$, type

```
octave:##> 1.2 * sin(40*pi/180 + log(2.4^2))
```

```
ans = 0.76618
```

There are several things to note here:

- An explicit multiplication sign is always needed in equations, for example between the `1.2` and `sin`.
- The trigonometric functions (for example `sin`) work in *radians*. The factor $\pi/180$ can be used to convert degrees to radians. `pi` is an example of a named *variable*, discussed in the next section.
- The function for a natural logarithm is called ‘`log`’, not ‘`ln`’.

Using these functions, and the usual mathematical constructions, OCTAVE can do all of the things that your normal calculator can do.

3 The Octave environment

As we can see from the examples so far, OCTAVE has an *command-line* interface—commands are typed in one at a time at the prompt, each followed by return. OCTAVE is an *interpreted* language, which means that each command is converted to machine code after it has been typed. In compiled languages, e.g. C++, language the whole program is typed into a text editor, these are all converted into machine code in one go using a *compiler*, and then the whole program is run. These compiled programs run more quickly than an interpreted program, but take more time to put together. It is quicker to try things out with OCTAVE, even if the calculation takes a little longer.⁶

⁵A function’s arguments are the values which are passed to the function which it uses to calculate its response. In this example the argument is the value ‘1’, so the exponent function calculates the exponential of 1 and returns the value (i.e. $e^1 = 2.7183$).

⁶OCTAVE can call external C++ functions however the functionality is less than MATLAB.

<code>cos</code>	Cosine of an angle (in radians)
<code>sin</code>	Sine of an angle (in radians)
<code>tan</code>	Tangent of an angle (in radians)
<code>exp</code>	Exponential function (e^x)
<code>log</code>	Natural logarithm (NB this is \log_e , not \log_{10})
<code>log10</code>	Logarithm to base 10
<code>sinh</code>	Hyperbolic sine
<code>cosh</code>	Hyperbolic cosine
<code>tanh</code>	Hyperbolic tangent
<code>acos</code>	Inverse cosine
<code>acosh</code>	Inverse hyperbolic cosine
<code>asin</code>	Inverse sine
<code>asinh</code>	Inverse hyperbolic sine
<code>atan</code>	Inverse tangent
<code>atan2</code>	Two-argument form of inverse tangent
<code>atanh</code>	Inverse hyperbolic tangent
<code>abs</code>	Absolute value
<code>sign</code>	Sign of the number (-1 or $+1$)
<code>round</code>	Round to the nearest integer
<code>floor</code>	Round down (towards minus infinity)
<code>ceil</code>	Round up (towards plus infinity)
<code>fix</code>	Round towards zero
<code>rem</code>	Remainder after integer division

Table 1: Basic maths functions

3.1 Named variables

In any significant calculation you are going to want to store your answers, or reuse values, just like using the memory on a calculator. OCTAVE allows you to define and use named variables. For example, consider the degrees example in the previous section. We can define a variable `deg` to hold the conversion factor, writing

```
octave:##> deg = pi/180

deg =0.017453
```

Note that the *type* of the variable does not need to be defined, unlike most high level languages e.g. in C++. All variables in OCTAVE are floating point numbers.⁷ Using this variable, we can rewrite the earlier expression as

```
octave:##> 1.2 * sin(40*deg + log(2.4^2))

ans =0.76618
```

which is not only easier to type, but it is easier to read and you are less likely to make a silly mistake when typing the numbers in. Try to define and use variables for all your common numbers or results when you write programs.

You will have already have seen another example of a variable in OCTAVE. Every time you type in an expression which is *not* assigned to a variable, such as in the most recent example, OCTAVE assigns the answer to a variable called `ans`. This can then be used in exactly the same way:

```
octave:##> new = 3*ans

new =2.2985
```

Note also that this is not the answer that would be expected from simply performing 3×0.76618 . Although OCTAVE *displays* numbers to only a few decimal places (usually five)⁸, it stores them internally, and in variables, to a much higher precision, so the answer given is the more accurate one.⁹ In all numerical calculations, an appreciation of the rounding errors is very important, and it is essential that you do not introduce any more errors than there already are! This is another important reason for storing numbers in variables rather than typing them in each time.

When defining and using variables, the capitalisation of the name is important: `a` is a different variable from `A`. There are also some variable names which are already defined and used by OCTAVE. The variable `ans` has also been mentioned, as has `pi`, and in addition `i` and `j` are also defined as $\sqrt{-1}$ (see Section 14). OCTAVE won't stop you redefining these

⁷Or strings, but those are obvious from the context. However, even strings are stored as a vector of character ID numbers.

⁸MATLAB normally displays to 4 or 5 decimal places

⁹OCTAVE stores all numbers in IEEE floating point format to double (64-bit) precision. The value of `ans` here is actually 0.766177651029692 (to 15 decimal places).

as whatever you like, but you might confuse yourself, and OCTAVE, if you do! Likewise, giving variables names like `sin` or `cos` is allowed, but also not to be recommended.

If you want to see the value of a variable at any point, just type its name and press return. If you want to see a list of all the named functions, variables¹⁰ you have created or used, type

```
octave:##> who

*** dynamically linked functions:

dispatch

*** currently compiled functions:

rem

*** local user variables:

deg new
```

You will occasionally want to remove variables from the workspace, perhaps to save memory, or because you are getting confusing results using that variable and you want to start again. The `clear` command will delete all variables, or specifying

```
clear name
```

will just remove the variable called *name*.

3.2 Numbers and formatting

We have seen that OCTAVE usually displays numbers to five significant figures. The `format` command allows you to select the way numbers are displayed. In particular, typing

```
octave:##> format long
```

will set OCTAVE to display numbers to fifteen+ significant figures, which is about the accuracy of OCTAVE's calculations. If you type `help format` you can get a full list of the options for this command. With `format long` set, we can view the more accurate value of `deg`:

```
octave:##> deg

deg = .0174532925199433

>> format short
```

¹⁰Normally MATLAB only displays the user (Your) variables + ans

The second line here returns OCTAVE to its normal display accuracy.

OCTAVE displays very large or very small numbers using *exponential notation*, for example: $13142.6 = 1.31426 \times 10^4$, which is displayed by OCTAVE as `1.3143e+04`. You can also type numbers into OCTAVE using this format.

There are also some other kinds of numbers recognised, and calculated, by OCTAVE:

Complex numbers (e.g. `3+4i`) Are fully understood by OCTAVE, as discussed further in Section 14

Infinity (`Inf`) The result of dividing a number by zero. This is a valid answer to a calculation, and may be assigned to a variable just like any other number

Not a Number (`NaN`) The result of zero divided by zero, and also of some other operations which generate undefined results. Again, this may be treated just like any other number (although the results of any calculations using it are still always `NaN`).

3.3 Number representation and accuracy

Numbers in OCTAVE, as in all computers, are stored in binary rather than decimal. In decimal (base 10), 12.25 means

$$12.25 = 1 \times 10^1 + 2 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2}$$

but in binary (base 2) it would be written as

$$1101.01 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} = 12.25$$

Using binary is ideal for computers since it is just a series of ones and zeros, which can be represented by whether particular circuits are on or off. A problem with representing numbers in a computer is that each number can only have a finite number circuits, or *bits*, assigned to it, and so can only have a finite number of digits. Consider this example:

```
octave:##> 1 - 0.2 - 0.2 - 0.2 - 0.2 - 0.2
```

```
ans = 5.5511e-017
```

The result is very small, but not exactly zero, which is the correct answer. The reason is that 0.2 cannot be *exactly* represented in binary using a finite number of digits (it is 0.0011001100...). This is for exactly the same reasons that 1/3 cannot be exactly written as a base 10 number. OCTAVE (and all other computer programs) represent these numbers with the closest one they can, but repeated uses of these approximations, as seen here, can cause problems. For this reason, you should think very carefully before checking that a number is *exactly equal* to another. It is usually best to check that it is the same to within a tolerance. We will return to this problem throughout this tutorial.

3.4 Loading and saving data

When you exit OCTAVE, you lose all of the variables that you have created. If you need to quit OCTAVE when you are in the middle of doing something, you can save your current session so that you can reload it later. If you type

```
octave:##> save anyname
```

it will save the entire workspace to a file called **anyname.mat** in your current directory (note that OCTAVE automatically appends **.mat** to the end of the name you've given it). You can then quit OCTAVE, and when you restart it, you can type

```
octave:##> load anyname
```

which will restore your previous workspace, and you can carry on from where you left off. You can also **load** and **save** specific variables. The format is

```
save filename var1 var2 ...
```

For example, if you wanted to save the **deg** variable, to save calculating it from scratch (which admittedly is not very difficult in this case!), you could type

```
octave:##> save degconv deg
```

This will save the variable into a file called **degconv.mat**. You can reload it by typing

```
octave:##> load degconv
```

OCTAVE will also load data from text files, which is particularly useful if you want to plot or perform calculations on measurements from some other source. The text file should contain rows of space-separated numbers.

3.5 Repeating previous commands

OCTAVE keeps a record of all the commands you have typed during a session, and you can use the cursor keys \uparrow and \downarrow to review the previous commands (with the most recent first). If you want to repeat one of these commands, just find it using the cursor keys, and press return.

Once a command has been recalled, you can edit it before running it again. You can use \leftarrow and \rightarrow to move the cursor through the line, and type characters or hit **Del** to change the contents. This is particularly useful if you have just typed a long line and then OCTAVE finds an error in it. Using the arrow keys you can recall the line, correct the error, and try it again.¹¹

3.6 Getting help

If you are not sure what a particular OCTAVE command does, or want to find a particular function, OCTAVE contains an integrated help system. The basic form of using help is to type

```
help commandname
```

¹¹On some systems the emacs key bindings are acknowledged, e.g. $\langle\text{ctrl}\rangle\text{p}=\uparrow$, $\langle\text{ctrl}\rangle\text{n}=\downarrow$, $\langle\text{ctrl}\rangle\text{f}=\rightarrow$, $\langle\text{ctrl}\rangle\text{b}=\leftarrow$

For example:

```
octave:1> help sqrt
sqrt is a built-in function
```

```
- Mapping Function:  sqrt (X)
  Compute the square root of X. If X is negative, a complex
  result is returned. To compute the matrix square root, see
  *Note Linear Algebra::.
```

Overloaded function

```
gsqrt(galois,...)
```

Additional help for built-in functions, operators, and variables is available in the on-line version of the manual. Use the command ‘help -i <topic>’ to search the manual index.

Help and information about Octave is also available on the WWW at <http://www.octave.org> and via the help-octave@bevo.che.wisc.edu mailing list.

¹²If you don’t know the name of the command you want, there are a number method to find if it exists. Typing `help -i` at the prompt will give a list of all the main areas of help.

More detailed information on a topic can be obtained by moving the cursor of the item of interest and pressing <return>. The list can be navigated using the key bindings **Up**, **Next**, **Previous**. etc. Or directly from the prompt, e.g. to find out about arithmetic functions type;

```
octave:##> help -i arithmetic
```

Use the letter ‘**q**’ to **quit from the help system** and return to the Octave command prompt.

Note that often the help information gives an indication of which area to search for further information, in the `help sqrt` example it is suggested to search the ‘Linear Algebra’ area, e.g.

```
octave:##> help -i linear algebra
```

Take some time browsing around the help system and anyone line help/manuals to get an idea of the commands that OCTAVE provides.¹³

¹²MATLAB help messages tend to be shorter, but always indicate the command in UPPER CASE. Don’t copy these exactly—OCTAVE and MATLAB commands are almost always in lower case, e.g. the square root function is `sqrt()`, *not* `SQRT()`

¹³MATLAB has some additional search facilities including the function `lookfor` which allows the user to search the help database

3.7 Cancelling a command

If you find yourself having typed a command which is taking a long time to execute (or perhaps it is one of your own programs which has a bug which causes it to repeat endlessly), it is very useful to know how to stop it. You can cancel the current command by typing

`Ctrl-C`

which should (perhaps after a small pause) return you to the command prompt.

