Web Appendix A - Introduction to MATLAB

MATLAB is a high-level computer-based mathematical tool for performing complex or repetitive calculations under program control. MATLAB can generate or read numerical data and store or display results of analysis of data. This tutorial will introduce the features of MATLAB most often used for signal and system analysis. It is by no means an exhaustive exploration of all the capabilities of MATLAB.

The easiest way to become familiar with MATLAB is to sit at a console and experiment with various operators, functions and commands until their properties are understood and then later to progress to writing MATLAB scripts and functions to execute a sequence of instructions to accomplish an analytical goal.

A logical progression in learning how to use MATLAB to solve signal and system problems is to understand

1. What kinds of numbers can be represented,
2. How variables are named and how values are assigned to them,
3. What mathematical operators are built into MATLAB and how they operate on numbers and variables,
4. What mathematical functions are intrinsic to MATLAB and how to get descriptions of their syntax and use,
5. How to write a script file which is a list of instructions which are executed sequentially or according to flow-control commands,
6. How to write function files which contain user-defined functions which can be used just like intrinsic functions,

and 7. How to display and format results of calculations in a graphical manner that aids quick comprehension of relationships among variables.

A.1 Numbers, Variables and Matrices

MATLAB is vector and matrix oriented. That is, everything in MATLAB is a matrix. A row vector with *m* elements is a 1 by *m* matrix and a column vector with *m* elements is an *m* by 1 matrix. A scalar is a 1 by 1 matrix. MATLAB can handle real numbers or complex numbers. For example, the real number 7.89 is represented as simply `7.89` in MATLAB. The real number, 15.8×10^{-11} can be written in MATLAB as `15.8e-11`. The complex number whose real part is 8 and whose imaginary part is 3 can be represented either as `8+3*i` or as `8+j*3` because the two letters, `i` and `j` are both pre-assigned by MATLAB to be equal to $\sqrt{-1}$. Other constants pre-assigned by MATLAB are

`pi` for π, `inf` for infinity, and `NaN` for not a number.

Any of these pre-defined constants can be re-defined by the user, but, unless the user has a very good reason to change them, they should be left as defined by MATLAB to avoid confusion.

A.2 Operators

MATLAB has the following mathematical operators:

`=`	assignment,	`==`	equality,
`+`	addition,	`-`	subtraction and unary minus,
`*`	matrix multiplication,	`.*`	array multiplication,
`^`	matrix power,	`.^`	array power,
`/`	division,	`./`	array division,
`<>`	relational operators,	`&`	logical AND,
`|`	logical OR,	`~`	logical NOT,

plus

`'` (apostrophe) transpose,

and

`.'` (dot-apostrophe) non-conjugated transpose.

(This is not a complete list, just the most commonly-used operators).

In the examples to follow, illustrating MATLAB features, the **`bold face`** text is what the user types in and the `plain face` text is the MATLAB response. The » character is the MATLAB prompt indicating MATLAB is waiting for user instructions.

Suppose we type in the assignment statement,

```
»a = 2 ;
```

This assigns the value 2 to the scalar variable named `a`. The semicolon, `;`, terminates the instruction and suppresses display of the result of the operation. This statement could also be written as

```
»a=2 ;
```

The extra spaces in `a = 2` compared with `a=2` have no effect. They are ignored by MATLAB. The syntax for assigning values to a row vector is illustrated with the following two assignment statements which are typed on a single line and both terminated by a semicolon.

```
»b = [1 3 2] ; c = [2,5,1] ;
```

The elements of a row vector can be separated by either a space or a comma. Given the assignments already made, if we now simply type the name of a variable MATLAB displays the value.

```
»a
a =
     2
»b
b =
     1     3     2
```

```
»c
c =
     2     5     1
```

Vectors and matrices can be multiplied by a scalar. The result is a vector or matrix whose elements have each been multiplied by that scalar.

```
»a*b
ans =
     2     6     4
```

`ans` is the name given to a result when the user does not assign a name.

```
»a*c
ans =
     4    10     2
```

A scalar can also be added to a vector or matrix.

```
»a + b
ans =
     3     5     4
```

Addition of a scalar and a vector or matrix, adds the scalar to every element of the vector or matrix. Subtraction is similarly defined.

```
»a + c
ans =
     4     7     3
»a - b
ans =
     1    -1     0
»a - c
ans =
     0    -3     1
```

Vectors and matrices add (or subtract) in the way normally defined in mathematics. That is, the two vectors or matrices must have the same shape to be added (or subtracted). (Unless one of them is a 1 by 1 matrix, a scalar, as illustrated above.)

```
»b + c
ans =
     3     8     3
»c - b
ans =
     1     2    -1
```

Vectors and matrices can be multiplied according to the usual rules of linear algebra.

```
»b*c
??? Error using ==> *
Inner matrix dimensions must agree.
```

This result illustrates a common error in MATLAB. Matrices must be *commensurate* to be multiplied using the `*` operator. Premultiplication of a 1 by 3 row vector like `c` by a 1 by 3 row vector like `b` is not defined. But if `c` were transposed to a 3 by 1 column vector, the multiplication would be defined. Transposition is done with the `'` operator

```
»c'
ans =
     2
     5
     1
»b*c'
ans =
    19
```

This is the product, bc^T.

```
»b'*c
ans =
     2     5     1
     6    15     3
     4    10     2
```

This is the product, $b^T c$.

Often it is very useful to multiply two vectors or matrices of the same shape element by element instead of using the usual rules of matrix multiplication. That kind of multiplication is called *array multiplication* in MATLAB and is done using the operator `.*` .

```
»b.*c
ans =
     2    15     2
```

Now define a 3 by 3 matrix, `A`.

```
»A = [3 5 1 ; 9 -1 2 ; -7 -4 3] ;
```

In this context, the `;` operator puts the next entry in the next row of the matrix. So the `;` operator has a dual use, to terminate instructions and to enter the next row when specifying a matrix.

```
»A
A =
     3     5     1
     9    -1     2
    -7    -4     3
```

Two-dimensional matrices are displayed in rows and columns. As indicated above we can multiply a matrix by a scalar.

```
»a*A
ans =
     6    10     2
    18    -2     4
   -14    -8     6
```

We can also multiply a matrix and a vector, if they are commensurate.

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

```
»A*b'
ans =
    20
    10
   -13
```

Now define two more 3 by 3 matrices.

```
»B = [3 2 7 ; 4 1 2 ; -1 3 1] ;
»C = [4 5 5 ; -1 -3 2 ; 8 3 1] ;
»B
B =
     3     2     7
     4     1     2
    -1     3     1
»C
C =
     4     5     5
    -1    -3     2
     8     3     1
»B + C
ans =
     7     7    12
     3    -2     4
     7     6     2
»B*C
ans =
    66    30    26
    31    23    24
     1   -11     2
»B.*C
ans =
    12    10    35
    -4    -3     4
    -8     9     1
```

Another important operator in MATLAB is the ^ operator for raising a number or variable to a power. We can raise a scalar to a power

```
»a^2
ans =
     4
```

We can raise a matrix to a power according to the rules of linear algebra.

```
»A^2
ans =
    47     6    16
     4    38    13
   -78   -43    -6
```

We can raise each element of a matrix to a power.

```
»A.^2
ans =
     9    25     1
    81     1     4
    49    16     9
```

Sometimes it is desired to know what variables are currently defined and have values stored by MATLAB. The `who` command accomplishes that.

```
»who
Your variables are:
A         C         ans       c
B         a         b
```

If you want to undefine a variable (take it out of the list MATLAB recognizes) the `clear` command accomplishes that.

```
»clear A
»A
???  Undefined function or variable.  Symbol in question ==> A
```

We can redefine `A` by a matrix multiplication.

```
»A = B*C
A =
    66    30    26
    31    23    24
     1   -11     2
```

Another important operation is division. We can divide a scalar by a scalar.

```
»a/3
ans =
    0.6667
```

We can divide a vector by a scalar.

```
»b/a
ans =
    0.5000    1.5000    1.0000
```

We can divide a matrix by a scalar.

```
»A/10
ans =
    6.6000    3.0000    2.6000
    3.1000    2.3000    2.4000
    0.1000   -1.1000    0.2000
```

If we divide anything by a vector or matrix we must choose which operator to use. The `/` operator can be used to divide one matrix by another in the sense that the MATLAB operation, `A/B` is equivalent to the mathematical operation, $\mathbf{AB}^{-1}$, where $\mathbf{B}^{-1}$ is the matrix inverse of $\mathbf{B}$. Of course, $\mathbf{A}$ and $\mathbf{B}^{-1}$ must be commensurate for `A/B` to work properly.

```
»A/B
ans =
   -2.6875   19.8125    5.1875
    0.2969    8.6719    4.5781
    1.3594   -1.7656   -3.9844
```

We can also do array division in which each element of the numerator is divided by the corresponding element of the denominator.

```
»A./B
ans =
   22.0000   15.0000    3.7143
    7.7500   23.0000   12.0000
   -1.0000   -3.6667    2.0000
```

The array division operator, `./` can also be used to divide a scalar by a matrix in the sense that a new matrix is formed, each element of which is the scalar divided by the corresponding element of the dividing matrix.

```
»10./A
ans =
    0.1515    0.3333    0.3846
    0.3226    0.4348    0.4167
   10.0000   -0.9091    5.0000
```

There is an array of *relational* and *logical* operators to compare numbers and to make logical decisions. The == operator compares two numbers and returns a logical result which is 1 if the two numbers are the same and 0 if they are not. This operator can be used on scalars, vectors or matrices, or on scalars with vectors or scalars with matrices. If two vectors are compared or if two matrices are compared they must have the same shape. The comparison is done on an element-by-element basis.

```
»1 == 2
ans =
     0

»5 == 5
ans =
     1

»a == b
ans =
     0     0     1

»a == B
ans =
     0     1     0
     0     0     1
     0     0     0

»B == C
ans =
     0     0     0
     0     0     1
     0     1     1
```

The operator, ~= compares two numbers and returns a 1 if they are not equal and a 0 if they are equal.

The operator, >, compares two numbers and returns a 1 if the first number is greater than the second and a 0 if it is not.

The operator, < compares two numbers and returns a 1 if the first number is less than the second and a 0 if it is not.

