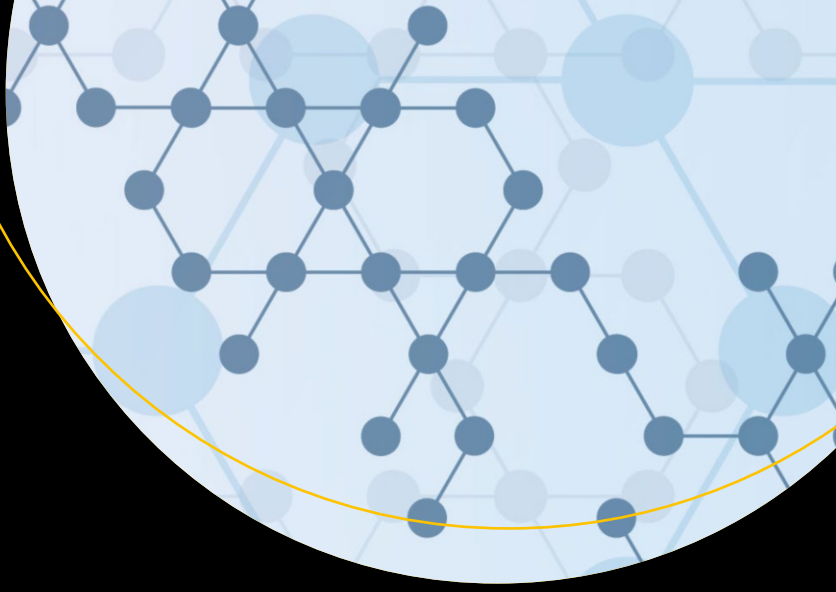

Introduction to MATLAB for Engineers and Scientists

Solutions for Numerical Computation and Modeling

Sandeep Nagar

Apress®

Introduction to MATLAB for Engineers and Scientists

Solutions for Numerical Computation and Modeling

Sandeep Nagar

Apress®

Introduction to MATLAB for Engineers and Scientists: Solutions for Numerical Computation and Modeling

Sandeep Nagar
New York, USA

ISBN-13 (pbk): 978-1-4842-3188-3 ISBN-13 (electronic): 978-1-4842-3189-0
https://doi.org/10.1007/978-1-4842-3189-0

Library of Congress Control Number: 2017960835

Cover image by Freepik (www.freepik.com)

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Technical Reviewer: Massimo Nardone
Coordinating Editor: Mark Powers
Copy Editor: Kezia Endsley

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484231883. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Dedicated to my wife Rashmi and my daughter Aliya

Table of Contents

About the Author

Sandeep Nagar, PhD (Material Science. KTH, Sweden), teaches and consults on the use of MATLAB for numerical computing and other open source software. In addition to teaching at universities, he frequently gives workshops covering open source software and is interested in developing hardware for scientific experiments.

About the Technical Reviewer

Massimo Nardone has more than 22 years of experience in security, web/mobile development, cloud and IT architecture. His true IT passions are security and Android.

He has been programming and teaching others how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years.

He holds a Master of Science degree in Computing Science from the University of Salerno, Italy.

He has worked as a project manager, software engineer, research engineer, chief security architect, information security manager, PCI/SCADA auditor, and senior lead IT security/cloud/SCADA architect for many years.

Massimo's technical skills include: Security, Android, Cloud, Java, MySQL, Drupal, Cobol, Perl, web and mobile development, MongoDB, D3, Joomla, Couchbase, C/C++, WebGL, Python, Pro Rails, Django CMS, Jekyll, and Scratch.

He currently works as the Chief Information Security Officer (CISO) for Cargotec Oyj.

He was a visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University). He holds four international patents (in the PKI, SIP, SAML, and Proxy areas).

Massimo has reviewed more than 40 IT books for different publishing companies and he is the co-author of *Pro Android Games* (Apress, 2015).

Acknowledgments

I wish to thank Steve, Mark, and the whole team at Apress for bringing this book to fruition. I also wish to thank the scientific community for answering questions on forums, which helped me learn some difficult concepts with ease.

CHAPTER 1

Introduction to MATLAB

1.1 Introduction to Numerical Computing

With the advent of computers in the post World War II era, the need to simulate physical problems using this new tool led to the invention of numerical computing. Whereas analytical computation required pen, paper, and the human mind, numerical computation required a calculating device too. Successful implementation of computing devices to solve problems (especially involving repeated tasks) over a large array of data points was observed in many fields of science and engineering. For example, breaking enemy's secret codes, simulating nuclear reactions before nuclear explosions, etc. The scope further expanded to civilian purposes, such as designing and simulating waterways, dams, electric power stations, town planning, etc. All of these applications need to use an equation or systems of equation for a physical model representing a physical problem. There are two ways that one can approach these equations—using analytical and numerical techniques. We concentrate only on the numerical methods of solving equations using MATLAB in this book.

S. Nagar, *Introduction to MATLAB for Engineers and Scientists*,
https://doi.org/10.1007/978-1-4842-3189-0_1

As time progressed, various schemes to define mathematical functions—differentiation, integration, trigonometric, etc.—were written for digital computers. This involved digitization, which certainly introduces errors. Knowledge of errors and their proper nullification could yield valuable information quicker than analytical results. Thus, it became one of the most actively researched fields of science and continues to be one. The search for faster and more accurate algorithms continues to drive innovation in the field of numerical computing and enables humanity to simulate otherwise impossible tasks.

1.2 Tools for Numerical Computing

As the numerical methods progressed as an alternative to analytical methods, computer programming languages were increasingly being used to codify them for programmed investigations of simulations. A number of options [1] exist to perform numerical computation. Programming languages written to handle mathematical functions like FORTRAN, C, Python, Java, and Julia, to name a few, can be used to write algorithms for numerical computation.

1.2.1 The Need for Specialized Software

While all problems can be coded in programming languages, it's necessary to change the approach to computing, file management, etc. when the microprocessor platform or operating system changes. This hinders interoperability. Modern programming languages address some of these issues, but the need for specialized software for numerical computing—where predefined tools of numerical methods can be simply *called* as and when required and customized tools can be developed—was being felt in academia. A number of attempts were made in this direction.

1.2.2 The History of MATLAB

MATLAB was one such program and it was developed by Cleve Moler [2], who was a math professor at the University of New Mexico, teaching numerical analysis and matrix theory. As a PhD student, he initially wrote a lot of code in FORTRAN to solve systems of simultaneous linear equations involving matrix algebra, which ultimately he called MATrixLABoratory (MATLAB). As a professor he wished his students could use the new packages without writing FORTRAN programs.

Hence, in late 1970s, the first version of MATLAB came out (written in FORTRAN). There were 80 functions for performing calculations involving linear algebra problems. Further down the line, Jack Little and Steve Bangert reprogrammed MATLAB in C with additional features for producing a commercial version of the software. Together, all three of them founded The MathWorks [3] in California in 1984, which develops, maintains, and distributes MATLAB and its products worldwide. MATLAB has proven to be an excellent tool for numerical methods [4].

Over a period of time, so many tools and features have been added to the base package of MATLAB that, along with this rich set of libraries, the installation requirements run it is many GBs of data. MATLAB became tremendously popular in the scientific community. It is used by more than 5,000 universities worldwide. It is sometimes rightly termed the "language of engineering". Cheap availability of digital computing resources propelled its usage in industry and academia to such an extent that virtually every lab needs MATLAB now.

1.3 Installation Requirements

MATLAB should be purchased from the official web site of MathWorks [3] or from an official distributer. The computer system requirements depend on the type and number of optional tools [5] installed with the base

MATLAB package. This book discusses the usage of the base MATLAB package. Hence, to have a good experience with your MATLAB software, use a laptop or workstation with 1GB RAM and any of operating systems—Windows, Linux, or MacOSX. Installation instructions are given with the product. The MATLAB environment is similar on all systems, so you need not worry about this while practicing with the book. This book has been tested for MATLAB R2017a version on the MacOSX 10.12 operating system.

1.4 Workspace

There are two ways to work within MATLAB. The first way is to work at the command line by writing one command at a time. The second method is to write a script (an `.m` file having a set of commands in a sequence) and run it from the command line by simply typing its name. For example, to run the `a.m` script file, you simply write the following at the command prompt:

```
1  >>a
```

The command prompt is represented by the symbol >> by default. You enter a command at the command prompt and then press the Enter key to execute the command. See Figure 1-1.

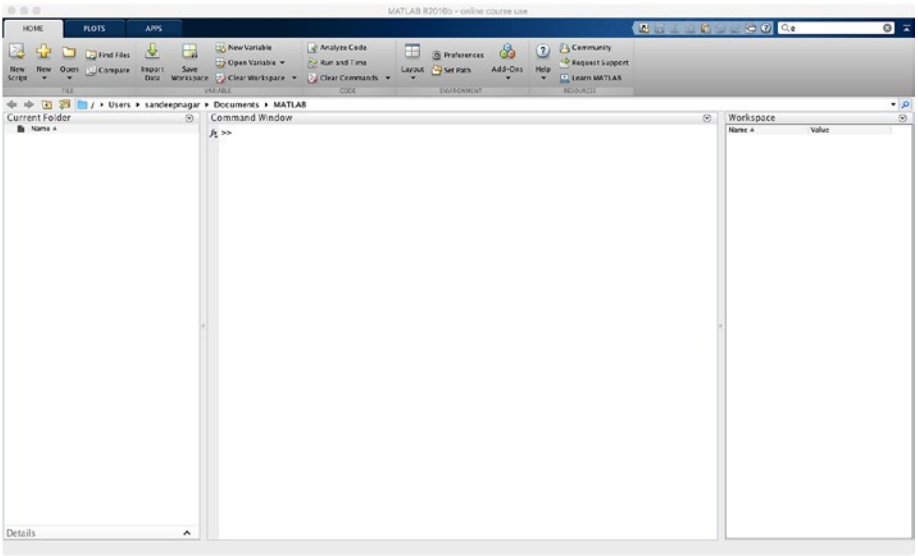

Figure 1-1. *MATLAB in action*

1.4.1 The REPL Principle

The MATLAB command line works on the principle of *REPL*, which stands for Read-Evaluates-Prints-Loop. When input is fed into the MATLAB command prompt, the Julia language:

- *Reads* what the user types
- *Evaluates* what it reads
- *Prints* out the return value after evaluation
- *Loops* back and does it all over again

All MATLAB commands are treated as expressions to be evaluated at REPL. Many programming environments, such as Python's interactive shell as well as the Jupyter notebook format, share the same approach. The new language called Julia also has a REPL and works in a similar fashion.

1.4.2 Calculator

In the simplest view, MATLAB works as a calculator with mathematical operators like multiplication (*), division (/), addition (+), subtraction (-), and exponentiation (^):

```
>> 3 + 5
ans = 8
>> 2 - 3
ans = -1
>> 3.0 * 5
ans = 15
>> 2 / 3
ans = 0.6667
>> format long
>> 2 / 3
ans = 0.666666666666667
>> format short
>> 2 / 3
ans = 0.6667
>> 2 % 3
ans = 2
>> 2 ^ 3
ans = 8
```

As seen in the previous example, when a command is fed into the command prompt >>, it is executed and an answer is given by displaying the results in the next line as `ans =`. To display more decimal digits in the result, you can use the `format long` command. By default, MATLAB works with the `format short` command.

1.4.3 Predefined Constants

```
>> pi
ans = 3.1416
>> i
ans = 0.0000 + 1.0000i
>> j
ans = 0.0000 + 1.0000i
>> Inf/Inf
ans = NaN
```

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

1.4.4 Common Mathematical Functions

```
>> abs(-10.034)
ans = 10.034
>> log10(10)
ans = 1
>> sin(10)
ans = −0.5440
>> cos(10)
ans = −0.8391
>> tan(10)
ans = 0.6484
>> asin(1)
ans = 1.5708
>> asin(10)
ans = 1.5708 + 2.9932i
>> acos(1)
```

```
ans = 0
>> acos(10)
ans = 0.0000 - 2.9932i
>> atan(1)
ans = 0.78540
>> atan(10)
ans = 1.4711
```

A number of predefined mathematical functions exist in MATLAB, including:

- Absolute value: `abs()`
- Logarithm: Natural logarithm `log()` and Base-10 logarithm `log10()`
- Trigonometric functions: `sin()`, `cos()`, and `tan()`. Arguments are taken in radians.
- Inverse-trigonometric functions: `asin()`, `acos()`, and `atan()`

When one works on the command prompt, it is often convenient to have a clear screen by getting rid of the previous command written at the command prompt. This is done using the command `clc`, which *clears the screen* by removing all inputs and outputs.

Complex calculations involving these functions and operations can be performed with ease, like the following

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

and

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

```
>> sqrt(((sin(10))^2)+(cos(10))^2)
ans = 1
>> sin(10)/sqrt(cos(10))
ans = 0.0000 + 0.5939i
```

1.5 Self Learning and Getting Help

Covering all the functions available with MATLAB is beyond the scope of this book (or any other book!). To understand how a particular function needs to be used, you can use the `help` and `doc` commands. For example, typing `help exp` gives you detailed information about how this function should be used, whereas `doc exp` opens the official documentation page for the built-in function, `exp`.

```
>> help exp
exp     Exponential.
exp(X) is the exponential of the elements of X, e to the X.
For complex Z = X+i*Y, exp(Z) = exp(X)*(COS(Y)+i*SIN(Y)).

See also expm1, log, log10, expm, expint.

Reference page for exp
```

Whereas `help` is typically used by programmers to get a quick overview of usage for a particular built-in command, the `doc` is used to learn about MATLAB structures. The `doc` provides detailed descriptions of usage as well as useful examples. For example, typing `doc exp` on the MATLAB command prompt will open a new window, which will show the documentation for using the `exp` facility.

1.6 Variables

To store values temporarily, we use variables that store the value at a particular memory location and address it with a symbol or a set of symbols (called *strings*). For example, you can store the value of `1/10 * pi` as a variable `a` and then use it in an equation like this:

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

To perform this calculation:

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

```
1  >> a=1/10* pi
2  a =  0.3142
3  >> a^2 + 10* sqrt(a)
4  ans = 5.7037
```

Hence, the symbol = works as an assignment operator. It assigns the value on the right side to the variable named on the left side. Multiple assignments can be performed by using the comma (`,`) operator. Also, if you don't want to produce the results on-screen, you can suppress this by using the `;` operator.

```
1  >> a1 = 1, a2 = 10, a3 = 100
2  a1 = 1
3  a2 = 10
4  a3 = 100
5  >> a1 = 1, a2 = 10, a3 = 100;
6  a1 = 1
7  a2 = 10
8  >> a1 = 1; a2 = 10; a3 = 100;
9  >> a1
```

```
10  a1 = 1
11  >> a2
12  a2 = 10
13  >> a3
14  a3 = 100
```

1.6.1 Data Types

While assigning data to a variable, it is important to understand that data can be defined as a variety of *objects* defined by their *data types,* as follows:

- `logical`: This type of data stores Boolean values `1` or `0`, which can be operated by Boolean operators like `AND`, `OR`, `XOR`, etc.
- `char`: This type of data stores alphabetic characters and strings (groups of characters written in a sequence).
- `int8`, `int16`, `int32`, and `int64`: This type of data is stored as integers within 8 bits, 10 bits, 32 bits, and 64 bits, respectively. The size of the integer is given by its bit counts.

  Both `logical` and `char` are one byte (8 bits) wide.
- `uint8`, `uint16`, `uint32`, and `uint64`: This type of data stores unsigned integer data in 8, 16, 32 and 64 bits, respectively.
- `double` and `single`: This type of data is stored as double and single precision floating types, respectively. Decimal numbers are represented by floating point data types. Single precision occupies 4 bytes (32 bits) and double precision occupies 8 bytes (64 bits) to store the floating point numbers.

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

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

Single and double precision matters when the precision of the result matters. In cases like GPS positioning for a projectile flying at high speeds, the results must be as precise as possible for greater accuracy of hit.

- `double complex` and `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.

1.6.2 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 afterward.
- Variable names are case sensitive.
- *Keywords* cannot be used as names.
- Names can include underscores (_).

While naming a variable, if you need to check that the name given is a keyword first, you can use the built-in function called `iskeyword(name)`. Simply typing `iskeyword()` produces a list of keywords, as shown here:

```
>> iskeyword()
ans =

20x1 cell array

'break'
'case'
'catch'
'classdef'
'continue'
'else'
'elseif'
'end'
'for'
'function'
'global'
'if'
'otherwise'
'parfor'
'persistent'
'return'
'spmd'
'switch'
'try'
'while'
```

1.6.3 List of Variables

While working on a project, it is useful to keep track of all the variables used in the project to avoid errors due to duplication of names. You can obtain a list of all variables by using the `who` and `whos` commands. Whereas the command `who` simply presents the list of variables in the workspace, `whos` presents the same with more information, like the size of the variable, the number of bytes used to store the variable, and the variable type.

```
>> who
Your variables are:

a    a1   a2   a3   ans

>> whos
Name        Size            Bytes  Class       Attributes

a           1x1                 8  double
a1          1x1                 8  double
a2          1x1                 8  double
a3          1x1                 8  double
ans        20x1              2462  cell
```

Note that the list of variables produced in this example represents the present state on my computer. If you have been working on projects other than practicing from the present book, all the variables defined in the present session will get reflected when you type `who` or `whos`. By using `who` and `whos`, you can keep track of memory requirements. Remember that judicious use of memory resources is important, especially on Raspberry Pi based systems. To wipe off the stored variables, you can use the `clear` command. It is also important to note that the variables list is session dependent. When you exit the session by closing MATLAB using the icon or using `exit`, the list of variables is erased from memory.

1.6.4 Global and Local Variables

A variable declared globally (i.e., within the main program) is known as a global variable, whereas a variable declared locally within a function (explained in later chapters) is known as a local variable. To define a global variable, you use the `global` declaration statement. Once defined, it remains the same irrespective of any new definition, unless you issue the `clear` command to clear the variable names and values from memory.

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

Global variables are used to define constants during numerical calculations. Suppose you want certain variables to change values, so you could make those unchanging values be global variables by giving the name of your choice. The predefined variables, like `pi`, `e`, etc., are defined in a similar manner.

1.6.5 The clear Command

As seen in the previous section, the `clear` command flushes out the variable names and their values from memory. It proves to be much more useful than that. Whereas `clear all` is the same as `clear`, it can also be used to selectively wipe out variables and their values. Simply type `help clear` to see a detailed view of its use, as shown in Listing 1-1.

Listing 1-1. The help clear Command

```
>> help clear
clear Clear variables and functions from memory.
clear removes all variables from the workspace.
clear VARIABLES does the same thing.
clear GLOBAL removes all global variables.
clear FUNCTIONS removes all compiled MATLAB and
MEX-functions.
Calling clear FUNCTIONS decreases code performance and is
usually unnecessary.
For more information, see the clear Reference page.

clear ALL removes all variables, globals, functions and MEX
links.
clear ALL at the command prompt also clears the base import
list.
Calling clear ALL decreases code performance and is usually
unnecessary.
For more information, see the clear Reference page.

clear IMPORT clears the base import list. It can only be
issued at the
command prompt. It cannot be used in a function or a
script.

clear CLASSES is the same as clear ALL except that class
definitions
are also cleared. If any objects exist outside the
workspace (say in
userdata or persistent in a locked program file) a warning
will be
```

```
21 issued and the class definition will not be cleared.
22 Calling clear CLASSES decreases code per formance and is
   usually unnecessary.
23 If you modify a class definition, MATLAB automatically
   updates it.
24 For more information, see the clear Reference page.
25
26 clear JAVA is the same as clear ALL except that java
   classes on the
27 dynamic java path (defined using JAVACLASSPATH) are also
   cleared.
28
29 clear VAR1 VAR2 ... clears the variables specified. The
   wildcard
30 character '*' can be used to clear variables that match a
   pattern. For
31 instance, clear X* clears all the variables in the current
   workspace
32 that start with X.
33
34 clear -REGEXP PAT1 PAT2 can be used to match all patterns
   using regular
35 expressions. This option only clears variables. For more
   information on
36 using regular expressions, type "doc regexp" at the command
   prompt.
37
38 If X is global, clear X removes X from the current
   workspace, but
39 leaves it accessible to any functions declaring it global.
40 clear GLOBAL -REGEXP PAT removes global variables that
   match regular
```

```
41 expression patterns.
42 Note that to clear specific global variables, the GLOBAL
   option must
43 come first. Otherwise, all global variables will be cleared.
44
45 clear FUN clears the function specified. If FUN has been
   locked by
46 MLOCK it will remain in memory. If FUN is a script or
   function that
47 is currently executing, then it is not cleared. Use a
   partial path
48 (see PARTIALPATH) to distinguish between different
   overloaded versions
49 of FUN. For instance, 'clear inline/display' clears only
   the INLINE
50 method for DISPLAY, leaving any other implementations in
   memory.
51
52 Examples for pattern matching:
53 clear a*                    % Clear variables starting with "a"
54 clear -regexp ^b\d{3} $   % Clear variables starting with
                               "b" and  %    followed by 3
                               digits
55
56 clear -regexp \d          % Clear variables containing any
                              digits
57
58 See also clearvars, who, whos, mlock, munlock, persistent,
   import.
59
60 Reference page for clear
61 >>
```

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

1.7 Summary

MATLAB is a high performance language for technical computing. By using MATLAB as a simple calculator (using numbers and basic operations) as well as a complex calculator (using variables with complex functions), you can perform numerical calculations with ease. The learning curve for MATLAB is quite flat, owing to its simple and intuitive syntax. Whenever you become confused, the documentation for the particular commands can be easily accessed using the `help` command. The MATLAB GUI (Graphic User Interface) also provides an integrated environment for working with many different kinds of computational tasks, as shall be explored in upcoming chapters.

1.8 Bibliography

[1] https://en.wikipedia.org/wiki/List_of_numerical_analysis_software

[2] https://mathworks.com/company/newsletters/articles/the-origins-of-matlab.html

[3] https://www.mathworks.com

[4] https://in.mathworks.com/discovery/numerical-analysis.html

[5] https://mathworks.com/products/

CHAPTER 2

Array Based Computing

2.1 Introduction

Matrices have become an integrated part of numerical computation for dealing with large quantities of data. For a two-dimensional matrix, elements have unique row and column indices through which you can access them. Rows and columns can be attributed to different properties under study. For example, if you measure the temperature and pressure at four corners of the square, the *x, y* coordinates associated with the corner points can be assigned to row and column numbers. Now the experimental data can be simply represented as a matrix. In this way, you can fit data for two properties as a matrix and then use these matrices for numerical calculations.

As an example, suppose an element of a row is defined as 1 if a compound is a conductor, it's 2 if it is a semiconductor, and it's 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 electrical conductivity experiments, this row vector (a 13×1 matrix) can be utilized as input. In this way, you need to model the system in terms of matrix formulation to be solved using MATLAB.

S. Nagar, *Introduction to MATLAB for Engineers and Scientists*,
https://doi.org/10.1007/978-1-4842-3189-0_2

MATLAB defines a data object for dealing with matrices. They are called *arrays*. Using different properties of this object, you can define various kinds of matrices. Built-in functions for matrix operations make it easier for a programmer to deal with large amounts of data by arranging it as a matrix in the desired format and performing array operations. This chapter explores the various options for defining and manipulating arrays.

Since MATLAB was made for matrix manipulation, it has a large set of built-in functions and a robust environment to define and work with matrices.

