INTRODUCTION TO MATLAB FOR ENGINEERING STUDENTS

David Houcque
Northwestern University

Contents

List of Tables

List of Figures

Preface

“Introduction to MATLAB for Engineering Students” is a document for an introductory course in MATLAB®[1] and technical computing. It is used for freshmen classes 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 sessions are used with one main goal: to allow students to become familiar with computer software (e.g., MATLAB) to solve application problems. We assume that the students have no prior experience with MATLAB.

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 students 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.

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 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 Engineering First®[2] courses taught at Northwestern University.

Finally, the students - Engineering Analysis (EA) Section - deserve my special gratitude. They were very active participants in class.

David Houcque
Evanston, Illinois
August 2005

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

[2] Engineering First® 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: Rebecca Swierz, Jesse Becker, Rick Mazec, Alan Wolff, Ken Kalan, Mike Vilches, and Daniel Lee.

About the author

David Houcque has more than 25 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 fields of nuclear engineering, oil rig platform offshore design, oil reservoir engineering, and steel industry. All of these 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 emphasis here is “learning by doing”. Therefore, the best way to learn is by trying it yourself. Working through the examples 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.

The name MATLAB stands for MATrix LABoratory. MATLAB was written originally to provide easy access to matrix software developed by the LINPACK (linear system package) and EISPACK (Eigen system package) projects.

MATLAB [1] is a high-performance language for technical computing. It integrates *computation*, *visualization*, and *programming* environment. Furthermore, MATLAB is a modern programming language 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 which is mostly available on-line, we would recommend the following books: [2], [3], [4], [5], [6], [7], [8], and [9]. They are excellent in their specific applications.

1.2 Basic features

As we mentioned earlier, 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 (also called *starting* and *exiting* sessions) 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 7.0.4) 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
- The Workspace
- The Current Directory
- The Help Browser
- The Start button