The operator, >= compares two numbers and returns a 1 if the first number is greater than or equal to the second and a 0 if it is not.

The operator, <= compares two numbers and returns a 1 if the first number is less than or equal to the second and a 0 if it is not.

All of these relational operators can be used with scalars, vectors and matrices in the same way that the operator, ==, can.

```
»-3 ~= 7
ans =
     1

»B ~= C
ans =
     1     1     1
     1     1     0
     1     0     0

»c > 2
ans =
     0     1     0

»B < 2
ans =
     0     0     0
     0     1     0
     1     0     1

»B <= 2
ans =
     0     1     0
     0     1     1
     1     0     1

»B >= 2
ans =
     1     1     1
     1     0     1
     0     1     0

»B > 2
```

```
ans =
     1     0     1
     1     0     0
     0     1     0

»b < c
ans =
     1     1     0

»b < B
??? Error using ==> <
Matrix dimensions must agree.
```

MATLAB has three logical operators, `&` (logical AND), `|` (logical OR) and `~` (logical NOT). Logical operators operate on any numbers. The number, 0, is treated as a logical 0 and any other number is treated as a logical 1 for purposes of the logical operation. The result of any logical operation is either the number 0 or the number 1.

The operator, `&`, returns a 1 if both operands are non-zero and a 0 otherwise.

The operator, `|`, returns a 0 if both operands are zero and a 1 otherwise.

The operator, `~`, is a unary operator which returns a 1 if its operand is a 0 and a 0 if its operand is non-zero.

```
»0 & 1
ans =
     0

»1 & 1
ans =
     1

»1 & [1 0 -3 22]
ans =
     1     0     1     1

»0 | [3 ; 0 ; -18]
ans =
     1
     0
     1

»~[0 ; 1 ; 0.3]
ans =
     1
     0
     0

»(1 < 2 ) | (-5 > -1)
ans =
     1

»(1 < 2 ) & (-5 > -1)
ans =
      0
```

One of the most powerful operators in MATLAB is the : operator. This operator can be used to generate sequences of numbers and it can also be used to select only certain rows and/or columns of a matrix. When used in the form, `a:b`, where `a` and `b` are scalars, this operator generates a sequence of numbers from `a` to `b` separated by one.

```
»3:8
ans =
      3      4      5      6      7      8
```

When used in the form, `a:b:c` where `a`, `b` and `c` are scalars, it generates a sequence of numbers from `a` to `c` separated by `b`.

```
»-4:3:17
ans =
     -4     -1      2      5      8     11     14     17

»-4:3:16
ans =
     -4     -1      2      5      8     11     14

»(2:-3:-11)'
ans =
      2
     -1
     -4
     -7
    -10
```

If the increment, `b`, is positive, the sequence terminates at the last value that is less than or equal to the specified end value, `c`. If the increment, `b`, is negative, the sequence terminates at the last value that is greater than or equal to the specified end value, `c`.

Another use of the : operator is to form a column vector consisting of all the elements of a matrix. For example, the instruction, `A(:)`, forms a column vector of the elements in `A`.

```
» A
A =
     66     30     26
     31     23     24
      1    -11      2

»A(:)
ans =
     66
     31
      1
     30
     23
    -11
     26
     24
      2
```

Another use of the : operator is to extract a submatrix from a matrix. For example, `A(:,2)` forms a matrix which is the second column of `A`.

```
»A(:,2)

ans =

    30
    23
   -11
```

The instruction `A(3,:)` forms a matrix which is the third row of `A`.

```
»A(3,:)
ans =
     1   -11     2
```

We can also extract partial rows and partial columns or combinations of those.

```
»A(1:2,3)
ans =
    26
    24

»A(1:2,:)
ans =
    66    30    26
    31    23    24

»A(1:2,2:3)
ans =
    30    26
    23    24
```

For a complete listing of MATLAB operators consult a MATLAB reference manual.

A.3 Scripts and Functions

The real power of any programming language lies in writing sequences of instructions to implement some algorithm. There are two types of programs in MATLAB, scripts and functions. Both types of programs are stored on disk as `.m` files, so called because the extension is `.m` . A script is what is normally called just a program. It is a sequence of instructions that is to executed sequentially except when flow control statements change the sequence. For example,

```
.
.
.
f01 = 6 ; f02 = 9 ; f0 = gcd(f01,f02) ; T0 = 1/f0 ;
T01 = 1/f01 ; T02 = 1/f02 ; T0min = min(T01,T02) ; dt = T0min/24 ;
nPts = 2*T0/dt ; t = dt*[0:nPts]' ;
th1 = (rand(1,1)-0.5)*2*pi ; x1 = cos(2*pi*f01*t + th1) ;
th2 = (rand(1,1)-0.5)*2*pi ; x2 = cos(2*pi*f02*t + th2) ;
.
```

.
.

Any text editor can be used to write a script file or a function file.

A function is a modular program which accepts arguments and returns results and is usually intended to be reusable in a variety of situations. The arguments passed to the function can be scalars, vectors or matrices and the results returned can also be scalars, vectors or matrices.

```
%      Chi-squared probability density function
%
%      N     degrees of freedom
%      xsq   chi-squared
function y = pchisq(N,xsq)
      y = xsq.^(N/2-1).*exp(-xsq/2).*u(xsq)/(2^(N/2)*gamma(N/2)) ;

%      Rectangular to polar conversion
%
%      x and y are the rectilinear components of a vector
%      r and theta are the magnitude and angle of the vector
function [r,theta] = rect2polar(x,y)
      r = sqrt(x^2 + y^2) ; theta = atan2(y,x) ;
```

The first executable line in a function must begin with the keyword, `function`. The variable representing the value returned by the function is next, in this case, `y`. The name of the function is next, in this case, `pchisq`. Then any parameters to be passed to the function are included in parentheses and separated by commas, in this case, `N,xsq`.

Any line in a script or function which begins with the character, `%` is a *comment* line and is ignored by MATLAB in executing the program. It is there for the benefit of anyone who reads the program to help in following the algorithm the program implements.

A.4 MATLAB Functions and Commands

MATLAB has a long list of intrinsic functions and commands to perform common mathematical tasks. In this tutorial the word, function, will be used for those MATLAB entities which are, or are similar to, mathematical functions. The word, command, will be used for other operations like plotting, input-output, formatting, etc... The word, instruction, will refer to a sequence of functions, commands and/or operations that work as a unit and are terminated by a "`;`". The word, operation, will refer to what is done by operators like +, -, *, /, ^, etc... The functions and commands are divided into groups. The exact groups and the number of groups varies slightly between platforms and between versions of MATLAB but the following groups are common to all recent versions of MATLAB.

general	-	General purpose commands.
ops	-	Operators and special characters.
lang	-	Programming language constructs.

elmat	-	Elementary matrices and matrix manipulation.
elfun	-	Elementary math functions.
specfun	-	Specialized math functions.
matfun	-	Matrix functions - numerical linear algebra.
datafun	-	Data analysis and Fourier transforms.
polyfun	-	Interpolation and polynomials.
funfun	-	Function functions and ODE solvers.
sparfun	-	Sparse matrices.
graph2d	-	Two dimensional graphs.
graph3d	-	Three dimensional graphs.
specgraph	-	Specialized graphs.
graphics	-	Handle Graphics.
uitools	-	Graphical user interface tools.
strfun	-	Character strings.
iofun	-	File input/output.
timefun	-	Time and dates.
datatypes	-	Data types and structures.

In addition to these groups there are *toolboxes* available from MATLAB which supply additional special-purpose functions. Three of the most common toolboxes are the *symbolic*, *signal* and *control* toolboxes. The symbolic toolbox adds capabilities for MATLAB to do symbolic, as opposed to numerical, operations. Examples would be differentiation, integration, solving systems of equations, integral transforms, etc.... The signal toolbox adds functions and commands which do common signal processing operations like waveform generation, filter design and implementation, numerical transforms, statistical analysis, windowing, parametric modeling, etc.... The control toolbox adds functions and commands which do common control-system operations like creation of LTI system models, state-space system description and analysis, responses to standard signals, pole-zero diagrams, frequency-response plots, root locus, etc.... The *ops* group of operators has already been discussed. This tutorial will cover only those function and command groups and functions and commands within those groups that are the most important in elementary signal and system analysis.

A.4.1 General Purpose Commands

The *help* command may be the most important in MATLAB. It is a convenient way of getting documentation on all MATLAB commands. If the user simply types help he gets a list of function and command groups available. For example,

```
»help

HELP topics:

Toolbox:symbolic       -   Symbolic Math Toolbox.
Toolbox:signal         -   Signal Processing Toolbox.
Toolbox:control        -   Control System Toolbox.
```

```
matlab:general         -   General purpose commands.
matlab:ops             -   Operators and special characters.
matlab:lang            -   Programming language constructs.
matlab:elmat           -   Elementary matrices and matrix manipulation.
matlab:elfun           -   Elementary math functions.
matlab:specfun         -   Specialized math functions.
matlab:matfun          -   Matrix functions - numerical linear algebra.
matlab:datafun         -   Data analysis and Fourier transforms.
matlab:polyfun         -   Interpolation and polynomials.
matlab:funfun          -   Function functions and ODE solvers.
matlab:sparfun         -   Sparse matrices.
matlab:graph2d         -   Two dimensional graphs.
matlab:graph3d         -   Three dimensional graphs.
matlab:specgraph       -   Specialized graphs.
matlab:graphics        -   Handle Graphics.
matlab:uitools         -   Graphical user interface tools.
matlab:strfun          -   Character strings.
matlab:iofun           -   File input/output.
matlab:timefun         -   Time and dates.
matlab:datatypes       -   Data types and structures.
matlab:demos           -   Examples and demonstrations.
Toolbox:matlab         -   (No table of contents file)
Toolbox:local          -   Preferences.
MATLAB 5:bin           -   (No table of contents file)
MATLAB 5:extern        -   (No table of contents file)
MATLAB 5:help          -   (No table of contents file)

For more help on directory/topic, type "help topic".
```