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*. The following example shows how this can be achieved:

```
>> 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, you can make a desired sized matrix. For example, `matrix22` is a 2×2 and `matrix33` is a 3×3 matrix, whereas `a` is a 1×5 matrix. The first number listed while defining the size indicates the number of rows, whereas the second number indicates the number of columns. It is also important to note that the comma `(,)` operator operates by defining the *next element* in the same row, whereas the semicolon `(;)` operator defines the numbers in the next line/row. A matrix is defined within the brackets of the type `[ ]` (commonly called *square brackets*).

If the number of elements in each row/column do not match, you get 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]
Dimensions of matrices being concatenated are
not consistent.
>> wrong33 = [1,2,3;4,5,6;8,9]
Dimensions of matrices being concatenated are
not consistent.
```

2.3 Creating Arrays from Other Arrays

Multi-dimensional arrays can be created from other multi-dimensional arrays too, as explained here:

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

a =

1     2     3
4     5     6

>> B = [(1:3);(4:6);(7:9)]

```

```
B =

1     2     3
4     5     6
7     8     9

>> c = [a;B]

c =

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

>>>c = [a,B]

Error using horzcat
Dimensions of matrices being concatenated are not
consistent.
```

Here, the matrix a has elements 1, 2, 3 in the first row. Then a row separator (;) defines the next row of elements as 4, 5, 6. Similarly, matrix B has the rows defined by these commands:

- (1 : 3) results in (1, 2, 3)
- (4 : 6) results in (4, 5, 6)
- (7 : 9) results in (178, 9)

Note the MATLAB variable names are case sensitive, so a is not the same as A. A new matrix called c is created by *vertically concatenating* the

matrices a and B. The resultant matrix c is made of elements of a stacked on top of elements of B. The c=[a, B] command yields an error because the dimensions of a and B are not consistent for horizontal concatenation.

Horizontal concatenation can instead be easily performed in the following ways in this example:

```
>> a = 1:3

a =

1     2     3

>> A = [a,a]

A =

1     2     3     1     2     3
```

For multidimensional arrays, use this code:

```
>> a = 1:4

a =

1     2     3     4

>> A = [a;a]

A =

1     2     3     4
1     2     3     4
```

```
>> AA = [A,A]

AA =

1     2     3     4     1     2     3     4
1     2     3     4     1     2     3     4
```

2.3.1 Appending Rows and Columns

When an entire row or column of a matrix needs to be appended, you must consider only one thing—the size of new matrix must match the row and column requirements. As an example, define an array `A, B, D` with sizes (2×2), (1×2), and (2×1), respectively. The row matrix `B` can be inserted as a row of `A` and the column matrix `D` can be inserted as a row of `A`, as shown here:

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

A =

1     2
3     4

>> B = [5,6]

B =

5     6

>> size(A)

```

```
ans =

2     2

>> size(B)

ans =

1     2

>> C = [A;B]

C =

1     2
3     4
5     6

>> size(C)

ans =

3     2

>> D = [5;6]

D =

5
6
```

```
>> size(D)

ans =

2     1

>> E = [A,D]

E =

1     2     5
3     4     6

>> size(E)

ans =

2     3
```

2.3.2 Deleting a Row and/or Column of a Matrix

Rows and columns can be deleted by assigning null matrices [ ] to them. For example, `(1,:)=[]` deletes the first row and `(:,1)=[]` deletes the first column of a matrix, as shown here:

```
>> A = rand(3,3)

A =

0.8147    0.9134    0.2785
0.9058    0.6324    0.5469
0.1270    0.0975    0.9575

```

```
>> A(1,:) =[]

A =

0.9058    0.6324    0.5469
0.1270    0.0975    0.9575

>> A(:,1) =[]

A =

0.6324    0.5469
0.0975    0.9575
```

2.3.3 Concatenation Along a Dimension

Concatenation of two matrices along a dimension can be obtained using `cat(dim, A, B, ...)`, where `dim` presents the dimension and `A` and `B` are the input matrices. Its usage is shown here:

```
>> A = [1,2;3,4]
A =
1    2
3    4
>> B = [5,6;7,8]
B =
5    6
7    8
>> cat(1,A,B)
ans =
1    2
3    4
```

```
13  5    6
14  7    8
15  >> cat(2,A,B)
16  ans =
17  1    2    5    6
18  3    4    7    8
19  >> C = cat(3,A,B)
20  ans(:,:,1) =
21  1    2
22  3    4
23  ans(:,:,2) =
24  5    6
25  7    8
26  >>> size(C)
27  ans =
28  2    2    2
```

When `cat(1,A,B)` is entered at the command prompt, `A` and `B` are concatenated row-wise and `cat(2,A,B)` performs concatenation column-wise. In case of `cat(3,A,B)`, a new matrix is created whose first element of the third dimension is the matrix `A` and the second element is the matrix `B`.

2.3.4 Selecting the Data Type of Elements

Elements of an array can be any data type, as explained in Chapter 1. All elements of an array can be set to a particular data type using the commands shown here:

```
1  >> x = uint32([1,65535])
2  x =
3
4  1x2 uint32 row vector
5
```

```
 6  1   65535
 7
 8  >> x = uint64([1,65535])
 9  x =
10
11  1x2 uint64 row vector
12
13  1   65535
14
15  >> x = int16([1,65535])
16  x =
17
18  1x2 int6 row vector
19
20  1   32767
21
22  >> x = int32([1,65535])
23  x =
24
25  1x2 int32 row vector
26
27  1   65535
28
29  >> x = int64([1,65535])
30  x =
31
32  1x2 int64 row vector
33
34  1   65535
35
```

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

1x2 single row vector

1       65535

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

1       65535

>> x = single([1.0,65535e10])
x =

1x2 single row vector

1.0e+14*

0.0000    6.5535

>> x = double([1.0,65535e10])
x =

1.0e+14*

0.0000    6.5535
```

Line 15 shows that if the element is set to `int16`, then it can store a maximum value of 32767, regardless of being commanded to store a value bigger than that. Hence, it becomes supremely important to understand

the data type of the elements beforehand, in order to avoid errors in numerical calculations. Keep in mind that storing very small numbers in larger numbers of bits is a waste of memory. (Line 46 displays that the number 1, which is stored as a *double precision floating point number,* occupies 64 bits, where essentially 63 bits except the last one are all zeros!)

2.4 Arithmetic Operations on Arrays

Operating on arrays involves two aspects:

- Operating on two or more arrays
- Element-wise operations

All arithmetic operators (such as +, -, *, /, %, ^, etc.) can be used in both cases. When you need to do element-wise operation, then a `.` (dot) is placed before the operator. The element-wise operators become `.+`, `.-`, `.*`, `./`, `.%`, and `.^`. This will become more clear in following example.

```
 1  >> a = [1,2;3,4]
 2  a =
 3
 4  1    2
 5  3    4
 6
 7  >> b = [5,6;7,8]
 8  b =
 9
10  5    6
11  7    8
12
```

```
>> 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, their elements are added/subtracted to elements in the same position. For this reason, the size of the two matrices should be same. On the other hand, when you write `2.+a`, you add the number 2 to each of the elements individually. This can be done regardless of the size and is implemented uniformly on all the elements of the matrix.

2.5 Built-In Functions

A host of built-in functions provide facilities to calculate properties of arrays for quick computation. This includes:

- Summing all elements using `sum()` function.
- Finding the product of all elements of an array using `prod()`.

- Finding the length of array using `length()`.
- Finding the mean of array elements using the `mean()` function.
- Finding the maximum and minimum amongst an element of an array using `max()` and `min()` of an array.
- Finding a particular element as per a logical expression using the `find()` function.
- The rounding elements are as follows:
  - Rounding the elements of an array to the nearest integer toward zero using the `fix()` function.
  - Rounding the elements of an array to the nearest integer toward $-\infty$ using the `floor()` function.
  - Rounding the elements of an array to the nearest integer toward $+\infty$ using the `ceil()` function.
  - Rounding the elements of an array to the nearest integer using the `rounding()` function.
- Sorting the elements of an array using `sort()` in ascending or descending order.

Their usage is demonstrated here:

```
>> A = 1:5
A =
1     2     3     4     5
>> sum(A)
ans =
15
>> prod(A)
ans =
120
```

```
>> length(A)
ans =
5
>> mean(A)
ans =
3
>> max(A)
ans =
5
>> min(A)
ans =
1
>> find(A>4)
ans =
5
>> find(A<4)
ans =
1     2     3
>> A= -1.1:0.5:1.1
A =
-1.1000   -0.6000   -0.1000    0.4000    0.9000
>> fix(A)
ans =
-1     0     0     0     0
>> floor(A)
ans =
-2    -1    -1     0     0
>> ceil(A)
ans =
-1    0      0     1     1
```

```
>> round(A)
ans =
-1   -1    0    0    1
>> A = [2,4.4,2,7,0,-2]
A =
2.0000    4.4000    2.0000    7.0000         0   -2.0000
>> sort(A,'ascend')
ans =
-2.0000         0    2.0000    2.0000    4.4000    7.0000
>> sort(A,'descend')
ans =
7.0000    4.4000    2.0000    2.0000         0   -2.0000
```

2.6 Matrix Algebra

Arithmetic on matrices can be placed into two classes:

- Algebraic operations (covered in Chapter 2)
- Matrix operations

2.6.1 Algebraic Operations on Matrices

Algebraic operations on matrices involve *element-wise operations*. For example:

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

```
>> a+2
ans =
3     4
5     6
7     8
```

Note that a defines a 3×2 matrix so the a+2 command performs element-wise addition of a with a number 2. Computationally, this is done by creating a 3×2 matrix with all its elements as the number 2 and adding them.

Similarly, some other operations are shown here:

```
>> 2*a
ans =
2     4
6     8
10    12
>> 2-a
ans =
1     0
-1    -2
-3    -4
>> a-2
ans =
-1    0
1     2
3     4
>> a/2
ans =
0.5000    1.0000
1.5000    2.0000
2.5000    3.0000
```

The problem starts with other arithmetic operations. For example, when we want to calculate a^2, this would mean multiplying a with itself, i.e., matrix multiplication. This requires either a square matrix or the inner dimensions to be similar because a matrix of dimension $n \times m$ can only be multiplied with `m×t` and the resultant matrix is of the dimension `n×t`. Hence, the command `a^(2)` will result in an error message, as shown here:

```
1  >> a^2
2  Error using ^
3  Input s must be a scalar and a square matrix.
4  To compute elementwise POWER, use POWER (.^) instead.
```

If we wanted to calculate element-wise squares of matrix a then the last line of the `error message` comes to the rescue. Adding a dot operator to a power operator (`.^`) will direct MATLAB to perform the same operation element-wise.

```
1  >> a.^2
2  ans =
3  1    4
4  9   16
5  25  36
```

On the other hand, multiplication of two matrices is the domain of matrix algebra, discussed next.

2.6.2 Matrix Operations on Matrices

Those who are familiar with matrix algebra know that matrix multiplication and division are not straightforward tasks. A `m×n` matrix can only be multiplied by a `n×t` matrix, which results in `a×c` matrix. This is performed by multiplying elements of rows with elements of columns to get new elements.

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

a =

0.8147     0.1270     0.6324
0.9058     0.9134     0.0975

>> b = rand(3,4)

b =

0.2785     0.9649     0.9572     0.1419
0.5469     0.1576     0.4854     0.4218
0.9575     0.9706     0.8003     0.9157
>> c = rand(2,3)

c =

0.7922     0.6557     0.8491
0.9595     0.0357     0.9340

>> a.*c

ans =

0.6454     0.0833     0.5370
0.8691     0.0326     0.0911
```

Here, the matrices a, b, and c are defined using the rand function (which generates uniformly distributed random numbers between 0 and 1).

Now, `a*b` performs matrix multiplication, whereas `a.*c` performs element-wise multiplication. The requirements for both are as follows:

- For matrix multiplication, the inner dimensions must match.
- For element-wise multiplication, all dimensions must match.

Transpose

A single hash mark (`'`), also called an apostrophe, transposes a matrix (rows become columns and vice versa). Performing division on a matrix involves *matrix inversion*.

```
>> a

a =

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

-1.3333    -0.3333    0.6667
1.0833     0.3333   -0.4167
>> b

b =

5    6
7    8
```

```
>> pinv(b)

ans =

-4.0000    3.0000
3.5000   -2.5000
```

Inverse

The inverse of a matrix a, denoted by a^{-1}, is a matrix such that

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

where I is an *identity matrix*. If the given matrix is a square matrix, then the function `inv()` can be used; otherwise, the function `pinv()` is used. Examples are given here:

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

a =

1     2
3     4
5     6

>> pinv(a)

ans =

-1.3333      -0.3333       0.6667
1.0833       0.3333      -0.4167

```

```
>> pinv(a)*a

ans =

1.0000      0.0000
-0.0000     1.0000

>> a = rand(5,5)

a =

0.9649     0.8003     0.9595     0.6787     0.1712
0.1576     0.1419     0.6557     0.7577     0.7060
0.9706     0.4218     0.0357     0.7431     0.0318
0.9572     0.9157     0.8491     0.3922     0.2769
0.4854     0.7922     0.9340     0.6555     0.0462

>> inv(a)

ans =

  2.5545    -0.3119    -0.0173    -0.4492    -1.9962
 -4.9167    -0.1095     0.8740     2.7919     2.5562
  3.3797    -0.1001    -1.3938    -1.5253    -0.8910
 -0.6203     0.4252     0.9340    -1.0120     1.2230
 -2.0554     1.1445     0.1203     2.0420    -0.5531

>> a_pinv(a)

```

```
ans =

 1.0000   -0.0000    0.0000   -0.0000   -0.0000
 0.0000    1.0000   -0.0000   -0.0000    0.0000
-0.0000    0.0000    1.0000    0.0000    0.0000
 0.0000   -0.0000   -0.0000    1.0000   -0.0000
-0.0000    0.0000    0.0000    0.0000    1.0000
```

I is called an identity matrix because all its diagonal elements are 1 and all its non-diagonal elements are zero, which makes its determinant 1. The determinant of a matrix a is calculated using the command det(a). Automatic generation of an identity matrix is done using the command eye(a,b), where a and b are values of the numbers of rows and columns.

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

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

1
>> eye(4,5)
ans =

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

rank()

The rank of a matrix, i.e., the number of linearly independent rows or columns, can be determined by the built-in `rank()` function.

```
a = ones(5,3)

a =

1    1    1
1    1    1
1    1    1
1    1    1
1    1    1

>> rank(a)

ans =

1

>> a = rand(3,2)

a =

0.4456    0.7547
0.6463    0.2760
0.7094    0.6797

>> rank(a)

ans =

2
```

2.6.3 trace()

The sum of the diagonal elements of a matrix is called the *trace* of the matrix. This is given by the built-in `trace()` function, as follows:

```
>> a = ones(4,4)

a =

1    1    1    1
1    1    1    1
1    1    1    1
1    1    1    1

>> trace(a)

ans =

4
```

norm()

The `norm()` function calculates the 2-norm of a matrix, which is equal to the Euclidean length of the vector.

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

A =

1     2
3     4
5     6

```

```
>> norm(A)

ans =

9.5255

>> A = [1,2,3]

A =

1     2     3

>> norm(A)

ans =

3.7417
```

Logical Operations

Two matrices can be compared to each other element-wise.

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

a =

0.6787    0.7431    0.6555
0.7577    0.3922    0.1712

>> b = rand(2,3)

```

```
b =

0.7060    0.2769    0.0971
0.0318    0.0462    0.8235

>> c = (a<b)

c =

2x3 logical array

1   0   0
0   0   1
>> whos
Name      Size            Bytes    Class     Attributes

a         2x3                48    double
b         2x3                48    double
c         2x3                 6    logical
>> a+c

ans =

1.6787   0.7431   0.6555
0.7577   0.3922   1.1712
```

The matrix c has elements, either 1 or 0, which are assigned by determining whether the corresponding elements of a are smaller than b. Note that using whos command, we can probe the variables a, b, and c. The matrix c contains logical data types, i.e., 1 and 0 represent the boolean quantities True and False. But performing a+c treats them as numerals.

This artifact leads to erroneous computations, hence some programming languages like Python explicitly use `True` and `False` representations for boolean values rather than 1 and 0.

2.6.4 Polynomials and Arrays

Every matrix has a characteristic polynomial associated with it. It can be found using the `poly()` function. Let's look at an example:

```
>> A1 = [-3 2 0 4]

A1 =

-3     2     0     4

>> B1 = poly(A1)

B1 =

1    -3    -10    24    0

>> A2 = [1,2;3,4]

A2 =

1    2
3    4

>> B2 = poly(A2)

B =

1.0000    -5.0000    -2.0000
```

In the first case, the resultant polynomial (given by `B1`) is $x^4 - 3x^3 - 10x^2 + 24x$, whereas in the second case (given by `B2`), it's $x^2 - 5x - 2$. The resultant matrix presents the coefficients of the characteristic polynomial.

find()

The built-in function `find()` returns the row and column indices of non-zero entries in a matrix. For example, in the 2×2 matrix defined by `A = [1,0;0,2]`, the non-zero elements exist at `A(1,1)` and `A(2,2)`. The information about rows and columns as a vector is demonstrated here:

```
>> A = [1,0;0,2]

A =

1    0
0    2

>> [row,col,v]=find(A)

row =

1
2


col =

1
2
```

```
v =

 1
 2
```

sort()

The built-in function `sort()` can be used to sort the elements of each column in a particular order. The order can be specified as a second argument to the function as a string (`ascend` or `descend`).

```
>> A = [1,-2,3;4,5,-2;0,-2,3]

A =

 1    -2     3
 4     5    -2
 0    -2     3

>> sort(A)

ans =

 0    -2    -2
 1    -2     3
 4     5     3

>> sort(A,'ascend')

ans =

```

```
21  0    -2    -2
22  1    -2     3
23  4     5     3
24
25  >> sort(A,'descend')
26
27  ans =
28
29  4    5    3
30  1   -2    3
31  0   -2   -2
```

2.7 Random Matrix

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

```
 1  >> rand(4,5)
 2  ans =
 3
 4  Columns 1 through 4
 5
 6  0.8147    0.6324    0.9575    0.9572
 7  0.9058    0.0975    0.9649    0.4854
 8  0.1270    0.2785    0.1576    0.8003
 9  0.9134    0.5469    0.9706    0.1419
10
11  Column 5
12
```

```
13  0.4218
14  0.9157
15  0.7922
16  0.9595
```

Note that the numbers generated here will be different each time even on the 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 using the `rand(a)` command. `help rand` provides a detailed description of various other features and arguments of the random number generator.

To create random integers, you can use `randi()` function. You can also specify a range for these random integers. For example, `randi([1,10],1,5)` will create five random integers (an array of 1× 5) within 1 to 10. On the other hand, `randi([1,10],5)` will create an array of random integers (an array of 5×5) within 1 to 10.

```
 1  >> randi([1,10],5)
 2
 3  ans =
 4
 5  5    3    5    8    10
 6  5    7   10    3     6
 7  7    7    4    6     2
 8  8    2    6    7     2
 9  8    2    3    9     3
10
11  >> randi([1,10],1,5)
12
13  ans =
14
15  9    3    9    3    10
```

A random complex number can be generated using the rand command, as follows:

```
>> rand + i* rand

ans =

0.3500 + 0.1966i

>> rand + i* rand

ans =

0.2511 + 0.6160i
```

Sometimes, you might want to generate the same set of random numbers each time the program executes. This can be done by setting the state of the random number function using the rng command, as follows:

```
>> state 1 = rng;
>> r1 = rand(2,3)

r1 =

0.4733    0.8308    0.5497
0.3517    0.5853    0.9172

>> r12= rand(2,3)

r12=

0.2858    0.7537    0.5678
0.7572    0.3804    0.0759

```

```
>> rng(s);
Undefined function or variable 's'.

>> rng(state 1);
>> r3= rand(2,3)

r3=

0.4733    0.8308    0.5497
0.3517    0.5853    0.9172
```

The state is saved in the `state1` variable and then `r1` and `r2` creates two arrays of 2×3 size. They have different elements. But when the state is reset using `rng(state1)`, the new array of the same size stored in `r3` is exactly the same as `r1`, which was created when the state of the machine was saved in the `state1` variable.

A normally distributed random number generator is given by the function `randn()`. The random numbers, thus generated, are normally distributed around 0. Figure 3-7 in Chapter 3 confirms this fact.

A 3D array of random numbers can be generated by inputting an array for each dimension. For example, if an array `A = [3,2,4]` is fed into the `rand()` function, an 3D array of random numbers is created, as shown here:

```
>> A = [3,2,4];
>> B = rand(A)

B(:,:,1) =

0.7482    0.2290
0.4505    0.9133
0.0838    0.1524


```

```
B(:,:,2) =

0.8258    0.0782
0.5383    0.4427
0.9961    0.1067


B(:,:,3) =

0.9619    0.8173
0.0046    0.8687
0.7749    0.0844


B(:,:,4) =

0.3998    0.4314
0.2599    0.9106
0.8001    0.1818

>> size(B)

ans =

3    2    4
```

2.7.1 Matrix Manipulations

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

2.7.2 Flipping a Matrix

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

2.7.3 Rotating a Matrix

Using the command `rot90(a,n)`, you can rotate a matrix `a 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
```

2.7.4 Reshaping a Matrix

The number of rows and columns in a matrix can be changed provided the total number of elements remains the same.

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

1    2
3    4
5    6

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

1
3
5
2
4
6
>> reshape(a,4,1)
Error using reshape
To RESHAPE the number of elements must not change.
```

2.7.5 Sorting

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

```
>> a = rand(1,5)
a =

0.7431    0.3922    0.6555    0.1712    0.7060

```

```
>> sort(a)
ans =

0.1712    0.3922    0.6555    0.7060    0.7431
```

2.7.6 Upper and Lower Triangular Matrix

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

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

0.0318    0.0971    0.3171
0.2769    0.8235    0.9502
0.0462    0.6948    0.0344

>> tril(a)
ans =

0.0318    0         0
0.2769    0.8235    0
0.0462    0.6948    0.0344

>> triu(a)
ans =

0.0318    0.0971    0.3171
0         0.8235    0.9502
0         0         0.0344
```

2.7.7 Ones and Zeros Matrix

A matrix having all its numbers as 1 or 0 make up a 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
```

2.8 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 =

0.6557    0.8491    0.6787
0.0357    0.9340    0.7577

```

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

0.6557    0.8491    0.6787
0.0357    0.9340    1.0000

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

0         0.8491    0.6787
0.0357    0.9340    1.0000
```

Note that `a(2,3)=1` sets the element at the second row and third column, i.e., number 0.3041072 to 1, and `a(1,1)=0` sets the element at the first row and first column, i.e., number 0.5248873 to 0. To index numbers in a vector, you need a single number.

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

Columns 1 through 7

1    2    3    4    5    6    7

Columns 8 through 9

8    9

>> a(1)
ans =

1
```

```
>> a(-1)
Subscript indices must either be real
positive integers or logicals.
>> a(5)
ans =

5
>> a(10)
Index exceeds matrix dimensions.
```

It is important to note that, unlike some programming languages where indices start at 0, MATLAB starts indices at 1 and does not take negative numbers as indices.

2.8.1 Using Indices to Create a 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 previous example, b is a new vector formed from vector a, where successive elements are made up of elements taken from an index vector [1 3 6 1].

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

```

```
 4  11    12    13
 5  40    50    60
 6  17    18    19
 7
 8  >> a([1,2], [2,3])
 9  ans =
10
11  12    13
12  50    60
```

Note that since the use of the comma operator is optional, we will define vectors and matrices by simply using whitespace.

