POLYTECHNIC UNIVERSITY
Department of Computer and Information Science

A Brief MATLAB Tutorial

K. Ming Leung

Abstract: We present a brief MATLAB tutorial covering only the bare-minimum that a beginner needs to know in order to start writing programs in MATLAB.

Directory

Last Revision Date: January 25, 2005

Table of Contents

1. Introduction: What is MATLAB?

MATLAB

1. integrates computation, visualization, and programming in an easy-to-use environment for the purpose of technical computing.
2. is an interactive system whose basic data element is an array that does not require dimensioning.
3. incorporates many heavy-duty mathematical libraries such as the LAPACK and BLAS for matrix computation.
4. features a family of add-on application-specific solutions called toolboxes which consist of comprehensive collections of MATLAB functions (M-files) that extend the MATLAB environment.

2. The MATLAB System

The MATLAB system consists of the following five main parts.

1. *The Development Environment* is the set of tools and facilities that help you use MATLAB functions and files. Many of these

tools are graphical user interfaces. It includes the MATLAB desktop and Command Window, a command history, an editor and debugger, and browsers for viewing help, the workspace, files, and the search path.

2. *The MATLAB Programming Language* is a high-level matrix language with control flow statements, functions, data structures, input/output, and object-oriented programming features. It allows one to rapidly create quick and dirty throw-away programs, as well as complete large and complex application programs.

3. *The MATLAB Mathematical Function Library* is a vast collection of functions ranging from elementary ones like sum, sine, cosine, and complex arithmetic, to more sophisticated ones like matrix inverse, matrix eigenvalues, Bessel functions, and fast Fourier transforms. We will illustrate how to use the mathematical function when we deal with the MATLAB language.

4. *MATLAB Graphics* provide extensive facilities for displaying vectors and matrices as graphs, as well as annotating and print-

ing these graphs. It includes high-level functions for two- and three-dimensional data visualization, image processing, animation, and presentation graphics. It also includes low-level functions that allow you to fully customize the appearance of graphics as well as to build your own graphical user interfaces.

5. *The MATLAB Application Program Interface (API)* is a library that allows you to write C and Fortran programs that interact with MATLAB. It includes facilities for calling routines from MATLAB (dynamic linking), calling MATLAB as a computational engine, and for reading and writing MAT-files. We will not cover the API here.

2.1. The Development Environment

You work with MATLAB through its development environment.

- **Starting/quitting MATLAB**

To start MATLAB, double-click the MATLAB shortcut icon on your Windows desktop.

To end your MATLAB session, select Exit MATLAB from the File menu in the desktop, or close the main MATLAB window or type quit or exit in the Command Window.

- **MATLAB desktop**

When you start MATLAB, the MATLAB desktop appears, containing tools (graphical user interfaces) for managing files, variables, and applications associated with MATLAB.

You can change the way your desktop looks by opening, closing, moving, and resizing the tools in it. Use the View menu to open or close the tools.

You can specify certain characteristics for the desktop tools by selecting Preferences from the File menu.

- **MATLAB desktop tools**

MATLAB has many desktop tools. The important ones are

- Command Window
- Editor

- Help Browser
- Current Directory Browser
- Workspace Browser

The most important tool is the Command Window. The command line is marked by the `>>` symbol. This is the place where the user enters variables and execute script files called M-files. User can enter statements on the command line. Each statement must be separated from each other by a comma, a semi-colon, or the new-line character. Output to the screen is suppressed if a statement is terminated with a semi-colon, otherwise results of a computation are displayed in the command window.

For example if the user types on the command line

```
>> rad = 2.3; area = pi*rad^2, vol = 4*pi*rad^3/3
```

and press `enter`, the following screen output will be displayed:

```
area =
        16.6190
vol =
```

```
        50.9650
>>
```

All variables retain their values unless these values are cleared. Value of a variable can be displayed on the screen using the `disp` function:

```
disp(rad);
```

produces the display

```
    2.3000
```

Note that the value is displayed without the name of the variable.

All computations in MATLAB are done in double precision. One can use the built-in function `format` to switch between different output display formats. The default format is `format short`, which uses the scaled fixed point format with 5 digits.

The value stored in `rad` is cleared by

```
clear rad;
```

All variables in memory are cleared using

```
clear
```

without an argument.

