Interactive Computing with MATLAB

Gerald W. Recktenwald
Department of Mechanical Engineering
Portland State University
gerry@me.pdx.edu

These slides are a supplement to the book *Numerical Methods with MATLAB: Implementations and Applications*, by Gerald W. Recktenwald, © 2002, Prentice-Hall, Upper Saddle River, NJ. These slides are © 2002 Gerald W. Recktenwald. The PDF version of these slides may be downloaded or stored or printed only for noncommercial, educational use. The repackaging or sale of these slides in any form, without written consent of the author, is prohibited.

The latest version of this PDF file, along with other supplemental material for the book, can be found at `www.prenhall.com/recktenwald`.

Version 1.02 October 2, 2002

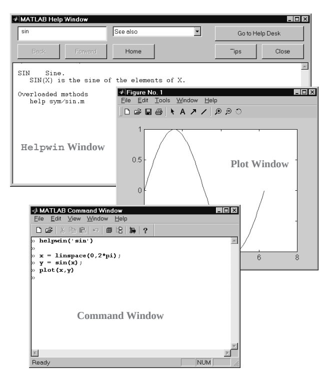
Overview

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

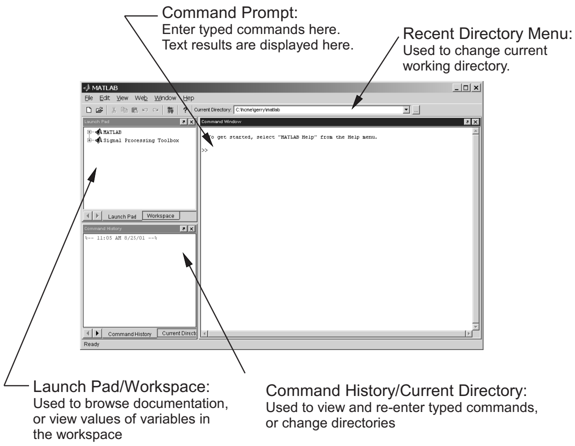
Starting MATLAB

- 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 Windows (version 5)



MATLAB Workspace (version 6)



MATLAB as a Calculator

Enter formulas at the command prompt

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

>> ans/2
ans =
     2
```

Or, 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

>> log10(256)
ans =
     2.4082

>> log2(256)
ans =
     8
```

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.
```


Ways to Get Help

- Use on-line help to request info on a specific function
  ```
  >> help sqrt
  ```
- The `helpwin` function opens a separate window for the *help browser*
  ```
  >> helpwin('sqrt')
  ```
- Use `lookfor` to find functions by keywords
  ```
  >> lookfor functionName
  ```
- 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
  ```


On-line Help

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.
```


Suppress Output with Semicolon

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
```

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

```
>> 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_baubble
mySum
mysum
```

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

Built-in MATLAB Variables

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 $0/0$

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 MATLAB vector is a matrix with one row *or* one column
A MATLAB scalar is a matrix with one row *and* one column

Overview of Working with matrices and vectors

- Creating vectors:
  `linspace` and `logspace`
- Creating matrices:
  `ones`, `zeros`, `eye`, `diag`, . . .
- Subscript notation
- Colon notation
- Vectorization

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 that evaluates to a matrix, vector or scalar.

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

Manual Entry

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
```

In a 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

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

>> v'
ans =
     2
     4
     1
     7

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

>> A'
ans =
     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
```

Creating vectors with `linspace`

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);

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

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

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 $10^{\text{startValue}}$ 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

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


Functions to Create Matrices (5)

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 (6)

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
```

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

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 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
```


Subscript Notation (2)

Referring to elements outside of current matrix dimensions results in an error

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

Assigning an elements outside of current matrix dimensions 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
```

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

Syntax:

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

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

Colon Notation (2)

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
```

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 (3)

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
```

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 =
ans =
     2     3
     5     6
```


Colon Notation (4)

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 (5)

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

Examples:

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

>> 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

The basic MATLAB variable is a matrix — a two dimensional array of values. The elements of a matrix variable can either be numeric values or characters. If the elements are numeric values they 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.

