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)

DRAFT VERSION 0.0.1
PLEASE email any errors to mdp-support@eng.cam.ac.uk

January 2005

This document has been produced as a tutorial to accompany the version of Octave supplied on the Cambridge-MIT Institute (CMI) [1] funded Multidisciplinary Design Project (MDP) Resource CD [2]. The close compatibility of the open-source Octave [3] package with MATLAB[4], 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.

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 which is a follow-on from a 1st year Introduction to C++. The course is run as a series of four laboratory practicals with the exercises given in this tutorial forming the basis for submitted work.

Conversion from Paul Smith's original to this current document has be relative 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 adminstration.

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

To maintain the ideal of learning both Octave and Matlab from this tutorial, the differences between Ocatve 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 MDP distribution and documented in aan appendix.

ACKNOWLEDGEMENTS

The any success of this tutorial is a major result of the background wok 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 and his acknowledgements are duplicated here

> ..... My thanks go to Roberto Cipolla and Richard Prager for sharing their advice and experience, to Hugh Hunt for MATLAB tips, and to Maurice Ringer for drawing the cricket stumps. And to all of those who tested early versions of this for me.
> The cricket example is based on David Mansergh's 2000–2001 4th year project. The video texture sequences are borrowed from Martin Szummer at MIT's AI lab. Inspiration was also drawn from various other OCTAVE tutorials and text books.

[1] CMI - www.cambridge-mit.org
[2] MDP - http://www-mdp.eng.cam.ac.uk
[3] www.octave.org
[4] MATLAB® The Mathworks, www.mathworks.com

Contents

1 Introduction **5**
1.1 What is OCTAVE? . . . 5
1.2 What OCTAVE is not . . . 5
1.3 Who uses OCTAVE? . . . 5
1.4 Why not use a 'normal' highlevel language, e.g. C++ . . . 5

2 Simple calculations **6**
2.1 Starting OCTAVE . . . 6
2.2 OCTAVE as a calculator . . . 6
2.3 Built-in functions . . . 7

3 The Octave environment **7**
3.1 Named variables . . . 9
3.2 Numbers and formatting . . . 10
3.3 Number representation and accuracy . . . 11
3.4 Loading and saving data . . . 11
3.5 Repeating previous commands . . . 12
3.6 Getting help . . . 13
3.7 Cancelling a command . . . 14
3.8 Semicolons and hiding answers . . . 14

4 Arrays and vectors **14**
4.1 Building vectors . . . 14
4.2 The colon notation . . . 15
4.3 Vector creation functions . . . 16
4.4 Extracting elements from a vector . . . 16
4.5 Vector maths . . . 17

5 Plotting graphs **18**
5.1 Improving the presentation . . . 19
5.2 Multiple graphs . . . 21
5.3 Multiple figures . . . 22
5.4 Manual scaling . . . 22
5.5 Saving and printing figures . . . 24

6 Octave programming I: Script files **25**
6.1 Creating and editing a script . . . 25
6.2 Running and debugging scripts . . . 26
6.3 Remembering previous scripts . . . 26

7 Control statements **27**
7.1 `if...else` selection . . . 27
7.2 `switch` selection . . . 28
7.3 `for` loops . . . 29
7.4 `while` loops . . . 30
7.5 Accuracy and precision . . . 30

8 Octave programming II: Functions **34**
8.1 Example 1: Sine in degrees 34
8.2 Creating and using functions 35
8.3 Example 2: Unit step 36
8.4 Summary 39

9 Matrices and vectors **42**
9.1 Matrix multiplication 42
9.2 The transpose operator 43
9.3 Matrix creation functions 44
9.4 Building composite matrices 45
9.5 Matrices as tables 45
9.6 Extracting bits of matrices 46

10 Basic matrix functions **46**

11 Solving `Ax` = b **51**
11.1 Solution when `A` is invertible 51
11.2 Gaussian elimination and LU factorisation 52
11.3 Matrix division and the slash operator 52
11.4 Singular matrices and `rank` 52
11.5 Ill-conditioning 54
11.6 Over-determined systems: Least squares 55
11.7 Example: Triangulation 55

12 More graphs **56**
12.1 Putting several graphs in one window 56
12.2 3D plots 57
12.3 Changing the viewpoint 57
12.4 Plotting surfaces 58
12.5 Images and Movies 59

13 Eigenvectors and the Singular Value Decomposition **64**
13.1 The `eig` function 64
13.2 The Singular Value Decomposition 65
13.3 Approximating matrices: Changing rank 66
13.4 The `svd` function 66
13.5 Economy SVD 67

14 Complex numbers **69**
14.1 Plotting complex numbers 69
14.2 Finding roots of polynomials 70

15 Appendix - Setup conditions **71**

16 Further reading **71**

1 Introduction

1.1 What is Octave?

Octave is an opensource 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 form of solutions. 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 though 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 design[5]. 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

[5] www.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 MERCHANTIBILITY 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 example[6]

```
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.[7]

[6] A 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. $e^1 = 2.7183$).

[7] OCTAVE can call external C++ functions however the functionality is less than MATLAB.

`cos`	Cosine of an angle (in radians)
`sin`	Sine of an angle (in radians)
`tan`	Tangent of an angle (in radians)
`exp`	Exponential function (e^x)
`log`	Natural logarithm (NB this is $\log_e$, not $\log_{10}$)
`log10`	Logarithm to base 10
`sinh`	Hyperbolic sine
`cosh`	Hyperbolic cosine
`tanh`	Hyperbolic tangent
`acos`	Inverse cosine
`acosh`	Inverse hyperbolic cosine
`asin`	Inverse sine
`asinh`	Inverse hyperbolic sine
`atan`	Inverse tangent
`atan2`	Two-argument form of inverse tangent
`atanh`	Inverse hyperbolic tangent
`abs`	Absolute value
`sign`	Sign of the number (-1 or $+1$)
`round`	Round to the nearest integer
`floor`	Round down (towards minus infinity)
`ceil`	Round up (towards plus infinity)
`fix`	Round towards zero
`rem`	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.[8] 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)[9], it stores them internally, and in variables, to a much higher precision, so the answer given is the more accurate one.[10] 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

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

[9] MATLAB normally displays to 4 or 5 decimal places

[10] OCTAVE 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, variables[11] 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
```

[11] Normally 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. One such file is `cricket.txt`, which will be considered in Exercise 5. If you type

```
octave:##>  load cricket.txt
```

this will create a new matrix, called `cricket`, which contains this data.

