MATLAB BASICS

Now that you've installed MATLAB, it's time to see what it can do. In this tutorial you will be shown some of its capabilities; to show all of what MATLAB can do would simply take too much time. As you follow this tutorial, you will begin to see the power of MATLAB to solve a wide variety of problems important to you. You may find it beneficial to go through this tutorial while running MATLAB. In doing so, you will be able to enter the MATLAB statements as described, confirm the results presented, and develop a hands-on understanding of MATLAB.

Perhaps the easiest way to visualize MATLAB is to think of it as a full-featured calculator. Like a basic calculator, it does simple math such as addition, subtraction, multiplication, and division. Like a scientific calculator, it handles complex numbers, square roots and powers, logarithms, and trigonometric operations such as sine, cosine, and tangent. Like a programmable calculator, it can be used to store and retrieve data; you can create, execute, and save sequences of commands to automate the computation of important equations; you can make logical comparisons and control the order in which commands are executed. Like the most powerful calculators available, it allows you to plot data in a wide variety of ways, perform matrix algebra, manipulate polynomials, integrate functions, manipulate equations symbolically, and so on.

In reality, MATLAB offers many more features and is more multifaceted than any calculator. MATLAB is a tool for making mathematical calculations. It is a user-friendly programming language with features more advanced and much easier to use than computer languages such as BASIC, Pascal, or C. It provides a rich environment for data visualization through its powerful graphics capabilities. MATLAB is an application development platform, where you can create graphical user interfaces (GUIs) that offer a visual approach to solving specific problems. In addition, MATLAB offers sets of problem-solving tools for specific application areas, called ***Toolboxes***. For example, this Student Edition of MATLAB includes the *Control System Toolbox,* the *Signal Processing Toolbox,* and the *Symbolic Math Toolbox*. In addition, you can create toolboxes of your own.

Because of the vast power of MATLAB, it is important to start with the basics. That is, rather than taking in everything at once and hoping that you understand some of it, in the beginning it is helpful to think of MATLAB as a calculator. First, as a basic calculator; next, as a scientific calculator; then, as a programmable calculator; then, finally, as a top-of-the-line calculator. By using this calculator analogy, you will see the ease with which MATLAB solves everyday problems, and will begin to see how MATLAB can be used to solve complex problems in a flexible, straightforward manner.

Depending on your background, you may find parts of this tutorial boring, or some of it may be over your head. In either case, find a point in the tutorial where you're comfortable, start up MATLAB, and begin. To assist you while learning, the following conventions are used throughout this text:

Bold	Important terms and facts
Bold italics	New terms
Bold initial caps	Keyboard key names, menu names, and menu items
`Constant width`	User input, function and file names, commands, and screen displays
Italics	Window names, book titles, toolbox names, company names, example text, and mathematical notation

Running MATLAB creates one or more windows on your monitor. Of these, the *Command* window is the primary place where you interact with MATLAB. This window has an appearance as shown below. The character string `EDU»` is the MATLAB prompt in the Student Edition. In other versions of MATLAB, the prompt is simply ». When the *Command* window is active, a cursor (most likely blinking) should appear to the right of the prompt, as shown in the figure. This

Figure 1.1 PC Command Window

Figure 1.2 Mac Initial Command Window

cursor and the MATLAB prompt signify that MATLAB is waiting to answer a mathematical question.

1.1 Simple Math

Just like a calculator, MATLAB can do simple math. Consider the following simple example: Mary goes to the office-supply store and buys 4 erasers at 25 cents each, 6 memo pads at 52 cents each, and 2 rolls of tape at 99 cents each. How many items did Mary buy, and how much did they cost?

To solve this using your calculator, you enter:

$$4 + 6 + 2 = 12 \text{ items} \qquad 4 \cdot 25 + 6 \cdot 52 + 2 \cdot 99 = 610 \text{ cents}$$

In MATLAB, this can be solved in a number of different ways. First, the above-mentioned calculator approach can be taken:

```
EDU» 4+6+2
ans =
    12

EDU» 4*25 + 6*52 + 2*99
ans =
   610
```

Note that MATLAB doesn't care about spaces, for the most part, and that multiplication takes precedence over addition. Note also that MATLAB calls the result `ans` (short for *answer*) for both computations.

