INTRODUCTION TO MATLAB FOR ENGINEERING STUDENTS

David Houcque
Northwestern University

Contents

Preface

"Introduction to MATLAB for Engineering Students" is a document for an introductory course in MATLAB®[1] and technical computing. It is used for freshman class at Northwestern University. This document is not a comprehensive introduction or a reference manual. Instead, it focuses on the specific features of MATLAB that are useful for engineering classes. The lab classes are used with two main goals in mind: (a) demonstrate concepts seen in class, (b) allow students to become familiar with computer software (MATLAB) to solve application problems.

The availability of technical computing environment such as MATLAB is now reshaping the role and applications of computer laboratory projects to involve students in more intense problem-solving experience. This availability also provides an opportunity to easily conduct numerical experiments and to tackle realistic and more complicated problems.

Originally, the manual is divided into computer laboratory sessions (labs). The lab document is designed to be used by the student while working at the computer. The emphasis here is "learning by doing". This quiz-like session is supposed to be fully completed in 50 minutes in class. We also assume that the students have no prior experience with MATLAB.

The seven lab sessions include not only the basic concepts of MATLAB, but also an introduction to scientific computing, in which they will be useful for the upcoming engineering courses. In addition, engineering students will see MATLAB in their other courses.

The end of this document contains two useful sections: a Glossary which contains the brief summary of the commands and built-in functions as well as a collection of release notes. The release notes, which include several new features of the Release 14 with Service Pack 2, well known as R14SP2, can also be found in Appendix. All of the MATLAB commands have been thoroughly tested to take advantage with new features of the current version of MATLAB available here at Northwestern (R14SP2). Although, most of the examples and exercises still work with previous releases as well.

This manual reflects the ongoing effort of the McCormick School of Engineering and Applied Science leading by Dean Stephen Carr to institute a significant technical computing in the EngineeringFirst®[2] courses taught at Northwestern University.

Finally, the students - Engineering Analysis (EA) Section - deserve my special gratitude. They were very active participants in class. It has been gratifying to see the progress and positive reaction from them.

David Houcque
Evanston, Illinois
May 2005

[1]MATLAB® is a registered trademark of MathWorks, Inc.

[2]EngineeringFirst® is a registered trademark of McCormick School of Engineering and Applied Science (Northwestern University)

Acknowledgements

I would like to thank Dean Stephen Carr for his constant support. I am grateful to a number of people who offered helpful advice and comments. I want to thank the EA1 instructors (Fall Quarter 2004), in particular Randy Freeman, Jorge Nocedal, and Allen Taflove for their helpful reviews on some specific parts of the document. I also want to thank Malcomb MacIver, EA3 Honors instructor (Spring 2005) for helping me to better understand the *animation* of system dynamics using MATLAB. I am particularly indebted to the many students (340 or so) who have used these materials, and have communicated their comments and suggestions. Finally, I want to thank IT personnel for helping setting up the classes and other computer related work: Joe Morrow, Rebecca Swierz, Jesse Becker, Rick Mazec, Alan Wolff, Peter Nyberg, Ken Kalan, and Mike Vilches.

About the author

David Houcque has more than 20 years' experience in the modeling and simulation of structures and solid continua including 14 years in industry. In industry, he has been working as R&D engineer in the field of nuclear engineering, oil rig platform offshore, oil reservoir engineering, and steel industry. All of them include working in different international environments: Germany, France, Norway, and United Arab Emirates. Among other things, he has a combined background experience: scientific computing and engineering expertise. He earned his academic degrees from Europe and the United States.

Here at Northwestern University, he is working under the supervision of Professor Brian Moran, a world-renowned expert in fracture mechanics, to investigate the integrity assessment of the aging highway bridges under severe operating conditions and corrosion.

Chapter 1

Tutorial lessons 1

1.1 Introduction

The tutorials are independent of the rest of the document. The primarily objective is to help you learn quickly the first steps. The best way to learn is by trying it yourself. Working through the examples below will give you a feel for the way that MATLAB operates. In this introduction we will describe how MATLAB handles simple numerical expressions and mathematical formulas.

1.2 Basic features

The name MATLAB stands for MATrix LABoratory. MATLAB [1] is a high-performance language for technical computing. It integrates *computation*, *visualization*, and *programming* environment. Furthermore, MATLAB is a modern programming language and problem solving environment: it has sophisticated *data structures*, contains built-in editing and *debugging tools*, and supports *object-oriented programming*. These factors make MATLAB an excellent tool for teaching and research.

MATLAB has many advantages compared to conventional computer languages (e.g., C, FORTRAN) for solving technical problems. MATLAB is an interactive system whose basic data element is an *array* that does not require dimensioning. The software package has been commercially available since 1984 and is now considered as a standard tool at most universities and industries worldwide.

It has powerful *built-in* routines that enable a very wide variety of computations. It also has easy to use graphics commands that make the visualization of results immediately available. Specific applications are collected in packages referred to as *toolbox*. There are toolboxes for signal processing, symbolic computation, control theory, simulation, optimization, and several other fields of applied science and engineering.

In addition to the MATLAB documentation, we would like to recommend the following books and documents: [2], [3], [4], [5], [6], [7], [8], and [9].

The following TUTORIAL lessons are designed to get you started quickly in MATLAB. The lessons are intended to make you familiar with the basics of MATLAB. We urge you to complete the EXERCISES given at the end of each lesson.

1.3 A minimum MATLAB session

The goal of this minimum session is to learn the first steps:

- How to log on
- Invoke MATLAB
- Do a few simple calculations
- How to quit MATLAB

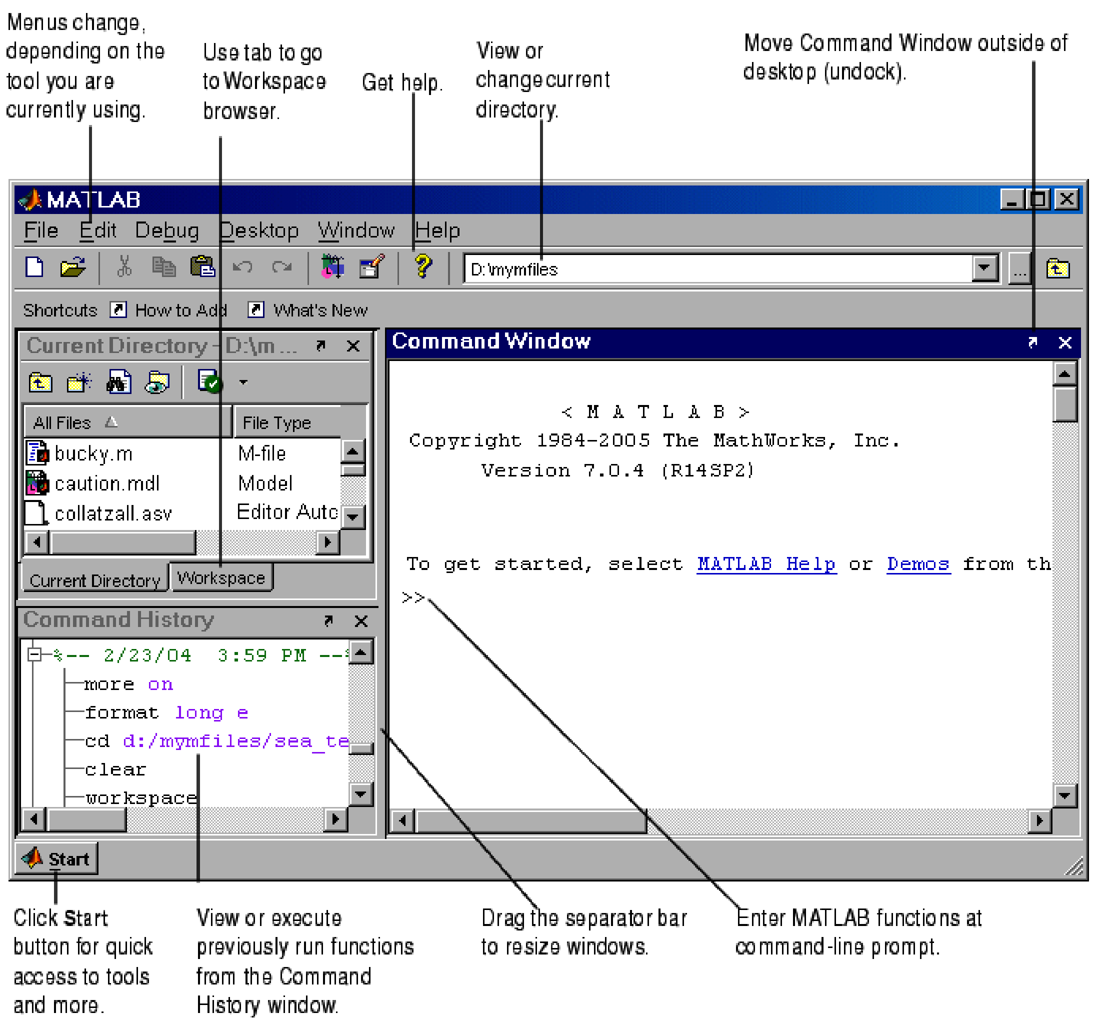
1.3.1 Starting MATLAB

After logging into your account, you can enter MATLAB by double-clicking on the MATLAB shortcut *icon* (MATLAB 7.0) on your Windows desktop. When you start MATLAB, a special window called the MATLAB desktop appears. The desktop is a window that contains other windows. The major tools within or accessible from the desktop are:

- The Command Window
- The Command History Window
- The Workspace Browser
- The Help Browser
- The Start button

When MATLAB is started for the first time, the screen looks like the one that shown in the Figure 1.1 below. The following illustration also shows the default configuration of the MATLAB desktop. You can customize the arrangement of tools and documents to suit your needs.

Now, we are interested in doing some simple calculations. We will assume that you have sufficient understanding of your computer under which MATLAB is being run. You are now faced with a window on your computer, which contains the MATLAB prompt (>>) in the Command Window. Usually, there are 2 types of prompt:



Menus change, depending on the tool you are currently using.

Use tab to go to Workspace browser.

Get help.

View or change current directory.

Move Command Window outside of desktop (undock).

```
            < M A T L A B >
Copyright 1984-2005 The MathWorks, Inc.
      Version 7.0.4 (R14SP2)


To get started, select MATLAB Help or Demos from th
>>
```

```
%-- 2/23/04  3:59 PM --%
  more on
  format long e
  cd d:/mymfiles/sea_te
  clear
  workspace
```

Click Start button for quick access to tools and more.

View or execute previously run functions from the Command History window.

Drag the separator bar to resize windows.

Enter MATLAB functions at command-line prompt.

Figure 1.1: The graphical interface to the MATLAB workspace

```
>>          for full version
EDU>        for educational version
```

NOTE: To simplify the notation, we will use `>>` as a prompt sign, though our MATLAB version is for educational purpose.