3.8 Semicolons and hiding answers

Semicolons ‘;’ are often used in programming languages to separate functions, denote line-ends, e.g. C++ where ‘;’ are added to almost every line. Semicolons are not required in OCTAVE, but do serve a useful purpose. If you type the command as we have been doing so far, without a final semicolon, OCTAVE always displays the result of the expression. However, if you finish the line with a semicolon, it stops OCTAVE displaying the result. This is particularly useful if you don’t need to know the result there and then, or the result would otherwise be an enormous list of numbers:

4 Arrays and vectors

Many mathematical problems work with sequences of numbers. In many languages they are called *arrays*; in OCTAVE they are just examples of *vectors*. Vectors are commonly used to represent the three dimensions of a position or a velocity, but a vector is really just a list of numbers, and this is how OCTAVE treats them. In fact, vectors are a simple case of a *matrix* (which is just a two-dimensional grid of numbers). A vector is a matrix with only one row, or only one column. We will see later that it is often important to distinguish between *row* vectors (\cdots) and *column* vectors $\begin{pmatrix} \cdot \\ \cdot \\ \cdot \end{pmatrix}$, but for the moment that won’t concern us.

4.1 Building vectors

There are lots of ways of defining vectors and matrices. Usually the easiest thing to do is to type the vector inside *square* brackets [], for example

```
octave:##> a=[1 4 5]
```

```
a =  
  1      4      5
```

```
octave:##> b=[2,1,0]
```

```
b =  
  2      1      0
```

```
octave:##> c=[4;7;10]
```

```
c =
```

```
4
7
10
```

A list of numbers separated by spaces or commas, inside square brackets, defines a row vector. Numbers separated by semicolons, or carriage returns, define a column vector.

You can also construct a vector from an existing vector by including it in the definition, for example

```
octave:##> a=[1 4 5]
a =
    1     4     5
```

```
octave:##> d=[a 6]
d =
    1     4     5     6
```

4.2 The colon notation

A useful shortcut for constructing a vector of counting numbers is using the colon symbol ‘:’, as in this example

```
octave:##> e=2:6
e =
    2     3     4     5     6
```

The colon tells OCTAVE to create a vector of numbers starting from the first number, and counting up to (and including) the second number. A third number may also be added between the two, making $a : b : c$. The middle number then specifies the increment between elements of the vector.

```
octave:##> e=2:0.3:4
e =
    2.0000    2.3000    2.6000    2.9000    3.2000    3.5000    3.8000
```

Note that if the increment step is such that it can’t exactly reach the end number, as in this case, it generates all of the numbers which do not exceed it. The increment can also be negative, in which case it will count down to the end number.

4.3 Displaying large vectors and matrices

If you ask Octave to display a matrix or vector that will not fit onto a single screen that it will present the values one page at a time. Try

```
octave:##> v = 1:1000
```

<code>zeros(M, N)</code>	Create a matrix where every element is zero. For a row vector of size n , set $M = 1, N = n$
<code>ones(M, N)</code>	Create a matrix where every element is one. For a row vector of size n , set $M = 1, N = n$
<code>linspace(x1, x2, N)</code>	Create a vector of N elements, evenly spaced between $x1$ and $x2$
<code>logspace(x1, x2, N)</code>	Create a vector of N elements, logarithmically spaced between 10^{x1} and 10^{x2}

Table 2: Vector creation functions

Press the space bar to see the next page of values and use the ‘q’ key to quit the display and return to the Octave command prompt. You can also use the ‘b’ key to scroll backwards up through the values being displayed.

It is sometimes convenient to turn off this pagination facility, for example when displaying intermediate values during a long calculation. This can be achieved using the

```
octave:##> more off
```

command. As you would expect, pagination can be turned back on again using

```
octave:##> more on
```

4.4 Vector creation functions

OCTAVE also provides a set of functions for creating vectors. These are outlined in Table 2. The first two in this table, `zeros` and `ones` also work for matrices, and the two function arguments, M and N , specify the number of *rows* and *columns* in the matrix respectively. A row vector is a matrix which has one row and as many columns as the size of the vector. Matrices are covered in more detail in Section 9.

4.5 Extracting elements from a vector

Individual elements are referred to by using normal brackets `()`, and they are numbered starting at *one*, not zero as in C++. If we define a vector

```
octave:##> a=[1:2:6 -1 0]
a =
    1     3     5    -1     0
```

then we can get the third element by typing

```
octave:##> a(3)
ans =
    5
```

The colon notation can also be used to specify a range of numbers to get several elements at one time

```
octave:##> a(3:5)
ans =
      5      -1       0
```

```
octave:##> a(1:2:5)
ans =
      1       5       0
```

4.6 Vector maths

Storing a list of numbers in one vector allows OCTAVE to use some of its more powerful features to perform calculations. In C++ if you wanted to do the same operation on a list of numbers, say you wanted to multiply each by 2, you would have to use a `for` loop to step through each element. This can also be done in OCTAVE (see Section 7), but it is much better to make use of OCTAVE's vector operators.

Multiplying all the numbers in a vector by the same number, is as simple as multiplying the whole vector by number. This example uses the vector `a` defined earlier:

```
octave:##> a * 2
ans =
      2       6      10      -2       0
```

The same is also true for division. You can also add the same number to each element by using the `+` or `-` operators, although this is not a standard mathematical convention.

Multiplying two vectors together in OCTAVE follows the rules of matrix multiplication (see Section 9), which doesn't do an element-by-element multiplication.¹⁴ If you want to do this, OCTAVE defines the operators `.*` and `./`, for example

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} .* \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_1 b_1 \\ a_2 b_2 \\ a_3 b_3 \end{pmatrix}$$

Note the `.'` in front of each symbol, which means it's a element-by-element operation. For example, we can multiply each of the elements of the vector `a`, defined earlier, by a different number:

```
octave:##> b=[1 2 3 4 5];

octave:##> a.*b
ans =
```

¹⁴Recall that the only vector products mathematically defined are the dot and cross product, both of which represent particular operations on two vectors, neither of which just multiply the elements together and return another vector.


```
1      6      15     -4      0
```

The element-by-element ‘to the power of’ operator `.^` is particularly useful. It can be used to raise a vector of numbers to a power, or to raise a number to different powers, depending on how it is used:

```
octave:##> b .^ 2
ans =
    1     4     9    16    25

octave:##> 2 .^ b
ans =
    2     4     8    16    32
```

The first example squares each element of `b`; the second raises 2 to each of the powers given in `b`.

All of the element-by-element vector commands (`+` `-` `./` `.*` `.^`) can be used between two vectors, as long as they are the *same size and shape*. Otherwise corresponding elements cannot be found and an error is given.

Most of OCTAVE’s functions also know about vectors. For example, to create a list of the value of sine at 60-degree intervals, you just need to pass a vector of angles to the `sin` function:

```
octave:##> angles=[0:pi/3:2*pi]
angles =
    0    1.0472    2.0944    3.1416    4.1888    5.2360    6.2832

octave:##> y=sin(angles)
y =
    0    0.8660    0.8660    0.0000   -0.8660   -0.8660   -0.0000
```

5 Plotting graphs

OCTAVE has powerful facilities for plotting graphs via a second open-source program GNUPLOT¹⁵, however some of the range of plotting options are restricted compared with MATLAB. The basic command is `plot(x,y)`, where `x` and `y` are the co-ordinates. If given just one pair of numbers it plots a point, but usually you pass *vectors*, and it plots all the points given by the two vectors, joining them up with straight lines.¹⁶ The sine curve defined in the previous section can be plotted by typing

```
octave:##> plot(angles,y)
```

¹⁵www.gnuplot.org

¹⁶The two vectors must, naturally, both be the same length.

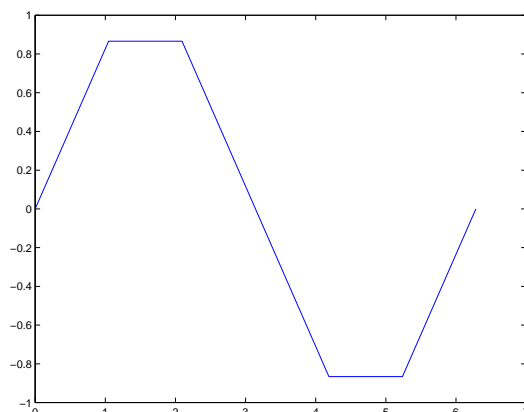


Figure 1: Graph of $y = \sin(x)$, sampled every 60° .

A new window should open up, displaying the graph, shown in Figure 1. Note that it automatically selects a sensible scale, and plots the axes.

At the moment it does not look particularly like a sine wave, because we have only taken values one every 60 degrees. To plot a more accurate graph, we need to calculate y at a higher resolution:

```
octave:##> angles=linspace(0,2*pi,100);
octave:##> y=sin(angles);
octave:##> plot(angles, y);
```

The `linspace` command creates a vector with 100 values evenly spaced between 0 and 2π (the value 100 is picked by trial and error). Try using these commands to re-plot the graph at this higher resolution. Remember that you can use the arrow keys \uparrow and \downarrow to go back and reuse your previous commands.

5.1 Improving the presentation

You can select the colour and the line style for the graph by using a third argument in the `plot` command. For example, to plot the graph instead with red circles, type

```
octave:##> plot(angles, y, 'ro')
```

The last argument is a string which describes the desired styles. Table 3 shows the possible values (also available by typing `help plot` in OCTAVE).

To put a title onto the graph, and label the axes, use the commands `title`, `xlabel` and `ylabel`:

```
octave:##> title('Graph of y=sin(x)')
octave:##> xlabel('Angle')
octave:##> >> ylabel('Value')
```

Strings in OCTAVE (such as the names for the axes) are delimited using apostrophes (`'`).

w	whitew	.	point	-	solid
m	magenta	o	circle	:	dotted†
c	cyan	x	x-mark	-.	dashdot†
r	red	+	plus	--	dashed†
g	green	*	star		
b	blue	s	square†		
y	yellow†	d	diamond†		
k	black†	v	triangle (down)†		
		^	triangle (up)†		
		<	triangle (left)†		
		>	triangle (right)†		
		p	pentagram†		
		h	hexagram†		

Table 3: Colours and styles for symbols and lines in the `plot` command (see `help plot`).. (†N.B. Only available in MATLAB)

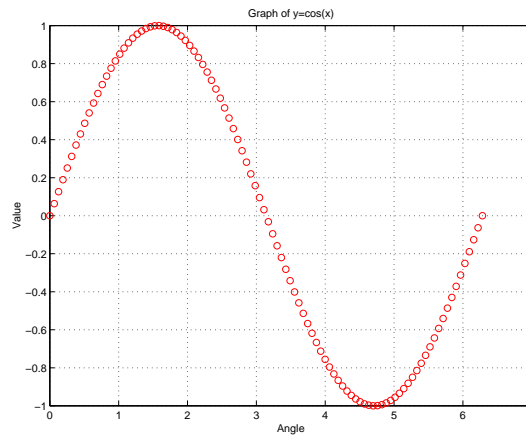


Figure 2: Graph of $y = \sin(x)$, marking each sample point with a red circle.

In some circumstances the command `replot` has to be called to enable the graph to update.