The exact list will vary somewhat between platforms and MATLAB versions. Then if the user types `help` followed by the name of a group he gets a list of functions and commands in that group. For example,

```
»help  elfun

   Elementary math functions.

   Trigonometric.
     sin         - Sine.
     sinh        - Hyperbolic sine.
     asin        - Inverse sine.
     asinh       - Inverse hyperbolic sine.
     cos         - Cosine.
     cosh        - Hyperbolic cosine.
     acos        - Inverse cosine.
     acosh       - Inverse hyperbolic cosine.
     tan         - Tangent.
     tanh        - Hyperbolic tangent.
     atan        - Inverse tangent.
     atan2       - Four quadrant inverse tangent.
     atanh       - Inverse hyperbolic tangent.
     sec         - Secant.
     sech        - Hyperbolic secant.
     asec        - Inverse secant.
     asech       - Inverse hyperbolic secant.
     csc         - Cosecant.
     csch        - Hyperbolic cosecant.
```

```
    acsc        - Inverse cosecant.
    acsch       - Inverse hyperbolic cosecant.
    cot         - Cotangent.
    coth        - Hyperbolic cotangent.
    acot        - Inverse cotangent.
    acoth       - Inverse hyperbolic cotangent.

  Exponential.
    exp         - Exponential.
    log         - Natural logarithm.
    log10       - Common (base 10) logarithm.
    log2        - Base 2 logarithm and dissect floating point number.
    pow2        - Base 2 power and scale floating point number.
    sqrt        - Square root.
    nextpow2    - Next higher power of 2.

  Complex.
    abs         - Absolute value.
    angle       - Phase angle.
    conj        - Complex conjugate.
    imag        - Complex imaginary part.
    real        - Complex real part.
    unwrap      - Unwrap phase angle.
    isreal      - True for real array.
    cplxpair    - Sort numbers into complex conjugate pairs.

  Rounding and remainder.
    fix         - Round towards zero.
    floor       - Round towards minus infinity.
    ceil        - Round towards plus infinity.
    round       - Round towards nearest integer.
    mod         - Modulus (signed remainder after division).
    rem         - Remainder after division.
    sign        - Signum.
```

Then if the user types `help` followed by a function or command name he gets a description of its use. For example,

```
»help  abs

 ABS    Absolute value.
    ABS(X) is the absolute value of the elements of X. When
    X is complex, ABS(X) is the complex modulus (magnitude) of
    the elements of X.

    See also SIGN, ANGLE, UNWRAP.

 Overloaded methods
    help sym/abs.m
```

The `who` and `clear` commands were covered earlier. The `who` command returns a list of the currently defined variables and the `clear` command can clear any or all of the variables currently defined.

A.4.2 Programming Language Flow Control

As in most programming languages, instructions in a MATLAB program are executed in sequence unless the flow of execution is modified by certain *flow control* commands.

The `if...then...else` construct allows the programmer to make a logical decision based on variable values and branch to one of two choices based on that decision. A typical `if` decision structure might look like

```
if a > b then
      Handle the case of a > b
      .
      .
else
      Handle the case of a <= b
      .
      .
end
```

The `else` choice is optional. It could be omitted to form

```
if a > b then
      Handle the case of a > b
      .
      .
end
```

The decision criterion between `if` and `then` can be anything that evaluates to a logical value. A logical 1 sends the program flow to the first group of statements between `if` and `else` (or `if` and `end`, if `else` is omitted) a logical 0 sends the program flow to the second group of statements between `else` and `end` (or past `end`, if `else` is omitted).

The `for` statement provides a way to specify that a group of instructions will be performed some number of times with a variable having a defined value each time.

```
for n = 1:15,
      .
      .
      .
end
```

This notation means set `n` to `1`, then execute the instructions between the `for` and `end` statements, then set `n` to `2` and repeat the instructions and keep doing that up through `n = 15` and then go to the next instruction after `end`. The more general syntax is

```
for n = N,
      .
      .
      .
end
```

where `N` is a vector of values of `n`. The vector, `N`, could be, for example,

`1:3:22`, `15:-2:6` or `[0,-4,8,27,-11,19]`.

```
»N = 1:10 ; x = [] ;
for n = N,
       x = [x,n^2] ;
end
x

x =

      1      4      9     16     25     36     49     64     81    100
```

The variable, n, is simply cycled through the values in the vector, in sequence, and the instructions in the `for` loop are executed each time.

The `while` flow-control command also defines a loop between the `while` command and an `end` statement.

```
while x > 32,
       .
       .
       .
end
```

If `x` is initially greater than 32, the instructions in this loop will execute sequentially once and then recheck the value of `x`. If it is still greater than 32 the loop will execute again. It will keep executing until `x` is not greater than 32. Whenever the condition, `x > 32` is no longer true, flow proceeds to the next instruction after the `end` statement. So it is important that something inside the loop change the value of `x`. Otherwise the loop will never terminate execution. The condition, `x > 32` is just an example. More generally the condition can be anything that evaluates to a logical value. While the value is a logical 1 the loop executes. When the value changes to a logical 0 the loop passes execution to the next instruction after `end`.

The `switch` command is a generalization of the `if...then...else` flow control. The syntax is

```
switch  expression
     case value1,
       ...
     case value2,
       ...
       .
       .
       .
     case valueN
       ...
```

```
end
```

The `expression` is evaluated. If its value is `value1` the instructions inside `case value1` (all instructions up to the `case value2` statement) are executed and control is passed to the instruction following the `end` statement. If its value is `value2` the instructions inside `case value2` are executed and control is passed to the instruction following the `end` statement, etc.... All `case`'s are checked in sequence.

The `input` statement is very useful. It allows the user to enter a variable value while the program is running. The value entered can be a number or a sequence of characters (a string). The general syntax for supplying a number is

```
n = input('message') ;
```

where `'message'` is any sequence of characters the programmer wants to appear to prompt the user of the program to enter a value. The general syntax for entering a string is

```
s = input('message','s') ;
```

where `'message'`is any sequence of characters the programmer wants to appear to prompt the user to enter the string and the `'s'` indicates to MATLAB to interpret whatever is entered by the user as a string instead of as a number.

The `pause` statement allows the programmer to cause the program to interrupt execution for a specified time and then resume execution. The syntax is

```
pause(n) ;
```

where `n` is the number of seconds to pause or simply

```
pause ;
```

which causes the program to wait for the user to press a key. These commands are useful for giving the user time to look at a graphical (or numerical) result before proceeding to the next set of instructions.

A.4.3 Elementary Matrices and Matrix Manipulation

There are several useful commands for generating matrices of various types. The most commonly-used ones in signal and system analysis are `zeros`, `ones`, `eye` , `rand` and "`randn`". The commands, `zeros` and `ones` are similar. Each one generates a matrix of a specified size filled with either 0's or 1's.

```
»zeros(1,3)
ans =
     0     0     0
```

```
»ones(5,2)
ans =
     1     1
     1     1
     1     1
     1     1
     1     1

»zeros(3)
ans =
     0     0     0
     0     0     0
     0     0     0
```

The command, `eye` generates an identity matrix (`eye` -> **I**, get it?).

```
»eye(4)
ans =
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
```

The commands, `rand` and `randn` are used to generate a matrix of a specified size filled with random numbers taken from a distribution of random (actually pseudorandom) numbers. The command, `rand` takes its numbers from a uniform distribution between zero and one. The command, `randn` takes its numbers from a normal distribution (Gaussian with zero mean and unit variance).

```
»rand(2,2)
ans =
    0.9501    0.6068
    0.2311    0.4860

»rand(2,2)
ans =
    0.8913    0.4565
    0.7621    0.0185
```

```
»randn(3,2)
ans =
   -0.4326    0.2877
   -1.6656   -1.1465
    0.1253    1.1909

»randn(3)
ans =
    1.1892    0.1746   -0.5883
   -0.0376   -0.1867    2.1832
    0.3273    0.7258   -0.1364
```

Two handy functions for determining the size of a matrix or the length of a vector are `size` and `length`. The function, `size`, returns a vector containing the number of rows, columns, etc... of a matrix.

```
» b
b =
     1     3     2

»size(b)
ans =
     1     3

» c
c =
     2     5     1

»size(c')
ans =
     3     1

» B
B =
     3     2     7
     4     1     2
    -1     3     1

»size(B)
ans =
     3     3
```

The function, `length`, returns the length of a vector.

```
»length(b)
ans =
     3

»length(c')
ans =
     3

» D
D =
     1     2
```

```
      3      6
     -2      9

»length(D)
ans =
      3
```

Notice that MATLAB returned a length for the 3 by 2 matrix, `D`. In this case the matrix, `D`, is interpreted as a column vector of row vectors and the length is the number of elements in that column vector, the number of row vectors in `D`.

The keyword, `eps`, is reserved to always hold information about the precision of calculations in MATLAB. Its value is the distance from the number, `1.0`, to the next largest number representable by MATLAB. Therefore it is a measure of the precision of number representation in MATLAB.

```
»eps
ans =
    2.2204e-16
```

This indicates that numbers near one are represented with a maximum error of about 2 parts in 10^{16} or about 2.22×10^{-14}%. In some numerical algorithms knowing the precision of representation can be important in deciding on when to terminate calculations.

