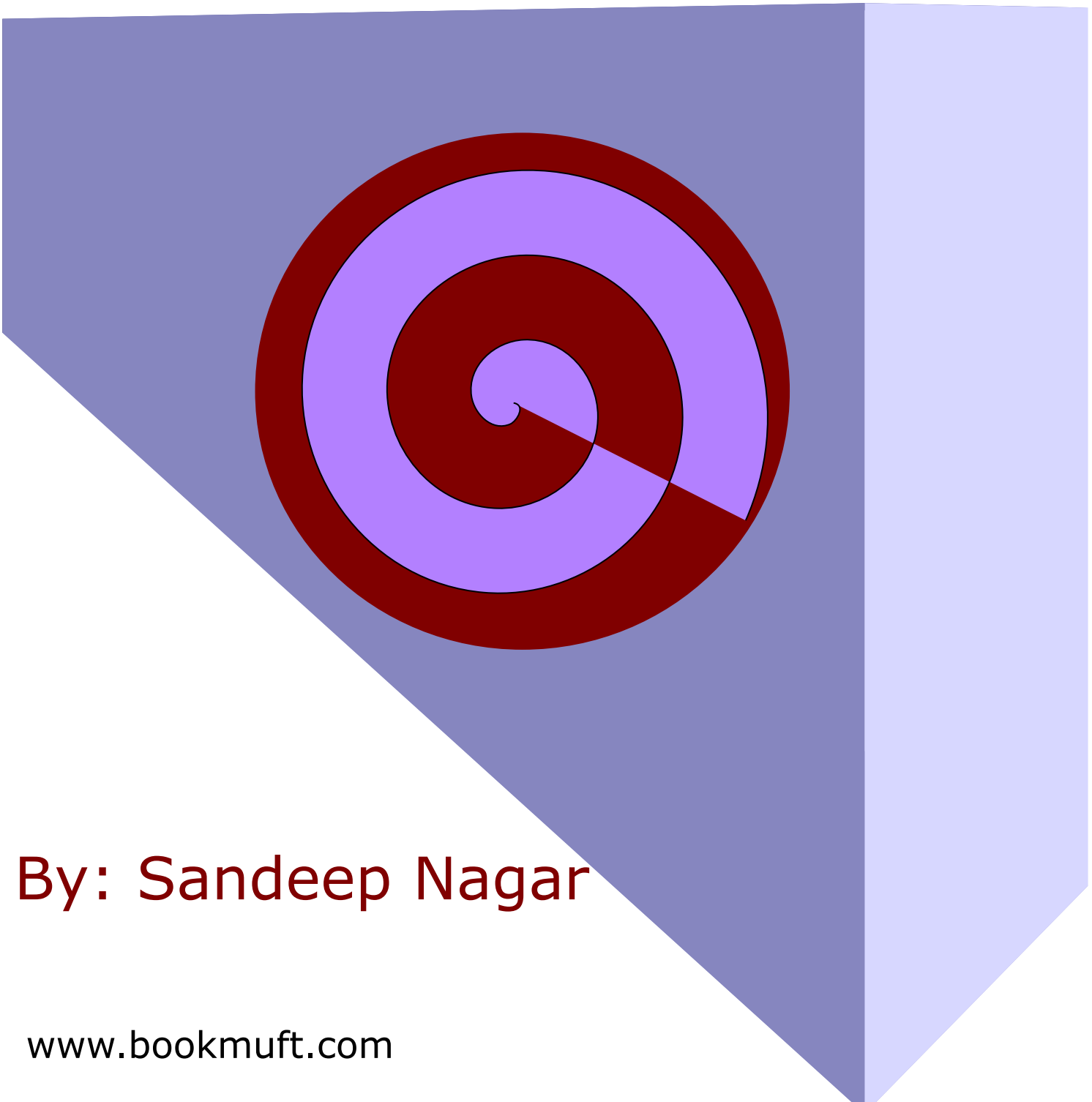
INTRODUCTION TO OCTAVE

For Scientists and Engineers



By: Sandeep Nagar

www.bookmuft.com

Introduction to Octave

For Scientists and Engineers

Dr. Sandeep Nagar

Tuesday 8th March, 2016

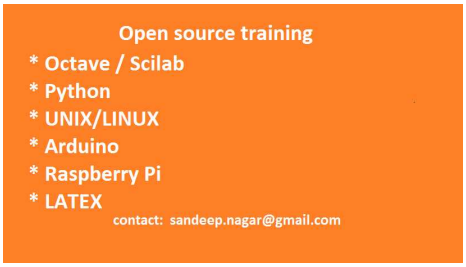
Contents

License information

Dedicated to two beautiful ladies in
my life, Rashmi and Aliya

1 Introduction to Octave

1.1 Introduction to numerical computing



With the advent of computers in post world war II era, desire to simulate all physical problems led to the invention of numerical computing. Whereas analytical computation required only pen, paper and human mind, numerical computation required a calculating device, or a computer. Successful implementation of computing device to solve problems (especially involving repeated tasks) over large array of data points was observed in many fields of science and engineering. For example, breaking enemy codes, simulating nuclear reactions before nuclear explosions etc.

As time progressed, various schemes to define analytical functions like differentiation, integration, trigonometric etc. were written for digital computers. This involved their digitization, which certainly introduces some errors. Knowledge of error introduced and its proper nullification could yielded valuable information quicker than analytical results. Thus it became one of the most actively researched field of science and continues to be the one till date. Search for faster and accurate algorithms continues to drive innovation in the field of numerical computing and enables humanity

to simulate otherwise impossible tasks.

1.2 Various alternatives

A number of alternatives exist to perform numerical computation. Programming languages written to handle mathematical functions like FORTRAN, C, python, Java to name a few, can be used to write algorithms for numerical computation. Specialized softwares like $MATLAB^{®}$, Scilab, Mathematica also exists to provide specialized packages for a particular field of problems. Their rich libraries now run in many GBs of data. Amongst them, $MATLAB^{®}$ became tremendously popular among scientific community since 1984. Cheap availability of digital computing resources propelled its use in industry and academia to such an extent that virtually every lab needed $MATLAB^{®}$. Whereas it wasn't very expensive for west, it proved to a costly piece of software for rest of the world, particularly third world countries. This part of world, which has otherwise a rich pool of scientific community, needed an open sourced alternative to $MATLAB^{®}$. Thus Octave and Scilab came into existence. Whereas Scilab is extrmemly powerful, it was not compatibale with $MATLAB^{®}$ syntax-wise. On the other hand Octave was developed so that `.m` files could directly run on octave.

1.3 $MATLAB^{®}$ and Octave

Octave is a opensource alternative which can run MATLAB$^{®}$ code. So existing MATLAB$^{®}$ users can swiftly change to this new system. Also new users can learn to code in octave and then shift to MATLAB$^{®}$ environment as and when required. GNU Octave, version 3.8.1 comes with a Graphic User Interface (GUI), hence it has been chosen for the present book. Older and future versions will also run well for the codes presented in the book, provided future versions choose to remain compatible with present version.

Other alternatives include softwares like Scilab and programming languages like python, C, C++, Java etc. They have their own merits and demerits and hence reader is advised to judge their choice based on their needs. Octave is a good choice to prototyping the problem quickly and checking the results. Other alternatives prove better while working with web-based data collection, analysis and visualization. Octave is a high-level language, primarily intended for numerical computations. Octave has a rich library of tools for solving numerical linear algebra problems, finding the

roots of non-linear equations, integrating ordinary functions, manipulating polynomials, and solving ordinary and partial differential and differential-algebraic equations. This makes it suitable for most of the basic numerical computational work.

1.4 Installation

Please note that following instructions are valid for Octave, version 3.8.1 only. GNU octave can be downloaded from the url `https://www.gnu.org/software/octave/download.html` as per the operating system. Installation is quite straight forward and user forums or simple google search yields useful answers to common problems encountered by users. As explained earlier version 3.8.1 comes with a GUI, hence it is advised that this should be installed for forthcoming discussions, but all older version will prove to be equally good.

1.5 Workspace

There are two ways to work within octave. First one is to work at command line by writing one command at a time. Second method is to write a script (a `.m` file having a set of commands in a sequence) and running it from the command line by simply writing its name. For example to run `a.m` script file, one simply writes at command prompt:

```
>>a
```

Octave command prompt is represented by the symbol "`>>`" by default. After entering a command at the command prompt, if enter key is pressed on keyboard, the command is executed.

1.5.1 Calculator

In simplest view, octave works as a calculator with mathematical operators like multiplication (symbol is `*`), division (symbol is `/`), addition (symbol is `+`), substraction (symbol is `-`) and exponentiation(symbol is `^`):

```
>> 3 + 5
ans =  8
```

```
>> 2 - 3
ans = -1
>> 3.0 * 5
ans =  15
>> 2 / 3
ans =  0.66667
>> format long
>> 2 / 3
ans =  0.666666666666667
>> format short
>> 2 / 3
ans =  0.66667
>> 2 % 3
ans =  2
>> 2 ^ 3
ans =  8
```

As seen in example above, when command is fed at the command prompt `>>` it is executed and answer is given by displaying the results in next line as `ans =`. To display more numbers in the result, `format long` command is used, whereas by default, octave works with the command `format short`.

1.5.2 Predefined constants

```
>> pi
ans =  3.1416
>> e
ans =  2.7183
>> i
ans =  0 + 1i
>> j
ans =  0 + 1i
>> Inf/Inf
ans = NaN
```

A number of physical constants are per-defined: `pi`, `e`(Euler's number), `i` and `j` (the imaginary number $\sqrt{-1}$), `inf` (Infinity), `NaN` (Not a Number - resulting from undefined operations, such as `Inf/Inf`.)

1.5.3 Common mathematical functions

```
>>  abs(-10.034)
```

```
ans =  10.034
>> log(e)
ans =  1
>> log10(10)
ans =  1
>> sin(10)
ans = -0.54402
>> cos(10)
ans = -0.83907
>> tan(10)
ans =  0.64836
>> asin(1)
ans =  1.5708
>> asin(10)
ans =  1.5708 + 2.9932i
>> acos(1)
ans = 0
>> acos(10)
ans =  0.00000 - 2.99322i
>> atan(1)
ans =  0.78540
>> atan(10)
ans =  1.4711
```

A number of predefined mathematical functions exists in octave like:

- **absolute value**: `abs()`
- **Logarithm**: Natural logarithm `log()`, Base 10 logarithm `log10()`
- **trigonometric functions**" `sin()`, `cos()`, `tan()`

  Arguments are taken in radians
- **inverse-trigonometric functions**" `asin()`, `acos()`, `atan()`

When we work on command prompt, it is often convenient to have a clear screen by getting rid of previous command written at command prompt. this is done by the command `clc` which *clears screen* by removing all inputs and outputs from the user screen.

