1 MATLAB ESSENTIALS

1.1 MATLAB Environment

1.2 MATRICES AND VECTORS

1.1 MATLAB Environment

The term *MATLAB* is an acronym for ***MAT***rix ***LAB***oratory, a commercial software product originally created by Cleve Moler in the late 1970s as a user-friendly interface to numerical libraries for linear algebra. Unlike *Mathematica* and *Maple*, which emphasize the manipulation of symbolic expressions, MATLAB's focus is computation. However, MATLAB is more than an engine for computation. It can be viewed as a "super-powerful graphics calculator with many buttons" ([p. 9] Scepanovic 2010). Clearly, MATLAB's primary strengths are rapid code prototyping and data manipulation. Consequently, it is excellent environment investigating for small and medium scale problems (Kelley, 2003).

Observe that the font convention in these handouts is that my text is typed in 10 pt Sans Serif, but MATLAB commands and responses are typed in **`bold 10 pt Courier font`**.

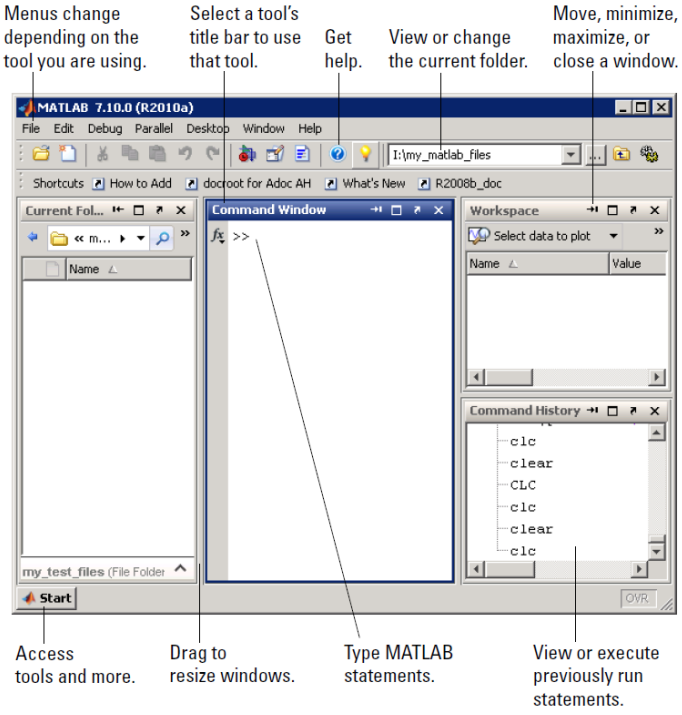
1.1.1 Start and Termination of a MATLAB Session

Start MATLAB by double clicking on the desktop icon labeled MATLAB 7.12 OR find it in **start programs – Classes – MATLAB 7.12 → window below appears!**

Quit MATLAB by typing **quit** OR **exit** in the command window.

1.1.2 Desktop Environment

The sizes of these windows and their screen locations, as well as well as which windows are present are all customizable via the Desktop Layout tab under the Desktop icon (top tool bar) on the MATLAB toolbar. To change the default layout read chapter 2 of *MATLAB® 7 Desktop Tools and Development Environment* (matlab_env.pdf), which I have placed in the mathlab_documentation_2011 directory on the class site.



[matlab_env.pdf, p. 2-3].

Command Window: <= Large panel in the center of the previous figure.
The Command Window is the primary window for communicating with MATLAB. Placing the prompt (the blinking vertical bar) at (>>) in the Command Window indicates that MATLAB is ready to accept input from you. When you see this, you can enter a variable or a name of a function via the keyboard. When you press the *Enter* key after typing this line, MATLAB will display the result -

```
>> 6*7
ans = 42
```

If you do not assign a variable name to a calculation, MATLAB assigns the default name, **`ans`**, to your result.

Suppression of screen output. The *default* MATLAB behavior is to display *all* output to the screen. However, inserting a semicolon at the end of a command line suppresses such screen output.

Previous commands can be accessed by hitting ↑ (the up arrow key).

Clear command window by typing **`clc`** and hit Enter. However, previously defined variables are unchanged.

Inserting comments – use % - text following % symbol is ignored by MATLAB.

Line continuation. Three or more periods at the end of a line indicates the continuation of an expression on the next line in the Command Window.

One can change the ***default*** command-window properties via

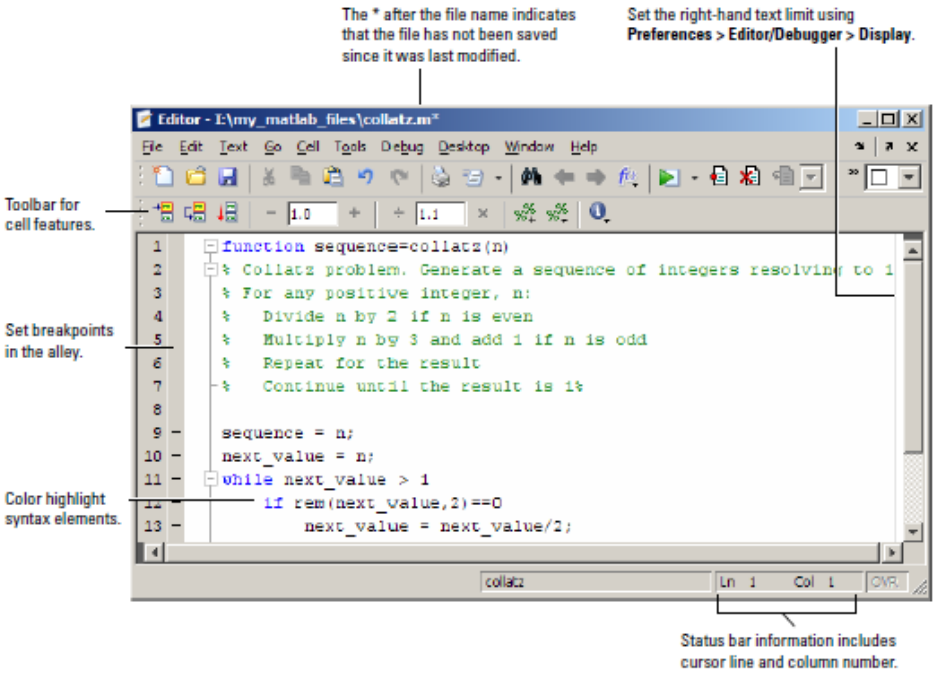
File tab → Preferences tab → click Command Window

In Text display set
Numeric format: short g ← my window settings
Numeric display: compact

In display set
Click off Show getting started message bar ← my window settings
Click on Show Function Browser button

Editor: [Not shown in the Desktop Environment figure, but is illustrated on the next page.]
Rather than type repeatedly MATLAB instructions, you can create files of these MATLAB commands and then activate them at a single command. The name of such command files must end with an **`.m`** file extension. To create a new m-file in the Editor, either click the new file icon (the white rectangle with a red dot in the upper left corner) on the second row of the MATLAB toolbar, OR select File -> New -> M-file from the MATLAB toolbar. Alternatively, type **`edit`** in the Command Window. To open an *existing* M-file in the Editor, click the open-file icon (open folder) on the toolbar (2nd line) or select File -> Open. The Open dialog box appears, listing all MATLAB files in the work-space directory.

Once you have finished typing your instructions, save the file via either the Save command found under the File tab, OR click the blue floppy disk icon in the editor window. You can run your file *in* the Editor/Debugger window by clicking the execute icon (solid green triangle superimposed on pale blue square) on the second row. Alternatively, you can run your m-file in the Command Window by typing its name (*without* the .m extension) and hitting the *Enter* key.

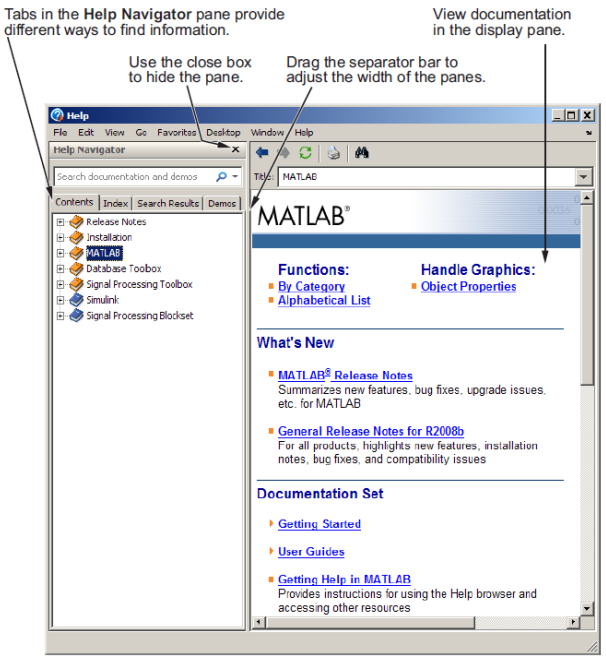


[matlab_env.pdf, p. 9-7].

Help Browser: [Not shown in the Desktop Environment figure but illustrated below.]
To open the help browser:

click on the Help tab on the top toolbar of the environment window, OR,
click the blue-circled **?** symbol (immediately below the Help tab), OR,
type **`helpbrowser`** in the Command Window.

NOTE that the tabs in the Help Window Navigator (shown below) provide different ways to search for information: contents, index, or search.



Function Browser: A *second* on-line help option is the *function browser*. MathWorks added this new help tool in MATLAB 7.7.0 (R2008b), It is easy to miss, since it is activated by clicking the ***fx*** symbol to the *immediate* left of the command prompt, >>. Alternatively, click on Function Browser under the Help tab. Then a screen similar to that below appears –

[matlab_env.pdf, p. 3-4].

Mathworks has created a t*hird* option for accessing help while running MATLAB - type help followed by the name of the function -

```
>> doc name_of_function
```

in the Command Window environment. If you do not know/remember the name of the function you can type –

```
>> lookfor name_of_keyword
```

Finally, you can access MathWorks website from MATLAB under the Help tab => Web Resources => The MathWorks Web Site.

Finally, for those of you who prefer to read manuals, I have placed in the matlab_documentation_2011 directory of the class site the following manuals for MATLAB 7.11.0 (R2010b): (The corresponding pdf files do not exist foe MATLAB 12.)

MATLAB® 7 Getting Started Guide (getstart.pdf)	(280 pg.)
MATLAB® 7 Desktop Tools and Development Environment (matlab_env.pdf)	(1063 pg.)
MATLAB® 7 Mathematics (math.pdf)	(532 pg.)
MATLAB® 7 Programming Fundamentals (matlab_prog.pdf)	(684 pg.)
MATLAB® 7 Programming Tips (programming_tips.pdf)	(64 pg.)
MATLAB® 7 Object-Oriented Programming (matlab_oop.pdf)	(551 pg.)
MATLAB® 7 Data Analysis (data_analysis.pdf)	(216 pg.)
MATLAB® 7 Graphics (graphg.pdf)	(687 pg.)
MATLAB® 7 3D Visualization (visualize.pdf)	(210 pg.)

There exist over *1200* books on MATLAB programming, but I recommend the following two volumes for those of you who prefer textbooks:

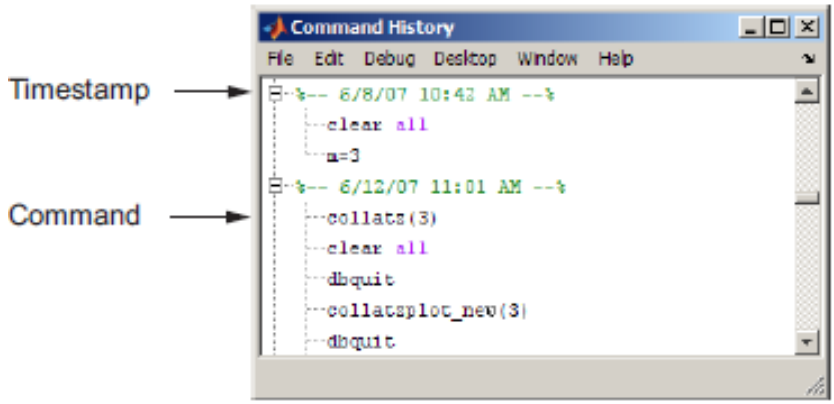
Driscoll. T. A. 2009 *Learning MATLAB* (SIAM: Philadelphia).