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 ↑ and ↓ 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.

If you are looking for a particular previous command, and you know the first few letters of the line you typed, you can type those letters and then press ↑, and it will find all previous lines starting with those letters. For example, typing `d`↑ will find first the `deg` command you typed earlier, and then if you press ↑ again, it will find the the `deg = pi/180` line.

Once a command has been recalled, you can edit it before running it again. You can use ← and → 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.[12]

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
```

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.
```

[13]If 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 keybindings **U**p, **N**ext, **P**revious. etc. Or directly from the prompt, e.g. to find ou about arithmetic functions type;

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

---

[12]On some systems the emacs key bindings are acknowledged, e.g. <ctrl>p =$\uparrow$, <ctrl>n=$\downarrow$, <ctrl>f=$\rightarrow$, < ctrl>b=$\leftarrow$

[13]MATLAB 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()`

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. [14]

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 lineends, 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

[14] MATLAB has some additional search facilities including the function lookfor which allows the user to search the help database

```
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 =
```

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

Table 2: Vector creation functions

```
   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 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.

4.4 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
```

4.5 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 (see the IA C++ Tutorial, pages 29–31). 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.[15] 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 =
     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:

[15] Recall 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.

```
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 opensource program GNUPLOT[16], however some of the range of plotting options are restricted relative to 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.[17] The sine curve defined in the previous section can be plotted by typing

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

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:

[16] www.gnuplot.org
[17] The two vectors must, naturally, both be the same length.

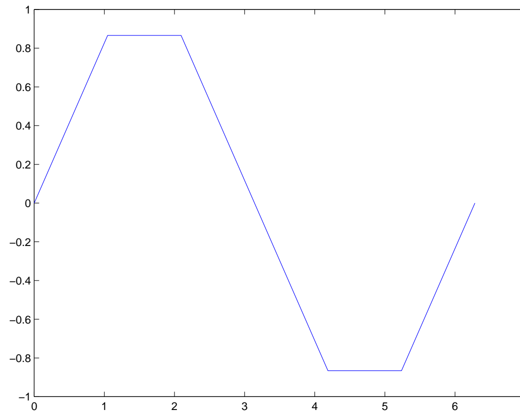


Figure 1: Graph of $y = \sin(x)$, sampled every 60°.

```
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 ↑ and ↓ to go back and reuse your previous commands. typeouttest

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 (').

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

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

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

`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)

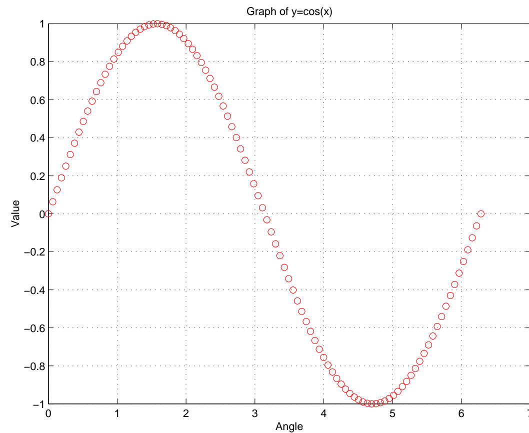


Figure 2: Graph of $y = \sin(x)$, marking each sample point with a red circle.

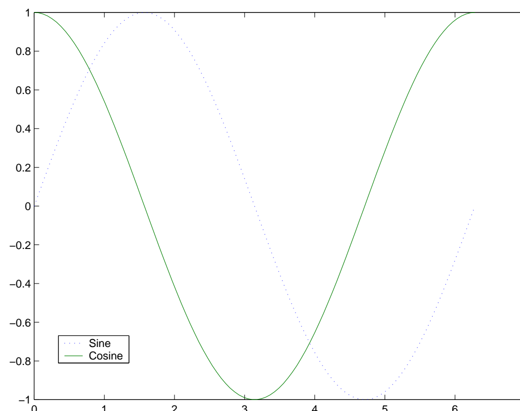


Figure 3: Graphs of $y = \sin(x)$ and $y = \cos(x)$.

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),'-')
```

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.

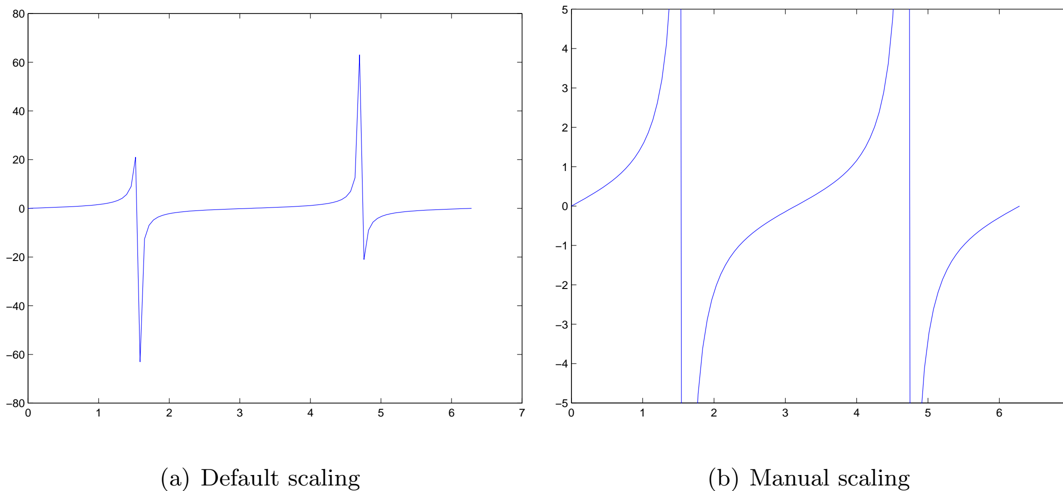
Thus far, every time you have issued a `plot` command, the existing contents of the figure have been removed. If you want to keep the current graph, and overlay new plots on top of it, you can use the `hold` command. Using this, the two sine and cosine graphs could have been drawn using two separate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

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

octave:##> hold on

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

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



(a) Default scaling

(b) Manual scaling

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

Notice that if you do this Octave does not automatically mark the graphs with different colours, but it can still work out the legend. 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
```

a new window will appear, with the title 'Figure No. 1'. All `plot` commands will now go to this new window. For example, typing

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

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

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.