Complex calculations using these functions and operations can be performed with ease like

$$\sqrt{sin(10)^2 + cos(10)^2}$$

and

$$\frac{sin(10)}{\sqrt{cos(10)}}$$

```
>> sqrt(((sin(10))^2)+(cos(10))^2)
ans =  1
>> sin(10)/sqrt(cos(10))
ans =  0.00000 + 0.59390i
```

1.6 help

Covering all the functions available with octave is beyond the scope of the present book. To understand how a particular function needs to be used, one can use `help()` command where argument can be the function whose usage needs to be found out. For example `help(exp)` gives a detailed view about how this function should be used.

```
>> help exp
'exp' is a built-in function from the file libinterp/corefcn/
    mappers.cc

-- Mapping Function: exp (X)
Compute 'e^x' for each element of X.  To compute the matrix
exponential, see *note Linear Algebra::.

See also: log.


Additional help for built-in functions and operators is
available in the online version of the manual.  Use the command
'doc <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.org
mailing list.
```

1.7 Variable

To store values temporarily, we use variables which store the value at a particular memory location and address it with a symbol or set of symbols

(called *strings*). For example: one can store the value of $1/10 * \texttt{pi}$ as a variable `a` and then use it in an equation like

$$a^2 + 10\sqrt{a}$$

```
>> a=1/10*pi
a =  0.31416
>> a^2 + 10* sqrt(a)
ans =  5.7037
```

Hence the symbol `=` works as an assignment operator, which assigns the value present on right hand to the variable name at left hand side.

Multiple assignments can be performed by using comma (`,`) operator. Also if we do not wish to produce results on screen, we can suppress this by using `;` operator.

```
>> a1 = 1, a2 = 10, a3 = 100
a1 =  1
a2 =  10
a3 =  100
>> a1 = 1, a2 = 10, a3 = 100;
a1 =  1
a2 =  10
>> a1 = 1; a2 = 10; a3 = 100;
>> a1
a1 =  1
>> a2
a2 =  10
>> a3
a3 =  100
```

1.7.1 Data types

While assigning data to a variable it is important to understand that data can be defined as a variety of *object* defined by its *datatype* as follows:

- **logical**: This type of data stores boolean values `1` or `0` boolean values and can be operated by boolean operators like AND, OR, XOR etc.
- **char**: This type of data stores alphabetic characters and strings (group of characters written in a sequence).

- **int8,int16,int32,int64**: This type of data is stored as integers within 8 bits, 10 bits, 32 bits and 64 bits respectively. Size of integer is given by its bit counts.

  Both `logical` and `char` are 1 byte (8 bits) wide.

- **uint8, uint16, uint32, uint64**: This type of data stores unsigned integer data in 8, 16, 32 and 64 bits respectively.

- **double, single** This type of data is stored as double and single precision floating type respectively. Decimal numbers are represented by floating point data types. Single precision occupies 4 bytes (32 bits) and double precision occupies (64 bits) to store the floating point numbers.

  In single precision system, 23 bits stores the fraction bits (i.e numbers after the decimal point), 8 bits stores the exponent (i.e the numbers before the decimal point) and 32^{nd} bit is reserved for storing the sign.

  In double precision system, 52 bits stores the fraction bits (i.e numbers after the decimal point), 11 bits stores the exponent (i.e the numbers before the decimal point) and 64^{th} bit is reserved for storing the sign.

  Single and double precision matters when precision of result matters. In cases like GPS position for a projectile flying at high speeds, it will be required that the results should be as precise as possible for greater accuracy of hit.

- **double complex,single complex**: Complex numbers have real and imaginary parts which are stored separately. These numbers can be stored as single or double precision numbers using these data types.

1.7.2 Naming conventions for variables

There are some naming conventions for variables names, which must be respected to avoid errors.

- Names should not start with a number however numbers can be used anywhere afterwards.
- Variable names are case sensitive
- *Keywords* cannot be used as names.

- Names can include underscore (_)

While naming a variable, if one needs to check that the name given is a keyword or not, then one can use a built-in function `iskeyword(name)`. Simply writing `iskeyword()` produces a list of keywords as shown below:

```
>> iskeyword()
ans =
{
[1,1] = __FILE__
[2,1] = __LINE__
[3,1] = break
[4,1] = case
[5,1] = catch
[6,1] = classdef
[7,1] = continue
[8,1] = do
[9,1] = else
[10,1] = elseif
[11,1] = end
[12,1] = end_try_catch
[13,1] = end_unwind_protect
[14,1] = endclassdef
[15,1] = endenumeration
[16,1] = endevents
[17,1] = endfor
[18,1] = endfunction
[19,1] = endif
[20,1] = endmethods
[21,1] = endparfor
[22,1] = endproperties
[23,1] = endswitch
[24,1] = endwhile
[25,1] = enumeration
[26,1] = events
[27,1] = for
[28,1] = function
[29,1] = global
[30,1] = if
[31,1] = methods
[32,1] = otherwise
[33,1] = parfor
[34,1] = persistent
[35,1] = properties
[36,1] = return
[37,1] = static
[38,1] = switch
[39,1] = try
[40,1] = until
```

```
[41,1] = unwind_protect
[42,1] = unwind_protect_cleanup
[43,1] = while
```

1.7.3 List of variables

List of all variables can be obtained by the commands `who` and `whos` where `who` simply presents the list of variables in the workspace whereas `whos` presents the same with more details like size of variable, number of bytes used to store the variable and variable type.

```
>> who
Variables in the current scope:

a    a1   a2   a3   ans

>> whos
Variables in the current scope:

Attr Name        Size                     Bytes  Class
==== ====        ====                     =====  =====
a               1x1                          8  double
a1              1x1                          8  double
a2              1x1                          8  double
a3              1x1                          8  double
ans             1x1                          8  double

Total is 5 elements using 40 bytes
```

By using `who` and `whos` one can keep track of memory requirements judicious use of memory resources are important such as Raspberry Pi based systems.

1.7.4 Global and Local Variables

A variable declared globally i.e. within the main program is known as global variable whereas a variable declared locally within a function is known as local variable. Using `global` declaration statement. Once defined, it remains same irrespective of any new definition unless `clear` command is issued for *clearing* variable names and values from the memory.

```
>> global a =1
>> global a = 2
>> a
a =  1
>> clear
>> who
>> whos
>> a=1
a =  1
>> a=2
a =  2
>>
```

As seen above, `a = 1` stays same irrespective of next definition `a = 2`. When the command `clear` is issued at command prompt, all variable anmes and values are flushed out of memory and the variable name can be sued again. This time, if it is not defined as `global` variable, then its value can be changed repeatedly. The command `isglobal()` lets one check if a variable name has been defined as global variable.

Global variables are used to define constants during numerical calculations. Suppose we wish that all variable except some should change values, then we name those unchanging values to be global variables by giving the name of our choice. The predefined variables like `pi, e` etc. have been defined in a similar manner.

1.7.5 clear

As seen in previous section, `clear` command flushes out variable names and their values from the memory. It proves to be much more useful than that. Whereas `clear all` is same as `clear`, it can also be used to selectively wipe out variables and their values. Simply type `help clear` gives a detailed view of its use:

```
>> help clear
'clear' is a built-in function from the file libinterp/corefcn/
    variables.cc

-- Command: clear [options] pattern ...
Delete the names matching the given patterns from the symbol
    table.
The pattern may contain the following special characters:

```

```
8  '?'
9  Match any single character.
10
11 '*'
12 Match zero or more characters.
13
14 '[ LIST ]'
15 Match the list of characters specified by LIST.  If the first
16 character is '!' or '^', match all characters except those
17 specified by LIST.  For example, the pattern '[a-zA-Z]' will
18 match all lowercase and uppercase alphabetic characters.
19
20 For example, the command
21
22 clear foo b*r
23
24 clears the name 'foo' and all names that begin with the letter '
       b'
25 and end with the letter 'r'.
26
27 If 'clear' is called without any arguments, all user-defined
28 variables (local and global) are cleared from the symbol table.
        If
29 'clear' is called with at least one argument, only the visible
30 names matching the arguments are cleared.  For example, suppose
       you
31 have defined a function 'foo', and then hidden it by performing
       the
32 assignment 'foo = 2'.  Executing the command 'clear foo' once
       will
33 clear the variable definition and restore the definition of 'foo
       '
34 as a function.  Executing 'clear foo' a second time will clear
       the
35 function definition.
36
37 The following options are available in both long and short form
38
39 '-all, -a'Clears all local and global user-defined variables and
        all
40 functions from the symbol table.
41
42 '-exclusive, -x'
43 Clears the variables that don't match the following pattern.
44
45 '-functions, -f'
46 Clears the function names and the built-in symbols names.
47
48 '-global, -g'
```

```
Clears the global symbol names.

'-variables, -v'
Clears the local variable names.

'-classes, -c'
Clears the class structure table and clears all objects.

'-regexp, -r'
The arguments are treated as regular expressions as any
variables that match will be cleared.

With the exception of 'exclusive', all long options can be used
without the dash as well.


Additional help for built-in functions and operators is
available in the online version of the manual.  Use the command
'doc <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.org
mailing list.
>>
```

Judicious use of `clear` command proves to be a very powerful tool in managing memory requirements for a memory intensive numerical calculation.

1.8 Summary

Using octave as a simple calculator (using numbers and basic operations) as well as a complex calculator (using variables with complex functions), one can perform numerical calculation at ease. Learning curve for octave is quite flat owing to is simple and intuitive syntax. In case of confusion, documentation for particular commands can be easily available using `help` command. Octave also provides an integrated environment for working with a lot of different kinds of computational tasks.

2 Working with Arrays

2.1 Introduction

Matrices have become integrated part of numerical computation while dealing with large quantity of data. For a 2 dimensional matrix, elements have unique row and column index through which one can access them. Rows and Columns can be attributed to different properties under study. In this way, one can fit data for two properties as a matrix and then use these matrices for numerical calculations. For example, suppose element of a row is defined as 1 if a compound is conductor and 2 if it is a semiconductor and 3 if it is an insulator. Then a row vector (a matrix composed of only one row) `[1 0 0 3 2 1 3 0 1 0 3 2 1]` has information about 13 compounds. In a numerical conductivity involving conductive nature of a compound, this row vector (a 13×1 matrix) can be utilized.

Octave has a class of objects for dealing with matrices. They are called arrays. Using different properties of this class, one can define various kinds of matrices. Built-in functions for matrix operations make it easier for a programmer to deal with large number of data by arranging them as a matrix in the desired format and performing array operations.

2.2 Arrays and vectors

Instead of just pointing to a single number, a variable name can also point to a sequential set of numbers called an `array`.

```
>> a = [1 , 2, 3, 4, 5]
a =

1   2   3   4   5

>> a1 = [10, 11, 12, 13, 14]
a1 =

10   11   12   13   14
>> matrix22 = [1 , 2 ; 3 , 4]
matrix22 =

1   2
3   4
>> matrix33 = [1, 2, 3; 4, 5, 6; 7, 8, 9]
matrix33 =

1   2   3
4   5   6
7   8   9
>> size(a)
ans =

1   5

>> size(matrix22)
ans =

2   2

>> size(matrix33)
ans =

3   3
```

As seen in the example code, an array can be understood as a matrix consisting of rows and columns. Thus one can make a desired sized matrix for example, `matrix22` is a $2X2$ and `matrix33` is a $3X3$ matrix where as `a` is a $1X5$ matrix. The first number while defining the size gives the number of rows while the second number gives the number of columns. The comma `(,)` operator operates by defining the *next element* in the same row whereas

the (;) operator defines the numbers in the next line/row.

