MATH 259
Numerical Methods
Introduction to MATLAB
Part 1

Original Slides by
Gerald W. Recktenwald

Version 1.1 August 21, 2006

Overview (1)

- Basic MATLAB Operations
  - ▷ Starting MATLAB
  - ▷ Using MATLAB as a calculator
  - ▷ Introduction to variables and functions
- Matrices and Vectors: *All variables are matrices.*
  - ▷ Creating matrices and vectors
  - ▷ Subscript notation
  - ▷ Colon notation

Overview (2)

- Additional Types of Variables
  - ▷ Complex numbers
  - ▷ Strings
  - ▷ Polynomials
- Working with Matrices and Vectors
  - ▷ Some basic linear algebra
  - ▷ Vectorized operations
  - ▷ Array operators
- Managing the Interactive Environment
- Plotting

Starting MATLAB

- Double click on the MATLAB icon, or on unix systems type "`matlab`" at the command line.
- After startup MATLAB displays a *command window* that is used to enter commands and display text-only results.
- Enter Commands at the command prompt:

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

- MATLAB responds to commands by printing text in the command window, or by opening a *figure window* for graphical output.
- Toggle between windows by clicking on them with the mouse.

MATLAB Desktop

Command Prompt:
Enter typed commands here.
Text results are displayed here.

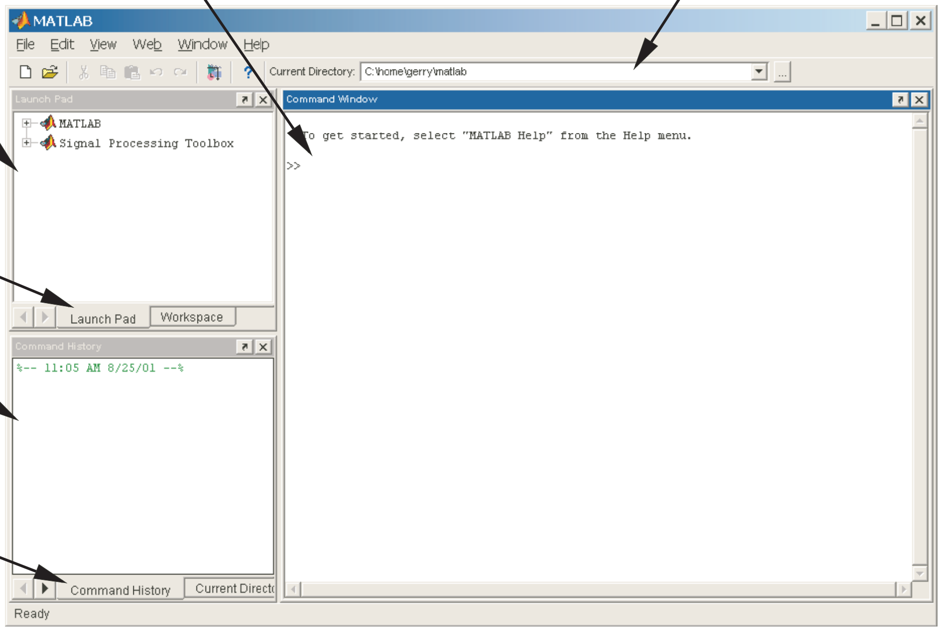
Recent Directory Menu:
Used to change current working directory.

Launch Pad/Workspace:
Used to browse documentation, or view values of variables in the workspace

Select Launch Pad tab or Workspace tab

Command History/Current Directory: View and re-enter previously typed commands, or change directories

Select Command History tab or Current Directory tab



```
%-- 11:05 AM 8/25/01 --%
```

```
To get started, select "MATLAB Help" from the Help menu.

>>
```

MATLAB Desktop

- The desktop provides different ways of interacting with MATLAB
  - ▷ Entering commands in the command window
  - ▷ Viewing values stored in variables
  - ▷ Editing statements in MATLAB functions and scripts
  - ▷ Creating and annotating plots

- Watch an animated demonstration of the MATLAB desktop by typing

```
playbackdemo('desktop')
```

  at the command prompt.

```
>> demo
```

MATLAB as a Calculator (1)

Enter formulas at the command prompt

```
>> 2 + 6 - 4                    (press return after "4")
ans =
    4

>> ans/2
ans =
    2
```

MATLAB **as a Calculator** (2)

Define and use variables

```
>> a = 5
a =
    5

>> b = 6
b =
    6

>> c = b/a
c =
    1.2000
```

Built-in Variables

`pi` $(= \pi)$ and `ans` are a built-in variables

```
>> pi
ans =
    3.1416

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

Note: There is no “degrees” mode. All angles are measured in radians.

Built-in Functions

Many standard mathematical functions, such as `sin`, `cos`, `log`, and `log10`, are built-in

`>> log(256)`
`ans =`
`    5.5452`
	`log(x)` computes the natural logarithm of `x`

`>> log10(256)`
`ans =`
`    2.4082`
	`log10(x)` is the base 10 logarithm

`>> log2(256)`
`ans =`
`    8`
	`log2(x)` is the base 2 logarithm

Ways to Get Help

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

```
>> help sqrt
```

- In MATLAB version 6 and later the `doc` function opens the on-line version of the manual. This is very helpful for more complex commands.

```
>> doc plot
```

- Use `lookfor` to find functions by keywords

```
>> lookfor functionName
```

On-line Help (1)

Syntax:

```
help functionName
```

Example:

```
>> help log
```

produces

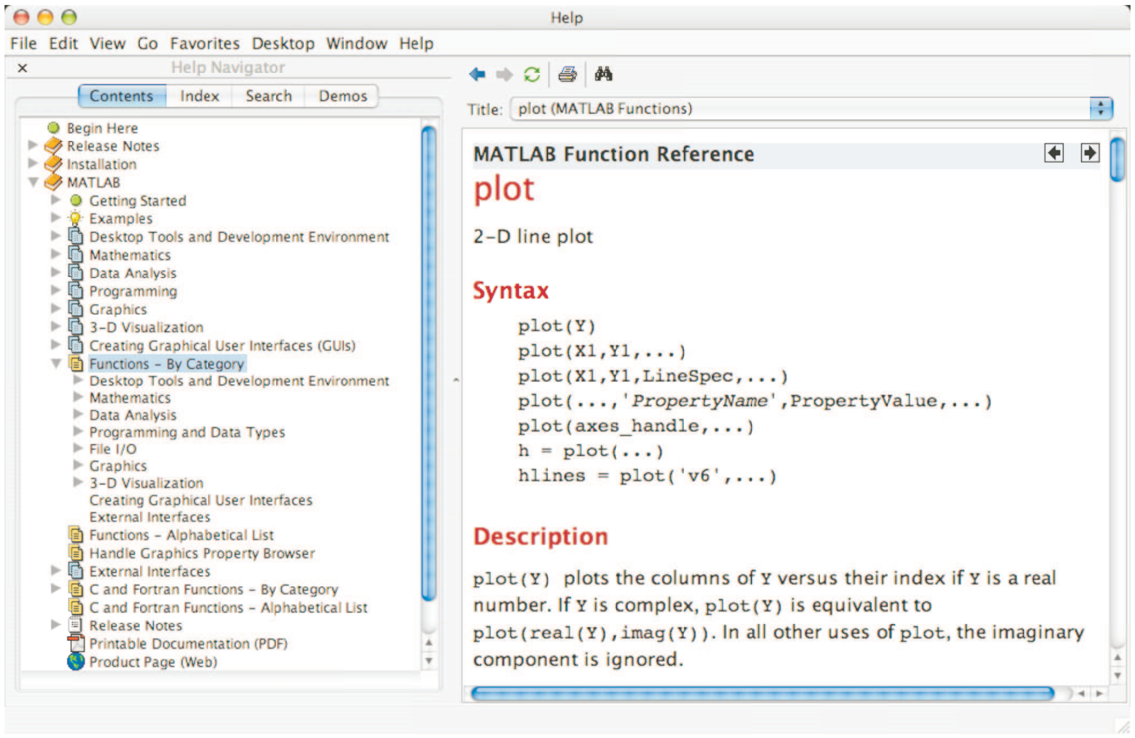
```
 LOG    Natural logarithm.
    LOG(X) is the natural logarithm of the elements of X.
    Complex results are produced if X is not positive.

    See also LOG2, LOG10, EXP, LOGM.
```

The `help` function provides a compact summary of how to use a command. Use the `doc` function to get more in-depth information.

On-line Help (2)

The help browser opens when you type a `doc` command:

```
>> doc plot
```


Help
File Edit View Go Favorites Desktop Window Help
Help Navigator
Contents
Index
Search
Demos
Begin Here
Release Notes
Installation
MATLAB
Getting Started
Examples
Desktop Tools and Development Environment
Mathematics
Data Analysis
Programming
Graphics
3-D Visualization
Creating Graphical User Interfaces (GUIs)
Functions – By Category
Desktop Tools and Development Environment
Mathematics
Data Analysis
Programming and Data Types
File I/O
Graphics
3-D Visualization
Creating Graphical User Interfaces
External Interfaces
Functions – Alphabetical List
Handle Graphics Property Browser
External Interfaces
C and Fortran Functions – By Category
C and Fortran Functions – Alphabetical List
Release Notes
Printable Documentation (PDF)
Product Page (Web)
Title: plot (MATLAB Functions)

MATLAB Function Reference

plot

2-D line plot

Syntax

```
plot(Y)
plot(X1,Y1,...)
plot(X1,Y1,LineSpec,...)
plot(...,'PropertyName',PropertyValue,...)
plot(axes_handle,...)
h = plot(...)
hlines = plot('v6',...)
```

Description

`plot(Y)` plots the columns of `Y` versus their index if `Y` is a real number. If `Y` is complex, `plot(Y)` is equivalent to `plot(real(Y),imag(Y))`. In all other uses of `plot`, the imaginary component is ignored.

Looking for Functions

Syntax:

```
lookfor string
```

searches first line of function descriptions for "*string*".

Example:

```
>> lookfor cosine
```

produces

```
ACOS    Inverse cosine.
ACOSH   Inverse hyperbolic cosine.
COS     Cosine.
COSH    Hyperbolic cosine.
```

Strategies for Interactive Computing

- Use the command window for short sequences of calculations
- Later we'll learn how to build reusable functions for more complex tasks.
- The command window is good for testing ideas and running sequences of operations contained in functions
- Any command executed in the command window can also be used in a function.

Let's continue with a tour of interactive computing.

Suppress Output with Semicolon (1)

Results of intermediate steps can be suppressed with semicolons.

Example: Assign values to x, y, and z, but only display the value of z in the command window:

```
>> x = 5;
>> y = sqrt(59);
>> z = log(y) + x^0.25
   z =
       3.5341
```

Suppress Output with Semicolon (2)

Type variable name and omit the semicolon to print the value of a variable (that is already defined)

```
>> x = 5;
>> y = sqrt(59);
>> z = log(y) + x^0.25
   z =
       3.5341
>> y
   y =
       7.6811                ( = log(sqrt(59)) + 5^0.25 )
```

Multiple Statements per Line

Use commas or semicolons to enter more than one statement at once. Commas allow multiple statements per line without suppressing output.

```
>> a = 5;   b = sin(a),  c = cosh(a)
b =
    -0.9589

c =
    74.2099
```

MATLAB **Variables Names**

Legal variable names:

- Begin with one of a–z or A–Z
- Have remaining characters chosen from a–z, A–Z, 0–9, or _
- Have a maximum length of 31 characters
- Should not be the name of a built-in variable, built-in function, or user-defined function

Examples:

```
xxxxxxxxx
pipeRadius
widgets_per_box
mySum
mysum
```

Note: `mySum` and `mysum` are *different* variables. MATLAB is *case sensitive*.

Built-in MATLAB **Variables** (1)

Name	Meaning
`ans`	value of an expression when that expression is not assigned to a variable
`eps`	floating point precision
`pi`	π, $(3.141492\ldots)$
`realmax`	largest positive floating point number
`realmin`	smallest positive floating point number
`Inf`	∞, a number larger than `realmax`, the result of evaluating $1/0$.
`NaN`	not a number, the result of evaluating $0/0$

Built-in MATLAB Variables (2)

Rule: Only use built-in variables on the right hand side of an expression. Reassigning the value of a built-in variable can create problems with built-in functions.

Exception: `i` and `j` are preassigned to $\sqrt{-1}$. One or both of `i` or `j` are often reassigned as loop indices. More on this later.

Matrices and Vectors

All MATLAB variables are matrices

- A vector is a matrix with one row *or* one column.
- A scalar is a matrix with one row *and* one column.
- A character string is a row of column vector of characters.

Consequences:

- Rules of linear algebra apply to addition, subtraction and multiplication.
- Elements in the vectors and matrices are addressed with Fortran-like subscript notation, e.g.,, `x(2)`, `A(4,5)`. Usually this notation is clear from context, but it can be confused with a function call,

```
y = sqrt(5)                        sqrt is a built-in function
z = f(3)                           Is f a function or variable?
```

Creating MATLAB Variables

MATLAB variables are created with an assignment statement