Higham, D. J., & Higham, N. J. 2005 *MATLAB Guide 2nd Ed.* (SIAM: Philadelphia).

Electronic Resources at http:/libraries.claremont.edu

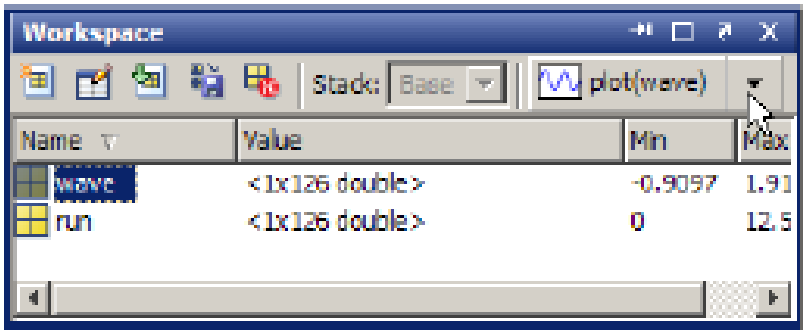
Brian R. Hunt, Ronald L. Lipsman, Jonathan M. Rosenberg, with Kevin R. Coombes, John E. Osborn, and Garrett J. Stuck, 2006 *A Guide to MATLAB for Beginners and Experienced Users 2nd Ed.* (Cambridge University Press: Cambridge, UK).

Command History: <= small panel on the lower right in the desktop environment figure.
The record of commands you create remains on your computer across all MATLAB sessions. You can view a log of these commands you have entered in previous sessions. Note a timestamp marks the beginning of each MATLAB session. Double-click any entry (entries) in the Command History window to execute these statement(s).



[matlab_env.pdf, p. 3-33].

Workspace Browser: <= small panel on the upper right in the desktop environment figure.
The Workspace browser shows the properties of all the currently defined variables. It is empty at startup of a session. The name of each variable, its array size, and its size in bytes can be listed.



[matlab_env.pdf, p. 6-10].

Note that the workspace variables are *not* maintained across MATLAB sessions. When you quit MATLAB, the workspace variables are deleted. To save *all* the workspace variables, from the File menu on the top bar of MATLAB window, select Save Workspace As, OR click the save button (the floppy disk icon – 4th icon over) in the Workspace browser toolbar.

1.1.3 Computer Representation of Real Numbers

This is an issue of hardware, e.g., floating-point co-processors, NOT software!

The following discussion is dry, arcane, and boring, but it is important!

Real numbers, the critical elements of science are an *uncountable* set spanning from $-\infty$ to $+\infty$; practically all *quantitative* descriptions of scientific phenomena are described by real numbers ([p. 106] Ueberhuber 1997) In computer arithmetic these real numbers are represented by floating point numbers. The fundamental properties of such floating-point numbers are considerably different from those of real numbers. Floating-point numbers are: *countable* (in other words both finite and discrete), and have *non-infinite* maximum and minimum values.

Any real number in a floating-point system can be described as –

$$y = \pm m \times \beta^{e-t}.$$

Formally such a floating-point number system is characterized by four *integer* parameters ([p. 37] Heath 2002; [p. 36] Higham 2002):

the base, β,
precision, t,
exponent range $e_{min} \le e \le e_{max}$.

MATLAB running on Intel[1] chips represents reals numbers in the IEEE double floating-point representation:

$$\begin{aligned} \beta &= 2 \\ t &= 53 \\ e_{min} &= -1021 \\ e_{max} &= +1024 \end{aligned}$$

([p. 37] Higham 2002). The *significand*, *m*, is an *integer* satisfying $0 \le m \le \beta^t - 1$. To create a unique number representation, for every nonzero y it is assumed that the system is normalized[2], thus always $m \ge \beta^{t-1}$. The range of nonzero floating-point numbers is

$$\beta^{e_{min}-1} \le |y| \le \beta^{e_{max}}(1 - \beta^{-t}).$$

[1] Hardware (Intel co-processors) inside our computers use what is termed IEEE double precision. IEEE stands for Institute of Electrical & Electronic Engineers.

[2] The term normalized means that the lead digit of the mantissa is not zero. Consequently, the most significant bit is always 1 and is not stored, and this so-called hidden bit is the "+1".

Thus the smallest positive normalized floating-point number is $\beta^{e_{min}-1} = 2.2251 \times 10^{-308}$. In MATLAB this number (2.2251×10^{-308}) is represented by the constant **realmin.** *Underflow* is the term employed to describe arithmetic operations, which create positive floating-point numbers, smaller that this limit. The largest positive normalized floating-point number is $\beta^{e_{max}}(1-\beta^{-t})$, which is 1.7977×10^{308}. In MATLAB this number (1.7977×10^{308}) is represented by the constant, **realmax.**

Note that the number *zero* cannot be represented by a normalized number. It is defined by one of the reserved exponent bits, by setting $e = e_{min} - 1$, and by setting the significand to zero (Goldberg, 1991). Perhaps surprisingly, the number of *normalized* numbers is finite and is $2(\beta - 1)\beta^{t-1}(e_{max} - e_{min} + 1)$, which for the Intel coprocessors corresponds to 1.8429×10^{19}. Finally, floating point numbers are not distributed uniformly between the values, **realmin** and realmax, but are uniformly spaced between increasing powers of β.

Note that there exists no positive *normalized* floating-point numbers between 0.0 and is $\beta^{e_{min}-1}$ (**realmin**). However, smaller numbers can be created by relaxing the constraint that significands must be normalized. Clearly, these *de-normalized* floating point numbers can be smaller than $\beta^{e_{min}-1}$, but at the expense of *reduced precision*. The smallest de-normalized positive number is $\beta^{e_{min}-t} \cong 5 \times 10^{-324}$. This use of de-normalized floating-point numbers is termed *gradual underflow*, which MATLAB employs.

Two additional values are associated with IEEE floating-point number systems, which you need to be familiar with. First, is **Inf**, which represents infinity. In MATLAB the largest number that can be represented in the floating-point system is **realmax**. Whenever, an operation produces a result greater than **realmax** (i.e. *overflows*), the resultant number becomes **Inf**. The symbol **Inf** is represented by $e = e_{max} + 1$ and a zero significand field ([p. 134] Ueberhuber 1997). If arithmetic operations are performed such that a result is undefined, e.g., 0/0, ∞/∞, and $0 \times \infty$, the result is replaced by the symbol **NaN,** *not a number*. **NaN** is represented by $e = e_{max} + 1$ and a non-zero significand ([p 134] Ueberhuber 1997).

The concept of machine precision represents the accuracy of the floating-point system. This machine parameter is important because it sets the level of the *smallest possible relative error* in representing a real number in the floating-point number system. Alternatively, it can be described as the smallest number ε such that $(1 + \varepsilon) > 1$. Depending on the rounding process, $\varepsilon = \beta^{1-t}$. In MATLAB this parameter is represented as the intrinsic constant, **eps** $= 2.2204 \times 10^{-16}$.

By *default* MATLAB represents a number as a *matrix* in the IEEE double floating-point representation. The double format and the 16-bit character are the primary data type used in MATLAB. However, other data types are supported: 32-bit singles, integers (8-bit, 16-bit, 32-bit, 64-bit), as well as 2-bit logicals.

Observe that MATLAB can display large/small *numeric* variables in scientific (floating point) notation, Thus

3.782×10^{11} is represented by 3.782e+011.

Note that here e+XX represents 10^{XX} and **not** 2.7183^{XX}. Moreover you cannot employ *fractional* powers of ten in scientific format -- you cannot use 3.46e9.5. Moreover, the following format is *not* an appropriate representation of a number - 3.56^10^9.

To *summarize* three built-in MATLAB parameters encode the fundamental properties of IEEE floating-pint representation: **`eps`**, **`realmin`**, and **`realmax`**. Emphasizing core points discussed above are the following!

precision: It is the *smallest* positive number, **`eps`**, such that MATLAB can distinguish **`(1.0 + eps)`** from **`eps`**. This value, **`eps`**, indicates the precision, or *smallest possible relative error*, at which calculations are performed –

```
>> eps
ans = 2.2204e-016
```

Thus MATLAB executes all elementary arithmetic operations for double class variables with ~16 *decimal* digits of accuracy.

realmin and realmax: The greatest number one can create in MATLAB is **`realmax`**. In decimal notation it is

```
>> realmax('double')
ans = 1.797693134862316e+308
```

Although numbers can possess up to 308 *decimal* digits, *but only 16 decimal digits are significant*! Observe, for example,

```
>> realmax('double')
ans = 1.797693134862316e+308
>> realmax ('double')+ 1.0e291
ans = 1.797693134862316e+308
>> realmax('double') + 9.0e291
ans = 1.797693134862316e+308
```

Numbers greater than this value can be created easily by accident (divide any positive or negative number by 0) and result in what is termed *overflow* and the resultant arithmetic value is set to infinity, represented by the symbol **`Inf`** -

```
>> realmax('double') + 1.0e292
ans = Inf
>> 10.0/0.0, -10.0/0.0
ans =   Inf
>>  -10.0/0.0
ans =  -Inf
```

Please observe that overflows do not halt calculations.

The behavior of small numbers is different. There exists another parameter called **`realmin`**,

```
>> realmin('double')
ans = 2.225073858507201e-308
```

`realmin` is the smallest floating-point number one can create with16-decimal digit *accuracy*. Such numbers are termed normal double-precision variables. Note you create numbers smaller than **`realmin`**, but at the cost of greatly reduced precision. However note that numbers less than **`eps*realmin`** are set to zero.

complex numbers: Complex numbers can be created simply by specifying, **`x + iy`**, where **`x`** represents the real part, **`y`** corresponds to the imaginary part, and i has been predefined by MATLAB to be √-1. Alternatively, a complex number can be created using the function **`complex`** -

```
>> b = 6 +9i
b = 6 +        9i
>> a = complex(5, 8)
a = 5 +        8i
```

Please use the function browser, **fx**, to identify some of the many intrinsic MATLAB functions which operate on complex numbers -

```
>> real(a)
ans = 5
>> imag(b)
ans = 9
```

Note that complex numbers do *NOT* constitute a distinct data class. Such numbers by default store their real and imaginary components in double precision.

```
      >> class(a)
ans = double
```

single 32-bit: The use of the function, **`single(A)`**, converts a numeric variable to the single 32-bit format. The major reasons for employing the *single* class are to save storage space of real data and investigate errors in algorithms by employing both single and double arithmetic. As seen immediately below, the use of the single data type reduces greatly the range and precision of floating point numbers. Since the employment of single precision does not lead to a significant speed advantage, it use in coding is not recommended.

```
>> realmax('single')
ans =  3.4028e+038
>> realmin('single')
ans =  1.1755e-038
>> eps('single')
ans =  1.1921e-007
```

logical: This data type possesses only values of **`0`**, signifying false, or **`1`**, representing true. More significantly, arrays of logical data type are created by the relational operators (==,<,>,~, etc.) and functions like **`any, all, isnan, isinf,`** and **`isfinite`**.

intXXX: MATLAB can create and manipulate integer variables with *eight* integer format lengths, both signed (**`int8, int16, int32, int64`**) and unsigned (**`uint8, uint16, unit32, unit64`**). MATLAB possesses limited arithmetic operations to manipulate this class of numeric data. Its primary use is the efficient storage of images.

```
>> intmin('int8')
ans = -128
>> intmax('int8')
127
>> intmin('uint8')
ans =  0
>> intmax('uint8')
255
>> intmin('int32')
ans = -2147483648
>> intmax('int32')
2147483647
```

Note the unusual behavior that integers created *outside* their designated range does NOT generate underflows/overflows (0, Inf), as found for floating-point numbers, but instead simply map to the ends of their range -

```
>> intmin('int8') - 10
ans =-128
>> intmax('int8') + 10
ans = 127
```

char: String arrays are created by placing single quotes around characters or by applying the **char** function to 65 to 90 to create the capitals A:Z and 97 to 122 to create the lower case, a:z. MATLAB has a number of built-in functions for manipulating strings.

```
>> s = 'name of city'
s = name of city
>> char([67 80 81])
ans = CPQ
```

Assigning Variable Names

Employ the = symbol to *assign* values to variables.