As an alternative, the above problem can be solved by storing information in ***MATLAB variables***:

```
EDU» erasers=4
erasers =
     4

EDU» pads=6
pads =
     6

EDU» tape=2;
```

```
EDU» items=erasers+pads+tape
items =
    12
EDU» cost=erasers*25+pads*52+tape*99
cost =
   610
```

Here we created three MATLAB variables—erasers, pads, and tape—to store the number of each items. After entering each statement, MATLAB displayed the results, except in the case of tape. The semicolon at the end of the line EDU» tape=2; tells MATLAB to evaluate the line but not tell us the answer. Finally, rather than calling the results ans, we told MATLAB to call the number of items purchased items and the total price paid cost. At each step, MATLAB remembered past information. Because MATLAB remembers things, let's ask what the average cost per item was.

```
EDU» average_cost=cost/items
average_cost =
   50.8333
```

Because *average cost* is two words and MATLAB variable names must be one word, the underscore was used to create the single MATLAB variable average_cost.

In addition to addition and multiplication, MATLAB offers the following basic arithmetic operations:

Operation	Symbol	Example
addition, $a + b$	+	5+3
subtraction, $a - b$	−	23−12
multiplication, $a \cdot b$	*	3.14*0.85
division, $a \div b$	/ or \	56/8 = 8\56
exponentiation, a^b	^	5^2

The order in which these operations are evaluated in a given expression is given by the usual rules of precedence, which can be summarized as follows: **Expressions are evaluated from left to right, with the exponentiation operation having the highest order of precedence, followed by both multiplication and division having equal precedence, followed by both**

addition and subtraction having equal precedence. Parentheses can be used to alter this usual ordering, in which case evaluation initiates within the innermost parentheses and proceeds outward.

1.2 The MATLAB Workspace

As you work in the *Command* window, MATLAB remembers the commands you enter as well as the values of any variables you create. These commands and variables are said to reside in the ***MATLAB Workspace,*** and may be recalled whenever you wish. For example, to check the value of `tape`, all you have to do is ask MATLAB for it by entering its name at the prompt:

```
EDU» tape
tape =
     2
```

If you can't remember the name of a variable, you can ask MATLAB for a list of the variables it knows by using the MATLAB command `who`:

```
EDU»who
Your variables are:

ans            cost        items      tape
average_cost   erasers     pads
```

Note that MATLAB doesn't tell you the value of all the variables; it merely gives you their names. To find their values, you must enter their names at the MATLAB prompt. Just like a calculator, there's only so much room to store variables.

To recall previous commands, MATLAB uses the **Cursor** keys (←→↑↓) on your keyboard. For example, pressing the ↑ key once recalls the most recent command to the MATLAB prompt. Repeated pressing scrolls back through prior commands one at a time. In a similar manner, pressing the ↓ key scrolls forward through commands. Moreover, entering the first few characters of a known previous command at the prompt and then pressing the ↑ key immediately recalls the most recent command having those initial characters. At any time, the ← and → keys can be used to move the cursor within the command at the MATLAB prompt. In this manner, the command can be edited. Alternatively, the **Mouse** can be used along with the **Clipboard** to cut, copy, paste, and edit the text at the command prompt.

1.3 About Variables

Like any other computer language, MATLAB has rules about variable names. Earlier it was noted that variable names must be a single word containing no spaces. More specifically, MATLAB variable-naming rules are:

Variable Naming Rules	Comments/Examples
Variable names are case sensitive.	Items, items, itEms, and ITEMS are all different MATLAB variables
Variable names can contain up to 31 characters, and characters beyond the thirty-first are ignored.	howaboutthisvariablename
Variable names must start with a letter, followed by any number of letters, digits, or underscores. Punctuation characters are not allowed, since many of them have special meaning to MATLAB.	how_about_this_variable_name X51483 a_b_c_d_e

In addition to these naming rules, MATLAB has several special variables. They are:

Special Variables	Value
ans	The default variable name used for results
pi	The ratio of the circumference of a circle to its diameter
eps	The smallest number such that, when added to one, creates a number greater than one on the computer
flops	Count of floating point operations
inf	Stands for infinity, e.g., 1/0
NaN (or) nan	Stands for Not-a-Number, e.g., 0/0
i (and) j	$i=j=\sqrt{-1}$
nargin	Number of function input arguments used
nargout	Number of function output arguments used
realmin	The smallest usable positive real number
realmax	The largest usable positive real number

As you create variables in MATLAB, there may be instances where you wish to redefine one or more variables. For example:

```
EDU» erasers=4;

EDU» pads=6;

EDU» tape=2;

EDU» items=erasers+pads+tape
items =
     12

EDU» erasers=6
erasers =
      6

EDU» items
items =
     12
```

Here, using the first example again, we found the number of items Mary purchased. Afterward, we changed the number of erasers to 6, overwriting its prior value of 4. In doing so, the value of `items` has not changed. Unlike a spreadsheet, MATLAB **does not** recalculate the number of items based on the new value of `erasers`. **When MATLAB performs a calculation, it does so using the values it knows at the time the requested command is evaluated.** In the above-mentioned example, if you wish to recalculate the number of items, the total cost, and the average cost, it is necessary to recall the appropriate MATLAB commands and ask MATLAB to evaluate them again.

The special variables given above follow this guideline also. When you start MATLAB, they have the values given above; if you change their values, the original special values are lost until you clear the variables or restart MATLAB. With this in mind, avoid redefining special variables unless absolutely necessary.

Variables in the MATLAB workspace can be unconditionally deleted by using the command `clear`. For example:

```
EDU» clear erasers
```

deletes just the variable `erasers`.

```
EDU» clear cost items
```

deletes both `cost` and `items`.

```
EDU» clear cl*
```

uses the wildcard * to delete all variables that start with the letters `cl`.

```
EDU» clear
```

deletes all variables in the workspace! You are not asked to confirm this command. All variables are cleared and cannot be retrieved!

Needless to say, the `clear` command is dangerous and should be used with caution. Thankfully, there is seldom a need to clear variables from the workspace.

1.4 Comments and Punctuation

All text after a percent sign (%) is taken as a comment statement, e.g.:

```
EDU» erasers=4  % Number of erasers.
erasers =
     4
```

The variable `erasers` is given the value 4, and MATLAB simply ignores the percent sign and all text following it. This feature makes it easy to document what you are doing.

Multiple commands can be placed on one line if they are separated by commas or semicolons, e.g.:

```
EDU» erasers=4, pads=6; tape=2
erasers =
     4
tape =
     2
```

Commas tell MATLAB to display results; **semicolons suppress printing**.

```
EDU» average_cost=cost/...
  items
average_cost =
   50.8333
```

As shown above, a succession of three periods tells MATLAB that the rest of a statement appears on the next line. Statement continuation, as shown above, works if the succession of three periods occurs between variable names or operators. That is, a variable name cannot be split between two lines:

```
EDU» average_cost=cost/it...
ems
??? age_cost=cost/items
                       |
Missing operator, comma, or semicolon.
```

Likewise, comment statements cannot be continued:

```
EDU» % Comments cannot be continued ...
EDU» either
??? Undefined function or variable either.
```

You can interrupt MATLAB at any time by pressing **Ctrl-C** (pressing the **Ctrl** and **C** keys simultaneously) on a PC. Pressing ⌘–. (pressing the ⌘ and **.** keys simultaneously) on a Macintosh does the same thing.