If the number of elements in each row/column do not match then one obtains an error message:

```
>> right33 = [1, 2, 3; 4, 5, 6; 7, 8, 9]
right33 =

1   2   3
4   5   6
7   8   9

>>wrong33 = [ 2, 3; 4, 5, 6; 7, 8, 9]
error: vertical dimensions mismatch (1x2 vs 1x3)
>> wrong33 = [ 1, 2, 3; 4, 5, 6; 8, 9]
error: vertical dimensions mismatch (2x3 vs 1x2)
```

Elements of an array can be any data-type as defined in section 1.7.1. All elements of an array can be set to a particular data-type by the commands as shown below:

```
>> x = uint32 ([1, 65535])
x =

1  65535

>> x = uint64 ([1, 65535])
x =

1  65535

>> x = int16 ([1, 65535])
x =

1  32767

>> x = int32 ([1, 65535])
x =

1  65535

>> x = int64 ([1, 65535])
x =

1  65535

>> x = float ([1, 65535])
```

```
27 error: 'float' undefined near line 1 column 5
28 >> x = single ([1, 65535])
29 x =
30
31 1   65535
32
33 >> x = double ([1, 65535])
34 x =
35
36 1   65535
37
38 >> x = single ([1.0, 65535e10])
39 x =
40
41 1.0000e+00   6.5535e+14
42
43 >> x = double ([1.0, 65535e10])
44 x =
45
46 1.0000e+00   6.5535e+14
```

Line number 14 shows that if the element is set to `int16` then it can store a maximum value of 32767 irrespective of being commanded to store a value bigger than that. Hence it becomes supremely important to understand the data-type of the elements beforehand, to avoid errors in numerical calculations. Also storing very small numbers in larger number of bits is a waste of memory system (line number 46 displays that the number 1 is stored as a double precision floating point number which occupies 64 bits where essentially 63 bits except the last one are all zeros).

2.3 Operations on arrays and vectors

Operating on arrays has two aspects:

- Operating on two or more arrays
- Element wise operations

All arithmetic operators like `+,-,*,/, %,^` etc. can be used in both cases. When we need to do element wise operation, then a `.` is placed before operator so that element-wise operators become `.+,.-,.*,./, .%,.^`. This will become more clear in following example.

```
>> a = [1, 2; 3, 4]
a =

1    2
3    4

>> b = [5, 6; 7, 8]
b =

5    6
7    8

>> a+b
ans =

6     8
10    12

>> 2.+a
ans =

3    4
5    6

>> -10.+b
ans =

-5   -4
-3   -2
```

When `a` and `b` are matrices to be added/subtracted, then their elements are added/subtracted with elements in the same position. For this reason, size of the two matrices added or subtracted should be same.

When we write `2.+a` then we add 2 to each of the element individually. This can be done irrespective of the size and is implemented uniformly on all the elements of the matrix.

Those who are familiar with matrix algebra, know that matrix multiplication and division is not a straightforward task. A `aXb` matrix can only be multiplied by a `bXc` matrix which results in `aXc` matrix and it is performed by multiplying elements of rows with elements of columns to get new elements.

```
>> a = [1, 2; 3, 4; 5,6]
a =

1    2
3    4
5    6
>> a'
ans =

1    3    5
2    4    6

>> a*a'
ans =

5    11    17
11    25    39
17    39    61
>> a/b
ans =

0.050000    0.050000    0.050000
0.116667    0.116667    0.116667
0.183333    0.183333    0.183333
```

`a'` gives the transpose of a matrix (rows are made columns and vice versa).

Performing division of a matrix involves the *matrix inversion*.

```
>> pinv(a)
ans =

-1.33333   -0.33333    0.66667
1.08333    0.33333   -0.41667

>> pinv(b)
ans =

0.016667    0.016667    0.016667
0.016667    0.016667    0.016667
```

Inverse of a matrix a, denoted by a_{-1}, is a matrix such that

$$a * a^{-1} = I$$

where I is identity matrix.

```
>> a
a =

1    2
3    4
5    6
>> pinv(a)
ans =

-1.33333   -0.33333    0.66667
1.08333    0.33333   -0.41667
>> pinv(a)*a
ans =

1.00000    0.00000
-0.00000    1.00000
>> det(pinv(a)*a)
ans =  1
```

I is called an identity matrix because all its diagonal elements are 1 and all non-diagonal elements are zero, which makes its determinant 1. Determinant of a matrix a is calculated by the command `det(a)`.

Automatic generation of an identity matrix is done by using the command `eye(a,b)` where `a` and `b` are values of number of rows and columns.

```
>> eye(2,2)
ans =

Diagonal Matrix

1    0
0    1
>> det(eye(2,2))
ans =  1
>> eye(4,5)
ans =

Diagonal Matrix

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

2.3.1 Random matrix

Using random number generators, a random matrix can be created by the command `rand(a,b)`

```
>> rand(4,5)
ans =

0.779821    0.904132    0.025018    0.118232    0.823903
0.963702    0.393643    0.148051    0.832420    0.316977
0.149530    0.943838    0.872814    0.699306    0.509816
0.133360    0.115337    0.401372    0.067246    0.264232
```

Please note that the numbers generated above will be different each time even on same machine since they are supposed to be random in nature. By default, they are uniformly distributed over the interval $(0, 1)$. A vector is simply a row vector so it can be generated randomly by command `rand(a)`. `help rand` gives detailed description about various other features and arguments of random number generator.

2.3.2 Indexing

Each element of the matrix is characterized by two numbers, the row number and the column number. This is used to pinpoint an element and operate on that.

```
>> a = rand(2,3)
a =

0.5248873    0.5531882    0.0051345
0.1597312    0.3685503    0.3041072

>> a(2,3)=1
a =

0.5248873    0.5531882    0.0051345
0.1597312    0.3685503    1.0000000

>> a(1,1)=0
a =

```

```
0.00000   0.55319   0.00513
0.15973   0.36855   1.00000
```

Please note that `a(2,3)=1` sets the element at 2^{nd} row and 3^{rd} column i.e. number 0.3041072 to 1 and `a(1,1)=0`sets the element at 1^{st} row and 1^{st} column i.e. number 0.5248873 to 0. To index numbers in a vector, one needs a single number.

```
>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
a =

1   2   3   4   5   6   7   8   9

>> a(1)
ans =  1
>> a(-1)
error: subscript indices must be either positive integers less
    than 2^31 or logicals
>> a(5)
ans =  5
>> a(10)
error: A(I): index out of bounds; value 10 out of bound 9
>>
```

It is important to note that unlike some programming languages, where indices start from 0, in octave indices start from 1 and it does not take negative numbers as indices.

2.3.3 Using indices to make new vector

```
>> a = [10 20 30 40 50 60]
a =

10   20   30   40   50   60

>> b = a([1 3 6 1])
b =

10   30   60   10
```

In the above example, `b` is a new vector formed from vector `a` where successive elements are made up of elements taken from a index vector

`[1 3 6 1]`.

```
>> a = [11, 12, 13; 40, 50, 60; 17, 18, 19]
a =

11    12    13
40    50    60
17    18    19

>> a([1, 2], [2, 3])
ans =

12    13
50    60
```

Please note that since use of comma operator is optional, so henceforth we will define vectors and matrices by simply putting a whitespace.

2.3.4 Slicing

Matrices can be sliced to desired portions by using indices and colon : operator.

```
>> a = [1 2 3 4 1 3 2 4 6 4 5]
a =

1    2    3    4    1    3    2    4    6    4    5

>> b =a(1:5)
b =

1    2    3    4    1

>> c = a(5:7)
c =

1    3    2
```

2.4 Automatic generation of vectors

One can generate a series of numbers and store them as arrays by using the command

`start:step:stop`

```
>> a=1:1:10
a =

1     2     3     4     5     6     7     8     9    10

>> a=[1:1:10]
a =

1     2     3     4     5     6     7     8     9    10
```

Please note that `[]` are optional here. If step is not defined then it is taken as 1.

```
>> a=1:10
a =

1     2     3     4     5     6     7     8     9    10

>> a=1:2:10
a =

1     3     5     7     9
```

2.4.1 Linearly spaced vector

The command `linspace(start,stop,n)` produces an array starting from first number and stopping at second one with a total of n numbers. Hence they are linearly spaced.

```
>> a = linspace(1,2,5)
a =

1.0000    1.2500    1.5000    1.7500    2.0000

>> a = linspace(1,2,10)
a =

1.0000    1.1111    1.2222    1.3333    1.4444    1.5556    1.6667
    1.7778    1.8889    2.0000
```

2.4.2 logspace

Similar to `linspace` `logspace(start, stop,n)` produces n number from start to stop which are linearly space in logarithmic nature.

```
>> logspace(1,10,5)
ans =

1.0000e+01    1.7783e+03    3.1623e+05    5.6234e+07    1.0000e+10
```

2.5 Matrix manipulations

Some common matrix manipulations have already been written in function form which makes it easier for developer to use them right away, rather than invest time to write an optimum code.

2.5.1 Flipping a matrix

`flipud(A)` returns a copy of matrix A with the order of the rows reversed. `flipud` stands for *flip-up-down*. `fliplr(A)` returns a copy of matrix A with the order of the rows reversed. `fliplr` stands for *flip left right*.

```
>> a = [1  2; 3  4; 5  6]
a =

1    2
3    4
5    6

>> fliplr(a)
ans =

2    1
4    3
6    5

>> flipud(a)
ans =

5    6
3    4
1    2
```

2.5.2 Rotating a matrix

Using the command `rot90(a,n)` a matrix `a` can be rotated `n` times by 90 degrees

```
>> a = [1  2; 3  4; 5  6]
a =

1    2
3    4
5    6

>> rot90(a,1)
ans =

2    4    6
1    3    5

>> rot90(a,2)
ans =

6    5
4    3
2    1

>> rot90(a,4)
ans =

1    2
3    4
5    6
```

2.5.3 Reshaping a matrix

Number of rows and columns can be changed provided total number of elements remains same.

```
>> a = [1  2; 3  4; 5  6]
a =

1    2
3    4
5    6

>> reshape(a,6,1)
ans =
```

```
10
11 1
12 3
13 5
14 2
15 4
16 6
17 >> reshape(a,4,1)
18 error: reshape: can't reshape 3x2 array to 4x1 array
```

2.5.4 Sorting

Numbers can be sorted in increasing order using `sort` function:

```
1 >> a = rand(1,5)
2 a =
3
4 0.577290   0.079980   0.880757   0.294744   0.964269
5
6 >> sort(a)
7 ans =
8
9 0.079980   0.294744   0.577290   0.880757   0.964269
```

2.6 Special matrices

Matrix algebra defines some kinds of matrices which are special in nature and find their use in some problems. Octave has some functions defined to create these matrices.

2.6.1 Upper and Lower triangular matrix

Upper triangular matrix is such that only diagonal and elements above diagonal are non-zero. Similarly, lower triangular matrix is such that diagonal and elements below diagonal are non-zero.