A grid may also be added to the graph, by typing

```
octave:##> grid on
```

Figure 2 shows the result. You can resize the figure or make it a different height and width by dragging the corners of the figure window.

5.2 Multiple graphs

Several graphs can be drawn on the same figure by adding more arguments to the `plot` command, giving the `x` and `y` vectors for each graph. For example, to plot a cosine curve as well as the previous sine curve, you can type

```
octave:##> plot(angles,y,':',angles,cos(angles),'-')
```

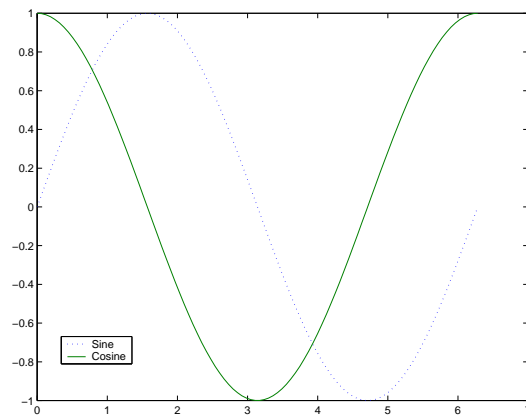


Figure 3: Graphs of $y = \sin(x)$ and $y = \cos(x)$.

where the extra three arguments define the cosine curve and its line style. You can add a legend to the plot using the `legend` command:

```
octave:##> legend('Sine', 'Cosine')
```

where you specify the names for the curves in the order they were plotted. If the legend doesn't appear in a sensible place on the graph, you can pick it up with the mouse and move it elsewhere. You should be able to get a pair of graphs looking like Figure 3.

Thus far, every time you have issued a `plot` command, the existing contents of the figure have been removed. If you want to keep the current graph, and overlay new plots on top of it, you can use the `hold` command. Using this, the two sine and cosine graphs could have been drawn using two separate `plot` commands:

```
octave:##> plot(angles,y,':')
octave:##> hold on
octave:##> plot(angles,cos(angles),'g-')
octave:##> legend('Sine', 'Cosine')
```

If you want to release the lock on the current graphs, the command is (unsurprisingly) `hold off`.

5.3 Multiple figures

You can also have multiple figure windows. If you type

```
octave:##> figure
```

the next plot command will produce a graph in a new figure window. For example,

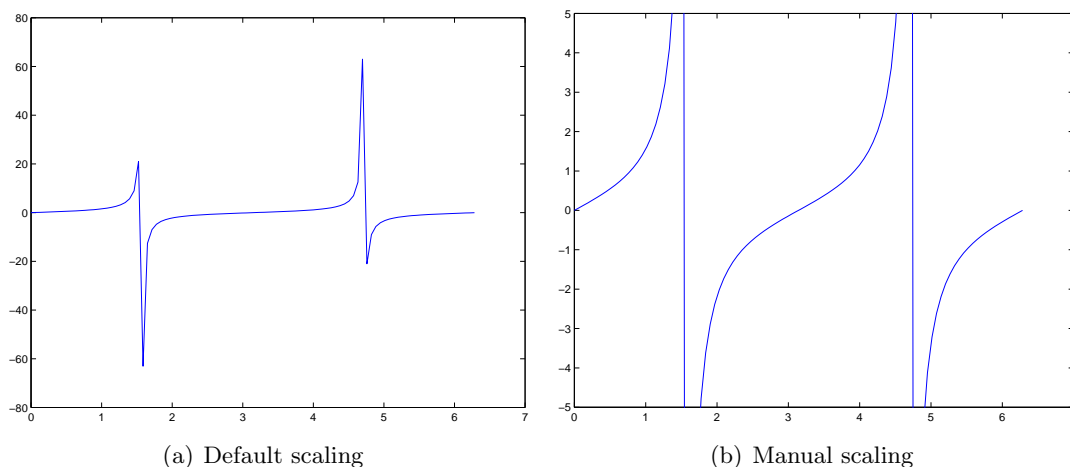


Figure 4: Graph of $y = \tan(x)$ with the default scaling, and using `axis([0 7 -5 5])`

typing

```
octave:##> plot(angles, tan(angles))
```

will plot the tangent function in this new window (see Figure 4(a)).¹⁷

If you want to go back and plot in the first figure, you can type

```
octave:##> figure(0)
```

5.4 Manual scaling

The tangent function that has just been plotted doesn't look quite right because the `angles` vector only has 100 elements, and so very few points represent the asymptotes. However, even with more values, and including $\pi/2$, it would never look quite right, since displaying infinity is difficult (OCTAVE doesn't even try).

We can hide these problems by zooming in on the part of the graph that we're really interested in. The `axis` command lets us manually select the axes. The `axis` command takes one argument which is a vector defined as $(x_{\min}, x_{\max}, y_{\min}, y_{\max})$. So if we type

```
octave:##> figure(2)
```

```
octave:##> axis([0 7 -5 5])
```

the graph is rescaled as shown in Figure 4(b). Note the two sets of brackets in the `axis` command—the normal brackets which surround the argument passes to the function, and the square brackets which define the vector, which *is* the argument.

Octave, combined with GnuPlot, allows direct interaction with the graphics window, e.g. to zoom in drag a box around the area of interest with the right mouse button pressed

¹⁷In Matlab the command `figure` brings up a new figure window immediately whereas in Octave the new window may not appear until you issue a plot command.

Mouse	Actions
MMB RMB	Annotate the graph. Mark Zoom region.
Key Bindings	Actions
a	Autoscale and Replot
b	Toggle Border
e	Replot (removes annotations)
g	Toggle grid
h	Help
l	Toggle Logscale
L	Toggle individual axis
m	Toggle Mouse control
r	Toggle Ruler
1	Decrement mousemode
2	Increment mousemode
3	Decrement clipboardmode
4	Increment clipboardmode
5	Toggle polardistance ¹⁸
6	Toggle verbose
7	Toggle graph size ratio
n	Go to next zoom in the zoom stack
p	Go to previous zoom in the zoom stack
u	Unzoom
Escape	Cancel zoom region

Table 4: Mouse and Keybinds for interaction with 2D graphs, (LMB - Left Mouse Button, RMB - Right Mouse Button), see also table 8.

and select with the left mouse button. Details of other mouse actions and key bindings (when the graphic window is selected) are shown in table 4.¹⁸

5.5 Saving and printing figures

Octave/Gnuplot do not offer a mouse/hotkey driven printing facility. However, graphs can be printed to the default printer from command line by typing `print` at the prompt. `help print` gives information about the many print options available, including

```
octave:##> print('graph1.eps', '-deps')
```

to save an encapsulated postscript version of the graph to a file graph1.eps.

```
octave:##> print('graph1.png', '-dpng')
```

to save an PNG format image

¹⁸In MATLAB access to graphics commands is via icons and menus surrounding the graph window

6 Octave programming I: Script files

If you have a series of commands that you will want to type again and again, you can store them away in a *OCTAVE script*. This is a text file which contains the commands, and is the basic form of a *OCTAVE* program. When you run a script in *OCTAVE*, it has the same effect as typing the commands from that file, line by line, into *OCTAVE*. Scripts are also useful when you're not quite sure of the series of commands you want to use, because it's easier to edit them in a text file than using the cursor keys to recall and edit previous lines that you've tried.

OCTAVE scripts are normal text files, but they must have a `.m` extension to the filename (e.g. `run.m`). For this reason, they are sometimes also called *M-files*. The rest of the filename (the word 'run' in this example) is the command that you type into *OCTAVE* to run the script.

6.1 Creating and editing a script

You can create a script file in any text editor (e.g. `emacs`, `notepad`), and you can start up a text editor from within *OCTAVE* by typing

```
octave:##> edit
```

This will start up the `emacs` editor in a new window.¹⁹ If you want to edit an existing script, you can include the name of the script. If, for example, you did have a script called `run.m`, typing `edit run`, would open the editor and load up that file for editing.

In the editor, you just type the commands that you want *OCTAVE* to run. For example, start up the editor if you haven't already and then type the following commands into the editor (but first delete any lines that `emacs` may have left)

```
% Script to calculate and plot a rectified sine wave
t = linspace(0, 10, 100);
y = abs(sin(t)); %The abs command makes all negative numbers positive
plot(t,y);
title('Rectified Sine Wave');
labelx('t');
```

The percent symbol (%) identifies a *comment*, and any text on a line after a % is ignored by *OCTAVE*. Comments should be used in your scripts to describe what it does, both for the benefit of other people looking at it, and for yourself a few weeks down the line.

Select **File** → **Save Buffer As...** from the editor's menu, and save your file as `rectsin.m`. You have now finished with the editor window, but you might as well leave it open, since you will no doubt need the editor again.

¹⁹`edit` can be configured to start up the text editor of your choice.

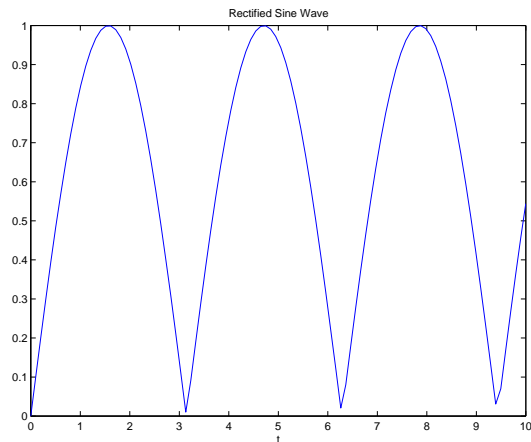


Figure 5: A rectified sine wave, produced by the `rectsin` script

6.2 Running and debugging scripts

To run the script, type the name of the script in the main OCTAVE command window. For the script you have just created, type

```
octave:##> rectsin
```

A figure window should appear showing a rectified sine wave. However, if you typed the script into the editor exactly as given above, you should also now see in the OCTAVE command window:

```
error: 'labelx' undefined near line 6 column 2
error: near line 6 of file '/mnt/hda7/Octave/rectsin.m'
```

which shows that there is an error in the script.²⁰ OCTAVE error messages make more sense if they are read from *bottom to top*. This one says that on line 6 of `rectsin.m`, it doesn't know what to do about the command `'labelx'`. The reason is, of course that the command should be `'xlabel'`. Go back to the editor, correct this line, and then *save the file again*. Don't forget to save the file each time you edit it.

Try running the corrected script by typing `rectsin` again, and this time it should correctly label the x -axis, as shown in Figure 5.

6.3 Remembering previous scripts

Scripts are very useful in OCTAVE, and if you are not careful you soon find yourself with many scripts to do different jobs and you then start to forget which is which. If you want to know what scripts you have, you can use the `what` command to give a list of all of your scripts and data files:

```
octave:##> what
m-files in current directory /mnt/hda7/Octave/tutorial
```

²⁰If you spotted the error when you were typing the script in, and corrected it, well done!


```
rectsin.m
```

```
mat-files in current directory /mnt/hda7/Octave/tutorial
```

The OCTAVE help system will also automatically recognise your scripts. If you ask for help on the `rectsin` script you should get

```
>> help rectsin
```

```
Script to calculate and plot a rectified sine wave
```

OCTAVE assumes that the first lines of comments in the M-file are a description of the script, and this is what it prints when you ask for help. You should add help lines to the top of every script you write to help you search through the scripts you write.

7 Control statements

Thus far the programs and expressions that we have seen have contained simple, sequential operations. The use of vectors (and later matrices) enable some more sophisticated computations to be performed using simple expressions, but to proceed further we need some more of the standard programming constructs. OCTAVE supports the usual loops and selection facilities.