2.9 Slicing

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

```
 1  >> a = [1 2 3 4 1 3 2 4 6 4 5]
 2  a =
 3
 4  Columns 1 through 7
 5
 6  1    2    3    4    1    3    2
 7
 8  Columns 8 through 11
 9
10  4     6     4     5
11
```

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

1   2   3   4   1

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

1    3    2
```

This is an important feature, as most of experimental calculations would demand filtering the data. Here, a slice of data will be stored separately in a variable and then various mathematical operations can be performed on it.

Now let's try to access slices of a multidimensional array. A matrix a is defined to be a 5×5 matrix.

```
>> a = rand(5,5)

a =

0.6948    0.3816    0.4456    0.6797    0.9597
0.3171    0.7655    0.6463    0.6551    0.3404
0.9502    0.7952    0.7094    0.1626    0.5853
0.0344    0.1869    0.7547    0.1190    0.2238
0.4387    0.4898    0.2760    0.4984    0.7513

>> b = a(1,1)

b =

0.6948
```

```
>> c = a(1,:)

c =

0.6948    0.3816    0.4456    0.6797    0.9597

>> d = a(:,1)

d =

0.6948
0.3171
0.9502
0.0344
0.4387
>> e = a(:)

e =

0.6948
0.3171
0.9502
0.0344
0.4387
0.3816
0.7655
0.7952
0.1869
0.4898
0.4456
0.6463
```

```
47  0.7094
48  0.7547
49  0.2760
50  0.6797
51  0.6551
52  0.1626
53  0.1190
54  0.4984
55  0.9597
56  0.3404
57  0.5853
58  0.2238
59  0.7513
60  >> f = a(:,[1,3])
61
62  f =
63
64  0.6948    0.4456
65  0.3171    0.6463
66  0.9502    0.7094
67  0.0344    0.7547
68  0.4387    0.2760
69  >> g= a([1,3],:)
70
71  g =
72
73  0.6948    0.3816    0.4456    0.6797    0.9597
74  0.9502    0.7952    0.7094    0.1626    0.5853
```

- To access a single element, we use the index value of the row and column, For example, `b = a(1,1)` accesses the element within the first row and first column.

- To access all elements of a row or column, you can use the `:` operator. Hence, `c = a(1,:)` accesses all elements of the first row. Similarly, `>> d = a(:,1)` accesses all elements of the first column. A simple way to remember in words is to read the colon (`:`) as *all elements for* and then the words *n^{th} row/column*, where *n* is a given value.
- Using `a(:)`, you can create a new column matrix that has all the elements.
- A sub-matrix can be accessed by defining *all elements for* column/row and then defining indices in square brackets. For example, `f = a(:,[1,3])` defines a new matrix where elements are composed of *all elements of the first and third columns*. Similarly, `a([1,3],:)` uses all elements of first and third rows.

You can compose complex sub-matrices using this powerful way of defining your choice of elements.

```
>> a = rand(5,6)

a =

0.3510    0.1233    0.9027    0.9001    0.2417    0.9561
0.5132    0.1839    0.9448    0.3692    0.4039    0.5752
0.4018    0.2400    0.4909    0.1112    0.0965    0.0598
0.0760    0.4173    0.4893    0.7803    0.1320    0.2348
0.2399    0.0497    0.3377    0.3897    0.9421    0.3532

>> b = a([2,5],1:3)

```

```
b =

0.5132    0.1839    0.9448
0.2399    0.0497    0.3377
>> c = a(2:5,[1,3])

c =

0.5132    0.9448
0.4018    0.4909
0.0760    0.4893
0.2399    0.3377
>> d = a([2,5],[1,3])

d =

0.5132    0.9448
0.2399    0.3377
>>e = a(2:5,1:3)

e=

0.5132    0.1839    0.9448
0.4018    0.2400    0.4909
0.0760    0.4173    0.4893
0.2399    0.0497    0.3377
```

We define a new 5×5 matrix a and then define a subset of this matrix using `a([2,5],1:3)`, which says that *from the second and third row, take elements from the first column to the third column.* Similarly, `c = a(2:5,[1,3])` creates a matrix using this logic: *from* the first and fifth *column, take elements from the second row to the third row*. Now you can easily guess what `a([2,5],[1,3])`

and `a(2:5,1:3)` should do. It's a good idea to practice slicing of arrays rigorously, as this is one of the most sought-after skills in data cleaning and data analysis in general.

2.10 Automatic Generation of Arrays

MATLAB presents a variety of ways to generate arrays of numbers automatically according to a specified rule. Three methods are discussed in the following sections.

2.10.1 The : Operator

One of the most useful operators in MATLAB, the : operator can be mastered easily. You have already seen its usage in selecting a sub-matrix in Chapter 2.

```
>> help:
:  Colon.
J:K is the same as [J,J+1,...,J+m], where m = fix(K-J). In the
case where both J and K are integers, this is simply
[J,J+1,...,K].
This syntax returns an empty matrix if J>K.

J:I:K is the same as [J,J+I,...,J+m_I], where
m = fix((K-J)/I).
This syntax returns an empty matrix when I == 0,
I>0 and J>K, or
I<0 and J<K.

colon (J,K) is the same as J:K and colon (J,I,K) is the
same as J:I:K.

```

```
13 The colon notation can be used to pick out selected rows,
   columns
14 and elements of vectors, matrices, and arrays. A(:) is all the
15 elements of A, regarded as a single column. On the left
   side of an
16 assignment statement, A(:) fills A, preserving its shape
   from before.
17 A(:,J) is the J-th column of A. A(J:K) is
   [A(J),A(J+1),...,A(K)].
18 A(:,J:K) is [A(:,J),A(:,J+1),...,A(:,K)] and so on.
19
20 The colon notation can be used with acellar ray to produce
   a comma-
21 separated list. C{:} is the same as Cf1g,Cf2g,...,Cfendg.
22 The comma separated list syntax is valid inside () for
   function calls, [] for
23 concatenation and function return arguments, and inside fg
   to produce
24 a cell array. Expressions such as S(:). name produce the
   comma
25 separated list S(1).name,S(2).name,...,S(end). name for the
   structure S.
26
27 For the use of the colon in the FOR statement, See FOR.
28 For the use of the colon in a comma separated list, See
   VARARGIN.
29
30 Reference page for colon
31 Other functions named colon
```

You can generate a series of numbers and store them as arrays by using the `start:step:stop` command.

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

Columns 1 through 7

1    2    3    4    5    6    7

Columns 8 through 10

8    9    10

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

Columns 1 through 7

1    2    3    4    5    6    7

Columns 8 through 10

8    9    10
```

Note that brackets ([ ]) are optional here. If a step is not defined, then it is taken as 1.

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

Columns 1 through 7

```

```
 6  1    2    3    4    5    6    7
 7
 8  Columns 8 through 10
 9
10  8    9    10
11
12  >> a=1:2:10
13
14  a =
15
16  1    3    5    7    9
```

2.10.2 Linearly Spaced Vectors

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

```
 1  >> a = linspace(1,2,5)
 2  a =
 3
 4  Columns 1 through 4
 5
 6  1.0000    1.2500    1.5000    1.7500
 7
 8  Column 5
 9
10  2.0000
11
12  >> a = linspace(1,2,10)
13  a =
14
```

```
15  Columns 1 through 4
16
17  1.0000    1.1111    1.2222    1.3333
18
19  Columns 5 through 8
20
21  1.4444    1.5556    1.6667    1.7778
22
23  Columns 9 through 10
24
25  1.8889    2.0000
```

2.10.3 logspace

Similar to the `linspace` command, `logspace(start, stop, n)` produces *n* numbers from start to stop, which are linearly spaced in logarithmic nature.

```
1  >>> help logspace
2  logspace Logarithmically spaced vector.
3  logspace(X1,X2) generates a row vector of 50
   logarithmically
4  equally spaced points between decades 10^X1 and 10^X2. If X2
5  is pi, then the points are between 10^X1 and pi.
6
7  logspace(X1,X2,N) generates N points.
8  For N = 1, logspace returns 10^X2.
9
10 Class support for inputs X1,X2:
11 float:double, single
12
```

```
13  See also linspace, colon.
14
15  Reference page for logspace
16  >>>logspace (1,5,10)
17
18  ans =
19
20  1.0e+05 *
21
22  Columns 1 through 4
23
24  0.0001    0.0003    0.0008    0.0022
25
26  Columns 5 through 8
27
28  0.0060    0.0167    0.0464    0.1292
29
30  Columns 9 through 10
31
32  0.3594    1.0000
```

2.11 Solving a System of Equations

Solving a system of equations in one line simply involves the \ operator. Suppose the following system of equations needs to be solved:

$$2x - 2y = 4 \qquad \text{(Equation 2-1)}$$

$$-3x + 4y = 9 \qquad \text{(Equation 2-2)}$$

You can define this problem in a matrix, as follows:

$$\begin{bmatrix} 2 & -2 \\ -3 & 4 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4 \\ 9 \end{bmatrix} \quad \text{(Equation 2-3)}$$

Suppose:

$$A = \begin{bmatrix} 2 & -2 \\ -3 & 4 \end{bmatrix} \quad \text{(Equation 2-4)}$$

$$X = \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{(Equation 2-5)}$$

$$B = \begin{bmatrix} 4 \\ 9 \end{bmatrix} \quad \text{(Equation 2-6)}$$

In this way, you can write the following:

$$A \times X = B \quad \text{(Equation 2-7)}$$

The solution is given by $X = A^{-1}B$. You can find the inverse of `A` (using the `inv()` or `pinv()`) function) and then multiply the resultant matrix with the matrix given by `B` to find a solution. Alternatively, you can accomplish this task in just one command, as `A\B`:

```
>> A = [2,-2;-3,4]

A =

2   -2
-3    4

>> B = [4;9]

```

```
B =

4   9

>> C = A/B

ans =

17.0000
15.0000
```

Hence, the solution is $x = 17$ and $y = 15$. Since the elements of the C matrix are solutions, this is often called a solution matrix.

2.12 Eigen Values and Eigen Vectors

The eigenvalue problem is to determine the solution to the equation $Av = \lambda v$, where A is an $n \times n$ matrix, v is a column vector of length n, and λ is a scalar. The values of λ that satisfy the equation are the eigenvalues. The corresponding values of v that satisfy the equation are the right eigenvectors. The left eigenvectors, w, satisfy the equation $w'A = \lambda w'$. The MATLAB function `eig()` returns the eigenvalues and eigenvectors. It also gives the matrix D (diagonal matrix D of eigenvalues), which is related to W and A as $W'A = DW'$:

```
>> A = rand(3,3)

A =

0.6551    0.4984    0.5853
0.1626    0.9597    0.2238
0.1190    0.3404    0.7513
```

```
>> [V,D,W] = eig(A)

V =

-0.7284   -0.9532   0.8945
-0.5300   0.2997    -0.4178
-0.4341   0.0411    0.1590


D =

1.3665        0             0
0             0.4732        0
0             0             0.5264


W =

-0.2724    -0.3066    -0.1266
-0.7915    -0.3145    -0.5186
-0.5471    0.8984     0.8456
```

2.13 Structure Arrays

Arrays stores elements of the same data types, whereas *structure arrays* can store data of different data types. Structures are collections of data organized by named fields. For example, one field may contain textual data, another a number, and a third may be an array, etc. A single structure is a `1-by-1` structure array. Let's understand how to create them by using an example. Let's create a structure array for this book and name this array `book`. Now, various fields can be added using the dot operator, such

as name, author, pages, and chapter. The book array is a 1-by-1 structure with four fields. This is demonstrated here.

```
>> book.name = 'Introducing MATLAB'

book =

struct with fields:

name:'Introducing MATLAB'

>> book.author = 'Sandeep Nagar'

book =

struct with fields:

name:'Introducing MATLAB'
author:'Sandeep Nagar'

>> book.pages = '175'

book =

struct with fields:

name:'Introducing MATLAB'
author:'Sandeep Nagar'
pages:'175'

>> book.chapters = [1 2 3 4 5 6 7]

```

```
30 book =
31
32 struct with fields:
33
34 name:'Introducing MATLAB'
35 author:'Sandeep Nagar'
36 pages:'175'
37 chapters:[1 2 3 4 5 6 7]
```

2.13.1 Defining a New Structure Element Within a Structure Array

A new structure element can be defined within an existing structure array (book, in this example) using index values in the following manner.

```
 1 >> book(2).name = 'Introducing SCILAB'
 2
 3 book =
 4
 5 1x2 struct array with fields:
 6
 7 name
 8 author
 9 pages
10 chapters
11
12 >> book(2).author = 'Sandeep Nagar'
13
14 book =
15
16 1x2 struct array with fields:
17
```

```
18  name
19  author
20  pages
21  chapters
22
23  >> book(2).pages = 175
24
25  book =
26
27  1x2 struct array with fields:
28
29  name
30  author
31  pages
32  chapters
33
34  >> book(2).chapters = [1 2 3 4 5 6 7 8 9]
35
36  book =
37
38  1x2 struct array with fields:
39
40  name
41  author
42  pages
43  chapters
```

In this way, the book is now a 1×2 structure array. All structures in a structure array have the same number of fields and all fields have the same number of field names. When the name of the structure array is entered at the command prompt, the summary of information and fields is displayed.

The `fieldnames()` function can be used to get a *cell array* having information about the fields. This is demonstrated in the following code.

```
>> book

book =

1x2 struct array with fields:

name
author
pages
chapters

>> fieldnames(book)

ans =

4x1 cell array

'name'
'author'
'pages'
'chapters'
```

While expanding a structure array, it is not mandatory to fill in all the fields. Fields that are not associated with values are left empty.

2.13.2 Adding and Removing Fields

A new field can be added at any point to a single structure. For example, let's add the field `publisher` to the structure `book`, as demonstrated here.

```
>> book(2).publisher = 'Apress'

book =

1x2 struct array with fields:

name
author
pages
chapters
publisher

>> book

book =

1x2 struct array with fields:

name
author
pages
chapters
publisher

>> book = rmfield(book,'publisher')

book =

1x2 struct array with fields:

```

```
name
author
pages
chapters
```

To remove a field, say `publisher`, from the structure `book`, you can use the `rmfield()` function, as demonstrated.

2.13.3 struct()

The function `struct()` can also be used to define a structured array with the syntax shown in the following code:

```
>> book1 = struct('name','Introducing MATLAB','author',
'Sandeep Nagar','pages',175,'chapters',[1, 2, 3, 4, 5, 6, 7])

book1 =

struct with fields:

name:'Introducing MATLAB'
author:'Sandeep Nagar'
pages:175
chapters:[1 2 3 4 5 6 7]

>> book1(2) = struct('name','Introducing python','author',
'Sandeep Nagar','pages',175,'chapters',[1, 2, 3, 4, 5, 6,
7, 8, 9])

book1 =

```

```
16  1x2 struct array with fields:
17
18  name
19  author
20  pages
21  chapters
```

A new structure named `book1` is created where *field names* and *values* are filled in successively. It can be expanded using the index number in a similar fashion, making it a `1-by-2` structure array.

A structure array may contain another structure or even a structure array as its fields. These are called nested array. This is demonstrated here, where `book1` (a structure array defined previously) is added as a new field to the structure array `book`.

```
 1  >> book(3).linked book = book1
 2
 3  book =
 4
 5  1x3 struct array with fields:
 6
 7  name
 8  author
 9  pages
10  chapters
11  linked_book
```

2.14 Getting Data from a Structure Array

Data *values* can be assigned from a structure array using index numbers, as demonstrated next. Here, `info1` stores the value of the field `name` for the second structure (signified by the syntax `book(2)`). In a similar fashion,

info2 stores the value of the field name for the first structure (signified by the syntax book(1)). The variable info3 extracts the third element of the field chapter from the second structure of the structure array book.

```
>> info1 = book(2).name()

info1 =

'Introducing SCILAB'

>> info2 = book(1).name()

info1 =

'Introducing MATLAB'

info3 = book(2).chapters(3)

info3 =

3
```

2.15 Cell Arrays

Cell arrays are arrays of cells where each cell stores an array. Within a cell, elements must be the same type (because cells store arrays), but two cells may have different types. For example, suppose you have three arrays—array1 (stores numerical values), array2 (stores textual values), and array2 (stores numerical values). You can then construct a cell array using these three arrays. The elements of this cell array store different types of arrays, but each element stores just one type of data.

2.15.1 Creating Cell Arrays

The `cell(m,n)` function makes an *empty* cell array of the size $m - by - n$. By assigning data values to this empty cell array, it can then be constructed as desired, *one cell at a time*. Let's first create an empty cell array, referenced by a variable, say `a`. There are two ways to assign the data:

- *Cell indexing*: Cell indices are mentioned within parentheses `()` and cell contents are mentioned within brackets `{}` on either side of assignment operator, like so:

```
>> a = cell(3,3)

a =

3x3 cell array

[]    []    []
[]    []    []
[]    []    []

>> a(1,1) = ([1,2,3]);
>> a(1,2) = (['a','b']);
>> a(1,3) = ("Sandeep");
>> a(2,3) = ([1.5,-2]);
>> a(2,2) = ([-200]);
>> a(2,1) = (["Nagar"]);
>> a(3,1) = ([-10,-15.5,5.3]);
>> a(3,2) = (["Hello"]);
>> a(3,3) = (["World"])

a =

```

```
23  3x3 cell array
24
25  [1x3 double]    'ab'           ["Sandeep"]
26  ["Nagar"]       [-200]         [1x2 double]
27  [1x3 double]    ["Hello"]      ["World"]
28
```

- *Content indexing*: Here, brackets/parentheses are used in reverse fashion, i.e., () for content and [ ] for indices.

```
 1  >> a = cell(3,3)
 2
 3  a =
 4
 5  3x3 cell array
 6
 7  []    []    []
 8  []    []    []
 9  []    []    []
10
11  >> a {1,1} = ([1,2,3]);
12  >> a {1,2} = (['a','b']);
13  >> a {1,3} = ("Sandeep");
14  >> a {2,3} = ([1.5,-2]);
15  >> a {2,2} = ([-200]);
16  >> a {2,1} = (["Nagar"]);
17  >> a {3,1} = ([-10,-15.5,5.3]);
18  >> a {3,2} = (["Hello"]);
19  >> a {3,3} = (["World"])
20
```

```
21  a =
22
23  3x3 cell array
24
25  [1x3 double]    'ab'          ["Sandeep"]
26  ["Nagar"]       [-200]        [1x2 double]
27  [1x3 double]    ["Hello"]     ["World"]
28
```

2.15.2 The celldisp() and cellplot() Functions

The constructed cell arrays can be displayed by using two functions called `celldisp()` and `cellplot()`. The `celldisp()` command displays the full cell contents, whereas `cellplot()` displays a graphical display of the cell architecture. See Figure 2-1.

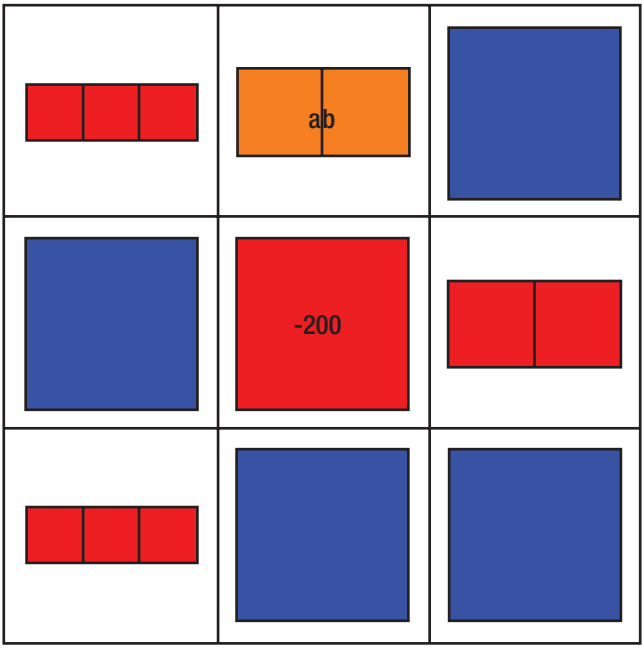

Figure 2-1. *Output of cellplot (a)*

2.15.3 The cell2struct(), num2cell(), and struct2cell() Functions

The `cell2struct()` command can be used to convert a cell array to a structure. Similarly, `num2cell()` can be used to convert a numeric array into a cell array and `struct2cell()` can be used to convert a structure into a cell array.

2.16 Summary

Array based computing lies at the very heart of modern computational techniques. MATLAB presents a very suitable platform to perform this technique with ease. A variety of predefined functions enable users to save time while prototyping a problem. Having flexible methods to define multidimensional arrays and perform fast computation is the necessity of our times. Most of the time spent on a simulation is either in loops or in array operations. Predefined array operations have been optimized with algorithms for reliability, time savings, and efficient memory management.

CHAPTER 3

Plotting

3.1 Introduction

Without visualization, numerical computations are difficult to judge. Producing publication quality images of complex plots that provide meaningful analysis of numerical results has been a challenge for scientists all over the world. Many commercial software programs have been very successful in satisfying this need. MATLAB also provides this facility. Its plotting features include choosing from various types of plots in 2D and 3D, enhancing plots with additional information like titles, labeled axes, grids, and labels for data, and writing equations and other important information about the data. The most important feature is that plots can be defined in a programmatic manner, i.e., you can enter the data for a plot using a computer program. This is quite different than entering the data by hand.

One of the advantages is that you can define the data using a variety of functions. The other advantage is when the data has a huge number of entries, it can be entered according to the rules that govern the computer program. The following sections 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 you an intuitive understanding of the problem under consideration.

S. Nagar, *Introduction to MATLAB for Engineers and Scientists*,
https://doi.org/10.1007/978-1-4842-3189-0_3

3.1.1 2D Plotting

plot(x,y)

Since you need data on two axes to be plotted, you first need to create the plots. Assume for this example that the x axis has 100 linearly spaced data points on which $y = x^2$ is defined. See Figure 3-1.

```
>> x = linspace(0,100,100);
>> y=x.^2;
>> plot(x,y)
>> xlabel('x')
>> ylabel('y=x^{2}')
>> title('Plot of x versus y=x^{2}')
```

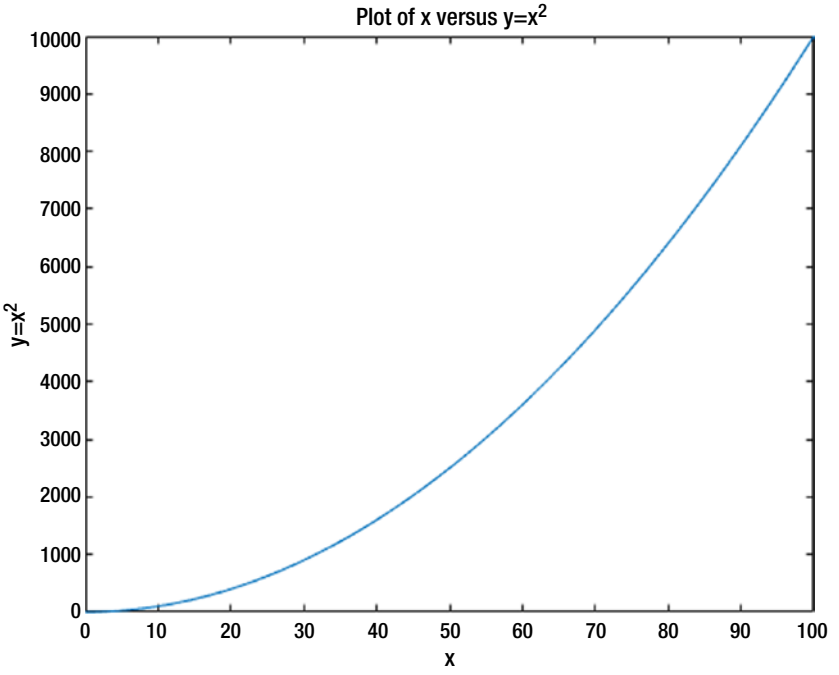

Figure 3-1. *The $y = x^2$ plot*

First we define a variable x and placed 100 equally spaced data points from 0 to 100. This create a 1×100 matrix. Using the scalar operation of exponentiation, we define a variable y as $y = x^2$. Then we use the `plot()` function, which takes two arguments as the x-axis and y-axis data points. Typing `help plot` on the command prompt gives useful insight into this wonderful function written to plot two dimensional data.

The x and y axis labels can be placed using the `xlabel()` and `ylabel()` functions, which take a string as input. The string can be formatted with LATEX commands—for example, x^2 can be printed by using the `x^{2}` syntax. Similarly, the title can be added for the graph with an appropriate string.

area()

The `area()` function creates a similar plot as `plot()`, but it also shades the area under the curve, as shown in Figure 3-2.