1.5 Complex Numbers

One of the most powerful features of MATLAB is that it does not require any special handling for complex numbers. Complex numbers are formed in MATLAB in several ways. Examples of complex numbers include:

```
EDU» c1=1-2i   % the appended i signifies the imaginary part
c1 =
   1.0000 - 2.0000i

EDU» c1=1-2j   % j also works
c1 =
   1.0000 - 2.0000i

EDU» c2=3*(2-sqrt(-1)*3)
c2 =
   6.0000 - 9.0000i

EDU» c3=sqrt(-2)
c3 =
        0 + 1.4142i

EDU» c4=6+sin(.5)*i
c4 =
   6.0000 + 0.4794i

EDU» c5=6+sin(.5)*j
c5 =
   6.0000 + 0.4794i
```

In the last two examples, the MATLAB default values of `i=j=`$\sqrt{-1}$ are used to form the imaginary part. Multiplication by `i` or `j` is required in these cases since `sin(.5)i` and `sin(.5)j` mean nothing to MATLAB. Termination with the characters `i` and `j`, as shown in the first two examples above, works only for numerical constants, not for expressions.

Some programming languages require special handling for complex numbers wherever they appear. In MATLAB, no special handling is required. Mathematical operations on complex numbers are written the same as those for real numbers:

```
EDU» c6=(c1+c2)/c3   % from the above data
c6 =
  -7.7782 - 4.9497i
```

```
EDU» check_it_out=i^2  % sqrt(-1) squared must be -1!
check_it_out =
  -1.0000 + 0.0000i
```

In general, operations on complex numbers lead to complex numbers. In those cases where a negligible real or imaginary part remains, you can use the functions `real` and `imag` to extract the real and imaginary parts, respectively.

As a final example of complex arithmetic, consider the Euler (sounds like *Oiler*) identity, which relates the ***polar*** form of a complex number to its ***rectangular*** form:

$$M\angle\theta \equiv M \cdot e^{i\theta} = a + bi$$

where the polar form is given by a ***magnitude*** M and an ***angle*** θ, and the rectangular form is given by $a + bi$. The relationships among these forms are:

$$M = \sqrt{a^2 + b^2}$$

$$\theta = \tan^{-1}(b/a)$$

$$a = M\cos\theta$$

$$b = M\sin\theta$$

In MATLAB, the conversion between polar and rectangular forms makes use of the functions `real`, `imag`, `abs`, and `angle`:

```
EDU» c1
c1 =
   1.0000 - 2.0000i

EDU» mag_c1=abs(c1)
mag_c1 =
    2.2361

EDU» angle_c1=angle(c1)
angle_c1 =
   -1.1071

EDU» deg_c1=angle_c1*180/pi
deg_c1 =
  -63.4349

EDU» real_c1=real(c1)
real_c1 =
     1

EDU» imag_c1=imag(c1)
imag_c1 =
    -2
```

The MATLAB function `abs` computes the magnitude of complex numbers or the absolute value of real numbers, depending upon which one you give it. Likewise, the MATLAB function `angle` computes the angle of a complex number in radians.

Common Functions	
abs(x)	Absolute value or magnitude of complex number
acos(x)	Inverse cosine
acosh(x)	Inverse hyperbolic cosine
angle(x)	Four-quadrant angle of complex
asin(x)	Inverse sine
asinh(x)	Inverse hyperbolic sine
atan(x)	Inverse tangent
atan2(x,y)	Four-quadrant inverse tangent
atanh(x)	Inverse hyperbolic tangent
ceil(x)	Round toward plus infinity
conj(x)	Complex conjugate
cos(x)	Cosine
cosh(x)	Hyperbolic cosine
exp(x)	Exponential: e^x
fix(x)	Round toward zero
floor(x)	Round toward minus infinity
gcd(x,y)	Greatest common divisor of integers x and y
imag(x)	Complex imaginary part
lcm(x,y)	Least common multiple of integers x and y
log(x)	Natural logarithm
log10(x)	Common logarithm
real(x)	Complex real part
rem(x,y)	Remainder after division: rem(x,y) gives the remainder of x/y
round(x)	Round toward nearest integer
sign(x)	Signum function: return sign of argument, e.g., sign(1.2)=1, sign(-23.4)=-1, sign(0)=0
sin(x)	Sine
sinh(x)	Hyperbolic sine
sqrt(x)	Square root
tan(x)	Tangent
tanh(x)	Hyperbolic tangent

```
EDU» 4*atan(1) % one way to approximate pi
ans =
       3.1416

EDU» help atan2 % asks for help on the function atan2

 ATAN2 Four-quadrant inverse tangent.
    ATAN2(Y,X) is the four-quadrant arctangent of the real
    parts of the elements of X and Y.
    -pi <= ATAN2(Y,X) <= pi.

    See also ATAN.

EDU» help atan % see the difference between atan2 and atan?

 ATAN  Inverse tangent.
    ATAN(X) is the arctangent of the elements of X.

    See also ATAN2.

EDU» 180/pi*atan(-2/3) % atan2 uses vector sign information
ans =
      -33.69

EDU» 180/pi*atan2(2,-3)
ans =
      146.31

EDU» 180/pi*atan2(-2,3)
ans =
      -33.69

EDU» 180/pi*atan2(2,3)
ans =
       33.69

EDU» 180/pi*atan2(-2,-3) % 180/pi converts angle to degrees
ans =
     -146.31
```