Any input longer than a few lines should be placed in a file which you can create using the MATLAB text editor (or any other text editor). The file must end with the .m extension. Such a file is call a script file. To run the script file, enter the name of the file (without the .m extension) on the command line (assuming that the file is in the path). One can use the path editor to change the path. One can also navigate to any directory of interest, thereby changing the current directory.

Any file you want to run in MATLAB must reside in the current directory or in a directory that is on the search path. A quick way to view or change the current directory is by using the Current Directory field in the desktop toolbar.

2.2. The MATLAB Programming Language

Although MATLAB has other ways of storing both numeric and non-numeric data, but in the beginning, it is best to think of everything as a matrix. Special meaning is sometimes attached to 1-by-1 matrices, which are scalars, and to matrices with only one row or column,

which are vectors. MATLAB allows you to work with entire matrices quickly and easily.

Other than one- and two-dimensional arrays MATLAB supports other data structures, such as multidimensional arrays, cell arrays, character and text data, structures and classes and objects.

It also has the `if`, `elseif`, and `switch` for controlling program flow and the `for` and `while` (but no `do-while`) for creating loops. It has practically every thing that any other modern programming language has. You will need to use the online help system or use the manual to look up the usages of various MATLAB built-in functions since the syntax are generally different than in other languages.

- **Entering Matrices**

The best way for you to get started with MATLAB is to learn how to handle matrices. Start MATLAB and follow along with each example. You can enter matrices into MATLAB in several different ways:

Enter an explicit list of elements. Load matrices from external data files. Generate matrices using built-in functions. Create matrices with your own functions in M-files.

Start by entering Dürer's matrix as a list of its elements. You only have to follow a few basic conventions: Separate the elements of a row with blanks or commas. Use a semicolon to indicate the end of each row. Surround the entire list of elements with square brackets. To enter Dürer's matrix, simply type on the command line

```
A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]
```

MATLAB displays the matrix you just entered.

```
A =
    16     3     2    13
     5    10    11     8
     9     6     7    12
     4    15    14     1
```

Once you have entered the matrix, it is automatically remembered in the MATLAB workspace. You can refer to it simply as `A`.

- **sum, transpose, and diag**

You are probably already aware that the special properties of a magic square have to do with the various ways of summing its elements. If

you take the sum along any row or column, or along either of the two main diagonals, you will always get the same number. Let us verify that using MATLAB. The first statement to try is `sum(A)` MATLAB replies with

```
ans =
    34    34    34    34
```

When you do not specify an output variable, MATLAB uses the variable `ans`, short for answer, to store the results of a calculation. You have computed a row vector containing the sums of the columns of `A`. Sure enough, each of the columns has the same sum, the magic sum, `34`. How about the row sums? MATLAB has a preference for working with the columns of a matrix, so the easiest way to get the row sums is to transpose the matrix, compute the column sums of the transpose, and then transpose the result. The transpose operation is denoted by an apostrophe or single quote, '. It flips a matrix about its main diagonal and it turns a row vector into a column vector. So `A'` produces

```
ans =
```

```
    16     5     9     4
     3    10     6    15
     2    11     7    14
    13     8    12     1
```

And

```
sum(A')'
```

produces a column vector containing the row sums

```
ans =
    34
    34
    34
    34
```

The sum of the elements on the main diagonal is obtained with the sum and the diag functions. `diag(A)` produces

```
ans =
    16
    10
     7
```

```
     1
```

and

```
sum(diag(A))
```

produces

```
ans =
    34
```

The other diagonal, the so-called antidiagonal, is not so important mathematically, so MATLAB does not have a ready-made function for it. But a function originally intended for use in graphics, `fliplr`, flips a matrix from left to right.

```
sum(diag(fliplr(A)))

ans =
    34
```

You have verified that matrix `A` is indeed a magic square.

- **Subscripts**

The element in row i and column j of `A` is denoted by `A(i,j)`. For example, `A(4,2)` is the number in the fourth row and second column. For our magic square, `A(4,2)` is `15`. So to compute the sum of the elements in the fourth column of `A`, type

```
A(1,4) + A(2,4) + A(3,4) + A(4,4)
```

This produces

```
ans =
    34
```

but is not the most elegant nor efficient way of summing a single column.

If you try to use the value of an element outside of the matrix, it is an error.

```
t = A(4,5)
```

Index exceeds matrix dimensions.