```
1  >> x = linspace(0,100,100);
2  >> y = x.^2;
3  >> area(x,y)
```

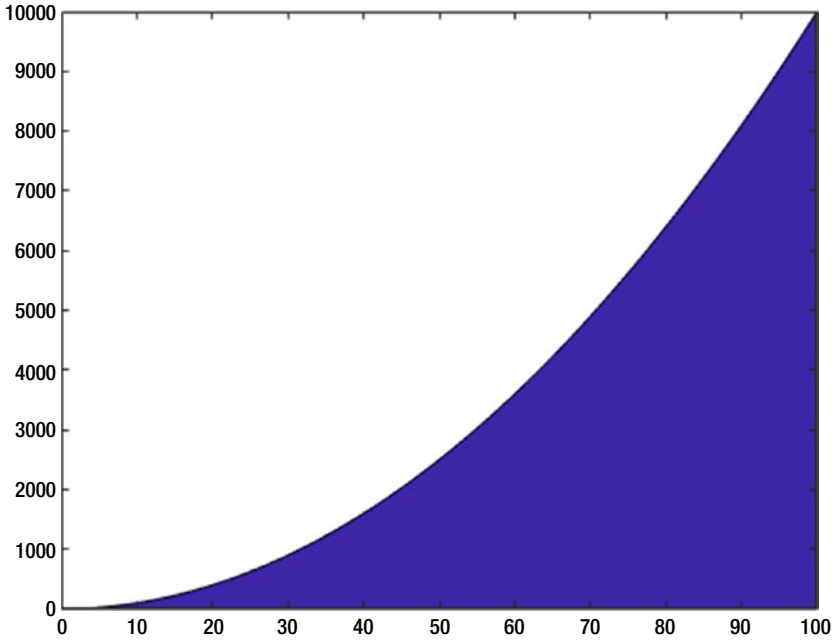

Figure 3-2. *The $y = x^2$ plot created with the area() function*

Plotting Multiple Plots on the Same Graph

You can plot multiple plots on the same graph by simply supplying x and y axes vectors, as shown in Listing 3-1. Figure 3-3 shows the result.

Listing 3-1. The multi.m Program

```
clear all;
clf;
x = linspace(1,100,100);
y1 = x.^2.0;
y2 = x.^2.1;
y3 = x.^2.2;
y4 = x.^2.3;
plot (x,y1,"@12",x,y2,x,y3,"4",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 '+'
```

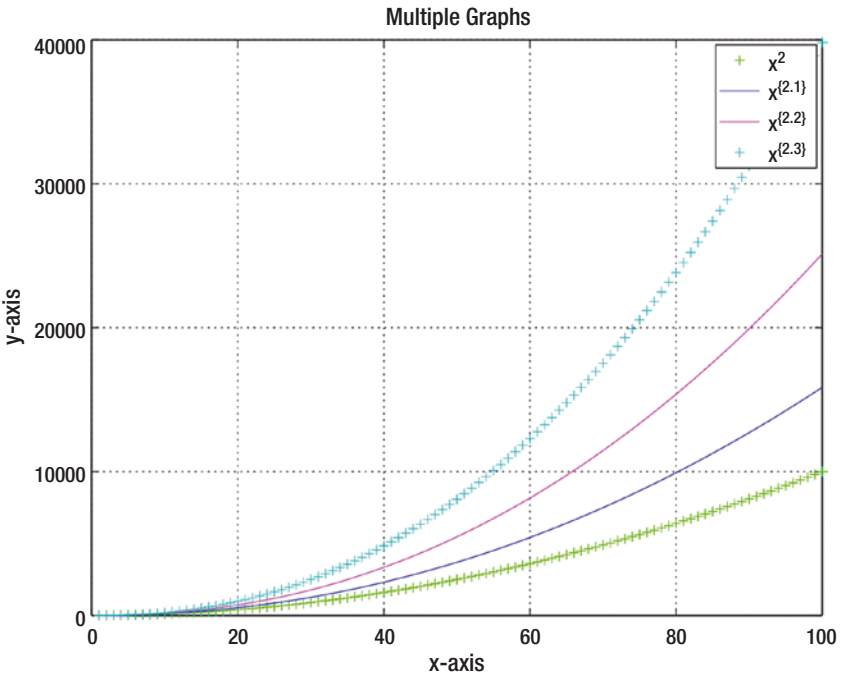

Figure 3-3. *Multiple plots within the same figure*

Explanations of the line numbers in the previous code are as follows:

- `clear all` clears the variable names and values from memory (line 1).
- `clf` clears any current figure window (line 2).
- `x = linspace(1,100,100)` creates a vector `x` made up of 100 equally spaced data points between 1 and 100 (line 3).
- y1 = x.$^{2.0}$; creates a new vector named `y1` having element-wise square of vector `x` (line 4).
- y1 = x.$^{2.1}$; creates a new vector named `y2` having element-wise exponentiation by 2.1 of vector `x` (line 5).
- y2 = x.$^{2.2}$; creates a new vector named `y3` having element-wise exponentiation by 2.2 of vector `x` (line 6).
- y3 = x.$^{2.3}$; creates a new vector named `y3` having element-wise exponentiation by 2.3 of vector `x` (line 7).
- y4 = x.$^{2.4}$; creates a new vector named `y4` having element-wise exponentiation by 2.4 of vector `x` (line 8).
- Plots as per comment given in lines 15, 16, 17 (line 9).
- The grid is turned on for the figure (line 10).
- `xlable` takes the value of string `x-axis` (line 11).
- `ylabel` takes the value of string `y-axis` (line 12).
- `title` takes the value of string `Multiple Graphs` (line 13).

Figure 3-3 is obtained by running the code. These types of plots are used to check the variation of results by varying a particular parameter.

Plotting Multiple Plots Separately

The `subplot(row,coloumn,index)` command is used to plot multiple plots on the same figure, but in separate views. `subplot(2,2,4)` means that the plot will be on the second row, the second column, and the fourth index. See Listing 3-2.

Listing 3-2. The multiSubplot.m Program

```
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 '+'
```

As shown in Figure 3-4, plots are organized as matrixes, where the row number and column number dictate its position. An index of the plot can then be used for further processing as a graphical object. There are

many commands for controlling font size, tick labels and fonts, as well as for inserting mathematical equations. You can view them by typing `help plot` or reading the documentation of this function. Ample examples can be obtained from the web. This function is used frequently, so you need to have good command over its use.

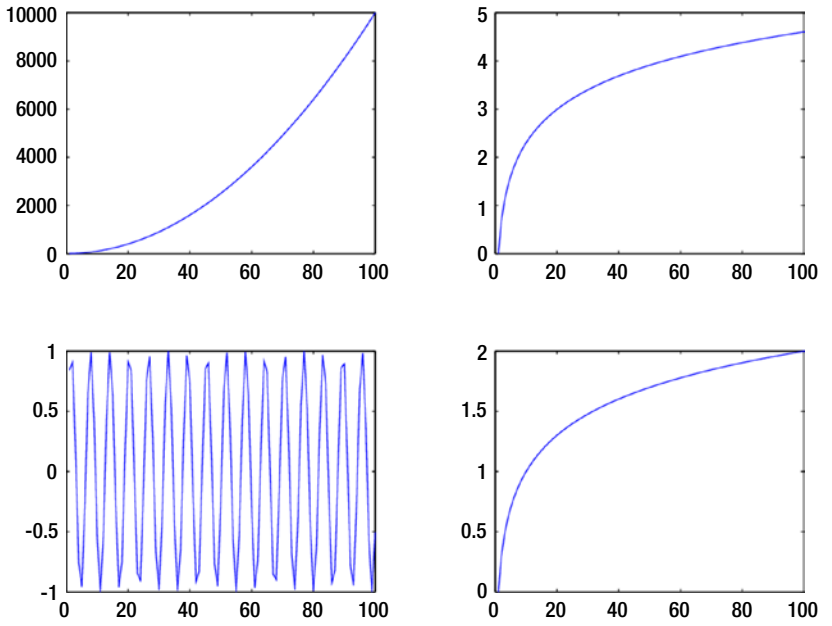

Figure 3-4. *Separate multiple plots within the same figure*

3.1.2 The bar(), barh(), and hist() Commands

Bar charts are a primitive but very effective visualization of primary statistical information. There are three options for plotting bar charts and histograms.

```
1  >> x = [1,2,3,4,5];
2  >> y = [12,5,11,9,10];
3  >> bar(x,y)
```

```
>> x = [1,2,3,4,5];
>> y = [12,5,11,9,10];
>> barh(x,y)
```

Figure 3-5 shows a standard bar chart and Figure 3-6 shows a horizontal bar chart.

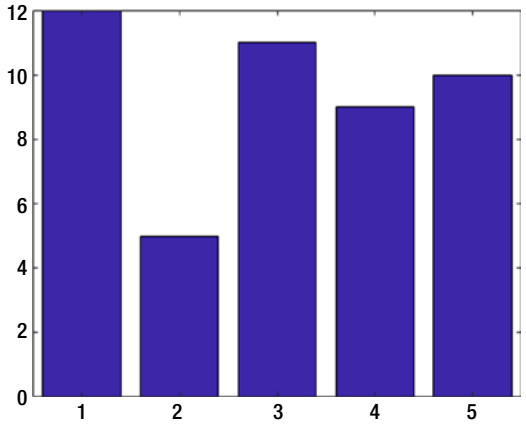

Figure 3-5. *A bar chart*

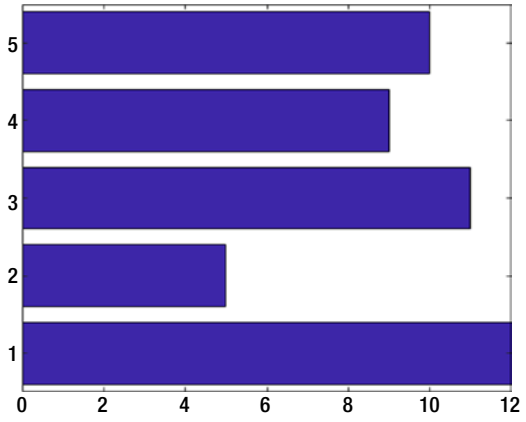

Figure 3-6. *A horizontal bar chart*

A histogram can be plotted using the `hist()` function in a similar fashion. Let's look at the behavior of a normalized distribution of random numbers generated by the `randn()` function.

```
1  >> x = randn(100);
2  >> hist(x)
```

In Figure 3-7, you can clearly observe the bell-shaped curve of the envelope to confirm that the random numbers are indeed normally distributed.

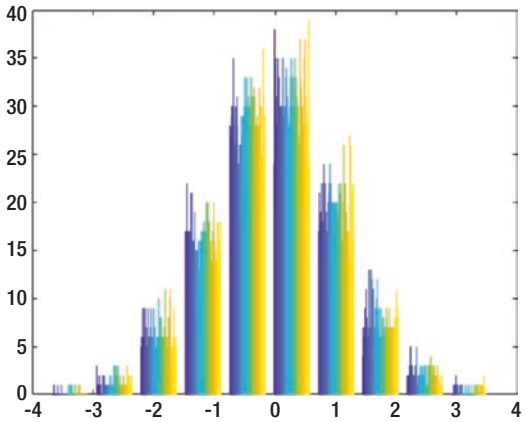

Figure 3-7. *A histogram showing normalized distribution of random numbers*

Logarithmic Plots

For plotting graphs involving logarithmic scale, MATLAB provides three options:

- `semilogx()`: Plots with a logarithmically spaced x-axis. As an example, consider the code `log1a.m` shown in Listing 3-3, which produces the plot shown in Figure 3-8.

Listing 3-3. The log1a.m Program

```
%MATLAB program to illustrate
%usage of semilogx() and
%semilogy() and loglog() command

%semilogx()
y = 0:2:50;
x = exp(-y/2);
subplot(3,1,1)
semilogx(x,y)
grid on
xlabel('x=e^{y/2}');
ylabel('y');
title('Using semilogx() command');

%semilogy()
x1 = 0:2:50;
y1 = exp(-x1/2);
subplot (3,1,2)
semilogy (x1,y1)
grid on
xlabel('y=e^{x/2}');
ylabel('x');
title('Using semilogy() command');

%loglog()
x2 = 0:2:50;
y1 = exp(x2);
y2 = exp(x2/2);
subplot(3,1,3)
loglog(x1,y1)
```

```
31  grid on
32  xlabel('y1=e^{x}');
33  ylabel('y2=e^{x/2}');
34  title('Using loglog() command');
```

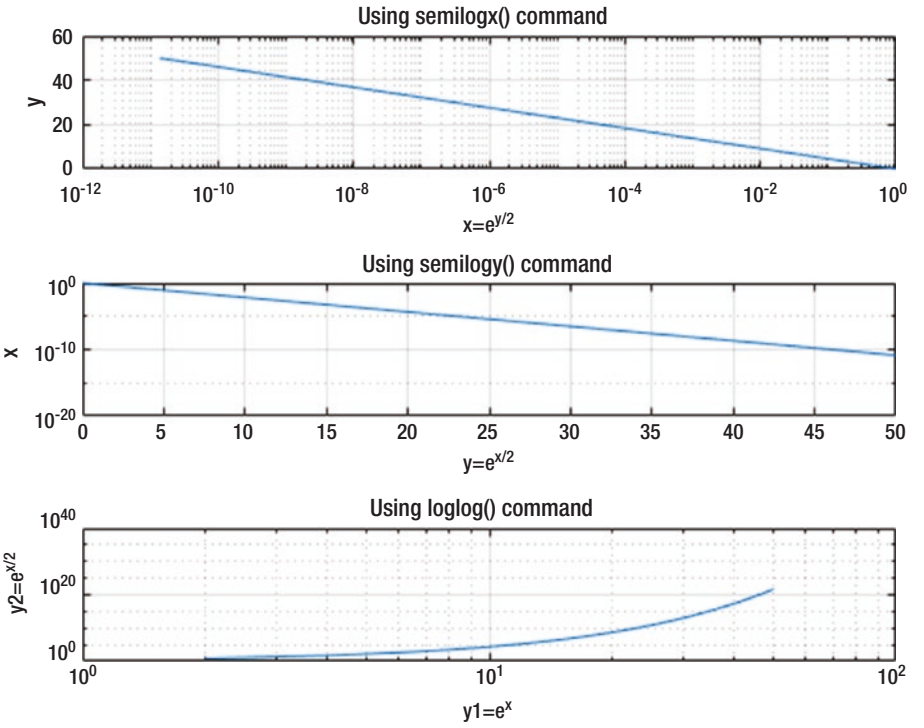

Figure 3-8. *Describing usage of semilogx()*

- `semilogy()`: Similarly, `semilogy()` plots a logarithmically spaced y-axis.
- `loglog()`: Plots with both axes logarithmically spaced.

Polar Plots

Sometimes you'll prefer to plot in polar coordinates, rather than Cartesian ones. Then, instead of *x and y,* the coordinates are *r and θ*. See Listing 3-4.

Listing 3-4. The CoordinatesPolar.m Program

```
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];
Polar Graph = polar(theta,r,"*");
set(Polar Graph,"LineWidth",2);
legend("spiral","circle","heart","Rose");
```

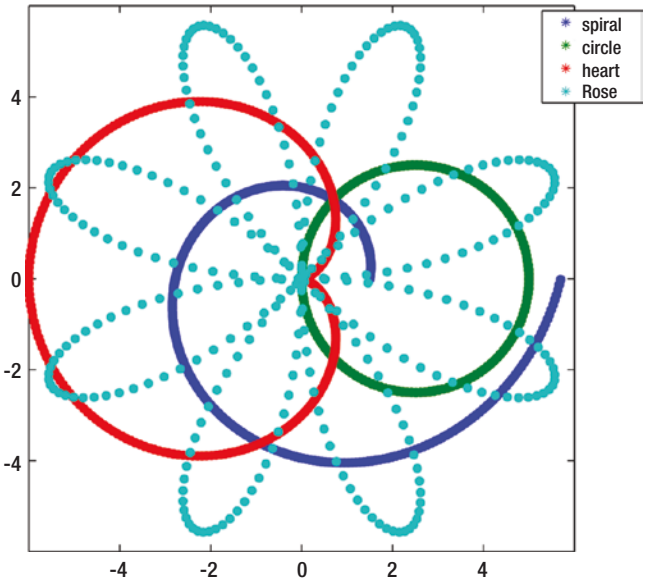

Figure 3-9. *Polar graph*

Figure 3-9 shows an example of a polar graph for the code given in the `CoordinatesPolar.m` example. Explanation of the program follows (according to the line number):

- A variable named `theta` representing θ is defined by points starting from 0 to 2π, with steps of 0.02 (line 1).
- A variable named `a1` representing r for *spiral* is calculated using the following equation (line 2).

$$r = 1.5(\theta)$$

- A variable named `a2` representing r for *circle* is calculated using the following equation (line 3).

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

- A variable named `a3` representing r for *heart* is calculated using the equation (line 4).

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

- A variable named `a4` representing r for *rose* is calculated using the equation (line 5).

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

- A variable named `r` stores all the r calculated using equations as a column vector (line 6).
- A variable named `PolarGraph` stores the values produced by the function `polar()`, which takes θ, *r* as arguments and `"*"` for the type of marker (line 7).
- The `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 this case, the property named `LineWidth` is set to 2 (line 8).

- The `legend()` function sets four legends in the same order that the polar function takes them from vector `r` (line 9).

The rose() Function

The `rose()` function draws an angled histogram, i.e., a polar histogram. The input should be a vector of numbers. Let's look at the usage by constructing a vector of 100 random numbers using `randn(100,1)*pi` and then feeding it to the `rose()` function. The resultant plot is shown in Figure 3-10.

```
1  >> x = randn(100,1)*pi;
2  >> rose(x)
```

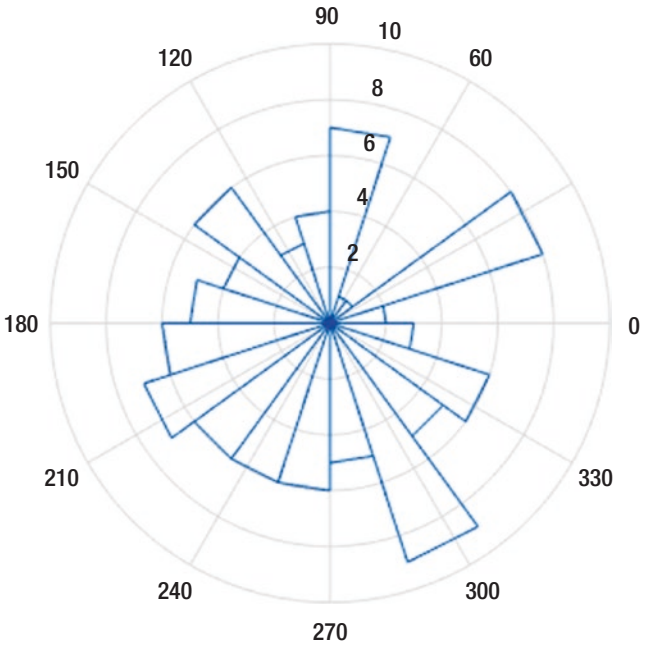

Figure 3-10. *Plot of random numbers by the rose() function*

pie()

A pie chart can be created using the `pie()` function. This provides a very powerful tool to visualize the parts of a whole. The usage is explained in the following code and graphs are shown in Figure 3-11. This function supports 34 items, which are distributed such that `a,b,c,d,e,f` get `4,7,2,8,4` and `9` parts. The pie chart can be made by first defining the parts as an array, then defining the labels as an array. Then if the `pie()` function is fed directly, you see a color coded exploded pie chart showing the percentages of each part. When a `show()` array is also entered, it explodes only those parts where the value of the corresponding element is 1.

```
1  >> x = [4,7,2,8,4,9];
2  >> subplot(2,1,1)
3  >> pie(x)
4  >> subplot(2,1,2)
5  >> show = [1,0,0,1,0,1];
6  >> pie(x,show,labels)
7  >>
```

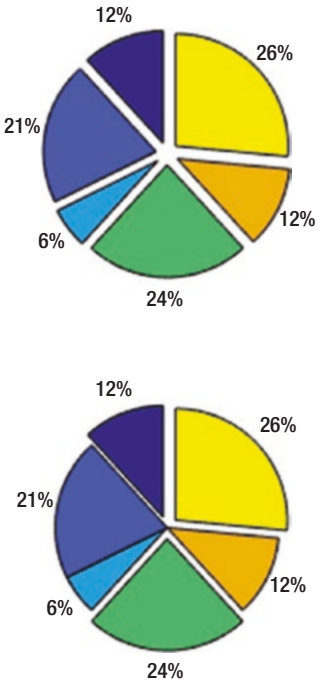

Figure 3-11. *Pie plots with all parts exploded and with some parts exploded*

stairs()

A staircase graph draws a stair-step graph for elements of a vector. Consider an example of plotting $y = x^{2.5}$, where x is a vector of 100 elements from $-\pi$ to π. As shown in Figure 3-12, the data points are connected in a stair-step fashion.

```
>> x = -pi:pi:100;
>> y = x.^(2.5);
>> stairs(y)
```

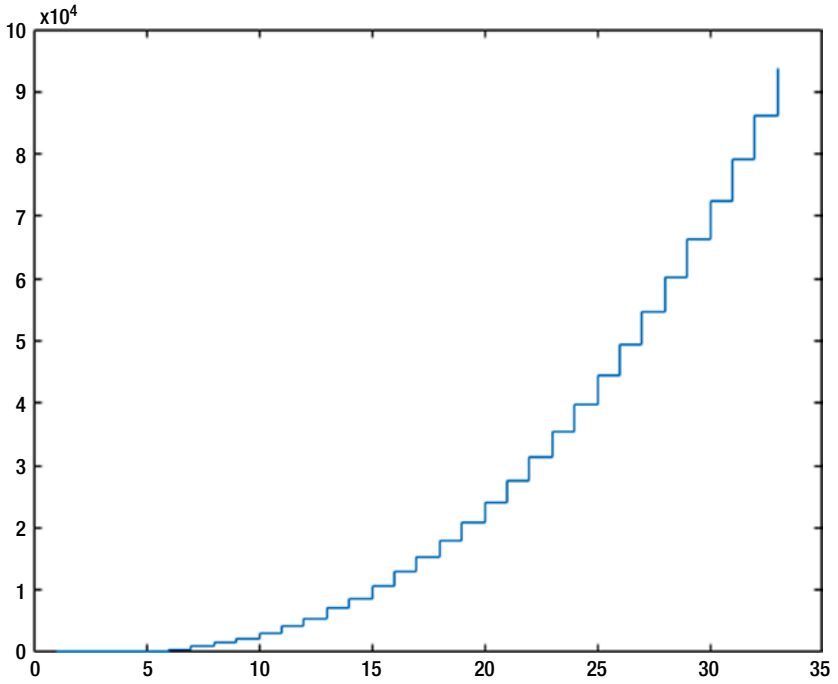

Figure 3-12. *Stair plot for* $y = x^{2.5}$

stem()

Stem plots draw data points as stems that extend from equally spaced values. Sample code for plotting $y = cos(x)\ \epsilon\ (-\pi, \pi)$ is presented here and it produces the graph shown in Figure 3-13.