A.4.4 Elementary Math Functions

All of the mathematical functions that important in signal and system analysis are available in MATLAB. Some common ones are the trigonometric functions, sine, cosine, tangent. They all accept arguments in radians and they will all accept vector or matrix arguments.

```
»sin(2)
ans =
     0.9093

»cos([2   4])
ans =
    -0.4161    -0.6536

»tan(B)
ans =
    -0.1425    -2.1850     0.8714
     1.1578     1.5574    -2.1850
    -1.5574    -0.1425     1.5574
```

Two other common functions are the exponential and logarithm functions and the square root function.

```
»exp([3   -1   j*pi/2])
```

```
ans =
  20.0855              0.3679                     0 + 1.0000i

»log([3  ;  6])
ans =
     1.0986
     1.7918

»sqrt([3  ;  j*2  ;  -1])
ans =
    1.7321
    1.0000 + 1.0000i
         0 + 1.0000i
```

Five functions are very useful with complex numbers, absolute value, angle, conjugate, real part and imaginary part.

```
»abs(1+j)
ans =
     1.4142

»angle([1+j   ;   -1+j])
ans =
     0.7854
     2.3562

»conj([1   j*3   -2-j*6])
ans =
   1.0000                  0 - 3.0000i  -2.0000 + 6.0000i

»real([1+j   ;   exp(j*pi/2)   ;   (2-j)^2])
ans =
     1.0000
          0
     3.0000

»imag([1+j   ;   exp(j*pi/2)   ;   (2-j)^2])
ans =
     1.0000
     1.0000
    -4.0000
```

Some other useful functions are fix, floor, ceiling, round, modulo, remainder and signum. The `fix` function rounds a number to the nearest integer toward zero. The `floor` function rounds a number to the nearest integer toward minus infinity. The `ceil` function rounds a number to the nearest integer toward plus infinity. The `round` function rounds a number to the nearest integer. The `mod` function implements the operation,

```
mod(x,y) = x - y.*floor(x./y)
```

if `y` is not zero. The `rem` function implements the function,

```
rem(x,y) = x - y.*fix(x./y)
```

if `y` is not zero. . The `sign` function returns a 1 if its argument is positive, a 0 if its argument is zero and a -1 if its argument is negative.

```
»fix([pi  2.6  -2.6  -2.4])
ans =
     3     2    -2    -2

»floor([pi  2.6  -2.6  -2.4])
ans =
     3     2    -3    -3

»ceil([pi  2.6  -2.6  -2.4])
ans =
     4     3    -2    -2

»round([pi  2.6  -2.6  -2.4])
ans =
     3     3    -3    -2

»mod(-5:5,3)
ans =
     1     2     0     1     2     0     1     2     0     1     2

»rem(-5:5,3)
ans =
    -2    -1     0    -2    -1     0     1     2     0     1     2

»sign([15  0  pi  -2.9  -1e-18])
ans =
     1     0     1    -1    -1

»sign(C)
ans =
     1     1     1
    -1    -1     1
     1     1     1
```

A.4.5 Specialized Math Functions

MATLAB provides many specialized and advanced functions. Four that are sometimes useful in signal and system analysis are the error function, the complementary error function, the least common multiple function and the greatest common divisor function. The error function, `erf`, is defined mathematically by $\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-u^2}du$ and the complementary error function, `erfc`, is defined mathematically by $\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}}\int_x^\infty e^{-u^2}du = 1 - \operatorname{erf}(x)$. These functions appear in the calculations of the probabilities of events which are Gaussian distributed. The least common multiple function, `lcm`, accepts as arguments two matrices of positive integers and returns a matrix of the smallest integer into which corresponding elements in both arguments divide an integer number of times. The greatest common divisor function, `gcd`, accepts as arguments two matrices of non-negative integers and returns a matrix of the greatest integer that will divide into the corresponding elements in both arguments an integer number of times. These functions can be used to determine the period or frequency of a sum of periodic functions.

```
»erf([-1  ;  0.5])
ans =
   -0.8427
    0.5205

»erfc([-1  ;  0.5])
ans =
    1.8427
    0.4795

»lcm([6  6],[9  7])
ans =
    18    42

»gcd([19  8],[41  12])
ans =
     1     4
```

A.4.6 Matrix Functions and Numerical Linear Algebra

Two specialized matrix operations are common in signal and system analysis, matrix inversion and finding eigenvalues. The function, `inv`, inverts a matrix. The matrix must be square and its determinant must not be zero.

```
»inv(A)
ans =
    0.0315   -0.0351    0.0124
   -0.0039    0.0108   -0.0789
   -0.0369    0.0767    0.0597

»inv(A)*A
ans =
    1.0000    0.0000    0.0000
    0.0000    1.0000    0.0000
    0.0000    0.0000    1.0000
```

The command, `eig` finds the eigenvalues of a square matrix.

```
»eig(B)
ans =
   6.5101
  -0.7550 + 3.0432i
  -0.7550 - 3.0432i
```

A.4.7 Data Analysis and Fourier Transforms

There are several functions in this group that are often useful. The function, `max`, can be used with either one argument or two. If one vector is passed to it, it returns the value of the largest element in that vector. If one matrix is passed to it, it returns a row vector of the largest element values in each column. If two matrices of the same shape are passed to `max`, it returns a matrix of the greater of the two element values for each pair of corresponding elements in the two matrices. The function, `min` operates exactly like `max` except that the minimum value or values are returned instead of the maximum.

```
» C
C =
     4     5     5
    -1    -3     2
     8     3     1

» max(C)
ans =
     8     5     5

» b
b =
     1     3     2

» max(b)
ans =
     3

» min(b)
ans =
     1

» c
c =
     2     5     1

» min(b,c)
ans =
     1     3     1
```

There are two statistical functions that are very useful, mean and standard deviation. The function, `mean` returns the mean value of a vector, and for a matrix it returns a row vector containing the mean values of the columns of the vector. The function, `std` returns the standard deviation of a vector, and for a matrix it returns a row vector containing the standard deviations of the columns of the vector.

```
» mean(A)
ans =
   32.6667   14.0000   17.3333

» std(B)
ans =
     2.6458     1.0000     3.2146

»mean(b)
ans =
     2
```

The function, `sort`, rearranges the elements of a vector into ascending order and rearranges the elements of each column of a matrix into ascending order.

```
»C
C =
     4     5     5
    -1    -3     2
     8     3     1

»sort(C)
ans =
    -1    -3     1
     4     3     2
     8     5     5
```

There are five functions, sum, product, difference, cumulative sum and cumulative product that are useful, especially with discrete-time functions. The function, `sum`, returns the sum of the element values in a vector and returns a row vector of the sums of the element values in the columns of a matrix. The function, `prod`, returns the product of the element values in a vector and returns a row vector of the products of the element values in the columns of a matrix. For a vector the function, `diff`, returns a vector for which each element is the difference between the corresponding element in the input vector and the previous element in the input vector. The returned vector is always one element shorter than the input vector. For a matrix, `diff` returns a matrix of difference vectors of the columns of the input matrix calculated in the same way. For vectors, the function, `cumsum`, returns a vector for which each element is the sum of the previous element values in the input vector and for matrices it returns a matrix in which each column contains the cumulative sum of the corresponding columns of the input matrix. The function, `cumprod` operates in the same that `cumsum` does except that it returns products instead of sums.

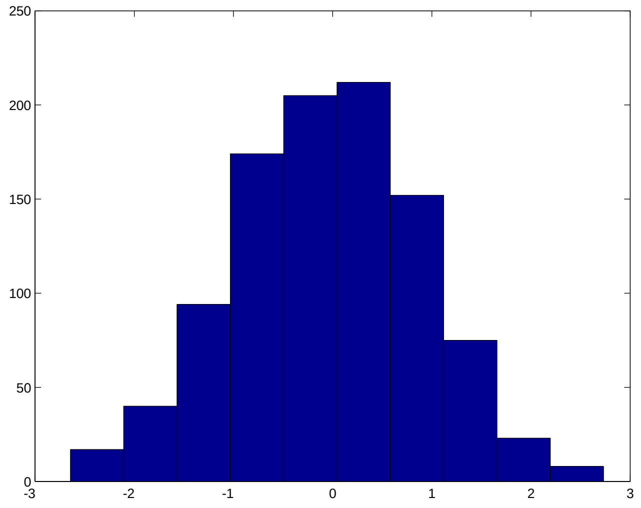
```
»sum(b)
ans =
     6

»sum(B)
ans =
     6     6    10

»prod(b)
ans =
     6

»prod(B)
ans =
   -12     6    14

»diff(A)
```

```
ans =
   -35     -7     -2
   -30    -34    -22

»diff(c)
ans =
     3     -4

»cumsum(C)
ans =
     4      5      5
     3      2      7
    11      5      8

»cumprod(C)
ans =
     4      5      5
    -4    -15     10
   -32    -45     10
```

Four more useful functions and commands are histogram , covariance, convolution and fast Fourier transform. The command, `hist`, graphs a histogram of a vector of data. It can also be used as a function to return data which can be used to graph a histogram. The function, `cov`, finds the variance of a vector or the covariance matrix of a matrix, considering each row to represent one observation and each column to contain data from one variable. The function, `conv`, convolves two vectors. The length of the returned vector is the sum of the lengths of the vectors convolved, minus one. The function, `fft`, computes the discrete Fourier transform of a vector. The command

»hist(randn(1000,1)) ,

produces the graph,



```
»cov(randn(5,5))
ans =
    0.9370    -0.0487    -1.3801     0.3545     0.1551
    -0.0487     1.2900    -0.8799     0.5086    -0.1737
    -1.3801    -0.8799     4.8511    -0.0141    -0.5036
     0.3545     0.5086    -0.0141     0.8204    -0.3830
     0.1551    -0.1737    -0.5036    -0.3830     0.5912

»conv(ones(1,5),ones(1,5))
ans =
     1     2     3     4     5     4     3     2     1
»fft(cos(2*pi*(0:15)'/4))
ans =
   -0.0000
   -0.0000 - 0.0000i
    0.0000 - 0.0000i
   -0.0000 - 0.0000i
    8.0000 - 0.0000i
    0.0000 - 0.0000i
   -0.0000 - 0.0000i
    0.0000 - 0.0000i
    0.0000
    0.0000 + 0.0000i
   -0.0000 + 0.0000i
    0.0000 + 0.0000i
    8.0000 + 0.0000i
   -0.0000 + 0.0000i
    0.0000 + 0.0000i
   -0.0000 + 0.0000i
```

A.4.8 Interpolation and Polynomials

In signal and system analysis it is often important to find the roots of equations. The function, `roots`, finds all the roots of an algebraic equation. The equation is assumed to be of the form, $a_N x^N + a_{N-1} x^{N-1} + \cdots a_2 x^2 + a_1 x + a_0 = 0$, and the vector, `a`, with the coefficients in descending order, $a_N \cdots a_0$, is the argument sent to `roots`. The roots are returned in a vector of length, $N-1$.

```
»roots([3  9  8  1])
ans =
   -1.4257 + 0.4586i
   -1.4257 - 0.4586i
   -0.1486
```

Another function that is useful, especially in the study of random variables, is `polyfit`. The name is a contraction of *polynomial curve fitting.* This function accepts an `X` vector containing the values of an independent variable, a `Y` vector containing the values of a dependent variable and a scalar, `N`, which is the degree of the polynomial used to fit the two sets of data. The function returns the coefficients, in ascending order, of a polynomial of degree, `N`, which has the minimum mean-squared error between its computed values of the dependent variable and the actual values of the dependent variable.

```
»X = 1:10 ;
```