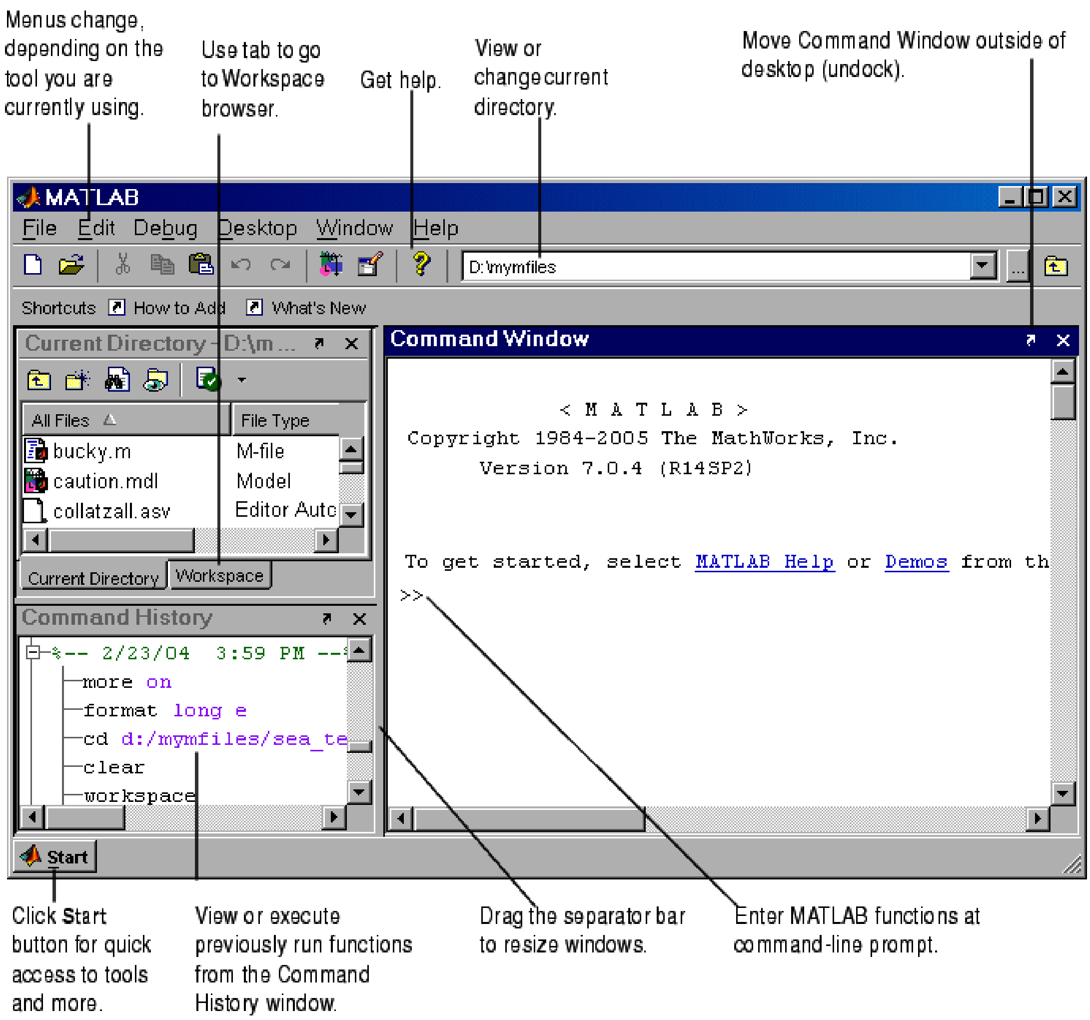

Figure 1.1: The graphical interface to the MATLAB workspace

When MATLAB is started for the first time, the screen looks like the one that shown in the Figure 1.1. This 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 the MATLAB desktop on your computer, which contains the prompt (`>>`) in the Command Window. Usually, there are 2 types of prompt:

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

NOTE: To simplify the notation, we will use this prompt, `>>`, as a standard 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 existed). To avoid this, you may 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 variable 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
```

Before we conclude this minimum session, Table 1.1 gives the partial list of arithmetic operators.

Table 1.1: Basic arithmetic operators

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** ⟶ **Exit MATLAB** in the desktop main menu.

1.4 Getting started

After learning the minimum MATLAB session, we will now learn to use some additional operations.

1.4.1 Creating MATLAB variables

MATLAB variables are created with an assignment 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. In addition, if you do not wish to see the intermediate results, you can suppress the numerical output by putting a semicolon (`;`) at the end of the line. Then the sequence of commands looks like this:

```
>> 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 multiplication sign, `*`, 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
```

and, from previous example

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

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

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 in some complex expressions 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 1.2.

Table 1.2: Hierarchy of arithmetic operations

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

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, consider another example:

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

In MATLAB, it becomes

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

or, if parentheses are missing,

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

So here what we get: two different results. Therefore, we want to emphasize the importance of precedence rule in order to avoid ambiguity.

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`.

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
```

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

It is possible to keep track of everything done during a MATLAB session with the `diary` command.

```
>> diary
```

or give a name to a created file,

```
>> diary FileName
```

where `FileName` could be any arbitrary name you choose.

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 directly 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 - At this particular time of our study, it is important to emphasize one main point. 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 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

NOTE: Due to the teaching class during this Fall 2005, the *problems* are *temporarily* removed from this section.

Chapter 2

Tutorial lessons 2

2.1 Mathematical functions

MATLAB offers many predefined mathematical functions for technical computing which 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*. 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`, and `log` respectively in MATLAB.

Table 2.1 lists some commonly used functions, where variables `x` and `y` can be numbers, vectors, or matrices.