To check a value of an assigned variable, double click on variable in the Workspace window OR type its name. Alternatively, type in the Command Window,

```
>> disp(x);
```

To *remove* a variable, type (in Command window)

```
>> clear variable_name,
```

OR use the Workspace window - click on variable and then click the delete icon (2-by-2 graph with red-circled x in lower right corner).

Variable Names

Variable names must:

Consist of letters (a-z, A-Z), digits (0-9), and underscore (_),
Distinguishes UPPER & lower case letters.
Must start with a letter.
Can contain a maximum of 63 characters.
Cannot contain blanks.

HINT: Do not use cryptic names, but employ name, which make sense, e.g., velocity, mass, etc.

Built-In MATLAB Variables

pi	3.14159265358979
i, j	$\sqrt{-1}$
Inf	result of n/0, where n is any non-zero real number
NaN	Not a number, 0/0 or 0*Inf,

Using the name of such pre-defined variables as names of variables by a user can lead to difficult to predict errors.

1.2 MATRICES AND VECTORS

Observe that arrays and matrices are *different* ([p. 11] Driscoll 2009). An array is simply a collection of numbers, termed *elements* or *entries*, referenced by one or more indices. In MATLAB the indices are consecutive integers starting at 1. The array *dimension* is the number of indices required to specify an element. A matrix is a two-dimensional array with *special* rules for arithmetic operations ([p. 11] Driscoll 2009). *Matrix* operations are predicated on the rules of linear algebra; a matrix represents a linear transformation, The two dimensions are termed rows and columns. A vector is a matrix with one dimension of 1; a row vector consecutive s one row and a column has only one column.

1.2.1 Creation of Vectors

Row vector: → separate elements in rows by blanks or commas, and enclose vector with ***square*** brackets, [ ].

```
>> row = [1 2 3]           % 1 by 3 row vector
row =  1   2     3
```

Column vector: → separate elements by semicolons.

```
>> column = [4; 5; 6]      % 3 by 1 column vector
column =
4
5
6
```

You discriminate row vectors from column vectors by:

- Inspect properties of vector in the Workspace
- Display variable.
- Use **`size`** function ([p. 24] Scepanovic 2010).

```
>> size(row)              % row vector 1 is 1st element of size
ans = 1     3
>> size(column)           % column vector 1 is 2nd element of size
ans = 3     1
```

To ascertain the *number* of elements in either a row or column vector use the MATLAB built-in functions, **`length`** or **`numel`** -

```
>> length(row)
ans = 3
>> length(column)
ans = 3
```

Colon Operators:

vector = value of 1st element: step : value of last element

```
>> x = 0.3*(0.0:.05:.15)
```

```
x = 0    0.0150    0.0300    0.0450
```

NOTE: step does NOT need to be positive

```
>> y = 10:-2:0
   y = 10   8    6     4     2     0
```

Subscript Indices:

Remember that the indices of vector elements start at 1 and *not* 0.

MATLAB syntax employs *matching* parentheses, v(k) identify to the k^{th} element of vector, v:

```
>> v = 12:2:18
v = 12    14    16    18
>> v(1)
ans = 12
>> v(3)
ans = 16
>> v(5)
??? Index exceeds matrix dimensions.
```

Observe that the index *itself* can also be a *vector*:

```
>> v([4 1 3 2 3])
ans =  18    12    16    14    16
```

One can extract element or elements from a previously defined vectors using colon notation:

```
>> v(2:2:4)
ans = 14    18
```

More sophisticated identifications/extractions can be performed using subscript indices -

```
>> a = y(2:2:end)
a = 8     4     0
>> b = y([3 1 4])
b = 6    10     4
```

Transpose:

Transpose of row vector is a column vector and the transpose of column vector is a row vector -

```
>> y = 10:-2:4
 y =   10   8    6     4

>> transpose(y)
ans =          10
               8
               6
               4

>> c = [4; 5; 6];
>> transpose(c)
 ans = 4     5     6
```

NOTE the following in transposing complex number vectors -

```
>> z = complex( [1; 2; 3], [4; 5; 6]) % creates complex column vector
z =          1 +          4i
             2 +          5i
             3 +          6i
>> size(z)
ans =  3     1     % three rows 1 column → column vector!
>> y = transpose(z)
y = 1 + 4i  2 + 5i  3 + 6i
>> size(y)
ans = 1 3          % 1 row three columns  → row vector

% Hermitian transpose (transposes & conjugates all complex numbers)

> g = ctranspose(z)
g = 1 - 4i  2 - 5i  3 - 6i
```

Additional vector functions:

Dot and cross product (from PHYS33/34) - `% some familiar vector functions`

```
>> y = 10:-2:0
 y =   10   8   6   4   2   0
>> w = 5:1:10
w  =   5   6   7   8   9   10
>> dot(w, y)
ans = 190
>> cross(w(1:3), y(1:3))  % w & v must be 3-element vectors
ans = -20    40   -20
```

`sum(x)`	returns the sum of the element of x.
`prod(x)`	returns the product of the elements of x.
`sort(x)`	sorts x in ascending order [sort in *descending* order, `sort(x, 'descend')`]
`sum(x)`	sum of the elements of x.
`min(x)`	returns the smallest element of x.
`[y, k]`	`= min(x)` (y is the smallest element of x and k is the index of y).

```
x = 158   971   958   486   801
>> length(x)
ans = 5
>> size(x)
ans = 1     5
>> sum(x)
ans = 3374
>> prod(x)
ans = 5.7215e+013
>> sort(x)
ans = 158   486   801   958   971
>> [v, k] = min(x)
v = 158
k = 1
```

1.2.2 Creation of Matrices

Matrix → *rectangular* tabulation of elements.

Observe that the indices of matrix elements, as do vectors, start at 1 and *not* 0.

Indexing matrices via *subscript* notation:

A(k, m) refers to A_{km} → element in k^{th} row and m^{th} column –

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} & A_{15} \\ A_{21} & A_{22} & A_{23} & A_{24} & A_{25} \\ A_{31} & A_{32} & A_{33} & A_{34} & A_{35} \\ A_{41} & A_{42} & A_{43} & A_{44} & A_{45} \end{bmatrix}$$

Note: MATLAB does *not* discriminate vectors from matrices –

Row vectors are just matrices with k = 1 and *column* vectors are matrices with m = 1!

One creates matrices in several ways:

Bracket Notation:

Construct *explicitly* element by element using [] notation with the ends of rows defined by semicolons --

```
>> A = [2 3 4; 5 6 7; 8 9 10]
A =
    2     3     4
    5     6     7
    8     9    10
```

Concatenating predefined vectors/matrices:

```
>> a = [1  2];
>> b = [3  4];
>> c = [5; 6];
>> d = [a; b]
d =
  1     2
  3     4
>> e = [d c]
e =
  1     2     5
  3     4     6
>> f = [[e e]; [a b a]]
f =
  1     2     5     1     2     5
  3     4     6     3     4     6
  1     2     3     4     1     2
```

Automatic Initialization: ↔ class of matrix/vector functions

`linspace(x1, x2, N)` generates a row vectors with N equi-spaced points between x1 and x2.

```
>> y = linspace(0, 1, 5)
y = 0    0.2500    0.5000    0.7500    1.0000
```

`zeros(m,n)`	m-by-n matrix of 0s.
`ones(n)`	n-by-n matrix of 1s.
`eye(n)`	n-by-n identity matrix.
`rand(n,m)`	n-by-m matrix of uniformly distributed *real* random numbers
`randm(n)`	n-by-n matrix of normally distributed random numbers
`randi([`i_{min}`,`i_{max}`],m,n)`	returns an m-by-n matrix of random *integers* uniformly distributed between i_{min} and i_{max}.

```
>> x = rand(3,1)
x =   0.82346
      0.69483
      0.31710

>> y = rand(1, 5)
y =  0.95022     0.034446      0.43874      0.38156      0.76552

>> z = randi(10000, 3, 3)
z =     7952         4456         7547
        1869         6464         2761
        4898         7094         6798

>> a = ones(2)
a =  1     1
     1     1

>> b = zeros(3,4)
b =  0     0     0     0
     0     0     0     0
     0     0     0     0

>> i = eye(3)
i =  1     0     0
     0     1     0
     0     0     1
```

Matrix Functions ↔ operates on *previously* created matrices

Essential property of *all MATLAB* matrix functions
Default behavior!
Each *column* of a matrix is treated as a *distinct* vector!

`sort(A)`	sorts each *column* of A in ascending order.
`sort(A, 1)`	sorts each *column* of A in ascending order [same as default behavior].
`sort(A, 2)`	sorts each *row* of A in ascending order.
`sum(A)`	is a *row* vector whose elements are sums of *columns* of A.
`sum(A, 2)`	is a *column* vector whose elements are sums of rows of A.
`size(A)`	returns a vector containing the number of rows & columns of A.
`min(A)`	row vector containing the minimum of each column of A.
`sum(A, 1)`	returns a vector whose elements are the sum of the columns of A.
`sum(A, 2)`	returns a vector whose elements are the sums of the rows of A.

```
>> A = [9 897 44; 31 20 50; -75 -62 -98]
A =
     9   897    44
    31    20    50
   -75   -62   -98
```

```
>> sort(A)
ans =
   -75   -62   -98
     9    20    44
    31   897    50

>> sort(A, 1)
ans =
   -75   -62   -98
     9    20    44
    31   897    50

>> sort(A, 2)
ans =
     9    44   897
    20    31    50
   -98   -75   -62
>> sum(A)
ans =
   -35   855    -4
>> sum(A, 1)
ans =   -35   855    -4
>> sum(A, 2)
ans =    950
         101
        -235
>> min(A)
ans =   -75   -62   -98
```

Subscript Indices and Colon Operators:

```
A =
   142   960   934   393    32
   422   656   679   656   277
   916    36   758   172    47
   793   850   744   707    98

>> x = A(1,:)  % Extract 1st row
x = 142   960   934   393    32

>> y = A(:,1)  % Extract 1st column
y =
   142
   422
   916
   793

>> A(:,1) = [1; 2; 3; 4] % Add new 1st column
A =
     1   960   934   393    32
     2   656   679   656   277
     3    36   758   172    47
     4   850   744   707    98

>> C = A(3:end, 2:end)  % Extract an array section
C =
    36   758   172    47
   850   744   707    98
```

Alternative way to reference matrix elements:

Often one sees reference to A(:) when A is a matrix. This operation transforms a two-dimensional matrix, A, into a corresponding vector, constructed in *column* major order:

```
>> c = C(:)
c = 36
   850
   758
   744
   172
   707
    47
    98
>> size(C)
ans = 2     4
>> size(c)
ans = 8     1
```

This approach leads to an alternative *linear* indexing of matrices/arrays exists (in addition to the more familiar *subscript* indexing):

```
>> C(1)
ans = 36
>> C(2)
ans = 850
>> C(3)
ans = 758
>> C(6)
ans = 707
```

Such linear indexing applies to *any* matrix independent of the use of the colon operator!