```
1  >> x = −2*pi:2*pi;
2  >> y = cos(x);
3  >> stem(y)
```

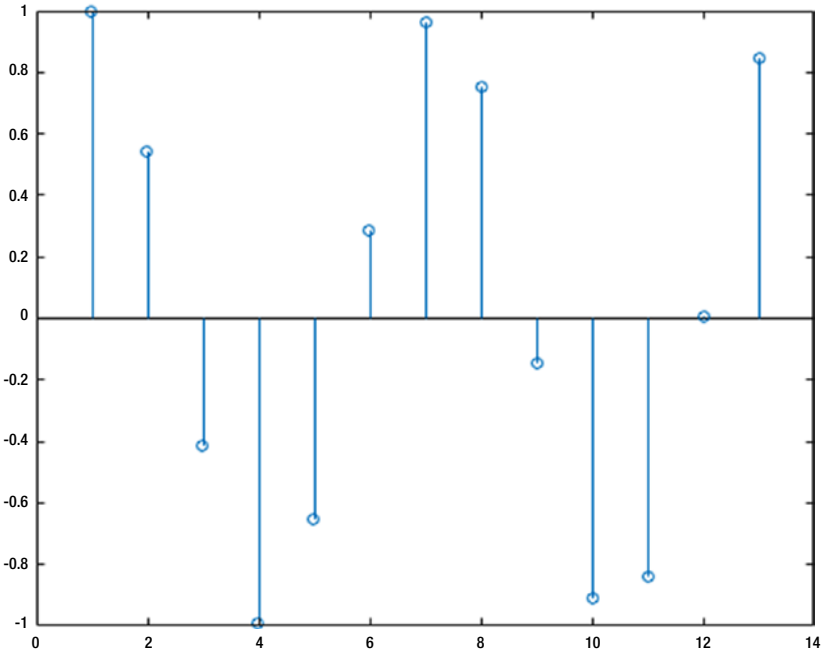

Figure 3-13. *Stem plot for y = cos(x) ϵ (−π, π)*

3.1.3 3D Plotting

There are various functions available for 3D plotting in MATLAB. Your choice of function depends on the particular problem.

mesh

Consider the `mesh` command shown in Listing 3-5. It produces the graph shown in Figure 3-14.

Listing 3-5. The ThreeDMesh.m Program

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

```
4  d = sin(c)./c;
5  mesh(a,b,d);
```

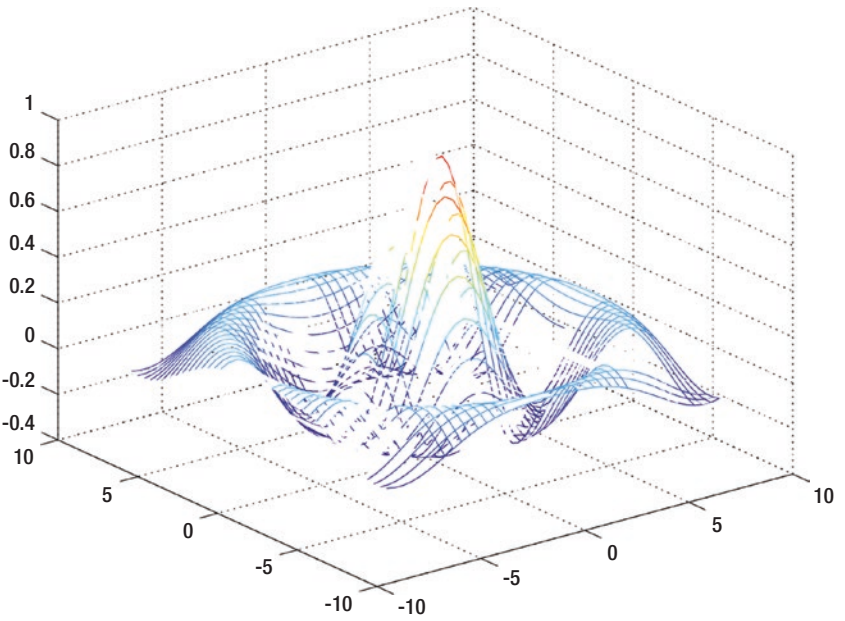

Figure 3-14. *3D meshing*

It's important to note that this code uses a new function named `meshgrid`. Do a quick search of it using `help meshgrid`.

`meshgrid` is used as follows:

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

Two variables are created, namely `a` and `b`, and they store 41 linearly spaced data points between −8 to 8, as a row vector. These two row vectors (both 1×41 in dimension) are passed as arguments for the function `meshgrid`, which gives two outputs: `xx` and `yy`. These are 41×41 dimensioned matrices where rows of `xx` are copies of `a` and columns of `yy` are copies of `b`.

`meshgrid` can also take a 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 the `a` and `b` vectors. In this case, the function is defined as follows:

$$c = \sqrt{x^2 + y^2} \qquad \text{(Equation 3-1)}$$

and

$$d = \frac{sin(c)}{c} \qquad \text{(Equation 3-2)}$$

Note Function `eps` produces a very small number ($2.2204.10^{-16}$ defined by machine precision [1]. It is widely used in numerical computation, where zero needs to be avoided, especially in 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 Step 3 is then used under division as a denominator in Step 4).

Continuing now with the plotting exercise, new arrays can be used to plot by applying the 3D plotting function `mesh()`, which takes these two arrays `a` and `d` as its arguments, resulting in Figure 3-14. If `mesh(x,y,z)` is used then a wire-frame mesh made up of rectangles is created. The vertices of the rectangles are made of data points generated by the function (in this case, Equations 3-1 and 3-2). The (*x, y*) coordinates of vertices are given by the `xx` and `yy` matrices, since the *x* coordinate comes from the `xx` matrix and the *y* coordinate comes from the `yy` matrix. `z` determines the height above the plane of each vertex.

In this way, a 3D plot is created. 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, the rectangles can be smaller, if possible. There are some other

variations of the same function, such as `ezmesh`, `meshc`, and `meshz`. A simple `help` command can be very useful to determine which one suits a particular problem best. 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 (type `help colormap` to learn more).

meshc

`meshc()` generates a 3D rectangulated mesh as well as a contour at the base. As shown in Figure 3-15, apart from producing a 3D plot for a given function, you also get a contour plot. Note that at this time, the equation working on matrices is written as an argument of the `meshc()` function, thus making the program even smaller. See Listing 3-6.

Listing 3-6. The ThreeDMeshc.m Program

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

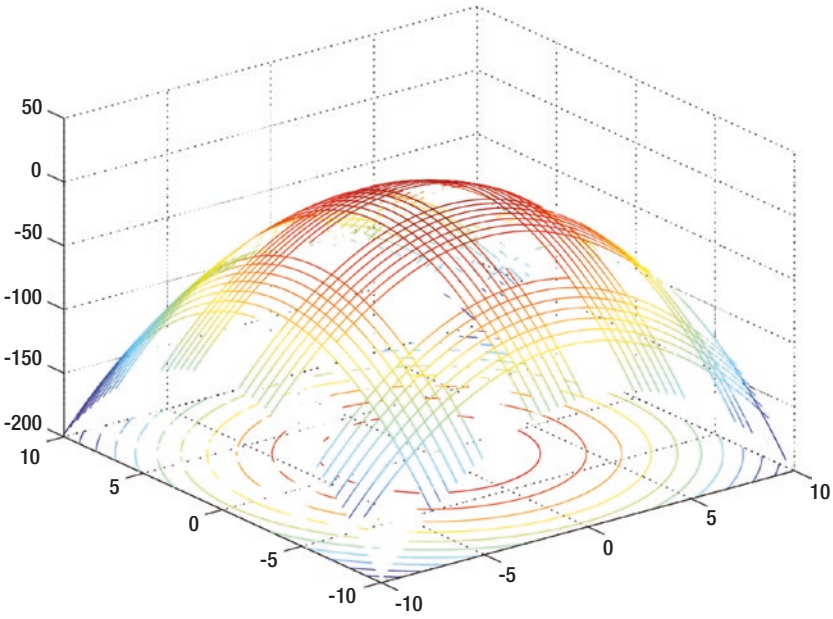

Figure 3-15. *3D meshing with the meshc() function*

surf()

`surf()` generates a surface plot where the wire mesh is simply filled up at the empty points, as shown in Figure 3-16. See Listing 3-7.

Listing 3-7. The ThreeDsurf.m Program

```
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);
```

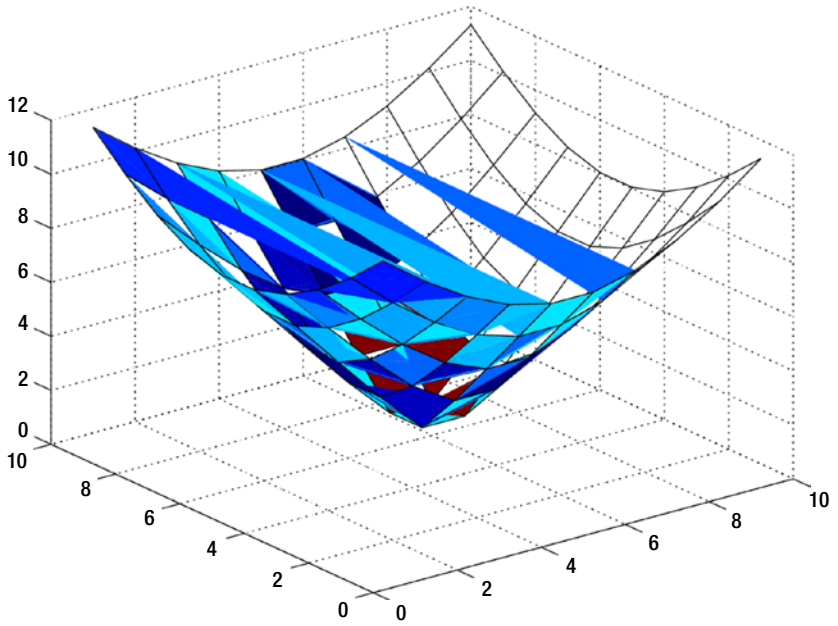

Figure 3-16. *3D meshing with the surf() function*

3.2 Summary

A rich library of plotting functions makes MATLAB a suitable choice for plotting data in a variety of publication-ready formats. Together with commands to access system files and folders, these plots can be directed to be saved at appropriate places for creating a suitable report. Plotting in 3D and viewing at different angles is quite intuitive in MATLAB. Hence, MATLAB is a suitable method to visualize data. The limitations of data and the speed of execution depend on the computer storage and RAM on the motherboard.

3.3 Bibliography

[1] https://en.wikipedia.org/wiki/Machineepsilon

CHAPTER 4

Input and Output

4.1 Introduction

The fundamental data type for MATLAB is an array. Most of numerical computations for scientific and engineering purposes involve dealing with data in various file formats. Scientific devices and computer programs themselves generate data as files. These files are then read and converted into arrays (mostly). These arrays can be manipulated as per mathematical requirements by the files of matrix algebra. The results generate a new set of arrays. These arrays are further converted into files for visualization.

Using the information in Chapters 2 and 3 (arrays and plotting), you can now formulate physical problems in terms of numerical computations and solve them on a digital computer. This process has some requirements such as:

- The data should be in a digital form (a digital file).
- The computer program should be able to read the file and make arrays from it without errors. If errors occur, a mechanism to check those errors and warning the user should be in place. If possible, a mechanism for correcting them should also be in place.
- The data should be stored as an array in the proper data type and should be displayed on demand in the proper format.

S. Nagar, *Introduction to MATLAB for Engineers and Scientists*,
https://doi.org/10.1007/978-1-4842-3189-0_4

- 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. Users should be able to check the status of memory as and when required.
- Post-processing tasks include displaying data in various formats—as a printout from a printer, on a terminal, as a graph on a terminal or printer/plotter, etc.
- If a report for a particular experiment has input parameters, processing the data and output as a file or graph will make the user's task easier.

MATLAB has some features for each of these steps. This chapter discusses them in brief.

4.2 Interactive Input from a Keyboard

A user interacts with MATLAB using a keyboard. Keyboards generate ASCII or Unicode strings for specific characters. These are fed into MATLAB, which then interprets them to perform a specific task. For an interactive session with MATLAB during the course of programming, MATLAB offers the functions discussed in the following sections.

4.2.1 input()

`input("Text")` displays the `Text` string at the MATLAB prompt (the default symbol is `>>>`) and waits for the user to input a value and press Enter. Users may enter any type of data. The input is treated as a MATLAB expression and it is *evaluated* in the current workspace. If the `input()` function is used for an assignment operation, then the data is assigned to a variable appropriately. If the user presses the Enter key without entering anything, then the input returns an empty matrix. When the user enters an

invalid expression into the prompt, a relevant error message is displayed at the prompt. When a character or a string of character vectors needs to be fed into `input()`, the user must define them as a string. Otherwise, `'s'` is used as a second argument and then input is treated as a string. The usage is demonstrated in this code.

```
>> prompt = 'What is your name:';
>> name = input(prompt)
What is your name:Sandeep
Error using input
Undefined function or variable 'Sandeep'.

What is your name:'Sandeep'

name =

 'Sandeep'

>> name = input(prompt,'s')
What is your name:Sandeep

name =

 'Sandeep'
```

While dealing with numerical values, a single value or an array must be used, but with valid MATLAB syntax. For example, in the following code, the user can type a single value of `r` (storing the radius of a circle) or multiple values as arrays. This information can then be used to find the circumference (= $2\pi r$) and area (= $\pi r2$).

```
>> r = input('Enter value of radius:');
Enter value of radius:2
>> circumference = 2*pi*r

circumference =

12.5664

>> area = pi*r^2

area =

12.5664

>> r = input('Enter value of radius:');
Enter value of radius:[1 2 3]
>> r

r =

1     2     3

>> circumference = 2*pi*r

circumference =

6.2832   12.5664   18.8496

>> area = pi*r.^2

```

```
area =

3.1416   12.5664   28.2743

```

4.2.2 keyboard()

The keyboard keyword gives control to the user while running a program so that user can enter data or additional MATLAB commands, if required. This process can be effectively used by the user to check the program. It is called *debugging*.

When this is done, the MATLAB prompt changes from >>> to k>. The keyboard mode is terminated by executing the command dbcont. dbquit can also be used to exit keyboard mode, but in this case the invoking MATLAB code file is terminated. Control returns to the invoking MATLAB code file.

A valid MATLAB expression must be entered here. This keyword can be used to change values of variables in the middle of programs very effectively. Its usage is shown in the sample code in Listing 4-1.

Listing 4-1. The keyboardCommand.m Program

```
%program to demonstrate
%usage of keyboard command

x = 10;
y = 12;
keyboard %change value of x here
answer = x^2
```

When this is executed, you'll see the following session:

```
>> keyboardCommand
K>> x=2.5

x =

2.5000

K>> dbcont

answer =

6.2500

>> x=2.5

x =

2.5000

>> x.^2

ans =

6.2500
```

When the `keyboard` keyword is encountered, the MATLAB session goes into debug mode and the user then alters the values of `x=2.5`. Typing `dbcont` continues the execution of the program. The answer is calculated as per the new assignment of value.

4.2.3 menu()

A graphical way of inputting values can be performed using the `menu()` command, where a title and a set of options are given as inputs (separated by commas). This works if the user has a computer terminal with graphics capabilities. Otherwise, a list of options is presented at the command prompt. The user is presented with a graphical window and can use a mouse or keyboard to select an option. The options return a scalar value, which can be stored in a variable. The options are numbered internally.

Let's look at the usage with an example. Create a menu with the title "Even or Odd Numbers" and two options—`Even` and `Odd`. Store this value in the variable `I`. When this command is executed, the graphical window shown in Figure 4-1 appears. The following outputs are possible:

- If the user closes the window without entering a value, the output is 0.
- If the user clicks on `Even`, the output is 1.
- If the user clicks on `Odd`, the output is 2.

Figure 4-1. *The Menu window output*

This is shown in the following code:

```
>> I = menu('Even or Odd Numbers','Even','Odd')

I =

0

>> I = menu('Even or Odd numbers','Even','Odd')

I =

1

>> I = menu('Even or Odd numbers','Even','Odd')

I =

2

```

- `pause()`: If you want to temporarily halt the program, you can use the `pause()` command. Press any key to continue the program execution. To understand its usage, consider an example. Suppose you have an 3×3 matrix of random numbers. Then MATLAB program `pauseCommand.m` will show its rank, transpose, and size. Each item is shown if the user presses a key. See Listing 4-2.

Listing 4-2. The pauseCommand.m Program

```
%Program to show usage
%of pause command

x = rand(3,3);
r = rank(x);
t = x';
s = size(x);
disp('Given matrix is:')
disp(x)
disp('To continue checking its rank, press any key')
pause
disp('Rank of matrix is:')
disp(r)
pause
disp('To continue checking its transpose, press any key')
pause
disp('Transpose of matrix is:')
disp(t)
pause
disp('To continue checking its size, press any key')
pause
disp('Size of matrix is:')
disp(s)
pause
```

The output for running `pauseCommand.m` is shown here:

```
>> pauseCommand
Given matrix is:
0.8147    0.9134    0.2785
0.9058    0.6324    0.5469
0.1270    0.0975    0.9575
```

```
 6
 7  To continue checking its rank, press any key
 8  Rank of matrix is:
 9  3
10
11  To continue checking its transpose, press any key
12  Transpose of matrix is:
13  0.8147    0.9058    0.1270
14  0.9134    0.6324    0.0975
15  0.2785    0.5469    0.9575
16
17  To continue checking its size, press any key
18  Size of matrix is:
19  3     3
20
```

4.3 File Path

A MATLAB session starts from a default folder, which depends on the installation of a particular operating system. Usually a dedicated folder is created at the time of installation and it is generally named `MATLAB` by default. This folder's path can be viewed by typing `pwd` at the command prompt.

```
1   >> pwd
2
3   ans =
4
5   '/Users/.../MATLAB'
```

Note that the path may be different on your computer and the three dots are used to represent a generalized representation of the path.

When you want to run a MATLAB's `.m` file, the file is searched first in the default folder. If it is not there, you must change the working directory to the one where the file is stored. This can be initiated by typing `pathtool` at MATLAB's command prompt (see Figure 4-2).

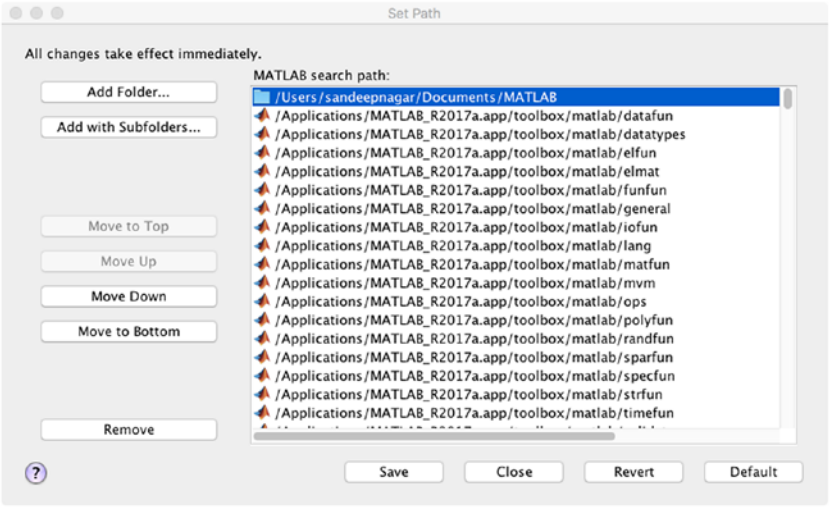

Figure 4-2. *Setting the path of working directory using the pathtool command*

You can simply add the directory to the list of directories for the session or save it. You can also choose to set the directory as the default for future MATLAB sessions. This is a good option if you will be working on a project for a long time and are sure that the directory will be used on a daily basis. It is strongly suggested that you keep all the files in either the default directory of the installed MATLAB program or make your working directory the default one.

4.4 File Operations

File operations constitute an important part of computation. It is important to note that the file system is OS (Operating System) dependent. Just like most scientific programs, MATLAB works with UNIX-like systems, so it works on Linux-based and MacOS X equally well with the same set of commands. On Windows, you use the same commands as the Linux version for dealing with files within the MATLAB environment. The code examples in this book were written and tested on Windows 8, MacOSX 10.10, and Ubuntu 14.04 systems.

4.4.1 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 that user. The 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. This is ensured by giving permissions to various users.

Reading and writing a file is restricted by permission. The administrator (fondly called the `admin`) is also called the superuser and has all privileges and permission to edit any file/folder. You must understand the defined user types for a computer system and then issue those commands accordingly. If you are not permitted to access certain folders and the input data you need is placed inside those files/folders, you will always get an error (unless the admin changes your permissions).

4.4.2 File Path

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

The pwd command stands for *print working directory*. On the MATLAB terminal, typing pwd displays the path of the present working directory, as shown in this example:

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

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

A file/folder is accessed by typing the file path into the terminal. Consider this small exercise to understand this process. To create a new directory, you use mkdir *name* as follows:

```
>> mkdir matlab_practice
ans = 1
>> ls
Downloads                       Music
R
Templates
matplab_practice
Videos
Desktop                         software
Work
Documents                       Library
Pictures
>> cd matplab_practice
>>
```

At line 1, `mkdir matplab-practice` creates a directory named `matplab-practice`. To see the contents of the present directory, you can use the `ls` command, as is done at line 3, which stands for `list`. To change the directory, you can use the `cd` *`file path`* command, as shown in line 13. I suggest that you work in this directory for rest of the book.

4.4.3 Creating and Saving Files

The `save` and `load` commands allow you to write and read data to memory.