```
>> x = expression
```

where *expression* is a legal combinations of numerical values, mathematical operators, variables, and function calls.

The *expression* can involve:

- Manual entry
- Built-in functions that return matrices
- Custom (user-written) functions that return matrices
- Loading matrices from text files or "`mat`" files

Element-by-Element Creation of Matrices and Vectors (1)

A matrix, a column vector, and a row vector:

$$A = \begin{bmatrix} 3 & 2 \\ 3 & 1 \\ 1 & 4 \end{bmatrix}$$

$$x = \begin{bmatrix} 5 \\ 7 \\ 9 \\ 2 \end{bmatrix}$$

$$v = \begin{bmatrix} 9 & -3 & 4 & 1 \end{bmatrix}$$

As MATLAB variables:

```
>> A = [3 2; 3 1; 1 4]
A =
     3     2
     3     1
     1     4
>> x = [5; 7; 9; 2]
x =
     5
     7
     9
     2
>> v = [9 -3 4 1]
v =
     9    -3     4     1
```

Element-by-Element Creation of Matrices and Vectors (2)

For manual entry, the elements in a vector are enclosed in square brackets. When creating a row vector, separate elements with a space.

```
>> v = [7 3 9]
v =
     7   3  9
```

Separate columns with a semicolon

```
>> w = [2; 6; 1]
w =
     2
     6
     1
```

Element-by-Element Creation of Matrices and Vectors (3)

When assigning elements to matrix, row elements are separated by spaces, and columns are separated by semicolons

```
>> A = [1 2 3; 5 7 11; 13 17 19]
A =
     1     2     3
     5     7    11
    13    17    19
```

Transpose Operator (1)

Once it is created, a variable can be transformed with other operators. The *transpose operator* converts a row vector to a column vector (and *vice versa*), and it changes the rows of a matrix to columns.

```
>> v = [2 4 1 7]
v =
     2     4     1     7

>> w = v'
w =
     2
     4
     1
     7
```

Transpose Operator (2)

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

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

Overwriting Variables

Once a variable has been created, it can be reassigned

```
>> x = 2;
>> x = x + 2
x =
    4

>> y = [1 2 3 4]
y =
     1     2     3     4

>> y = y'
y =
     1
     2
     3
     4
```

Using Functions to Create Matrices and Vectors

Create vectors with built-in functions:

`linspace` and `logspace`

Create matrices with built-in functions:

`ones`, `zeros`, `eye`, `diag`, . . .

Note that `ones` and `zeros` can also be used to create vectors.

Creating vectors with `linspace` (1)

The `linspace` function creates vectors with elements having uniform linear spacing.

Syntax:

```
x = linspace(startValue,endValue)
x = linspace(startValue,endValue,nelements)
```

Examples:

```
>> u = linspace(0.0,0.25,5)
u =
         0    0.0625    0.1250    0.1875    0.2500

>> u = linspace(0.0,0.25);
```

Remember: Ending a statement with semicolon suppresses the output.

Creating vectors with `linspace` (2)

Column vectors are created by appending the transpose operator to `linspace`

```
>> v = linspace(0,9,4)'
v =
     0
     3
     6
     9
```

Example: A Table of Trig Functions

```
>> x = linspace(0,2*pi,6)';          (note transpose)
>> y = sin(x);
>> z = cos(x);
>> [x y z]
ans =
         0         0    1.0000
    1.2566    0.9511    0.3090
    2.5133    0.5878   -0.8090
    3.7699   -0.5878   -0.8090
    5.0265   -0.9511    0.3090
    6.2832         0    1.0000
```

The expressions `y = sin(x)` and `z = cos(x)` take advantage of *vectorization*. If the input to a vectorized function is a vector or matrix, the output is often a vector or matrix having the same shape. More on this later.

Creating vectors with `logspace`

The `logspace` function creates vectors with elements having uniform logarithmic spacing.

Syntax:

```
x = logspace(startValue,endValue)
x = logspace(startValue,endValue,nelements)
```

creates *nelements* elements between $10^{\mathrm{startValue}}$ and 10^{endValue}. The default value of *nelements* is 100.

Example:

```
>> w = logspace(1,4,4)
w =
          10         100        1000       10000
```

Functions to Create Matrices (1)

Name	Operation(s) Performed
`diag`	create a matrix with a specified diagonal entries, or extract diagonal entries of a matrix
`eye`	create an identity matrix
`ones`	create a matrix filled with ones
`rand`	create a matrix filled with random numbers
`zeros`	create a matrix filled with zeros
`linspace`	create a row vector of linearly spaced elements
`logspace`	create a row vector of logarithmically spaced elements

Functions to Create Matrices (2)

Use `ones` and `zeros` to set intial values of a matrix or vector.

Syntax:

```
A = ones(nrows,ncols)
A = zeros(nrows,ncols)
```

Examples:

```
>> D = ones(3,3)
D =
     1     1     1
     1     1     1
     1     1     1
>> E = ones(2,4)
E =
     1     1     1     1
     1     1     1     1
```

Functions to Create Matrices (3)

`ones` and `zeros` are also used to create vectors. To do so, set either *`nrows`* or *`ncols`* to 1.

```
>> s = ones(1,4)
s =
     1     1     1     1

>> t = zeros(3,1)
t =
     0
     0
     0
```

Functions to Create Matrices (4)

The `eye` function creates identity matrices of a specified size. It can also create non-square matrices with ones on the main diagonal.

Syntax:

```
A = eye(n)
A = eye(nrows,ncols)
```

Examples:

```
>> C = eye(5)
C =
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1
```

Functions to Create Matrices (5)

The optional second input argument to eye allows non-square matrices to be created.

```
>> D = eye(3,5)
D =
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
```

where $D_{i,j} = 1$ whenever $i = j$.

Functions to Create Matrices (6)

The `diag` function can *either* create a matrix with specified diagonal elements, *or* extract the diagonal elements from a matrix

Syntax:

```
A = diag(v)
v = diag(A)
```

Example: Use `diag` to create a matrix

```
>> v = [1 2 3];
>> A = diag(v)
A =
     1     0     0
     0     2     0
     0     0     3
```

Functions to Create Matrices (7)

Example: Use `diag` to extract the diagonal of a matrix

```
>> B = [1:4; 5:8; 9:12]
B =
     1     2     3     4
     5     6     7     8
     9    10    11    12

>> w = diag(B)
w =
     1
     6
    11
```

Functions to Create Matrices (8)

The action of the `diag` function depends on the characteristics and number of the input(s). This polymorphic behavior of MATLAB functions is common. Refer to the on-line documentation for the possible variations.