1.3.2 Using MATLAB as a calculator

As an example of a simple interactive calculation, just type the expression you want to evaluate. Let's start at the very beginning. For example, let's suppose you want to calculate the expression, $1 + 2 \times 3$. You type it at the prompt command (>>) as follows,

```
>> 1+2*3
ans =
    7
```

You will have noticed that if you do not specify an output variable, MATLAB uses a default variable `ans`, short for *answer*, to store the results of the current calculation. Note that the variable `ans` is created (or overwritten, if it is already exists). You can also assign a value to a variable or output argument name. For example,

```
>> x = 1+2*3
x  =
   7
```

will result in `x` being given the value $1 + 2 \times 3 = 7$. This name can always be used to refer to the results of the previous computations. Therefore, computing $4x$ will result in

```
>> 4*x
ans  =
     28.0000
```

Here is a partial list of arithmetic operators used in MATLAB:

SYMBOL	OPERATION	EXAMPLE
$+$	Addition	$2 + 3$
$-$	Subtraction	$2 - 3$
$*$	Multiplication	$2 * 3$
$/$	Division	$2/3$

1.3.3 Quitting MATLAB

To end your MATLAB session, type **quit** in the Command Window, or select **File** $\longrightarrow$ **Exit MATLAB** in the desktop main menu.

1.4 Getting started

After quitting, we are now going to re-enter MATLAB session again using the same operations as described above. This time we will learn to use some additional operations. After MATLAB is launched, a prompt `>>` appears in the command window. Right next to the prompt is a cursor blinking, indicating that the system is ready to receive commands.

1.4.1 Creating MATLAB variables

MATLAB variables are created with an assignment statement statement. The syntax of variable assignment is

```
variable name = a value (or an expression)
```

For example,

```
>> x = expression
```

where `expression` is a combination of numerical values, mathematical operators, variables, and function calls. On other words, expression can involve:

- manual entry
- built-in functions
- user-defined functions

1.4.2 Overwriting variable

Once a variable has been created, it can be reassigned.

```
>> t=5;
>> t=t+1
t   =
    6
```

1.4.3 Error messages

If we enter an expression incorrectly, MATLAB will return an error message. For example, in the following, we left out the * in the following expression

```
>> x=10;
>> 5x
??? 5x
     |
Error: Unexpected MATLAB expression.
```

1.4.4 Making corrections

To make corrections, we can, of course retype the expressions. But if the expression is lengthy, we make more mistakes by typing a second time. A previously typed command can be recalled with the up-arrow key ↑. When the command is displayed at the command prompt, it can be modified if needed and executed.

1.4.5 Controlling the hierarchy of operations or precedence

Let's consider the previous arithmetic operation, but now we will include *parentheses*. For example, $1 + 2 \times 3$ will become $(1 + 2) \times 3$

```
>> (1+2)*3
ans   =
       9
```

By adding parentheses, these two expressions give different results: 7 and 9.

The order in which MATLAB performs arithmetic operations is exactly that taught in high school algebra courses. *Exponentiations* are done *first*, followed by *multiplications* and *divisions*, and finally by *additions* and *subtractions*. However, the standard order of precedence of arithmetic operations can be changed by inserting *parentheses*. For example, the result of $1+2\times3$ is quite different than the similar expression with parentheses $(1+2)\times3$. The results are 7 and 9 respectively. Parentheses can always be used to overrule *priority*, and their use is recommended to avoid ambiguity.

Therefore, to make the evaluation of expressions unambiguous, MATLAB has established a series of rules. The order in which the arithmetic operations are evaluated is given in Table 1.1,

MATLAB arithmetic operators obey the same *precedence* rules as those in most computer programs. For operators of *equal* precedence, evaluation is from *left* to *right*. Now,

Precedence	Mathematical operations
First	The contents of all parentheses are evaluated first, starting from the innermost parentheses and working outward.
Second	All exponentials are evaluated, working from left to right
Third	All multiplications and divisions are evaluated, working from left to right
Fourth	All additions and subtractions are evaluated, starting from left to right

Table 1.1: Hierarchy of arithmetic operations

consider another example:

$$\frac{1}{2+3^2}+\frac{4}{5}\times\frac{6}{7}$$

In MATLAB,

```
>> 1/(2+3^2)+4/5*6/7                      (2.1)
ans  =
     0.7766
```

or, if parentheses are missing,

```
>> 1/2+3^2+4/5*6/7                        (2.2)
ans  =
     10.1857
```

Notice that (2.2) give the wrong answer.

1.4.6 Controlling the appearance of floating point number

MATLAB by default displays only 4 decimals in the result of the calculations, for example −163.6667, as shown in above examples. However, MATLAB does numerical calculations in *double* precision, which is 15 digits. The command `format` controls how the results of computations are displayed. Here are some examples of the different formats together with the resulting outputs.

```
>> format short
>> x=-163.6667
```

If we want to see all 15 digits, we use the command `format long`

```
>> format long
>> x= -1.636666666666667e+002
```

To return to the standard format, enter `format short`, or simply `format`.

There are several other formats. For more details, see the MATLAB documentation, or type `help format`.

It is important to realize that although MATLAB only displays four significant digits in the default format (short), it is computing the answer to an accuracy of sixteen significant digits.

Note - Up to now, we have let MATLAB repeat everything that we enter at the prompt (>>). Sometimes this is not quite useful, in particular when the output is pages en length. To prevent MATLAB from echoing what we type, simply enter a semicolon (;) at the end of the command. For example,

```
>> x=-163.6667;
```

and then ask about the value of `x` by typing,

```
>> x
x   =
    -163.6667
```

Note that MATLAB commands can be placed on a single line separated by semi-colons (;) or commas (,).

1.4.7 Managing the workspace

The contents of the workspace persist between the executions of separate commands. Therefore, it is possible for the results of one problem to have an effect on the next one. To avoid this possibility, it is a good idea to issue a `clear` command at the start of each new independent calculation.

```
>> clear
```

The command `clear` or `clear all` removes all variables from the workspace. This frees up system memory. In order to display a list of the variables currently in the memory, type

```
>> who
```

while, `whos` will give more details which include size, space allocation, and class of the variables.

1.4.8 Keeping track of your work session

Finally, it is possible to keep track of everything done during a MATLAB session with the diary command.

```
>> diary filename
```

The function `diary` is useful if you want to save a complete MATLAB session. They save all input and output as they appear in the MATLAB window. When you want to stop the recording, enter `diary off`. If you want to start recording again, enter `diary on`. The file that is created is a simple text file. It can be opened by an editor or a word processing program and edited to remove extraneous material, or to add your comments. You can use the function type to view the diary file or you can edit in a text editor or print. This command is useful, for example in the process of preparing a homework or lab submission.

1.4.9 Entering multiple statements per line

It is possible to enter multiple statements per line. Use commas (,) or semicolons (;) to enter more than one statement at once. Commas (,) allow multiple statements per line without suppressing output.

```
>> a=7; b=cos(a), c=cosh(a)
b   =
    0.6570
c   =
    548.3170
```

1.4.10 Miscellaneous commands

Here are few additional useful commands:

- To clear the Command Window, type `clc`
- To abort a MATLAB computation, type `ctrl-c`
- To continue a line, type ...

1.4.11 Getting help

To view the online documentation, select MATLAB Help from Help menu or MATLAB Help in the Command Window. The preferred method is to use the Help Browser. The Help Browser can be started by selecting the ? icon from the desktop toolbar. On the other hand, information about any command is available by typing To view the online documentation, select MATLAB Help from Help menu or MATLAB Help in the Command Window. The preferred method is to use the Help Browser. The Help Browser can be started by selecting the ? icon from the desktop toolbar. On the other hand, information about any command is available by typing

```
>> help command
```

Another way to get help is to use the `lookfor` command. The `lookfor` command differs from the `help` command. The help command searches for an exact function name match, while the lookfor command searches the quick summary information in each function for a match. For example, suppose that we were looking for a function to take the inverse of a matrix. Since MATLAB does not have a function named inverse, the command help inverse will produce nothing. On the other hand, the command lookfor inverse will produce detailed information, which includes the function of interest, inv.

```
>> lookfor inverse
```

NOTE - Because MATLAB is a huge program; it is impossible to cover all the details of each function one by one. However, we will give you information how to get help. Here are some examples:

- Use on-line `help` to request info on a specific function

  ```
  >> help sqrt
  ```

- In MATLAB version 6 and the current version (MATLAB version 7), the `doc` function opens the on-line version of the help manual. This is very helpful for more complex commands.

  ```
  >> doc plot
  ```

- Use `lookfor` to find functions by keywords. The general form is

  ```
  >> lookfor functionName
  ```

1.5 Exercises

1. Use the standard procedure on your computer to create a folder named *mywork* (or your own name) In MATLAB change to this folder either by using the command cd at the prompt or by clicking the ellipsis (...) button next to the Current Directory edit box on your MATLAB toolbar and browsing to the folder *mywork*. Look at the Current Directory edit box to make sure that *mywork* is the current directory. You can also use the command `pwd` at the prompt. Clear the window with the command `clc`, then clear the workspace of all variables with the command clear. Start a diary session with `diary lab1`. Continue to do the subsequent exercises (e.g., 2, 3, 4, etc.) When you are finished to solve these problems, enter the command `diary off`. Open the file `lab1` with your favorite editor - or open it in MATLAB's editor by typing `edit lab1` at the prompt. Edit and correct any mistakes that you made. Save and print the edited file and submit the result to your instructor.

2. Variables a, b, c, and d have been initialized to the following values $a = 3$, $b = 2$, $c = 5$, $d = 3$. Evaluate and compare the following MATLAB assignment statements:

   (a) output $= a(b + c)d$

   (b) output $= (ab) + (cd)$

   (c) output $= (a^b)^d$

   (d) output $= a^{(b^d)}$

3. Suppose that $x = 3$ and $y = 4$. Evaluate the expression:

$$\frac{x^2y^3}{(x-y)^2}$$

4. Compute the following quantities:

$$\frac{2^5}{2^5-1}; \quad 3\frac{\sqrt{5}-1}{(\sqrt{5}-1)^2} - 1$$

Chapter 2

Tutorial lessons 2

2.1 Mathematical functions

MATLAB contains a large set of mathematical functions. Typing `help elfun` and `help specfun` calls up full lists of *elementary* and *special* functions respectively.

There is a long list of mathematical functions that are *built* into MATLAB. These functions are called *built-ins*. Examples of very common functions are the *trigonometric* functions, *logarithms*, and *square roots*. Therefore, many standard mathematical functions, such as $\sin(x)$, $\cos(x)$, $\tan(x)$, e^x, $ln(x)$, are evaluated by the functions `sin, cos, tan, exp, log, sqrt(x)` in MATLAB.

Examples of other sophisticated functions include the *hyperbolic* functions, *Bessel* functions, and so forth. MATLAB has a variety of built-in functions ready for use. A few of the most common and useful MATLAB functions are shown in Tables 2.1, 2.2, 2.3, 2.6, and 2.5.

2.1.1 Elementary functions

Included are most of the functions that are standard in CALCULUS courses.