Mouse	**Actions**
MMB	Annotate the graph.
RMB	Mark Zoom region.
Key Bindings	**Actions**
a	Autoscale and Replot
b	Toggle Border
e	Replot (removes annotations)
g	Toggle grid
h	Help
l	Toggle Logscales
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‘
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.

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 and select with the left mouse button. Details of other mouse actions and keybindings (when the graphic window is selected) are shown in table 4. [18]

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

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

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 its 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.

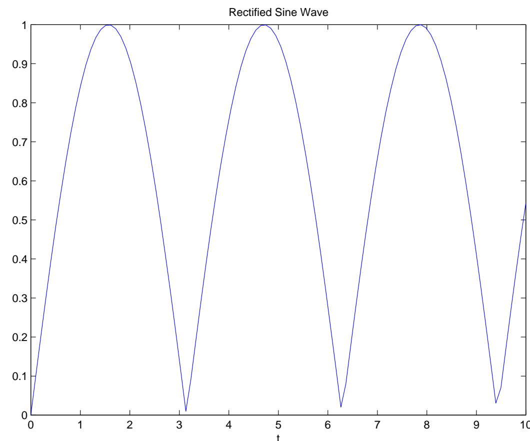


Figure 5: A rectified sine wave, produced by the `rectsin` script

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/retsin.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
```

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

```
drawstumps.m randommovie.m rectsin.m

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

flush.mat fountain.mat steam.mat
```

You should see that as well as `rectsin` there are two more scripts (M-files) in the directory, which will be needed for later exercises. The MAT-files are data files (see section 3.4) needed for Exercise 6.

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. You should be familiar with all of these from the IA C++ computing course.

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:

<table>
<tr><th>symbol</th><th>meaning</th><th>example</th></tr>
<tr><td>==</td><td>equal</td><td>if x == y</td></tr>
<tr><td>~=</td><td>not equal</td><td>if x ~= y</td></tr>
<tr><td>></td><td>greater than</td><td>if x > y</td></tr>
<tr><td>>=</td><td>greater than or equal</td><td>if x >= y</td></tr>
<tr><td><</td><td>less than</td><td>if x < y</td></tr>
<tr><td><=</td><td>less than or equal</td><td>if x <= y</td></tr>
<tr><td>&</td><td>AND</td><td>if x == 1 & y > 2</td></tr>
<tr><td>|</td><td>OR</td><td>if x == 1 | y > 2</td></tr>
<tr><td>~</td><td>NOT</td><td>x = ~y</td></tr>
</table>

Table 5: Boolean expressions

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

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 `case`s, 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
```

[21] This 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'.