On the other hand, if you store a value in an element outside of the matrix, the size increases to accommodate the newcomer. For example

```
X = A;
X(4,5) = 17
```

gives

```
X =
    16     3     2    13     0
     5    10    11     8     0
     9     6     7    12     0
     4    15    14     1    17
```

Notice that all elements which have not been explicitly assigned a value, take on the default value of zero.

- **The Colon Operator**

The colon, :, is one of the most important MATLAB operators. It occurs in several different forms. The expression `2:9` is a row vector containing the integers from `2` to `9`

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

To obtain non-unit spacing, specify an increment. For example,

```
100:-7:50
```

is

```
    100     93     86     79     72     65     58     51
```

and

```
    0:pi/4:pi
```

is

```
    0    0.7854    1.5708    2.3562    3.1416
```

One can use subscript expressions involving colons to refer to portions of a given matrix. `A(1:k,j)` is the first k elements of the jth column of A. So `sum(A(1:4,4))` computes the sum of the fourth column. But there is a better way. The colon by itself refers to all the elements in a row or column of a matrix and the keyword `end` refers to the last row or column. So `sum(A(:,end))` computes the sum of the elements in the last column of `A`.

```
    ans =
        34
```

Why is the magic sum for a 4-by-4 square equal to `34`? If the integers from `1` to `16` are sorted into four groups with equal sums, that sum must be `sum(1:16)/4` which, of course, is

```
ans =
    34
```

- **The magic Function**

MATLAB actually has a built-in function that creates magic squares of almost any size. Not surprisingly, this function is named magic.

```
B = magic(4)
B =
   16     2     3    13
    5    11    10     8
    9     7     6    12
    4    14    15     1
```

This matrix is almost the same as the one in the Dürer engraving and has all the same "magic" properties; the only difference is that the two middle columns are exchanged. To make this `B` into Dürer's `A`, swap the two middle columns.

```
A = B(:,[1 3 2 4])
```

This says, for each of the rows of matrix `B`, reorder the elements in the order 1, 3, 2, 4.

It produces

```
A =
    16     3     2    13
     5    10    11     8
     9     6     7    12
     4    15    14     1
```

Why would Dürer go to the trouble of rearranging the columns when he could have used MATLAB ordering? No doubt he wanted to include the date of the engraving, 1514, at the bottom of his magic square.

- **Expressions**

Like most other programming languages, MATLAB provides mathematical expressions, but unlike most programming languages, these expressions involve entire matrices. The building blocks of expressions are variables,numbers, operators and functions. MATLAB does not

require any type declarations or dimension statements. When MATLAB encounters a new variable name, it automatically creates the variable and allocates the appropriate amount of storage. If the variable already exists, MATLAB changes its contents and, if necessary, allocates new storage.

For example,

```
numStudents = 25
```

creates a 1-by-1 matrix named numStudents and stores the value 25 in its single element. Variable names consist of a letter, followed by any number of letters, digits, or underscores. MATLAB uses only the first 31 characters of a variable name. MATLAB is case sensitive; it distinguishes between uppercase and lowercase letters. A and a are not the same variable. To view the values assigned to any variable, simply enter the variable name.

- **Numbers**

MATLAB uses conventional decimal notation, with an optional decimal point and leading plus or minus sign, for numbers. Scientific

notation uses the letter e to specify a power-of-ten scale factor. Imaginary numbers use either i or j as a suffix. Some examples of legal numbers are

```
3              -99            0.0001
9.6397238      1.60210e-20    6.02252e23
1i             -3.14159j      3e5i
```

All numbers are stored internally using the long format specified by the IEEE floating-point standard. Floating-point numbers have a finite precision of roughly 16 significant decimal digits and a finite range of roughly 10^{-308} to 10^{+308}.

- **Functions**

MATLAB provides a large number of standard elementary mathematical functions, including abs, sqrt, exp, and sin. Taking the square root or logarithm of a negative number is not an error; the appropriate complex result is produced automatically. MATLAB also provides many more advanced mathematical functions, including Bessel and gamma functions. Most of these functions accept complex arguments. For a list of the elementary mathematical functions, type help

elfun For a list of more advanced mathematical and matrix functions, type help specfun help elmat Some of the functions, like sqrt and sin, are built in. They are part of the MATLAB core so they are very efficient, but the computational details are not readily accessible. Other functions, like gamma and sinh, are implemented in M-files. You can see the code and even modify it if you want. Several special functions provide values of useful constants.

pi	3.14159265...
i	Imaginary unit, $\sqrt{-1}$
j	Same as i
eps	Floating-point relative precision,
Inf	Infinity
NaN	Not-a-number