7.1 if...else selection

In programs you quite often want to perform different commands depending on some test. The `if` command is the usual way of allowing this. The general form of the `if` statement in OCTAVE is

```
if expression
    statements
elseif expression
    statements
else
    statements
end
```

This is slightly different to the syntax in seen C++: brackets `()` are not needed around the expression (although can be used for clarity), and the block of *statements* does not need to be delimited with braces `{}`. Instead, the `end` command is used to mark the end of the `if` statement.

While control statements, such as `if`, are usually seen in OCTAVE scripts, they can also be typed in at the command line as in this example:

```
octave:##> a=0; b=2;
```

symbol	meaning	example
==	equal	if x == y
~=	not equal	if x ~= y
>	greater than	if x > y
>=	greater than or equal	if x >= y
<	less than	if x < y
<=	less than or equal	if x <= y
&	AND	if x == 1 & y > 2
	OR	if x == 1 y > 2
~	NOT	x = ~y

Table 5: Boolean expressions

```
octave:##> if a > b
            c=3
        else
            c=4
        end
c =
    4
```

If you are typing them in at the command prompt, OCTAVE waits until you have typed the final **end** before evaluating the expression.

Many control statements rely on the evaluation of a *logical expression*—some statement that can be either true or false depending on the current values. In OCTAVE, logical expressions return numbers: 0 if the expression is false and 1 if it is true:

```
octave:##> 1==2
ans =
    0

octave:##> pi > exp(1) & sqrt(-1) == i
ans =
    1
```

a complete set of relational and logical operators are available, as shown in Table 5. Note that they are not quite the same as in C++.

7.2 switch selection

If you find yourself needing multiple **if/elseif** statements to choose between a variety of different options, you may be better off with a **switch** statement. This has the following format:

```
switch x
```

```

    case x1,
        statements
    case x2,
        statements
    otherwise,
        statements
end

```

In a **switch** statement, the value of *x* is compared with each of the listed **cases**, and if it finds one which is equal then the corresponding statements are executed. Note that, unlike C++, a **break** command is not necessary—OCTAVE only executes commands until the next **case** command. If no match is found, the **otherwise** statements are executed, if present. Here is an example:

```

octave:##>    a=1;

octave:##>    switch a
    case 0
        disp('a is zero');
    case 1
        disp('a is one');
    otherwise
        disp('a is not a binary digit');
    end
a is one

```

The **disp** function displays a value or string. In this example it is used to print strings, but it can also be used with variables, e.g. **disp(a)** will print the value of *a*.

7.3 for loops

The programming construct you are likely to use the most is the **for** loop, which repeats a section of code a number of times, stepping through a set of values. In OCTAVE you should try to use vector arithmetic rather than a **for** loop, if possible, since a **for** loop is about 40 times slower.²¹ However, there are times when a **for** loop is unavoidable. The syntax is

```

for variable = vector
    statements
end

```

where *vector* contains the numbers to step through. Usually, this is expressed in the colon notation (see Section 4.2), as in this example, which creates a vector holding the first 5 terms of *n* factorial:

²¹This does not mean that **for** loops are slow, just that OCTAVE is highly optimised for matrix/vector calculations. Furthermore, many modern computers include special instructions to speed up matrix computations, since they are fundamental to the 3D graphics used in computer games. For example, the ‘MMX’ instructions added to the Intel Pentium processor in 1995, and subsequent processors, are ‘Matrix Maths eXtensions’.

```
octave:##> for n=1:5
            nf(n) = factorial(n);
        end

octave:##> disp(nf)
     1     2     6    24   120
```

Note the use of the semicolon on the end of the line in the `for` loop. This prevents OCTAVE from printing out the current value of `nf(n)` every time round the loop, which would be rather annoying (try it without the semicolon if you like).

7.4 while loops

If you don't know exactly how many repetitions you need, and just want to loop until some condition is satisfied, OCTAVE provides a `while` loop:

```
while expression
    statements
end
```

For example,

```
octave:##> x=1;

octave:##> while 1+x > 1
            x = x/2;
        end

octave:##> x
x =
    1.1102e-016
```

7.5 Accuracy and precision

The above `while` loop continues halving `x` until adding `x` to 1 makes no difference i.e. `x` is zero as far as OCTAVE is concerned, and it can be seen that this number is only around 10^{-16} . This does *not* mean that OCTAVE can't work with numbers smaller than this (the smallest number OCTAVE can represent is about 2.2251×10^{-308}).²² The problem is that the two numbers in the operation are of different orders of magnitude, and OCTAVE can't simultaneously maintain its accuracy in both.

²²The OCTAVE variables `realmax` and `realmin` tell you what the minimum and maximum numbers are on any computer (the maximum on the teaching system computers is about 1.7977×10^{308}). In addition, the variable `eps` holds the 'distance from 1.0 to the next largest floating point number'—a measure of the possible error in any number, usually represented by ϵ in theoretical calculations. On the teaching system computers, `eps` is 2.2204×10^{-16} . In our example, when `x` has this value, `1+x` does make a difference, but then this value is halved and no difference can be found.

Consider this example:

$$\begin{aligned}a &= 13901 = 1.3901 \times 10^4 \\b &= 0.0012 = 1.2 \times 10^{-3}\end{aligned}$$

If we imagine that the numerical accuracy of the computer is 5 significant figures in the mantissa (the part before the $\times 10^k$), then both a and b can be exactly represented. However, if we try to add the two numbers, we get the following:

$$\begin{aligned}a + b &= 13901.0012 = 1.39010012 \times 10^4 \\&= 1.3901 \times 10^4 \quad \text{to 5 significant figures}\end{aligned}$$

So, while the two numbers are fine by themselves, because they are of such different magnitudes their sum cannot be exactly represented.

This is exactly what is happening in the case of our `while` loop above. OCTAVE (and most computers) are accurate to about fifteen significant figures, so once we try to add 1×10^{-16} to 1, the answer requires a larger number of significant figures than are available, and the answer is truncated, leaving just 1.

There is no general solution to these kind of problems, but you need to be aware that they exist. It is most unusual to need to be concerned about the sixteenth decimal place of an answer, but if you are then you will need to think very carefully about how you go about solving the problem. The answer is to *think* about how you go about formulating your solution, and make sure that, in the solution you select, the numbers you are dealing with are all of about the same magnitude.

8 Octave programming II: Functions

Scripts in OCTAVE let you write simple programs, but more powerful than scripts are user-defined *functions*. These let you define your own OCTAVE commands which you can then use either from the command line, or in other functions or scripts.

In OCTAVE functions, variables are always passed by *value*, never by reference. OCTAVE functions can, however, return more than one value.²³ Basically, functions in OCTAVE are passed numbers, perform some calculations, and give you back some other numbers.

A function is defined in a text file, just like a script, except that the first line of the file has the following form:

```
function [output1,output2,...] = name(input1,input2,...)
```

Each function is stored in a different M-file, which *must have the same name as the function*. For example, a function called `sind()` must be defined in a file called `sind.m`. Each function can accept a range of arguments, and return a number of different values.

Whenever you find yourself using the same set of expressions again and again, this is a sign that they should be wrapped up in a function. When wrapped up as a function, they are easier to use, make the code more readable, and can be used by other people in other situations.

²³When variables are passed by value, they are *read only*—the values can be read and used, but cannot be altered. When variables are passed by reference (in C++), their values can be altered to pass information back from the function. This is required in C++ because usually only one value can be returned from a function; in OCTAVE many values can be returned, so passing by reference is not required.

8.1 Example 1: Sine in degrees

OCTAVE uses radians for all of its angle calculations, but most of us are happier working in degrees. When doing OCTAVE calculations you could just always convert your angle `d` to radians using `sin(d/180*pi)`, or even using the variable `deg` as defined in Section 3.1, writing `sin(d*deg)`. But it would be simpler and more readable if you could just type `sind(d)` (‘sine in degrees’), and we can create a function to do this. Such a function would be defined by creating a file `sind.m` containing just the following lines:

```
function s = sind(x)
%SIND(X)   Calculates sine(x) in degrees
s = sin(x*pi/180);
```

This may seem trivial, but many functions *are* trivial and it doesn’t make them any less useful. We’ll look at this function line-by-line:

Line 1 Tells OCTAVE that this file defines a function rather than a script. It says that the function is called `sind`, and that it takes one argument, called `x`. The result of the function is to be known, internally, as `s`. Whatever `s` is set to in this function is what the user will get when they use the `sind` function.

Line 2 Is a comment line. As with scripts, the first set of comments in the file should describe the function. This line is the one printed when the user types `help sind`. It is usual to use a similar format to that which is used by OCTAVE’s built-in functions.

Line 3 Does the actual work in this function. It takes the input `x` and saves the result of the calculation in `s`, which was defined in the first line as the name of the result of the function.

End of the function Functions in OCTAVE do not need to end with `return` (although you can use the `return` command to make OCTAVE jump out of a function in the middle). Because each function is in a separate M-file, once it reaches the end of the file, OCTAVE knows that it is the end of the function. The value that `s` has at the end of this function is the value that is returned.

8.2 Creating and using functions

Create the above function by opening the editor (type `edit` if it’s not still open) and typing the lines of the function as given above. Save the file as `sind.m`, since the text file must have the same name as the function, with the addition of `.m`.

You can now use the function in the same way as any of the built-in OCTAVE functions. Try typing

```
octave:##> help sind
```

```
SIND(X)   Calculates sine(x) in degrees
```

which shows that the function has been recognised by OCTAVE and it has found the help line included in the function definition. Now we can try some numbers

```

octave:##>   sind(0)
ans =
    0

octave:##>   sind(45)
ans =
    0.7071

octave:##>   t = sind([30 60 90])
t =
    0.5000    0.8660    1.0000

```

This last example shows that it also automatically works with vectors. If you call the `sind` function with a vector, it means that the `x` parameter inside the function will be a vector, and in this case the `sin` function knows how to work with vectors, so can give the correct response.

8.3 Example 2: Unit step

Here is a more sophisticated function which generates a unit step, defined as

$$y = \begin{cases} 0 & \text{if } t < t_0 \\ 1 & \text{otherwise} \end{cases}$$

This function will take two parameters: the times for which values are to be generated, and `t0`, the time of the step. The complete function is given below:

```

function y = ustep(t, t0)
%USTEP(t, t0) unit step at t0
%   A unit step is defined as
%       0 for t < t0
%       1 for t >= t0
[m,n] = size(t);
% Check that this is a vector, not a matrix i.e. (1 x n) or (m x 1)
if m ~= 1 & n ~= 1
    error('T must be a vector');
end
y = zeros(m, n); %Initialise output array
for k = 1:length(t)
    if t(k) >= t0
        y(k) = 1; %Otherwise, leave it at zero, which is correct
    end
end

```

Again, we shall look at this function definition line-by-line:

Line 1 The first line says that this is a function called `ustep`, and that the user must supply two arguments, known internally as `t` and `t0`. The result of the function is one variable, called `y`.

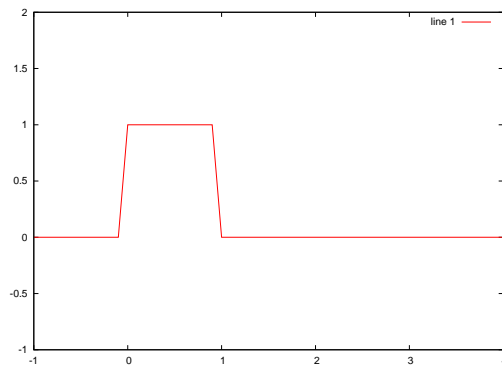


Figure 6: A unit, one second pulse created from two unit steps, using the function `ustep`.

Lines 2-5 Are the description of the function. This time the help message contains several lines.