where *vector* contains the numbers to step though. 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:

```
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

[22]The 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.

the two numbers in the operation are of different orders of magnitude, and OCTAVE can't simultaneously maintain its accuracy in both.

Consider this example:

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

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:

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

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 of about the same magnitude.

Summary

After reading through sections 6 and 7 of the tutorial guide and working through the examples you should be able to:

- ☐ Create and use scripts to store a series of commands
- ☐ Use loops and conditionals to control program execution

Exercise 2: Finding π using the arctangent series

A. Theory

The search to determine π to a greater and greater accuracy has occupied mathematicians for millennia. In the IA C++ course you considered two methods for calculating π, one by Euler and one by Archimedes. Archimedes's approach, of inscribing and circumscribing circles, was the method of choice from the 3rd century BC until the 17th century AD. In 1671, James Gregory (1638–1675) found the now-standard power series for arctangent, which can be found in the Mathematics Databook:

$$\tan^{-1} z = z - \frac{z^3}{3} + \frac{z^5}{5} - \cdots$$

Three years later, Gottfried Wilhelm Leibniz (1646–1716) independently found and published the same series, noting a special case: that $\tan \frac{\pi}{4} = 1$. In other words, putting $z = 1$ into the arctangent series, gives an approximation for $\frac{\pi}{4}$, and hence for π. The arctangent series is still the way that π is calculated today.

In 1699, Abraham Sharp (1651-1742) broke the previous record for the accuracy of π, finding 72 digits (which may not seem many, but is more than the native accuracy of Octave). He also used the arctangent series but realised that if, instead of $z = 1$, he used the identity $\tan \frac{\pi}{6} = \frac{1}{\sqrt{3}}$, he would get much quicker convergence.

B. Exercise

1. Write a Octave script to calculate the first 50 terms in the arctangent series using $z = 1$ and plot a graph showing the value of π estimated after each term. Calculate the difference between your estimate and π after 25, 30 and 50 terms.

2. Now modify the script to also calculate the first 50 terms in the estimate for π using $z = \frac{1}{\sqrt{3}}$, plotting these in a different colour on the same graph. Why do you think the second series converges so much faster? Again, calculate the difference between your estimate and π after 25, 30 and 50 terms. What behaviour do you see, and why?

C. Implementation notes

i. **Try things out in Octave first**
Remember that a Octave script is just a series of commands that you could equally well type in at the command line. Try things out in Octave first of all, before adding them to the script.

ii. **Read error messages from bottom to top**
If you get errors when you run your script, remember that these are best read from bottom to top.

iii. **Alternating signs and odd numbers**
You can get a sequence of alternating signs by using $(-1)^n$, which is $+1$ when n is

even, and -1 when n is odd. To produce only odd numbers from a counting sequence involving all positive integers n, you can use $(2n-1)$.

iv. **Don't plot points inside the loop**
Producing output to the screen or a file is the slowest part of many programs, and you should think about how to keep I/O to a minimum if you want to write an efficient program. If you plot each point as you calculate it inside your loop, the program will run very slowly. Instead, store your points in a vector as you calculate them and then output them all in one go at the end of the script. This will also allow you to join the points up, and to easily extract the spot points that the exercise asks for.

D. Understanding

Why does the $z = 1/\sqrt{3}$ series converges faster, and why does it behave as it does for a high number of terms.

Exercise 2*: Vectorising

The usual solution to Exercise 2 (and a perfectly acceptable one) is to use a `for` loop to step through the series, calculating the term and adding it on. However, the solution can instead be vectorised, i.e. performed entirely using vector calculations. This runs much faster (because OCTAVE is good at vectors), and can be written in just one line. Try writing the general equation for the nth term of the series in such a way that it will also work when a vector is given to it. Give it a vector of numbers from 1 to 50, pass the result through the `cumsum` (look it up in the help system) function and finally plot that vector to give the answer. Try this approach, and the solution with a `for` loop, with 50,000 terms to see the difference that vectorising makes. Be prepared to wait a while for the `for` loop solution!

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.

You will have come across functions in the C++ computing course, where they are used for exactly the same purposes as in Octave. However, Octave functions are somewhat simpler: 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 an 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.

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.

[23]When 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.

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`.

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.

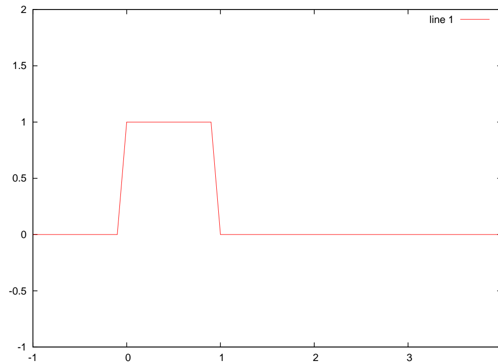


Figure 6: A unit, one second pulse created from two unit steps, using the function `ustep`.

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.

8.4 Summary

After reading through sections 8 of the tutorial guide and working through the examples you should be able to:

☐ Define and use your own functions

Exercise 3: Finding roots by the Newton-Raphson method

A. Theory

You may remember that in the IA C++ computing exercise you found the solution to $f(x) = 0$ using the *Bisection method*, which simply finds points where the function changes sign. This is a perfectly acceptable method, but can converge rather slowly. A better approach is to use extra information: the derivative of the function as well as its value, and this is the basis of *Newton's method*, commonly called the *Newton-Raphson method*.

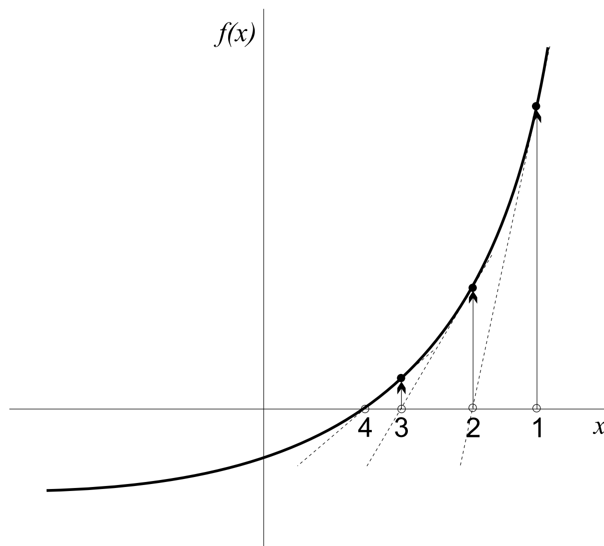


Figure 7: Example of the Newton-Raphson method. Starting from an initial guess (labelled '1'), the local derivative is extrapolated to generate the next guess of the root, and so on.

The method is best understood graphically, as demonstrated in Figure 7. The tangent line to the curve is calculated at the current guess, x_i, (here starting at point '1') and is extrapolated until it crosses the x-axis. This x-value is then the next guess, x_{i+1}, and the process is repeated.

Algebraically, Newton-Raphson derives from the Taylor series, with which you should already be familiar from the IA Mathematics course. This states how the value of the function at a new (nearby) point $x + h$ can be calculated from the value and derivatives at a known point x (Maths databook page 3):

$$f(x + h) = f(x) + hf'(x) + \frac{h^2}{2!}f''(x) + \dots$$

To find a root of our function, let us say that our current guess is at x and we want to find out how much to move to get to the root. Calling this correction term h, this means

that the root is at $x + h$ and so, by definition, $f(x + h) = 0$. Using the Taylor series expansion:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!}f''(x) + \ldots = 0$$

If h is small and the function is well-behaved then we can neglect the higher-order terms and just use a linear approximation, needing only the first derivative:

$$f(x) + hf'(x) \approx 0$$

and so the correction term h can be estimated by

$$h \approx -\frac{f(x)}{f'(x)}$$

The update rule for Newton-Raphson just applies this correction at each iteration, i.e.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{1}$$

Given an initial guess, x_0, we iterate according to equation (1) until the desired accuracy is reached. Being a numerical solution, and given the limits to floating-point accuracy, it may not be possible to *exactly* reach the root. Instead, it is usual to continue until the difference between successive estimates is below a certain tolerance (i.e. we are close enough for our purposes).

The Newton-Raphson method is very powerful, and it is the method of choice for any function whose derivative can be evaluated efficiently, and is continuous and non-zero in the neighbourhood of the root. It is also not restricted to just one-dimensional problems, and readily generalises to multiple dimensions. Its main power lies in its rate of convergence: near the root it converges *quadratically*, meaning that the number of significant digits approximately *doubles* with each step. However, it is important to start near the neighbourhood of the root and to be sure the function is well-behaved there, so plotting the graph of the function whose root you are finding is a very useful first step.

B. Exercise

We shall use the Newton-Raphson method to find the roots of

$$f(x) = x^5 - 5x^4 + x^2 - 8x + 50 \tag{2}$$

to a tolerance of 0.00001.

1. Write two functions: `fun(x)`, which returns the value of $f(x)$ (equation (2)) for a given value of x; and `dfun(x)`, which returns the value of its derivative, $f'(x)$. Plot both of these over the range -3 to 6. Check that they appear correct, and make initial guesses to the values of the roots (to the nearest whole number will do).

(Continued over the page)

2. Write a third function, `newton(x, tol)` which performs the Newton-Raphson estimation of a root. This should make use of `fun` and `dfun`, iterate according to equation (1) and return the final estimate. The first parameter, `x`, is the initial guess x_0. The second parameter, `tol` is the tolerance; the iteration stops once the difference between successive estimates is less than this.

3. Find all of the real roots of equation (2) to the specified tolerance. Which root do you find with an initial guess of 0, and why is this not one of the nearby roots? Are there any cases where the Newton-Raphson method will fail to find a root?

C. Implementation notes

i. **Writing functions**
Don't forget that each of your three functions needs to be in a separate M-file, each with the same name as the function except for the `.m` added. Each file should begin with a `function` line, and then some help comments.

ii. **Loops**
Think carefully about what kind of loop you need in `newton.m` to perform the iteration, and what the termination criterion is.

iii. **Absolute errors**
Remember, when you check whether the remaining error is less than `tol`, that the error value could be negative. Use the `abs` function to take the absolute value of this error.

Exercise 3*: Passing functions (optional extension)

The `newton` function written for Exercise 3 is not as general as it could be, since it assumes that the functions called `fun` and `dfun` exist, and do what they are supposed to do. This can be improved by passing the names of the functions to `newton` as two additional parameters. These function names are passed as strings, and you can tell OCTAVE to run a function named by a string using the `feval` command: `feval(`*`fname, parameters, ...`*`)` where *`fname`* is the function name string, and *`parameters, ...`* are the parameters to pass to it. Try modifying your `newton` function to use `feval` rather than referring to `fun` and `dfun`.

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:

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

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 `A*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 `A` is represented as $\mathtt{A}^T$. In Octave an apostrophe performs this operation:

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

octave:##>   A'
```