```
1 >> a = rand(3,3)
2 a =
3
4 0.414936   0.399589   0.269880
5 0.070691   0.405602   0.378955
```

```
0.169398    0.850042    0.919782

>> tril(a)
ans =

0.41494    0.00000    0.00000
0.07069    0.40560    0.00000
0.16940    0.85004    0.91978

>> triu(a)
ans =

0.41494    0.39959    0.26988
0.00000    0.40560    0.37896
0.00000    0.00000    0.91978
```

2.6.2 Ones and zeros matrix

A matrix having all its numbers as 1 or 0 make up ones and zeros matrix respectively:

```
>> ones(3,3)
ans =

1    1    1
1    1    1
1    1    1

>> zeros(3,3)
ans =

0    0    0
0    0    0
0    0    0
```

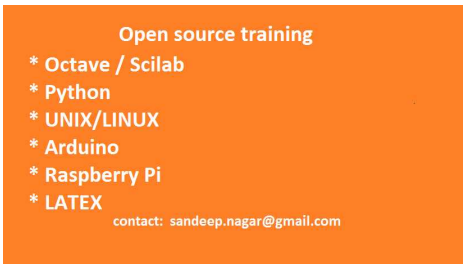
2.7 Summary

Array based computing lies at the very heart of modern computational techniques. Octave presents a very suitable platform to perform this technique with ease. A variety of predefined functions enable user to save time while prototyping a problem. Flexible methods to define multidimensional arrays

and performing fast computation is the main necessity of our times. Most of the time spent during a simulation is either in loops or in array operations. Predefined array operations have been optimized with algorithms for reliability, time saving and efficient memory management.

3 Plotting

3.1 Introduction



Without visualization, numerical computations are difficult to judge. Producing publication quality images of complex plots which give a meaningful analysis of numerical results, has been a challenge for scientists all over the world. Many commercial softwares made good business satisfying this need. Octave also provides this facility. Plotting features includes choosing from various types of plots in 2D and 3D regime, decorating plots with additional information like titles, labeled axes, grids, label for data and writing equations and other important information about data etc. Following sections will describe these actions in detail. It is worth mentioning that plotting capabilities are essential to machine learning experiments since visual directions from the progressive steps give intuitive understanding of the problem under consideration.

3.1.1 2D plotting

plot(x,y)

Since we need data on two axes to be plotted, we first need to create them. Lets assume that x axis has 100 linearly space data points on which $y = x^2$.

```
>>x = linspace(0,100,100);
>> y = x.^2
>> plot(x,y)
```

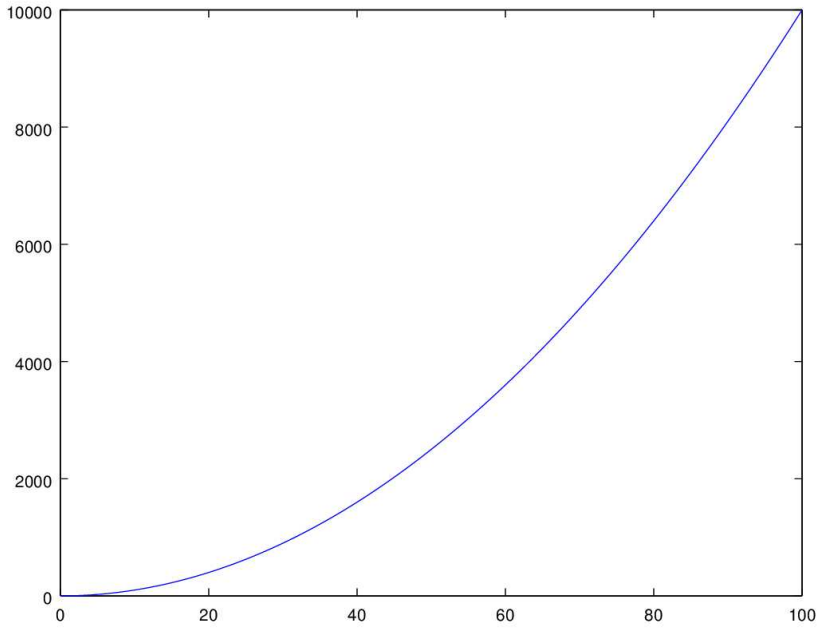


Figure 3.1: $y = x^2$

First we defined a variable x and placed 100 equally spaced data points from 0 to 100. This made a $1x100$ matrix. Using scalar operation of exponentiation, we defined a variable y as x^2. Then we use the function `plot()` which takes two arguments as x-axis and y-axis data points.

Writing `help plot` on the command prompt gives useful insight into this wonderful function written to plot two dimensional data.

polar

Sometimes we prefer to plot in polar coordinates, rather than Cartesian coordinates. Then instead of x, y our coordinates are r, θ.

```
theta = 0:0.02:2*pi;
a1 = 0.5 + 1.3 .^ theta;
a2 = 5 * cos(theta);
a3 = 3 * (1 - cos(theta));
a4 = 6*sin(4*theta);
r = [a1; a2; a3; a4];
PolarGraph = polar(theta, r, "*");
set(PolarGraph, "LineWidth", 2);
legend("spiral", "circle", "heart", "Rose");
```

CoordinatesPolar.m

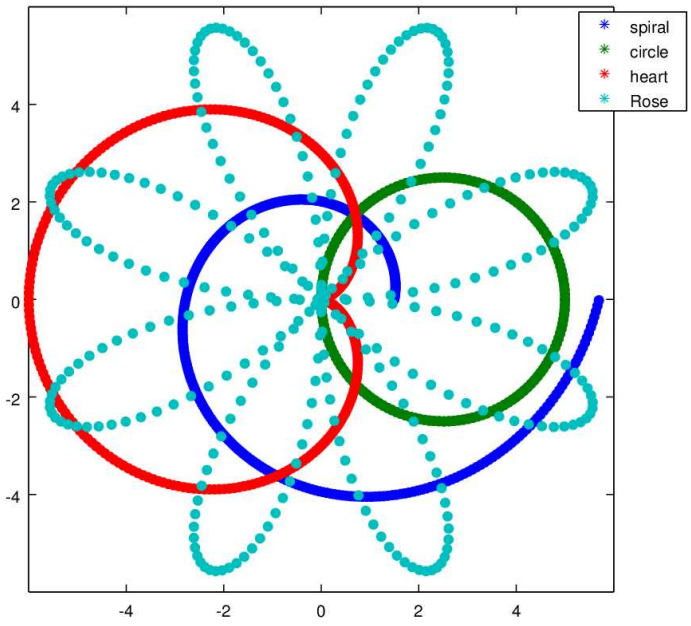


Figure 3.2: Polar Graph

Figure 3.2 gives an example of a polar graph for code given by `CoordinatesPolar.m` example. Explanation of program is given as follows(according to line number):

1. A variable named `th` representing θ is defined by points starting from 0 to 2π with steps of 0.02.

2. A variable named `a1` representing r for **spiral** is calculated by equation

$$r = 1.5(\theta)$$

3. A variable named `a2` representing r for **circle** is calculated by equation

$$r = 5(cos(\theta))$$

4. A variable named `a3` representing r for **heart** is calculated by equation

$$r = 3(1 - cos(\theta))$$

5. A variable named `a4` representing r for **rose** is calculated by equation

$$r = 6(sin(4\theta))$$

6. A variable named `r` stores all the r calculated using equations as a column vector.

7. A variable named `PolarGraph` stores the values produced by the function `polar()` which takes θ, r as arguments and also `"*"` for the type of marker.

8. `set` function is used to set the *property values* for the graph function. This is a neat way of setting properties of the graph and experimenting with them later. In present case, property named `LineWidth` is set to be 2.

9. `legend()` function sets four legends in the same order as the polar function has taken them from the vector `r`

plotting multiple plots is same graph

Multiple plots can be plotted within the same figure by simply supplying x and y axes vectors.

```
clear all;
clf;
x = linspace(1,100,100);
y1 = x.^2.0;
y2 = x.^2.1;
```

```
6  y3 = x.^2.2;
7  y4 = x.^2.3;
8  plot (x, y1, "@12", x, y2, x, y3, "4", x, y4, "+")
9  grid on
10 legend('x^2','x^{2.1}', 'x^{2.2}','x^{2.3}');
11 xlabel('x-axis')
12 ylabel('y-axis')
13 title('Multiple Graphs')
14
15 %plot y with points of type 2 (displayed as '+')
16 %and color 1 (red), y2 with lines, y3 with lines
17 %of color 4 (magenta) and y4 with points displayed as '+'
```

multi.m

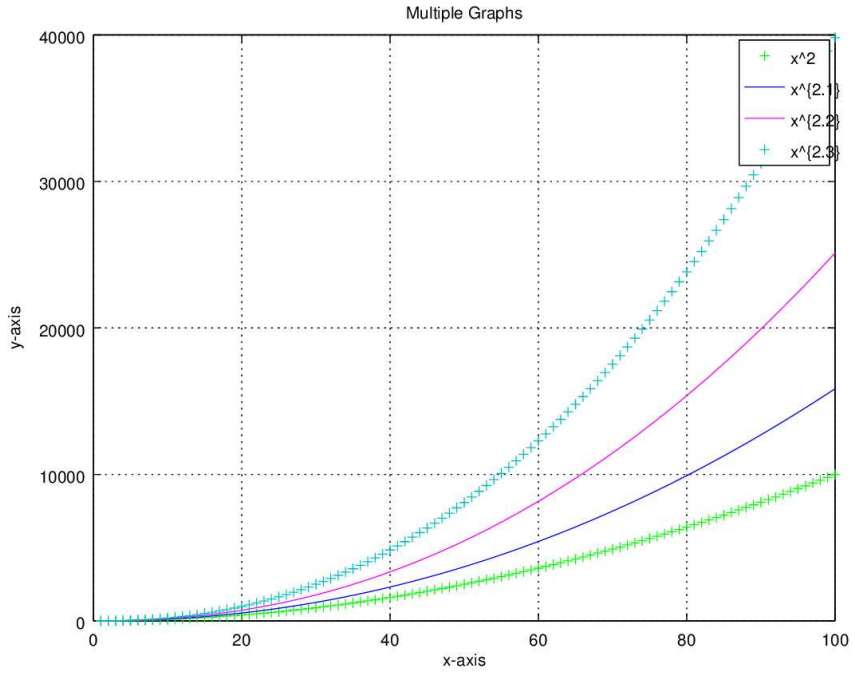


Figure 3.3: Multiple plots within same figure

Explanation of line numbers for above code is as follows:

1. `clear all` clears variable names and values from the memory
2. `clf` clears any current figure window.

3. `x = linspace(1,100,100)` makes a vector `x` made up of 100 equally spaced data points between 1 and 100.
4. $y1 = x.^2.0$; makes a new vector named `y1` having element wise square of vector `x`
5. $y1 = x.^2.1$; makes a new vector named `y2` having element wise exponentiation by 2.1 of vector `x`
6. $y2 = x.^2.2$; makes a new vector named `y3` having element wise exponentiation by 2.2 of vector `x`
7. $y3 = x.^2.3$; makes a new vector named `y3` having element wise exponentiation by 2.3 of vector `x`
8. $y4 = x.^2.4$; makes a new vector named `y4` having element wise exponentiation by 2.4 of vector `x`
9. plots as per comment given in line 15,16,17.
10. grid is turned on for the figure.
11. `xlable` takes the value of string `x-axis`
12. `ylabel` takes the value of string `y-axis`
13. `title` takes the value of string `Multiple Graphs`

Figure 3.3 is obtained by running the code. These types of plots are used to check the variation of result by varying a particular parameter.

Plotting multiple plots separately

`subplot(row,coloumn, index)` command is used to plot multiple plots within the same figure separately. `subplot(2,2,4)` means that plot will be on 2^{nd} row, 2^{nd} column and 4^{th} index.