```
 1  >> matrix = rand(3,3);
 2  >> save MyFirstFile.mat matrix
 3  >> ls
 4  MyFirstFile.mat
 5  >> load MyFirstFile.mat
 6  >> matrix
 7  matrix =
 8
 9  0.467414     0.610273     0.429941
10  0.568490     0.037898     0.734682
11  0.547370     0.275421     0.539650
12
13  >>
```

At line 1, A variable named `matrix` is created first, which stores a random-value 3×3 matrix. At line 2, this data is stored as a `.mat` file named `MyFirstFile.mat`, which is passed the variable name as the argument. When required, this file can be loaded in the workspace using the `load MyFirstFile.mat` command and then by calling the variable named `matrix`. The random numbers recorded when the file was saved are loaded as the data for the 3×3 matrix. Note that this data does need not be numbers. It can be anything that a digital computer can handle, including pictures, videos, strings, and characters, just to name a few.

Multiple variables can be stored in the same file by passing the name of the 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
   0.7924558   0.9130480   0.1976146   0.4635055

>> matrix2
matrix2 =

   0.35215   0.55770   0.66650
   0.98515   0.98677   0.45513

>> matrix3
matrix3 =

   0.097693   0.540354
   0.923853   0.329501

>>>> save –binary SavedAsBinary m*
>> ls
MyFirstFile.mat  SavedAsBinary  SavingMultipleVariables.mat
```

The `help save` and `help load` commands provide very useful instructions about using `save` and `load`. Using the options, you can save the file in a specific format. For example, on line 26, all variables names starting with `m` are saved as binary data inside a binary file named `SavedAsBinary`. This is particularly important when data generated from MATLAB-based numerical computations is used in other software programs. You can also specify the precision of saved data using options. You can also compresses a big file using the `-zip` command. This is very useful when the data generated by MATLAB is large in size and needs to be transmitted.

The `load` function follows the same logic as the `save` function. Data can be unzipped and loaded from a particular formatted file as an array. The array, thus populated, can be used for computation and the resultant files can be made by using the `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 the required data in a short time.

Delimited numeric data files (numerical data values separated by a delimiter) can be read and written using `dlmread()` and `dlmwrite()`. The functions produce ASCII-delimited files. To illustrate this, the following MATLAB code performs the following task:

- Stores a 3×3 matrix in variable `A`.
- Using the `dlmwrite()` function, a file named `randomNumbers.txt` is written, which takes its inputs from the matrix stored in `A`.
  - The delimiter is defined to `;`
  - You can check the file in the working directory and open it with an appropriate text editor or spreadsheet software.
- A new variable named `B` is initialized to be an empty matrix.

- Using the function `dlmread()`, this file is read. It is important to define the delimiter used during the creation of the file. The results are stored in `B` and found to be exactly same as that of `A`.

```
>> A = randn(3,3)

A =

 0.3252   −1.7115    0.3192
−0.7549   −0.1022    0.3129
 1.3703   −0.2414   −0.8649

>> dlmwrite('randomNumbers.txt',A,';')
>> B = []
>> B = dlmread('randomNumbers.txt',';')

B =

 0.3252   −1.7115    0.3192
−0.7549   −0.1022    0.3129
 1.3703   −0.2414   −0.8649
```

4.4.4 Using the Diary and History Commands

A MATLAB session can be recorded in a file by using the command `diary`. Type `help diary` to see information about its use. Writing `help` *`filename`* allows recording the session in a file with given filename. The commands and their outputs are continuously updated using this function.

You can use the `history` command to display a list of executed commands. Various options are available to see this history in particular formats.

4.4.5 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 to. This is defined by the following options: `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), and `a+` for reading, writing, and appending. The opening mode can be set to `t` for text mode or `b` for binary mode. `z` enables opening a gzipped file for reading and writing.

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

```
1  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 the author of the file wants the information to remain unchanged while sharing it. This might be necessary for files containing constants or important pieces of code that should not be changed.

The `freport()` command prints a list of opened files and whether they are opened for reading, writing, or both. For example:

```
1  >> freport
2
3  number    mode    arch          name
4  ------    ----    ----          ----
5  0         r       ieee-le       stdin
6  1         w       ieee-le       stdout
7  2         w       ieee-le       stderr
8
9  >>
```

4.4.6 Reading and Writing Binary Files

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

4.4.6.1 The csvread and csvwrite Functions

The `csvread` and `csvwrite` functions are used to read data from `.csv` files, which stands for comma separated 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
```

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

```
 1  >> a = [1,2,3,4;5,6,7,8;8,7,6,5;4,3,2,1];
 2  >> a
 3  a =
 4
 5  1   2   3   4
 6  5   6   7   8
 7  8   7   6   5
 8  4   3   2   1
 9  >> csvwrite('csvTestData.dat',a)
10  >> csvwrite('csvTestData1.dat',a,1,2)
11  >> a1 = csvread('csvTestData.dat')
```

```
12  a 1 =
13
14  1   2   3   4
15  5   6   7   8
16  8   7   6   5
17  4   3   2   1
18
19  >> a1 = csvread('csvTestData.dat',1,2)
20  a1 =
21
22  7   8
23  6   5
24  2   1
25
26  >>
```

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

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. You can access the documentation to learn about using these specialized functions, if required.

4.4.7 Working with Excel Files

A lot of data is presented on the Internet in the form of Excel files. Note that one must be connected to the Internet in this case.

The `xlsopen`, `xlswrite`, `xlsclose`, `odsopen`, `odswrite`, and `odsclose` commands open, write, and close `.xls` and `.ods` files, respectively.

While `.xls` files are generated using Microsoft Excel, `.ods` files are generated using Open/Libre Office software, which is the open source equivalent of Microsoft Excel. 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 `Filename.xls` worksheet named `3rd 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 named `Raw`, and the ranges where the actual data came in is saved in `limits`.

- `xlswrite('new.xls',a)` writes the data in an array named `a` into an `.xls` formatted Excel sheet named `new.xls`.
- `xlsclose`

```
1  >> a = rand(10,10);
2  >> odswrite('a.ods',a)
3  ans = 1
4  >> ls
5  a.ods
```

4.5 Reading Data from the Internet

Most often, large data sets that you need to access are kept on some remote server. Using `urlread()`, you can read a remote file. To save data to the local disk, you use the `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)`.

4.6 Printing and Saving Plots

Some commands, like `print` and `saveas`, exist to save graphs/figures generated by MATLAB programs, to be saved in desired formats. They are discussed in the following sections.

4.6.1 The print Command

The `print` command prints jobs, including printing using a printer and/or plotter, printing to a file, etc. This command is very useful if you need 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
```

The `clf` function clears the current graphic window. A lot of other options for saving in different formats exist for the `print` command. To learn more, type `help print` into the MATLAB terminal.

4.6.2 The saveas Function

The saveas function saves a graphic object in a desired format, as follows:

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

The orient(a,orientation) function defines the orientation of an graphical object a. The valid values for the 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.

4.7 Summary

This chapter explained various functions enabling reading and writing permission as well as taking data to and from a file. This becomes an essential part of a numerical computation exercise. The data can be generated in the form of files using software or hardware (an instrument). MATLAB does not care about its origin. It treats data by its type and by file type. Determining the appropriate function when using files has to be done by the user as per the situation.

File operations do provide faculties to trim the data so that only the useful part is used as an array. Further trimming can be performed by slicing operations. With the art of handling files under your belt, you can confidently proceed toward handling sophisticated numerical computations.

CHAPTER 5

Functions and Loops

5.1 Introduction

When a particular numerical tasks needs to be "repeated" over different data points, digital computers become a useful tool since they can do this with greater speed than humans. *Loops* perform exactly these tasks. Using a condition to check the start and termination rules, you can perform repetitive parts of a process easily. Different programming languages and environments have different rules for defining loops. MATLAB provides a much simpler way to define and run loops. They will be discussed shortly.

It's 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 it's quite logical, better at visualizing the problem, and easy to debug. The most popular way of defining these small sets of instructions is to define them as functions. This chapter discusses both of these concepts in detail.

S. Nagar, *Introduction to MATLAB for Engineers and Scientists*,
https://doi.org/10.1007/978-1-4842-3189-0_5

5.2 Loops

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

A variety of functions and their usage are listed in the following sections. Judging their usage critically becomes very important because the looping part of the algorithm consumes most of the execution time.

5.2.1 The while Loop

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

```
1  while condition
2  BODY
3  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 a set of commands that is executed until the condition holds true (see Listing 5-1).

Listing 5-1. The while1.m Program

```
1  x = 1.0;
2  while x<10
3     disp(sqrt(x));
4     x = x+1;
5  endwhile
```

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

$$x<10$$

In the first step of the loop, x = 1, this condition is satisfied since 1 < 10. Since this condition is satisfied, `disp(sqrt(x))` is executed and displays the square root of `x`. Then line 4 is executed, where `x = x + 1` increments `x`. With the new incremented value of `x` being 2, the logical condition x < 10 is again checked and the body of loop given in lines 3 and 4 is executed. This is done until $x = 10$, when the loop condition is not satisfied. At that point, line 5 is executed and declares the end of the `while` loop. The execution of `while1.m` yields:

```
 1  >> while1
 2  1
 3  1.4142
 4  1.7321
 5  2
 6  2.2361
 7  2.4495
 8  2.6458
 9  2.8284
10  3
```

5.2.2 The do-until Loop

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

```
1  do
2  BODY
3  until condition
```

The loop first executes the body of the code and then checks for the condition. This way, the code block comprising the BODY of loop is executed *at least once*. The usage can be understood in the example shown in Listing 5-2.

Listing 5-2. The dountil1.m Program

```
1  %Displaying square root of
2  %first ten positive natural numbers
3
4  x = 1.0;
5  do
6    disp(sqrt(x));
7    x = x+1;
8  until x == 10
```

The execution of the code yields the following:

```
 1  >> dountil1
 2  1
 3  1.4142
 4  1.7321
 5  2
 6  2.2361
 7  2.4495
 8  2.6458
 9  2.8284
10  3
11  >>
```

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

5.2.3 The for Loop

The for loop is used to perform computations on a list of known values. The syntax of the for loop is as follows:

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

The keyword for declares the start of the loop where a variable takes the values stored in a vector. Then the body of the code (here represented by BODY) is executed. The keyword end declares the end of the for loop. This is explained in the example in Listing 5-3.

Listing 5-3. The for1.m Program

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

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

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 The if-elseif-else Loop

When you need a number of conditions to be checked at different times, the `if-elseif-else` loop works well. The syntax for this 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 line 2 is executed; otherwise, line 3 is executed. Hence, `BODY1` and `BODY2` are the blocks of code that are executed by checking for different sets of conditions and `BODY3` is the code that's executed when none of the conditions are executed. See Listing 5-4.

Listing 5-4. The ifelse1.m Program

```
 1  %Program to check if a
 2  %number is even or odd
 3
 4  x = 33;
 5
 6  if(rem(x,2) == 0)
 7    printf("x is even\n");
 8  elseif(rem(x,5) == 0)
 9    printf("x is odd and divisible by 5\n");
10  else
11    printf("x is odd\n");
12  endif
```

Executing `ifelse1.m` yields:

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

At line 4, `x` is initialized as `33`. Then, at line 6, the remainder of $\frac{x}{2}$ is checked. If it is zero, then line 7 is executed. Otherwise, line 8 is executed and the remainder of $\frac{x}{5}$ is checked. If it is zero, then line 9 is executed. If neither of the conditions is satisfied, then line 11 is executed. Line 12 ends the `if-else` loop.

5.3 Functions

A *function* is code that can be called as and when required. Hence, it can be defined separately, either in a separate file or within the body of program. MATLAB presents several ways to define functions, which are discussed in the following subsections.

5.3.1 The function Function

The definition of a `function` follows this syntax:

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

Here, the `function` keyword defines the object types as a function. Then, a set of variables are defined that this function is expected to return. Next comes an = operator, and then the name of the function. In this case, it's called `name`. Name objects takes a set of arguments, which are objects that the function defined. Then comes the main body of the function.

The last part defines the end of the function. For example, you can write a function to find $x^2 - y^2$ and assign the result to a variable named z, as follows:

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

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

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

You can see that the 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:

```
1  function[y1,y2,y3] = fn2(x,y)
2  y1 = x^2-y^2;
3  y2 = x^2+y^2;
4  y3 = y2-y1;
5  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, a factorial of a number can be calculated using the function given here:

```
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 the number with all its successive numbers. When called from the MATLAB command line, the function yields the following result.

```
>> factorial(50)
ans = 3.0414e+064
>> factorial(1)
ans = 1
>> factorial(0)
```

```
 6  ans = 1
 7  >> factorial(100)
 8  ans = 9.3326e+157
 9  >> factorial(1000)
10  ans = NaN
11  >> factorial(-1)
12  error:factorial:N must all be non-negative integers
```

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

5.3.2 The inline Function

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

```
1  >> f = inline("x^2+y");
2  >> f(1,2)
3  ans = 3
4  >> f(10,10)
5  ans = 110
6  >> f(0,2)
7  ans = 2
8  >>
```

Line 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, the function outputs the calculated values.

5.3.3 Anonymous Functions

Anonymous functions are unnamed function objects defined in a program. Their 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 is as follows.

$$\int_0^1 sin(x)cos(x) = 0.35404$$

$$\int_0^{\pi} sin(x)cos(x) = 7.3031 \times 10^{-17}$$

$$\int_{-\pi}^{\pi} sin(x)cos(x) = 0$$

$$\int_{-\pi}^{2\pi} sin(x)cos(x) = -2.8435 \times 10^{-16}$$

$$\int_{-\pi}^{2\pi} sin(x)cos(x) = 0$$

Hence, if you use the anonymous function definition, you do not need to name a function.

5.4 Summary

Defining functions is the key to modular programming. MATLAB 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 just a few lines of code. This requires an artistic attitude while designing an algorithm, where functions and loops are the paintbrushes that help you devise an elegant solution to a given numerical problem.

CHAPTER 6

Numerical Computing Formalism

6.1 Introduction

Numerical computation enables you to compute solutions to numerical problems, provided you can frame them into a proper format. This requires certain considerations. For example, if you digitize continuous functions, then you are going to introduce certain errors due to the sampling at a finite frequency. Hence, a very accurate result would require very a fast sampling rate. When a large data set needs to be computed, it becomes a computationally intensive and time consuming task. Also you 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 you. The art of discarding solutions that do not have meaning in real world scenarios is something that a scientist/engineer develops over the years. Also, a computational device is only as intelligent as its operator. The law of GIGO (garbage-in-garbage-out) is followed very strictly in this domain.

This chapter attempts to explain some of the important steps you must consider in order to solve a physical problem using numerical computations. Defining a problem in the proper terms is just the first step. Making the right model and then using the right method to solve (solver) the issue is the difference between a naive and an experienced scientist/engineer.

S. Nagar, *Introduction to MATLAB for Engineers and Scientists*,
https://doi.org/10.1007/978-1-4842-3189-0_6

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 the things happening around us, we obtained mathematical theories for physical laws. To test these mathematical formalisms of physical laws, we use numerical computations. If it yields the same results as that of a real experiment, they validate each other. Numerical simulations can remove the need to do an experiment altogether, provided you have a well tested mathematical formalism. For example, nuclear powers of our times need not test nuclear bombs for real any more. The data related to nuclear explosion, which was obtained during real nuclear explosions, enables scientists to model these physical systems quite accurately, thus eliminating the need to do real testing.

Apart from applications like simulating a real experiment, modeling physical problems are good educational exercises. While modeling, hands-on exercises enable students to explore the subject in depth and give proper meaning to the topic under study. Solving numerical problems and visualizing results makes the learning permanent and also elucidates any flaws in the 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 equals (=) sign, where terms on the left side are equal to terms on the right side. The terms on either side of an equation can be numbers or expressions. For example:

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

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

Expressions are written using functions, which is simply a relationship between two domains. Like $f(x) = y$ is a relationship from y to x using the rules of algebra. Mathematics has a rich library of functions, which you can use to make expressions.

Choosing the proper functions depends on the problem. Some functions describe some situations best. For example, the 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 the present position, m is the constant rate of change of x, and c is the offset position. Objects moving in a curved fashion can be described by various non-linear functions (where the power of the dependent variable is not 1).

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

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

Suppose the following equations define the physical behavior of a system:

$$-x+3y=4 \qquad \text{(Equation 6-1)}$$

$$2x-4y=-3 \qquad \text{(Equation 6-2)}$$

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 2D space (because there are two dependent variables) where the system will find stability for that physical problem. In this way, you can predict the behavior of system without actually doing an experiment.

Mathematical concepts of differentiation and integration become very important when you need to work with dynamic systems. When the system is constantly changing the values of its dependent variables to produce a scenario, it's important to know the rate of change of these variables. When these variables are independent of each other, you can use simple derivatives to define their rate of change. When they are not independent of each other, you must use partial derivatives for the same.

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

$$F \alpha \frac{dy}{dx} \qquad \text{(Equation 6-3)}$$

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

$$F = m \times \frac{dy}{dx} \qquad \text{(Equation 6-4)}$$

If you 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, you can digitize the system and find a numerical solution.

There are many methods to digitize and numerically solve a given function. Programs used to implement a particular method to solve a function numerically are called *solvers*. A lot of solvers exist to solve a function. The 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 such problems, like Euler, Runge-Kutta, etc. The choice of the particular solver depends on the accuracy of its solution, the time taken for obtaining a solution, and the amount of memory used during the process. The last point is especially important when memory is not an freely expendable commodity, such as when you're using micro-computers with limited memory storage.

The advantage of using MATLAB to perform numerical computations lies in the fact that it has a very rich library of functions to perform the various tasks required. The predefined functions have been optimized for speed and accuracy (in some cases, accuracy can be predefined). This enables you 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 Example: Polynomials

The coefficients of a vector are defined as elements of a vector. In this manner, a coefficient is defined for numerical computing. For example, consider defining two arrays, `p1` and `p2`, as shown:

```
>> p1 = [1 0 3 2]

p1 =

1     0     3     2

>> p2 =[3 4 0 5]

p2 =

3     4     0     5
```

The corresponding polynomial for `p1` is $p_1(s) = s^3 + 3s - 2 = 0$ and for `p2`, it's $p_2(s) = 3s^3 + 4s^2 - 5 = 0$. See Figure 6-1.

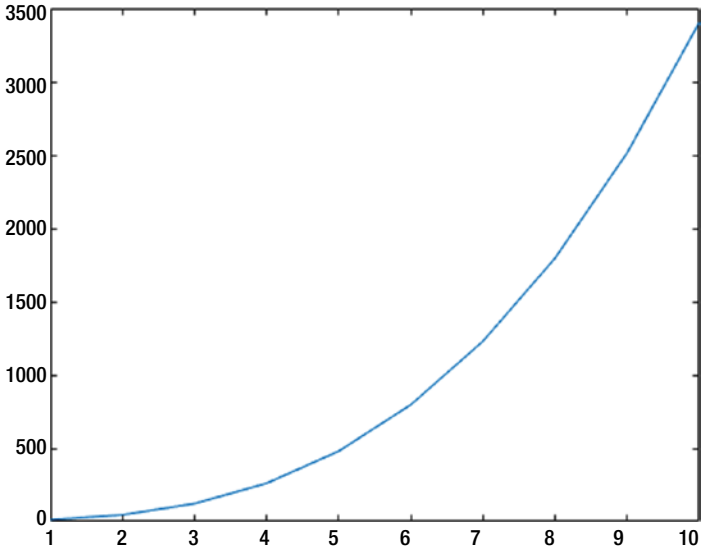

Figure 6-1. *Plot for equation* $p_2(s) = 3s^3 + 4s^2 - 5 = 0$

6.4.1 polyval()

Polynomials can be evaluated for a single value or multiple values using the `polyval()` function. Consider the polynomials defined in `p1` and `p2`. Let's calculate the values for $p_1(5)$ and p_{2}(-2).

```
>> s = 5

s =

5

>> polyval(p1,s)

```

```
 9  ans =
10
11  142
12
13  >> s=2
14
15  s =
16
17  2
18
19  >> polyval(p2,s)
20
21  ans =
22
23  45
```

If a polynomial needs to be calculated on multiple values, say from 1 to 10 for p_1, then an array `1:10` can be fed to the `s` variable and this can be used in the `polyval()` function.

```
1  >> s = 1:10;
2  >> polyval(p2,s)
3
4  ans =
5
6  12     45      122       261       480       797
   1230   1797    2516      3405
```

This facility can be used to plot polynomials easily. The `plot()` command can be fed `s` and `polyval()` output as the x and y axes to visualize a plot.

6.4.2 roots()

The *roots* of a polynomial are the numerical values where the evaluated polynomial is valued at zero. Roots can be found easily using the `roots()` function. Here's an example using the previously defined polynomials, `p1` and `p2`.

```
>> p1

p1 =

1     0     3     2

>> p2

p2 =

3     4     0     5

>> roots(p1)

ans =

0.2980 + 1.8073i
0.2980 - 1.8073i
-0.5961 + 0.0000i

>> roots(p2)

ans =

```

```
25  −1.8307 + 0.0000i
26  0.2487 + 0.9212i
27  0.2487 − 0.9212i
28
29  >> polyval(p1,roots(p1))
30
31  ans =
32
33  1.0e−14*
34
35  −0.1776 − 0.2720i
36  −0.1776 + 0.2720i
37  0.0888 + 0.0000i
38
39  >> polyval(p2,roots(p2))
40
41  ans =
42
43  1.0e−13*
44
45  −0.2753 + 0.0000i
46  −0.0089 − 0.0111i
47  −0.0089 + 0.0111i
```

As per the definition of a root, the polynomial should be valued at zero at its roots, but the value for `roots(p1,roots(p1))` is not zero. Instead, it's a very small number in the order of 10^{-14}. This is due to errors introduced in the numerical approximations for calculating the roots.