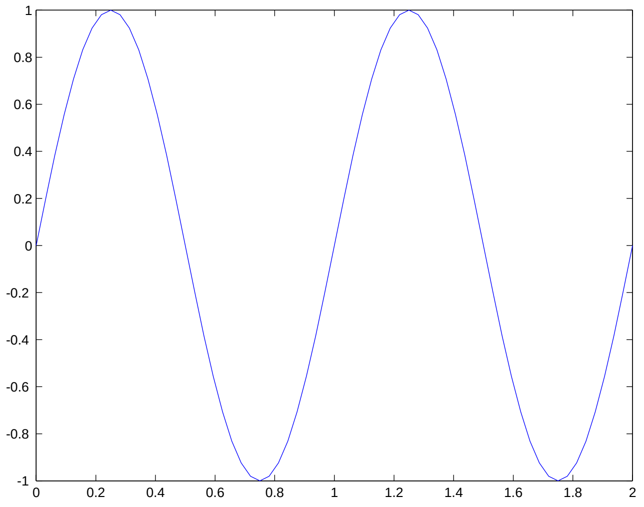
```
»Y = 5 + 3*X + randn(1,10)*2 ;

»polyfit(X,Y,2)
ans =
     0.0170     2.4341     6.4467
```

A.4.9 Two-Dimensional Graphs

One of the most important capabilities of MATLAB for signal and system analysis is its graphical data-plotting. Two-dimensional plotting is the plotting of one vector versus another in a two-dimensional coordinate system. The most commonly used plotting command is, as one might expect, `plot`. It accepts two vector arguments plus, optionally, some formatting commands and plots the second vector vertically versus the first horizontally. The instruction sequence,

```
»t = 0:1/32:2 ; x = sin(2*pi*t) ;
»plot(t,x) ;
```

produces the plot, (with a blue curve although it appears black here)



If a third argument is added after the second vector, it controls the plotting color and style. It is a string which determines the color of the line, the style of the line and the symbols (if any) used to mark points. Below is a listing of the characters and what they control.

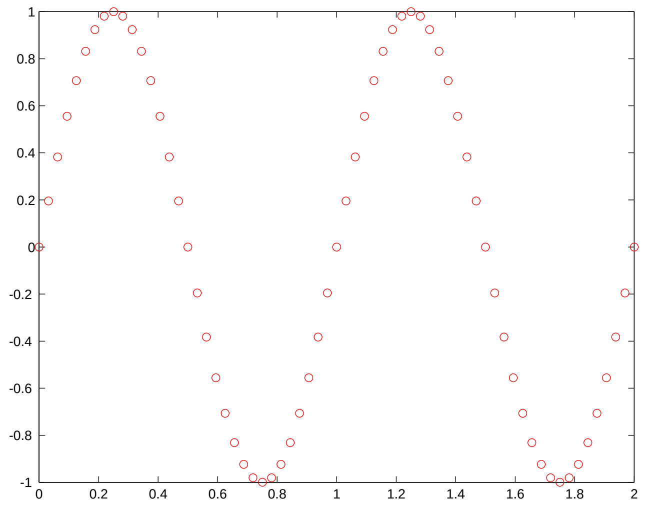
Color		Marker		Line Style	
y	yellow	.	point	-	solid
m	magenta	o	circle	:	dotted
c	cyan	x	x-mark	-.	dashdot
r	red	+	plus	--	dashed
g	green	*	star		
b	blue	s	square		
w	white	d	diamond		

`k`	`black`	`v`	`triangle (down)`
		`^`	`triangle (up)`
		`<`	`triangle (left)`
		`>`	`triangle (right)`
		`p`	`pentagram`
		`h`	`hexagram`

The instruction,

```
»plot(t,x,'ro')   ;
```

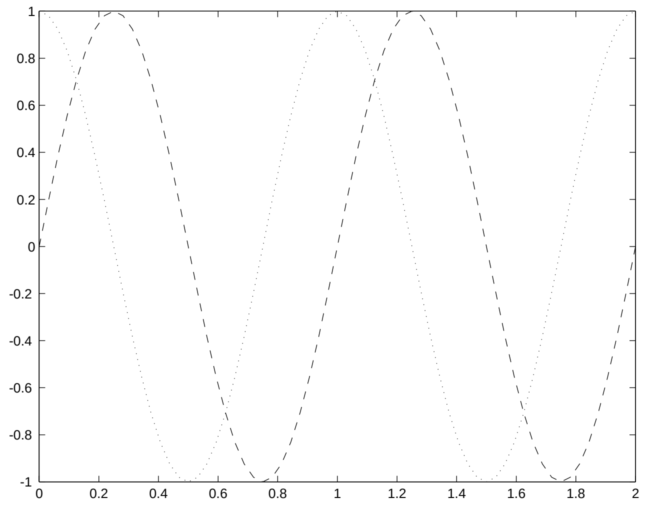
produces the plot, (with red circles although they appear black here),



and the instructions,

```
»y  =  cos(2*pi*t)  ;
»plot(t,x,'k--',t,y,'r:')   ;
```

produce the plot,



The function, `linspace`, is a convenient tool for producing a vector of equally-spaced independent variable values. It accepts either two or three scalar arguments. If two scalars, `d1` and `d2`, are provided, `linspace` returns a row vector of 100 equally-spaced values between `d1` and `d2` (including both end points). If three scalars, `d1`, `d2` and `N` are provided, `linspace` returns a row vector of `N` equally-spaced values between `d1` and `d2` (including both end points).

```
»linspace(0,5,11)
ans =
   Columns 1 through 7
           0    0.5000    1.0000    1.5000    2.0000    2.5000    3.0000
   Columns 8 through 11
     3.5000    4.0000    4.5000    5.0000
```

The function, `logspace`, is a convenient tool for producing a vector of equally-logarithmically-spaced independent variable values. It also accepts either two or three scalar arguments. If two scalars, `d1` and `d2`, are provided, `logspace` returns a row vector of 50 equally-logarithmically-spaced values between `10^d1` and `10^d2` (including both end points). If three scalars, `d1`, `d2` and `N` are provided, provided `logspace` returns a row vector of `N` equally-logarithmically-spaced values between `10^d1` and `10^d2` (including both end points).

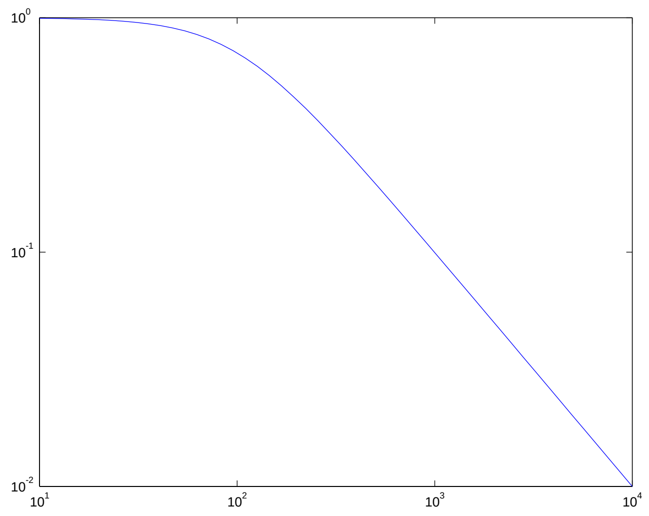
```
»logspace(1,2,10)
ans =
   Columns 1 through 7
    10.0000   12.9155   16.6810   21.5443   27.8256   35.9381   46.4159
   Columns 8 through 10
    59.9484   77.4264  100.0000
```

The command, `loglog` plots one vector versus another also, but with a logarithmic scale both vertically and horizontally. The syntax of formatting commands is exactly like `plot`.

The instruction sequence,

```
»f = logspace(1,4) ;
»H = 1./(1+j*f/100) ;
»loglog(f,abs(H)) ;
```

produces the plot, (with a blue curve although it appears black here)

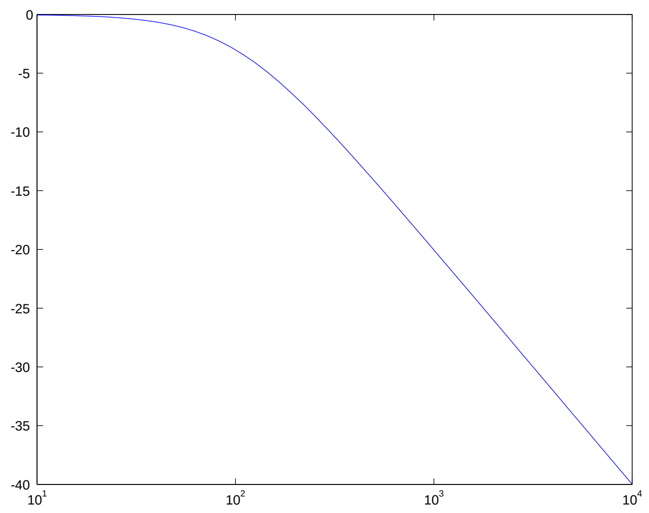


The command, `semilogx`, produces a plot with a logarithmic horizontal scale and a linear vertical scale and the command, `semilogy`, produces a plot with a logarithmic vertical scale and a linear horizontal scale. The syntax of formatting commands is exactly like `plot`.