Infinity is generated by dividing a nonzero value by zero, or by evaluating well defined mathematical expressions that overflow, i.e., exceed `realmax`. Not-a-number is generated by trying to evaluate expressions like 0/0 or Inf-Inf that do not have well defined mathe-

matical values. The function names are not reserved. It is possible to overwrite any of them with a new variable, such as `eps = 1.e-6` and then use that value in subsequent calculations. The original function can be restored with clear eps

- **Examples of Expressions**

You have already seen several examples of MATLAB expressions. Here are a few more examples, and the resulting values.

```
phi = (1+sqrt(5))/2
phi =
    1.6180

a = abs(3+4i)
a =
     5

z = sqrt(besselk(4/3,rho-i))
z =
    0.3730+ 0.3214i
```

```
huge = exp(log(realmax))
huge =
    1.7977e+308

toobig = pi*huge
toobig =
    Inf
```

- **Generating Matrices**

MATLAB provides four functions that generate basic matrices.

`zeros`	All zeros
`ones`	All ones
`rand`	Uniformly distributed random elements
`randn`	Normally distributed random elements

Here are some examples.

```
Z = zeros(2,4)
Z =
```

```
     0     0     0     0
     0     0     0     0

F = 5*ones(3,3)
F =
     5     5     5
     5     5     5
     5     5     5

N = fix(10*rand(1,9))
N =
     4     9     4     4     8     5     2     6     8

R = randn(4,4)
R =
    1.0668    0.2944   -0.6918   -1.4410
    0.0593   -1.3362    0.8580    0.5711
   -0.0956    0.7143    1.2540   -0.3999
   -0.8323    1.6236   -1.5937    0.6900
```

- **M-Files**

You can create your own matrices using M-files, which are text files containing MATLAB code. Use the MATLAB Editor or another text editor to create a file containing the same statements you would type at the MATLAB command line. Save the file under a name that ends in .m. For example, create a file containing these five lines.

```
A = [ ...
     16.0     3.0     2.0    13.0
      5.0    10.0    11.0     8.0
      9.0     6.0     7.0    12.0
      4.0    15.0    14.0     1.0 ];
```

Store the file under the name magik.m. Then the statement magik reads the file and creates a variable, A, containing our example matrix.

Concatenation is the process of joining small matrices to make bigger ones. In fact, you made your first matrix by concatenating its individual elements. The pair of square brackets, `[ ]`, is the concatenation operator. For an example, start with the 4-by-4 magic square, A, and form `B = [A A+32; A+48 A+16]`. The result is an 8-by-8 ma-

trix, obtained by joining the four submatrices.

```
B =
    16     3     2    13    48    35    34    45
     5    10    11     8    37    42    43    40
     9     6     7    12    41    38    39    44
     4    15    14     1    36    47    46    33
    64    51    50    61    32    19    18    29
    53    58    59    56    21    26    27    24
    57    54    55    60    25    22    23    28
    52    63    62    49    20    31    30    17
```

This matrix is halfway to being another magic square. Its elements are a rearrangement of the integers `1:64`. Its column sums are the correct value for an 8-by-8 magic square.

```
sum(B)
ans =
    260   260   260   260   260   260   260   260
```

But its row sums, `sum(B')'`, are not all the same. Further manipulation is necessary to make this a valid 8-by-8 magic square.

• Deleting Rows and Columns

You can delete rows and columns from a matrix using just a pair of square brackets. Start with `X = A`; Then, to delete the second column of `X`, use `X(:,2) = [ ]`. This changes `X` to

```
X =
    16     2    13
     5    11     8
     9     7    12
     4    14     1
```

If you delete a single element from a matrix, the result is not a matrix anymore. So, expressions like `X(1,2) = [ ]` result in an error. However, using a single subscript deletes a single element, or sequence of elements, and reshapes the remaining elements into a row vector.