Yet more examples include:

```
EDU» y=sqrt(3^2 + 4^2) % show 3-4-5 right triangle relationship
y =
     5

EDU» y=rem(23,4) % 23/4 has a remainder of 3
y =
     3
```

```
EDU» x=2.6, y1=fix(x), y2=floor(x), y3=ceil(x), y4=round(x)
x =
    2.6000
y1 =
     2
y2 =
     2
y3 =
     3
y4 =
     3

EDU» gcd(18,81) % 9 is the largest factor common to 18 and 81
ans =
     9

EDU» lcm(18,81) % 162 is the least common multiple of 18 and 81
ans =
   162
```

Example: Estimating the Height of a Building

Problem: Consider the problem of estimating the height of a building, as illustrated in Fig. 2.1. If the observer is a distance D from the building, the angle from the observer to the top of the building is θ, and the height of the observer is h, what is the building height?

Solution: Draw a simple diagram as shown in Fig. 2.1. The building height is h + H, where H is the length of the triangle side opposite the observer. This length can be found from the triangle relationship

$$\tan(\theta) = \frac{H}{D}$$

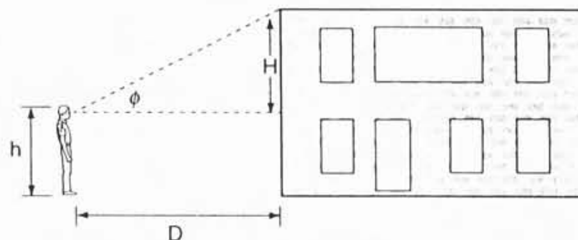

Figure 2.1 Estimating Building Height

Therefore, the building height is $h + H = h + D \cdot \tan(\theta)$.

If $h = 2$ meters, $D = 50$ meters, and θ is 60 degrees, MATLAB gives a solution of:

```
EDU» h=2
h =
     2
EDU» theta=60
theta =
    60
EDU» D=50
D =
    50
EDU» building_height=h+D*tan(theta*pi/180)
building_height =
       88.603
```

Note that since MATLAB always uses radians, θ was converted to radians by multiplying by pi/180 before being passed to the tangent function.

Example: Radioactive Decay Example

Problem: The radioactive element polonium has a half-life of 140 days, which means that because of radioactive decay the amount of polonium remaining after 140 days is one-half of the original amount. Starting with 10 grams of polonium today, how much is left after 250 days?

Solution: After one half-life, or 140 days, $10 \cdot 0.5 = 5$ grams remain. After two half-lives or 280 days, $5 \cdot 0.5 = 10 \cdot 0.5 \cdot 0.5 = 10 \cdot (0.5)^2 = 2.5$ grams remain. Therefore, the correct solution should be between 5 and 2.5 grams, and the amount remaining after any period of time is given by

$$\text{amount remaining} = \text{initial amount} \cdot (0.5)^{\text{time/half-time}}$$

For this example, time = 250, and the MATLAB solution is

```
EDU» initial_amount=10;
EDU» half_life=140;
EDU» time=250;
EDU» amount_left=initial_amount*0.5^(time/half_life)
amount_left =
       2.9003
```

Thus, approximately 2.9 grams of polonium is left after 250 days. Note that exponentiation ^ takes precedence over multiplication *. Therefore, `0.5^(time/half_life)` is computed before multiplying by `initial_amount`.