The instruction,

```
»semilogx(f,20*log10(abs(H)))  ;
```

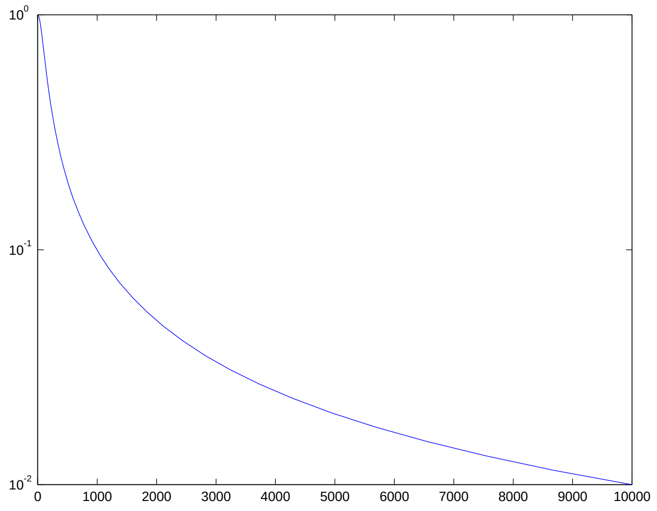
produces the plot, (with a blue curve although it appears black here)



and the instruction,

```
»semilogy(f,abs(H))  ;
```

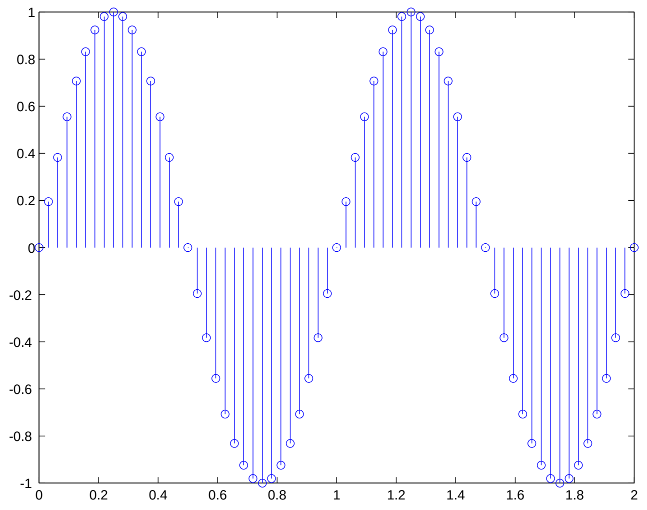
produces the plot, (with a blue curve although it appears black here)



The command, `stem`, is used to plot discrete-time functions. Each data point is indicated by a small circle at the end of a line connecting it to the horizontal axis. The instruction sequence,

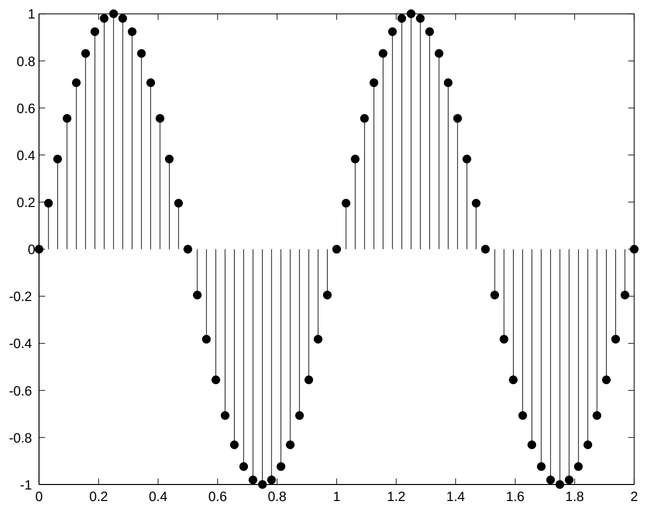
```
»t  =  0:1/32:2  ;  x  =  sin(2*pi*t)  ;
»stem(t,x)   ;
```

produces the plot, , (with blue stems and circles although they appear black here)



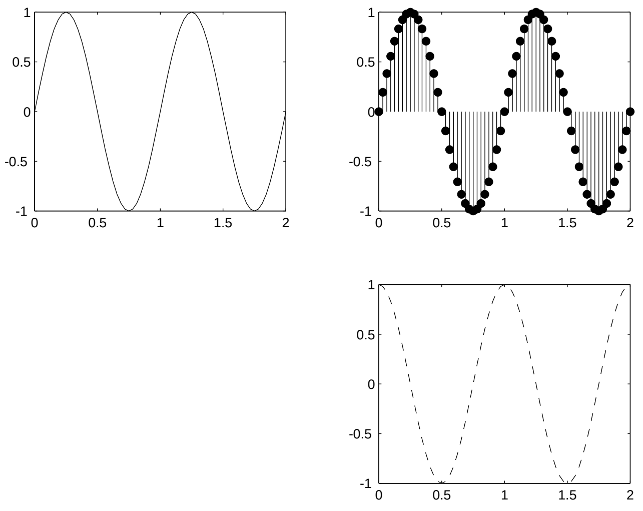
This plot can be modified to look more like a conventional stem plot by using two formatting commands.

```
»stem(t,x,'k','filled')   ;
```

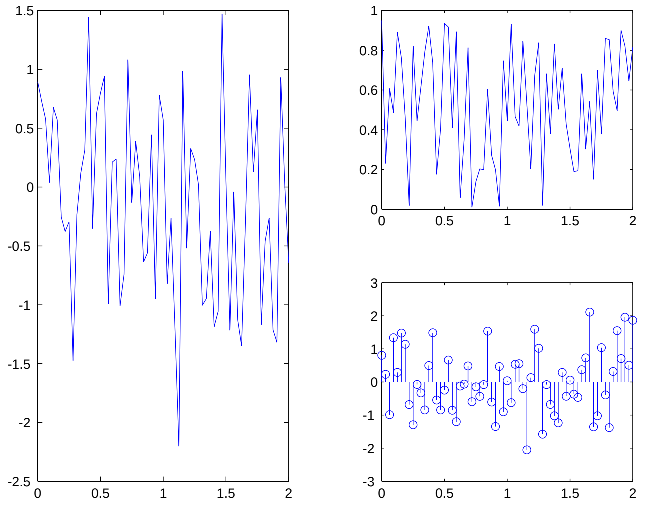


Often it is useful to display multiple plots simultaneously on the screen. The command, `subplot` allows that. Plots are arranged in a matrix. Three arguments are passed to `subplot`. The first is the number of rows of plots, the second is the number of columns of plots and the third is the number designating which of the plots is to be used by the next `plot` command. It is not necessary to actually use all the plots indicated by the number of rows and columns of any `subplot` command. Different `subplot` commands using different numbers of rows and columns can be used to place plots on one screen display.

```
»subplot(2,2,1) ; plot(t,x,'k') ;
»subplot(2,2,2) ; stem(t,x,'k','filled') ;
»x = cos(2*pi*t) ;
»subplot(2,2,4) ; plot(t,x,'k--') ;
```

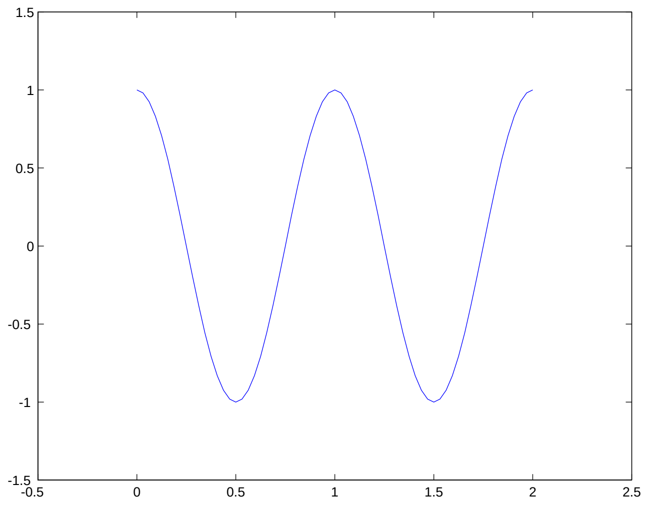


```
»subplot(1,2,1) ; plot(t,randn(length(t),1)) ;
»subplot(2,2,2) ; plot(t,rand(length(t),1)) ;
»subplot(2,2,4) ; stem(t,randn(length(t),1)) ;
```

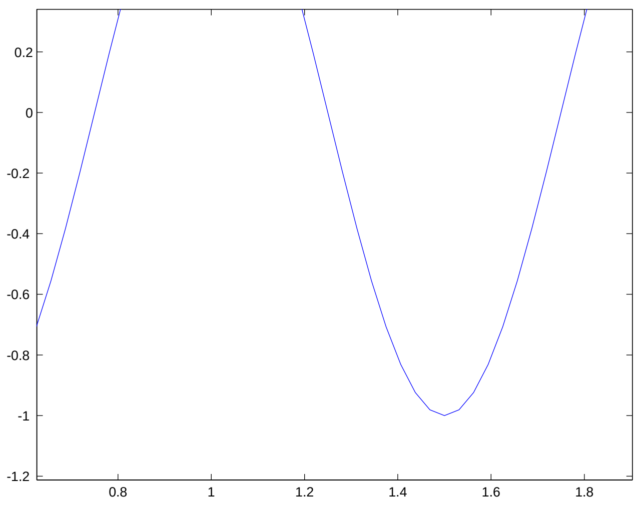


There are several useful commands which control the appearance of plots. The `axis` command allows the user to set the scale ranges arbitrarily instead of accepting the ranges assigned by MATLAB by default. The command, `axis`, takes an argument which is a four-element vector containing, in this sequence, the minimum x (horizontal) value, the maximum x value, the minimum y (vertical) value and the maximum y value.

```
»plot(t,x)   ;   axis([-0.5,2.5,-1.5,1.5])   ;
```

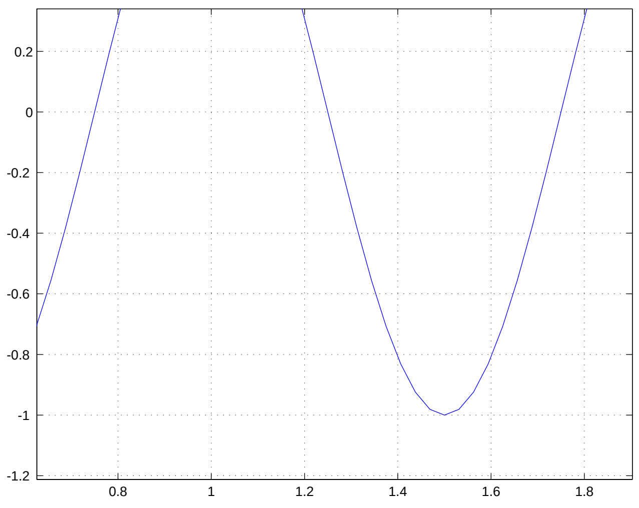


The command, `zoom`, allows the user to interactively zoom in and out of the plot to examine fine detail. Zooming is accomplished by clicking the mouse at the center of the area to be magnified, or dragging a selection box around the area to be magnified. A double click returns the plot to its original scale and a shift click zooms out instead of in. The plot below was made by zooming in on the previous plot. The command, `zoom`, enables zooming if it is currently disabled and disables it if it is currently enabled. The command, `zoom on` enables zooming unconditionally and the command, `zoom off` disables zooming unconditionally.