So

```
X(2:2:10) = [ ]
```

results in

```
X =
    16     9     2     7    13    12     1
```

- **Linear Algebra**

Informally, the terms matrix and array are often used interchangeably. More precisely, a matrix is a two-dimensional numeric array that represents a linear transformation. The mathematical operations defined on matrices are the subject of linear algebra. Dürer's magic square

```
A =
    16     3     2    13
     5    10    11     8
     9     6     7    12
     4    15    14     1
```

provides several examples that give a taste of MATLAB matrix operations. You have already seen the matrix transpose, `A'`. Adding a matrix to its transpose produces a symmetric matrix. `A + A'`

```
ans =
    32     8    11    17
     8    20    17    23
    11    17    14    26
    17    23    26     2
```

The multiplication symbol, *, denotes the matrix multiplication involving inner products between rows and columns. Multiplying the transpose of a matrix by the original matrix also produces a symmetric matrix. `A'*A`

```
ans =
    378    212    206    360
    212    370    368    206
    206    368    370    212
    360    206    212    378
```

The determinant of this particular matrix happens to be zero, indicating that the matrix is singular. `d = det(A)`

```
d =
     0
```

The reduced row echelon form of A is not the identity.

```
R = rref(A)

R =
     1     0     0     1
```

```
     0     1     0    -3
     0     0     1     3
     0     0     0     0
```

Since the matrix is singular, it does not have an inverse. If you try to compute the inverse with `X = inv(A)` you will get a warning message Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. `RCOND = 1.175530e-017`. Roundoff error has prevented the matrix inversion algorithm from detecting exact singularity. But the value of `rcond`, which stands for reciprocal condition estimate, is on the order of eps, the floating-point relative precision, so the computed inverse is unlikely to be of much use. The eigenvalues of the magic square are interesting.

```
e = eig(A)

e =
   34.0000
    8.0000
    0.0000
```

```
    -8.0000
```

One of the eigenvalues is zero, which is another consequence of singularity. The largest eigenvalue is `34`, the magic sum. That is because the vector of all ones is an eigenvector.

```
v = ones(4,1)

v =
     1
     1
     1
     1

A*v

ans =
    34
    34
    34
```

```
34
```

- **Arrays**

When they are taken away from the world of linear algebra, matrices become two-dimensional numeric arrays. Arithmetic operations on arrays are done element-by-element. This means that addition and subtraction are the same for arrays and matrices, but that multiplicative operations are different. MATLAB uses a dot, or decimal point, as part of the notation for multiplicative array operations. The list of operators includes

$+$	Addition
$-$	Subtraction
$.*$	Element-by-element multiplication
$*$	Matrix multiplication
$./$	Element-by-element division
$.'$	Unconjugated array transpose

If the Dürer magic square is multiplied by itself with array multiplication `A.*A` the result is an array containing the squares of the

integers from 1 to 16, in an unusual order.

```
ans =
    256     9     4   169
     25   100   121    64
     81    36    49   144
     16   225   196     1
```

- **Building Matrices and Tables**

Array operations are useful for building tables. Suppose n is the column vector

```
n = (0:9)';
```

Then

```
pows = [n  n.^2  2.^n]
```

builds a table of squares and powers of 2.

```
pows =
     0     0     1
     1     1     2
```

```
     2     4     4
     3     9     8
     4    16    16
     5    25    32
     6    36    64
     7    49   128
     8    64   256
     9    81   512
```

The elementary math functions operate on arrays element by element. So

```
format short g
x = (1:0.1:2)';
logs = [x log10(x)]
```

builds a table of logarithms.

```
logs =
    1.0          0
    1.1    0.04139
    1.2    0.07918
```

```
    1.3      0.11394
    1.4      0.14613
    1.5      0.17609
    1.6      0.20412
    1.7      0.23045
    1.8      0.25527
    1.9      0.27875
    2.0      0.30103
```

- **Scalar Expansion**

Matrices and scalars can be combined in several different ways. For example, a scalar is subtracted from a matrix by subtracting it from each element. The average value of the elements in our magic square is `8.5`, so `B = A - 8.5` forms a matrix whose column sums are zero.

```
B =
     7.5     -5.5     -6.5      4.5
    -3.5      1.5      2.5     -0.5
     0.5     -2.5     -1.5      3.5
    -4.5      6.5      5.5     -7.5
```

```
sum(B)

ans =
     0     0     0     0
```

With scalar expansion, MATLAB assigns a specified scalar to all indices in a range. For example, `B(1:2,2:3) = 0` zeroes out a portion of `B`.

```
B =
     7.5     0     0     4.5
    -3.5     0     0    -0.5
     0.5    -2.5  -1.5    3.5
    -4.5     6.5   5.5   -7.5
```

- **Logical Subscripting**

The logical vectors created from logical and relational operations can be used to reference subarrays.