6.4.3 Addition and Subtraction of Polynomials

Two polynomials are added by adding their coefficients. Since they are defined as arrays in MATLAB, polynomial addition and subtraction is simply an element-wise operation.

```
>> p1+p2

ans =

4     4     3     7

>> p1-p2

ans =

-2    -4     3    -3
```

This effectively means that:

$$p_1(s) = s^3 + 3s - 2 = 0 \quad \text{(Equation 6-5)}$$

$$p_2(s) = 3s^3 + 4s^2 - 5 = 0 \quad \text{(Equation 6-6)}$$

$$p_1(s) + p_2(s) = 4s^3 + 4s^2 + 3s + 7 \quad \text{(Equation 6-7)}$$

$$p_1(s) - p_2(s) = -2s^3 - 4s^2 + 3 - 3 \quad \text{(Equation 6-8)}$$

6.4.4 Polynomial Multiplication

The product of two polynomials can be found using a convolution operation, which is provided using the `conv()` function in MATLAB.

```
>> p1

p1 =

1     0     3     2

>> p2

p2 =

3     4     0     5

>> conv(p1,p2)

ans =

3     4     9     23     8     15     10
```

$$p_1(s) = s^3 + 3s - 2 = 0 \quad \text{(Equation 6-9)}$$

$$p_2(s) = 3s^3 + 4s^2 - 5 = 0 \quad \text{(Equation 6-10)}$$

$$P_1(a) \times p_2(a) = 3s^6 + 4s^5 + 9s^4 + 23s^3 + 8s^2 + 15x + 10 = 0 \quad \text{(Equation 6-11)}$$

6.4.5 Polynomial Division

Polynomial division is performed by using deconvolving operations, which are provided by the `deconv()` function. It gives two outputs—a quotient and a remainder.

```
>> p1

p1 =

1     0     3     2

>> p2

p2 =

3     4     0     5

>> [q,r] = deconv(p1,p2)

q =

0.3333


r =

0     −1.3333     3.0000     0.3333

>> [q,r] = deconv(p2,p1)

q =

3
```

```
31  r =
32
33  0     4     −9     −1
```

This means that if:

$$p_1(s) = s^3 + 3s - 2 = 0 \quad \text{(Equation 6-12)}$$

$$p_2(s) = 3s^3 + 4s^2 - 5 = 0 \quad \text{(Equation 6-13)}$$

Then:

$$\frac{p_1}{p_2} \rightarrow q = 0.333, r = -1.3333s^2 + 3s + 0.3333 \quad \text{(Equation 6-14)}$$

$$\frac{p_2}{p_1} \rightarrow q = 3, r = -4s^2 - 9s - 1 = 0 \quad \text{(Equation 6-15)}$$

6.4.6 Polynomial Differentiation

Polynomial differentiation can be accomplished using the `polyder()` function. For example, say you have a polynomial $y(x) = x^3 - 2x^2 + 4x - 5 = 0$. That means:

$$\frac{dy}{dx} = 3x^2 - 4x + 4 = 0$$

This can be calculated by MATLAB as follows.

```
1  >> y = [1 −2 4 −5]
2
3  y =
4
5  1     −2     4     −5
6
```

```
>> dydx = polyder(y)

dydx =

3     −4     4
```

6.4.7 Polynomial Integration

Just as with differentiation, you can define integration of polynomials using the `polyint()` function. For example, say you have a polynomial $y(x) = x^3 - 2x^2 + 4x - 5 = 0$. Then:

$$\int y(x)dx = 0.25x^4 - 0.6667x^3 + 2x^2 - 5x = 0$$

```
>>>> y = [1 −2 4 −5]

y =

1     −2     4     −5

>> integration =vpolyint(y)

integrationv=

0.2500     −0.6667     2.0000     −5.0000     0
```

6.4.8 Polynomial Curve Fitting

Suppose you are given some data and need to find a polynomial that fits the data. This task can be performed using the `polyfit()` function. For example, suppose you want to fit the data given here:

x	1	2	3	4	5	6
y	10	11	21	2	3	7

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

x =

1     2     3     4     5     6

>> y = [10,11,21,2,3,7]

y =

10     11     21     2     3     7

>> polyfit(x,y,2)

ans =

−0.3750     0.9679     11.3000

>> polyfit(x,y,3)

ans =

1.0833     −11.7500     35.3095     −16.0000
```

Second and third degree polynomials that fit the data are $-0.375x^2 + 0.9679x + 11.3 = 0$ and $1.0833x^3 - 11.75x^2 + 35.3095x - 16 = 0$, respectively.

6.5 Summary

Almost all branches of science and engineering require you to perform numerical computations. MATLAB is one of the alternatives for doing so. MATLAB has a library of optimized functions for general computation. It also has a variety of packages that perform specialized jobs. This makes it an ideal choice for prototyping a numerical computation problem efficiently. This chapter summarized various issues related to errors generated during numerical computation and various methods to obtain their value or order of magnitude. These quantities are important to measure, since in real life, you will need these values to define the accuracy of the final product.

CHAPTER 7

Approximate answers in numerical computation

7.1 Numerical Approximations

In the course of scientific investigation, finding exact answers may not be possible at times. Instead of devoting a lot of effort trying to find an exact answer by solving the problem analytically, another alternative is to develop methods to produce approximate answers. This is particularly true for solutions involving irrational numbers like `pi`. You can choose the number of significant digits to be used with `pi` and determine the accuracy of the result.

The degree of accuracy required always depends on the targeted application. For example, when measuring the length of a building, we don't need the answer to be accurate to the length of an atom (Å). When measuring a person's body temperature, we don't need to be accurate to more than two decimal places for most applications. In the era of faster and more efficient computers, higher accuracies of computations can be calculated by investing more time and memory storage, whenever required. But this must be used judiciously.

S. Nagar, *Introduction to MATLAB for Engineers and Scientists*,
https://doi.org/10.1007/978-1-4842-3189-0_7

7.2 Tolerance

When an approximated answer or a set of approximated answers is available to the user, one answer must be chosen depending on the requirements of the application. One of the ways to make this decision is to define a *tolerance* limit. Tolerance can be defined as a single number or a range of numbers (having a maximum and a minimum). The rules to define tolerance limits are entirely application dependent. For example, while measuring human height, we could define the tolerance to be 1 centimeter but at the same time, while measuring the diameter of a human hair, we might like to be more accurate, by going down to 1 micron. At the same time, while measuring the size of a red blood cell, we would need to go further down, to 1 nm (nanometer). Whereas the decision to define tolerance is simpler when measuring sizes—i.e., tolerance is one or two orders of magnitude smaller than the size of the object—it may not be a straightforward task in other applications. For example, measuring land when constructing a building would require a tolerance of a fraction of meters, whereas positioning a screw in a hole requires an accuracy within a fraction of a centimeter.

In mathematical terms, if $\in$ is the tolerance limit, x represents the real values, and x^* represents the approximated value:

$$\left|x - x^*\right| \leq \in \qquad \text{(Equation 7-1)}$$

In this case, the absolute error (e_a) and the relative error (e_r) in the measurements are given by:

$$e_a = \left|x - x^*\right| \qquad \text{(Equation 7-2)}$$

$$e_r = \frac{\left|x - x^*\right|}{x} \qquad \text{(Equation 7-3)}$$

Hence, if absolute error is less than or equal to the tolerance limit, the approximate solution or set of solutions is acceptable. However, if x is known, why do we need to calculate an approximate solution?

When solutions of physical systems are unknown, x^* can be calculated and then be compared to the physical measurements. The physical measurements constitute the value of x in this case. Then, by using Equation 7-2, we can calculate errors. Tolerance can then be determined using the fact that some x^* will differ from x *insignificantly,* i.e., the errors don't matter much.

7.3 Taylor Series

Most mathematical functions require many complex operators—other than the simpler ones like +, −, ×, and ÷—to be computed. However, a polynomial requires only these basic ones to be computed. Hence, if other mathematical functions can be represented in terms of polynomials, they can be approximated with relative ease.

A polynomial is defined as follows:

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \qquad \text{(Equation 7-4)}$$

where $a_n \in R$ (The as are called the *coefficients*). For the largest n that corresponds to $a_n \neq 0$, the degree of polynomial is defined to be n.

7.4 Taylor Polynomials

Taylor's theorem explains how to define a great many mathematical functions, which can be defined as polynomials and are called *Taylor polynomials.* The accuracy of the final answer shown by a Taylor polynomial depends on its degree, i.e., the number of terms defined in the polynomial. This provides a convenient method to customize the polynomial based on the desired tolerance.

Suppose a mathematical function $f(x)$ needs to be approximated around $x = a$. A Taylor Polynomial $p_n(x)$ of degree n centered at $x = a$ is a polynomial (of degree at-most n) that has the same value as n^{th} derivative at $x = a$.

Here's how to derive the formula for a Taylor Polynomial:

1. The zero order polynomial $p_0(x)$ has degrees of at most zero.
   - $p_0(x)$ must be a constant function (a horizontal line function, graphically).
   - Approximating around $x = a$: $p_0(x) = f(a)$.
2. The first order polynomial $p_0(x)$ has a degree at most of 1.
   - p1(x) must satisfy two conditions:

$$p_1(a) = f(a)$$

and

$$p_1'(a) = f_1'(a)$$

   - $p_1(x)$ must be of the form $p_1(x) = mx + c$ (a straight line with slope m and c as the intercept).
   - Since $p'(a) = f'(a)$ so $m = f'(a)$
   - So we can write $c = f(a) - f'(a)a$
   - Substituting back the values of m and c, we get

$$p_1(x) = f'(a)x + f(a) - f'(a)a = f(a)(x - a)$$

3. Carrying forward the same arguments in a similar fashion, you can write the general form of the Taylor Polynomial of order n as:

$$p_n(x) = f(a) + f'(a)(x-a) + \frac{1}{2}f''(a)(x-a)^2 + \frac{1}{3!}f'''(a)(x-a)^3 + \cdots + \frac{1}{n!}f^n(a)(x-a)^n$$

which can be rewritten in sigma notation as follows:

$$p_n(x) = \sum_{k=0}^{n} \frac{1}{k!} f^k(a)(x-a)^k \qquad \text{(Equation 7-5)}$$

This definition requires that the polynomial must have *n* derivatives at $x = a$.

The Maclaurin Series is simply the Taylor Series defined for $a = 0$. You can use algebraic manipulations of the Taylor/Maclaurin Series for basic functions like $sin(x)$, $cos(x)$, and e^x to define other complicated functions in their series forms. These can be performed by simply using algebraic operators in addition to substitutions, derivatives, and integrations. This mathematical convenience comes in handy in formulating approximate solutions for physical systems defined by complicated functions.

7.4.1 Maclaurin Series for *sin(x)* and *cos(x)*

To check Maclaurin expansion, let's start with the trigonometric functions $sin(x)$ and $cos(x)$. Both are continuous and differentiable in the range given by any set of real numbers. Hence their differentials exist in the same. They can be expanded in the form of a Maclaurin Series as follows.

Suppose $f(x) = sin(x)$ needs to be approximated at $a = 0$.

Using Table 7-1 and Equation 7.5 results in this equation:

$$sin(x) = x - \frac{1}{3!}x^3 + \frac{1}{5!}x^5 - \frac{1}{7!}x^7 + \frac{1}{9!}x^9 - \cdots \pm \frac{1}{n!}x^n \qquad \text{(Equation 7-6)}$$

Table 7-1. *Calculating Coefficients for the Maclaurin Series of sin(x) at x = 0*

n	*f(x)*	*f(a)*
0	*sin(x)*	0
1	*cos(x)*	1
0	*–sin(x)*	0
1	*–cos(x)*	–1
0	*sin(x)*	0

Similarly for $f(x) = cos(x)$ approximated at $a = 0$, using Table 7-2 and Equation 7-5 results in this equation:

$$cos(x) = 1 - \frac{x^2}{2} + \frac{1}{4!}x^4 - \frac{1}{6!}x^6 + \frac{1}{8!}x^8 - \cdots \pm \frac{1}{n!}x^n \qquad \text{(Equation 7-7)}$$

Table 7-2. *Calculating Coefficients for the Maclaurin Series of cos(x) at x = 0*

n	*f(x)*	*f(a)*
0	*cos(x)*	1
1	*–sin(x)*	0
0	*–cos(x)*	–1
1	*sin(x)*	0
0	*cos(x)*	1

Choosing Tolerance While Calculating cos(x)

The program `MaclaurinCos.m` in Listing 7-1 shows how error is reduced by many orders of magnitude, as more and more terms of the Taylor Series are included for calculating *cos*(15^0). See Figure 7-1.

Listing 7-1. The MaclaurinCos.m Program

```
%A program to show usage of Taylor Series expansion of cos(x)
%Suppose we wish to calculate cos(15) where argument of cos
function is given in degrees

x = 15*pi/180; %converts 15 degrees into radian

format long %show results in long format having a lot of
decimal places for numbers

%Calculating each term of Taylor Series

p1 = 1;
p2 = x^(2)/2;
p4 = x^(4)/factorial(4);
p6 = x^(6)/factorial(6);
p8 = x^(8)/factorial(8);
p10 = x^(10)/factorial(10);

approx_1= p1-p2; %approximate values using two terms
approx_2= p1-p2+p4; %approximate values using three terms
approx_3= p1-p2+p4-p6; %approximate values using four terms
approx_4= p1-p2+p4-p6+p8; %approximate values using five
terms
```

```
21 approx_5= p1−p2+p4−p6+p8−p10; %approximate values using
   six terms
22
23 real_value = cos(x); %calculating the real value to find
   errors
24
25 %calculation of final errors
26
27 error_1 = abs(real_value − approx_1);
28 error_2 = abs(real_value − approx_2);
29 error_3 = abs(real_value − approx_3);
30 error_4 = abs(real_value − approx_4);
31 error_5 = abs(real_value − approx_5);
32
33 %making an error vector for plotting
34
35 error = [error_1, error_2, error_3, error_4, error_5];
36
37 %plotting error versus number of terms
38
39 figure(1)
40 semilogy(error,'*r−')
41 title('Variation of error in calculating cos(15^{0}) using
   Taylor Series')
42 xlabel('Number of terms on Taylor Series')
43 ylabel('log(error)')
44
45 %plotting cos(x) and its various approximations
46
47 t = 0:0.001:20;
48 %length(t)
49
```

```
figure(2)
y = cos(t);
subplot(2,3,1)
plot(t,y,t,ones(length(t)))
subplot(2,3,2)
plot(t,y,t,(1–t.^2/2))
subplot(2,3,3)
plot(t,y,t,(1–t.^2/2+t.^4/factorial(4)))
subplot(2,3,4)
plot(t,y,t,(1–t.^2/2+t.^4/factorial(4)–t.^6/factorial(6)))
subplot(2,3,5)
plot(t,y,t,(1–t.^2/2+t.^4/factorial(4)–t.^6/factorial(6)+t.^8/
factorial(8)))
subplot(2,3,6)
plot(t,y,t,(1–t.^2/2+t.^4/factorial(4)–t.^6/factorial(6)+t.^8/
factorial (8)–t . ^ 10 / factorial (10)))
```

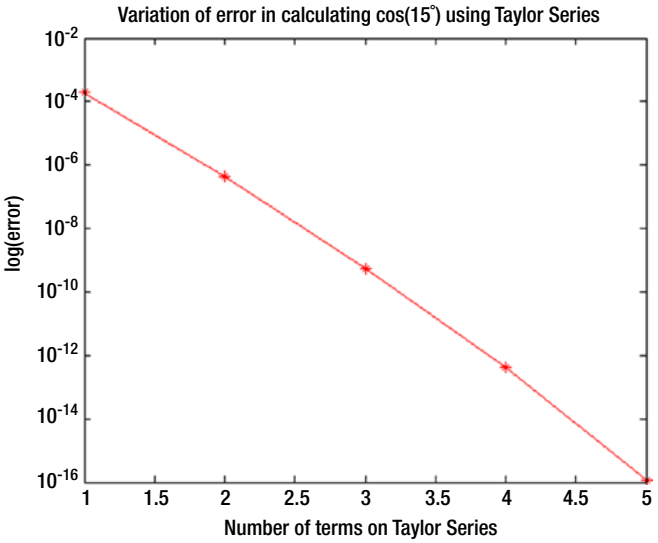

Figure 7-1. *Variation of logarithmic error in the number of terms used to define a Maclaurin Series for cos(x)*

As seen from Figure 7-1, you can now choose to insert certain numbers of terms as per the given tolerance for calculating $cos(x)$. To make a judicious decision about the number of terms, you must inspect the function in a similar fashion (as was done by `MaclaurinCos.m`). Inserting a lot of terms while demanding less accuracy is a waste of time, energy, and resources (both human and computational).

Instead of expanding around one particular point, the series can be defined for a set of points. The Octave program called `CosApprox.m` attempts the same, as shown in Listing 7-2. See Figure 7-2.

Listing 7-2. The CosApprox.m Program

```
%plotting cos(x) and its various approximations

t = -3*pi:pi/10:3*pi; %defining an array of points for
x-axis
l = length(t); %to be used for defining pi
y = cos(t); %real values of cosine function

%defining various terms of Maclaurin Series
a1 = ones(l); %only first term
a2 = (1-t.^2/2); %first and second term
a3 = (a2+t.^4/factorial(4)); %first, second and third term
a4 = (a3-t.^6/factorial(6)); %first, second, third and
fourth term
a5 = (a4+t.^8/factorial(8)); %first, second, third, fourth
and fifth term
a6 = (a5-t.^10/factorial(10)); %first, second, third,
fourth, fifth and sixth term

%plotting fitting of cos(x) with increasing number of terms
figure(1)

```

```
subplot(3,2,1)
plot(t,y,'*r–',t,a1,'*b–')
axis([–3*pi 3*pi –1.2 1.2])
title('fitting p_{1} to cos(x)')
xlabel('t')
ylabel('cos(t)')

subplot(3,2,2)
plot(t,y,'*r–',t,a2,'*b–')
axis([–3*pi 3*pi –1.2 1.2])
title('fitting p_{2} to cos(x)')
xlabel('t')
ylabel('cos(t)')

subplot(3,2,3)
plot(t,y,'*r–',t,a3,'*b–')
axis([–3*pi 3*pi –1.2 1.2])
title('fitting p_{3} to cos(x)')
xlabel('t')
ylabel('cos(t)')

subplot(3,2,4)
plot(t,y,'*r–',t,a4,'*b–')
axis([–3*pi 3*pi –1.2 1.2])
title('fitting p_{4} to cos(x)')
xlabel('t')
ylabel('cos(t)')

subplot(3,2,5)
plot(t,y,'*r–',t,a5,'*b–')
axis([–3*pi 3*pi –1.2 1.2])
```

```
49  title('fitting p_{5} to cos(x)')
50  xlabel('t')
51  ylabel('cos(t)')
52
53  subplot(3,2,6)
54  plot(t,y,'*r-',t,a6,'*b-')
55  axis([-3*pi 3*pi -1.2 1.2])
56  title('fitting p_{6} to cos(x)')
57  xlabel('t')
58  ylabel('cos(t)')
```

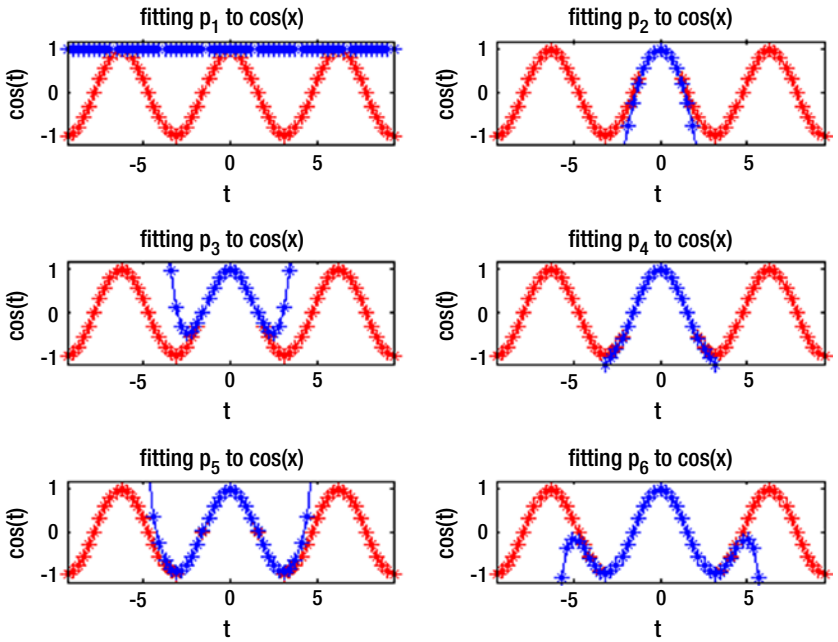

Figure 7-2. *Fitting of Maclaurin Series with different numbers of terms to cos(x)*

As you can see from Figure 7-2, as higher orders of terms are used to describe `cos(x)`, the error reduces by fitting with increasing accuracy. For ideal fitting, very large numbers of terms must be used to describe the approximated `cos(x)` function. The choice of tolerance is user defined. Depending on the tolerance value, a particular number of terms can be determined.

7.4.2 The Maclaurin Series for e^x

Let's explore the concept of errors using another example of the Maclaurin Series—for e^x:

$$e^a = 1 + a + \frac{a^2}{2!} + \frac{a^3}{3!} + \frac{a^4}{4!} + \cdots \qquad \text{(Equation 7-8)}$$

For programming purposes, it's easier to derive an inherent relationship between the terms of the Maclaurin Series. The first terms is the number 1, but afterward, each term can be obtained by multiplying the previous terms by this equation:

$$\frac{a}{n} \qquad \text{(Equation 7-9)}$$

where n represents the n^{th} term. This fact is used in the `MaclaurinExp.m` code (see Listing 7-3), where the first term is defined at line number 5 in variable `expVal` and then this variable is added to the `currentTerm` variable, which is simply calculated using the formula in Equation 7-9.

Listing 7-3. The MaclaurinExp.m Program

```
1  %Maclaurin Series for exp(0.1)
2
3  n = 5; %Number of terms
4  a = 0.1; %Functional value of x for e^(x)
```

```
 5  expVal = 1.0;
 6  currentTerm = 1.0;
 7  for i=1:n
 8    currentTerm = currentTerm*a/i;
 9    expVal = expVal+currentTerm
10  endfor
11
12  trueVal = exp(0.1);
13  error = abs(trueVal –expVal)
```

The output is displayed as follows:

```
 1  >> MaclaurinExp
 2  expVal = 1.1000
 3  expVal = 1.1050
 4  expVal = 1.1052
 5  expVal = 1.1052
 6  expVal = 1.1052
 7  error = 1.4090e –09
 8  >> format long
 9  >> MaclaurinExp
10  expVal = 1.10000000000000
11  expVal = 1.10500000000000
12  expVal = 1.10516666666667
13  expVal = 1.10517083333333
14  expVal = 1.10517091666667
15  error  = 1.40898115397192e–09
```

Notice that while the numeric display is usually set for just four numerical values after the decimal point, the `format long` command increases this accuracy (the `format short` command returns to the default behavior). Thus, you can clearly observe that by increasing the number of

terms, the error is reduced drastically as it approaches the true value and you will achieve an error of the order 10^{-9} in just five terms.

If you want to store all the calculated values in the `expVal` variable, you must define it as a vector, as shown in Listing 7-4 (the `MaclaurinExp1.m` program).

Listing 7-4. The MaclaurinExp1.m Program

```
%Maclaurin Series for exp(0.1)

n = 5; %Number of terms
a = 0.1; %Functional value of x fore^(x)
expVal = 1.0;
currentTerm = 1.0;
for i =1:n
  currentTerm = currentTerm*a/i;
  expVal(i+1) = expVal(i)+currentTerm;
endfor

trueVal = exp(0.1);
error = abs(trueVal-expVal)
```

Here, line 9 dictates that the $(i+1)^{th}$ is modified as per Equation 7-9, using the previous term, i.e., the $(i)^{th}$ term. Also notice that printing line 9 has been suppressed by using the `;` operator. The output is shown as follows:

```
>> MaclaurinExp1
error =

 1.0517e-01   5.1709e-03   1.7092e-04   4.2514e-06
 8.4742e-08   1.4090e-09
>>>plot(error,'r*-')
```

Using the `plot(error,'r*-')`, you can generate graphs where error values are plotted (the `r*-` argument shows red stars connected with rules). This is shown in Figure 7-3.

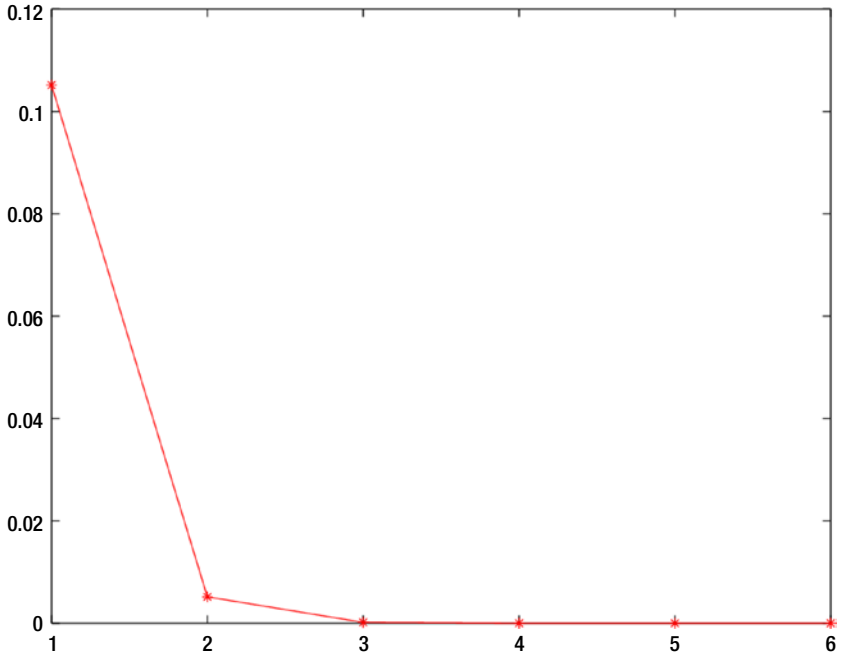

Figure 7-3. *Error in calculating $e^{0.1}$ with an increasing number of terms*

Since the error drops by orders of magnitude with each new term, the effect can be best seen in a logarithmic plot. This can be generated using the `semilogy(error,'r*-')` command. Figure 7-4 will be generated.

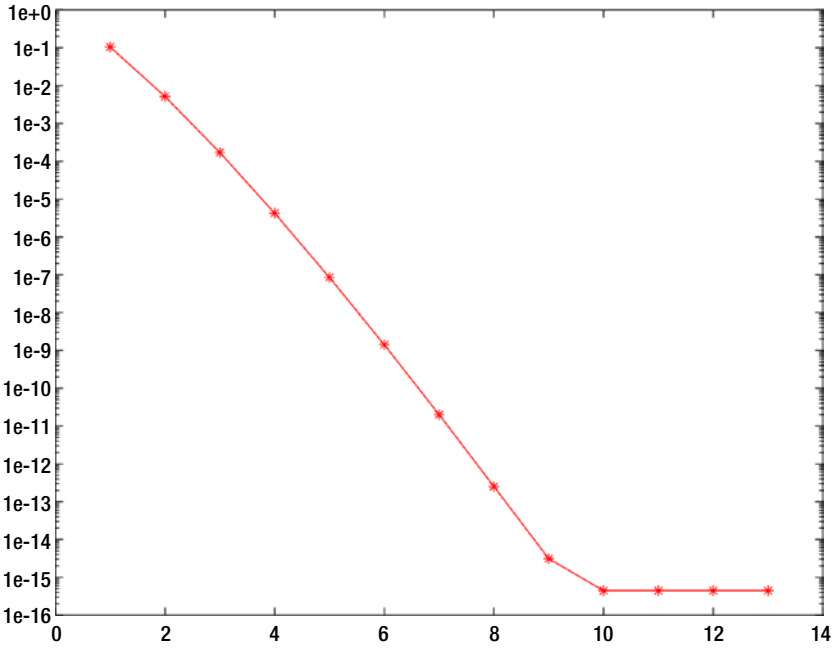

Figure 7-4. *Error in calculating $e^{0.1}$ with an increasing number of terms*

It seems that beyond 10 terms, the error flattens out. But you will see that this is an erroneous result, as this graph will depend on the least count of your computing machine.

How Many Number of Terms?

You can observe from Figure 7-4 that, by increasing the number of terms, you reduce the error by two orders of magnitude when calculating $e^{0.1}$. But does this trend mean that to achieve true values, you must include an infinite number of terms? After all, each time you add a new term, you invest time and energy into the computation. In general, the Maclaurin Series has the accuracy of a^{n+1} when n terms are used:

$$e^a = 1 + a + \frac{a^2}{2!} + \frac{a^3}{3!} + \frac{a^4}{4!} + \ldots + \frac{a^n}{n!} + O\left(a^{n+1}\right) \quad \text{(Equation 7-10)}$$

Analytically, you can choose n to be any large number, but this cannot be done on a computing machine. The reason for this is explored next.

Figure 7-4 shows one interesting fact that beyond 10 terms, the error no longer changes by orders of magnitude and instead just flattens out. This is a misleading result. Each computing machine has limits for storing the smallest floating point number. This can be obtained by issuing the `eps` command. The system on which the program has been run shows the following output.