`sqrt(x)`	The square root of x, i.e. $\sqrt{x}$
`abs(x)`	The absolute of x, i.e. $\|x\|$
`sign(x)`	The signum of x, i.e. 0 if $x = 0$

Table 2.1: Elementary functions

`cos(x)`	The cosine of x, i.e. $\cos(x)$
`sin(x)`	The sine of x, i.e. $\sin(x)$
`tan(x)`	The tangent of x, i.e. $\tan(x)$
`cot(x)`	The cotangent of x, i.e. $\cot(x)$
`sec(x)`	The secant of x, i.e. $\sec(x)$
`csc(x)`	The cosecant of x, i.e. $\csc(x)$

Table 2.2: The trigonometric functions

`acos(x)`	The inverse of cosine of x, i.e. $\arccos(x)$
`asin(x)`	The sine of x, i.e. $\arcsin(x)$
`atan(x)`	The tangent of x, i.e. $\arctan(x)$

Table 2.3: The inverse trigonometric functions

`log(x)`	The natural logarithm of x, i.e. $\ln(x)$
`log10(x)`	The logarithm of x to base 10, i.e. $\log_{10}(x)$
`exp(x)`	The exponential of x, i.e. e^x

Table 2.4: The logarithm and exponential functions

`cosh(x)`	The hyperbolic cosine of x, i.e. $\cosh(x)$
`sinh(x)`	The hyperbolic sine of x, i.e. $\sinh(x)$
`tanh(x)`	The hyperbolic tangent of x, i.e. $\tanh(x)$
`coth(x)`	The hyperbolic cotangent of x, i.e. $\coth(x)$
`sech(x)`	The hyperbolic secant of x, i.e. sech(x)
`csch(x)`	The hyperbolic cosecant of x, i.e. csch(x)

Table 2.5: The hyperbolic functions

2.1.2 Predefined constant values

In addition to the elementary functions, MATLAB includes a number of predefined values. A list of the most common predefined constant values is given in **??**

`pi`	The π number, i.e. $\pi = 3.14159\ldots$
`i,j`	The imaginary unit i, i.e. $\sqrt{-1}$
`Inf`	The infinity, i.e. ∞
`NaN`	Not a number

Table 2.6: Predefined constant values

2.1.3 Examples

We illustrate here some examples which related to the elementary functions previously defined.

```
>> log(142)
ans   =
      4.9558

>> log10(142)
ans   =
      2.1523
```

For instance, to calculate $sin(\pi/4)$ and e^{10} expressions, we enter the following commands in MATLAB,

```
>> sin(pi/4)
ans   =
      0.7071

>> exp(10)
ans   =
      2.2026e+004
```

NOTES:

- Only use built-in functions on the right hand side of an expression. Reassigning the value of a built-in function can create problems.
- There are some exceptions. For example, `i` and `j` are pre-assigned to $\sqrt{-1}$. However, one or both of `i` or `j` are often used as loop indices. To avoid any possible confusion, it is suggested to use instead `ii` or `jj` as loop indices.

2.2 Basic plotting

MATLAB has an excellent set of graphic tools. In this section, we will present only some of the most elementary ones. We begin with two-dimensional (2D) graphs. The basic MATLAB graphing procedure in 2D is to take a vector of x-coordinates, $\mathbf{x} = (x_1, \ldots, x_N)$, and a vector of y-coordinates, $\mathbf{y} = (y_1, \ldots, y_N)$, locate the points (x_j, y_j), and then join them by straight lines. The MATLAB command is `plot(x,y)`.

2.2.1 Creating simple plots

The vectors $\mathbf{x} = (1, 2, 3, 4, 5, 6)$ and $\mathbf{y} = (3, -1, 2, 4, 5, 1)$ produce the picture shown in Figure 2.1.

```
>> x = [1 2 3 4 5 6];
>> y = [3 -1 2 4 5 1];
>> plot(x,y)
```

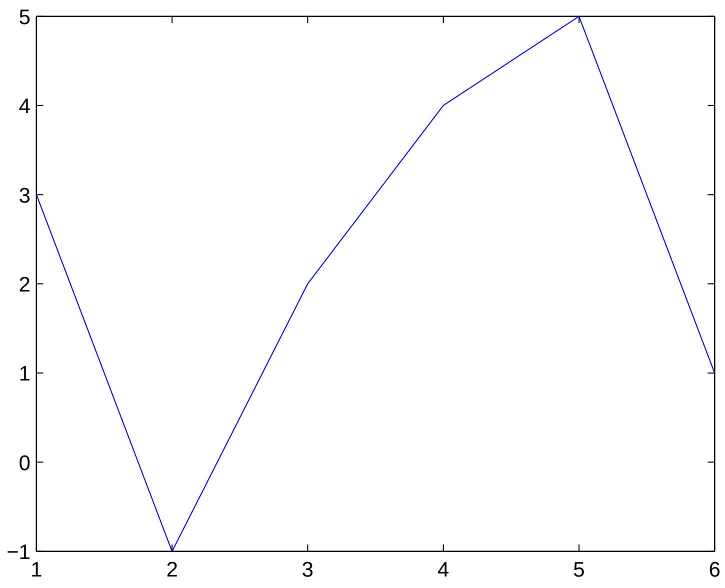


Figure 2.1: Plot for the vectors x and y

The `plot` functions has different forms depending on the input arguments. If `y` is a vector `plot(y)`produces a piecewise linear graph of the elements of `y` versus the index of the elements of `y`. If we specify two vectors, as mentioned above, `plot(x,y)` produces a graph of `y` versus `x`.

For example, to plot the function $\sin(x)$ on the interval $[0, 2\pi]$, we first create a vector of x values ranging from 0 to 2π, then compute the *sine* of these values, and finally plot the result:

```
>> x = 0:pi/100:2*pi;
>> y = sin(x);
>> plot(x,y)
```

NOTES:

- `0:pi/100:2*pi` yields a vector that
  - starts at 0,
  - takes steps (or increments) of $\pi/100$,
  - stops when 2π is reached.
- If you omit the increment, MATLAB automatically increments by 1.

Now label the axes and add a title. The character `\pi` creates the symbol π.

```
>> xlabel('x = 0:2\pi')
>> ylabel('Sine of x')
>> title('Plot of the Sine function')
```

The color of a single curve is, by default, blue, but other colors are possible. The desired color is indicated by a third argument. For example, red is selected by `plot(x,y,'r')`. Note the single quotes around `r`.

2.2.2 Multiple data sets in one graph

Multiple (x, y) *pairs* arguments create *multiple* graphs with a single call to *plot*. For example, these statements plot three related functions of x: $y_1 = \sin(x)$, $y_2 = \sin(x - 0.25)$, and $y_3 = \sin(x - 0.50)$, in the interval $0 \leq x \leq 2\pi$.

```
>> x = 0:pi/100:2*pi;
>> y1 = sin(x);
>> y2 = sin(x-0.25);
>> y3 = sin(x-0.50);
>> plot(x,y1,x,y2,x,y3)
>> legend('sin(x)','sin(x-0.25)','sin(x-0.50)')
>> xlabel('x')
>> ylabel('Sine functions')
```

The result is shown in Figure 2.3

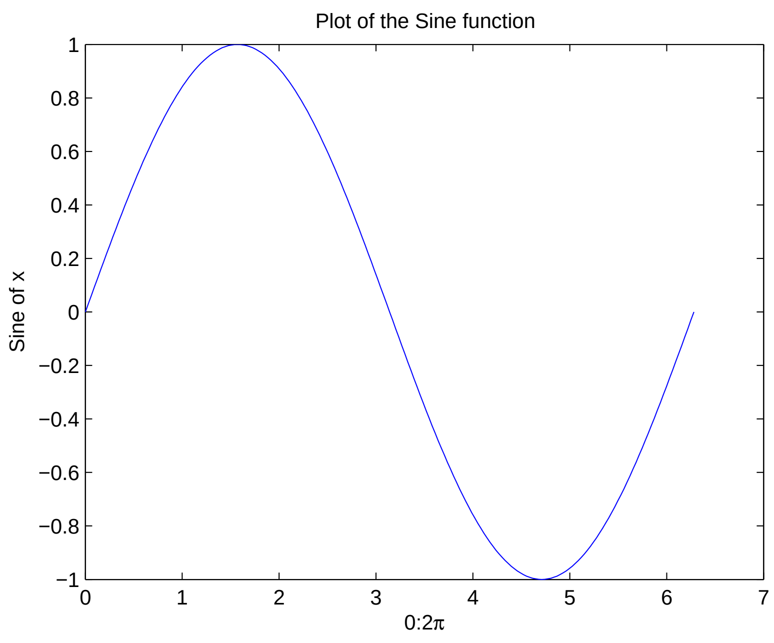


Figure 2.2: Example of 2D function plot

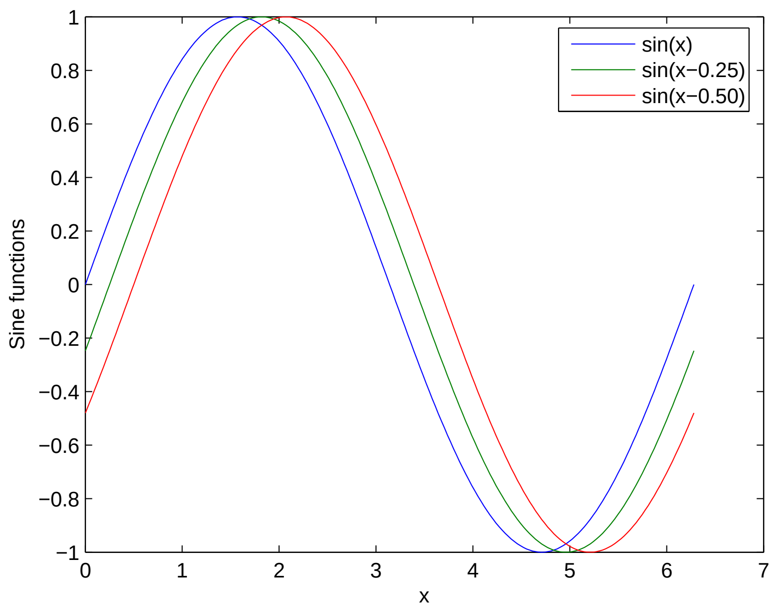


Figure 2.3: Example of multiple plots in one graph

2.2.3 Specifying line styles and colors

It is possible to specify line styles, colors, and markers (e.g., circles, plus signs) using the `plot` command:

```
plot(x,y,'style_color_marker')
```

`style_color_marker` is a string containing from one to four characters constructed from a line style, a color, and a marker type:

- `Lines`: "-" for solid, "–" for dashed, ":" for dotted, "-." for dashed dot.
- `Colors`: `c`, `m`, `y`, `r`, `g`, `b`, `w`, and `k` correspond to cyan, magenta, yellow, red, green, blue, white, and black.
- `Markers`: $+$, $\circ$, $*$, and $\times$.

2.3 Exercises

1. Evaluate the fractions
$$\frac{22}{7}, \quad \frac{311}{99}, \text{ and } \quad \frac{355}{113}$$
and determine which is the best approximation to π.

2. Evaluate to 15 digits:
$$\text{(a) } \frac{\sin(0.1)}{0.1}, \quad \text{(b) } \frac{\sin(0.01)}{0.01}, \quad \text{(c) } \frac{\sin(0.0001)}{0.0001}$$

3. Graph the following:
$$\begin{array}{l}\text{(a) } y = x^2 + x - 1, \quad \text{for } -5 \le x \le 5 \\ \text{(b) } y = \tan x, \quad \text{for } -\pi/2 \le x \le \pi/2\end{array}$$

4. Plot the functions x^4 and 2^x on the same graph. Determine how many times their graphs intersect. (*Hint*: You will probably have to make several plots, using various intervals, in order to find all the intersection points).