Table 2.1: Elementary functions

`cos(x)`	Cosine	`abs(x)`	Absolute value
`sin(x)`	Sine	`sign(x)`	Signum function
`tan(x)`	Tangent	`max(x)`	Maximum value
`acos(x)`	Arc cosine	`min(x)`	Minimum value
`asin(x)`	Arc sine	`ceil(x)`	Round towards $+\infty$
`atan(x)`	Arc tangent	`floor(x)`	Round towards $-\infty$
`exp(x)`	Exponential	`round(x)`	Round to nearest integer
`sqrt(x)`	Square root	`rem(x)`	Remainder after division
`log(x)`	Natural logarithm	`angle(x)`	Phase angle
`log10(x)`	Common logarithm	`conj(x)`	Complex conjugate

In addition to the elementary functions, MATLAB includes a number of predefined constant values. A list of the most common values is given in Table 2.2.

Table 2.2: Predefined constant values

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

2.1.1 Examples

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

As a first example, the value of the expression $y = e^{-a}\sin(x) + 10\sqrt{y}$, for $a = 5$, $x = 2$, and $y = 8$ is computed by

```
>> a = 5; x = 2; y = 8;
>> y = exp(-a)*sin(x)+10*sqrt(y)
y  =
   28.2904
```

The subsequent examples are

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

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

Note the difference between the natural logarithm `log(x)` and the decimal logarithm (base 10) `log10(x)`.

To calculate $\sin(\pi/4)$ and e^{10}, 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 to 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

2.2.1 overview

MATLAB has an excellent set of graphic tools. Plotting a given data set or the results of computation is possible with very few commands. You are highly encouraged to plot mathematical functions and results of analysis as often as possible. Trying to understand mathematical equations with graphics is an enjoyable and very efficient way of learning mathematics. Being able to plot mathematical functions and data freely is the most important step, and this section is written to assist you to do just that.

2.2.2 Creating simple plots

The basic MATLAB graphing procedure, for example 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_i, y_i), with $i = 1, 2, \ldots, n$ and then join them by straight lines. You need to prepare x and y in an identical array form; namely, x and y are both row arrays or column arrays of the *same* length.

The MATLAB command to plot a graph is `plot(x,y)`. 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)
```

NOTE: 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:

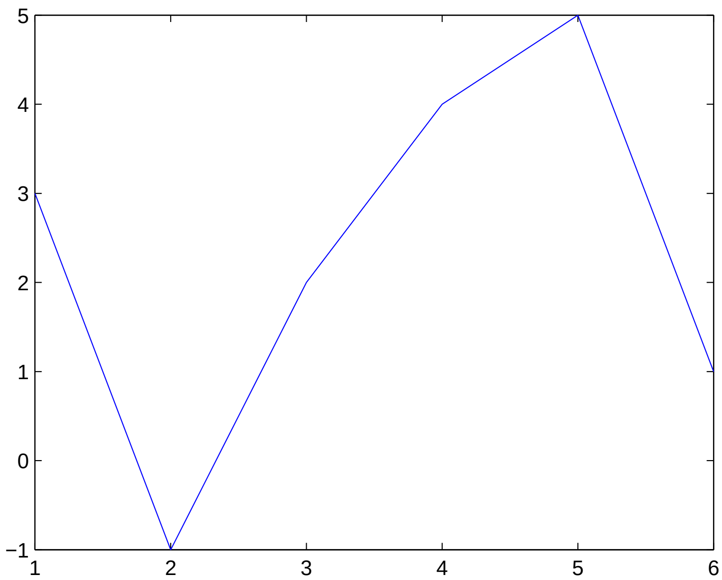

Figure 2.1: Plot for the vectors x and y

```
>> 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.

2.2.3 Adding titles, axis labels, and annotations

MATLAB enables you to add axis labels and titles. For example, using the graph from the previous example, add an x- and y-axis labels.

Now label the axes and add a title. The character `\pi` creates the symbol π. An example of 2D plot is shown in Figure 2.2.