1.2.3 save/clear/load

You can save the variables you have created during a MATLAB session to a separate file via the **save** command. To save *all* the workspace variables for later use, type in the Command Window –

```
>> save name_of_file
```

All the variables are stored in a (machine-readable not text readable) file called **name_of_file.mat** in the current directory. If you type

```
>> save
```

All the variables are store in a file with the default name, **matlab.mat**. If you type at the prompt

```
>> save name_of_file_2 x y z
```

only the variables x, y, and z are saved in a file named **name_of_file_2.mat**. To retrieve the variables in another MATLAB session, type at the prompt

```
>> load name_of_file
```

You do not need to use the extension **.mat** – **.mat** is assumed.

Employ **clear** to remove variables, **x, y, z**, from the desktop:

```
>> clear x y z
```

Typing **clear** by itself, removes *all* variables from the command window!

To save a *copy* of all the *commands* as well as the resulting screen output, type at the *beginning* of a MATLAB session, at the prompt, >> **diary('Name_of_file')**. Don't forget to do this! All *subsequent* commands and outputs will be copied to the text file named, **Name_of_file.** NOTE that if you type ONLY

```
>> diary
```

your subsequent commands and returning outputs will be copied to a file named **diary**. NOTE: the **diary** command does *not* save graphics and variables. To turn off diary storage, type

```
>> diary off
```

To turn on diary storage type

```
>> diary on
```

Observe that you *cannot* access the diary file while it is on!

The use of the **diary** command is an excellent way to create home works and tests.

1.2.4 Arithmetic Operations

Scalar operations: add (+), subtract(-), multiply (*), divide (/), & exponentiation (^).

```
>> x = 5^4
x = 625
>> y = x/12
y = 52.083
>> z = x + y
z = 677.08
>> w = z - 3333.0
w = -2655.9
>> u = w/0
u = -Inf
>> t = u/Inf
t =  NaN
```

In complicated arithmetic operations Scepanovic (2010) urges MATLAB users to employ parentheses! Why, is *not* explained, the use of matched is parentheses is suggested because the rules of *precedence* apply!

Precedence are presented here as an encouragement to employ parentheses!
You are not expected to remember them!

Precedence: How does MATLAB interpret, 5*3^3? Is it the cube of the product of 5 times 3 [no], or is it the product five times three cubed [yes]? What is the actual order of the operations? The order is set by rules of precedence.

The precedence rules are ordered from the highest (1) to lowest precedence level (4):

1) Parentheses ().

2) Exponentiation (^).
3) Multiplication (*) and division (/).
4) Addition (+) and subtraction (-).

Within *each* precedence level, operators with *equal* precedence, say a multiplication and a division, are evaluated in order from left to right -

```
>> 1 - 1/2
ans =  0.5
>> 3 + 4/3 + 4^2
ans = 20.333
```

Use parentheses to make your intent clear!

Common Math Functions

To identify *intrinsic* functions ---
Help tab -> MATLAB Help -> Functions: By Category -> Mathematics -> Elementary Math

NOTE: access a function by passing input arguments enclosed by a matched pair of parentheses -

Some common built-in functions:

Math Notation	MATLAB name	
sinx	sin(x)	
cosx	cos(x)	
tanx	tan(x)	
|a|	abs(a)	
lnx	log(x)	% In MATLAB natural log is log NOT ln
$\log_{10}x$	log10(x)	% In MATLAB $\log_{10}$ is NOT log
$\sqrt{x}$	sqrt(x)	

Vector operations

Vector arithmetic operations act *element by element*. NOTE: *presence* of the periods!
Observe that *sizes* of x and y must be the *same* unless x or y is a scalar!

```
x.*y  → [x1*y1 + x2*y2 + …]
x./y  → [x1/y1 + x2/y2 + …]
x.^2  → [x1^2 + x2^2  + …]
```

NOTE: *without* period these operations are very different!

```
>> x = 1:4;
>> y = 10:-1:7;
>> x.*y
ans = 10    18    24       28
>> y./x
ans = 10    4.5   2.6667   1.75
>> x.^2
ans = 1     4     9        16
>> x*y
??? Error using ==> mtimes      % error message → explained below!
Inner matrix dimensions must agree.
```

Bulk of MATLAB's Intrinsic functions work with vector inputs -

```
>> exp(x)
ans =  2.7183       7.3891       20.086       54.598
>> sin(x)
ans = 0.84147       0.9093      0.14112      -0.7568
>> exp(-x).*sin(x)
ans = 0.30956      0.12306     0.007026    -0.013861
```

Array Operations

Some (*, /, ^) arithmetic operations function *differently* for arrays and matrices. *Matrix* operations are predicated on the rules of linear algebra. However, *all* arithmetic *array* operations work on an *element-by-element* basis and thus require same-sized parent arrays.

```
>> A = [1 2; 3 4]; B = 3*ones(2);
A =   1     2
      3     4

B =   3     3
      3     3

>> A + B
ans = 4     5
      6     7

>> C = A - B
C =   -2    -1
       0    +1
```

Array multiplication (multiply each *single* element pairs): `C(i, j) = A(i, j).* B(i, j)`

```
>> A.*B
ans = 3     6
      9    12
```

NOTE the presence of the *period* before the asterisk.

Array division: `C(i, j) = A(i, j)./ B(i, j)`

```
>> A./B
ans = 3.3333  0.66667
      1.0000  1.33333
```

NOTE the presence of the period before the slash.

Matrix Operations:

i) A + B *same* result as array additions, but valid only if A & B same size

ii) A – B *same* result as array additions, but valid only if A & B same size

BUT

iii) A*B (matrix operation defined by the *absence* of the period!)
Valid if number of columns of A = number of rows of B →
Matrix multiplication does *not* require A and B be same size

Matrix multiplication ↔ *Defining* property of a matrix

Matrix multiplication *is* *different* from element–by-element array multiplication. Observe that

A = n-by-m sized matrix, B = m-by-k sized matrix

C ≡ A*B ⇒

C = n-by-k sized array

C_{ij} = ith row of A • jth column of B

$$C_{ij} \equiv \sum_{k=1}^{k=m} A_{ik} \bullet B_{kj}$$

```
>> E = [1 2 3; 5 7 9]
E = 1     2     3
    5     7     9

>> G = [6 11; 21 -1; 19 77]
G =
 6    11
21    -1
19    77

>> H = E*G
 H = 105   240
     348   741
```

Observe that matrix multiplication is *not* commutative $B \bullet A \neq A \bullet B$!

```
>> G*E     % G*E ≠ E*G
ans =
     61     89    117
     16     35     54
     404    577   750
```

1.3 FUNCTIONS AND FLOW CONTROL

1.3.1 M-Files

⇨ *Programs you create*

M-file is a *text* file, entitled say ZZZ.m, created with an .m extension and it contains a sequence of MATLAB commands. An m-file is equivalent to typing all the commands stored in the file.

Why use files of commands?

- It is easier to edit a file of commands rather than retyping a long list of commands.
- Makes a permanent record of your commands.
- Creates procedures that can be reused at a later date.

1.3.2 Scripts

Simply a list of MATLAB commands.

Works on variables currently present in the Workspace. Variables created in script execution *remain* in the Workspace after the script execution has ended. Scripts can *modify* preexisting variables in the Workspace.

1.3.3 Functions

Functions are similar to script files but functions possess one critical difference.

Variables created in a function exist only in the local function space and do *not overwrite Workspace variables!* Such behavior protects the workspace and permits development of complicated simulations without worrying about variable name clashes. Functions can exchange data with the workspace through the function's input/output arguments. After execution, only variables in the Workspace are the function output variables.

Note a function *must* start with function *declaration.*

```
function [out_arg1, out_arg2, …] = name_of_fun(in_arg1,in_arg2, …)
```

First line of a function begins with the keyword **`function`**:

- followed by output argument**s** (plural) in *square* brackets.[ ], where the variables are separated by commas,
- followed by the assignment symbol, **= ,**
- followed by the user-defined function name,
- followed by a comma-separate list of input arguments enclosed in parentheses, ().

You should place, *immediately* following the **`function`** declaration line, a contiguous section of comments detailing the properties of input/output arguments as well as a *short* description of the function's purpose. *Write such comments during function creation and not afterwards.* Such comments will be displayed on the command window, when one types (at MATLAB prompt) -

```
>> help function_name
```

The name of a function *should* be the name of .m file. Please note that functions are not required to have input and/or output arguments. Finally, the execution of a function terminates when the last line is executed OR when a **`return`** statement is encountered.

An example of a (simple) function is the following -

```
  function [mean, standard_dev] = stats(x)
% function [mean, standard_dev] = stats(x)
%  INPUT:
%          x             vector
%  OUTPUT:
%          mean          scalar
%          standard_dev  scalar

       n            = length(x);
       xb           = sum(x)/n;
       standard_dev = sqrt(sum((x - xb).^2)/(n - 1));
```

```
mean          = xb;
```

Examples of function calling sequences are:

```
>> [r, a, z] = motion(x, y, t);
>> [r, p, q] = motion(x, y, [0:1:100]);
>> [a, b, c] = motion(0.5, 12.0, 66.0);
```

1.3.4 Accessing Functions and Scripts

Create a new m-file: File Menu (Top row) → File tab → New → double click on Script *OR* Function.

Save an m-file: When in the editor window, go to File tab and click on Save (for a function) *OR* Save As xxxx for a script where xxxx is the name of the script.

Accessing an existing m-file: Click on 2nd ICON 2nd row → double click open folder → double click on appropriate file!

Executing an m-file: In the command window type the name of m-file *without* the .m extension *OR* in the editor itself double click the save and run icon, the green triangle on top of light blue square on 2nd row of bar.

1.3.5 Anonymous Functions ← *I use frequently!*

Writing an anonymous function is an *elegant* way to create *short* user-defined functions without creating m files!

```
f1 = @(x, y, …) expression
```

From the left:

`f1`	Name of the function,
`=`	Assignment operator,
`@`	symbol called function handle,
`(x,y, … )`	comma separated list of input arguments,
`expression`	body of the function, consisting of a *single* MATLAB expression.

```
>> f1 = @(x) x^2 – 1;
>> f1(5)
ans =  24
```

If you want to *input* vectors, more care is needed in the formulation of an anonymous function!

```
>> f1([4 5 6])
??? Error using ==> mpower
Inputs must be a scalar and a square matrix.

Error in ==> @(x)x^2-1    % Another unclear error message!
```

Create in *vector* format, employing the period notation:

```
>> f1 = @(x) x.^2 –1;
>> f1( [ 4 5 6])
ans =  15    24    35
```

Note that anonymous functions can depend on multiple variables:

```
>> g =   @(u, v) u.^2 + v.^2;
```

1.3.6 Passing Functions as Arguments to other Functions

You will need to pass functions to other functions in order to plot curves, solve integrations and ordinary differential equations, etc.

The primary way to do this is to use function handles, @.

Let's employ MATLAB intrinsic **ezplot** as example (**ezplot** - an easy to use *function* plotter) -

```
>> ezplot(sin)
??? Error using ==> sin
Not enough input arguments    ← opaque error message!
```

What happened here is that we need to discriminate names of *functions* from names of *variables*! The simplest way to do this is to use function handles!

```
>> ezplot(@sin)    % works
```

The *similar* behavior occurs with user-defined m files! Say we have created a simple m-file named **f3.m**, whose content is,

```
y = function f3(x)
y = exp(-x).*sin(x);
```

If we attempt to plot this function with **ezplot**,

```
>> ezplot(f3)
??? Input argument "x" is undefined.

Error in ==> f3 at 3
   y = sin(x).*exp(-x);
```

However, using a function handle, @, with **ezplot** works!

```
>> ezplot(@f3)   % works
```

Note however, when we pass anonymous functions to other functions we must remember that anonymous functions were already created with function handles, and, consequently, we do not need to include them again.

```
>> f1 = @(x) exp(-x).*sin(x);
>> ezplot(f1) % works
```

Since the MATLAB error messages are opaque, the simplest way to deal with passing functions to other functions would be to pass a function to the another function, first, with @ and, second, without @ and see which one works!