We now 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

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

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

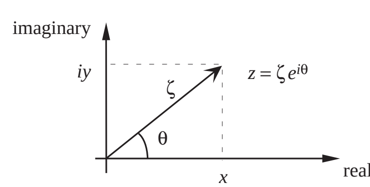
Both or either `i` and `j` can be reassigned

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

>> u*u
ans =
   -3.0000

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

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

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 measured 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`	convert an integer to the character using ASCII codes, or combine 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)

Examples:

```
>> msg1 = ['There are ',num2str(100/2.54),' inches in a meter']
message1 =
There are 39.3701 inches in a meter

>> msg2 = sprintf('There are %5.2f cubic inches in a liter',1000/2.54^3)
message2 =
There are 61.02 cubic inches in a liter

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

>> strcmp(msg1,msg2)
ans =
     0

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

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


Polynomials

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

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

Example:

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

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

Manipulation of Matrices and Vectors

The name "MATLAB" evolved as an abbreviation of "MATrix LABoratory". The data types and syntax used by MATLAB make it easy to perform the standard operations of linear algebra including addition and subtraction, multiplication of vectors and matrices, and solving linear systems of equations.

Chapter 7 provides a detailed review of linear algebra. 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 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 executing a loop. The loop *is* executed by the MATLAB kernel, which is much more efficient at looping than 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 Fortran implementation:

```
    real x(5),y(5)
    pi = 3.14159624
    dx = pi/4.0
    do 10 i=1,5
        x(i) = (i-1)*dx
        y(i) = sin(x(i))
10  continue
```

No explicit loop is necessary in MATLAB.

Vector Calculations (3)

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 are designated by a period prepended to the 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];
>> w = u.*v                     (element-by-element product)
w =
     4    10    18

>> x = u./v                     (element-by-element division)
x =
    0.2500    0.4000    0.5000

>> 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.

>> A*B'
ans =
    60    20
   164    60

>> 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
```

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,...)
```

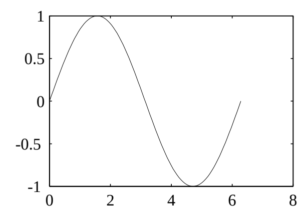
Note: *x* and *y* must have the same *shape*, *x1* and *y1* must have the same *shape*, *x2* and *y2* must have the same *shape*, etc.

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

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.

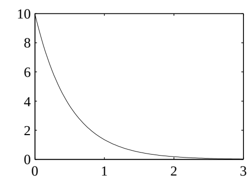
Name	Axis scaling
`loglog`	$\log_{10}(y)$ versus $\log_{10}(x)$
`plot`	linear y versus x
`semilogx`	linear y versus $\log_{10}(x)$
`semilogy`	$\log_{10}(y)$ versus linear x

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.

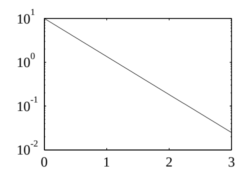
Alternative Axis Scaling (2)

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`*`)`

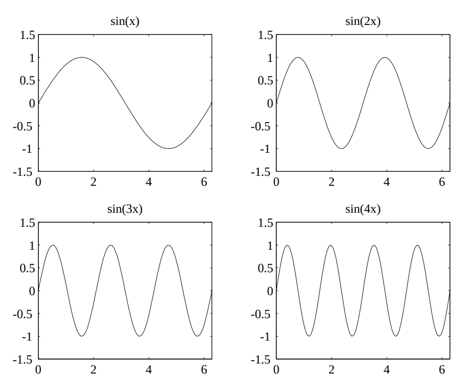
Repeat the values of *`nrows`* and *`ncols`* for all plots in a single figure window. Increment *`thisPlot`* for each plot

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)

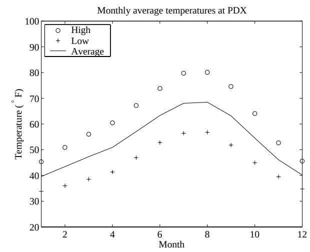


Plot Annotation

Name	Operation(s) performed
`axis`	Reset axis limits
`grid`	Draw grid lines corresponding to the major major ticks 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

```
>> 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.