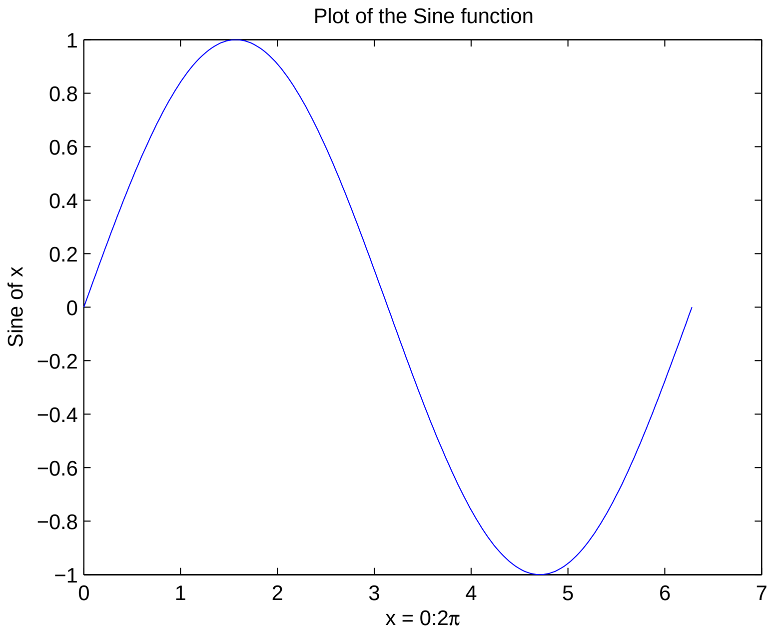

Figure 2.2: Plot of the Sine function

```
>> 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`.

2.2.4 Multiple data sets in one plot

Multiple (x, y) *pairs* arguments create *multiple* graphs with a single call to *plot*. For example, these statements plot three related functions of x: $y_1 = 2\cos(x)$, $y_2 = \cos(x)$, and $y_3 = 0.5 * \cos(x)$, in the interval $0 \leq x \leq 2\pi$.

```
>> x = 0:pi/100:2*pi;
>> y1 = 2*cos(x);
>> y2 = cos(x);
>> y3 = 0.5*cos(x);
>> plot(x,y1,'--',x,y2,'-',x,y3,':')
>> xlabel('0 \leq x \leq 2\pi')
>> ylabel('Cosine functions')
>> legend('2*cos(x)','cos(x)','0.5*cos(x)')
```

```
>> title('Typical example of multiple plots')
>> axis([0 2*pi -3 3])
```

The result of multiple data sets in one graph plot is shown in Figure 2.3.

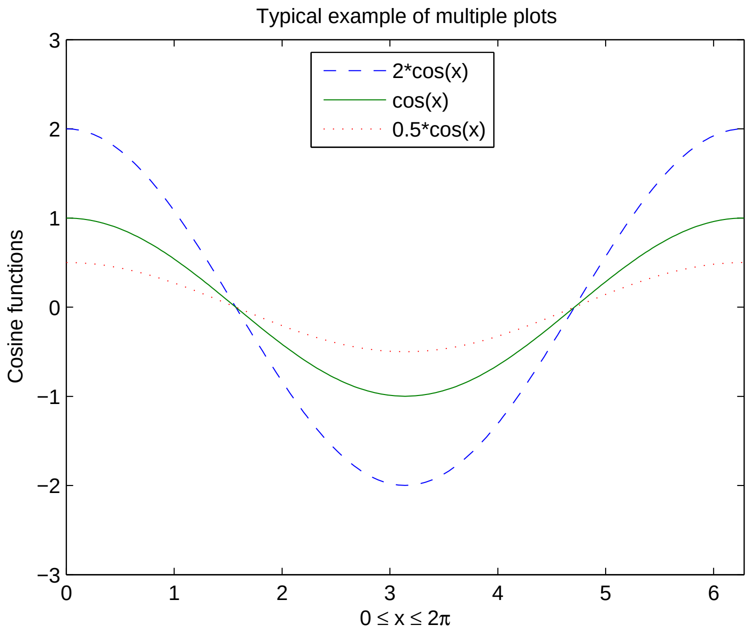

Figure 2.3: Typical example of multiple plots

By default, MATLAB uses *line style* and *color* to distinguish the data sets plotted in the graph. However, you can change the appearance of these graphic components or add annotations to the graph to help explain your data for presentation.

2.2.5 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')
```

where `style_color_marker` is a *triplet* of values from Table 2.3.