```
clear all;
clf;
x = linspace(1,100,100);
y1 = x.^2.0;
y2 = log(x);
y3 = sin(x);
y4 = log10(x);
subplot(2,2,1), plot(x, y1)
subplot(2,2,2), plot(x, y2)
```

```
subplot(2,2,3), plot(x, y3)
subplot(2,2,4), plot(x, y4)
%grid on
%legend('x^2','x^{2.1}', 'x^{2.2}','x^{2.3}');
%xlabel('x-axis')
%ylabel('y-axis')
%title('Multiple Graphs')

%plot y with points of type 2 (displayed as '+')
%and color 1 (red), y2 with lines, y3 with lines
%of color 4 (magenta) and y4 with points displayed as '+'
```

multiSubplot.m

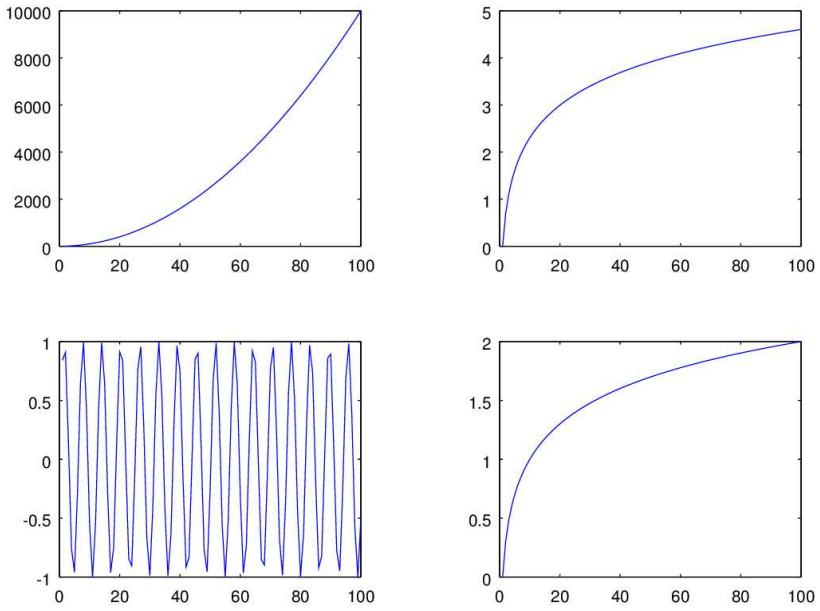


Figure 3.4: Separate Multiple plots within same figure

As seen in figure 3.4, plots are organized as matrix where row number as well as column number dictates its position. Index of the plot can then be used to treat it as an object for further processing on graphical object.

Many commands for controlling the font size, tick labels, fonts, inserting mathematical equations etc. can be known by writing `help plot` or reading the documentation of this function. Ample of examples can be obtained from the web. This function will be used frequently, so its is necessary that one has good command over its use.

3.1.2 3D plots

There are various functions available for 3D plotting in octave. Choosing one of them depends on particular problem.

mesh

```
a = b = linspace (-8, 8, 41)';
[xx, yy] = meshgrid (a, b);
c = sqrt (xx .^ 2 + yy .^ 2) + eps;
d = sin (c) ./ c;
mesh (a, b, d);
```

ThreeDMesh.m

Its important to note that we used a new function named `meshgrid`. Doing a quick search around it, using `help meshgrid` will be very useful. It is used as follows:

```
>> a = b = linspace(-8,8,41);
>> [xx,yy] = meshgrid(a,b);
```

Two variables are created namely `a` and `b` and they store linearly spaced 41 data points between -8 to 8, as a row vector. These two row vectors (both $1X41$ in dimension) are passed as arguments for the fucntion `meshgrid` which gives two outputs: `xx` and `yy`. These are $41X41$ dimensioned matrices where rows of `xx` are copies of `a` and columns of `yy` are copies of `b`. `meshgrid` can also take third argument whose copes make a complete 3D grid. Otherwise on this two dimensional base grid, a function can be defined for data points defined by copies of `a` and `b` vector. In our case the function is defined as:

$$c = \sqrt{x^2 + y^2} \tag{3.1}$$

and

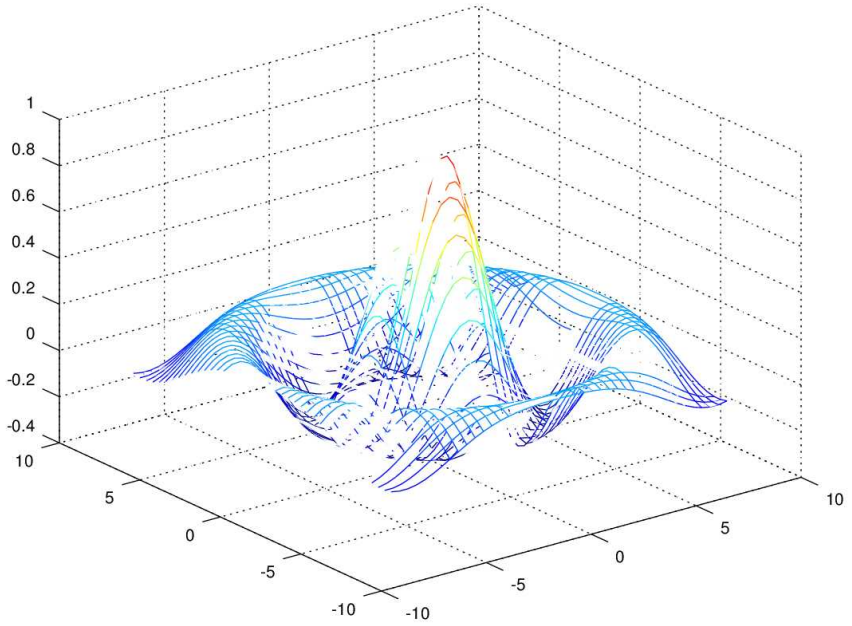


Figure 3.5: 3D Meshing

$$d = \frac{sin(c)}{c} \tag{3.2}$$

Note: Function `eps` produces a very small number ($2.2204.10^{-16}$ for machine used test at the time of writing the book). It is widely used in numerical computation where zero needs to be avoided especially the case of division by zero. By adding a very small number to large numbers, we avoid this problem (remember that variable `c` calculated in 3 is then used under division as a denominator in step 4).

Continuing now the plotting exercise, new arrays can then be used to plot by applying a 3D plotting function `mesh()` which takes these two arrays `a` and `d` as its arguments resulting in figure 3.5. If `mesh(x,y,z)` is used then a wire-frame mesh made up of rectangles. The vertices of the rectangles are made of data points generated by the function (in our case equation. 3.1 and eq. 3.2). The (x, y) coordinated of vertices are given by `xx` and `yy` matrices since x coordinated comes from `xx` matrix and y coordinate comes from `yy` matrix. `z` determines the height above the plane of each vertex. In this way

a 3D plot is plotted. It is important to note that the original 3D "curve" is interpreted as a surface made of flat "rectangles" which is at best an approximation. In some cases, this error can be ignored. To get less error, size of rectangles can be made small, if possible. There are some other variations of the same function like `ezmesh, meshc, meshz`. A simple `help` command can be very useful to judge which one will suit best for a particular problem.

The mesh also codes color for height (z-value). This is computed by linearly scaling the Z values to fit the range of the current color-map (write `help colormap` to know more).

meshc

`meshc()` generates a 3D rectangulated mesh as well a contour at base. As seen in figure 3.6, apart from producing a 3D plot for given function, one also obtains a contour plot. Please note that this time, the equation working on matrices, is written as an argument of `meshc()` function, this making the programs even smaller.

```
x=linspace(-10,10,50);
y=linspace(-10,10,50);
[xx,yy]=meshgrid(x,y);
meshc(xx,yy,2-(xx.^2+yy.^2))
```

ThreeDMeshc.m

surf()

`surf()` generates a surface plot where wire-mesh is simply filled up at empty points, as seen in figure 3.7,