Example: Acid Concentration Problem

Problem: As part of the manufacturing process for cast parts at an automotive plant, parts are dipped in water to cool them, then dipped in an acid–water bath to clean them. Over time, the concentration of the acid–water solution decreases because of the water introduced at immersion and the solution removed when the parts are taken from the acid–water bath. To maintain quality, the acid–water solution's acidity must not fall below some minimum. Start with a 90% acid–water concentration. If the minimum concentration is 50%, the water introduced into the acid–water bath is equal to 1% of the bath's volume, and 1% of the solution is removed when the part is removed, how many parts can be dipped into the acid–water bath before it drops below the minimum acidity?

Solution: Initially, the acid concentration is initial_con = 90% = acid/(acid + water). When the first part is dipped in the acid–water bath, the concentration is reduced to

$$\begin{aligned}\text{con} &= \frac{\text{acid}}{(\text{acid} + \text{water}) + \text{water added}} \\ &= \frac{\text{acid}}{(\text{acid} + \text{water}) + \text{lost}(\text{acid} + \text{water})} \\ &= \frac{\text{acid}}{(1 + \text{lost})(\text{acid} + \text{water})} \\ &= \frac{\text{initial_con}}{(1 + \text{lost})}\end{aligned}$$

where "acid" is the initial acid volume, "water" is the initial water volume, and "lost" is the fractional volume of water added. The amount of acid remaining in the solution after this first dip is, therefore

$$\text{acid_left} = \frac{\text{acid}}{(1 + \text{lost})}$$

This means that, when the second part is dipped into the acid–water bath, the concentration is

$$\begin{aligned}\text{con} &= \frac{\text{acid_left}}{(\text{acid} + \text{water}) + \text{water added}} \\ &= \frac{\text{acid_left}}{(1 + \text{lost})(\text{acid} + \text{water})} \\ &= \frac{\text{initial_con}}{(1 + \text{lost})^2}\end{aligned}$$

Following this process, after n dips, the concentration of the acid–water bath is

$$\text{con} = \frac{\text{initial_con}}{(1 + \text{lost})^n}$$

If min_con is the minimum acceptable concentration, the maximum number of dips is the integer less than or equal to

$$n = \frac{\log(\text{initial_con}/\text{min_con})}{\log(1 + \text{lost})}$$

In MATLAB, the solution is

```
EDU» initial_con=90
initial_con =
    90
EDU» min_con=50
min_con =
    50
EDU» lost=0.01;
EDU» n=floor(log(initial_con/min_con)/log(1+lost))
n =
    59
```

Fifty-nine dips can be completed before the concentration drops below 50%. Note that the `floor` function was used to round `n` down to the nearest integer. Also note that while the natural logarithm was used, `log10` or `log2` could have also been used.

Example: Interest Calculations

Problem: You've agreed to buy a new car for $18,500. The car dealer is offering two financing options: (1) 2.9% interest over 4 years, or (2) 8.9% interest over 4 years, with a factory rebate of $1500. Which one is the better deal?

Solution: The monthly payment P on a loan of A dollars, having a monthly interest rate of R, paid off in M months, is

$$P = A\left[\frac{R(1 + R)^M}{(1 + R)^M - 1}\right]$$

giving a total amount paid of $T = P \cdot M$.

In MATLAB, the solution is

```
EDU» format bank % use bank display format
EDU» A=18500;  % amount of loan
EDU» M=12*4;   % number of months
EDU» FR=1500;  % factory rebate
EDU» % first financing offer
EDU» R=(2.9/100)/12; % monthly interest rate
EDU» P=A*( R*(1+R)^M/( (1+R)^M -1)) % payment
P =
       408.67
EDU» T1=P*M % total car cost
T1 =
         19616.06
EDU» % second financing offer
EDU» R=(8.9/100)/12; % monthly interest rate
EDU» P=(A-FR)*( R*(1+R)^M/( (1+R)^M -1)) % payment
P =
       422.24
EDU» T2=P*M % total car cost
T2 =
         20267.47
EDU» Diff=T2-T1
Diff =
       651.41
```

Based on these results, the first financing offer is the better of the two.