Chapter 3

Matrix operations

3.1 Introduction

Matrices are the basic elements of the MATLAB environment. A matrix is a two-dimensional array consisting of m rows and n columns. Special cases are *column vectors* ($n = 1$) and row vectors ($m = 1$).

In this section we will illustrate how to apply different operations on matrices. MATLAB supports two types of operations, known as *matrix operations* and *array operations*.

In this section, the following topics are discussed: vectors and matrices in MATLAB, the inverse of a matrix, determinants, and matrix manipulation.

3.2 Matrix generation

Matrices are fundamental to MATLAB. Therefore, you need to become familiar with matrix generation and manipulation. Matrices can be generated in several ways. A vector is a special case of a matrix.

3.2.1 Entering a vector

The purpose of this section is to show how to create vectors and matrices in MATLAB.

This command creates a row vector

```
>> a = [1 2 3]
a  =
     1     2     3
```

Column vectors are created in a similar way, however, semicolon (;) must separate the components of a column vector

```
>> b = [1;2;3]
b  =
      1
      2
      3
```

3.2.2 Entering a matrix

A matrix is an array of numbers. To type a matrix into MATLAB you must

- begin with a square bracket, [
- separate elements in a row with spaces or commas (,)
- use a semicolon (;) to separate rows
- end the matrix with another square bracket,].

Here is a typical example. To enter a matrix **A**, such as,

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \tag{3.1}$$

we will enter each element of the matrix **A** as follow,

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

MATLAB then displays the matrix we just entered as

```
A    =
     1    2    3
     4    5    6
     7    8    9
```

Once we have entered the matrix, it is automatically stored and remembered in the *workspace*. We can refer to it simply as matrix `A`. We can then view a particular element in a matrix by specifying its location:

```
>> A(2,1)
ans  =
      4
```

`A(2,1)` is an element located in the second row and first column. Its value is 4.

3.2.3 Indexing

Once a matrix exists, it elements are accessed by specifying row and column indices. The element of row i and column j of the matrix **A** is denoted by `A(i,j)`. Thus, `A(i,j)` in MATLAB refers to the element A_{ij} of matrix **A**. The *first* index is the *row* number and the *second* index is the *column* number. For example, `A(1,3)` is an element of *first* row and *third* column. Here, `A(1,3)=3`.

Correcting any entry is easy through indexing. Here we substitute `A(3,3)=9` by `A(3,3)=0`. The result is

```
>> A(3,3) = 0
A   =
     1    2    3
     4    5    6
     7    8    0
```

Single elements of a matrix are accessed as `A(i,j)`, where $i \geq 1$ and $j \geq 1$ (zero or negative subscripts are not supported in MATLAB).

3.2.4 Colon operator and Linear spacing

Often we must deal with matrices or vectors that are too large to enter one element at a time. For example, suppose we want to enter a vector x consisting of points $(0, 0.1, 0.2, 0.3, \cdots, 5)$. we can use the command

```
>> x = 0:0.1:5;
```

This row vector has 51 elements. The colon operator can also be used to pick out a certain row or column. For example, the statement `A(m:n,k:l` specifies rows m to n and column k to l. The colon operator will prove very useful and understanding how it works is the key to efficient and convenient usage of MATLAB. It occurs in several different forms. Subscript expressions refer to portions of a matrix. For example,

```
>> A(2,:)
ans  =
      4   5   6
```

is the second row elements of **A**.

The colon operator can also be used to extract a sub-matrix from **A**.

```
>> A(:,2:3)
```

```
ans  =
     2   3
     5   6
     8   0
```

`A(:,2:3)` is a sub-matrix with the last two columns of **A**.

A row or a column of a matrix can be deleted by setting it to a *null* vector, [ ].

```
>> A(:,2)=[]
ans  =
     1   3
     4   6
     7   0
```

On the other hand, there is a command to generate linearly spaced vectors: `linspace`. It is similar to the colon operator (`:`), but gives direct control over the number of points. For example,

```
y = linspace(a,b)
```

generates a row vector `y` of 100 points linearly spaced between and including `a` and `b`.

```
y = linspace(a,b,n)
```

generates a row vector `y` of `n` points linearly spaced between and including `a` and `b`. This is useful when we want to divide an interval into a number of subintervals of the same length. For example,

```
>> theta = linspace(0,2*pi,101)
```

divides the interval $[0, 2\pi]$ into 100 equal subintervals, then creating a vector of 101 elements.

3.2.5 Creating a sub-matrix

To extract a *submatrix* **B** consisting of rows 2 and 3 and columns 1 and 2 of the matrix **A**, do the following

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

To interchange rows 1 and 2 of **A**, use the vector of row indices together with the colon operator.

```
>> C = A([2 1 3],:)
C    =
     4    5    6
     1    2    3
     7    8    0
```

It is important to note that the *colon operator* (:) stands for *all columns* or *all rows.* To create a vector version of matrix **A**, do the following

```
>> A(:)
ans =
    1
    2
    3
    4
    5
    6
    7
    8
    0
```

The submatrix comprising the intersection of rows `p` to `q` and columns `r` to `s` is denoted by `A(p:q,r:s)`.

As a special case, a colon (:) as the row or column specifier covers all entries in that row or column; thus

- `A(:,j)` is the `j`th column of `A`, while
- `A(i,:)` is the `i`th row, and
- `A(end,:)` picks out the last row of `A`.

The keyword `end`, used in `A(end,:)`, denotes the last index in the specified dimension. Here are some examples.

```
>> A
A   =
    1    2    3
    4    5    6
    7    8    9
```

```
>> A(2:3,2:3)
ans =
    5    6
    8    9

>> A(end:-1:1,end)
ans =
    9
    6
    3

>> A([1 3],[2 3])
ans =
    2    3
    8    9
```

3.2.6 Deleting row or column

To delete a row or column of a matrix, use the *empty vector* operator, [ ]. For the (full) matrix **A** from the previous example

```
>> A(3,:) = []
A  =
   1    2    3
   4    5    6
```

Third row of matrix **A** is now deleted. To restore the third row, we use a technique for creating a matrix

```
>> A = [A(1,:);A(2,:);[7 8 0]]
A  =
   1    2    3
   4    5    6
   7    8    0
```

Matrix **A** is now restored to its original form.

3.2.7 Dimension

To determine the *dimensions* of a matrix or vector, use the command `size`, as follows:

```
>> size(A)
ans   =
      3     3
```

means 3 rows and 3 columns.

Or more explicitly with,

```
>> [m,n]=size(A)
```

3.2.8 Continuation

If it is not possible to type the entire input on the same line, use consecutive periods, called an ellipsis $\cdots$, to signal continuation, then continue the input on the next line.

```
B = [4/5        7.23*tan(x)       sqrt(6); ...
     1/x^2      0                 3/(x*log(x)); ...
     x-7        sqrt(3)           x*sin(x)];
```

Note that *blank* spaces around $+$, $-$, $=$ signs are optional, but they improve readability.

3.2.9 Special cases: vectors and scalars

1. A vector is a special case of a matrix with just one row or one column. It is entered the same way as matrix. EXAMPLES:
   (a) `u = [3 4 5]` produces a row vector.
   (b) `v = [3;4;5]` produces a column vector.
   (c) `w = []` produces a null vector.
2. A scalar does not need brackets.
   - `g = 9.81`

3.2.10 Entering vector elements

For manual entry, the elements in a vector are enclosed in square brackets ([ ]). When creating a row vector, separate elements with a space or a comma. For example, a vector **v** could be

$$\mathbf{v} = \begin{bmatrix} 8 & 2 & \sqrt{9} & \pi & 7.1 \end{bmatrix}$$

or,

$$\mathbf{w} = \begin{bmatrix} 3 \\ \sqrt{\pi} \\ -2.4 \\ -7 \\ 1 \end{bmatrix}$$

We say that **v** is a *row* vector and that **w** is a *column* vector. In MATLAB, these will be

```
>> v = [8 2 sqrt(9) pi 7.1]
v =
  8.0000  2.0000  3.0000  3.1416  7.1000
```

and,

```
>> w = [3 sqrt(pi) -2.4 -7 1]'
w =
   3.0000
   1.7725
  -2.4000
  -7.0000
   1.0000
```

The *dimension* of a matrix are the number of *rows* and the number of *columns*, with the number of rows usually given first. The above matrix `A` is 3×3 matrix. The vector `v` is 1×5 matrix, and the column vector `w` is 5×1 matrix. A single number, such as 1.7725, is a scalar and can be considered as a 1×1 matrix.

3.2.11 Transposing a matrix

The *transpose* operation is denoted by an apostrophe or a single quote ('). It flips a matrix about its main diagonal and it turns a row vector into a column vector. Thus,

```
>> A'
ans =
    1   4   7
    2   5   8
    3   6   0
```

By using linear algebra notation, the transpose of $m \times n$ real matrix **A** is the $n \times m$ matrix that results from interchanging the rows and columns of **A**. The transpose matrix is denoted $\mathbf{A}^T$.

3.2.12 Concatenating matrices

Matrices can be made up of sub-matrices. Here is a typical example. First, let's recall our previous matrix A.

```
A   =
    1   2   3
    4   5   6
    7   8   9
```

The new matrix B will be,

```
>> B = [A 10*A; -A [1 0 0; 0 1 0; 0 0 1]]
B   =
    1    2    3    10    20    30
    4    5    6    40    50    60
    7    8    9    70    80    90
   -1   -2   -3     1     0     0
   -4   -5   -6     0     1     0
   -7   -8   -9     0     0     1
```

3.2.13 Matrix generators

MATLAB provides functions that generates elementary matrices. The matrix of zeros, the matrix of ones, and the identity matrix are returned by the functions `zeros`, `ones`, and `eye`, respectively.

Matrix	**Description**
`eye(m,n)`	Returns an m-by-n matrix with 1 on the main diagonal
`eye(n)`	Returns an n-by-n square identity matrix
`zeros(m,n)`	Returns an m-by-n matrix of zeros
`ones(m,n)`	Returns an m-by-n matrix of ones
`diag(A)`	Extracts the diagonal of matrix A
`rand(m,n)`	Returns an m-by-n matrix of random numbers

Table 3.1: Elementary matrices

For a complete list of *elementary matrices* and *matrix manipulations*, type `help elmat`. Here are some examples:

1.
```
>> b=ones(3,1)
```

`length`	Length of a vector
`size`	Size of an array
`disp`	Display matrix or text
`ndims`	Number of dimensions
`numel`	Number of elements
`isempty`	True for empty array
`isequal`	True if arrays are numerically equal

Table 3.2: Basic array information

`cat`	Concatenate arrays
`reshape`	Change size
`diag`	Diagonal matrices and diagonals of matrix
`find`	Find indices of nonzero elements
`end`	Last index
`:`	Regularly spaced vector and index into matrix

Table 3.3: Matrix manipulation

```
b   =
    1
    1
    1
```

Equivalently, we can define `b` as `>> b=[1;1;1]`

2.
```
>> eye(3)
ans   =
    1    0    0
    0    1    0
    0    0    1
```

3.
```
>> c=zeros(2,3)
c   =
    0    0    0
    0    0    0
```

In addition, it is important to remember that the three elementary operations of *addition* (+), *subtraction* (−), and *multiplication* (∗) apply also to matrices whenever the dimensions are *compatible*.