To find additional information, type `help plot` or `doc plot`.

Table 2.3: Attributes for `plot`

SYMBOL	COLOR	SYMBOL	LINE STYLE	SYMBOL	MARKER
`k`	Black	$-$	Solid	$+$	Plus sign
`r`	Red	$--$	Dashed	o	Circle
`b`	Blue	:	Dotted	$*$	Asterisk
`g`	Green	$-.$	Dash-dot	.	Point
`c`	Cyan	none	No line	$\times$	Cross
`m`	Magenta			s	Square
`y`	Yellow			d	Diamond

2.3 Exercises

NOTE: Due to the teaching class during this Fall Quarter 2005, the *problems* are *temporarily* removed from this section.

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*. 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, we need to become familiar with matrix generation and manipulation. Matrices can be generated in several ways.

3.2.1 Entering a vector

A vector is a special case of a matrix. The purpose of this section is to show how to create vectors and matrices in MATLAB. As discussed earlier, an array of dimension $1 \times n$ is called a *row* vector. The elements of vectors in MATLAB are enclosed by square brackets and are separated by spaces or by commas. For example,

```
>> v = [1 4 7 10 13]
v  =
     1     4     7     10     13
```

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

```
>> w = [1;4;7;10;13]
w  =
     1
     4
     7
     10
     13
```

On the other hand, a *row* vector is converted to a *column* vector using the *transpose* operator. The *transpose* operation is denoted by an apostrophe or a single quote (`'`).

```
>> w = v'
w  =
     1
     4
     7
     10
     13
```

Thus, `v(1)` is the first element of vector `v`, `v(2)` its second element, and so forth.

Furthermore, to access *blocks* of elements, we use MATLAB's colon notation. For example, to access the first three elements of `v`, we write,

```
>> v(1:3)
ans =
    1    4    7
```

Or, all elements from the third through the last elements,

```
>> v(3,end)
ans =
    7    10    13
```

where `end` signifies the *last* element in the vector. If `v` is a vector, writing

```
>> v(:)
```

produces a column vector, whereas writing

```
>> v(1:end)
```

produces a row vector.

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 3×3 matrix,

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

Note that the use of semicolons (`;`) here is different from their use mentioned earlier to suppress output or to write multiple commands in a single line.

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. We write

```
>> 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 Matrix indexing

We select elements in a matrix just as we did for vectors, but now we need two indices. The element of row i and column j of the matrix $\mathbf{A}$ is denoted by `A(i,j)`. Thus, `A(i,j)` in MATLAB refers to the element A_{ij} of matrix $\mathbf{A}$. 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

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.

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;
```

The 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. Subscript expressions refer to portions of a matrix. For example,

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

is the second row elements of $\mathbf{A}$.

The colon operator can also be used to extract a sub-matrix from $\mathbf{A}$.

```
>> 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
```

3.2.5 Linear spacing

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.

3.2.6 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
```

3.2.7 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.

3.2.8 Dimension

To determine the *dimensions* of a matrix or vector, use the command `size`. For example,

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

means 3 rows and 3 columns.

Or more explicitly with,

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

3.2.9 Continuation

If it is not possible to type the entire input on the same line, use consecutive periods, called an ellipsis ..., 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.

3.2.10 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 $\mathbf{A}$ is the $n \times m$ matrix that results from interchanging the rows and columns of $\mathbf{A}$. The transpose matrix is denoted $\mathbf{A}^T$.

3.2.11 Concatenating matrices

Matrices can be made up of sub-matrices. Here is an 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
```

3.2.12 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.