```
a = b = linspace (-8, 8, 10)';
[xx, yy] = meshgrid (a, b);
c = sqrt (xx .^ 2 + yy .^ 2) + eps;
d = sin (c) ./ c;
surf (c,d);
```

ThreeDsurf.m

3.2 Summary

A rich library of plotting functions makes octave a suitable choice for plotting data in a variety of publication-ready formats. Together with commands to

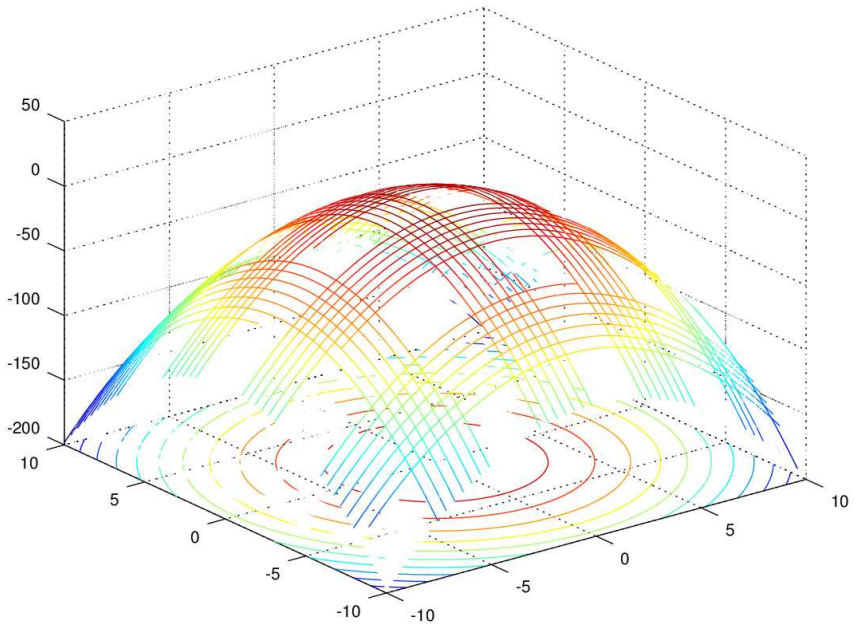


Figure 3.6: 3D Meshing with the function meshc()

access systems files and folders, these plots can be directed to be saved at appropriate places for making a suitable report. Plotting in 3D and viewing with different angles is quite intuitive in octave. Hence octave presents a suitable choice to visualize the data.

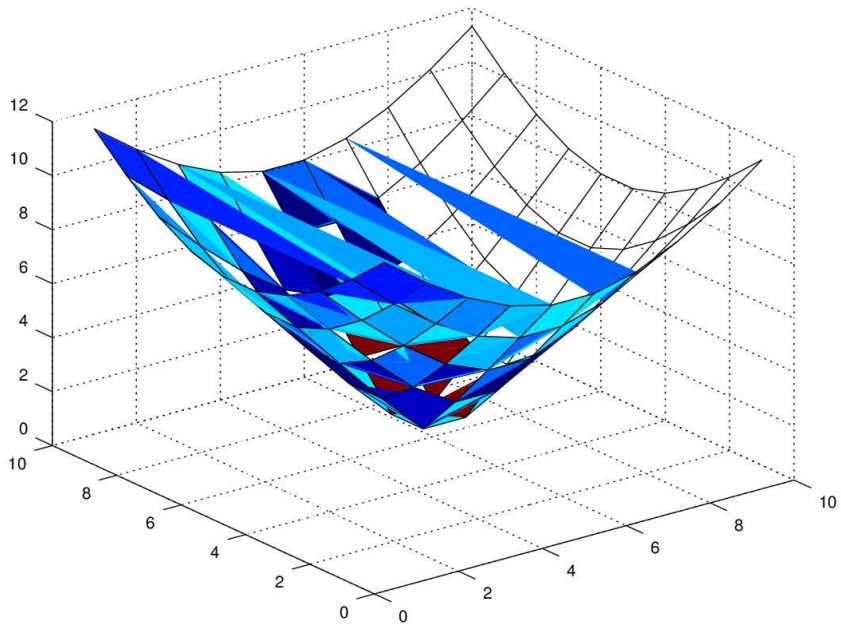


Figure 3.7: 3D Meshing with the function surf()

4

Data through File reading and writing

4.1 Introduction

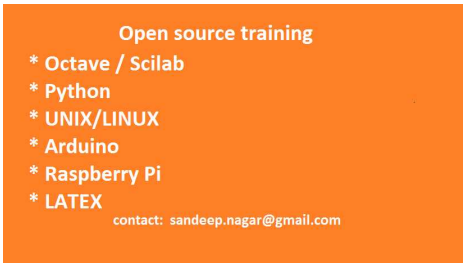
Using the information in chapter on arrays and chapter on plotting, one can now formulate physical problems in terms of numerical computations and solve them on digital computer. This process has some requirements such as:

- Data should be in digital form (a digital file)
- Computer program should be able to read the file and make array without errors. If errors have been made, then a mechanism to check those errors and giving a warning to the user should be in place. If possible correcting them should also be in place.
- Data should be stored as an array in proper data type and should be displayed on demand in proper format.
- Array operations on data will result in memory usage in terms of reading and writing data on disk. This should be facilitated by the system. User should be able to check the status of memory as and when required.
- Post-processing tasks includes displaying data in various formats: As a printout from printer, on a terminal, as a graph on terminal or printer/plotter etc.

- If a report for particular experiment having input parameters, processing data and output as file or graph, can be generated, then it makes the task of user easier.

Octave has some features for each of these steps. Present chapter will discuss them in brief.

4.2 File operations



File operations constitutes an important part of computation. It is important to note that file system is OS (Operating System) dependent. Octave was traditionally written for UNIX-like systems so it works on Linux based and Mac OS X equally well with same set of commands. On windows, one uses same commands as that of Linux for dealing with file. Codes written below has been tested on Windows 8, Mac OSX 10.10 and Ubuntu 14.04 systems.

4.2.1 Users

A computing system is accessed by different users. Each user defines a workspace to avoid damaging each other's work. After login, a user's workspace becomes active for a user. Workspace is made up of various files and folders. Some files are essential for the OS to define the workspace and its properties, hence they should not be altered at each time. This is ensured by giving permissions for various users. "Reading" and "Writing" a file is restricted by permissions. "Administrator" (fondly called `admin`) is also called "super user" who has all the privileges of having all permissions to edit any file/-folder. One must understand the defined user-type for on computer system and then issue commands accordingly. If one is not permitted to access certain folders and input data is placed inside those files/folders, then unless one seeks to change the permission from the `admin`, one would always get an error.

4.2.2 File Path

Directory/Folder can contain sub-directories/sub-folders and files again. This can go to any level if this process if not restricted by the administrator.

`pwd` command stands for **print working directory**. On octave terminal, typing `pwd` displays the path of present working directory as shown in example below.

```
1 >> pwd
2 ans = /home/sandeep
```

Under the user, `/home` directory, contains another directory named `/sandeep`. This is the present working space. When `pwd` is typed on the terminal, a variable name named `ans` stores this data (file path). A variable name of choice, can be assigned to store the filename as a string.

A file/folder is accessed by writing file-path on the terminal. Lets do a small exercise to understand this process. To make a new directory, use `mkdir 'name'` as follows:

```
1  >> mkdir octave
2  ans =  1
3  >> ls
4  Downloads                                    Music
5  R
6  Templates
7  octave
8  Videos
9  Desktop                                        software
10 Work
11 Documents              Library
12 Pictures
13 >> cd octave
14 >>
```

At line number 1, `mkdir octave` makes a directory named "octave". To see the contents of the present directory, one can use the command `ls` ,as done at line number 3, which stands for *list*. To change directory, one can use the command `cd "file path"` as shown in line number 13. It is suggested that one works in this directory for rest of the book.

4.2.3 Creating files and saving them

`save` and `load` commands enables one to **write** and **read** data to the memory.

```
>> matrix = rand(3,3);
>> save MyFirstFile.mat matrix
>> ls
MyFirstFile.mat
>> load MyFirstFile.mat
>> matrix
matrix =

    0.467414    0.610273    0.429941
    0.568490    0.037898    0.734682
    0.547370    0.275421    0.539650

>>
```

At line number 1, A variable named "matrix" is created first, which stored a random values 3×3 matrix. At line number 2, this data is stored as a .mat file named `MrFirstFile.mat`, which is passed the variable name as the argument. When required, this file can be loaded in workspace using `load MyFirstFile.mat` and then calling the variable name `matrix`. Those random number which were recorded at the time of saving the file, are loaded as the data for 3×3 matrix. Please note that the data need not be numbers always. It can anything which a digital computer can handle, like pictures, videos, strings, character just to name a few.

Multiple variables can be stored in the same file by passing the name of variables at the time of saving.

```
>> matrix1 = rand(4,4);
>> matrix2 = rand(2,3);
>> matrix3 = rand(2,2);
>> save ("SavingMultipleVariables.mat", "matrix1", "matrix2", "
    matrix3")
>> load SavingMultipleVariables.mat
>> matrix1
matrix1 =

    0.8598130    0.0118250    0.9803720    0.3044413
    0.6676748    0.0056845    0.1101545    0.2183920
    0.2547204    0.8192626    0.8056112    0.6961116
```

```
12     0.7924558    0.9130480    0.1976146    0.4635055
13
14 >> matrix2
15 matrix2 =
16
17     0.35215    0.55770    0.66650
18     0.98515    0.98677    0.45513
19
20 >> matrix3
21 matrix3 =
22
23     0.097693    0.540354
24     0.923853    0.329501
25
26 >>>> save -binary SavedAsBinary m*
27 >> ls
28 MyFirstFile.mat   SavedAsBinary   SavingMultipleVariables.mat
```

`help save` and `help load` gives very useful instructions about using them. Using "options" one can save the file in a specific format. For example, at line number 26, all variables names starting with "m" are saved as binary data inside a binary file named "SavedAsBinary". This is particularly important for the case were data generated from octave based numerical computation is used to feed another software. One can also specify precision of saved data using options. Also one can compresses a big file using `-zip` command. This is very useful in case the data generated by octave is large in size and needs to be transmitted.

`load` function follows the same logic as `save` function. Data can be unzipped and loaded from a particular formatted file as an array. Array thus populated, can be used for computation and resultant files can be made using `save` function again (if required). Elaborate computations require this procedure to be repeated successively many times, thus the functions have been optimized to locate and load required data in a short time.

diary

An octave session can be recorded in a file by using the command `diary`. Using `help diary` its use can be obtained. Writing `help "filename"` allows recording the session at a file with given filename. The commands and their outputs are continuously updated using this function.

Using the command `history` a list of executed commands is displayed. Various options are available to see this history in particular formats.

Opening and closing files

To read and write data files, they must be opened and defined as readable and/or writable. The `fopen` function returns a pointer to an open file that is ready to be read or written. This is defined by an option "r" as readable, "w" as writable, "r+" as readable and writable, "a" for appending i.e. writing new content at the end of the file, "a+" for reading, writing and appending. Opening **mode** can be set as "t" for text mode or "b" for binary mode. "z" enables opening a gzipped file for reading and writing.

Once all data has been read from or written to the opened file it should be closed. The `fclose` function does this.

```
MyFile = fopen ("a.dat", "r");
```

A variable `MyFile` is created which is used to store the contents of the file `a.dat`. This file is opened in "reading mode" only in the sense that it cannot be edited. This is important if author of the file wants it to remain unchanged while sharing like file containing constants or important piece of code which should not be changed etc. `freport()` prints a list of files opened and whether they are opened for reading, writing or both. For example:

```
>> freport

number   mode   arch         name
------   ----   ----         ----
0        r      ieee-le      stdin
1        w      ieee-le      stdout
2        w      ieee-le      stderr

>>
```

Reading and writing binary files

A binary file is computer readable file. They are simply sequence of bytes. Same as C functions, `fread` and `fwrite` functions can read and write binary data from a file.

csvread and csvwrite

Functions `csvread` and `csvwrite` are used to read data from .csv filed which stand for **comma seperated values**.

Suppose the following data needs to be stored as a csv file.

1 2 3 4
5 6 7 8
8 7 6 5
4 3 2 1

Following code makes an array using `csvwrite` to create a file named `csvTestData.dat` containing the matrix values. One can check by simply opening this newly created file in a text editor. At line number 3, a new file named `csvTestData1.dat` is created with offset defined at row 1 and column 2.

```
>> a = [1,2,3,4 ; 5,6,7,8 ; 8,7,6,5 ; 4,3,2,1 ];
>> a
a =

1   2   3   4
5   6   7   8
8   7   6   5
4   3   2   1
>> csvwrite('csvTestData.dat',a)
>> csvwrite('csvTestData1.dat',a, 1,2)
>> a1 = csvread('csvTestData.dat')
a1 =

1   2   3   4
5   6   7   8
8   7   6   5
4   3   2   1

>> a1 = csvread('csvTestData.dat', 1, 2)
a1 =

7   8
6   5
2   1

>>
```

Now `csvread` function can be used to create matrices with desired offsets just as the function `csvwrite`.

Note: A number of other functions to read and write files exist, but the present section focuses on some of the most commonly used ones. Documentation can be accessed to know about using these specialized functions, if required.

4.2.4 Working with Excel files

A lot of data is present on Internet, in the form of an excel file. Octave has a separate module to work with these files but it first needs to be installed. The module "IO" is part of octave-forge project where to install a module, one has to write `pkg install -forge package_name` at octave command prompt:

```
pkg install -forge io
```

Please note that one must be connected to Internet in above case.

Once the module has been automatically installed at a proper place, its functions can be used. Following is the list of file extensions and associated permissions.