Two other important matrix generation functions are `rand` and `randn`, which generate matrices of (pseudo-)random numbers using the same syntax as `eye`.

In addition, matrices can be constructed in block form. With `C` defined by `C = [1 2; 3 4]`, we may create a matrix `D` as follows

```
>> D = [C zeros(2); ones(2) eye(2)]
D =
  1     2     0     0
  3     4     0     0
  1     1     1     0
  1     1     0     1
```

3.2.14 Special matrices

MATLAB provides a number of special matrices (see Table 3.4. These matrices have interesting properties that make them useful for constructing examples and for testing algorithms. For more information, see MATLAB documentation.

`hilb`	Hilbert matrix
`invhilb`	Inverse Hilbert matrix
`magic`	Magic square
`pascal`	Pascal matrix
`toeplitz`	Toeplitz matrix
`vander`	Vandermonde matrix
`wilkinson`	Wilkinson's eigenvalue test matrix

Table 3.4: Special matrices

3.3 Exercises

1. Enter the following three matrices:

$$\mathbf{A} = \begin{bmatrix} 2 & 6 \\ 3 & 9 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} -5 & 5 \\ 5 & 3 \end{bmatrix}$$

   Check the following linear algebra rules:

   (a) Is matrix addition commutative? Compute **A**+**B** and **B**+**A**.

   (b) Is matrix addition associative? Compute **(A+B)+C** and **A+(B+C)**.

   (c) Is multiplication with a scalar distributive? Compute $\alpha(\mathbf{A}+\mathbf{B})$ and $\alpha\mathbf{A}+\alpha\mathbf{B}$ Take $\alpha = 5$ and show that the results are the same.

(d) Is multiplication with a matrix distributive? Compute $\mathbf{A} * (\mathbf{B} + \mathbf{C})$ and compare with $\mathbf{A} * \mathbf{B} + \mathbf{A} * \mathbf{C}$.

2. Determine the size and contents of the following arrays:

   (a) $a = 1 : 2 : 5$

   (b) $b = [a', a', a']$

   (c) $c = b(1 : 2 : 3, 1 : 2 : 3)$

   (d) $d = a + b(2, :)$

   (e) $e = [\text{zeros}(1, 3), \text{ones}(3, 1)', 3 : 5']$

3. Create the following matrices with the help of the matrix generation functions `zeros`, `eye`, and `ones`.

$$\mathbf{D} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{E} = \begin{bmatrix} 5 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 5 \end{bmatrix}, \quad \mathbf{F} = \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}$$

4. Create a big matrix with sub-matrices (concatenating matrices). Putting matrices **A**, **B**, and **C** given above on its diagonal to create the following matrix **G**, as follows,

$$\mathbf{G} = \begin{bmatrix} 2 & 6 & 0 & 0 & 0 & 0 \\ 3 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 0 & 0 \\ 0 & 0 & 3 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & -5 & 5 \\ 0 & 0 & 0 & 0 & 5 & 3 \end{bmatrix}$$

5. Construct a matrix **M** using the function `diag`, as well as colon (:) and transpose (′) operators.

$$\mathbf{M} = \begin{bmatrix} 1 & 2 & \dots & 10 \\ 2 & 2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 10 & 0 & \dots & 10 \end{bmatrix}$$

   where the ellipsis $\dots$, $\vdots$, and $\ddots$ indicate the series of numbers; i.e., $1\ 2 \dots 10$ is equivalent to 1 2 3 4 5 6 7 8 9 10.

6. Construct a matrix **T** using the functions `diag` and `ones`.

$$\mathbf{T} = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 \end{bmatrix}$$

7. The matrix is the $n - \text{by} - n$ upper triangular matrix A with elements

$$a_{ij} = \begin{cases} -1, & \text{if } i < j \\ 1, & \text{if } i = j \\ 0, & \text{if } i > j \end{cases}$$

Show how to generate this matrix in MATLAB.

Chapter 4

Array operations and Linear equations

MATLAB has two different types of arithmetic operations: matrix arithmetic operations and array arithmetic operations.

4.1 Array operations

MATLAB has two different types of arithmetic operations: matrix arithmetic operations and array arithmetic operations.

4.1.1 Matrix arithmetic operations

As we mentioned earlier, MATLAB allows arithmetic operations: $+$, $-$, $*$, and ˆ to be carried out on matrices. Thus,

A+B or **B+A**	is valid if **A** and **B** are of the same size
A*B	is valid if **A**'s number of column equals **B**'s number of rows
Aˆ2	is valid if **A** is square and equals **A*A**
α***A** or **A***α	multiplies each element of **A** by α

4.1.2 Array arithmetic operations

On the other hand, array arithmetic operations or *array operations* for short are done *element-by-element*. The period character (.) distinguishes the array operations from the matrix operations. However, since the matrix and array operations are the same for addition

(+) and subtraction (-), the character pairs (.+) and (.-) are not used. The list of array operators is shown below in Table 4.2.

Operators	**Description**
`.*`	Element-by-element multiplication
`./`	Element-by-element division
`.^`	Element-by-element exponentiation

Table 4.1: Array operators

If **A** and **B** are two matrices of the same size with elements $\mathbf{A} = [a_{ij}]$ and $\mathbf{B} = [b_{ij}]$, then the command

```
>> C = A.*B
```

produces another matrix **C** of the same size with elements $c_{ij} = a_{ij}b_{ij}$. For example, using the same 3×3 matrices,

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \qquad \mathbf{B} = \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{bmatrix}$$

we have

```
>> C = A.*B
C   =
     10     40     90
    160    250    360
    490    640    810
```

To raise a scalar to a power, we use for example the command `10^2`. If we want the operation to be applied to each element of a matrix, we use `.^2`. For example, if we want to produce a new matrix whose elements are the square of the elements of the matrix **A**, we enter

```
>> A.^2
ans    =
      1     4     9
     16    25    36
     49    64    81
```

The relations below summarize the above operations. To simplify, let's consider two vectors **U** and **V** with elements $\mathbf{U} = [u_i]$ and $\mathbf{V} = [v_j]$.

$\mathbf{U}.*\mathbf{V}$	produces	$[u_1v_1\ u_2v_2 \ldots\ u_nv_n]$
$\mathbf{U}./\mathbf{V}$	produces	$[u_1/v_1\ u_2/v_2 \ldots\ u_n/v_n]$
$\mathbf{U}.\hat{}\mathbf{V}$	produces	$[u_1^{v_1}\ u_2^{v_2} \ldots\ u_n^{v_n}]$

4.1.3 Summary

In order to avoid any confusion on this topic of "Matrix and Array Operations", the above description can be summarized in Table 4.2.

Operation	**Matrix**	**Array**
Addition	+	+
Subtraction	−	−
Multiplication	*	.*
Division	/	./
Left division	\	.\
Exponentiation	^	.^

Table 4.2: Summary of matrix and array operations

4.2 Solving linear equations

One of the problems encountered most frequently in scientific computation is the solution of systems of simultaneous linear equations.

With matrix notation, a system of simultaneous linear equations is written

$$\mathbf{Ax} = \mathbf{b} \tag{4.1}$$

when there are as many equations as unknown, $\mathbf{A}$ is a given square matrix of order n, $\mathbf{b}$ is a given column vector of n components, and $\mathbf{x}$ is an unknown column vector of n components.

Students of linear algebra learn that the solution to $\mathbf{Ax} = \mathbf{b}$ can be written as $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, where $\mathbf{A}^{-1}$ is the inverse of $\mathbf{A}^{-1}$ is the inverse of $\mathbf{A}$. However, in the vast majority of practical computational problems, it is unnecessary and inadvisable to actually compute $\mathbf{A}^{-1}$. The inverse requires more arithmetic and produces a less accurate answer.

For example, consider the following system of linear equations

$$\begin{cases} x + 2y + 3z &= 1 \\ 4x + 5y + 6z &= 1 \\ 7x + 8y &= 1 \end{cases}$$

The coefficient matrix **A** is

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \text{and the vector} \quad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

With matrix notation, a system of simultaneous linear equations is written

$$\mathbf{Ax} = \mathbf{b} \tag{4.2}$$

This equation can be solved for **x** using linear algebra. The result is $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$.

There are typically two ways to solve for **x** in MATLAB:

1. The first one is to use the matrix inverse, `inv`.

```
>> A = [1 2 3; 4 5 6; 7 8 0];
>> b = [1; 1; 1];
>> x = inv(A)*b
x   =
    -1.0000
     1.0000
    -0.0000
```

2. The second one is to use the *backslash* (\)operator. The numerical algorithm behind this operator is computationally efficient. This is a numerically reliable way of solving system of linear equations by using a well-known process of Gaussian elimination.

```
>> A = [1 2 3; 4 5 6; 7 8 0];
>> b = [1; 1; 1];
>> x = A\b
x   =
    -1.0000
     1.0000
    -0.0000
```

This problem is at the heart of many problems in scientific computation. Hence it is important that we know how to solve this type of problem efficiently.

Now, we know how to solve a system of linear equations. In addition to this, we will see also some additional details which relate to this particular topic.

4.2.1 Matrix inverse

Let's consider the same matrix $\mathbf{A}$.

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$$

Calculating the inverse of $\mathbf{A}$ manually is probably not a pleasant work. Here the hand-calculation of $\mathbf{A}^{-1}$ gives as a final result:

$$\mathbf{A}^{-1} = \frac{1}{9} \begin{bmatrix} -16 & 8 & -1 \\ 14 & -7 & 2 \\ -1 & 2 & -1 \end{bmatrix}$$

In MATLAB, however, it becomes as simple as the following commands:

```
>> A = [1 2 3; 4 5 6; 7 8 0];
>> inv(A)
ans =
    -1.7778    0.8889   -0.1111
     1.5556   -0.7778    0.2222
    -0.1111    0.2222   -0.1111
```

which is similar to:

$$\mathbf{A}^{-1} = \frac{1}{9} \begin{bmatrix} -16 & 8 & -1 \\ 14 & -7 & 2 \\ -1 & 2 & -1 \end{bmatrix}$$

and the determinant of $\mathbf{A}$ is

```
>> det(A)
ans  =
     27
```

For further details on applied numerical linear algebra, see [10] and [11].

4.2.2 Matrix functions

MATLAB provides many matrix functions for various matrix/vector manipulations; see Table **??** for some of these functions. Use the online help of MATLAB to find how to use these functions.

`det`	Determinant
`diag`	Diagonal matrices and diagonals of a matrix
`eig`	Eigenvalues and eigenvectors
`inv`	Matrix inverse
`norm`	Matrix and vector norms
`rank`	Number of linearly independent rows or columns

Table 4.3: Matrix functions

4.3 Exercises

1. Assume that **a**, **b**, **c**, and **d** are defined as follows:

$$\mathbf{a} = \begin{bmatrix} 2 & -2 \\ -1 & 2 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \quad \mathbf{d} = \text{eye}(2)$$

   Calculate the results of the following operations and then compare them between matrix and array operations.