Table 3.1: Elementary matrices

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

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

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

3.2.13 Special matrices

MATLAB provides a number of special matrices (see Table 3.2). These matrices have interesting properties that make them useful for constructing examples and for testing algorithms. For more information, see MATLAB documentation.

Table 3.2: Special matrices

`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

3.3 Exercises

NOTE: Due to the teaching class during this Fall Quarter 2005, the *problems* are *temporarily* removed from this section.

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 $\mathbf{A}$ and $\mathbf{B}$ 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 $\mathbf{C}$ 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 $\mathbf{A}$, 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 $\mathbf{U}$ and $\mathbf{V}$ 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 $\mathbf{x}$ using linear algebra. The result is $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$.

There are typically two ways to solve for $\mathbf{x}$ 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 **A**.

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

Calculating the inverse of **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 **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

NOTE: Due to the teaching class during this Fall Quarter 2005, the *problems* are *temporarily* removed from this section.

Chapter 5

Introduction to M-File programming

5.1 Introduction

So far in this manual, all the commands were executed in the Command Window. The problem is that the commands entered in the Command Window cannot be saved and executed again for several times. 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 or *scripts* for short.

This section covers the following topics:

- M-File Scripts
- M-File Functions

5.2 M-File Scripts

A *script file* is an external file that contains a sequence of MATLAB statements. Script files have a filename extension `.m` and are often called M-files. M-files can be *scripts* that simply execute a series of MATLAB statements, or they can be *functions* that can accept arguments and can produce one or more outputs.

5.2.1 Examples

Here are two simple scripts.

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 $\mathbf{x}$ to the system of equations.

SOLUTION:

- Use the MATLAB *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.

NOTE: The MATLAB editor is both a text editor specialized for creating M-files and a graphical MATLAB debugger. The MATLAB editor has numerous menus for tasks such as *saving*, *viewing*, and *debugging*. Because it performs some simple checks and also uses color to differentiate between various elements of codes, this text editor is recommended as the tool of choice for writing and editing M-files. There is another way to open the editor:

```
>> edit
```

or

```
>> edit filename.m
```

to open `filename.m`.

Example 2

Plot the following cosine functions, $y_1 = 2\cos(x)$, $y_2 = \cos(x)$, and $y_3 = 0.5 * \cos(x)$, in the interval $0 \leq x \leq 2\pi$. This example has been presented in previous Chapter. Here we put the commands in a file.

- Create a file, `example2.m`, which contains the following commands:

```
x = 0:pi/100:2*pi;
y1 = 2*cos(x);
y2 = cos(x);
y3 = 0.5*cos(x);
plot(x,y1,'--',x,y2,'-',x,y3,':')
xlabel('0 \leq x \leq 2\pi')
ylabel('Cosine functions')
legend('2*cos(x)','cos(x)','0.5*cos(x)')
title('Typical example of multiple plots')
axis([0 2*pi -3 3])
```

- Run the file by typing `example2` in the Command Window.

5.2.2 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 overwritten.
- The execution of the script can be affected by the state variables in the workspace.

As a result, because scripts have some undesirable side effects, it is better to code any complicated applications using rather function M-file.

5.3 M-File functions

As mentioned earlier, functions are programs (or *routines*) that accept *input* arguments and return *output* arguments. 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 Anatomy of a M-File function

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

```
function f = factorial(n)                    (1)
% FACTORIAL(N) returns the factorial of N.   (2)
% Compute a factorial value.                 (3)

f = prod(1:n);                               (4)
```

The first line of a function M-file starts with the keyword `function`. It gives the function *name* and order of *arguments*. In the case of function `factorial`, there are up to one output argument and one input argument. Table 5.1 summarizes the anatomy of a M-file function.

As an example, for $n = 5$, the result is,

```
>> f = factorial(5)
f =
  120
```

Table 5.1: Anatomy of a M-File function

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

Both *functions* and *scripts* can have all of these parts, except for the *function definition 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`.

Table 5.2 summarizes the differences between *scripts* and *functions*.