Line 6 The first argument to the `ystep` function, `t`, will usually be a vector, rather than a scalar, which contains the time values for which the function should be evaluated. This line uses the `size` function, which returns *two* values: the number of rows and then the number of columns of the vector (or matrix). This gives an example of how functions in OCTAVE can return two things—in a vector, of course. These values are used to create an output vector of the same size, and to check that the input *is* a vector.

Lines 7-10 Check that the input `t` is valid i.e. that it is not a matrix. This checks that it has either one row or one column (using the result of the `size` function). The `error` function prints out a message and aborts the function if there is a problem.

Line 11 As the associated comment says, this line creates the array to hold the output values. It is initialised to be the same size and shape as the input `t`, and for each element to be zero.

Line 12 For each time value in `t`, we want to create a value for `y`. We therefore use a `for` loop to step through each value. The `length` function tells us how many elements there are in the vector `t`.

Lines 13-15 According to our definition, if $t < t_0$, then the step function has the value zero. Our output vector `y` already contains zeros, so we can ignore this case. In the other case, when $t \geq t_0$, the output should be 1. We test for this case and set `y`, our output variable, accordingly.

As in most high level languages, all variables created inside a function (`m`, `n` and `k` in this case) are *local* to the function. They only exist for the duration of the function, and do not overwrite any variables of the same name elsewhere in OCTAVE. The only variables which are passed back are the return values defined in the first `function` line: `y` in this case.

Type this function into the editor, and save it as `ustep.m`. We can now use this function to create signal. For example, to create a unit pulse of duration one second, starting at $t = 0$, we can first define a time scale:


```
octave:##> t=-1:0.1:4;
```

and then use `ustep` twice to create the pulse:

```
octave:##> v = ustep(t, 0) - ustep(t, 1);
```

```
octave:##> plot(t, v)
```

```
octave:##> axis([-1 4 -1 2])
```

This should display the pulse, as shown in Figure 6. If we then type

```
octave:##> who
```

```
*** dynamically linked functions:
```

```
dispatch
```

```
*** currently compiled functions:
```

```
__plt2__ __plt__ isscalar isvector rows
```

```
__plt2vv__ axis isstr plot ustep
```

```
*** local user variables:
```

```
t v
```

we can confirm that the variables `m` and `n` defined in the `ustep` function only lasted as long as the function did, and are not part of the main workspace. Any other variable listed e.g. `y` variable will still have the value from an earlier definition, e.g. `y` by the `rectsin` script, rather than the values defined for the variable `y` in the `ustep` function. Variables defined and used inside functions are completely separate from the main workspace.

9 Matrices and vectors

Vectors are special cases of *matrices*. A matrix is a rectangular array of numbers, the size of which is usually described as $m \times n$, meaning that it has m rows and n columns. For example, here is a 2×3 matrix:

$$A = \begin{bmatrix} 5 & 7 & 9 \\ -1 & 3 & -2 \end{bmatrix}$$

To enter this matrix into OCTAVE you use the same syntax as for vectors, typing it in row by row:

```
octave:##> A=[5 7 9  
             -1 3 -2]
```

```
A =
      5      7      9
     -1      3     -2
```

alternatively, you can use semicolons to mark the end of rows, as in this example:

```
octave:##> B=[2 0; 0 -1; 1 0]
B =
      2      0
      0     -1
      1      0
```

You can also use the colon notation, as before:

```
octave:##> C = [1:3; 8:-2:4]
C =
      1      2      3
      8      6      4
```

A final alternative is to build up the matrix row-by-row (this is particularly good for building up tables of results in a `for` loop):

```
octave:##> D=[1 2 3];
octave:##> D=[D; 4 5 6];
octave:##> D=[D; 7 8 9]
D =
      1      2      3
      4      5      6
      7      8      9
```

9.1 Matrix multiplication

With vectors and matrices, the `*` symbol represents matrix multiplication, as in these examples (using the matrices defined above)

```
octave:##> A*B
ans =
     19     -7
     -4     -3
```

```
octave:##> B*C
ans =
      2      4      6
     -8     -6     -4
      1      2      3
```

```
octave:##> A*C
error: operator *: nonconformant arguments (op1 is 2x3, op2 is 2x3)
error: evaluating binary operator '*' near line 11, column 2
```

You might like to work out these examples by hand to remind yourself what is going on. Note that you cannot do $\mathbf{A}*\mathbf{C}$, because the two matrices are incompatible shapes.²⁴

When just dealing with vectors, there is little need to distinguish between row and column vectors. When multiplying vectors, however, it will only work one way round. A row vector is a $1 \times n$ matrix, but this can't post-multiply a $m \times n$ matrix:

```
octave:##> x=[1 0 3]
x =
    1    0    3

octave:##> A*x
error: operator *: nonconformant arguments (op1 is 2x3, op2 is 1x3)
error: evaluating binary operator '*' near line 12, column 2
```

9.2 The transpose operator

Transposing a vector changes it from a row to a column vector and vice versa. The transpose of a matrix interchanges the rows with the columns. Mathematically, the transpose of \mathbf{A} is represented as \mathbf{A}^T . In OCTAVE an apostrophe performs this operation:

```
octave:##> A
A =
    5    7    9
   -1    3   -2

octave:##> A'
ans =
    5   -1
    7    3
    9   -2

octave:##> A*x'
ans =
    32
   -7
```

In this last example the `x'` command changes our row vector into a column vector, so it can now be pre-multiplied by the matrix \mathbf{A} .

9.3 Matrix creation functions

OCTAVE provides some functions to help you build special matrices. We have already met `ones` and `zeros`, which create matrices of a given size filled with 1 or 0.

²⁴In general, in matrix multiplication, the matrix sizes are

$$(l \times m) * (m \times n) \rightarrow (l \times n)$$

When we try to do $\mathbf{A}*\mathbf{C}$, we are attempting to do $(2 \times 3) * (2 \times 3)$, which does not agree with the definition above. The middle pair of numbers are not the same, which explains the wording of the error message.

A very important matrix is the *identity matrix*. This is the matrix that, when multiplying any other matrix or vector, does not change anything. This is usually called **I** in formulæ, so the OCTAVE function is called **eye**. This only takes one parameter, since an identity matrix must be square:

```
octave:##> I = eye(4)
```

```
I =
```

```

    1    0    0    0
    0    1    0    0
    0    0    1    0
    0    0    0    1
```

we can check that this leaves any vector or matrix unchanged:

```
octave:##> I * [5; 8; 2; 0]
```

```
ans =
```

```

    5
    8
    2
    0
```

The identity matrix is a special case of a *diagonal matrix*, which is zero apart from the diagonal entries:

$$D = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 7 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

You could construct this explicitly, but the OCTAVE provides the **diag** function which takes a vector and puts it along the diagonal of a matrix:

```
octave:##> diag([-1 7 2])
```

```
ans =
```

```

   -1    0    0
    0    7    0
    0    0    2
```

The **diag** function is quite sophisticated, since if the function is called for a matrix, rather than a vector, it tells you the diagonal elements of that matrix. For the matrix **A** defined earlier:

```
octave:##> diag(A)
```

```
ans =
```

```

    5
    3
```

Notice that the matrix does not have to be square for the diagonal elements to be defined, and for non-square matrices it still begins at the top left corner, stopping when it runs out of rows or columns.

Finally, it is sometimes useful to create an empty matrix, perhaps for adding elements to later. You can do this by defining the matrix with an empty pair of braces:

```
octave:##> E = []
```

```
E =
```

```

[]
```

9.4 Building composite matrices

It is often useful to be able to build matrices from smaller components, and this can easily be done using the basic matrix creation syntax:

```
octave:##> comp = [eye(3) B;
                   A    zeros(2,2)]
comp =
    1    0    0    2    0
    0    1    0    0   -1
    0    0    1    1    0
    5    7    9    0    0
   -1    3   -2    0    0
```

You just have to be careful that each sub-matrix is the right size and shape, so that the final composite matrix is rectangular. Of course, OCTAVE will tell you if any of them have the wrong number of row or columns.

9.5 Matrices as tables

Matrices can also be used to simply tabulate data, and can provide a more natural way of storing data:

```
octave:##> t=0:0.2:1;
octave:##> freq=[sin(t)' sin(2*t)', sin(3*t)']
freq =
    0    0    0
  0.1987  0.3894  0.5646
  0.3894  0.7174  0.9320
  0.5646  0.9320  0.9738
  0.7174  0.9996  0.6755
  0.8415  0.9093  0.1411
```

Here the n th column of the matrix contains the (sampled) data for $\sin(nt)$. The alternative would be to store each series in its own vector, each with a different name. You would then need to know what the name of each vector was if you wanted to go on and use the data. Storing it in a matrix makes the data easier to access.

9.6 Extracting bits of matrices

Numbers may be extracted from a matrix using the same syntax as for vectors, using the `()` brackets. For a matrix, you specify the row co-ordinate first and then the column co-ordinate (note that, in Cartesian terms, this is y and then x). Here are some examples:

```
octave:##> J = [
    1    2    3    4
    5    6    7    8
   11   13   18   10];
octave:##> J(1,1)
ans =
```

```

      1
octave:##> J(2,3)
ans =
      7
octave:##> J(1:2, 4)    %Rows 1-2, column 4
ans =
      4
      8
octave:##> J(3,:)      %Row 3, all columns
ans =
    11     13     18     10

```

The `:` operator can be used to specify a range of elements, or if used just by itself then it refers to the entire row or column.

These forms of expressions can also be used on the left-hand side of an expression to write elements into a matrix:

```

octave:##> J(3, 2:3) = [-1 0]
J =
      1      2      3      4
      5      6      7      8
     11     -1      0     10

```

10 Basic matrix functions

OCTAVE allows all of the usual arithmetic to be performed on matrices. Matrix multiplication has already been discussed, and the other common operation is to add or subtract two matrices of the same size and shape. This can easily be performed using the `+` and `-` operators. As with vectors, OCTAVE also defines the `.*` and `./` operators, which allow the corresponding elements of two matching matrices to be multiplied or divided. All the elements of a matrix may be raised to the same power using the `.^` operator.

Unsurprisingly, OCTAVE also provides a large number of functions for dealing with matrices, and some of these will be covered later in this tutorial. Try typing

```
help -i
```

and then search the various options. NB the table loaded is searchable by putting the cursor over a topic and pressing `<RETURN>`, press `Q` to exit the help. See also Table 6. For the moment we'll consider some of the fundamental matrix functions.²⁵

The `size` function will tell you the dimensions of a matrix. This is a function which returns a vector, specifying the number of rows, and then columns:

```

octave:##> size(J)
ans =
      3      4

```

The inverse of a matrix is the matrix which, when multiplied by the original matrix, gives the identity ($\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$). It ‘undoes’ the effect of the original matrix. It is only defined for square matrices, and in OCTAVE can be found using the `inv` function:

²⁵MATLAB has a slightly different index system, to access just type `help` at the `>>` prompt

eye	Create an identity matrix
zeros	Create a matrix of zeros
ones	Create a matrix of ones
rand	Create a matrix filled with random numbers
diag	Create a diagonal matrix, or extract the diagonal of the given matrix
inv	Inverse of a matrix
det	Determinant of a matrix
trace	Trace of a matrix
eig	Calculate the eigenvectors and eigenvalues of a matrix
rank	Calculate an estimate of the rank of a matrix
null	Calculate a basis for the null space of a matrix
rref	Perform Gaussian elimination on an augmented matrix
lu	Calculate the LU decomposition of a matrix
qr	Calculate the QR decomposition of a matrix
svd	Calculate the SVD of a matrix
pinv	Calculate the pseudoinverse of a matrix

Table 6: Basic matrix functions and decompositions

```
octave:##> A = [
    3    0    4
    0    1   -1
    2    1   -3];
octave:##> inv(A)
ans =
    0.1429   -0.2857    0.2857
    0.1429    1.2143   -0.2143
    0.1429    0.2143   -0.2143
octave:##> A*inv(A) %Check the answer
ans =
    1.0000    0.0000   -0.0000
         0    1.0000         0
         0    0.0000    1.0000
```

Again, note the few numerical errors which have crept in, which stops OCTAVE from recognising some of the elements as exactly one or zero.