   (a) $\text{result1} = \mathbf{a} * \mathbf{d}$, $\text{result2} = \mathbf{a}. * \mathbf{d}$

   (b) $\text{result3} = \mathbf{a} * \mathbf{c}$, $\text{result4} = \mathbf{a}. * \mathbf{c}$

   (c) $\text{result5} = \mathbf{a} \backslash \mathbf{b}$, $\text{result6} = \mathbf{a}. \backslash \mathbf{b}$

2. Consider the system of equations:

$$\begin{cases} x + 2y + 3z & = & 1 \\ 3x + 3y + 4z & = & 1 \\ 2x + 3y + 3z & = & 2 \end{cases}$$

   (a) On paper, find the solution **x** to the system of equations

   (b) Check your hand-calculation with MATLAB.

3. Suppose that, in MATLAB, you have an $n-\text{by}-n$ matrix **A** and an $n-\text{by}-1$ matrix **b**. What do $\mathbf{A}\backslash\mathbf{b}$, $\mathbf{b}'/\mathbf{A}$, and $\mathbf{A}/\mathbf{b}$ mean in MATLAB. How does $\mathbf{A}\backslash\mathbf{b}$ differ from $\text{inv}(\mathbf{A}) * \mathbf{b}$?

Chapter 5

Script files

5.1 Introduction

This section covers the following topics:

- M-File Scripts
- M-File Functions

So far in this manual, all the commands were executed in the Command Window. The problem is that the commands in the Command Window cannot be saved and executed again. Therefore, a different way of executing commands with MATLAB is

- first, to create a file with a list of commands,
- then, save it, and
- finally, run the file.

If needed, corrections or changes can be made to the commands in the file. The files that are used for this purpose are called script files.

5.2 M-File Scripts

A *script file* is an external file that contains a sequence of MATLAB statements. By typing the *filename*, you can obtain subsequent MATLAB input from the file. Script files have a filename extension `.m` and are often called M-files.

Script files (or *script* for short) is the simplest type of M-file, because they have no *input* or *output* arguments. They are useful for *automating* series of MATLAB commands, such as computations that you have to perform repeatedly from the command line (or Command Window).

5.2.1 Simple script examples

When we write a program in MATLAB, we save it to a file called an M-file (named after its `.m` extension). Here are two simple script examples.

Example 1

Consider the system of equations:

$$\begin{cases} x + 2y + 3z &= 1 \\ 3x + 3y + 4z &= 1 \\ 2x + 3y + 3z &= 2 \end{cases}$$

Find the solution **x** to the system of equations.

SOLUTION:

- Use an *editor* to create a file: **File** → **New** → **M-file**.
- Enter the following statements in the file:

```
A = [1 2 3; 3 3 4; 2 3 3];
b = [1; 1; 2];
x = A\b
```

- Save the file, for example, `example1.m`.
- Run the file, in the command line, by typing:

```
>> example1
x   =
    -0.5000
     1.5000
    -0.5000
```

When execution completes, the variables (`A`, `b`, and `x`) remain in the workspace. To see a listing of them, enter `whos` at the command prompt.

Example 2

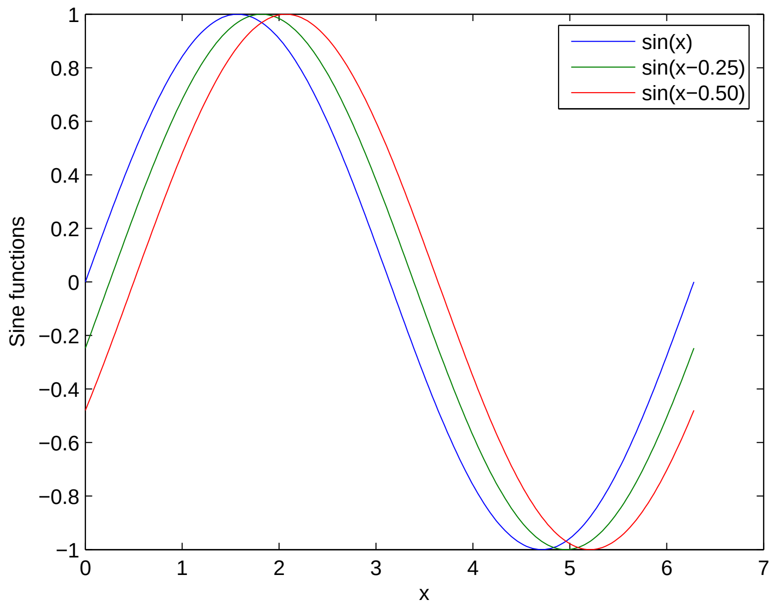
Plot the following sine functions, $y_1 = \sin(x)$, $y_2 = \sin(x - 0.25)$, and $y_3 = \sin(x - 0.50)$, in the interval $0 \leq x \leq 2\pi$.

SOLUTION:

- Create a file, `example2.m`, as follows:

```
x = 0:pi/100:2*pi;
y1 = sin(x);
y2 = sin(x-0.25);
y3 = sin(x-0.50);
plot(x,y1,x,y2,x,y3)
legend('sin(x)','sin(x-0.25)','sin(x-0.50)')
xlabel('x')
ylabel('Sine functions')
```

- Display the plot



Script side-effects

All variables created in a script file are added to the workspace. This may have undesirable effects, because:

- Variables already existing in the workspace may be overwtitten.
- The execution of the script can be affected by the state variables in the workspace.

Because scripts have side effects, it is better to code any complicated applications in a function M-file.

5.3 M-File Functions

Functions are program *routines* that accept *input* arguments and return *output* argument. Each M-file function (or *function* or *M-file* for short) has its own area of workspace, separated from the MATLAB base workspace.

5.3.1 Basic parts of an M-File

This simple function shows the basic parts of an M-file.

```
function f = factorial(n)                (1)
% Compute a factorial value.             (2)
% FACT(N) returns the factorial of N.    (3)
f = prod(1:n);                           (4)
```

The table below briefly describes each of these M-file parts. The *first* of a function M-file

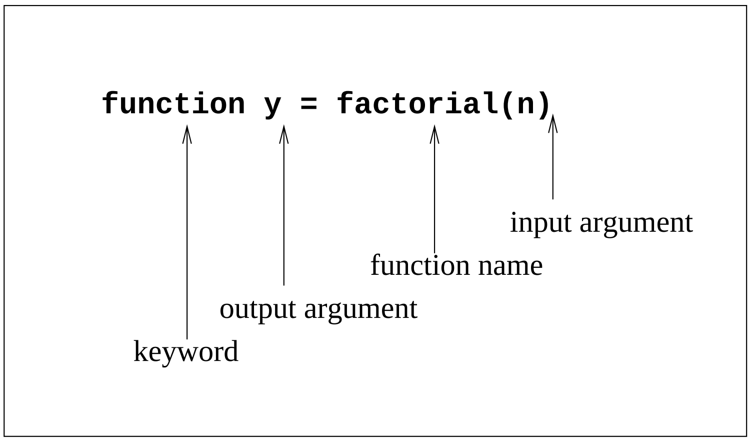


Figure 5.1: M-file parts

starts with the keyword `function`. Its give the function *name* and order of *arguments*. In the case of function `factorial`, there are up to one output argument and one input argument. Both *functions* and *scripts* can have have all of these parts, except for the *function definition*

Part no.	**M-file element**	**Description**
`(1)`	Function definition line	Define the function name, and the number and order of input and output arguments
`(2)`	H1 line	A one line summary description of the program, displayed when you request Help
`(3)`	Help text	A more detailed description of the program
`(4)`	Function body	Program code that performs the actual computations

line which applies to *function* only. In addition, it is important to note that *function name* must begin with a letter, and must be no longer than than the maximum of 63 characters. Furthermore, the name of the text file that you save will consist of the function name with the extension `.m`. Thus, the above example file would be `factorial.m`.

Therefore, the differences between *scripts* and *functions* are summarized in the table below.

Scripts	**Functions**
- Do not accept input arguments or return output arguments.	- Can accept input arguments and return output arguments.
- Store variables in a workspace that is shared with other scripts	- Store variables in a workspace internal to the function.
- Are useful for automating a series of commands	- Are useful for extending the MATLAB language for your application

5.4 Input to a script file

When a script file is executed, the variables that are used in the calculations within the file must have assigned values. The assignment of a value to a variable can be done in three ways.

1. The variable is defined in the script file.
2. The variable is defined in the command prompt.

3. The variable is entered when the script is executed.

We have already seen the two first cases. Here, we will focus our attention on the third one. In this case, the variable is defined in the script file. When the file is executed, the user is *prompted* to assign a value to the variable in the command prompt. This is done by using the `input` command. Here is an example.

```
% This script file calculates the average of points
% scored in three games.
% The point from each game are assigned to a variable
% by using the ‘input’ command.

game1 = input('Enter the points scored in the first game ');
game2 = input('Enter the points scored in the second game ');
game3 = input('Enter the points scored in the third game ');
average = (game1+game2+game3)/3
```

The following shows the command prompt when this script file (saved as ch5ex2) is executed.

```
>> ch5ex2
>> Enter the points scored in the first game   15
>> Enter the points scored in the second game  23
>> Enter the points scored in the third game   10

average =
        16
```

The `input` command can also be used to assign *string* to a variable. For more information, see MATLAB documentation.

5.5 Output commands

As discussed before, MATLAB automatically generates a *display* when commands are executed. In addition to this automatic display, MATLAB has several commands that can be used to generate displays or outputs. Two commands that are frequently used to generate output are: `disp` and `fprintf`. The main differences between these two commands can be summarized as follows. Therefore, output to the Command Window is achieved with either the `disp` command or the `fprintf` command. On the other hand, output to a file requires the `fprintf` command.

`disp`	. Simple to use. . Provide limited control over the appearance of output
`fprintf`	. Slightly more complicated than `disp`. . Provide total control over the appearance of output

5.5.1 Summary of input and output arguments

As mentioned above, the input arguments are listed inside parentheses following the function name. The output arguments are listed inside the brackets on the left side. They are used to transfer the output from the function file. Function file can have none, one, or several output arguments. Table 5.1 illustrates some possible combinations of input and output arguments.

`function C=FtoC(F)`	One input argument and one output argument
`function area=traparea(a,b,h)`	three inputs and one output
`function [h,d]=motion(v,angle)`	Two inputs and two outputs

Table 5.1: Example of input and output arguments

5.5.2 Examples

The `fprintf` command

The `fprintf` command can be used to display output (text and data) on the screen or to save it to a file. With this command, unlike the `disp` command, the output can be formatted. With many available options, the `fprintf` can be long and complicated. Recall the above example.

```
% This script file calculates the average of points
% scored in three games.
% Here we will use 'fprintf' command for output.

game(1) = input('Enter the points scored in the first game ');
game(2) = input('Enter the points scored in the second game ');
game(3) = input('Enter the points scored in the third game ');
average = mean(game);
fprintf('An average of %8.4f points was scored in 3 games ',average)
```

```
>> ch5ex2
Enter the points scored in the first game 15
Enter the points scored in the second game 23
Enter the points scored in the third game 10
An average of  16.0000 points was scored in 3 games
```

Here is another simplified example.

```
>> x = 2;
>> fprintf('The square root of %g is %8.5f\n',x,sqrt(x))

The square toot of x is 1.41421
```

The following table shows the basic conversion codes. In addition to MATLAB documentation, we would recommend a typical example of application described in our recent paper [12].

code	**Description**
`%f`	Format as a floating-point value
`%e`	Format as a floating-point value in scientific notation
`%g`	Format in the most compact form of either `%f` or `%e`
`%s`	Format as a string
`\n`	Insert new line in output string
`\t`	Insert tab in output string

5.6 Exercises

1. Liz buys three apples, a dozen bananas, and one cantaloupe for \$2.36. Bob buys a dozen apples and two cantaloupe for \$5.26. Carol buys two bananas and three cantaloupe for \$2.77. How much do single pieces of each fruit cost?