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

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

When we try to do `A*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.

```
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 `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.

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:

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

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 nth 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.

`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

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 sursor 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 ($AA^{-1} = A^{-1}A = I$). 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:

```
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
```

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

```
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
```

Summary

After reading through sections 9 and 10 of the tutorial guide and working through the examples you should be able to:

- ☐ Define matrices in OCTAVE
- ☐ Use matrices and vectors in simple calculations
- ☐ Perform standard operations on matrices

Exercise 4: Euler angles

A. Theory

A rotation matrix in two dimensions is easy to define. There is only one axis of rotation, which is normal to the 2D plane, and a rotation of a positive angle θ (in a right-handed co-ordinate system) is given by the matrix

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Defining rotations in three dimensions is more difficult. One method of defining rotations is known as *Euler angles*, which are often used in mechanics.[26]

The method of Euler angles, specifies three angles. These are commonly called azimuth, elevation and roll (or in aeronautical terms yaw, pitch and roll). We shall label these three angles as ψ, θ and ϕ, and our object exists in a normal right-handed co-ordinate set, x, y and z.

The rotations must be applied in a specific order. A point (x, y, z) is first rotated from its original location by an angle ψ about the x axis (roll):

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & -\sin\psi \\ 0 & \sin\psi & \cos\psi \end{bmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Then it is rotated by an angle θ about the original y axis (elevation):

$$\begin{pmatrix} x'' \\ y'' \\ z'' \end{pmatrix} = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix}$$

The final rotation, ϕ is then about the original z axis (azimuth), defining the new location x''', y''' and z''' as

$$\begin{pmatrix} x''' \\ y''' \\ z''' \end{pmatrix} = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x'' \\ y'' \\ z'' \end{pmatrix}$$

B. Computing exercise

1. Write a OCTAVE function which takes as parameters three Euler angles and returns a single 3D rotation matrix, transforming points (x, y, z) to (x''', y''', z'''), as defined above.

2. Now write a separate script, in which you call the function written in part 1 in order to find the rotation matrix `R` which rotates a co-ordinate system by the Euler angles $(90^\circ, 20^\circ, 15^\circ)$. After calling this function, your script should verify that the matrix returned is a valid rotation matrix i.e. that it is orthogonal, and has a determinant of $+1$.

[26] Named after Leonhard Euler (1707–1783) who was responsible for much of the early work on differential equations.

3. The point $\mathbf{p} = (2, 3, 0)^T$ is to be mapped to a new location $\mathbf{q}$ using the matrix `R` from part 2. Add commands to your script to define $\mathbf{p}$ and find $\mathbf{q}$. Which matrix reverses this transformation?

4. Extend your script to also find the matrix `S` which represents the rotation given by $(-90°, -20°, -15°)$. Transform the vector $\mathbf{q}$, found in part 3, using the matrix `S` to give a new vector $\mathbf{r}$. Why does $\mathbf{r} \neq \mathbf{p}$?

C. Implementation notes

i. **Degrees and radians**
Remember that OCTAVE works in radians, not degrees. You might like to make use of the `sind` function you defined earlier, and also to define a `cosd` function.

11 Solving $\mathtt{A}\mathbf{x} = \mathbf{b}$

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

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

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

$$\mathtt{A}\mathbf{x} = \mathbf{b}$$

In this representation, `A` is the matrix of coefficients, **b** are the constants, and **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 `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 `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 `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).[27] These provide a more efficient way of solving $\mathtt{A}\mathbf{x} = \mathbf{b}$, 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 $\mathtt{A}\mathbf{x} = \mathbf{b}$, however, division is not defined and the solution is given by $\mathbf{x} = \mathtt{A}^{-1}\mathbf{b}$. 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 $\mathtt{A}^{-1}\mathbf{b}$.

You should recall that matrix multiplication is not commutative i.e. $\mathtt{AB} \neq \mathtt{BA}$. This means that, while the solution to $\mathtt{AX} = \mathtt{B}$ is given by $\mathtt{X} = \mathtt{A}^{-1}\mathtt{B}$, the solution to $\mathtt{XA} = \mathtt{B}$ is $\mathtt{X} = \mathtt{BA}^{-1}$, 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
\	$\mathtt{AX} = \mathtt{B}$	A\B	$\mathtt{A}^{-1}\mathtt{B}$	left division (backslash)
/	$\mathtt{XA} = \mathtt{B}$	B/A	$\mathtt{BA}^{-1}$	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:

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

[27] The `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 $\mathtt{A}\mathbf{x} = \mathbf{b}$. 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 `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 `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.[28]

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.[29] For the singular

[28] Some 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.

[29] This 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 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![30]

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 $\mathtt{A}\mathbf{x} = \mathbf{b}$, the least-squares solution is given by the *pseudoinverse* (see your IB Linear Algebra notes):

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

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 8. 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 =
```

[30] The 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.