All of the computations considered to this point have involved single numbers called ***scalars***. Operations involving scalars are the basis of mathematics. At the same time, when you wish to perform the same operation on more than one number at a time, repeated scalar operations are time-consuming and cumbersome. To solve this problem, MATLAB defines operations on data arrays.

6.1 Simple Arrays

Consider the problem of computing values of the sine function over one-half of its period, namely: $y = \sin(x)$ over $0 \le x \le \pi$. Since it is impossible to compute $\sin(x)$ at all points over this range (there is an infinite number of them!), we must choose a finite number of points. In doing so, we are *sampling* the function. To pick some number, let's evaluate $\sin(x)$ every 0.1π in this range, i.e., let $x = 0, 0.1\pi, 0.2\pi, \ldots, 1.0\pi$. If you were using a scientific calculator to compute these values, you would start by making a list, or ***array***, of the values of x. Then you would enter each value of x into your calculator, find its sine, and write down the result as the second array y. Perhaps you would write them in an organized fashion as follows:

x	0	$.1\pi$	$.2\pi$	$.3\pi$	$.4\pi$	$.5\pi$	$.6\pi$	$.7\pi$	$.8\pi$	$.9\pi$	π
y	0	.31	.59	.81	.95	1.0	.95	.81	.59	.31	0

As shown, x and y are ordered lists of numbers, i.e., the first value or element in y is associated with the first value or element in x, the second element in y is associated with the second element in x, and so on. Because of this ordering, it is common to refer to individual values or elements in x and y with subscripts, e.g., x_1 is the first element in x, y_5 is the fifth element in y, x_n is the nth element in x.

MATLAB handles arrays in a straightforward and intuitive way. Creating arrays is easy—just follow the visual organization given above:

```
EDU» x=[0 .1*pi .2*pi .3*pi .4*pi .5*pi .6*pi .7*pi .8*pi .9*pi pi]
x =
  Columns 1 through 7
         0    0.3142    0.6283    0.9425    1.2566    1.5708    1.8850
  Columns 8 through 11
    2.1991    2.5133    2.8274    3.1416
```

```
EDU» y=sin(x)
y =
  Columns 1 through 7
         0    0.3090    0.5878    0.8090    0.9511    1.0000    0.9511
  Columns 8 through 11
    0.8090    0.5878    0.3090    0.0000
```

To create an array in MATLAB, all you have to do is to start with a left bracket, enter the desired values separated by spaces (or commas), then close the array with a right bracket. Notice that finding the sine of the values in x follows naturally. MATLAB understands that you want to find the sine of each element in x and place the results in an associated array called y. This fundamental capability makes MATLAB different than other computer languages.

Since spaces separate array values, complex numbers entered as array values cannot have embedded spaces unless expressions are enclosed in parentheses. For example, [1 -2i 3 4 5+6i] contains five elements, whereas the identical arrays [(1 -2i) 3 4 5+6i] and [1-2i 3 4 5+6i] contain four.

6.2 Array Addressing

Now, since x in the preceding example has more than one element (namely, it has 11 values separated into columns), MATLAB gives you the result back with the columns identified. As shown previously, x is an array having one row and eleven columns, or, in mathematical jargon, it is a ***row vector***, a ***one-by-eleven array,*** or simply an *array* of length 11.

In MATLAB, individual array elements are accessed using ***subscripts***, e.g., $x(1)$ is the first element in x, $x(2)$ is the second element in x, and so on. For example:

```
EDU» x(3)  % The third element of x
ans =
    0.6283

EDU» y(5)  % The fifth element of y
ans =
    0.9511
```

To access a block of elements at one time, MATLAB provides ***colon notation***:

```
EDU» x(1:5)
ans =
         0    0.3142    0.6283    0.9425    1.2566
```

These are the first through fifth elements in x. 1:5 says "start with 1 and count up to 5."

```
EDU» x(7:end)
ans =
        1.885     2.1991     2.5133     2.8274     3.1416
```

starts with the seventh element and continues to the last element. Here, the word end signifies the last element in the array x.

```
EDU» y(3:-1:1)
ans =
    0.5878    0.3090         0
```