2. Write a function file that converts temperature in degrees Fahrenheit (°F) to degrees Centigrade (°C). Use `input` and `fprintf` commands to display a mix of text and numbers. Recall the conversion formulation, $\mathrm{C} = 5/9 * (\mathrm{F} - 32)$.

3. Write a user-defined MATLAB function, with two input and two output arguments that determines the height in centimeters (`cm`) and mass in kilograms (`kg`)of a person from his height in inches (`in.`) and weight in pounds (`lb`).

   (a) Determine in SI units the height and mass of a 5 ft.15 in. person who weight 180 lb.

   (b) Determine your own height and weight in SI units.

Chapter 6

MATLAB programming

6.1 Introduction

MATLAB is also a *programming language*. By creating a file with the extension `.m`, we can easily write and run programs. We do not need to *compile* the program since MATLAB is an interpretative (not compiled) language. MATLAB has thousand of *functions*, and you can add your own using m-files.

MATLAB provides several tools that can be used to control the *flow* of a program (*script* or *function*). In a simple program as shown in the previous Chapter, the commands are executed one after the other. Here we introduce the flow control structure that make possible to skip commands or to execute specific group of commands. We are going to describe the basic programming constructions.

6.2 Flow control

MATLAB supports the basic flow control structures found in most high level programming languages. The syntax is a hybrid of C and FORTRAN.

6.2.1 The `if...end` structure

MATLAB supports the variants of "`if`" construct.

- `if ... end`
- `if ... else ... end`
- `if ... elseif ... else ... end`

Here are some examples based on the familiar quadratic formula.

1.
```
discr = b*b - 4*a*c;
if discr < 0
   disp('Warning: discriminant is negative, roots are
   imaginary');
end
```

2.
```
discr = b*b - 4*a*c;
if discr < 0
   disp('Warning: discriminant is negative, roots are
   imaginary');
else
   disp('Roots are real, but may be repeated')
end
```

3.
```
discr = b*b - 4*a*c;
if discr < 0
   disp('Warning: discriminant is negative, roots are
   imaginary');
elseif discr == 0
   disp('Discriminant is zero, roots are repeated')
else
   disp('Roots are real)
end
```

It should also be noted that:

- `elseif` has no space between `else` and `if` (one word)
- the `end` statement is required
- no semicolon (`;`) is needed at the end of lines containing `if`, `else`, `end`
- indentation of `if` block is not required, but facilitate the reading.

6.2.2 Relational operators

A relational operator compares two numbers by determining whether a comparison is *true* or *false*. Relational operators are shown in Table below. Note that the "equal to" relational operator consists of two == signs (with no space between them), since one = is reserved for the *assignment* operator.

Operator	**Description**
$>$	Greater than
$<$	Less than
$>=$	Greater than or equal to
$<=$	Less than or equal to
$==$	Equal to
$\sim=$	Not equal to

Operator precedence

We can build expressions that use any combination of *arithmetic*, *relational*, and *logical operators*. Precedence rules determine the order in which MATLAB evaluates an expression. We have already seen this in the "Tutorial Lessons". Her we add other operators in the list. The precedence rules for MATLAB are shown in this list, ordered from highest to lowest precedence level. Operators are evaluated from left to right.

1. Parentheses ()
2. Transpose (.′), power (.ˆ), matrix power (ˆ)
3. Unary plus (+), unary minus (−), logical negation (∼)
4. Multiplication (. ∗), right division (. /), left division (.\), matrix multiplication (∗), matrix right division (/), matrix left division (\)
5. Addition (+), subtraction (−)
6. Colon operator (:)
7. Less than (<), less than or equal to (≤), greater (>), greater than or equal to (≥), equal to (==), not equal to (∼=)
8. Element-wise AND, (&)
9. Element-wise OR, (|)
10. Short-circuit AND (&&)
11. Short-circuit OR, (| |)

6.2.3 The `for...end` loop

In "`for ... end`" loop, the execution of a command is repeated at a fixed and predetermined number of times. A matching `end` delineates the statements. A simple example of `for` loop is:

```
for ii=1:5
    x=ii*ii
end
```

It is a good idea to indent the loops for readability, especially when they are nested.

```
for i=1,m
    for j=1,n
        A(i,j)=1/(i+j);
    end
end
```

6.2.4 The `while...end` loop

This loop is used when the number of *passes* is not specified. The looping continues until a stated condition is satisfied. An example of a simple "`while ... end`" is shown in the following program.

```
x = 1
while x <= 10
      x = 3*x
end
```

It is important to note that if the condition inside the looping is not well defined, the looping will continue *indefinitely*. If this happens, we can stop the execution by pressing **Ctrl-C**.

6.2.5 Other flow structures

Other control statements include `return`, `continue`, `switch`, and `break`. For more detail about these commands, consul MATLAB documentation. Table xx briefly summarize the main points on how to use them.

6.3 Saving output to a file

In addition to displaying output on the screen, the `fprintf` command can be used for writing the output to a file.The saved data can subsequently be used by MATLAB or other softwares.

To save the results of some computation to a file in text format requires the following steps:

1. Open a file using `fopen`
2. Write the output using `fprintf` command
3. Close the file using `fclose`

Here is an example (script) of its use.

```
% write some variable length strings to a file
op = fopen('weekdays.txt','wt');
fprintf(op,'Sunday\nMonday\nTuesday\nWednesday\n');
fprintf(op,'Thursday\nFriday\nSaturday\n');
fclose(op);
```

This file (`weekdays.txt`) can be opened with any program that can read `.txt` file.

Here is a second example.

```
fid = fopen('alphabet.txt', 'r');
c = fread(fid, 5, 'uint8=>char')'
fclose(fid);

c   =
    ABCDE
```

Exercises

1. Use MATLAB in two different ways to plot the function:

$$f(x) = \begin{cases} 4e^{x+2} & \text{for } -6 \le x \le -2 \\ x^2 & \text{for } -2 \le x \le 2 \\ (x+62)^{1/3} & \text{for } 2 \le x \le 6 \end{cases}$$

   (a) Create a user-defined function for $f(x)$

   (b) Write a program in a script file, use the above function, and make the plot.

2. The function e^x can be represented in Taylor's series by

$$\begin{aligned} e^x &= 1 + x + \frac{x^2}{2!} + \cdots \qquad (6.1) \\ &= \sum_{n=0}^{\infty} \frac{x^n}{n!} \end{aligned}$$

(a) Write a program in a script file that determines e^x by using the Taylor series expansion. The program calculates e^x by adding terms of the series and stopping when the absolute value of the term that was added last is smaller than 10^{-6}. The limit number of passes is 30. If in the 30^{th} pass, the value of the term that is added is not smaller than 10^{-6}, the program stops and displays a message that more than 30 terms are needed. Use the program to calculate e^3, e^{-5}, and e^{10}.

(b) Write a program in a function file, using the same procedure mentioned in (a). Then calculate e^3, e^{-5}, and e^{10}.

(c) Compare the two approaches.

3. Write a user-defined function which generates two unit conversion tables. One table converts velocity units from miles per hour to kilometers per hour, and the other table converts force units from pounds to Newtons. Each conversion table is saved to a different text file.

4. Write a script file to solve the problem in number theory. Start with any positive integer n. Repeat the following steps:

   - If $n = 1$, stop.
   - If n is even, replace it with $n/2$.
   - If n is odd, replace it with $3n + 1$.

   For example, starting with $n = 7$ produces 7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1. The sequence terminates after 17 steps. Plot the corresponding graph.

Chapter 7

Debugging M-files

7.1 Introduction

This section introduces general techniques for finding *errors* in M-files. *Debugging* is the process by which you isolate and fix *errors* in your program or code.

Debugging helps to correct two kind of errors:

- **Syntax errors** - For example omitting a parenthesis or misspelling a function name.
- **Run-time errors** - Run-time errors are usually apparent and difficult to track down. They produce unexpected results.

7.2 Debugging process

We can debug the M-files using the Editor/Debugger as well as using debugging functions from the Command Window. The debugging process consists of

- Preparing for debugging
- Setting breakpoints
- Running an M-file with breakpoints
- Stepping through an M-file
- Examining values
- Correcting problems
- Ending debugging

7.2.1 Preparing for debugging

Here we use the Editor/Debugger for debugging. Do the following to prepare for debugging:

- Open the file
- Save changes
- Be sure the file you run and any files it calls are in the directories that are on the search path.

7.2.2 Setting breakpoints

Set breakpoints *to pause* execution of the function, so we can examine where the problem might be. There are three basic types of breakpoints:

- *A standard breakpoint*, which stops at a specified line.
- *A conditional breakpoint*, which stops at a specified line and under specified conditions.
- *An error breakpoint* that stops when it produces the specified type of *warning*, *error*, *NaN*, or infinite value.

You cannot set breakpoints while MATLAB is busy, for example, running an M-file.

7.2.3 Running with breakpoints

After setting breakpoints, run the M-file from the Editor/Debugger or from the Command Window. Running the M-file results in the following:

- The prompt in the Command Window changes to

  ```
  K>>
  ```

  indicating that MATLAB is in debug mode.
- The program pauses at the *first* breakpoint. This means that line will be executed when you continue. The pause is indicated by the green arrow.
- In breakpoint, we can examine variable, step through programs, and run other calling functions.

7.2.4 Examining values

While the program is paused, we can view the value of any variable currently in the workspace. Examine values when we want to see whether a line of code has produced the expected result or not. If the result is as expected, step to the next line, and continue running. If the result is not as expected, then that line, or the previous line, contains an *error*. When we run a program, the current workspace is shown in the **Stack** field. Use `who` or `whos` to list the variables in the current workspace.

Viewing values as datatips

First, we position the cursor to the left of a variable on that line. Its current value appears. This is called a *datatip*, which is like a *tooltip* for data. If you have trouble getting the datatip to appear, click in the line and then move the cursor next to the variable.

7.2.5 Correcting and ending debugging

While debugging, we can change the value of a variable to see if the *new* value produces expected results. While the program is paused, assign a new value to the variable in the Command Window, Workspace browser, or Array Editor. Then continue running and stepping through the program.

7.2.6 Ending debugging

After identifying a problem, end the debugging session. It is best to quit *debug mode* before editing an M-file. Otherwise, you can get unexpected results when you run the file. To end debugging, select **Exit Debug Mode** from the **Debug** menu.

7.2.7 Correcting an M-file

To correct errors in an M-file,

- Quit debugging
- Do not make changes to an M-file while MATLAB is in debug mode
- Make changes to the M-file
- Save the M-file
- Clear breakpoints

- Run the M-file again to be sure it produces the expected results.

For details on debugging process, see MATLAB documentation.