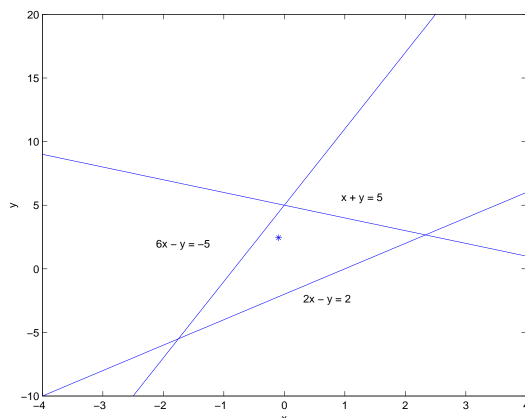


Figure 8: Three lines and the least squares estimate of the common intersection point

```
-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 8 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 9.

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

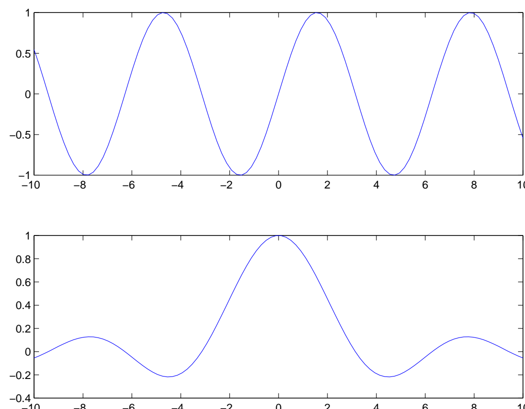


Figure 9: Example of the `subplot` command, showing a sine and sinc curve one above the other.

```
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.[31]

```
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 10. 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 8[32]

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

[31] A 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.'

[32] Matlab has an enhanced graphical interaction using a GUI

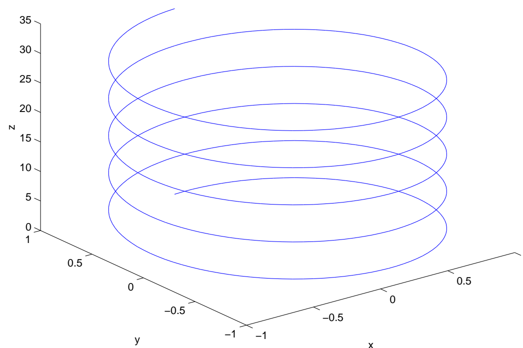


Figure 10: Example of the `plot3` command, showing a helix.

Mouse	Actions
LMB + motion	Rotate View
<Ctrl> + LMB + motion	Rotate axes (View update when button released)
MMB + motion	Scale View.
<Ctrl> + MMB + motion	Scale axes (View update when button released)
<Shift> + MMB + motion	vertical motion – change ticslevel
Key Bindings	**Actions**
$\rightarrow$	Rotate right (<shift> increases amount)
$\uparrow$	Rotate up (<shift> increases amount)
$\leftarrow$	Rotate left (< shift> increases amount)
$\downarrow$	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

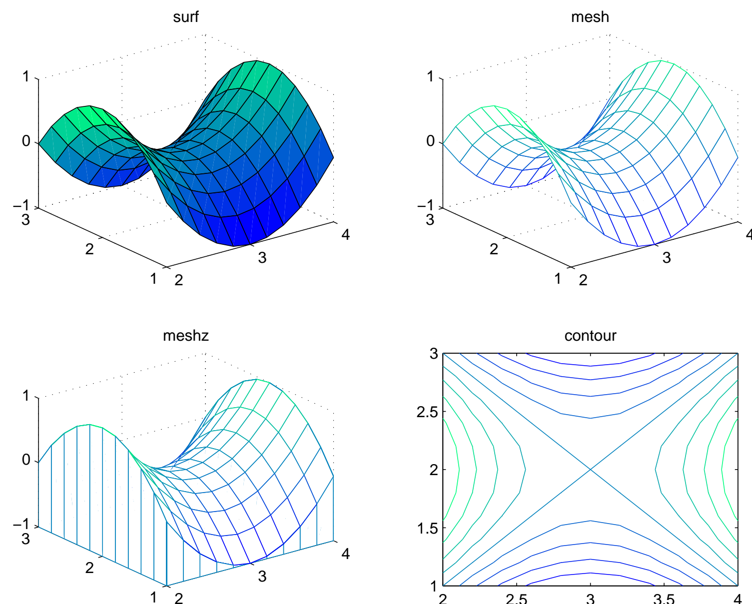


Figure 11: Examples of the same saddle-shaped surface visualised using different OCTAVE commands. (The 4 graphs are plotted in the same figure using the command `subplot(2,2,i)`.)

```
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 11 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. There are several example images supplied with OCTAVE on the MDP CD and MDP website[33], and the following commands load and display one of them:

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

The file `cued.mat` contains the matrix `a`. If you look at the elements of this matrix, you will see that it simply contains numbers between 0 and 63, which represent the brightness of each pixel (0 is black and 63 is white). 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.[34] N.B. Octave uses Imagemagick, by default, as the display program giving the user full acces to file format conversion and printing.

[33] http://www-mdp.eng.cam.ac.uk

[34] Try `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.