```
File extension      COM POI POI/OOXML JXL OXS UNO OTK JOD OCT
-------------------------------------------------------------
.xls (Excel95)       R                R       R
.xls (Excel97-2003)  +   +       +    +   +   +
.xlsx (Excel2007+)   ~           +        (+) +           +
.xlsb, .xlsm         ~                    ?   R           R?
.wk1                 +                        R
.wks                 +                        R
.dbf                 +                        +
.ods                 ~                        +   +   +   +
.sxc                                          +       +
.fods                                         +
.uos                                          +
.dif                 +                        +
.csv                 +                        R
.gnumeric                                                +
-------------------------------------------------------------

R : only read; + : full read/write; ~ : dependent on Excel
    version
```

To open, read, write and close an excel file

`xlsopen, xlswrite, xlsclose, odsopen, odswrite, odsclose` commands open, write and close the `.xls` and `.ods` files respectively. While `.xls` files are generated using Microsoft Excel software, `.ods` files are generated using Open/Libre Office software, which is open source equivalent of Microsoft Excel software. The process of opening, reading and writing data is as follows:

- `xlsopen('Filename.xls')`
- `a = xlsread ('Filename.xls', '3rd_sheet', 'B3:AA10');`

  Numeric Data from the file `Filename.xls`'s worksheet named 3^{rd} sheet will be read from cell B3 to AA10. This data is stored as an array named `a`.
- `[Array, Text, Raw, limits] = xlsread ('a.xls', 'hello');`

  The file `a.xls` is read from the worksheet named `hello`, and the whole numeric data is fed into an array named "Array", the text data is fed into array named "Text", the raw cell data into cell array "Raw" and the ranges from where the actual data came in "limits"
- `xlswrite('new.xls', a)` writes the data in array named `a`, into `.xls` formated excel sheet named `new.xls`.
- `xlsclose`

```
>> pkg load io
>> a = rand(10,10);
>> odswrite('a.ods',a)
ans =  1
>> ls
a.ods
```

4.3 Taking data from the Internet

Most often, useful large data sets are kept at some remote server. Using `urlread()` one can read the remote file. For saving at the local disk, one can use `urlwrite()` functions.

```
>> a = urlread('http://www.fs.fed.us/land/wfas/fdr_obs.dat');
>> who
Variables in the current scope:

a    ans

>> whos
Variables in the current scope:

Attr Name        Size                     Bytes  Class
==== ====        ====                     =====  =====
a               1x147589                147589  char
ans             1x1                          8  double

Total is 147590 elements using 147597 bytes

>> urlwrite('http://www.fs.fed.us/land/wfas/fdr_obs.dat','fire.
    dat')
>> ls
fire.dat
>>
```

Here a variable named `a` stores the data from the data file stored at `http://www.fs.fed.us/land/wfas/fdr_obs.dat`. Alternatively, the whole data is stored as a file named `a.dat` using the function `urlwrite(URL)`.

4.4 Printing and saving plots

Some commands like `print, saveas` exist to save graphs/figures generated by octave programs, to be saved in desired formats. They are discussed below:

4.4.1 print

`print` command handles the printing jobs such as printing using a printer and/or plotter, printing to a file etc. Especially for figures, this command is very useful to save a figure automatically by a desired filename in a specified format.

```
% Saving in svg format
figure (1);
clf ();
peaks();
```

```
print -dsvg figure1.svg

% Saving in png format
figure (1);
clf ();
sombrero ();
print -dpng figure2.png

% Printing to a HP DeskJet 550C
clf ();
sombrero ();
print -dcdj550
```

`clf` function clears the current graphic window. A lot of other "options" for saving in different formats exist for `print` command. To know more, please type `help print` at octave terminal.

4.4.2 saveas

`saveas` functions saves a graphic object in a desired format as follows:

```
clf ();
a = sombrero ();
saveas (a, "figure3.png");
```

4.4.3 orient

`orient(a,orientation)` function defines the orientation of an graphical object "`a`". The valid values for `orientation` parameters are `portrait`, `landscape`, and `tall`.The `landscape` option changes the orientation so the plot width is larger than the plot height. The `tall` option sets the orientation to `portrait` and fills the page with the plot, while leaving a 0.25 inch border. The `portrait` option (default) changes the orientation so the plot height is larger than the plot width.

4.5 Summary

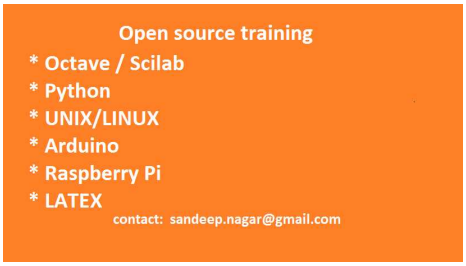
In present chapter, various functions enabling reading and writing permission as well as taking data to and from a file, has been illustrated. This becomes an essential part of a numerical computation exercise. The data

can be generated in the form of files using a software or hardware (an instrument). Octave does not care for its origin. It treats data by its type and by file-type. Judging an appropriate function to operate using files, has to be done by the user as per the situation. File operation does provide faculties to trim the data so that only useful part of data is fed as an array. Further trimming can be performed by slicing operations. With the art of handling files, one can confidently proceed towards handling sophisticated numerical computations.

5 Functions and loops

5.1 Introduction

When a particular numerical tasks needs to be "repeated" over different data points, digital computers becomes a useful tool since they can perform the same with greater speeds than humans. **Loops** perform exactly this tasks. Using a condition to check the start and termination rules, one can perform repetitive parts of a process easily. Different programming languages and environments have different rules of defining loops. Octave provide a much simpler way to define and run loops. They will be discussed shortly. Its useful to define the term **function** here. A big program may require a set of instructions to be called at different times. Hence these set of instructions can be defined as a sub-program, which can be requested to perform the computation at a desired time. In this way, a complicated task can be divided into many small parts. This architecture of programming is called *modular programming*. This is the most popular way of programming since its quite logical, better at visualizing the problem and easy to debug. The most popular way of defining the these small set of instructions is to define them as functions. Present chapter will discuss both these concepts in details.



5.2 Loops

Loops form an essential part of an algorithm since they perform the tasks which computers perform best: doing repetitive actions in a very fast manner. Loops can come in many flavors like `for` loop which repeats certain tasks over a list of variable values, `while` loop which checks a logical condition before executing certain task and `if-then-else` loop which checks a condition and directs the flow of algorithm. Choice of a particular loop depends on the problem at hand.

A variety of functions and their usage is listed below. Judging their usage critically becomes supremely important because the looping part of algorithm consumes most of execution time.

5.2.1 while

`while` loop defines a logical condition and until it is satisfied, it run a block of code. The syntax for while loop is:

```
while condition
BODY
endwhile
```

Here the keyword `while` initiates the execution of a while loop. The `condition` is a logical condition whose answer can be '`true`' (1) or '`false`' (0). The `BODY` encompasses the st of commands which is executed until the condition holds true.

```
x = 1.0;
while x < 10
    disp(sqrt(x));
    x = x+1;
endwhile
```

while1.m

The program `while1.m` runs by first initializing a variable `x` to value 1.0. Then it lists a logical condition:

$$x < 10$$

.

At the first step of loop, $x = 1$, this condition is satisfied since $1 < 10$. When this condition is satisfied, `disp(sqrt(x))` is executed which displays

the square root of x. Then line number 4 is executed where `x = x + 1` increments `x`. With new incremented value of `x` to 2, the logical condition $x < 10$ is again checked and the body of loop given by lines 3 and 4 are executed. This is done until $x = 10$ when the loop condition is not satisfied, hence line number 5 is executed, which declares the end of `while` loop. The execution of file `while1.m` yields:

```
>> while1
1
1.4142
1.7321
2
2.2361
2.4495
2.6458
2.8284
3
```

5.2.2 do-until

It is important to note that there can be cases where the body of a loop might not get executed even one in the case of `while` loop. This is the case when after initialization, condition is not satisfied. To overcome this kind of scenario `do-until` loop is framed whose syntax is as follows:

```
do
BODY
until condition
```

The loop first executes the body of code and then check for condition. This way, the code block comprising the `BODY` of loop is **at least executed once**. The usage can be understood in the example below:

```
% Displaying square root of
% first ten positive natural numbers

x = 1.0;
do
  disp(sqrt(x));
  x = x+1;
until x == 10
```

dountil1.m

The execution of code yields:

```
>> dountil1
1
1.4142
1.7321
2
2.2361
2.4495
2.6458
2.8284
3
>>
```

At line number 4, x is initialized at 1.0. Then the body of loop is written for displaying the square root of x and then incrementing it by 1. This is done until $x = 10$ i.e value of x becomes 10.

5.2.3 for

`for` loop is used to perform computation on a list of known values. The syntax of `for` loop is:

```
for variable = vector
  BODY
end
```

The keyword `for` declared the starting of loop where a variable takes the values stored in a vector. Then a body of code (here represented by `BODY`) is executed. The keyword `end` declares end of `for` loop. This is explained in the example below:

```
% program to calculate squate root
% of first 10 numbers

for i = 1:10
  ans = sqrt(i)
end
```

for1.m

Executing `for1.m` yields:

```
>> for1
ans =  1
ans =  1.4142
ans =  1.7321
ans =  2
ans =  2.2361
ans =  2.4495
ans =  2.6458
ans =  2.8284
ans =  3
ans =  3.1623
```

5.2.4 if-elseif-else

Situation where a number of conditions needs to be checked at different points of times, `if-elseif-else` loop works well. The syntax for the loop is given by:

```
1 if condition1
2 BODY1
3 elseif condition2
4 BODY2
5 else
6 BODY3
7 endif
```

At line 1, a condition is defined. If this condition is satisfied then the line 2 is executed or else line 3 is executed. Hence `BODY1, BODY2` are the blocks of codes which are executed by checking for different set of conditions and `BODY3` set of codes is executed in the case when none of the condition is executed.

```
% Program to check if a
% number is even or odd

x = 33;

if (rem (x, 2) == 0)
  printf ("x is even\n");
elseif (rem (x, 5) == 0)
  printf ("x is odd and divisible by 5\n");
else
```

```
11   printf ("x is odd\n");
12 endif
```

ifelse1.m

Executing `ifelse1.m` yields:

```
1 >> ifelse1
2 x is odd and divisible by 5
```

At line number 4, x is initialized as 33. Then at line number 6, the remainder of $\frac{x}{2}$ is checked. If it is zero then line number 7 is executed or else line number 8 is executed where remainder of $\frac{x}{5}$ is checked. If it is zero then line number 9 is executed. If both the conditions are not satisfied then Line number 11 is executed and then line number 12 declares ending the `if-else` loop.

5.3 Functions

Function is a set of code which can be called as and when required. Hence it can be defined separately either in a separate file or within the body of program. Octave presents some ways to define a functions as discussed in following subsections.

5.3.1 function

The definition of a function follows the syntax:

```
1 function [return value 1, return value 2, ...] = name( [arg1,
      arg2, ...] )
2 body
3 endfunction
```

Here `function` keyword defines the object types as function. Then a set of variables are defined which this function is expected to return. Next comes an `=` operator. Then the name of function. In above case its `name`. Name objects takes a set of arguments which are objects using which the function is defined. Then comes the main body of function. The last part is to define the end of function.

For example, one can write a function to find $x^2 - y^2$ and assign it to variable name z.