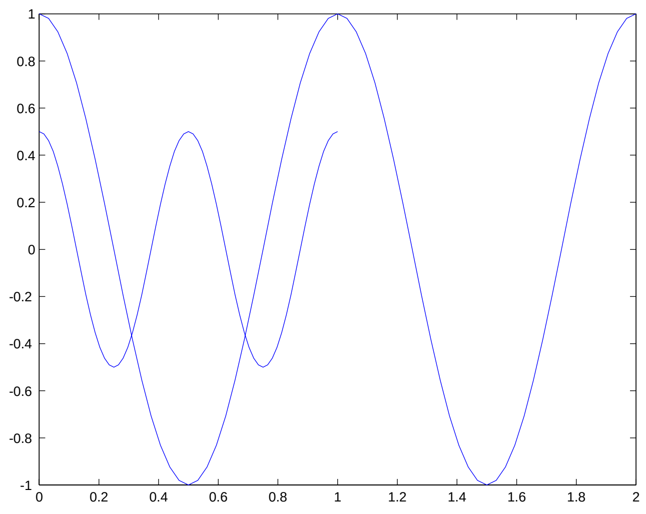
The command, `grid`, puts a set of gridlines on the current plot. The command `grid` turns the grid on if it is currently off and off if it is currently on. The command, `grid on`, unconditionally turns the grid on and the command, `grid off`, unconditionally turns the grid off.

```
»grid  on  ;
```



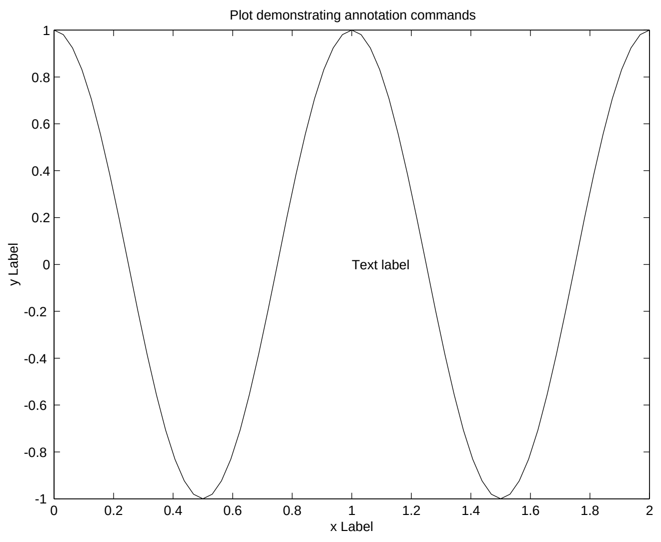
Sometimes it is desirable to plot one curve and later plot another on the same scale and the same set of axes. That can be done using the `hold` command. If a plot is made and then the command `hold on` is executed, the next time a `plot` command is executed the plot will be on the same set of axes. All subsequent plots will be on that same set of axes until a `hold off` command is executed.

```
»plot(t,x)   ;
»hold  on  ;
»plot(t/2,x/2)   ;
»hold  off  ;
```



There are four commands for annotating plots, `title`, `xlabel`, `ylabel` and `text`. As the names suggest, `title`, `xlabel` and `ylabel` are used to place a title on the plot, label the x axis and label the y axis. Their syntax is the same in each case. Each requires a string argument to be displayed. The `text` command is used to place a text string at an arbitrary position on the plot. Its arguments are an x position, a y position and a string. In addition, for each command, the font type, size and style, as well as how the text is positioned and other formatting can be set with optional extra arguments. (Type `help title` or `xlabel` or `ylabel` or `text` for more detail.)

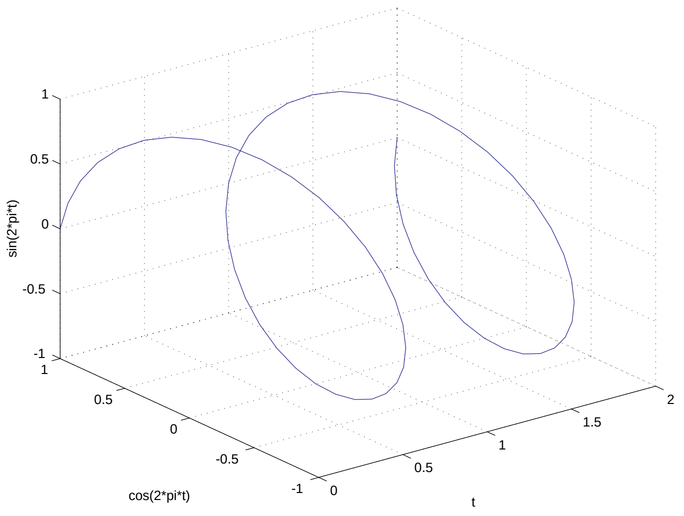
```
»plot(t,x)
»title('Plot demonstrating annotation commands') ;
»xlabel('x Label') ;
»ylabel('y Label') ;
»text(1,0,'Text label') ;
```



A.4.10 Three-Dimensional Graphs

Sometimes it is useful to plot curve in three-dimensional space. The command, `plot3` plots a curve in three-dimensional space determined by its three vector arguments.

```
»plot3(t,cos(2*pi*t),sin(2*pi*t)) ;
»grid on ;
»xlabel('t') ; ylabel('cos(2*pi*t)') ; zlabel('sin(2*pi*t)') ;
```

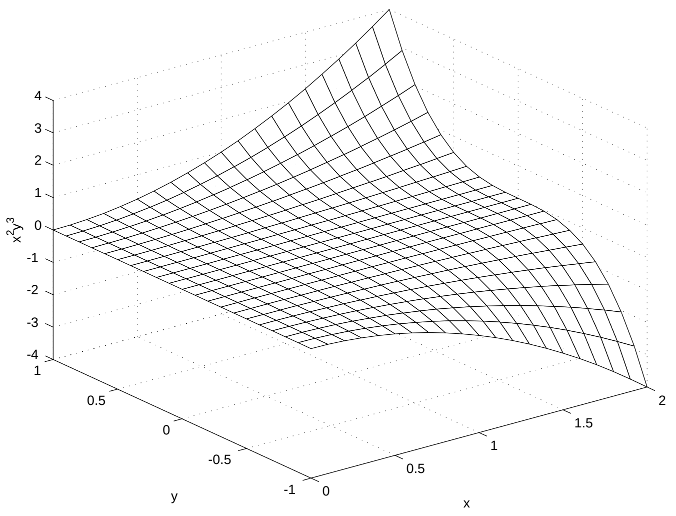


Notice the use of `zlabel` which is directly analogous to `xlabel` and `ylabel`.

The other common type of three-dimensional plot is a function of two independent variables. The command `mesh` plots a view of a function of two independent variables which is geometrically a surface above a plane determined by two orthogonal axes representing those two independent variables. The arguments are a vector or matrix containing values of the `x` variable, a vector or matrix containing values of the `y` variable and a matrix containing values of the function of `x` and `y`, `z`. If `x` is a vector of length, L_x, and `y` is a vector of length, L_y, then `z` must be an L_y by L_x matrix. The other choice is to make `x` and `y` and `z` all matrices of the same shape. A related function which aids in setting up `x` and `y` for three-dimensional plotting is `meshgrid`. This command takes two vector arguments and returns two matrices. If `x` and `y` are the two vector arguments and `X` and `Y` are the two returned matrices, the rows of `X` are copies of the vector, `x` and the columns of `Y` are copies of the vector, `y`.

```
»x = 0:0.5:2 ; y = -1:0.5:1 ;
»[X,Y] = meshgrid(x,y) ;
»X
X =
          0    0.5000    1.0000    1.5000    2.0000
          0    0.5000    1.0000    1.5000    2.0000
          0    0.5000    1.0000    1.5000    2.0000
          0    0.5000    1.0000    1.5000    2.0000
          0    0.5000    1.0000    1.5000    2.0000
```

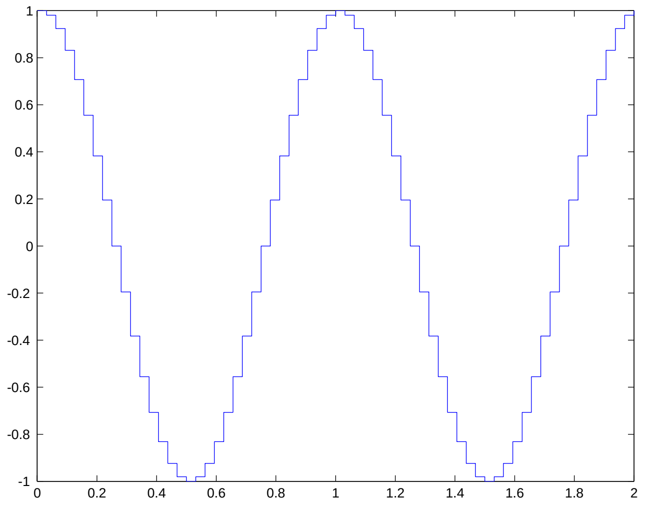
```
» Y
Y =
   -1.0000    -1.0000    -1.0000    -1.0000    -1.0000
   -0.5000    -0.5000    -0.5000    -0.5000    -0.5000
         0          0          0          0          0
    0.5000     0.5000     0.5000     0.5000     0.5000
    1.0000     1.0000     1.0000     1.0000     1.0000

»x = 0:0.1:2 ; y = -1:0.1:1 ;
»[X,Y] = meshgrid(x,y) ;
»z = (X.^2).*(Y.^3) ;
»mesh(x,y,z) ;
»xlabel('x') ; ylabel('y') ; zlabel('x^2y^3') ;
```