```
>> A = diag([3 2 1])              Create a matrix with a specified diagonal
A =
     3     0     0
     0     2     0
     0     0     1

>> B = [4 2 2; 3 6 9; 1 1 7];
>> v = diag(B)                    Extract the diagonal of a matrix
v =
     4
     6
     7
```

Subscript Notation (1)

If `A` is a matrix, `A(i,j)` selects the element in the ith row and jth column. Subscript notation can be used on the right hand side of an expression to refer to a matrix element.

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

>> c = A(1,1)
c =
     1
```

Subscript Notation (1)

Subscript notation is also used to assign matrix elements

```
>> A(1,1) = c/b
A =
    0.2500    2.0000    3.0000
    4.0000    5.0000    6.0000
    7.0000    8.0000    9.0000
```

Referring to elements beyond the dimensions the matrix results in an error

```
>> A = [1 2 3; 4 5 6; 7 8 9];
>> A(1,4)
???  Index exceeds matrix dimensions.
```

Subscript Notation (1)

Assigning an element that is beyond the existing dimensions of the matrix causes the matrix to be resized!

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

>> A(4,4) = 11
A =
     1     2     3     0
     4     5     6     0
     7     8     9     0
     0     0     0    11
```

In other words, MATLAB automatically resizes matrices on the fly.

Colon Notation (1)

Colon notation is very powerful and very important in the effective use of Matlab. The colon is used as both an operator and as a wildcard.

Use colon notation to:

- create vectors
- refer to or extract ranges of matrix elements

Colon Notation (2)

Syntax:

```
startValue:endValue
startValue:increment:endValue
```

Note: *`startValue`*, *`increment`*, and *`endValue`* do not need to be integers

Colon Notation (3)

Creating row vectors:

```
>> s = 1:4
s =
     1     2     3     4

>> t = 0:0.1:0.4
t =
         0    0.1000    0.2000    0.3000    0.4000
```

Colon Notation (4)

Creating column vectors:

```
>> u = (1:5)’
u =
     1
     2
     3
     4
     5

>> v = 1:5’
v =
     1     2     3     4     5
```

`v` is a row vector because `1:5’` creates a vector between 1 and the transpose of 5.

Colon Notation (5)

Use colon as a wildcard to refer to an entire column or row

```
>> A = [1 2 3; 4 5 6; 7 8 9];
>> A(:,1)
ans =
     1
     4
     7

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

Colon Notation (6)

Or use colon notation to refer to subsets of columns or rows

```
>> A(2:3,1)
ans =
     4
     7

>> A(1:2,2:3)

ans =
     2     3
     5     6
```

Colon Notation (7)

Colon notation is often used in compact expressions to obtain results that would otherwise require several steps.

Example:

```
>> A = ones(8,8);
>> A(3:6,3:6) = zeros(4,4)
A =
     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1
     1     1     0     0     0     0     1     1
     1     1     0     0     0     0     1     1
     1     1     0     0     0     0     1     1
     1     1     0     0     0     0     1     1
     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1
```

Colon Notation (8)

Finally, colon notation is used to convert any vector or matrix to a column vector.

Example:

```
>> x = 1:4;
>> y = x(:)
y =
     1
     2
     3
     4
```

Colon Notation (9)

Colon notation converts a matrix to a column vector by appending the columns of the input matrix

```
>> A = rand(2,3);
>> v = A(:)
v =
    0.9501
    0.2311
    0.6068
    0.4860
    0.8913
    0.7621
    0.4565
```

Note: The `rand` function generates random elements between zero and one. Repeating the preceding statements will, in all likelihood, produce different numerical values for the elements of `v`.

Additional Types of Variables

Matrix elements can either be numeric values or characters. Numeric elements can either be real or complex (imaginary).

More general variable types are available: n-dimensional arrays (where $n > 2$), structs, cell arrays, and objects. Numeric (real and complex) and string arrays of dimension two or less will be sufficient for our purposes.

Consider some simple variations on numeric and string matrices:

- Complex Numbers
- Strings
- Polynomials

Complex Numbers

MATLAB *automatically* performs complex arithmetic

```
>> sqrt(-4)
ans =
        0 + 2.0000i

>> x = 1 + 2*i                (or,  x = 1 + 2*j)
x =
   1.0000 + 2.0000i

>> y = 1 - 2*i
y =
   1.0000 - 2.0000i

>> z = x*y
z =
     5
```

Unit Imaginary Numbers (1)

`i` and `j` are ordinary MATLAB variables *preassigned* with the value $\sqrt{-1}$.

```
>> i^2
ans =
    -1
```

Both or either `i` and `j` can be *reassigned*

```
>> i = 5;
>> t = 8;
>> u = sqrt(i-t)                (i-t = -3,  not -8+i)
u =
        0 + 1.7321i
>> u*u
ans =
   -3.0000
```

Unit Imaginary Numbers (2)

The `i` and `j` variables are often used for array subscripting. Set `i` (or `j`) to an integer value that is also a valid array subscript.

```
>> A = [1 2; 3 4];
>> i = 2;
>> A(i,i) = 1
A =
     1     2
     3     1

>> x = A(2,j)
??? Subscript indices must either be real positive integers or logicals.
```

Note: When working with complex numbers, it is a good idea to reserve either `i` or `j` for the unit imaginary value $\sqrt{-1}$.

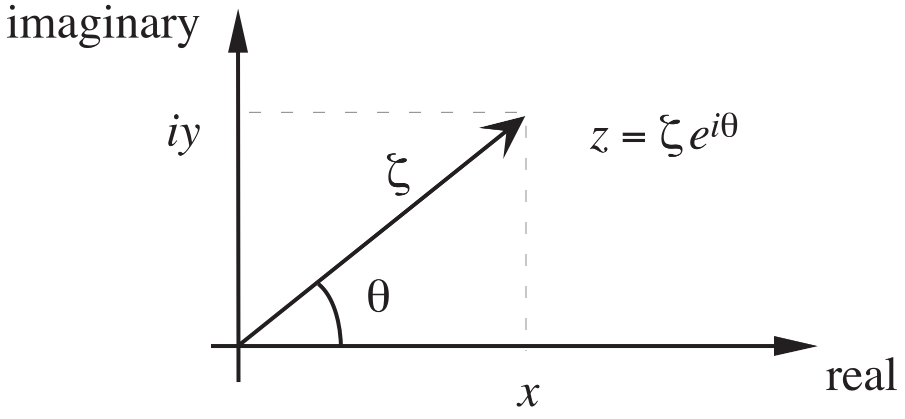
Euler Notation (1)