```
function y = fn1 (x,y)
y = x^2 - y^2;
end
```

Save this as `fn1.m` in the present working directory. Now go to the octave terminal and type:

```
>> fn1(5,1)
ans =  24
>> fn1(5,2)
ans =  21
>> fn1(5,3)
ans =  16
>> fn1(5,4)
ans =  9
>> fn1(5,5)
ans = 0
```

Hence one can see that function named `fn1` is performing the computation $x^2 - y^2$ on the two input arguments for which it is defined.

It is a good practice to define the program as a group of **function files** and call them in the master program stored as a **script file**. This modular approach makes it easy to experiment with the idea and also makes it easier to debug and test the code. A function can return more than two values too. For example:

```
function [y1,y2,y3] = fn2 (x,y)
y1 = x^2 - y^2;
y2 = x^2 + y^2;
y3 = y2 - y1;
end
```

This gives the following result:

```
>> [a,b,c] = fn2 (5,2)
a =  21
b =  29
c =  8
>> [a,b,c] = fn2 (5,0)
```

```
a =  25
b =  25
c = 0
```

Functions can incorporate loops to regulate the repetitive tasks inside the program. For example, factorial of a number can be calculated using a function given below:

```
function result = factorial( n )
  if( n == 0 )
    result = 1;
    return;
  else
    result = prod( 1:n );
  endif
endfunction
```

A function named factorial, which takes a number `n` as an argument calculates the product of number with all its successive numbers. When called from octave command line, the function yields the following result.

```
>> factorial(50)
ans =  3.0414e+064
>> factorial(1)
ans =  1
>> factorial(0)
ans =  1
>> factorial(100)
ans =  9.3326e+157
>> factorial(1000)
ans = NaN
>> factorial(-1)
error: factorial: N must all be non-negative integers
```

`help NaN` and `help prod` gives useful insights into the behavior of these commands.

5.3.2 inline

Functions can also be defined **inline** using the command `inline` as follows:

```
>> f = inline("x^2+y");
>> f(1,2)
```

```
3 ans =  3
4 >> f(10,10)
5 ans =  110
6 >> f(0,2)
7 ans =  2
8 >>
```

Line number 1 defines a function named `f` with two variables `x` and `y` to calculate $f(x, y) = x^2 + y$. When called with values of these two variables, it outputs the calculated values.

5.3.3 Anonymous function

Anonymous functions are unnamed function objects defined in the program. There definition follows a simple syntax:

```
@(argument list) expression
```

For example:

```
1  >> a = @(x) sin(x)*cos(x);
2  >> quad(a, 0, 1)
3  ans =  0.35404
4  >> quad(a, 0, pi)
5  ans =  7.3031e-017
6  >> quad(a, -pi, pi)
7  ans = 0
8  >> quad(a, -pi, 2*pi)
9  ans =  -2.8435e-016
10 >> quad(a, -2*pi, 2*pi)
11 ans = 0
```

`help quad` tells us that the function `quad` evaluated the integration of a function between two values. Hence line 1 defines a function $sin(x)cos(x)$ whose integration.

$$\int_0^1 sin(x)cos(x) = 0.35404$$
$$\int_0^\pi sin(x)cos(x) = 7.3031 \times 10^{-17}$$
$$\int_{-\pi}^\pi sin(x)cos(x) = 0$$
$$\int_{-\pi}^{2\pi} sin(x)cos(x) = -2.8435 \times 10^{-16}$$
$$\int_{-2\pi}^{2\pi} sin(x)cos(x) = 0$$

Hence using anonymous function definition, one need not name a function.

5.4 Summary

Defining functions is the key to modular programming. Octave presents an elegant way to define and use functions both inline and in separate files. When combined with the ability to write functions inside a loop, complex problems can be implemented in few lines of codes. It requires an artistic attitude while designing an algorithm where functions and loops are the paintbrush to devise an elegant solutions to a given numerical problem.

6 Numerical Computing formalism

6.1 Introduction

Numerical computation enables us to compute solutions to numerical problems, provided we can frame them into a proper format. This requires certain considerations. For example, if we digitize continuous functions, then we are going to introduce certain errors due to the sampling at a finite frequency. Hence a very accurate result would require very fast sampling rate. In cases when a large data set needs to be computed, it becomes computationally intensive and time consuming task. Also one must understand that the numerical solutions are an approximation at best, compared to analytical solutions. The onus of finding their physical meaning and significance lies on us. The art of discarding solutions which do not have a meaning for real world scenario, is something which a scientist/engineer develops over the years. Also, a computational device is just as intelligent as its operator. The law of GIGO (Garbage-In-Garbage-Out) is followed very strictly in this domain.

In the present chapter, we shall try to understand some of the important steps one must consider to solve a physical problem using numerical com-

putation. Defining a problem in proper term is just the first step. Making the right model and then using the right method to solve (solver) enables to distinguish between a naive and experienced scientist/engineer.

6.2 Physical problems

Everything in our physical world is governed by physical laws. Owing to men and women of science who toiled under difficult circumstances and came up with fine solutions to things happening around us, we obtained mathematical theories for physical laws. To test these mathematical formalisms of physical laws, we use numerical computation. If it yields the same results as that of a real experiment, the validate each other. Numerical simulations can remove the need of doing an experiment altogether provided we have a well tested mathematical formalism. For example, nuclear powers of our times need not test nuclear bombs for real any more. The data about nuclear explosion, which was obtained during real nuclear explosions, enabled scientists to model these physical systems quite accurately, thus eliminating the need to a real testing.

Apart from applications like simulating a real experiment, modeling physical problems are good educational exercises. While modeling, hands-on exercises enables students explore the subject in depth and give a proper meaning of topic under study. Solving numerical problem and visualization of results makes the learning permanent and also ignites the research about flaws in mathematical theory which ultimately leads to new discoveries.

6.3 Defining a model

Modeling means writing equations for a physical system. As the name suggests, an equation is about equating two sides. An equation is written using an = sign where terms on left hand side is equal to term on right hand side. The terms on either sides of equations can be numbers or expressions. For example:

$$3x + 4y + 9z = 10$$

This is an equation having a term $3x + 4y + 9z$ on left hand side (LHS) and a term 10 on right hand side (RHS). Please note that whereas LHS is an algebraic term, RHS is a number.

Expressions are written using functions which is simply a relation between two domains. Like $f(x) = y$ is a relation from y to x using rules of algebra. Mathematics has a rich library of functions using which one can make expressions. Choice of proper functions depend on problem. Some functions describe some situations best. For example, oscillatory behavior can be described in a reasonable manner using trigonometric functions like $sin(x), cos(x)$ etc. Objects moving in straight lines can be described well using linear equations like $y = mx + c$ where x is their present position, m is constant rate of change of x w.r.t y and c is the offset position. Objects moving in a curved fashion can be described by various non-linear functions (where power of dependent variable like x above, is not 1).

In real life we can have situations which can be mixture of these scenarios. Like an object can oscillate and move in curved fashion at the same time. In that case we write an expression using mixture of functions or find new functions which could explain the behavior of object. Verifying the functions is done by finding solutions to equations describing the behavior and matching it with observations taken on object. If they match perfectly, we obtain perfect solutions. In most cases, an exact solutions might be difficult to obtain. In these cases, we get an "approximate" solution. If the errors involved while obtaining an approximate solution are within toleration limits, the models can be acceptable.

As discussed above, physical situations can be analytically solved by writing mathematical expressions in terms of functions involving dependent variables. Simplest problems have simple functions between dependent variables with a single equation. There can be situation where multiple equations are needed to explain a physical behavior. In case of multiple equations being solved, the theory of matrix comes handy.

Suppose equations below define the physical behavior of a system:

$$-x + 3y = 4 \tag{6.1}$$
$$2x - 4y = -3 \tag{6.2}$$

Then this system of two equations can be represented by a matrix equation as follows:

$$\begin{bmatrix} -1 & 3 \\ 2 & -4 \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \end{bmatrix}$$

Now using matrix algebra, values of variables x and y can be found such that they satisfy the equations. Those values are called roots of these equations. These roots are the point in 2-D space (because we had 2 dependent variables) where the system will find stability for that physical problem. In this way, we can predict the behavior of system without actually doing an experiment.

Mathematical concept of differentiation and integration becomes very important where we need to work with dynamic system. When the system is constantly changing the values of dependent variables to produce a scenario, then its important to know the rate of change of these variables. When these variables are independent of each other, we use simple derivatives to define their rate of change. When they are not independent of each other, we use partial derivatives for the same.

For example, Newtons second law of motion says that rate of change of velocity of an object is directly proportional to the force applied on it. Mathematically:

$$F \propto \frac{dy}{dx} \tag{6.3}$$

The proportionality is turned into equality by substituting for a constant of multiplication m such that:

$$F = m \times \frac{dy}{dx} \tag{6.4}$$

If we know values or expressions for F, this equation can be solved analytically and solutions can be found to this equation. But in some cases, the analytical solution may be too difficult to obtain. In those cases, we digitize the system and find a numerical solution.

There are many methods to digitize and numerically solve a given function. Programs to implement a particular method to solve a function numerically, is called a solver. A lot of solvers exist to solve a function. Choice of solver is critical to successfully obtain a solution. For example, equation 6.4 is a differential equation. It is a first order ordinary differential equation. A number of solvers exist to solve it like Euler, Runge-Kutta etc. Choice of particular solver depends on accuracy of its solution, time taken for obtaining a solution and amount of memory used during the process. The latter is important where memory is not an freely expendable commodity

like micro-computers with limited memory storage.

The advantage of using octave to perform a numerical computation lies in the fact that it has a very rich library of functions to perform various tasks required. The predefined functions has been optimized for speed and accuracy (in some cases, accuracy can be predefined). This enables the user to rapidly prototype the problem instead of concentrating on writing functions to do basic tasks and optimizing them for speed, accuracy and memory usage.

6.4 Octave Packages

A number of packages exist to perform numerical computation in a particular scientific domain. The website `http://octave.sourceforge.net/` gives a list of packages. Installing package can be simply attained by writing the command

```
>> pkg install -forge package_name
```

on the octave command line.

6.5 Summary

Almost all branch of science and engineering requires one to perform numerical computation. Octave is one of the alternative to do so. Octave has a library of optimized functions for general computation. Also it has a variety of packages are present to perform a specialized job. This makes it an ideal choice for prototyping a numerical computation problem efficiently.