1.3.7 Relational Operators

```
==  equal              % Not single = which mean assignment
~=  not equal
<   less than
```

```
 >   greater than
<=   less than or equal
 >=  greater than or equal
```

Single = denotes assignment! Do not test for equality, using X = Y, which sets X equals the value of Y. The equality test for scalars is X == Y!

Result of relational operation --
1 if relation is TRUE
0 if relation is FALSE

Scalars:

```
>> pi >= 3.0
ans =  1
>> 0.0 > 1.0
ans =  0
```

Vectors:

```
>> x = [-1 1 1]; y = [1 2 -3];

>> x > 0
ans =  0     1     1

>> y > 0
ans =  1     1     0
```

However, comparisons between matrices (MUST be same size):

Outcome of a relational comparison produces same-sized matrix of 0s and 1s:

```
>> B = 2.0*ones(3);
>> A = [1 2 3; 4 5 6; 7 8 9]
A =
    1     2     3
    4     5     6
    7     8     9

>> A == B
ans =
    0     1     0
    0     0     0
    0     0     0

    >> A > B
ans =
    0     0     1
    1     1     1
    1     1     1
```

MATLAB possesses a number of relational *functions*: **isequal, isempty, isfinite, isnan,** etc. For example, to test equality of matrices use **isequal -**

```
>> isequal(A, B)
ans =     0
```

NOTE: Similar behavior occurs, when comparing matrices and scalars -

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

>> A == 7
ans =
      0     0     0
      0     0     1

>> isequal (A, 7)
      ans =  0
```

1.3.8 Logical Operators

Logical Operators	element-by-element	scalars (short-circuit)
and	&	&&
or	|	||
not	~	
all nonzero	all	
any nonzero	any	

Short-circuit somewhat *obscure concept* yet MATLAB's code analyzer cites it often. Consequently, I need to address the concept. Remember these operators, && and ||, work only with scalars! Note that

(a && b) returns 1 (true) if *both* a and b evaluate to 1 (true).

if a returns 0 (false), then short circuit means that

(a && b) returns 0 (false) *without* evaluating b.

Similarly,

(c || d) returns 1 (true) if *either* or *both* c and d evaluate to 1 (true).

if c returns 1 (true), then short circuit means that

(c || d) returns 1 (true) *without* evaluating d .

Short circuiting is described here because the MATLAB's code analyzer cites it often as a warning, you do not need to use it.

Relational operations

```
>> x = [-1 1 1]; y = [1 2 -3];

>> x > 0 & y > 0
ans = 0     1     0

>> x > 0 | y > 0
ans = 1     1     1
```

Logical functions

`all` TRUE if all elements are ***nonzero*** (don't have to be 1s).
`any` TRUE if any element are ***nonzero*** (don't have to be 1s).

```
>> x = [-1 1 1]; y = [1 2 -3];
>> any(x)
ans = 1
>> all(y)
ans =  1
```


all(X) operates on the *columns* of X, returning a *row* vector of 1's and 0's.
any(X) operates on the columns of X, returning a row vector of 1's and 0's.

1.3.9 FIND ← matrix/vector function I use most frequently!

A *very* useful function is **find**, which returns the indices of the *nonzero* elements of a matrix.

```
>> x = [-3 1 0 -inf 0]
>> find(x == -inf)
ans =  4

>> f = find(x)
f = 1     2     4

>> x(f)
ans =  -3     1  -Inf

>> x(find(x < 0))
ans =  -3  -Inf

>> A = [4 2 16; 12 4 3]
A =     4     2    16
       12     4     3

>> B = [12 3 1; 10 -1 7]
B =    12     3     1
       10    -1     7

>> A < B
ans =  1     1     0
       0     0     1
```

Say that you want to set the elements above of A, which are less than the corresponding elements of B, equal to -9, using **find** -

```
>>  A( find(A < B) ) = -9
A =    -9    -9    16
       12     4    -9
```

1.3.10 if-end Constructs ← if constructs work best with scalars

IF

```
if cond
  commands
end
```

IF-ELSE

```
if cond
  commands 1
else
  commands 2
end
```

IF-ELSEIF-ELSE

```
if cond 1
  commands 1
elseif cond 2
  commands 2
else
  commands 3
end
```

Logical conditions *usually* are variables coupled by relational operators (x < 2.0) or logical functions, `all(x < 0).`

Can be written on a single line –

```
if  x >= 0.0; y = sqrt(x); end
```

Function example:

```
function [mean, standard_dev] = stats(x)
     [r, c]       = size(x);
     % error if x is an array
     if r == 1 | c == 1;
        % c == 1 x => column vector
        % r == 1 x => row vector
        n            = length(x);
        xb           = sum(x)/n;
        standard_dev = sqrt(sum((x - xb).^2)/(n - 1));
        mean         = xb;
     else
        disp('You input an array & not a vector');
     end
```

Beware with using matrices with if constructs!

With an array, A, if-end acts the same as if all(all(A)) end.

If you testing individual elements of matrices DO NOT use the if-else-end construct! USE find(A)!

1.3.2 for loops

Used when you know ahead of time the number of iterations needed.

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

expression of the form → lower:step:upper (a row vector).
NOTE: step can be a negative
NOTE: if step is omitted, it is assumed to be 1.

```
% Sum of first 25 terms of series 1/k
```

```
>> s = 0.0; % NOTE: must initialize s to 0.0
>>  for k = 1:25; s = s + 1.0/k; end
>> s
   s = 3.8160
```

1.3.13 while loops

Used when know the number of iterations is not known prior to execution.

```
while logical_expression
     statements
end
```

NOTE: statements executed as long as **`logical_expression`** is **true**

It is not obvious before the calculation below how many time you must divide x by 2 -

```
>> x = 1;
>> while abs(x) > 0
      xmin = x;
      x    = x/2;
   end
>> xmin
   xmin = 4.9407e-324
>> realmin*eps    % xmin is identical to following
ans =  4.9407e-324
```

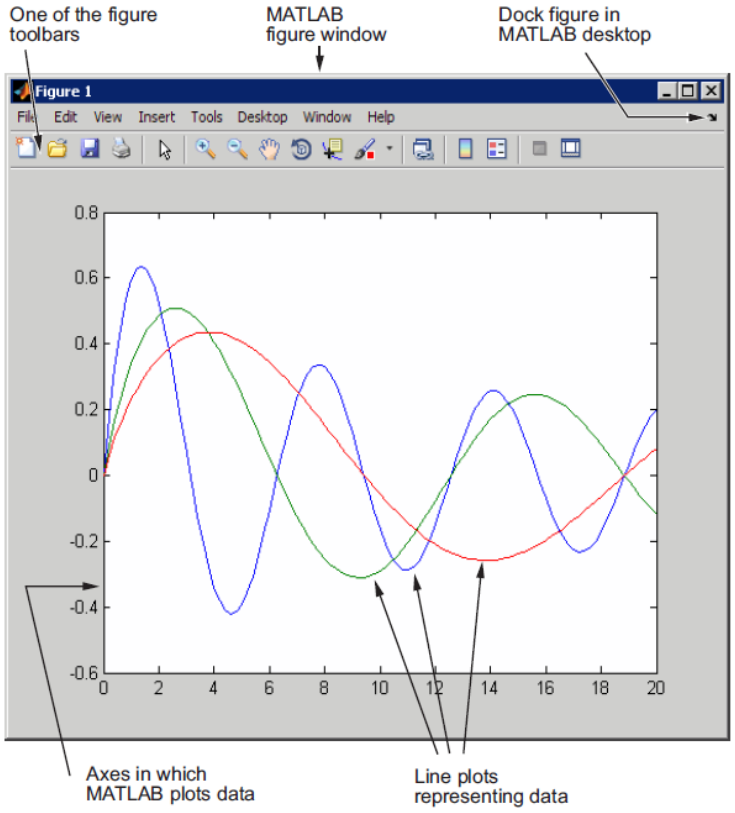
Good programming practice suggests that you limit the number of repeats to avoid possibility of *infinite* repetitions. The most common way to do this to use the **`break`** construct -

```
>> n    = 0;       % initialize counter
>> ntot = 10000;
>> x = 1;
>>   while abs(x) > 0
       xmin = x;
       n    = n + 1; % update counter at each repetition
       x    = x/2;
       if n > ntot; break; end
     end
>> xmin
   xmin = 4.9407e-324
>> n
n = 1075   % check counter
```

1.4 PLOTS

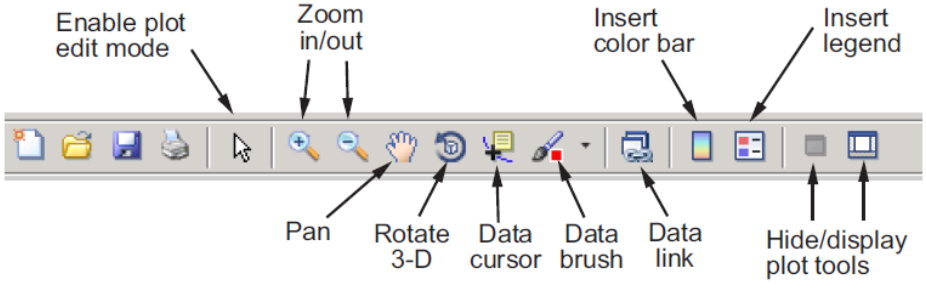
1.4.1 Structure of a MATLAB Plot

The plot below illustrates the various components of a MATLAB figure.



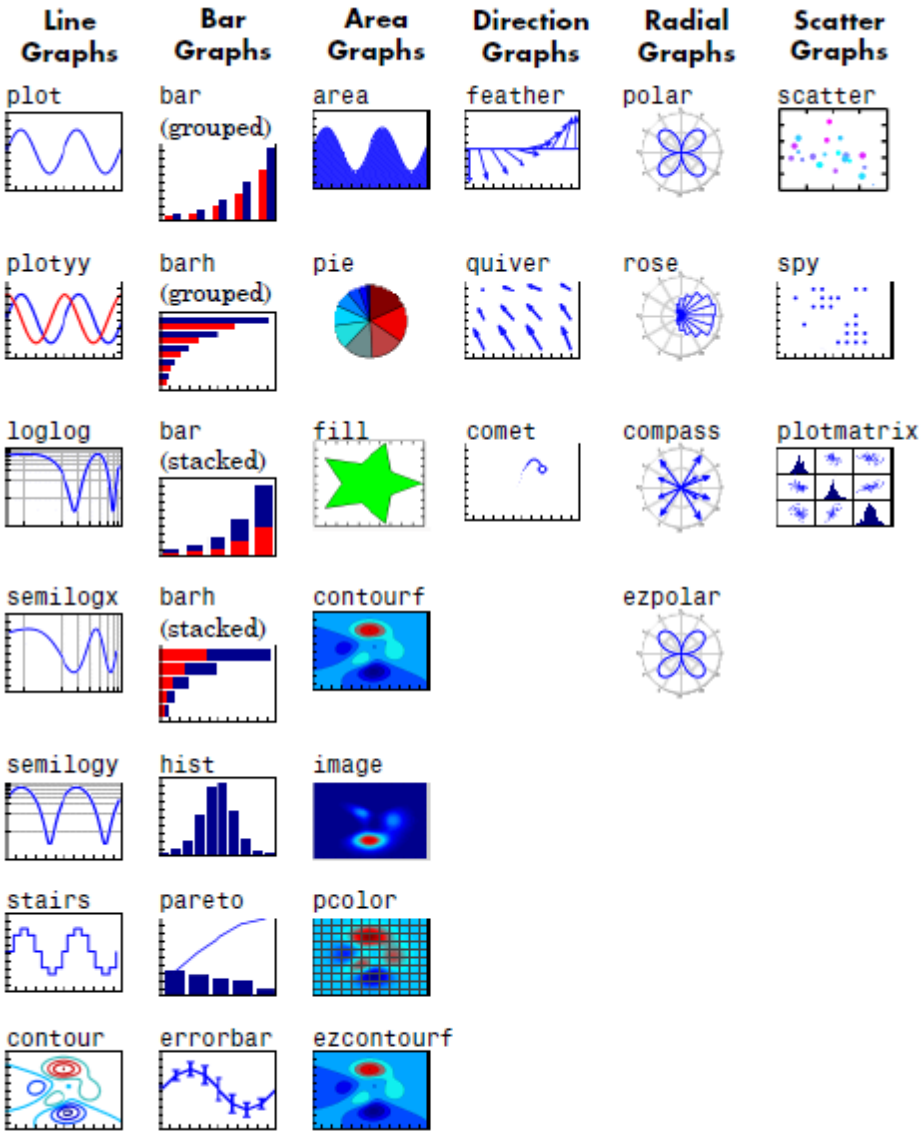
[p. 1-4, graphg.pdf]

Observe that colors delineate different curves. Alternatively, different line types, dots, dashes, etc. can be used instead of colors. By default MATLAB employs *colors* to discriminate data sets in plots. The sequence of the colors of the different-colored line plots can be changed by the user. Also text can be added to your figure. The easiest way to modify a plot is to activate the Plot Tool icon (the last icon on the right – the white square with double-lined border).



[p. 1-5, graphp.pdf]

Note that MATLAB has a wide range of 2D plot functions as can be seen by the table on the following page which tabulates *all* the available 2D MATLAB plot functions as of MATLAB R2010b.



[p 1-7, graphp.pdf]

Due to their significance in displaying data sets, we will investigate primarily line graphs (the first column) in this course.

1.4.2 Line Types and Data Symbols Designated by Text Commands

The many properties of 2D plot functions can be controlled (or changed) by sometimes esoteric, arcane text commands. In the following we will investigate some of the most basic text commands.

For example, replace **`plot(x, y)`** with **`plot(x, y, string)`** to gain more control of curve markers, colors and line type. *string* combines up to *three* elements, which control marker, line style, and color of lines and markers. NOTE: string components can appear in *ANY* order.

LINE STYLES:

-	solid line (default)
--	dashed line
:	dotted line
-.	dot-dashed line

COLORS:

r	red
g	green
b	blue
c	cyan
m	magenta
y	yellow
k	black (default)
w	white

MARKERS:

o	circle
*	asterisk
.	point
+	plus
x	cross
s	square
d	diamond
^	upward triangle
v	downward triangle
>	right triangle
<	left triangle
p	five-point star
h	six-point star