Suppose `X` is an ordinary matrix and `L` is a matrix of the same size that is the result of some logical operation. Then `X(L)` specifies

the elements of `X` where the elements of `L` are nonzero. This kind of subscripting can be done in one single step by specifying the logical operation as the subscripting expression. For example if you have the following set of data.

```
x =
  2.1 1.7 1.6 1.5 NaN 1.9 1.8 1.5 5.1 1.8 1.4 2.2
```

The `NaN` is a marker for a missing observation, such as a failure to respond to an item on a questionnaire. To remove the missing data with logical indexing, use `finite(x)`, which is true for all finite numerical values and false for `NaN` and `Inf`. Thus

```
b = finite(x)
x = x(b)
```

gives the result

```
b =
   1   1   1   1   0   1   1   1   1   1   1   1
x =
   2.1 1.7 1.6 1.5 1.9 1.8 1.5 5.1 1.8 1.4 2.2
```

Vector `b` is a logical variable with elements either 0 (for false) or 1 (for

true). Logical variables are typically obtained as results of evaluating boolean expressions.

Other than the `NaN`, the element with a value `5.1` seems to be significantly larger than the others. It may represent an erroneous measurement. The following statement removes these suspicious values, in this case those elements more than three standard deviations from the mean.

```
x = x(abs(x-mean(x)) <= 3*std(x))
x =
   2.1 1.7 1.6 1.5 1.9 1.8 1.5 1.8 1.4 2.2
```

Here `mean` computes the average (or mean), and `std` computes the standard deviation of a sequence of numbers. Both are built-in MATLAB functions.

- **The find Function**

The `find` function determines the indices of array elements that meet a given logical condition. In its simplest form, find returns a column vector of indices. Transpose that vector to obtain a row vector of indices. For example, `k = find(isprime(A))'` picks out the locations,

using one-dimensional indexing, of the primes in the magic square.

```
k =
     2     5     9    10    11    13
```

Display those primes, as a row vector in the order determined by `k`, with `A(k)`

```
ans =
     5     3     2    11     7    13
```

When you use `k` as a left-hand-side index in an assignment statement, the matrix structure is preserved.

```
A(k) = NaN

A =
    16   NaN   NaN   NaN
   NaN    10   NaN     8
     9     6   NaN    12
     4    15    14     1
```

- **Controlling Command Window Input and Output**

The format Function can be used to control the appearance of numeric format of the values displayed by MATLAB. The function affects only how numbers are displayed, not how MATLAB computes or saves them. The MATLAB statement is

```
format choice
```

where `choice` should be replaced by one of the following formatting choices. For example, given a vector `x` with components of different magnitudes

```
x = [4/3 1.2345e-6]
```

here are the different format choices, together with the resulting output produced from them

choices		
short	1.3333	0.0000
short e	1.3333e+000	1.2345e-006
short g	1.3333	1.2345e-006
long	1.33333333333333	0.00000123450000
long e	1.33333333333333e+000	1.23450000000000e-006
long g	1.33333333333333	1.2345e-006
bank	1.33	0.00
rat	4/3	1/810045
hex	3ff5555555555555	3eb4b6231abfd271

If the largest element of a matrix is larger than 10^3 or smaller than 10^{-3}, MATLAB applies a common scale factor for the short and long formats. In addition to the format functions shown above format compact suppresses many of the blank lines that appear in the output. This lets you view more information on a screen or window. If you want more control over the output format, use the sprintf and fprintf functions.

- **Suppressing Output**

If you simply type a statement and press Return or Enter, MATLAB automatically displays the results on screen. However, if you end the line with a semicolon, MATLAB performs the computation but does not display any output. This is particularly useful when you generate large matrices. For example, `A = magic(100);`

- **Entering Long Statements**

If a statement does not fit on one line, use an ellipsis (three periods), ..., followed by Return or Enter to indicate that the statement continues on the next line.

For example,

```
s = 1 -1/2 + 1/3 -1/4 + 1/5 - 1/6 + 1/7 ...
    - 1/8 + 1/9 - 1/10 + 1/11 - 1/12;
```

Blank spaces around the =, +, and - signs are optional, but they improve readability.

- **Vectorization**

To obtain the most speed out of MATLAB, it's important to vectorize the algorithms in your M-files. Where other programming languages might use `for` or `do` loops, MATLAB can use vector or matrix operations. A simple example involves creating a table of logarithms.

```
x = .01;
for k = 1:10000
    y(k) = log10(x);
    x = x + .01;
end
```

A vectorized version of the same code is

```
x = .01:.01:100;
y = log10(x);
```

For more complicated code, vectorization options are not always so obvious. When speed is important, however, you should always look for ways using built-in MATLAB functions to vectorize your algorithms by eliminating loops.