Table 5.2: Main differences between scripts and functions

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

5.3.2 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. The general form looks like this

```
function [outputs] = function_name(inputs)
```

Function file can have none, one, or several output arguments. Table 5.3 illustrates some possible combinations of input and output arguments.

Table 5.3: Example 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

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 `example3`) is executed.

```
>> example3
>> 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.

A typical example of M-file function programming can be found in a recent paper which related to the solution of the ordinary differential equation (ODE) [12].

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 (Table 5.4).

Table 5.4: `disp` and `fprintf` commands

`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 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.
% 'fprintf' will be used 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)
```

```
>> example3
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
```

The following table (Table 5.5) shows the basic conversion codes.

Table 5.5: Conversion codes for output command `fprintf`

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

NOTE: Due to the teaching class during this Fall Quarter 2005, the *problems* are *temporarily* removed from this section.

Chapter 6

Control flow and operators

6.1 Introduction

MATLAB is also a *programming language*. Like other computer programming languages, MATLAB has some decision making structures for control of command execution. These decision making or *control flow* structures include `for` loops, `while` loops, and `if-else-end` constructions. Control flow structures are often used in script M-files and function M-files.

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.

6.2 Control flow

MATLAB has four control flow structures: the `if` statement, the `for` loop, the `while` loop, and the `switch` statement.

6.2.1 The ‘‘if...end’’ structure

MATLAB supports the variants of “`if`” construct.

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

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

The simplest form of the `if` statement is

```
if expression
   statements
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 be noted that:

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

6.2.2 Relational and logical operators

A relational operator compares two numbers by determining whether a comparison is *true* or *false*. Relational operators are shown in Table 6.1.

Table 6.1: Relational and logical operators

OPERATOR	DESCRIPTION
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
~=	Not equal to
&	AND operator
|	OR operator
~	NOT operator

Note that the "equal to" relational operator consists of two equal signs (==) (with no space between them), since = is reserved for the *assignment* operator.

6.2.3 The ``for...end'' loop

In the `for ... end` loop, the execution of a command is repeated at a fixed and predetermined number of times. The syntax is

```
for variable = expression
    statements
end
```

Usually, `expression` is a vector of the form `i:s:j`. 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. Note that MATLAB editor does it automatically.

Multiple `for` loops can be nested, in which case *indentation* helps to improve the readability. The following statements form the 5-by-5 symmetric matrix `A` with (i, j) element i/j for $j \geq i$:

```
n = 5; A = eye(n);
for j=2:n
    for i=1:j-1
        A(i,j)=i/j;
        A(j,i)=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. The `while` loop has the form:

```
while expression
      statements
end
```

The `statements` are executed as long as `expression` is true.

```
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

- The `break` statement. A `while` loop can be terminated with the `break` statement, which passes control to the first statement after the corresponding `end`. The `break` statement can also be used to exit a `for` loop.

- The `continue` statement can also be used to exit a `for` loop to pass immediately to the next iteration of the loop, skipping the remaining statements in the loop.

- Other control statements include `return`, `continue`, `switch`, etc. For more detail about these commands, consul MATLAB documentation.

6.2.6 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".

Here we add other operators in the list. The precedence rules for MATLAB are shown in this list (Table 6.2), ordered from *highest* (1) to *lowest* (9) precedence level. Operators are evaluated from left to right.

Table 6.2: Operator precedence

PRECEDENCE	OPERATOR
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, (|)

6.3 Saving output to a file

In addition to displaying output on the screen, the command `fprintf` 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 a text format requires the following steps:

1. Open a file using `fopen`
2. Write the output using `fprintf`
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.

6.4 Exercises

Note: Due to the teaching class during this Fall Quarter 2005, the *problems* are *temporarily* removed from this section.

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)
^	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 . . .)

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. This document doesn't attempt to provide a complete specification of every single feature, but instead provides a brief introduction to each of them. For full details, you should refer to the MATLAB documentation (Release Notes).

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

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

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

[6] 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.

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

[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.