Euler notation represents a complex number by a *phaser*

$$z = \zeta e^{i\theta}$$

$$x = \mathrm{Re}(z) = |z|\cos(\theta) = \zeta\cos(\theta)$$

$$y = i\mathrm{Im}(z) = i|z|\sin(\theta) = i\zeta\sin(\theta)$$



Functions for Complex Arithmetic (1)

Function	Operation
`abs`	Compute the magnitude of a number `abs(z)` is equivalent to `sqrt( real(z)^2 + imag(z)^2 )`
`angle`	Angle of complex number in Euler notation
`exp`	If `x` is real, $\texttt{exp(x)} = e^x$ If `z` is complex, $\texttt{exp(z)} = e^{\mathrm{Re(z)}}(\cos(\mathrm{Im}(z) + i\sin(\mathrm{Im}(z))$
`conj`	Complex conjugate of a number
`imag`	Extract the imaginary part of a complex number
`real`	Extract the real part of a complex number

Functions for Complex Arithmetic (2)

Examples:

```
>> zeta = 5;  theta = pi/3;
>> z = zeta*exp(i*theta)
z =
   2.5000 + 4.3301i

>> abs(z)
ans =
     5

>> sqrt(z*conj(z))
ans =
     5
```

```
>> x = real(z)
x =
    2.5000

>> y = imag(z)
y =
    4.3301

>> angle(z)*180/pi
ans =
   60.0000
```

Remember: There is no “degrees” mode in MATLAB. All angles are in radians.

Strings

- Strings are matrices with character elements.
- String constants are enclosed in single quotes
- Colon notation and subscript operations apply

Examples:

```
>> first = 'John';
>> last  = 'Coltrane';
>> name  = [first,' ',last]
name =
John Coltrane

>> length(name)
ans =
    13

>> name(9:13)
ans =
trane
```

Functions for String Manipulation (1)

Function	Operation
`char`	Converts an integer to the character using ASCII codes, or combines characters into a character matrix
`findstr`	Finds one string in another string
`length`	Returns the number of characters in a string
`num2str`	Converts a number to string
`str2num`	Converts a string to a number
`strcmp`	Compares two strings
`strmatch`	Identifies rows of a character array that begin with a string
`strncmp`	Compares the first n elements of two strings
`sprintf`	Converts strings and numeric values to a string

Functions for String Manipulation (2)

`num2str` converts a number to a string

```
>> msg1 = ['There are ',num2str(100/2.54),' inches in a meter']
msg1 =
There are 39.3701 inches in a meter
```

For greater control over format of the number-to-string conversion, use `sprintf`

```
>> msg2 = sprintf('There are %5.2f cubic inches in a liter',1000/2.54^3)
msg2 =
There are 61.02 cubic inches in a liter
```

The MATLAB `sprintf` function is similar to the C function of the same name, but it uses single quotes for the format string.

Functions for String Manipulation (3)

The `char` function can be used to combine strings

```
>> both = char(msg1,msg2)
both =
There are 39.3701 inches in a meter
There are 61.02 cubic inches in a liter
```

or to refer to individual characters by their ASCII codes[1]

```
>> char(49)
ans =
1
>> char([77 65 84 76 65 66])
ans =
MATLAB
```

[1] See e.g.,, `www.asciicodes.com` or `wikipedia.org/wiki/ASCII`.

Functions for String Manipulation (4)

Use `strcmp` to test whether two strings are equal, i.e., if they contain the same sequence of characters.

```
>> msg1 = ['There are ',num2str(100/2.54),' inches in a meter'];
>> msg2 = sprintf('There are %5.2f cubic inches in a liter',1000/2.54^3);
>> strcmp(msg1,msg2)
ans =
     0
```

Compare the first `n` characters of two strings with `strncmp`

```
>> strncmp(msg1,msg2,9)
ans =
     1
```

The first nine characters of both strings are "`There are`", so `strncmp(msg1,msg2,9)` returns 1, or `true`.

Functions for String Manipulation (5)

Locate occurances of one string in another string with `findstr`

```
>> findstr('in',msg1)
ans =
    19    26

>> msg1(19:20)
ans =
in
```

Polynomials

MATLAB polynomials are stored as vectors of coefficients. The polynomial coefficients are stored in *decreasing powers* of x

$$P_n(x) = c_1 x^n + c_2 x^{n-1} + \ \ldots \ + c_n x + c_{n+1}$$

Example: Evaluate $x^3 - 2x + 12$ at $x = -1.5$

Store the coefficients of the polynomial in vector c:

```
>> c = [1  0  -2  12];
```

Use the built-in `polyval` function to evaluate the polynomial.

```
>> polyval(c,1.5)
ans =
   12.3750
```

Functions for Manipulating Polynomials

Function	Operations performed
`conv`	Product (convolution) of two polynomials
`deconv`	Division (deconvolution) of two polynomials
`poly`	Create a polynomial having specified roots
`polyder`	Differentiate a polynomial
`polyval`	Evaluate a polynomial
`polyfit`	Polynomial curve fit
`roots`	Find roots of a polynomial

A Quick Overview of Linear Algebra in MATLAB

The name "MATLAB" is a shortened form of "MATrix LABoratory".

MATLAB data types and syntax make it easy to perform standard operations of linear algebra including addition, subtraction, and multiplication of vectors and matrices.

Here we provide a simple introduction to some operations that are necessary for routine calculation.

- Vector addition and subtraction
- Inner and outer products
- Vectorization
- Array operators

Vector Addition and Subtraction

Vector and addition and subtraction are element-by-element operations.

Example:

```
>> u = [10 9 8];                    (u and v are row vectors)
>> v = [1 2 3];
>> u+v
ans =
    11    11    11

>> u-v
ans =
     9     7     5
```

Vector Inner and Outer Products

The inner product combines two vectors to form a scalar

$$\sigma = u \cdot v = u\,v^T \Longleftrightarrow \sigma = \sum u_i\,v_i$$

The outer (cross) product combines two vectors to form a matrix

$$A = u^T v \Longleftrightarrow a_{i,j} = u_i\,v_j$$

Inner and Outer Products in MATLAB

Inner and outer products are supported in MATLAB as natural extensions of the multiplication operator

```
>> u = [10 9 8];            (u and v are row vectors)
>> v = [1 2 3];
>> u*v'                     (inner product)
ans =
    52

>> u'*v                     (outer product)
ans =
    10    20    30
     9    18    27
     8    16    24
```

Vectorization

- **Vectorization** is the use of single, compact expressions that operate on all elements of a vector without explicitly writing the code for a loop. The loop *is* executed by the MATLAB kernel, which is much more efficient at evaluating a loop in interpreted MATLAB code.
- Vectorization allows calculations to be expressed succintly so that programmers get a high level (as opposed to detailed) view of the operations being performed.
- Vectorization is important to make MATLAB operate efficiently.