```
1  >> error(9:12)
2  ans =
3
4  3.10862446895044e-15   4.44089209850063e-16
   4.44089209850063e-16   4.44089209850063e-16
5  >>> eps
6  ans = 2.22044604925031e-16
```

You can now see that, when error values are very close to `eps` values, they cannot be stored reliably anymore. The command `error(9:12)` outputs a similar viewpoint. It can be seen that while the ninth term yields an error of the order of 10^{-15}, the eleventh term onward have similar values of the order 10^{-16}. This is done so that the computer avoids crashing the calculation by going beyond its limits defined by the `eps` value.

The `eps` command gives the machine precision. The `help('eps')` command shows the documentation for the `eps` command and its usage. Technically, `eps` is the relative spacing between any two adjacent numbers in the machine's floating point system, i.e., computational machines' least count. This number is obviously system dependent as you can devise specialized hardware where machine precision can be enhanced. In fact, this is done when increased precision matters, such as for missile guidance, space navigation, etc. On machines that support IEEE floating point arithmetic, `eps` is approximately 2.2204×10^{-16} for double precision and 1.1921×10^{-7} for single precision.

It is interesting to note that $2^{-25} \approx 2.2204 \times 10^{-16}$. This essentially signifies that the double precision mode of software can store 52 digits after the decimal point. You learn more about this in the "Computational Errors" section of this chapter, which discusses the machine precision aspect of numerical computations and the importance of knowing which precision you need to work on for a particular numerical problem.

The `realmax`, `realmin`, `intmax`, and `intmin` commands show the maximum and minimum values of real numbers and integers on the particular machine where the software is installed.

```
>> realmax
ans = 1.79769313486232e+308
>> intmax
ans = 2147483647
>> realmin
ans = 2.22507385850720e-308
>> intmin
ans = -2147483648
```

It is useful to know these numbers, as the numbers beyond these limits will be prone to error because of machine precision.

7.5 Computational Errors

Up until now, you have read about the inherent errors that are due to the inclusion of a certain number of terms while calculating a mathematical function. There is, in fact, another kind of error, which is introduced due to the fact that computers can store only numbers of finite lengths.

7.5.1 Significant Digits

The concept of *significant digits* plays an important role here. If computers can store all the significant digits of the final answer, the errors become irrelevant. Otherwise, it is important to identify them and, if possible, rectify them when reporting a final answer. For example, while dealing with `pi`, if only three significant digits are desired, this can be stored easily on any low-end computing solution.

Computers can store numbers as floating point objects. A floating point object stores a number as follows:

$$\pm d_1 d_2 \ldots d_s \times \beta^e \qquad \text{(Equation 7-11)}$$

Where $d_i = 0,1,2 \ldots \beta - 1$ but $d_1 \neq 0$ and $m \leq e \leq M$ where $m \in I^-$ and $M \in I^+$.

Three parts of a floating point number are:

- Sign (±)
- Mantissa ($d_1 d_2 \ldots d_s$)
- Exponent (β)

In IEEE double precision roundoff, MATLAB uses binary arithmetic where:

- $\beta = 2$
- $s = 53$
- $m = -1074$
- $M = +1023$

Since humans are used to *decimal arithmetic* systems, these binary numbers are converted to decimal numbers for reporting purposes. It is important to understand the key point that all internal calculations are done in binary form but input and output for humans are fed in decimal

form. The rounding-off error due to conversion is given by the unit roundoff, u, which is the maximum relative error while approximating a real number as a floating point number.

MATLAB can handle numbers with absolute values from $2^{-1074} \simeq 10^{-324}$ and $2^{-1023} \simeq 10^{308}$ with a unit roundoff of $u = 2^{-53} \simeq 10^{-16}$.

7.6 Challenges in Real Number to Floating Point Number Conversion

A real number x can be stored in floating point representation given by Equation 7-11 as:

$$x = \pm d_1 d_2 \ldots d_s d_{s+1} \ldots \times 10^e \qquad \text{(Equation 7-12)}$$

Now note that $s = 53$, but the previous description does not restrict representation of a floating point number. Its storage is, however, an altogether different game. When it is stored, the number is rounded off and stored as per the guidelines, i.e., $s = 53$.

7.6.1 Overflow

From Equations 7-11 and 7-12, if $e > M$, computation is said to have *overflowed*. That means a number bigger than possible has been presented and hence the storage container has *overflowed*. In this case, MATLAB produces `Inf` or `-Inf` as the answer, which represents the fact that the answer is a very large number.

The following exercise, performed in a MATLAB terminal, explains the process clearly. `Inf` is displayed as an answer when `e900` is attempted. When this number is divided by a negative number, `-Inf` is displayed, signifying an overflow while storing a negative number. When `Inf-Inf` is attempted, `NaN` (which stands for *Not a Number*) is displayed, signifying that the large numbers cannot produce a result that's meaningful.

```
>> format long
>> exp(50)

ans =

5.184705528587072e+21

>> exp(100)

ans =

2.688117141816136e+43

>> exp(500)

ans =

1.403592217852837e+217

>> exp(700)

ans =

1.014232054735005e+304

>> exp(900)

ans =

Inf

```

```
>> exp(900)/-2

ans =

-Inf

>> exp(900)-exp(900)

ans =

NaN
```

7.6.2 Underflow

If $e < m$, then *underflow* is said to have occurred. Octave represents an underflow by showing zero and the answer. It would seem that underflow is not serious, but consider the fact that, as per basic rules of exponentiation:

$$e^{a}e^{-a} = e^{a-a} = e^{0} = 1$$

When you perform the same calculations for numbers representing overflow and underflow, Octave has to perform `Inf X 0`, which results in `NaN`. This is demonstrated in the following example:

```
>> exp(900)*exp(-900)

ans =

NaN

>> exp(900)

```

```
 9  ans =
10
11  Inf
12
13  >> exp(-900)
14
15  ans =
16
17  0
```

7.7 Actual Conversions of Real Numbers to Floating Point Numbers

After look at the two extreme cases, overflow and underflow, you need to understand the real number to floating point number conversion process. Recall from Equations 7-11 and 7-12 that a real number can be stored with s significant digits, as follows:

$$\pm d_1 d_2 \ldots d_s \times \beta^e$$

whereas it can be written in floating point notation (for base 10) as follows:

$$x = \pm d_1 d_2 \ldots d_s d_{s+1} \ldots \times 10^e$$

There are two ways to achieve the conversion: using the *method of truncation* and using the *method of rounding off*. The method of truncation will simply discard all digits after s, i.e., it will produce the following:

$$x = \pm d_1 d_2 \ldots d_s \times 10^e \qquad \text{(Equation 7-13)}$$

On the other hand, the method of rounding off recommends the following process:

1. If $s_{s+1} < 5$, then perform truncation and retain the sign of x.
2. If $s_{s+1} > 5$, then d_s is incremented. Then truncation is performed and retain the sign of x.

This seemingly simple scheme has a flaw. Suppose for $s = 4$, you need to round off 2.9345. The answer would be 2.934, i.e., last digit 5 is simply discarded. In a similar fashion, when 2.9355 is rounded off, the answer can be written as 2.936, where the last digit is discarded and the last significant digit is incremented. In both cases, only one digit changed. But suppose you need to round off 2.9999. In this case, the answer comes out to be 3.000, where four numeral values changed.

7.8 Alternatives to MATLAB

With growing computational power, advances in numerical computers, and the dropping prices of computational resources, MATLAB has become the language of engineering. With challenges posed from open source alternatives like Scilab [1], Octave [2], and Python [3], it now needs to innovate in new dimensions to remain relevant, both commercially and academically. My books on Octave, Scilab, and Python (available on Amazon [4, 5, 6]) run parallel with the contents of this book for easier comparison [7]. I highly recommended that you study all these options so you can make the best decision for your needs.

7.9 Summary

It should be clear now that performing mathematical modeling of physical systems using numerical computation does not mean merely entering input and getting out *some* output. This must be done judiciously. The very act of performing calculations on a discrete system having a finite level for computation introduces errors. These errors must be mentioned while presenting results so that they can be cross-checked by other investigators. But despite the fact that numerical computations are tedious to code and they introduce error, they remain tremendously popular among scientists. This is because modern computers offer greater speeds of calculation and analytical solutions to most of the physical world's problems, and these calculations remain too tedious to be done by humans. This trend is expected to continue in the near future.

7.10 Bibliography

[1] Sandeep Nagar. *Introduction to Scilab: For Engineers and Scientists*, volume 1 of *1*. Self-Published, 2 edition, 1 2016.

[2] Sandeep Nagar. *Introduction to Octave: For Engineers and Scientists*, volume 1 of *1*. Self-Published, 2 edition, 1 2016.

[3] Sandeep Nagar. *Introduction to Python for Engineers and Scientists: Open Source Solutions for Numerical Computation*, volume 1 of *1*. Self-Published, 2 edition, 1 2016.

[4] https://www.amazon.com/dp/1520158106

[5] https://www.amazon.com/dp/1520153686

[6] https://www.amazon.com/dp/152015111X

[7] Sai K Popuri, Andrew M Raim, Matthew W Brewster, and Matthias K Gobbert. A comparative evaluation of matlab, octave, freemat, scilab, and r on tara. Technical report, Technical Report HPCF-2012-7, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2012.

CHAPTER 8

Symbolic Computation

8.1 Introduction

Until now, we have been dealing with numeric computation where variables store numeric values. In Chapter 7, you learned that numerical computation involves working with *approximate* solutions. On the other hand, an analytical solution is not an approximation since one uses *symbols* rather than numbers. MATLAB provides the means to perform symbolic computations, too.

8.2 Defining a Symbolic Variable

The keyword `syms` is used to define single or multiple *symbolic variable(s)*. The key feature of a symbolic variable is that it just stores a symbol to perform symbolic calculations.

```
1  >> syms x y z
2  >>
```

After executing the command, inspect the Workspace window (see Figure 8-1) and note that three new variables—`x`, `y`, and `z`—have been created.

S. Nagar, *Introduction to MATLAB for Engineers and Scientists*,
https://doi.org/10.1007/978-1-4842-3189-0_8

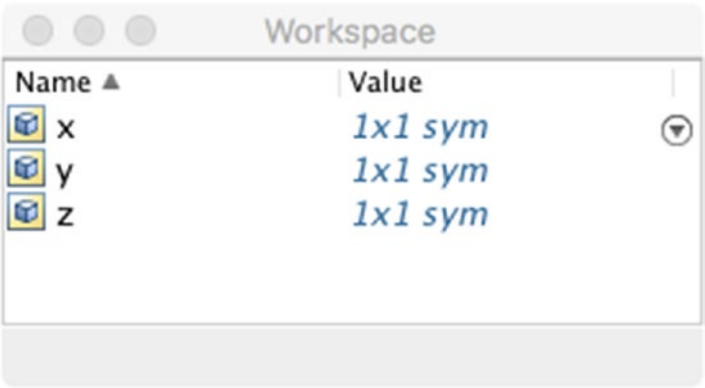

Figure 8-1. *New symbolic variables appearing in the workspace*

8.3 Defining a Symbolic Equation

Once the variables have been defined, you can define an equation:

$$z = x^2 + y \qquad \text{(Equation 8-1)}$$

using these variables as follows:

```
1  >> z = x^2+y
2  z =
3  x^2 + y
```

In the present example, z was predefined as a symbolic variable. The output variable is created by MATLAB and becomes a symbolic variable by default. Its inputs have been defined as symbolic variables. For example, suppose you want to define this equation:

$$a = x^3 + y^2 + z \qquad \text{(Equation 8-2)}$$

This results in the creation of a new symbolic variable (which can be verified by checking the Workspace window). The following MATLAB code performs this task:

```
>> syms x y z
>> a = x^3+y^2+z
a =
x^3 + y^2 + z
```

8.4 Performing Symbolic Computations

Symbolic computations are same as what we are used to doing by hand on paper. You define a variable and use mathematical rules of algebra as well as calculus to perform calculations. For example, two roots (r_1 and r_2) of a quadratic equation:

$$y = ax^2 + bx + c \qquad \text{(Equation 8-3)}$$

can be written as follows:

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad \text{(Equation 8-4)}$$

$$r_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \qquad \text{(Equation 8-5)}$$

This can be performed using the following MATLAB code:

```
>> syms a b c x
>> y = a*(x^2)+(b*x)+c
y =
a*x^2 + b*x + c
>> solve(y)
ans =
-(b + (b^2 - 4*a*c)^(1/2))/(2*a)
-(b - (b^2 - 4*a*c)^(1/2))/(2*a)
```

Similarly, a symbolic mathematical expression can be integrated and differentiated as follows:

```
>> syms a b c x
>> y = a*(x^2)+(b*x)+c
y =
a*x^2 + b*x + c
>> int(y)
ans =
(a*x^3)/3 + (b*x^2)/2 + c*x
>> diff(y)
ans =
b + 2*a*x
```

This can be verified using paper-based calculation by hand, where we know the following:

$$y = ax^2 + bx + c \qquad \text{(Equation 8-6)}$$

$$\int y = \frac{ax^3}{3} + \frac{bx^2}{2} + cx \qquad \text{(Equation 8-7)}$$

$$\frac{dy}{dx} = 2ax + b \qquad \text{(Equation 8-8)}$$

8.4.1 Arithmetic Expressions

Simple arithmetic expressions can be dealt with using symbols. For example, two polynomials can be used to define a new polynomial.

```
>> syms x y z
>> a1 = x^2+2*y+z
a1 =
x^2 + 2*y + z
```

```
>> a2 = x^(-2)-2*y+3*z
a2 =
3*z - 2*y + 1/x^2
>> a3 = a1/a2
a3 =
(x^2 + 2*y + z)/(3*z - 2*y + 1/x^2)
>> a4 = a1*a2
a4 =
(x^2 + 2*y + z)*(3*z - 2*y+1/x^2)
```

8.4.2 Trigonometric Expressions

Trigonometric variables defined using symbolic variables can also be used in mathematical calculations, as follows:

$$y = \sin(x) \qquad \text{(Equation 8-9)}$$

$$\frac{dy}{dx} = -\cos(x) \qquad \text{(Equation 8-10)}$$

```
>> syms a b c x
>> y = sin(x)
y =
sin(x)
>> int(y)
ans =
-cos(x)
```

Even more complicated calculations can be performed by a click of a button.

```
>> z = cos(x^(1/2)) - (sin(y))^(1/3)
z =
cos(x^(1/2)) - sin(y)^(1/3)
```

```
>> int(z)
ans =
2*cos(x^(1/2)) – x*sin(y)^(1/3) + 2*x^(1/2)*sin(x^(1/2))
>> diff(z)
ans =
–sin(x^(1/2))/(2*x^(1/2))
```

8.4.3 Expanding and Factorizing an Expression

The `expand()` function can be used to write equations with individual terms of expanded polynomials. The most important use of `expand()` is the application of the distributivity law to rewrite products of sums as sums of products. If `f` represents a symbolic expression, then `expand(f)` is calculated using the following set of rules:

- $x^{a+b} = x^a \times x^b$
- $(xy)^b = x^b \times y^b \forall x, y \geq 0, b \in I$
- $\left(x^a\right)^b = x^{a+b}$

It is also important to note that the `expand()` function will work recursively on the subexpressions of a given expression.

```
>> syms x y z
>> a1 = x^2+2*y+z
a1 =
x^2 + 2*y + z
>> a2 = x^(–2)–2*y+3*z
a2 =
3*z – 2*y + 1/x^2
>> a3 = a1/a2
a3 =
(x^2 + 2*y + z)/(3*z – 2*y + 1/x^2)
```

```
>> a4 = a1*a2
a4 =
(x^2 + 2*y + z)*(3*z - 2*y + 1/x^2)
>> a5 = expand(a3)
a5 =
(2*y)/(3*z - 2*y + 1/x^2) + z/(3*z - 2*y + 1/x^2) + x^2/
(3*z - 2*y + 1/x^2)
>> a6 = expand(a4)
a6 =
4*y*z + (2*y)/x^2 - 2*x^2*y + z/x^2 + 3*x^2*z - 4*y^2 +
3*z^2 + 1
```

The function named `factor` produces factors of an expression such that multiplying all factors results in the final expression. Let's try to factorize the values stored in symbolic variable `a5` and `a6`.

```
>> a7 = factor(a5)
a7 =
[-1,x,x, x^2 + 2*y + z, -1/(3*x^2*z - 2*x^2*y + 1)]
>> a8 = factor(a5)
a8 =
[-1,x,x, x^2 + 2*y + z, -1/(3*x^2*z - 2*x^2*y + 1)]
```

The factors are present as elements of an array, which can be accessed using their index. This comes in handy when extracting a factor and its usage in mathematical analysis.

```
>> a7[2] = x
>> a7[4] = x^2 + 2*y + z
>> a8[3:5] = [x, x^2 + 2*y + z, -1/(3*x^2*z - 2*x^2*y + 1)]
```

In the previous example, `a7[2]` extracts the second element of variable `a7`, `a7[4]` extracts the fourth element of variable `a7`, and `a8[3:5]` extracts all elements from the third to fifth element and stores them as a list of symbolic expressions.

When an expression is written as a power of another expression, `expand()` works just like mathematical rules. For example, consider the case when an expression:

$$a = xy^{(z+y)}$$

is defined. Its expansion is given as:

$$x \times y^{y} \times y^{z}$$

Each term is clearly a factor of the expression. This can be verified with the following MATLAB code:

```
>> syms x y z
>> a = x*y^(z+y)
a =
x*y^(y + z)
>> b = expand(a)
b =
x*y^y*y^z
>> c = factor(b)
c =
[x,y^y,y^z]
```

When an expression is powered by another expression, the `expand()` function works recursively.

```
>> a=((x+y)^(x+z+2))
a =
(x + y)^(x + z + 2)
>> expand(a)
```

```
ans =
x^2*(x + y)^x*(x + y)^z + y^2*(x + y)^x*(x + y)^z + 2*x*y*
(x + y)^x*
(x + y)^z
```

It can be used to check out trigonometric identities:

```
>> expand(sin(x+y))
ans =
cos(x)*sin(y) + cos(y)*sin(x)
>> expand(cos(x+y))
ans =
cos(x)*cos(y) - sin(x)*sin(y)
>> expand(tan(x+y))
ans =
-(tan(x) + tan(y))/(tan(x)*tan(y) - 1)
>> expand(sec(x+y))
ans =
1/(cos(x)*cos(y) - sin(x)*sin(y))
>> a = cosh(x+y)
a =
cosh(x+y)
>> expand(a)
ans =
cosh(x)*cosh(y) + sinh(x)*sinh(y)
>> a = cosh(2*x)
a =
cosh(2*x)
>> expand(a)
ans =
2*cosh(x)^2 - 1
```

```
>> a = coth(x+y)
a =
coth(x + y)
>> expand(a)
ans =
(coth(x)*coth(y) + 1)/(coth(x) + coth(y))
```

8.5 Summary

This chapter illustrated the usage of symbols to solve mathematical equations. Symbolic computation proves useful when error-prone numerical computing is not acceptable, but it has its limits. A limited set of built-in functions must be appended by user-defined functions, and this requires experience with writing MATLAB packages. But it is definitely worth exploring.

This book has illustrated the use of MATLAB as a tool for efficient scientific computing. It first illustrated the basic usage using single-line commands and then illustrated writing multi-line commands as an `.m` file. Arrays for the fundamental blocks of scientific computing and thus matrix-based calculations can be performed using arrays. Plotting graphs is simplified to the extent that even a beginner can easily plot a equation to visualize a graph. Using loops and functions, programs can be made modular and information flow can be controlled in an efficient fashion. You also saw some basic examples of numerical computing. The book should enable any beginner to enter the world of scientific computing with ease. Its widely popular usage has rightly coined the phrase "MATLAB is the language of engineering".

Index

A, B

S. Nagar, *Introduction to MATLAB for Engineers and Scientists*,
https://doi.org/10.1007/978-1-4842-3189-0

C

D

E

F

G

H

I, J

K

L

M

N, O

P, Q

R

S

T, U, V, W

X

Y, Z