These are the third, second, and first elements in reverse order. 3:-1:1 says "start with 3, count down by 1, and stop at 1."

```
EDU» x(2:2:7)
ans =
    0.3142    0.9425    1.5708
```

These are the second, fourth, and sixth elements in x. 2:2:7 says "start with 2, count up by 2, and stop when you get to 7." In this case, adding 2 to 6 gives 8, which is greater than 7, so the eighth element is not included.

```
EDU» y([8 2 9 1])
ans =
    0.8090    0.3090    0.5878         0
```

Here we used another array [8 2 9 1] to extract the elements of the array y in the order we wanted them! The first element taken is the eighth, the second is the second, the third is the ninth, and the fourth is the first. In fact, [8 2 9 1] is an array that addresses the desired elements of y.

6.3 Array Construction

Earlier we entered the values of x by typing each individual element in x. While this is fine when there are only 11 values in x, what if there are 111 values? Using the colon notation, two other ways of entering x are:

```
EDU» x=(0:0.1:1)*pi
x =
  Columns 1 through 7
         0    0.3142    0.6283    0.9425    1.2566    1.5708    1.8850
  Columns 8 through 11
    2.1991    2.5133    2.8274    3.1416
```

```
EDU» x=linspace(0,pi,11)
x =
  Columns 1 through 7
         0    0.3142    0.6283    0.9425    1.2566    1.5708    1.8850
  Columns 8 through 11
    2.1991    2.5133    2.8274    3.1416
```

In the first case, the colon notation (0:0.1:1) creates an array that starts at 0, increments by 0.1, and ends at 1. Each element in this array is then multiplied by π to create the desired values in x. In the second case, the MATLAB function linspace is used to create x. This function's arguments are described by:

```
linspace(first_value,last_value,number_of_values)
```

Both of these array creation forms are common in MATLAB. The colon notation form allows you to directly specify the increment between data points, but not the number of data points. linspace, on the other hand, allows you to directly specify the number of data points, but not the increment between the data points.

Both of the preceding array creation forms create arrays where the individual elements are linearly spaced with respect to each other. For the special case where a logarithmically spaced array is desired, MATLAB provides the logspace function:

```
EDU» logspace(0,2,11)
ans =
  Columns 1 through 7
    1.0000    1.5849    2.5119    3.9811    6.3096   10.0000   15.8489
  Columns 8 through 11
   25.1189   39.8107   63.0957  100.0000
```

Here, we created an array starting at 10^0, ending at 10^2, containing **11** values. The function arguments are described by:

```
logspace(first_exponent,last_exponent,number_of_values)
```

Though it is common to begin and end at integer powers of ten, logspace works equally well with nonintegers.

Sometimes an array is required that is not conveniently described by a linearly or logarithmically spaced element relationship. There is no uniform way to create these arrays. However, array addressing and the ability

to combine expressions can help eliminate the need to enter individual elements one at a time:

```
EDU» a=1:5,b=1:2:9
a =
     1     2     3     4     5
b =
     1     3     5     7     9
```

creates two arrays. Remember that multiple statements can appear on a single line if they are separated by commas or semicolons.

```
EDU» c=[b a]
c =
     1     3     5     7     9     1     2     3     4     5
```

creates an array c composed of the elements of b followed by those of a.

```
EDU» d=[a(1:2:5) 1 0 1]
d =
     1     3     5     1     0     1
```

creates an array d composed of the first, third, and fifth elements of a followed by three additional elements.

The basic array construction features of MATLAB are summarized in the following table.

Basic Array Construction	
x=[2 2*pi sqrt(2) 2-3j]	Create row vector x containing elements specified
x=first:last	Create row vector x starting with first, counting by one, ending at or before last
x=first:increment:last	Create row vector x starting with first, counting by increment, ending at or before last
x=linspace(first,last,n)	Create row vector x starting with first, ending at last, having n elements
x=logspace(first,last,n)	Create logarithmically spaced row vector x starting with 10^{first}, ending at 10^{last}, having n elements