Vectorization of Built-in Functions

Most built-in function support *vectorized* operations. If the input is a scalar the result is a scalar. If the input is a vector or matrix, the output is a vector or matrix with the same number of rows and columns as the input.

Example:

```
>> x = 0:pi/4:pi                    (define a row vector)
x =
         0    0.7854    1.5708    2.3562    3.1416

>> y = cos(x)                       (evaluate cosine of each x(i))
y =
    1.0000    0.7071         0   -0.7071   -1.0000
```

Contrast with C++ Implementation

The MATLAB statements

```
x = 0:pi/4:pi;
y = cos(x);
```

are equivalent to the following C++ code

```
double x[5],y[5];
const double pi=3.14159624;
const double dx = pi/4.0;
for (int i=0; i<5, i++) {
    x[i] = i*dx;
    y[i] = sin(x[i]);}
```

No explicit loop is necessary in MATLAB.

Vectorized Calculations (6)

More examples

```
>> A = pi*[ 1 2; 3 4]
A =
    3.1416    6.2832
    9.4248   12.5664

>> S = sin(A)
S =
     0     0
     0     0
```

```
>> B = A/2
B =
    1.5708    3.1416
    4.7124    6.2832

>> T = sin(B)
T =
     1     0
    -1     0
```

Array Operators

Array operators support element-by-element operations that are not defined by the rules of linear algebra.

Array operators have a period prepended to a standard operator.

Symbol	Operation
`.*`	element-by-element multiplication
`./`	element-by-element "right" division
`.\`	element-by-element "left" division
`.^`	element-by-element exponentiation

Array operators are a very important tool for writing vectorized code.

Using Array Operators (1)

Examples: Element-by-element multiplication and division

```
>> u = [1 2 3];
>> v = [4 5 6];
```

Use `.*` and `./` for element-by-element multiplication and division

```
>> w = u.*v
w =
     4    10    18

>> x = u./v
x =
    0.2500    0.4000    0.5000
```

Using Array Operators (1)

Examples: Element-by-element multiplication and division

```
>> u = [1 2 3];
>> v = [4 5 6];
>> y = sin(pi*u/2) .* cos(pi*v/2)
y =
     1     0     1

>> z = sin(pi*u/2) ./ cos(pi*v/2)

Warning: Divide by zero.
z =
     1   NaN     1
```

Using Array Operators (2)

Examples: Application to matrices

```
>> A = [1 2 3 4; 5 6 7 8];
>> B = [8 7 6 5; 4 3 2 1];
>> A.*B
ans =
     8    14    18    20
    20    18    14     8

>> A*B
??? Error using ==> *
Inner matrix dimensions must agree.
```

The last statement causes an error because the number of columns in `A` is not equal to the number of rows in `B` — a requirement for `A` and `B` to be compatible for matrix multiplication.

Using Array Operators (3)

```
>> A = [1 2 3 4; 5 6 7 8];
>> B = [8 7 6 5; 4 3 2 1];
>> A*B'
ans =
    60    20
   164    60
```

The number of columns in `A` is equal to the number of rows in $\mathtt{B}^T$, so `A*B'` is a legal matrix-matrix multiplication.

Array operators also apply to matrix powers.

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

The MATLAB Workspace (1)

All variables defined as the result of entering statements in the command window, exist in the MATLAB *workspace*.

At the beginning of a MATLAB session, the workspace is empty.

Being aware of the workspace allows you to

- Create, assign, and delete variables
- Load data from external files
- Manipulate the MATLAB path

The MATLAB Workspace (2)

The `clear` command deletes variables from the workspace. The `who` command lists the names of variables in the workspace

```
>> clear              (Delete all variables from the workspace)
>> who
                      (No response, no variables are defined after 'clear')

>> a = 5;   b = 2;  c = 1;
>> d(1) = sqrt(b^2 - 4*a*c);
>> d(2) = -d(1);
>> who
Your variables are:

a         b         c         d
```

The MATLAB Workspace (3)

The `whos` command lists the name, size, memory allocation, and the class of each variables defined in the workspace.

```
>> whos

  Name         Size            Bytes  Class

  a            1x1                 8  double array
  b            1x1                 8  double array
  c            1x1                 8  double array
  d            1x2                32  double array (complex)

Grand total is 5 elements using 56 bytes
```

The MATLAB Workspace (4)

The `whos` command returns the array dimensions, memory requirement and class for each variable in the workspace.

Built-in variable classes are `double`, `char`, `sparse`, `struct`, and `cell`. The class of a variable determines the type of data that can be stored in it. We will be dealing primarily with numeric data, which is the `double` class, and occasionally with string data, which is in the `char` class.

Working with External Data Files

Write data to a file

```
save fileName
save fileName variable1 variable2 ...
save fileName variable1 variable2 ... -ascii
```

Read in data stored in matrices

```
load fileName
load fileName matrixVariable
```

Loading Data from External File

Example: Load data from a file and plot the data

```
>> load wolfSun.dat;
>> xdata = wolfSun(:,1);
>> ydata = wolfSun(:,2);
>> plot(xdata,ydata)
```

The MATLAB Path

MATLAB will only use those functions and data files that are in its path.

To add `N:\IMAUSER\ME352\PS2` to the path, type

```
>> p = path;
>> path(p,'N:\IMAUSER\ME352\PS2');
```

MATLAB version 5 and later has an interactive path editor that makes it easy to adjust the path.

The path specification string depends on the operating system. On a Unix/Linux computer a path setting operation might look like:

```
>> p = path;
>> path(p,'~/matlab/ME352/ps2');
```

Plotting

- Plotting (x, y) data
- Axis scaling and annotation
- 2D (contour) and 3D (surface) plotting

Plotting (x, y) Data (1)

Two dimensional plots are created with the `plot` function

Syntax:

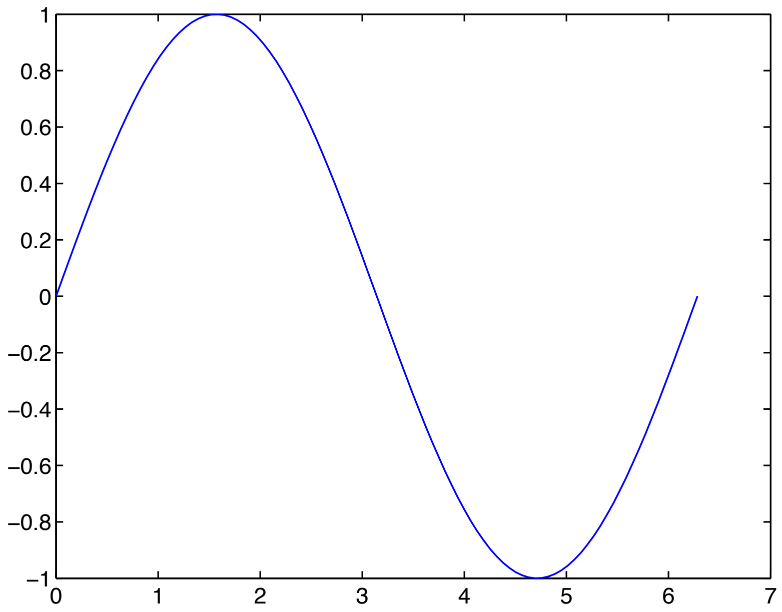
```
plot(x,y)
plot(xdata,ydata,symbol)
plot(x1,y1,x2,y2,...)
plot(x1,y1,symbol1,x2,y2,symbol2,...)
```

Note: `x` and `y` must have the same *shape*, `x1` and `y1` must have the same *shape*, `x2` and `y2` must have the same *shape*, etc.

Plotting (x, y) Data (2)

Example: A simple line plot

```
>> x = linspace(0,2*pi);
>> y = sin(x);
>> plot(x,y);
```



Line and Symbol Types (1)

The curves for a data set are drawn from combinations of the color, symbol, and line types in the following table.

Color		Symbols				Line	
`y`	yellow	`.`	point	`^`	triangle (up)	`-`	solid
`m`	magenta	`o`	circle	`<`	triangle (left)	`:`	dotted
`c`	cyan	`x`	x-mark	`>`	triangle (right)	`-.`	dashdot
`r`	red	`+`	plus	`p`	pentagram	`--`	dashed
`g`	green	`*`	star	`h`	hexagram		
`b`	blue	`s`	square				
`w`	white	`d`	diamond				
`k`	black	`v`	triangle (down)				

To choose a color/symbol/line style, chose *one* entry from each column.

Line and Symbol Types (2)

Examples:

Put yellow circles at the data points:

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

Plot a red dashed line with no symbols:

```
plot(x,y,'r--')
```

Put black diamonds at each data point and connect the diamonds with black dashed lines:

```
plot(x,y,'kd--')
```

Alternative Axis Scaling (1)

Combinations of linear and logarithmic scaling are obtained with functions that, other than their name, have the same syntax as the `plot` function.

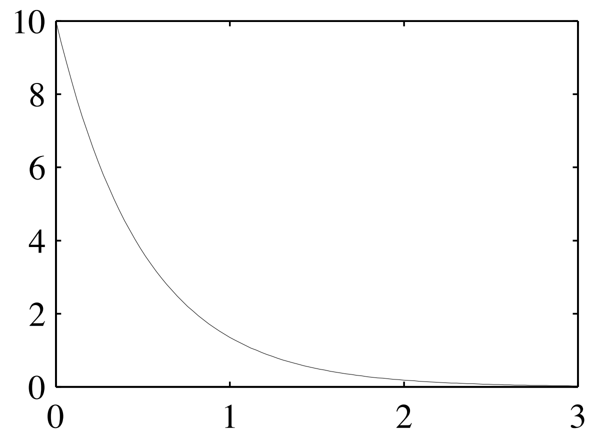
Name	Axis scaling
`loglog`	$\log_{10}(y)$ versus $\log_{10}(x)$
`plot`	linear y versus x
`semilogx`	linear y versus $\log_{10}(x)$
`semilogy`	$\log_{10}(y)$ versus linear x

Note: As expected, use of logarithmic axis scaling for data sets with negative or zero values results in a error. MATLAB will complain and then plot only the positive (nonzero) data.

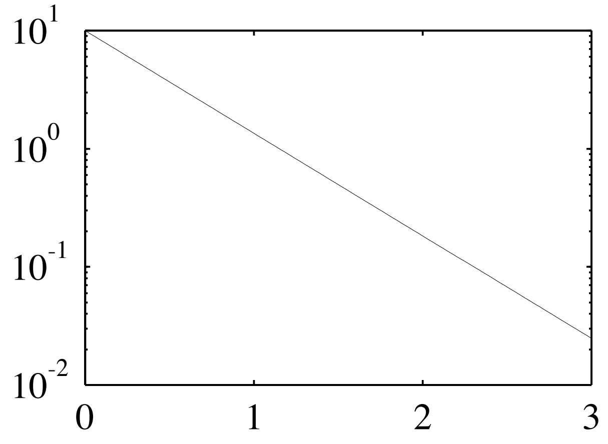
Alternative Axis Scaling (2)

Example:

```
>> x = linspace(0,3);
>> y = 10*exp(-2*x);
>> plot(x,y);
```



```
>> semilogy(x,y);
```



Multiple plots per figure window (1)

The `subplot` function is used to create a matrix of plots in a single figure window.

Syntax:

```
subplot(nrows,ncols,thisPlot)
```

Repeat the values of *`nrows`* and *`ncols`* for all plots in a single figure window. Increment *`thisPlot`* for each plot

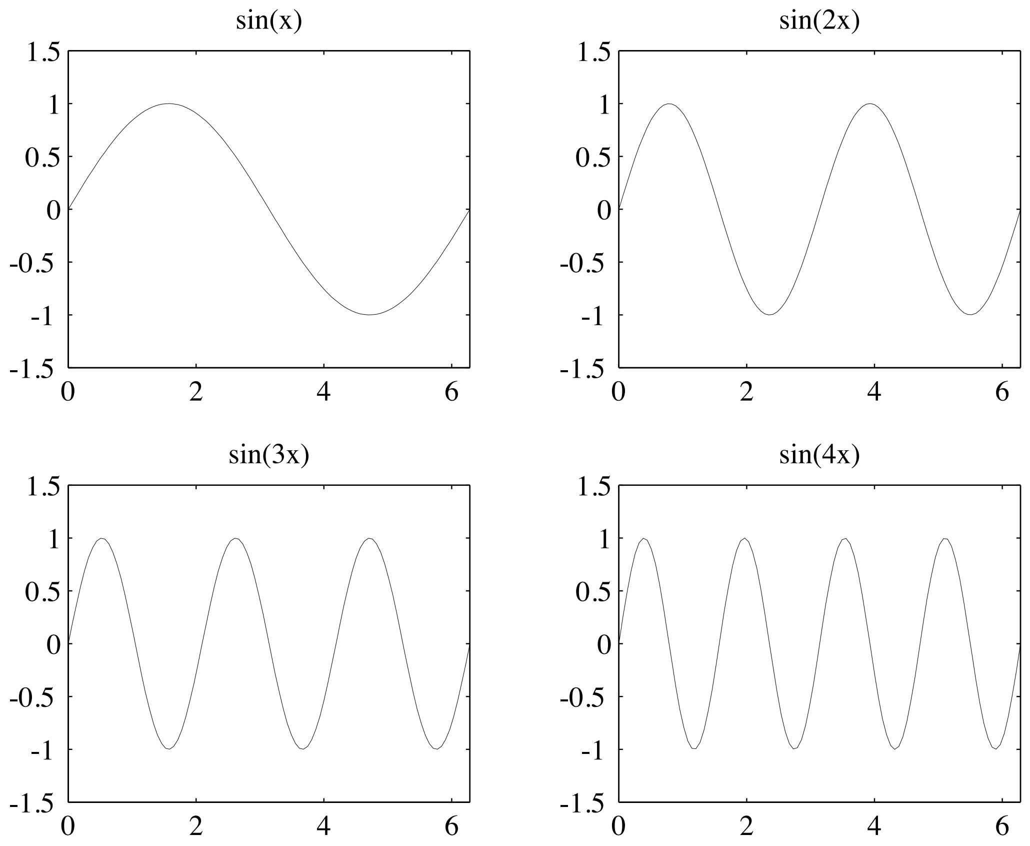
Multiple plots per figure window (1)