```
>> plot(x, y, 'r*--')  % red asterisks placed at each point and
                          points joined by red dashed lines

>> plot(x, y, 'y+')  % specifies yellow cross markers with
                       with no lines joining points

>> plot(x, y, 'kd:') % back dotted line with diamond marker
```

1.4.3 Multiple Curves

As I will demonstrate the term, *multiple curves*, covers many possible options. I will employ the functions to create the first set of multiple plots -

```
>> f = @(x) cos(10*x).*exp(-x.^2);
>> g = @(x) sin(4*x).*cos(8*x);
>> h = @(x) sin(x).*cos(x).*exp(-0.5*x);
>> x = linspace(0,pi, 100);
```

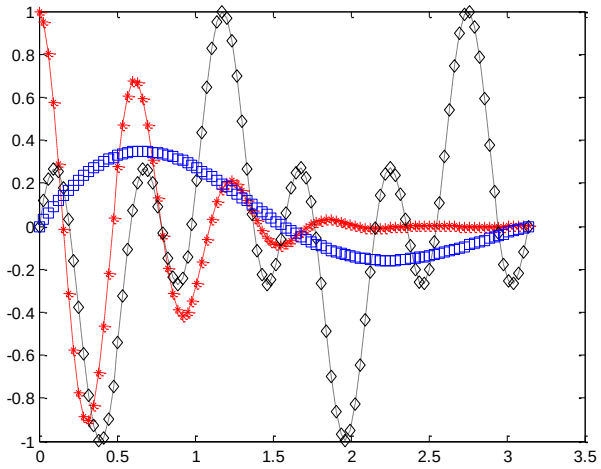
To create multiple plots on the desktop employ the **`figure(n)`** command where n is an integer:

```
>> figure(1); plot(x, f(x), 'r*--');
```

```
>> figure(2); plot(x, g(x),'kd:');
>> figure(3); plot(x, h(x),'bs');
% This graphic output has not been copied to this document.
```

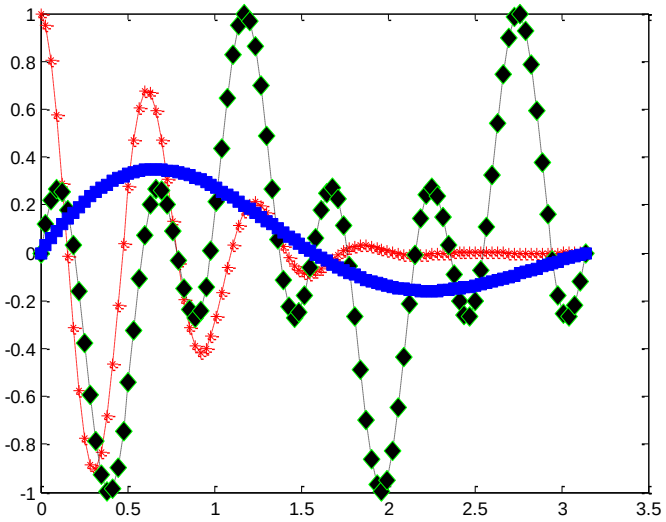
Observe that you can illustrate several curves in a single plot with a *single* command, e. g.

```
>> plot(x, f(x), 'r*--', x, g(x),'kd:', x, h(x),'bs');
```



An alternative *set* of commands, using **`hold on/hold off`**, can create a *similar* figure. One employs such multiple commands to modify separately the properties of the individual curves -

```
>> hold on;
>> plot(x, f(x), 'r*--');
>> plot(x, g(x),'kd:','MarkerSize',10, 'MarkerFaceColor', …
       'k','MarkerEdgeColor', 'g');
>> plot(x, h(x),'bs',  'MarkerFaceColor', 'b');
```



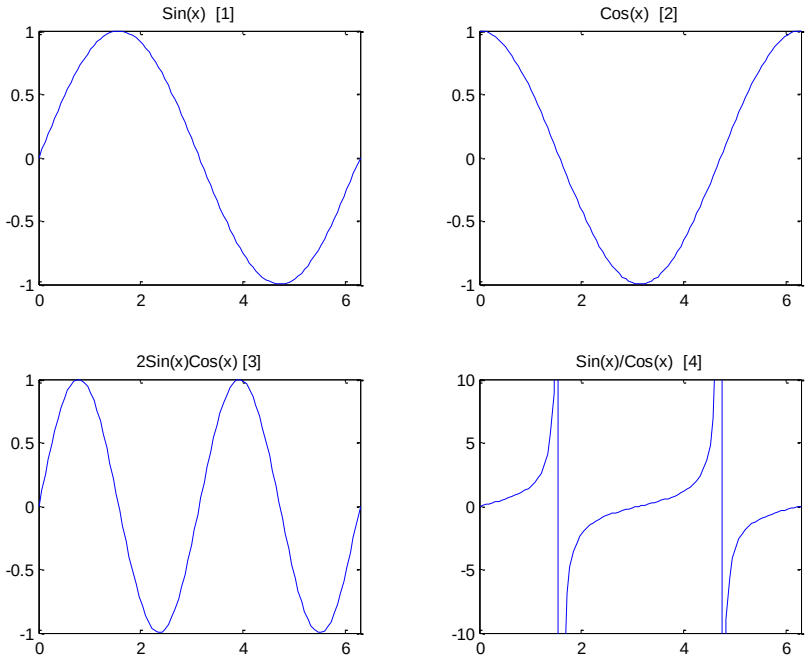
See the following documentation for the *complete* list of plot properties which can be changed –

```
>> doc line_props
```

Observe that one can plot multiple plots on a *single* page via command-line function, **`subplot:`**

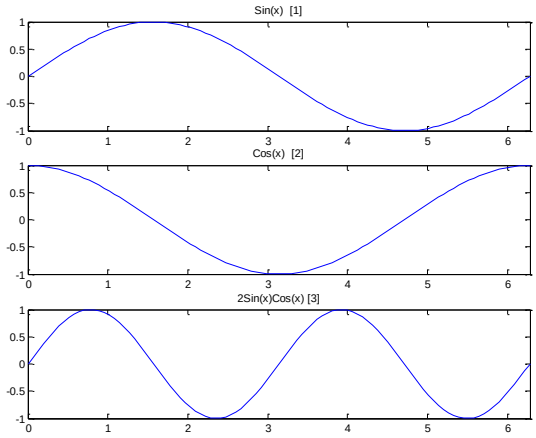
`subplot(m, n, p)` divides the current figure window into an m-by-n array of subplots and treats the p^{th} subplot as active. NOTE: the active plots are numbered in *row-major* fashion from left to right starting with the top row, then left to right on the second row and so on. NOTE: when a chosen subplot is active, relevant commands such as axis applies only to the chosen subplot.

```
>> x = linspace(0.0, 2.0*pi);
>> y = sin(x); z = cos(x);
>> a = 2.0*sin(x).*cos(x); b = sin(x)./(cos(x) + eps);
>> subplot(2,2,1); % Upper left of a 2 by 2 grid
>> plot(x,y); axis([0.0 2*pi -1 1]); title('Sin(x)  [1]');
>> subplot(2,2,2); % Upper right of a 2 by 2 grid
>> plot(x,z); axis([0.0 2*pi -1 1]); title('Cos(x)  [2]');
>> subplot(2,2,3); % Lower left of a 2 by 2 grid
>> plot(x,a); axis([0.0 2*pi -1 1]); title('2Sin(x)Cos(x) [3]');
>> subplot(2,2,4); % Lower right of a 2 by 2 grid
>> plot(x,b); axis([0.0 2*pi -10 10]);
    title('Sin(x)/Cos(x)  [4]');
```



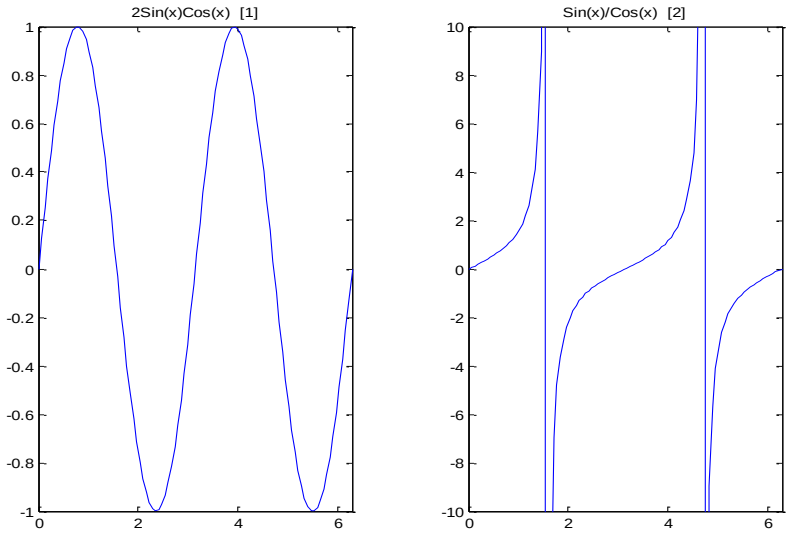
Vertical Subplots:

```
>> x = linspace(0.0, 2.0*pi);
>> y = sin(x); z = cos(x);  a = 2.0*sin(x).*cos(x);
>> b = sin(x)./(cos(x) + eps);
>> subplot(3,1,1); % Upper
>> plot(x,y); axis([0.0 2*pi -1 1]); title('Sin(x)  [1]');
>> subplot(3,1,2); % mid
>> plot(x,z); axis([0.0 2*pi -1 1]); title('Cos(x)  [2]');
>> subplot(3,1,3); % Lower
>> plot(x,a); axis([0.0 2*pi -1 1]);
>> title('2Sin(x)Cos(x) [3]');
```



Horizontal Subplots:

```
>> x = linspace(0.0, 2.0*pi); a = 2.0*sin(x).*cos(x);
>> b = sin(x)./(cos(x) + eps);
>> subplot(1,2,1); % Left
>> plot(x,a); axis([0.0 2*pi -1 1]);
>> title('2Sin(x)Cos(x)  [1]');
>> subplot(1,2,2); % Right
>> plot(x,b); axis([0.0 2*pi -10 10]);
>> title('Sin(x)/Cos(x)  [2]');
```



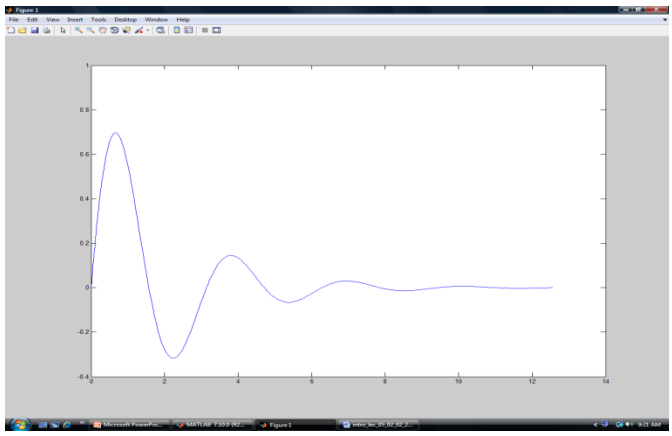
1.4.4 *Interactive* Plotting

This is the preferred approach in creating a plot, at least when a few plot are needed to be created. Such an approach minimizes the number functions you need to remember. Create some vectors in the command window:

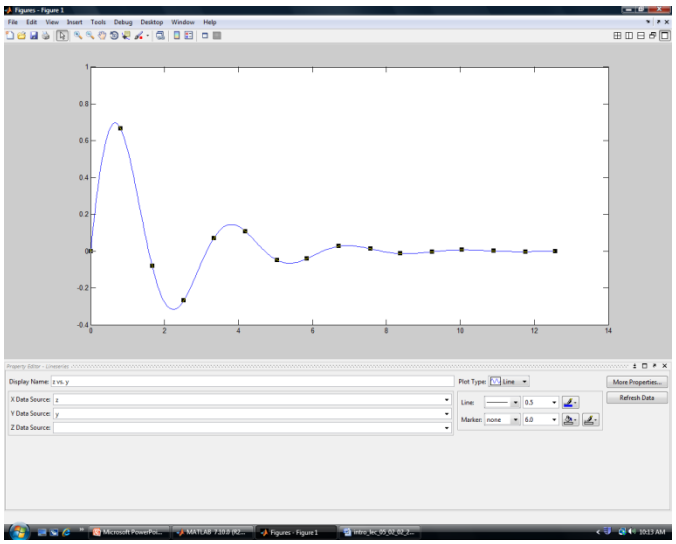
```
>> f = @(x) sin(2*x).*exp(-0.5*x);
>> z = linspace(0, 4*pi, 250);
>> y = f(z);
>> g = @(x)sin(3*x).*cos(6*x).*exp(-0.5*x.^2);
```