Appendix A

Summary of commands

Table A.1: **Arithmetic operators and special characters**

Character	**Description**
+	Addition
−	Subtraction
∗	Multiplication (scalar and array)
/	Division (right)
ˆ	Power or exponentiation
:	Colon; creates vectors with equally spaced elements
;	Semi-colon; suppresses display; ends row in array
,	Comma; separates array subscripts
...	Continuation of lines
%	Percent; denotes a comment; specifies output format
′	Single quote; creates string; specifies matrix transpose
=	Assignment operator
( )	Parentheses; encloses elements of arrays and input arguments
[ ]	Brackets; encloses matrix elements and output arguments

Table A.2: **Array operators**

Character	**Description**
.*	Array multiplication
./	Array (right) division
.^	Array power
.\	Array (left) division
.'	Array (nonconjugated) transpose

Table A.3: **Relational and logical operators**

Character	**Description**
<	Less than
≤	Less than or equal to
>	Greater than
≥	Greater than or equal to
==	Equal to
~=	Not equal to
&	Logical or element-wise AND
|	Logical or element-wise OR
&&	Short-circuit AND
||	Short-circuit OR

Table A.4: Managing workspace and file commands

Command	**Description**
`cd`	Change current directory
`clc`	Clear the Command Window
`clear (all)`	Removes all variables from the workspace
`clear x`	Remove x from the workspace
`copyfile`	Copy file or directory
`delete`	Delete files
`dir`	Display directory listing
`exist`	Check if variables or functions are defined
`help`	Display help for MATLAB functions
`lookfor`	Search for specified word in all help entries
`mkdir`	Make new directory
`movefile`	Move file or directory
`pwd`	Identify current directory
`rmdir`	Remove directory
`type`	Display contents of file
`what`	List MATLAB files in current directory
`which`	Locate functions and files
`who`	Display variables currently in the workspace
`whos`	Display information on variables in the workspace

Table A.5: Predefined variables and math constants

Variable	**Description**
`ans`	Value of last variable (answer)
`eps`	Floating-point relative accuracy
`i`	Imaginary unit of a complex number
`Inf`	Infinity (∞)
`eps`	Floating-point relative accuracy
`j`	Imaginary unit of a complex number
`NaN`	Not a number
`pi`	The number π ($3.14159\ldots$)

Table A.6: **Elementary matrices and arrays**

Command	**Description**
`eye`	Identity matrix
`linspace`	Generate linearly space vectors
`ones`	Create array of all ones
`rand`	Uniformly distributed random numbers and arrays
`zeros`	Create array of all zeros

Table A.7: **Arrays and Matrices: Basic information**

Command	**Description**
`disp`	Display text or array
`isempty`	Determine if input is empty matrix
`isequal`	Test arrays for equality
`length`	Length of vector
`ndims`	Number of dimensions
`numel`	Number of elements
`size`	Size of matrix

Table A.8: **Arrays and Matrices: operations and manipulation**

Command	**Description**
`cross`	Vector cross product
`diag`	Diagonal matrices and diagonals of matrix
`dot`	Vector dot product
`end`	Indicate last index of array
`find`	Find indices of nonzero elements
`kron`	Kronecker tensor product
`max`	Maximum value of array
`min`	Minimum value of array
`prod`	Product of array elements
`reshape`	Reshape array
`sort`	Sort array elements
`sum`	Sum of array elements
`size`	Size of matrix

Table A.9: **Arrays and Matrices: matrix analysis and linear equations**

Command	**Description**
`cond`	Condition number with respect to inversion
`det`	Determinant
`inv`	Matrix inverse
`linsolve`	Solve linear system of equations
`lu`	LU factorization
`norm`	Matrix or vector norm
`null`	Null space
`orth`	Orthogonalization
`rank`	Matrix rank
`rref`	Reduced row echelon form
`trace`	Sum of diagonal elements

Appendix B

Release notes for Release 14 with Service Pack 2

B.1 Summary of changes

MATLAB 7 Release 14 with Service Pack 2 (R14SP2) includes several new features. The major focus of R14SP2 is on *improving* the quality of the product. The following key points may be relevant:

1. **Spaces before numbers** - For example: `A* .5`, you will typically get a mystifying message saying that A was previously used as a variable. There are two workarounds:

   (a) Remove all the spaces:

   ```
   A*.5
   ```

   (b) Or, put a zero in front of the dot:

   ```
   A * 0.5
   ```

2. **RHS empty matrix** - The right-hand side must literally be the empty matrix [ ]. It cannot be a variable that has the value [ ], as shown here:

   ```
   rhs = [];
   A(:,2) = rhs
   ??? Subscripted assignment dimension mismatch
   ```

3. **New format option** - We can display MATLAB output using two *new* formats: `short eng` and `long eng`.

- `short eng` – Displays output in *engineering* format that has at least 5 digits and a power that is a multiple of three.

```
>> format short eng
>> pi
ans   =
      3.1416e+000
```

- `long eng` – Displays output in *engineering* format that has 16 significant digits and a power that is a multiple of three.

```
>> format long eng
>> pi
ans   =
      3.14159265358979e+000
```

4. **Help** - To get help for a *subfunction*, use

```
>> help function_name>subfunction_name
```

In previous versions, the syntax was

```
>> help function_name/subfunction_name
```

This change was introduced in R14 (MATLAB 7.0) but was not documented. Use the MathWorks Web site search features to look for the latest information.

5. **Publishing** - Publishing to LaTeXnow respects the image file type you specify in preferences rather than always using EPSC2-files.

   - The Publish image options in Editor/Debugger preferences for Publishing Images have changed slightly. The changes prevent you from choosing invalid formats.
   - The files created when publishing using cells now have more natural extensions. For example, JPEG-files now have a .jpg instead of a .jpeg extension, and EPSC2-files now have an .eps instead of an .epsc2 extension.
   - Notebook will no longer support Microsoft Word 97 starting in the next release of MATLAB.

6. **Debugging** - Go directly to a subfunction or using the enhanced **Go To** dialog box. Click the **Name** column header to arrange the list of function alphabetically, or click the **Line** column header to arrange the list by the position of the functions in the file.

B.2 Other changes

1. There is a new command `mlint`, which will scan an M-file and show inefficiencies in the code. For example, it will tell you if you've defined a variable you've never used, if you've failed to pre-allocate an array, etc. These are common mistakes in EA1 which produce runnable but inefficient code.

2. You can comment-out a block of code without putting `% at the beginning of each line`. The format is

   ```
   %{

     Stuff you want MATLAB to ignore...

   %}
   ```

   The delimiters `%{` and `%}` must appear on lines by themselves, and it may not work with the comments used in functions to interact with the help system (like the H1 line).

3. There is a new function `linsolve` which will solve $Ax = b$ but with the user's choice of algorithm. This is in addition to left division $x = A\backslash b$ which uses a default algorithm.

4. The `eps` constant now takes an optional argument. `eps(x)` is the same as the old `eps*abs(x)`.

5. You can break an M-file up into named cells (blocks of code), each of which you can run separately. This may be useful for testing/debugging code.

6. Functions now optionally end with the `end` keyword. This keyword is mandatory when working with nested functions.

B.3 Further details

1. You can *dock* and *un-dock* windows from the main window by clicking on an icon. Thus you can choose to have all Figures, M-files being edited, help browser, command window, etc. All appear as panes in a single window.

2. Error messages in the command window resulting from running an M-file now include a clickable link to the offending line in the editor window containing the M-file.

3. You can customize figure interactively (labels, line styles, etc.) and then automatically generate the code which reproduces the customized figure.

4. `feval` is no longer needed when working with function handles, but still works for backward compatibility. For example, `x=@sin; x(pi)` will produce `sin(pi)` just like `feval(x,pi)` does, but faster.

5. You can use function handles to create anonymous functions.

6. There is support for nested functions, namely, functions defined within the body of another function. This is in addition to sub-functions already available in version 6.5.

7. There is more support in arithmetic operations for numeric data types other than double, e.g. `single, int8, int16, uint8, uint32,` etc.

Finally, please see our webpage for other details:

http://computing.mccormick.northwestern.edu/matlab/

Appendix C

Main characteristics of MATLAB

C.1 History

- Developed primarily by Cleve Moler in the 1970's
- Derived from FORTRAN subroutines LINPACK and EISPACK, linear and eigenvalue systems.
- Developed primarily as an interactive system to access LINPACK and EISPACK.
- Gained its popularity through word of mouth, because it was not officially distributed.
- Rewritten in C in the 1980's with more functionality, which include plotting routines.
- The MathWorks Inc. was created (1984) to market and continue development of MATLAB.

According to Cleve Moler, three other men played important roles in the origins of MATLAB: J. H. Wilkinson, George Forsythe, and John Todd. It is also interesting to mention the authors of LINPACK: Jack Dongara, Pete Steward, Jim Bunch, and Cleve Moler. Since then another package emerged: LAPACK. LAPACK stands for Linear Algebra Package. It has been designed to supersede LINPACK and EISPACK.

C.2 Strengths

- MATLAB may behave as a *calculator* or as a *programming language*
- MATLAB combine nicely calculation and graphic plotting.
- MATLAB is relatively easy to learn

- MATLAB is interpreted (not compiled), errors are easy to fix
- MATLAB is optimized to be relatively fast when performing matrix operations
- MATLAB does have some object-oriented elements

C.3 Weaknesses

- MATLAB is not a *general* purpose programming language such as C, C++, or FORTRAN
- MATLAB is designed for scientific computing, and is not well suitable for other applications
- MATLAB is an interpreted language, slower than a compiled language such as C++
- MATLAB commands are specific for MATLAB usage. Most of them do not have a direct equivalent with other programming language commands

C.4 Competition

- One of MATLAB's competitors is **Mathematica**, the *symbolic* computation program.
- MATLAB is more convenient for *numerical analysis* and *linear algebra*. It is frequently used in *engineering community*.
- Mathematica has superior symbolic manipulation, making it popular among *physicists*.
- There are other competitors:
  - **Scilab**
  - **GNU Octave**
  - **Rlab**

Bibliography

[1] The MathWorks Inc. *MATLAB 7.0 (R14SP2)*. The MathWorks Inc., 2005.

[2] S. J. Chapman. *MATLAB Programming for Engineers*. Thomson, 2004.

[3] A. Gilat. *MATLAB: An introduction with Applications*. John Wiley and Sons, 2004.

[4] C. B. Moler. *Numerical Computing with MATLAB*. Siam, 2004.

[5] C. F. Van Loan. *Introduction to Scientific Computing*. Prentice Hall, 1997.

[6] D. J. Higham and N. J. Higham. *MATLAB Guide*. Siam, second edition edition, 2005.

[7] K. R. Coombes, B. R. Hunt, R. L. Lipsman, J. E. Osborn, and G. J. Stuck. *Differential Equations with MATLAB*. John Wiley and Sons, 2000.

[8] J. Cooper. *A MATLAB Companion for Multivariable Calculus*. Academic Press, 2001.

[9] J. C. Polking and D. Arnold. *ODE using MATLAB*. Prentice Hall, 2004.

[10] D. Kahaner, C. Moler, and S. Nash. *Numerical Methods and Software*. Prentice-Hall, 1989.

[11] J. W. Demmel. *Applied Numerical Linear Algebra*. Siam, 1997.

[12] D. Houcque. Applications of MATLAB: Ordinary Differential Equations. *Internal communication, Northwestern University*, pages 1–12, 2005.