Example:

```
>> x = linspace(0,2*pi);
>> subplot(2,2,1);
>> plot(x,sin(x));     axis([0 2*pi -1.5 1.5]);   title('sin(x)');

>> subplot(2,2,2);
>> plot(x,sin(2*x));   axis([0 2*pi -1.5 1.5]);   title('sin(2x)');

>> subplot(2,2,3);
>> plot(x,sin(3*x));   axis([0 2*pi -1.5 1.5]);   title('sin(3x)');

>> subplot(2,2,4);
>> plot(x,sin(4*x));   axis([0 2*pi -1.5 1.5]);   title('sin(4x)');
```

(See next slide for the plot.)

Multiple plots per figure window (2)



Plot Annotation

Name	Operation(s) performed
`axis`	Reset axis limits
`grid`	Draw grid lines at the major ticks marks on the x and y axes
`gtext`	Add text to a location determined by a mouse click
`legend`	Create a legend to identify symbols and line types when multiple curves are drawn on the same plot
`text`	Add text to a specified (x, y) location
`xlabel`	Label the x-axis
`ylabel`	Label the y-axis
`title`	Add a title above the plot

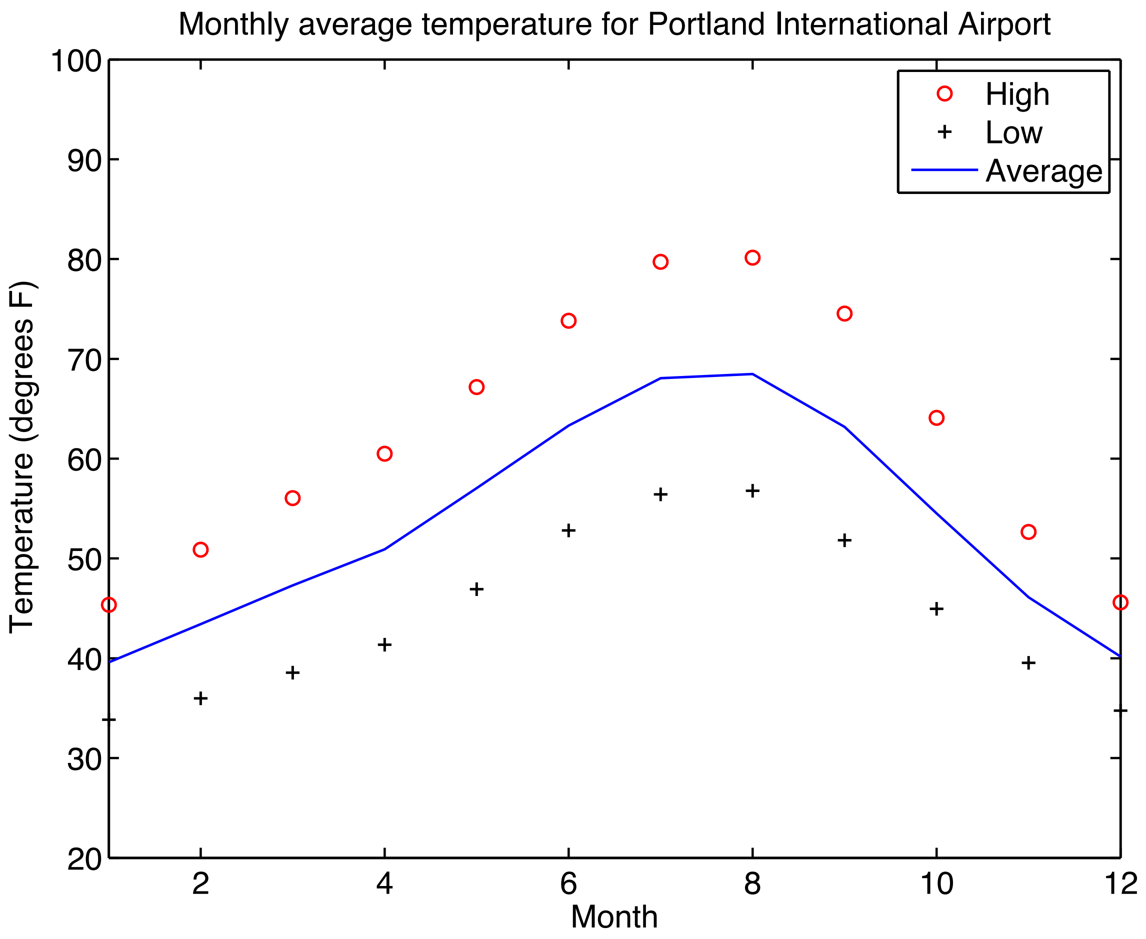
Plot Annotation Example (1)

```
>> D = load('pdxTemp.dat');   m = D(:,1);   T = D(:,2:4);

>> plot(m,t(:,1),'ro',m,T(:,2),'k+',m,T(:,3),'b-');
>> xlabel('Month');
>> ylabel('Temperature ({}^\circ F)');
>> title('Monthly average temperature at PDX');
>> axis([1 12 20 100]);
>> legend('High','Low','Average',2);
```

Note: The `pdxTemp.dat` file is in the `data` directory of the NMM toolbox. Make sure the toolbox is installed and is included in the MATLAB path.

(See next slide for the plot.)

Plot Annotation Example (2)


Monthly average temperature for Portland International Airport
Temperature (degrees F)
Month
High
Low
Average