- **User-defined Functions**

New functions may be added by the user to MATLAB's vocabulary if they are expressed in terms of other existing functions. A very simple function that can be specified on a single line can be defined using the `inline` function. For example,

```
g = inline('sin(2*pi*f + phase)'); y = g([0 0.5 1],pi/3)
```

will give

```
y =
    0.8660   -0.8660    0.8660
```

The first argument of `g` will be used as the first variable, `f`, and the second argument will be used as the second variable, `phase`.

A more complicated function that cannot be specified on a single line should be defined by writing a stand-alone MATLAB function. The commands and functions that comprise the new function must be put in a file whose name defines the name of the new function, with a filename extension of '.m'. At the top of the file must be a line that contains the syntax definition for the new function. For example, the existence of a file on the MATLAB path called `stat.m` with:

```
function [mean,stdev] = stat(x)
% STAT returns the mean and
% the standard deviation of a vector of real numbers.
n = length(x);
mean = sum(x) / n;
stdev = sqrt(sum((x - mean).^2)/n);
```

defines a new function called `stat` that calculates the mean and standard deviation of a vector. The variables within the body of the function are all local variables. As long as the function in on the MATLAB path, it can be used just like any other built-in functions. The following example computes the mean and standard deviation of a sequence of 10000 normally distributed numbers:

```
[m,s] = stat(rand(10000,1))
```

The output will be

```
m =
    0.5005
s =
    0.2887
```

A subfunction that is visible to the other functions in the same file is created by defining a new function with the FUNCTION keyword after the body of the preceding function or subfunction. For example, avg is a subfunction within the file stat2.m and is not visible to any other functions:

```
function [mean,stdev] = stat2(x)
% STAT returns the mean and
% the standard deviation of a vector of real numbers.
n = length(x);
mean = avg(x,n);
stdev = sqrt(sum((x-avg(x,n)).^2)/n);

%-----------------------
function a = avg(x,n)
%MEAN subfunction
a = sum(x)/n;
```

Notice `avg` returns the value of a single variable `a`, while `stat2` returns the values of two separate variables, `mean` and `stdev`. In general, the

values of any number of variables of different types and sizes can be returned by a function. Remember to put commas separating the variable names in the return argument list.

2.3. MATLAB Graphics

MATLAB has extensive facilities for displaying vectors and matrices as graphs, as well as annotating and printing these graphs.

- **Simple graphs**

The plot function 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 you specify two vectors as arguments, `plot(x,y)` produces a graph of `y` versus `x`. For example, these statements use the colon operator to create a vector of `x` values ranging from zero to 2π, compute the sine of these values, and plot the result.

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

```
plot(x,y)
```

Now label the axes and add a title. The characters pi create the symbol π.

```
xlabel('x = 0:2\pi')
ylabel('sin(x)')
title('Plot of the Sine Function','FontSize',12)
```

- **Multiple Data Sets in One Graph**

Multiple `x-y` pair arguments create multiple graphs with a single call to plot. MATLAB automatically cycles through a predefined (but user settable) list of colors to allow discrimination among sets of data. For example, these statements plot three related functions of x, each curve in a separate distinguishing color.

```
y2 = sin(x-.25);
y3 = sin(x-.5);
plot(x,y,x,y2,x,y3)
```

The legend command provides an easy way to identify the individual plots.

```
legend('sin(x)','sin(x-.25)','sin(x-.5)')
```

Most of the attributes of a plot, such as color, line style, line thickness and font size, can be edited and modified interactively using tools that are available in the figure window.

References

[1] Most the materials here are adopted from MATLAB's online documentation, which can be accessed within MATLAB by selecting MATLAB Help from the Help menu. The documentation can also be accessed through the *MATLAB website*.

[2] K. Sigmon and T. A. Davis, *MATLAB Primer*, Sixth Edition, Chapman and Hall/CRC, 2002.

[3] D. J. Higham and N. J. Higham, *MATLAB Guide*, SIAM, 2000.

[4] D. C. Hanselman and B. Littlefield, *Mastering MATLAB 6, A Comprehensive Tutorial and Reference*, Prentice-Hall, 2000.