The *determinant* of a matrix is a very useful quantity to calculate. In particular, a zero determinant implies that a matrix does not have an inverse. The **det** function calculates the determinant:

```
octave:##> det(A)
ans =
   -14
```

11 Solving $\mathbf{Ax} = \mathbf{b}$

One of the most important use of matrices is for representing and solving simultaneous equations. A system of linear equations is

$$\begin{aligned}a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\&\vdots = \vdots \\a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m\end{aligned}$$

where the a_{ij} and b_i are known, and we are looking for a set of values x_i that simultaneously satisfy all the equations. This can be written in matrix-vector form as

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

or

$$\mathbf{Ax} = \mathbf{b}$$

In this representation, \mathbf{A} is the matrix of coefficients, \mathbf{b} are the constants, and \mathbf{x} is the vector of parameters that we want to find.

Since OCTAVE is designed to process matrices and vectors, it is particularly appropriate to use it to solve these forms of problems.

11.1 Solution when \mathbf{A} is invertible

The simplest set of linear equations to solve occur when you have both n equations and n unknowns. In this case the matrix \mathbf{A} will be square and can often be inverted. Consider this simple example:

$$\begin{aligned}x + y &= 3 \\2x - 3y &= 5\end{aligned}$$

Solving this in OCTAVE is a case of turning the equations into matrix-vector form and then using the inverse of \mathbf{A} to find the solution:

```
octave:##>A=[1 1
> 2 -3];
octave:##> b=[3 5]';
octave:##> inv(A)*b
ans =
    2.8000
    0.2000
octave:##>> A*ans    %Just to check
ans =
    3.0000
    5.0000
```

So the solution is $x = 2.8$, $y = 0.2$.

11.2 Gaussian elimination and LU factorisation

Calculating the inverse of a matrix is an inefficient method of solving these problems, even if OCTAVE can still invert large matrices a lot more quickly than you could do by hand. From the IB Linear Algebra course, you should be familiar with Gaussian elimination, and LU factorisation (which is just Gaussian elimination in matrix form).²⁶ These provide a more efficient way of solving $\mathbf{Ax} = \mathbf{b}$, and OCTAVE makes it very easy to use Gaussian elimination by defining this as the meaning of *matrix division* for invertible matrices.

11.3 Matrix division and the slash operator

In a normal algebraic equation, $ax = b$, if you want to find the value of x you would just calculate $x = \frac{b}{a}$. In a matrix-vector equation $\mathbf{Ax} = \mathbf{b}$, however, division is not defined and the solution is given by $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. As a shortcut for this, OCTAVE defines a special operator `\` (note that this is a backslash, not the divide symbol), and can be thought of as ‘matrix division’. Using this, the solution to our earlier equation may be calculated as:

```
octave:##>A\b ans =
2.8000 0.2000
```

Note that this is *not a standard notation* and in written mathematical expressions you should still always write $\mathbf{A}^{-1}\mathbf{b}$.

You should recall that matrix multiplication is not commutative i.e. $\mathbf{AB} \neq \mathbf{BA}$. This means that, while the solution to $\mathbf{AX} = \mathbf{B}$ is given by $\mathbf{X} = \mathbf{A}^{-1}\mathbf{B}$, the solution to $\mathbf{XA} = \mathbf{B}$ is $\mathbf{X} = \mathbf{BA}^{-1}$, and these two expressions are different. As a result, OCTAVE also defines the `/` (forward slash) operator which performs this other variety of matrix division. The former case, however, is more likely, and it is usually the backslash that you will need to use. The two slash operators are summarised in Table 7.

operator	to solve	OCTAVE command	mathematical equivalent	name
<code>\</code>	$\mathbf{AX} = \mathbf{B}$	<code>A\B</code>	$\mathbf{A}^{-1}\mathbf{B}$	left division (backslash)
<code>/</code>	$\mathbf{XA} = \mathbf{B}$	<code>B/A</code>	\mathbf{BA}^{-1}	right division (forward slash)

Table 7: Summary of OCTAVE’s slash operators. These use Gaussian elimination if the matrix is invertible, and finds the least squares solution otherwise.

11.4 Singular matrices and rank

The slash operator tries to do its best whatever matrix and vector it is given, even if a solution is not possible, or is undetermined. Consider this example:

$$\begin{aligned}u + v + w &= 2 \\2u + 3w &= 5 \\3u + v + 4w &= 6\end{aligned}$$

²⁶The `rref` function will perform Gaussian elimination if you give it an augmented matrix `[A b]`, or the `lu` function in OCTAVE will perform the LU factorisation of a matrix (see the help system for more information).

This is again an equation of the form $\mathbf{Ax} = \mathbf{b}$. If you try to solve this in OCTAVE using the slash operator, you get

```
octave:##>A=[1 1 1
> 2 0 3
> 3 1 4];
octave:##>b=[ 2 5 6]';
octave:##>x=a\b;
warning: matrix singular to machine precision, rcond = 1.15648e-17
warning: attempting to find minimum norm solution
x =

0.69048
-0.11905
1.09524

octave:##>a*x ans =

1.6667
4.6667
6.3333
```

So the solution given is not a solution to the equation! Y In this case the matrix \mathbf{A} is *singular* as OCTAVE had warned. This means that it has a zero determinant, and so is non-invertible. (Even if there had not been a warning you should perhaps have been suspicious, anyway, from the size of the numbers in \mathbf{x} . This is also seen when checking the determinant which gives a near-zero number, e.g.

```
octave:##>det(A)
ans = 5.5511e-16
```

When the matrix is singular it means that there is not a unique solution to the equations. But we have three equations and three unknowns, so why is there not a unique solution? The problem is that there are *not* three unique equations. The `rank` function in OCTAVE estimates the rank of a matrix—the number linearly of independent rows or columns in the matrix:

```
octave:##> rank(A)
ans =
    2
```

So there are only two independent equations in this case. Looking at the original equations more carefully, we can see that the first two add to give $3u + v + 4w = 7$, which is clearly inconsistent with the third equation. In other words, there is no solution to this equation. What has happened in this case is that, thanks to rounding errors in the Gaussian elimination, OCTAVE has found an incorrect solution.

The moral of this section is that the slash operator in OCTAVE is very powerful and useful, but you should not use it indiscriminately. You should check your matrices to ensure that they are not singular, or near-singular.²⁷

11.5 Ill-conditioning

Matrices which are near-singular are an example of *ill-conditioned* matrices. A problem is ill-conditioned if small changes in the data produce large changes in the results. Consider this system:

$$\begin{bmatrix} 1 & 1 \\ 1 & 1.01 \end{bmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 2.01 \end{pmatrix}$$

for which OCTAVE displays the correct answer:

```
octave:##>M=[1 1; 1 1.01]; b=[2; 2.01];
octave:##>x=M\b
x =

1.00000
1.00000
```

Let us now change one of the elements of M by a small amount, and see what happens:

```
octave:##>M(1,2)=1.005;
octave:##>x=M\b
x =

-0.0100000
2.0000000
```

Changing this one element by 0.5% has decreased X(1) by approximately 101%, and increased X(2) by 100%! The sensitivity of a matrix is estimated using the *condition number*, the precise definition of which is beyond the scope of this tutorial. However, the larger the condition number, the more sensitive the solution is to numerical errors. In OCTAVE, the condition number can be found by the `cond` function:

```
octave:##> cond(M)

ans = 402.01
```

A rule of thumb is that if you write the condition number in exponential notation $a \times 10^k$, then you last k significant figures of the result should be ignored. OCTAVE works to about fifteen significant figures, so the number of significant figures that you should believe is $(15 - k)$. In this example, the condition number is 4×10^2 , so all of the the last 2 decimal places of the solution should be dropped i.e. the result is perfectly valid.²⁸ For the singular

²⁷Some singular matrices can have an infinity of solutions, rather than no solution. In this case, the slash operator gives just one of the possible values. This again is something that you need to be aware of, and watch out for.

²⁸This assumes that the values entered into the original matrix **M** and vector **b** were accurate to fifteen significant figures. If those values were known to a lesser accuracy, then you lose k significant figures from *that* accuracy.

matrix \mathbf{A} from earlier, which gave the spurious result, `cond` gives a condition number of 2.176×10^{16} . In other words, all of the result should be ignored!²⁹

11.6 Over-determined systems: Least squares

A common experimental scenario is when you wish to fit a model to a large number of data points. Each data point can be expressed in terms of the model's equation, giving a set of simultaneous equations which are *over-determined* i.e. the number of independent equations, m , is larger than the number of variables, n . In these cases it is not possible to find an *exact* solution, but the desired solution is the 'best fit'. This is usually defined as the model parameters which minimise the squared error to each data point.

For a set of equations which can be written as $\mathbf{Ax} = \mathbf{b}$, the least-squares solution is given by the *pseudoinverse* (see your IB Linear Algebra notes):

$$\mathbf{x} = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{b}$$

In OCTAVE you can again use the `\` operator. For invertible matrices it is defined to solve the equation using Gaussian elimination, but for over-determined systems it finds the least squares solution. The pseudoinverse can also be calculated in OCTAVE using the `pinv` function, as demonstrated in the following example.

11.7 Example: Triangulation

A hand-held radio transmitter is detected from three different base stations, each of which provide a reading of the direction to the transmitter. The vector from each base station is plotted on a map, and they should all meet at the location of the transmitter. The three lines on the map can be described by the equations

$$\begin{aligned} 2x - y &= 2 \\ x + y &= 5 \\ 6x - y &= -5 \end{aligned}$$

and these are plotted in Figure 7. However, because of various errors, there is not one common point at which they meet.

The least squares solution to the intersection point can be calculated in OCTAVE in a number of different ways once we have defined our matrix equation:

```
octave:##>A=[2 -1; 1 1; 6 -1];
octave:##>b = [2 5 -5]';
octave:##>x = inv(A'*A)*A'*b
x =

-0.094595
2.445946
octave:##>x=pinv(A)*b
x =
```

²⁹The condition number for this singular matrix again shows one of the problems of numerical computations. All singular matrices should have a condition number of infinity.

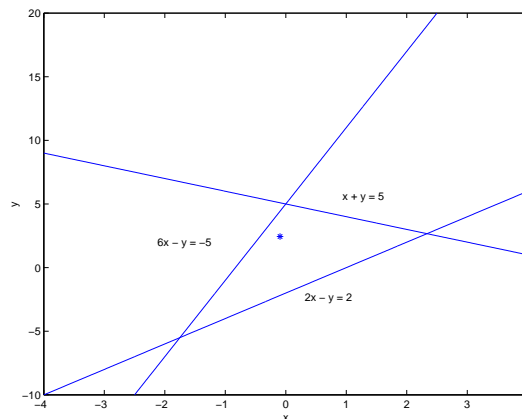


Figure 7: Three lines and the least squares estimate of the common intersection point

```
-0.094595
2.445946
octave:##>x = A\b
x =

-0.094595
2.445946
```

All of these, of course, give the same solution. This least squares solution is marked on Figure 7 with a ‘*’.