```
>> w = g(z);
```

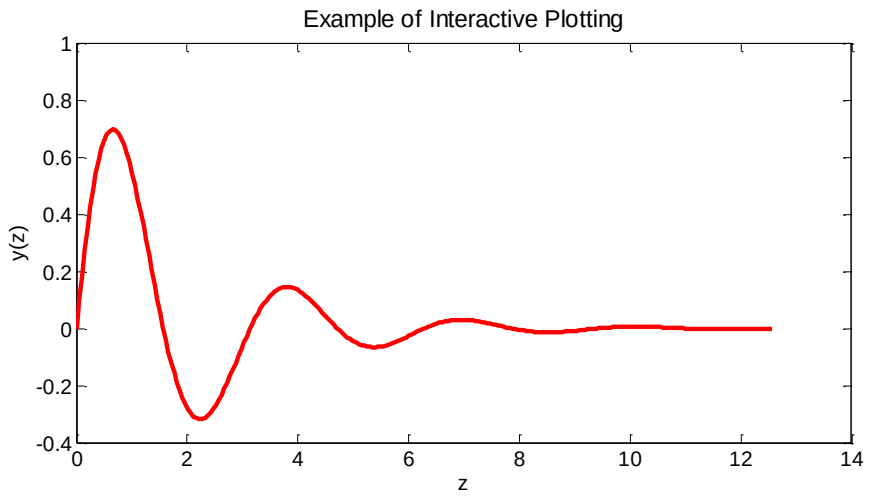
We want to plot y(z). *Left Shift* click *first* on z, then left shift click *second* on y in *Workspace.* To *right* of the graph icon (white rectangle - *blue* curve) now appears *plot(z, y)* . Please verify order of plot variables is (z, y) NOT (y, z). Click on graph icon – following figure window appears



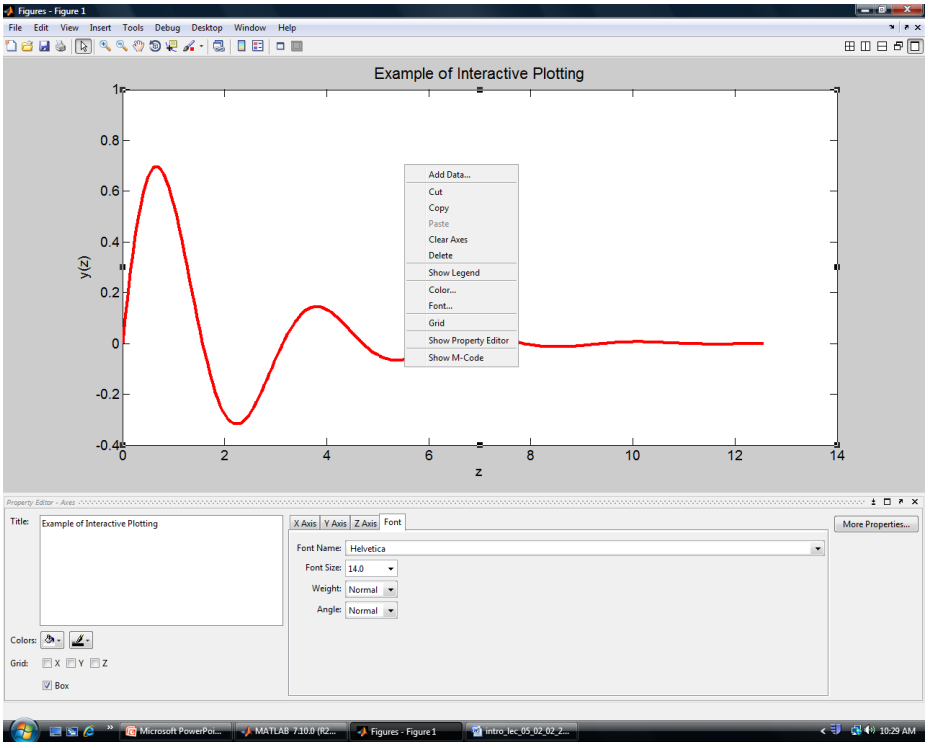
Under Tools tab click Edit Plot. Now small solid black squares appear at the corners as well as at the centers of the sides of the above figure window. Also a check mark symbol appears next to *Edit Plot* under the *Tools* tab. Now double click on the curve to activate editing process and the property editor below appears below the figure window -



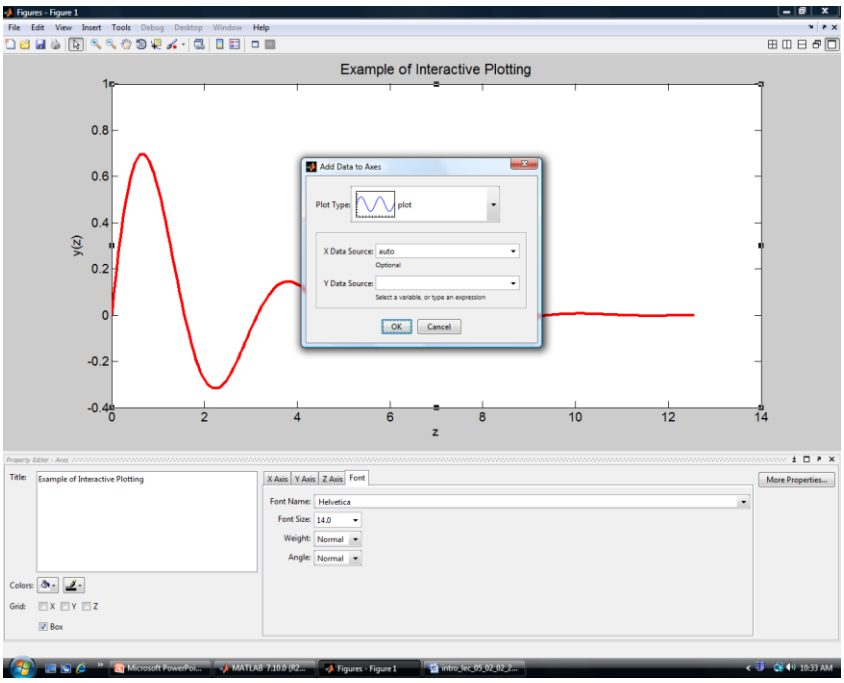
Using the dialog boxes change: plot type, line thickness, color. Then click on either axis, which activates the axes boxes, and add axes labels, and change fonts and font size, Copy Figure (under Edit tab) and paste into the Word document -



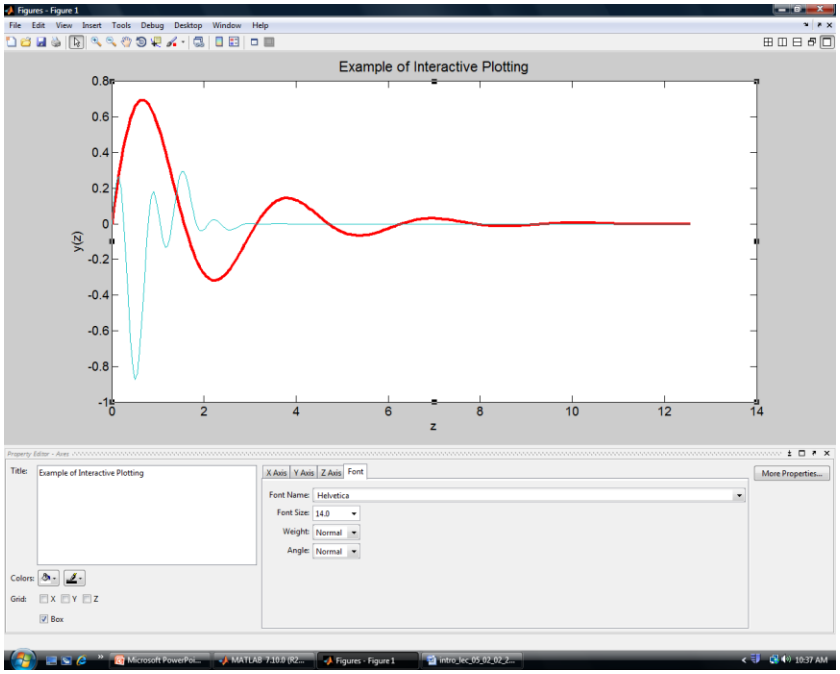
Now let us add another curve to this plot *interactively.* We will add w(z). Start with above the plot in edit mode. Now click on an axis and right click anywhere and the tab starting with *Add Data ..now* appears -



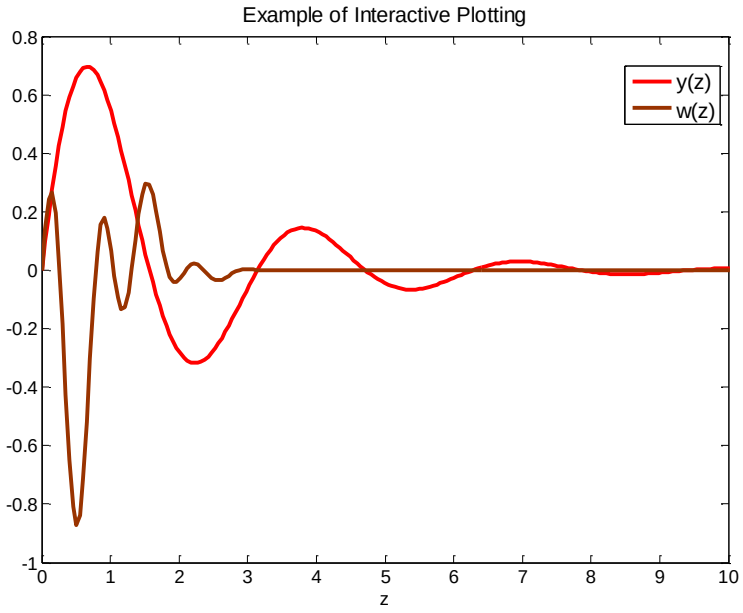
Click on *Add Data …* Tab below appears and in X Data Source scroll to z and in the Y Data Source scroll to w and click OK -



An additional curve (default color green), representing w(z), appears (pasted below),



Thicken green curve (3.0)
Color the curve *brown*
Remove y-axis label
Trim x axis to [0, 10]
On Insert tab → click Legend → edit legend text
Copy Figure (under Edit Tab)
Paste into Word -



Now let us create a *new* plot. Plot the following parametric curve in three dimensions (x(t), y(t), z(t)).

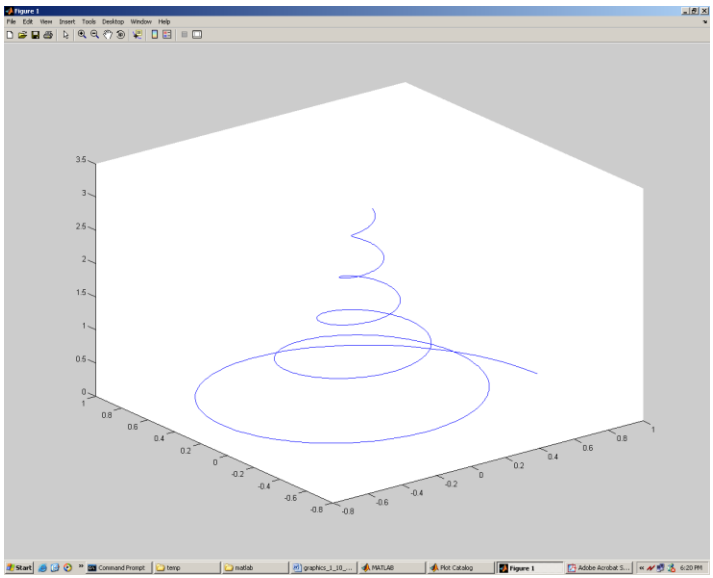
```
>> clear;
>> t = 0:0.05:10*pi;
>> y = sin(t).*exp(-t/10);
>> x = cos(t).*exp(-t/10);
>> z = t/10;
```

Shift click on x, y, z in Workspace

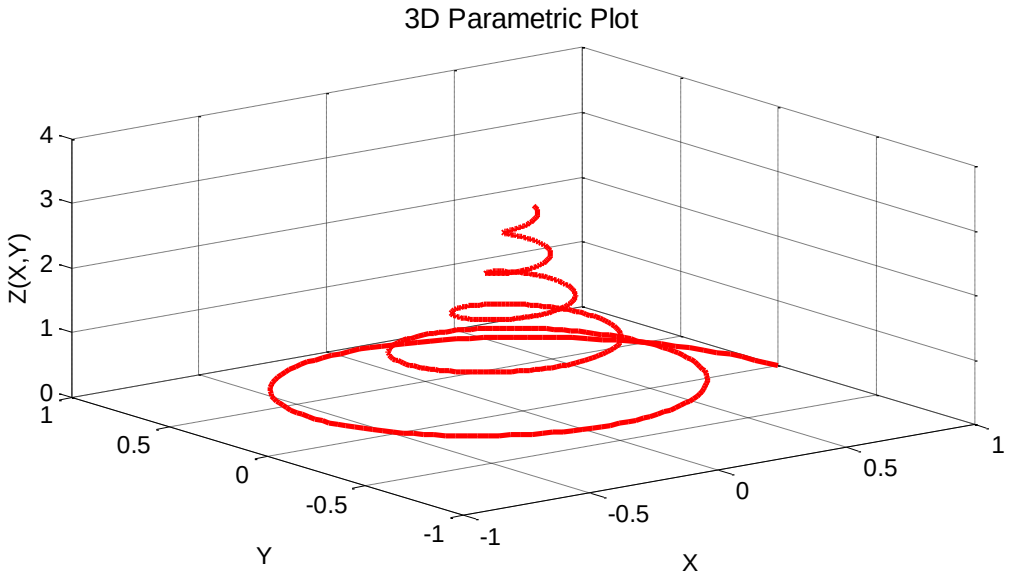
To right of plot - *plot as multiple series* appears

Not correct plot type – activate pull down menu

Double click on plot3(x, y, z) panel icon
Following appears!



Thickened the line.
Colored curve *red*.
Added X, Y, Z grids.
Increased font size.
Labeled axes.
Created title,
Copy Figure (under Edit tab).
Pasted into Word.



1.4.5 Three-Dimensional Plots

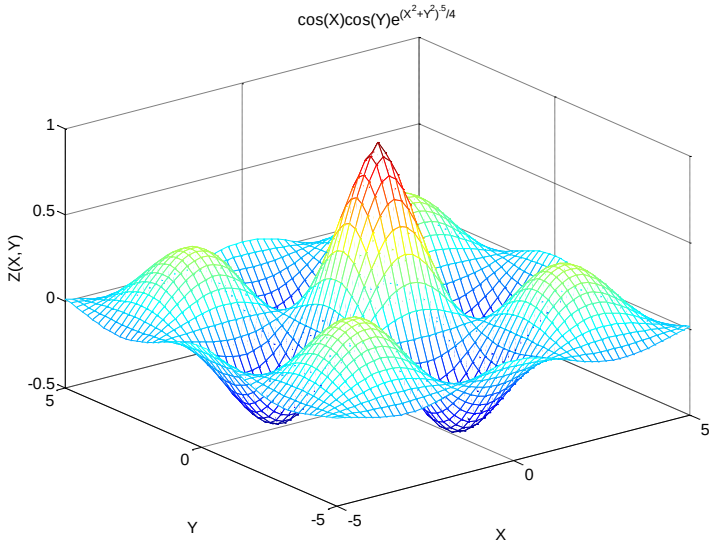
Three-dimensional plots from f(x, y) are more difficult to construct than two-dimensional plots. First one must generate a *uniform* grid of points in the x-y plane where the function, f(x ,y), must be evaluated to construct a mesh in three dimensions. The simplest way to do this is to use the built-in function, **`meshgrid`**. ([p.

MESHGRID X and Y arrays for 3-D plots.
`[X,Y] = meshgrid(x,y)` transforms the domain specified by vectors x and y into arrays X and Y that can be used for the evaluation of functions of two variables and 3-D surface plots. The rows of the output array X are copies of the vector x and the columns of the output array Y are copies of the vector y.

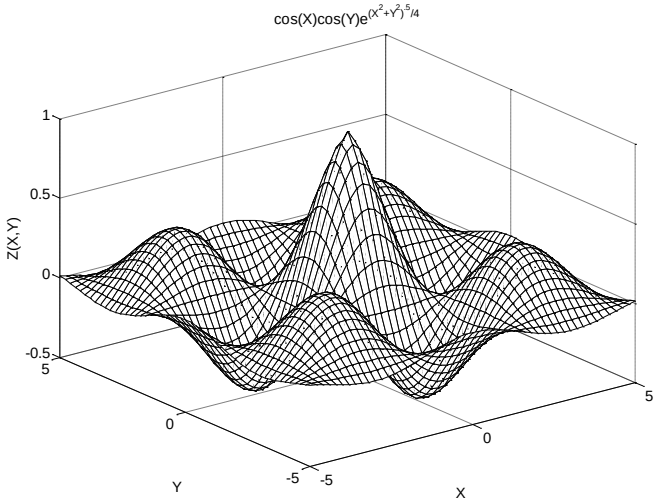
```
>> x = -1:0.5:1;
>> y =  0:1:3;
>> [X, Y] = meshgrid(x, y)
X =-1.0000   -0.5000         0    0.5000    1.0000
   -1.0000   -0.5000         0    0.5000    1.0000
   -1.0000   -0.5000         0    0.5000    1.0000
   -1.0000   -0.5000         0    0.5000    1.0000
Y =  0     0     0     0     0
         1     1     1     1     1
         2     2     2     2     2
         3     3     3     3     3
```

`mesh(X, Y, Z, C)` plots the *colored* parametric mesh defined by four matrix arguments.

```
>> u = -5:.2:5;
>> [X, Y] = meshgrid(u, u);
>> Z = @(x, y) cos(x).*cos(y).*exp(-sqrt(x.^2+y.^2)/4);
>> mesh(X, Y, Z(X, Y))
>> title('cos(X)cos(Y)e^{(X^2+Y^2)^{.5}/4}')
>> xlabel('X'); ylabel('Y'); zlabel( 'Z(X,Y)');
```



Note the effect of employing `colormap([ 0 0 0])` - `colormap([0 0 0])`



1.4.6 How to Save a Figure

name_of_figure2.emf for insert into Word

name_of_figure3.jpg for insert into Word

To insert the figure in a Word document:

- [Figure 1] File ⇒Save ⇒ Save as ⇒ plot3.jpg [Saved in jpg format.]
- [Word] Insert ⇒ Picture ⇒ From File … {plot3d.jpg}
- Tried other file formats: emf, eps, pdf, & tif
- Storage varies from plot.eps (5.3 M) to plot.pdf (13 KB).

1.5 REFERENCES

Driscoll. T. A. 2009 *Learning MATLAB* (SIAM: Philadelphia).

Goldberg, G. 1991 *ACM Computing Surveys*, **23**, 5

Heath, M. T. 2002 *Scientific Computing: An Introductory Survey*, 2nd Ed. (McGraw-Hill: NY).

Higham, N. J. 2002 *Accuracy and Stability of Numerical Algorithms* (SIAM: Philadelphia).

Higham, D. J., & Higham N. J. 2005 *MATLAB Guide 2nd Ed.* (SIAM: Philadelphia).

Hunt, B. R. et al. *A Guide to MATLAB for Beginners and Experienced Users*, 2001 (Cambridge University Press; Cambridge, UK).

Kahaner, D, Moler, C., & Nash, S. 1989 *Numerical Methods and Software* (Prentice-Hall: Englewood, Cliffs, NJ).

Kelley, C. T. 2003 *Solving Nonlinear Equations with Newton's Method* (SIAM: Philadelphia).

Scepanovic, D. 2010, *Introduction to MATLAB, 6.094*, http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-094-introduction-to-matlab-january-iap-2010

Ueberhuber, C. W. 1997 *Numerical Computation 1: Methods, Software, and Analysis* (Springer-Verlag: Berlin).