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

Summary

After reading through Sections 11 and 12 of the tutorial guide and working through the examples you should be able to:

- ☐ Write linear simultaneous equations in matrix-vector form
- ☐ Use the slash operator in OCTAVE to solve $\mathtt{A}\mathbf{x} = \mathbf{b}$
- ☐ Understand what the slash operator does in different scenarios
- ☐ Appreciate the problems of ill-conditioned matrices
- ☐ Produce more advanced figures, including displaying three-dimensional data

---

Exercise 5: Leg Before Wicket?

A. Description

In high speed ball games, e.g. Tennis, Football and Cricket, the precise trajectory of a ball can lead to contentious decisions. The Leg Before Wicket (LBW) rule in cricket is the cause of much contention, but the problem is a very simple one: if the batsman had not got in the way, would the ball have hit the stumps? This is an extrapolation problem. We can track the ball's position from the point it bounces, until it hits the batsman, and the problem is then just one of fitting a curve to the observed trajectory and extrapolating this to see if it hits the stumps.

A tracking system estimates the ball's position at a rate of 150Hz, and for each sample stores the x, y and z co-ordinate of the ball. Measurements are in metres, according to the axes defined in Figure 12, centred on the bottom of the middle stump. A cricket ball is approximately 72mm in diameter.

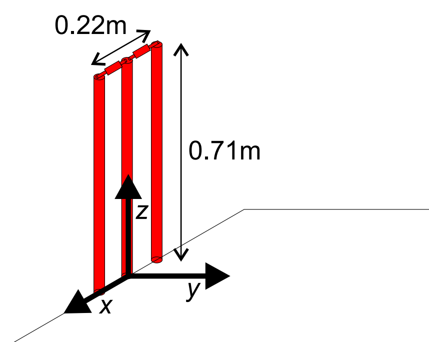


Figure 12: Definition of axes, and dimensions of the stumps

(Continued over the page)

B. Exercise

Write a program which will read a file giving ball position data, fit an extrapolated trajectory to the data, and determine whether the ball would have hit the stumps or not. There are three data files to be analysed: `cricket.txt`, `cricket2.txt` and `cricket3.txt`. Each file contains 4 columns of data, in the format `(`*`time x y z`*`)`, giving the ball positions from the point they pitched until the point the ball hit the batsman.

1. Write a script to load the `cricket.txt` file into OCTAVE and produce a three dimensional plot of the position data, marking each sample with a blob.

2. Assuming a linear horizontal trajectory (i.e. that the ball's velocity is constant in x and y), and that its height describes a parabola, express each of the three coordinates as a function of time. You should assume that $g = -9.81\,\mathrm{ms}^{-2}$. Extend your script to use least squares to estimate the motion parameters and so fit a curve to the sample points. The script should add your estimated curve to the 3D plot, extrapolating it forward in time so that it can be seen passing the stumps.

3. Add lines to your script to calculate the exact (extrapolated) position of the ball at the time it passes the stumps. Your script should then test this position to see whether the ball would hit the stumps, and print out a message giving the final decision as 'out' if it would have hit, or 'not out' otherwise. Use your script to determine whether the batsman in the case of `cricket.txt` should be given out or not.

4. You should ensure that your script is general enough to work with new sets of data. Test your script with the other two data sets, `cricket2.txt` and `cricket3.txt`, making any changes needed to your script, so that the only change necessary to try each new data file is to change the name of the file to be loaded. Determine whether the batsman should be given out in these other two cases.

C. Implementation notes

i. **Loading text files**
The `load` command can be used to load data from a text file. Its functional form, `N=load('file.txt')`, can be used to both load a file and assign its data to a particular variable name.

ii. **Number of samples**
The `size` command can be used to to determine the size of a matrix, and thus the number of samples in the a data file once it has been loaded. See Section 8.3 for an example of its use.

iii. **Solving equations of motion by least squares**
For each sample point you can write three equations in terms of time: one for x, one for y, and one for z. The first two are linear, in the form

$$x = x_0 + u_x t$$

And similarly for y. Each of these equations has two parameters, the speeds u_x and u_y, and the initial positions x_0 and y_0. The z motion will have an additional quadratic term involving g:

$$z = z_0 + u_z t + \tfrac{1}{2} g t^2$$

Despite having an extra term, this equation also has only two unknown parameters, since both z and $\frac{1}{2}gt^2$ are known.

The three equations may be written in matrix-vector form as

$$\begin{matrix} x \text{ equation} & \rightarrow \\ y \text{ equation} & \rightarrow \\ z \text{ equation} & \rightarrow \end{matrix} \qquad \underbrace{\begin{pmatrix} \cdot \\ \cdot \\ \cdot \end{pmatrix}}_{\mathbf{b}} = \underbrace{\begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}}_{\mathtt{A}} \begin{pmatrix} u_x \\ u_y \\ u_z \\ x_0 \\ y_0 \\ z_0 \end{pmatrix}$$

where the elements in `A` are given by a comparison of coefficients with the three motion equations, and **b** gathers together the constants from the three equations.

Each sample point gives different values of x, y and z, and so three different equations. Each of these equations gives different values for the matrix `A`, but the parameters u_x etc. *stay exactly the same* (these are our initial conditions, and define the fitted trajectory). We can therefore stack all of our equations on top of each other to create some huge matrices and vectors:

$$\begin{pmatrix} \mathbf{b}_1 \\ \hline \mathbf{b}_2 \\ \hline \vdots \\ \hline \mathbf{b}_n \end{pmatrix} = \begin{bmatrix} \mathtt{A}_1 \\ \hline \mathtt{A}_2 \\ \hline \vdots \\ \hline \mathtt{A}_n \end{bmatrix} \begin{pmatrix} u_x \\ u_y \\ u_z \\ x_0 \\ y_0 \\ z_0 \end{pmatrix}$$

This then an over-determined equation of the form $\mathtt{A}'\mathbf{x} = \mathbf{b}'$, and can be solved in OCTAVE with the slash operator or the pseudoinverse.

iv. **Building the matrix $\mathtt{A}'$**

Each data point from the text file contributes three equations, and so three lines to both $\mathtt{A}'$ and $\mathbf{b}'$. The best way to build this matrix and vector is with a `for` loop which steps through the rows of the raw data and, for each row, fills in three rows of $\mathtt{A}'$ and $\mathbf{b}'$. Remember that you can accumulate matrices as in this example:

(Continued over the page)

```
octave:##> A=[];
octave:##> A=[A; 1 2 3];
octave:##> A=[A; 4 5 6]
A =

     1     2     3

     4     5     6
```

(see also Section 9).

v. **The position of the stumps**
The dimensions of the stumps are given in Figure 12. In addition, the script `drawstumps.m` will draw a 3D representation of the stumps in the current figure window.

vi. **Axis scaling**
Your stumps and trajectory may look a little strange because of OCTAVE's automatic scaling of the axes. Typing `axis equal` will ensure that the scale on all three axes is the same.