12 More graphs

OCTAVE provides more sophisticated graphing capabilities than simply plotting 2D Cartesian graphs, again by using GNUPlot. It can also produce histograms, 3D surfaces, contour plots and polar plots, to name a few. Details of all of these can be found in the `help` system, additional information can be found in the GNUPlot manual.

12.1 Putting several graphs in one window

If you have several graphs sharing a similar theme, you can plot them on separate graphs within the same graphics window. The `subplot` command splits the graphics window into an array of smaller windows. The general format is

```
subplot(rows,columns,select)
```

The `select` argument specifies the current graph in the array. These are numbered from the top left, working along the rows first. The example below creates two graphs, one on top of the other, as shown in Figure 8.

```
octave:##> x = linspace(-10, 10);
octave:##> subplot(2,1,1) % Specify two rows, one column, and select
```

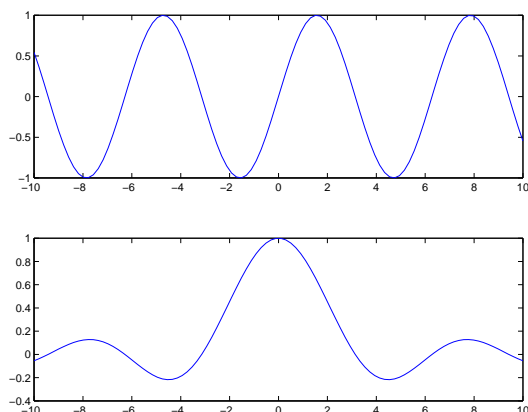


Figure 8: Example of the `subplot` command, showing a sine and sinc curve one above the other.

```
octave:##>                                % the top one as the current graph
octave:##> plot(x, sin(x));
octave:##> subplot(2,1,2);
octave:##> plot(x, sin(x)./x);
```

The standard axes labelling and title commands can also be used in each `subplot`.

12.2 3D plots

OCTAVE provides a wide range of methods for visualising 3D data. The simplest form of 3D graph is `plot3`, which is the 3D equivalent of the `plot` command. Used for plotting points or drawing lines, this simply takes a list of x , y and z values. The following example plots a helix, using a parametric representation.³⁰

```
octave:##> t = 0:pi/50:10*pi;
octave:##> x = sin(t); y = cos(t); z = t;
octave:##> plot3(x, y, z);
```

The graph is shown in Figure 9. The `xlabel` and `ylabel` commands work as before, and now the `zlabel` command can also be used. When plotting lines or points in 3D you can use all of the styles listed in Table 3.

12.3 Changing the viewpoint

If you want to rotate the 3D plot to view it from another angle, the easiest method is to use interact using the mouse. (N.B. Clicking and dragging with the mouse on the plot will now rotate and zoom the graph etc. It helps if the graph has been labelled first) Details of the most used commands are given in Table 8³¹

The OCTAVE command `view` can also be used to select/set a particular viewpoint, (use `help view` to obtain more information).

³⁰A line is only a one dimensional shape—you can define a point along its length, but it has no width or height. Lines can therefore be described in terms of only one parameter: the distance along the line. The x , y (and in 3D z) co-ordinates of points on the line can all be written as functions of this distance along the line: ‘if I am this far along the line, I must be at this point.’

³¹Matlab has an enhanced graphical interaction using a GUI

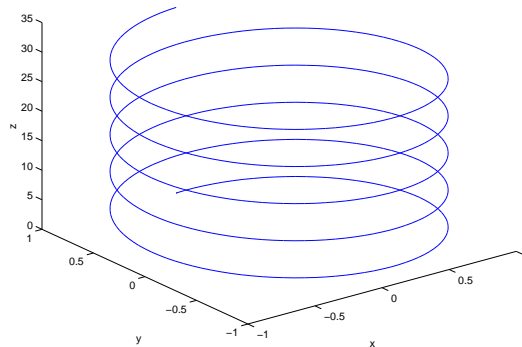


Figure 9: Example of the `plot3` command, showing a helix.

Mouse	Actions
LMB + motion <Ctrl> + LMB + motion MMB + motion <Ctrl> + MMB + motion <Shift> + MMB + motion	Rotate View Rotate axes (View update when button released) Scale View. Scale axes (View update when button released) vertical motion – change ticslevel
Key Bindings	Actions
→ ↑ ← ↓	Rotate right (<shift> increases amount) Rotate up (<shift> increases amount) Rotate left (< shift> increases amount) Rotate down (<shift> increases amount)

Table 8: Mouse and Keybinds for interaction with 3D graphs, (LMB - Left Mouse Button, MMB - Middle Mouse Button), see also table 4.

12.4 Plotting surfaces

Another common 3D graphing requirement is to plot a surface, defined as a function of two variables $f(x, y)$, and these can be plotted in a number of different ways in OCTAVE. First, though, in order to plot these functions we need a grid of points at which the function is to be evaluated. Such a grid can be created using the `meshgrid` function:

```
octave:##> x = 2:0.2:4; % Define the x- and y- coordinates
octave:##> y = 1:0.2:3; % of the grid lines
octave:##> [X,Y] = meshgrid(x, y); %Make the grid
```

The matrices `X` and `Y` then contain the x and y coordinates of the sample points. The function can then be evaluated at each of these points. For example, to plot

$$f(x, y) = (x - 3)^2 - (y - 2)^2$$

over the grid calculated earlier, you would type

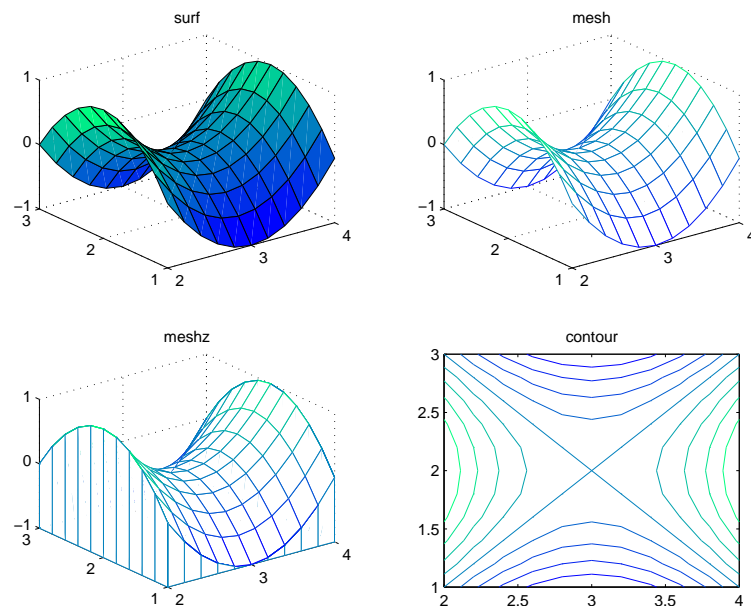


Figure 10: Examples of the same saddle-shaped surface visualised using different OCTAVE commands. (The 4 graphs are plotted in the same figure using the command `subplot(2,2,i)`.)

```
octave:##> Z=(X-3).^2 - (Y-2).^2;
octave:##> surf(X,Y,Z)
```

`surf` is only one of the possible ways of visualising a surface. Figure 10 shows, in the top left, this surface plotted using the `surf` command, and also the results of some of the other possible 3D plotting commands.

12.5 Images and Movies

An image is simply a matrix of numbers, where each number represents the colour or intensity of that pixel in the image. It should be no surprise, therefore, that OCTAVE can also perform image processing. Suppose a matrix of the brightness values of an image has been stored in a file called `cued.mat`, the following commands load and display it:

```
octave:##> load cued
octave:##> colormap(gray(64)) % Tell Octave to expect a greyscale image
octave:##> image(a)
```

The `image` command displays this matrix as an image; the `colormap` command tells OCTAVE to interpret each number as a shade of grey, and what the range of the numbers is.³² N.B. Octave uses Imagemagick, by default, as the display program giving the user full access to file format conversion and printing.

OCTAVE can also create and display movies, but unlike MATLAB this is awkward. It is hoped to enhance the functionality in this area shortly.

³²Try `colormap(jet)` for an example of an alternative interpretation of the numbers (this is the default colormap). The colormap is also used to determine the colours for the height in the 3D plotting commands.

13 Eigenvectors and the Singular Value Decomposition

After the solution of $\mathbf{Ax} = \mathbf{b}$, the other important matrix equation is $\mathbf{Ax} = \lambda\mathbf{x}$, the solutions to which are the *eigenvectors* and *eigenvalues* of the matrix \mathbf{A} . Equations of this form crop up often in engineering applications, and OCTAVE again provides the tools to solve them.

13.1 The eig function

The `eig` function in OCTAVE calculates eigenvectors and eigenvalues. If used by itself, it just provides a vector containing the eigenvalues, as in this example:

```
octave:##>A=[1 3 -2
> 3 5 1
> -2 1 4];
octave:##>eig(A)
ans =

-1.5120
4.9045
6.6076
```

In order to obtain the eigenvectors, you need to provide two variables for the answer:

```
octave:##>[V,D]=eig(A)
V =

-0.818394 -0.315302 -0.480434
0.434733 0.207055 -0.876433
-0.375818 0.926128 0.032380

D =

-1.51204 0.00000 0.00000
0.00000 4.90448 0.00000
0.00000 0.00000 6.60756
```

The columns of the matrix \mathbf{V} are the eigenvectors, and the eigenvalues are this time returned in a diagonal matrix. The eigenvectors and eigenvalues are returned in this format because this is then consistent with the *diagonal form* of the matrix. You should recall that, if \mathbf{U} is the matrix of eigenvectors and $\mathbf{\Lambda}$ the diagonal matrix of eigenvalues, then $\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{-1}$. We can easily check this using the matrices returned by `eig`:

```
octave:##> V*D*inv(V)
ans =

1.0000    3.0000   -2.0000
3.0000    5.0000    1.0000
-2.0000    1.0000    4.0000
```

which is the original matrix \mathbf{A} .

13.2 The Singular Value Decomposition

Eigenvectors and eigenvalues can only be found for a square matrix, and thus the decomposition into $U\Lambda U^{-1}$ is only possible for these matrices. The Singular Value Decomposition (SVD) is a very useful technique, which provides a similar decomposition for any matrix. The specifics are covered in the IB Linear Algebra course, but an outline of the outcome of the decomposition is given here.

The SVD takes an $m \times n$ matrix A and factors it into $A = Q_1 \Sigma Q_2^T$, where

- Q_1 is an $m \times m$ orthogonal matrix whose columns are the eigenvectors of AA^T
- Q_2 is an $n \times n$ orthogonal matrix whose columns are the eigenvectors of $A^T A$
- Σ is an $m \times n$ diagonal matrix whose elements are the square roots of the eigenvalues of AA^T and $A^T A$ (both have the same eigenvalues, but different eigenvectors)

This is better shown graphically:

$$\begin{bmatrix} & & & & \\ & & & & \\ & & & & \\ & & & & \\ \cdot & \cdot & n & \cdot & \cdot \end{bmatrix} = \begin{bmatrix} & & & & \\ & & & & \\ & & & & \\ & & & & \\ \cdot & m & \cdot & & \end{bmatrix} \begin{bmatrix} & & & & \\ & & & & \\ & & & & \\ & & & & \\ \cdot & \cdot & n & \cdot & \cdot \end{bmatrix} \begin{bmatrix} & & & & \\ & & & & \\ & & & & \\ & & & & \\ \cdot & \cdot & n & \cdot & \cdot \end{bmatrix} Q_2^T$$

The most important part of this decomposition to consider is the matrix Σ , the diagonal matrix, whose elements are called the *singular values*, σ_i :

$$\Sigma = \begin{bmatrix} \sigma_1 & & & & & \\ & \sigma_2 & & & & \\ & & \ddots & & & \\ & & & \sigma_r & & \mathbf{0} \\ & & & & 0 & \\ \mathbf{0} & & & & & \ddots \\ & & & & & & 0 \end{bmatrix}$$

Being part of a diagonal matrix, these only multiply particular columns of Q_1 or Q_2 .³³ The size of the singular value tells you exactly how much influence the corresponding rows and columns of Q_1 and Q_2^T have over the original matrix. Clearly, if a singular value is tiny, very little of the corresponding rows and columns get added into the matrix A when it is reconstituted. It is quite common for a matrix to have some zero eigenvalues, as shown above. These zero elements on the diagonal indicate correspond to rows or columns that give *no* extra information. This indicates, of course, that not all of the matrix is useful—that it is *rank deficient*. Or, in terms of simultaneous equations, that not all the equations are independent.