13 Eigenvectors and the Singular Value Decomposition

After the solution of $\mathtt{A}\mathbf{x} = \mathbf{b}$, the other important matrix equation is $\mathtt{A}\mathbf{x} = \lambda\mathbf{x}$, the solutions to which are the *eigenvectors* and *eigenvalues* of the matrix 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 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 U is the matrix of eigenvectors and Λ the diagonal matrix of eigenvalues, then $\mathtt{A} = \mathtt{U}\Lambda\mathtt{U}^{-1}$. 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 A.

13.2 The Singular Value Decomposition

Eigenvectors and eigenvalues can only be found for a square matrix, and thus the decomposition into $\mathtt{U\Lambda U}^{-1}$ 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 $\mathtt{A}$ and factors it into $\mathtt{A} = \mathtt{Q}_1 \Sigma \mathtt{Q}_2^T$, where

- $\mathtt{Q}_1$ is an $m \times m$ orthogonal matrix whose columns are the eigenvectors of $\mathtt{AA}^T$
- $\mathtt{Q}_2$ is an $n \times n$ orthogonal matrix whose columns are the eigenvectors of $\mathtt{A}^T\mathtt{A}$
- Σ is an $m \times n$ diagonal matrix whose elements are the square roots of the eigenvalues of $\mathtt{AA}^T$ and $\mathtt{A}^T\mathtt{A}$ (both have the same eigenvalues, but different eigenvectors)

This is better shown graphically:

$$\overset{\mathtt{A}}{\begin{bmatrix} & & & & \cdot \\ & & & & m \\ \cdot & \cdot & n & \cdot & \cdot \end{bmatrix}} = \overset{\mathtt{Q}_1}{\begin{bmatrix} & & \cdot \\ & & m \\ \cdot & m & \cdot \end{bmatrix}} \overset{\Sigma}{\begin{bmatrix} & & & & \cdot \\ & & & & m \\ \cdot & \cdot & n & \cdot & \cdot \end{bmatrix}} \overset{\mathtt{Q}_2^T}{\begin{bmatrix} & & & & \cdot \\ & & & & \cdot \\ & & & & n \\ & & & & \cdot \\ \cdot & \cdot & n & \cdot & \cdot \end{bmatrix}}$$

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:

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

Being part of a diagonal matrix, these only multiply particular columns of $\mathtt{Q}_1$ or $\mathtt{Q}_2$.[35] The size of the singular value tells you exactly how much influence the corresponding rows and columns of $\mathtt{Q}_1$ and $\mathtt{Q}_2^T$ 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 $\mathtt{A}$ 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 nullspace

[35] The singular values Σ of course multiply the *rows* of $\mathtt{Q}_2^T$, and hence the columns of the un-transposed version, $\mathtt{Q}_2$.

of the matrix (the solution to $\mathtt{A}\mathbf{x} = \mathbf{0}$) is simply the columns of $\mathtt{Q}_2$ which correspond 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 $\mathtt{Q}_1$ and $\mathtt{Q}_2$ are therefore also arranged in importance. What this means is that, while we can find the exact value of A by multiplying $\mathtt{Q}_1\Sigma\mathtt{Q}_2^T$, if we removed (for example) the last columns of $\mathtt{Q}_1$ and $\mathtt{Q}_2$, 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:

$$\overset{\texttt{A}}{\begin{bmatrix} & \cdot \\ & \cdot \\ & m \\ & \cdot \\ \cdot \quad n & \cdot \end{bmatrix}} = \overset{\texttt{Q}_1}{\begin{bmatrix} & \cdot \\ & \cdot \\ & m \\ & \cdot \\ \cdot \quad n & \cdot \end{bmatrix}} \overset{\Sigma}{\begin{bmatrix} & \cdot \\ & n \\ \cdot \quad n & \cdot \end{bmatrix}} \overset{\texttt{Q}_2^T}{\begin{bmatrix} & \cdot \\ & n \\ \cdot \quad n & \cdot \end{bmatrix}}$$

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.

Summary

After reading through Sections 13 of the tutorial guide and working through the examples you should be able to:

- ☐ Calculate eigenvectors and eigenvalues using OCTAVE
- ☐ Have an understanding of what the Singular Value Decomposition (SVD) tells you about a matrix
- ☐ Be able to perform the SVD in OCTAVE

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 + 0.60000
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.

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.

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

Table 9: Complex number functions

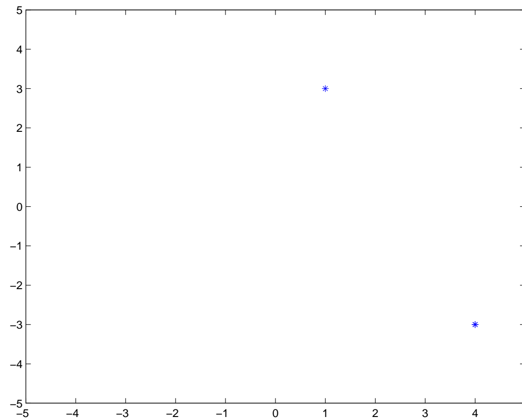


Figure 13: Graphical representation of the complex numbers $z_1 = 4 - 3i$ and $z_2 = 1 + 3i$ using the `plot` command.

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:

```
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 addtion 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