The SVD is a very useful tool for analysing a matrix, and the solutions to many problems can simply be read off from the decomposed matrices. For example, the null-space of the matrix (the solution to $A\mathbf{x} = \mathbf{0}$) is simply the columns of Q_2 which correspond

³³The singular values Σ of course multiply the *rows* of Q_2^T , and hence the columns of the un-transposed version, Q_2 .

to the zero singular values. The SVD is also highly numerically stable—matrices which are slightly ill-conditioned stand more chance of working with SVD than with anything else.

13.3 Approximating matrices: Changing rank

Another important use of the SVD is the insight it gives into how much of a matrix is important, and how to simplify a matrix with minimal disruption. The singular values are usually arranged in order of size, with the first, σ_1 , being the largest and most significant. The corresponding columns of Q_1 and Q_2 are therefore also arranged in importance. What this means is that, while we can find the exact value of A by multiplying $Q_1 \Sigma Q_2^T$, if we removed (for example) the last columns of Q_1 and Q_2 , and the final singular value, we would be removing the least important data. If we then multiplied these simpler matrices, we would only get an approximation to A , but one which still contained all but the most insignificant information. Exercise 6 will consider this in more detail.

13.4 The svd function

The SVD is performed in OCTAVE using the `svd` function. As with the `eig` function, by itself it only returns the singular values, but if given three matrices for the answer then it will fill them in with the full decomposition:

```
octave:##> A=[1 3 -2 3
>           3 5 1 5
>           -2 1 4 2];
octave:##> svd(A)
ans =

    8.9310
    5.0412
    1.6801

octave:##> [U,S,V] = svd(A,0)
U =

   -4.6734e-01    3.8640e-01    7.9516e-01
   -8.6205e-01    3.3920e-04   -5.0682e-01
   -1.9611e-01   -9.2233e-01    3.3294e-01

S =

    8.93102    0.00000    0.00000
    0.00000    5.04125    0.00000
    0.00000    0.00000    1.68010

V =

   -0.297983    0.442764   -0.828029
```

```

-0.661559    0.047326    0.109729
-0.079700   -0.885056   -0.455551
-0.683516   -0.135631    0.307898
octave:##> U*S*V' % Check Answers
ans =

    1.00000    3.00000   -2.00000    3.00000
    3.00000    5.00000    1.00000    5.00000
   -2.00000    1.00000    4.00000    2.00000

```

Note that OCTAVE automatically orders the singular values in decreasing order.

13.5 Economy SVD

If a $m \times n$ matrix A is overdetermined (i.e. $m > n$), then the matrix of singular values, Σ , will have at least one row of zeros:

```

octave:##> [U,S,V]=svd(A)
U =

   -0.22985    0.88346    0.40825
   -0.52474    0.24078   -0.81650
   -0.81964   -0.40190    0.40825

S =

    9.52552    0.00000
    0.00000    0.51430
    0.00000    0.00000

V =

   -0.61963   -0.78489
   -0.78489    0.61963

```

If we then consider the multiplication $A = U*S*V'$, we can see that the last column of U serves no useful purpose—it multiplies the zeros in S , and so might as well not be there. Therefore this last column may be safely ignored, and the same is true of the last row of S . What we are then left with is matrices of the following form:

$$\begin{array}{c} A \\ \left[\begin{array}{ccc} & \cdot & \\ & \cdot & \\ & m & \\ & \cdot & \\ \cdot & n & \cdot \end{array} \right] \end{array} = \begin{array}{c} Q_1 \\ \left[\begin{array}{ccc} & \cdot & \\ & \cdot & \\ & m & \\ & \cdot & \\ \cdot & n & \cdot \end{array} \right] \end{array} \begin{array}{c} \Sigma \\ \left[\begin{array}{ccc} & \cdot & \\ & n & \\ \cdot & n & \cdot \end{array} \right] \end{array} \begin{array}{c} Q_2^T \\ \left[\begin{array}{ccc} & \cdot & \\ & n & \\ \cdot & n & \cdot \end{array} \right] \end{array}$$

OCTAVE can be asked to perform this ‘economy’ SVD by adding ‘,0’ to the function argument (NB this is a zero, not the letter ‘O’):

```
octave:##> [U,S,V]=svd(A,0)
```

```
U =
```

```
-0.22985    0.88346
-0.52474    0.24078
-0.81964   -0.40190
```

```
S =
```

```
9.52552    0.00000
0.00000    0.51430
```

```
V =
```

```
-0.61963   -0.78489
-0.78489    0.61963
```

This is highly recommended for matrices of this shape, since it takes far less memory and time to process.

14 Complex numbers

Apart from matrix and vector computations, OCTAVE also supports many other mathematical and engineering concepts, and among these are complex numbers. Complex numbers can be typed into OCTAVE exactly as written, for example

```
octave:##> z1=4-3i
```

```
z1 = 4 - 3i
```

Alternatively, both `i` and `j` are initialised by OCTAVE to be $\sqrt{-1}$ and so (if you've not redefined them) you can also type

```
octave:##> z2 = 1 + 3*j
```

```
z2 = 1 + 3i
```

You can perform all the usual arithmetic operations on the complex numbers, exactly as for real numbers:

```
octave:##> z2-z1
```

```
ans = -3 + 6i
```

```
octave:##> z1+z2
```

```
ans = 5
```

```
octave:##> z2/z1
```

```
ans = -0.20000 + 0.60000i
```

```
octave:##> z1^2
```

```
ans = 7 - 24i
```

OCTAVE also provides some functions to perform the other standard complex number operations, such as the complex conjugate, or the modulus and phase of the number, and these are shown in Table 9. Other functions which have mathematical definitions for complex numbers, such as $\sin(x)$ or e^x , also work with complex numbers:

```
octave:##> sin(z2)
ans = 8.4716 + 5.4127i
octave:##> exp(j*pi)    %Should be 1
ans = -1.0000e+00 + 1.2246e-16i
octave:##> exp(j*2)
ans = -0.41615 + 0.90930i
octave:##> cos(2) + j*sin(2) %Should be the same as e^2i
ans = -0.41615 + 0.90930i
```

Matrices and vectors can, of course, also contain complex numbers.

function	meaning	definition ($z = a + bi$)
imag	Imaginary part	a
real	Real part	b
abs	Absolute value (modulus)	$r = z $
conj	Complex conjugate	$\bar{z} = a - bi$
angle	Phase angle (argument)	$\theta = \tan^{-1}\left(\frac{b}{a}\right)$

Table 9: Complex number functions

14.1 Plotting complex numbers

The `plot` command in OCTAVE also understands complex numbers, so you can use this to produce an Argand diagram (where the x -axis represents the real component and the y -axis the imaginary):

```
octave:##> plot(z1,'*', z2,'*')
octave:##> axis([-5 5 -5 5])
```

These commands produce the plot shown in Figure 14.1.

14.2 Finding roots of polynomials

OCTAVE provides a function, `roots`, which can be used to find the roots of polynomial equations. The equation is entered into OCTAVE, surprise, surprise, using a vector for the array of coefficients. For example, the polynomial equation

$$x^5 + 2x^4 - 5x^3 + x + 3 = 0$$

would be represented as

```
octave:##> c = [1 2 -5 0 1 3];
```

The `roots` function is then called using this vector:

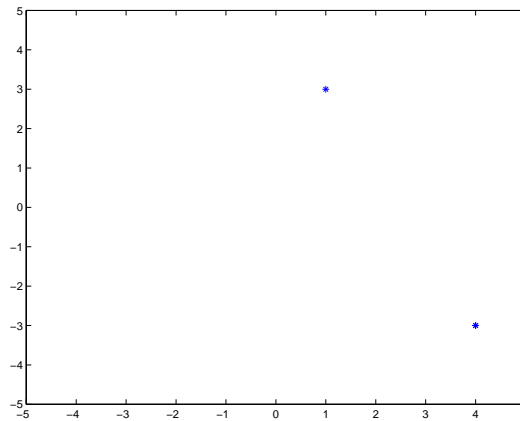


Figure 11: Graphical representation of the complex numbers $z_1 = 4 - 3i$ and $z_2 = 1 + 3i$ using the `plot` command.

```
octave:##> roots(c)
ans =

-3.44726 + 0.00000i
 1.17303 + 0.39021i
 1.17303 - 0.39021i
-0.44940 + 0.60621i
-0.44940 - 0.60621i
```

15 Appendix - Setup conditions

By default Octave distribution obtained from www.octave.org has some options set to minimise problems. In the implementation included on the MDP cd the following options have been set in the `.octaverc` file.

```
global gnuplot_has_pm3d=1
gset mouse
automatic_replot=1
```

16 Further reading

This tutorial has introduced the basic elements of using OCTAVE. The main document for the use of OCTAVE is the OCTAVE manual which is included in the MDP distribution and can be obtained from <http://www.octave.org>. A print version is also available with profits going to further development of the code.

In addition there is the on-line help and lists of available functions at <http://octave.sourceforge.net/index> and included on the MDP resources.

In addition searching for `octave + tutorial` on the web reveals a number of on-line tutorials.

Also most of the MATLAB tutorials and textbooks are also relevant, including:

- *Matlab 6 for engineers*, A. Biran, M. Breiner, Pearson Education, 2002
- *Getting Started with MATLAB: A Quick Introduction for Scientists and Engineers*, R. Pratap, Oxford University Press, 2002

17 Acknowledgements

This document has been produced as a tutorial to accompany the version of Octave supplied on the Cambridge-MIT Institute (CMI) ³⁴ funded Multidisciplinary Design Project (MDP) Resource CD ³⁵.

The text of this tutorial has been adapted from the student notes developed by Dr. Paul Smith from the Cambridge University Engineering Department for a 2nd year Introduction to Matlab course. Conversion from Paul Smith's original to this current document has been relatively easy with in many cases needing only to exchange

- **Octave** for **Matlab** in the text
- the prompt and precise format of the answers given in the example scripts
- removal of specific course administration.

However in a number of cases there are significant differences, e.g. graphical performance (Gnuplot is assumed to be the output plugin for Octave), where changes have had to be made.

Any success of this tutorial is a major result of the background work carried out by Tim Froggatt in setting up the MDP resource distribution and requests from Gareth Wilson, the author of the RED Tools web interface for Octave scripts, for additional information and scripts.

Significant reference has been made to the resources generated and maintained by John Eaton and his team at **www.octave.org**. As indicated above, the tutorial is heavily based on Paul Smith's original in which he acknowledges advice and help from Roberto Cipolla, Richard Prager, Hugh Hunt, Maurice Ringer, David Mansergh and Martin Szummer.

³⁴CMI - www.cambridge-mit.org

³⁵MDP - <http://www-mdp.eng.cam.ac.uk>