A.4.11 Specialized Graphs

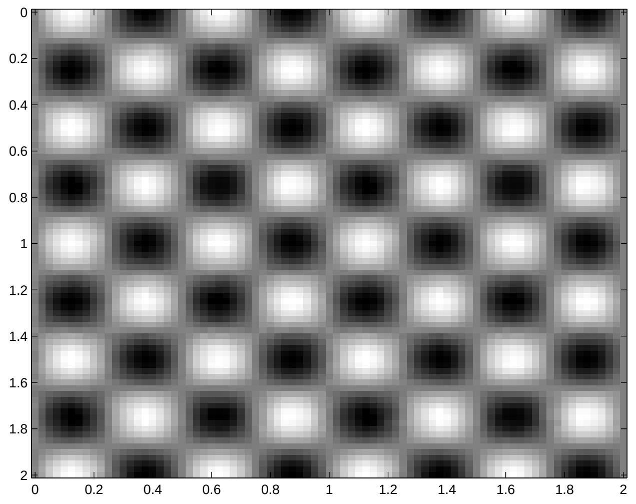
The command `stairs` is a special plotting command. It accepts two arguments in the same way `plot` and `stem` do but plots a “stairstep” plot.

```
»t = 0:1/32:2 ; x = cos(2*pi*t) ;
»stairs(t,x) ;
```



The command, `image`, accepts a matrix and interprets its elements as specifying the intensity or color of an image. The argument can be an *m* by *n* matrix or an *m* by *n* by 3 matrix. If the argument is an *m* by *n* matrix the values of the elements are interpreted as colors according to the current `colormap`. A `colormap` is an *m* by 3 array containing specifications of the red, blue and green intensities of colors. The value of an element of the matrix argument is used as an index into the current `colormap` array to determine what color is displayed on the screen. For example, the `colormap`, `gray`, is 64 by 3. So an element value of one would select the first color in the `gray colormap` and an element value of 64 would select the last color in the `gray colormap`. If the argument passed to `image` is or an *m* by *n* by 3 matrix the 3 element values at each row and column position are interpreted as the specification of the intensity of red, green and blue components of a color at that position.

```
»x = 0:0.025:2 ; y = 0:0.025:2 ;
» [X,Y] = meshgrid(x,y) ;
»colormap(gray) ;
»z = sin(4*pi*X).*cos(4*pi*Y) ;
»minz = min(min(z)) ; maxz = max(max(z)) ;
»z = (z - minz)*64/(maxz-minz) ; z = uint8(z) ;
»image(x,y,z) ;
```



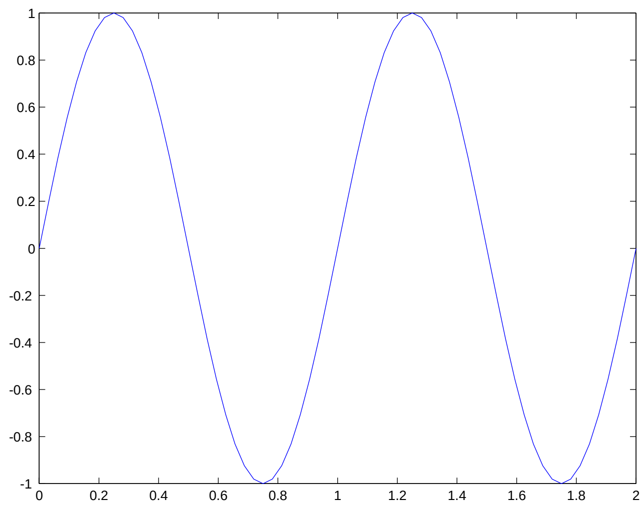
The command `imagesc` does the same thing as `image` except that it is scaled to use the entire colormap.

A.4.12 Handle Graphics

In MATLAB the every graph or plot is an *object* whose appearance can be modified by using the `set` command to change its properties. Each time a graph is created with the `plot` or `stem` or `stairs` commands (or many other commands) a graphic object is created. That object can be modified by reference to its *handle*. In the instruction,

```
»h  =  plot(t,sin(2*pi*t))  ;
```

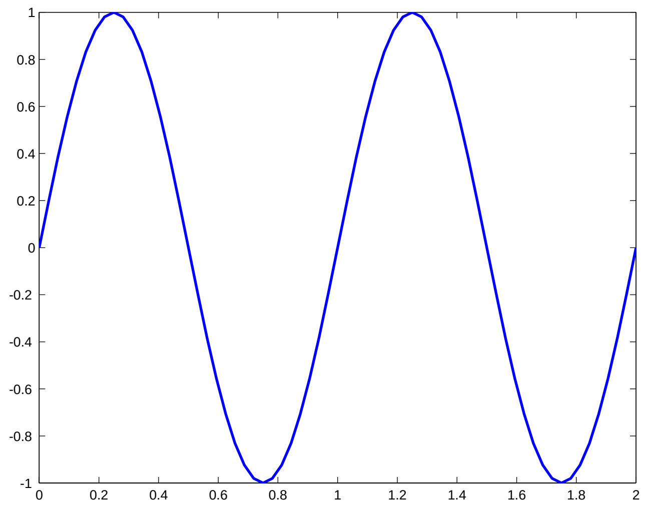
`h` is a handle to the plot created by the `plot` command, creating the plot,



Then we can use a command like

```
»set(h,'LineWidth',2)   ;
```

to change the plot to



Graphic objects have many properties. Their properties and some possible values of the properties can be seen by using the `set` command.

```
»set(h)
      Color
      EraseMode: [ {normal} | background | xor | none ]
      LineStyle: [ {-} | -- | : | -. | none ]
      LineWidth
      Marker: [ + | o | * | . | x | square | diamond | v | ^ | > | < |
            pentagram | hexagram | {none} ]
      MarkerSize
      MarkerEdgeColor: [ none | {auto} ] -or- a ColorSpec.
      MarkerFaceColor: [ {none} | auto ] -or- a ColorSpec.
      XData
      YData
      ZData

      ButtonDownFcn
      Children
      Clipping: [ {on} | off ]
      CreateFcn
      DeleteFcn
      BusyAction: [ {queue} | cancel ]
      HandleVisibility: [ {on} | callback | off ]
      HitTest: [ {on} | off ]
      Interruptible: [ {on} | off ]
      Parent
      Selected: [ on | off ]
      SelectionHighlight: [ {on} | off ]
      Tag
      UIContextMenu
```

```
UserData
Visible: [ {on} | off ]
```

The command, `get`, lists the properties of an object and the current value of each property.

```
»get(h)
      Color = [0 0 1]
      EraseMode = normal
      LineStyle = -
      LineWidth = [2]
      Marker = none
      MarkerSize = [6]
      MarkerEdgeColor = auto
      MarkerFaceColor = none
      XData = [ (1 by 65) double array]
      YData = [ (1 by 65) double array]
      ZData = []

      ButtonDownFcn =
      Children = []
      Clipping = on
      CreateFcn =
      DeleteFcn =
      BusyAction = queue
      HandleVisibility = on
      HitTest = on
      Interruptible = on
      Parent = [3.00134]
      Selected = off
      SelectionHighlight = on
      Tag =
      Type = line
      UIContextMenu = []
      UserData = []
      Visible = on
```

The number of properties that can be modified is much to great to explore in this limited tutorial. (For more information consult a MATLAB manual.)

When a plotting command like plot or stem is used, a `figure` window with default properties is automatically created by MATLAB and the plot is put into that `figure`. The user can also create a figure window and specify its properties using the `figure` command. A figure window can be removed from the screen by closing it with the `close` command.

Not only is the graph an object, the axes on which the graph is drawn are themselves an object and the figure window is also an object. The command, `gcf` (get current figure) returns a handle to the current figure and the command, `gca` (get current axes) returns a handle to the current axes.

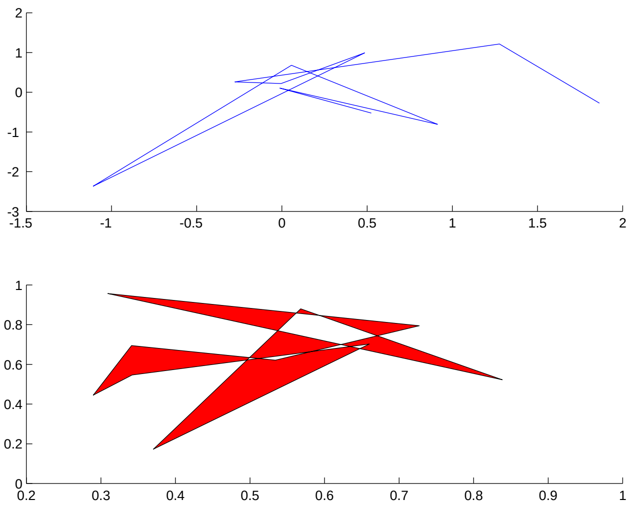
```
»set(gcf)
      BackingStore: [ {on} | off ]
      CloseRequestFcn
      Color
      Colormap
      CurrentAxes
      CurrentObject
      CurrentPoint
      Dithermap
      DithermapMode: [ auto | {manual} ]
      IntegerHandle: [ {on} | off ]
      InvertHardcopy: [ {on} | off ]
      KeyPressFcn
      MenuBar: [ none | {figure} ]
      MinColormap
      Name
      NextPlot: [ {add} | replace | replacechildren ]
      NumberTitle: [ {on} | off ]
      PaperUnits: [ {inches} | centimeters | normalized | points ]
      PaperOrientation: [ {portrait} | landscape ]
      PaperPosition
      PaperPositionMode: [ auto | {manual} ]
      PaperType: [ {usletter} | uslegal | A0 | A1 | A2 | A3 | A4 | A5 | B0
            | B1 | B2 | B3 | B4 | B5 | arch-A | arch-B | arch-C | arch-D |
            arch-E | A | B | C | D | E | tabloid ]
      Pointer: [ crosshair | fullcrosshair | {arrow} | ibeam | watch | topl
            | topr | botl | botr | left | top | right | bottom | circle |
            cross | fleur | custom ]
      PointerShapeCData
      PointerShapeHotSpot
      Position
      Renderer: [ {painters} | zbuffer | OpenGL ]
      RendererMode: [ {auto} | manual ]
      Resize: [ {on} | off ]
      ResizeFcn
      ShareColors: [ {on} | off ]
      Units: [ inches | centimeters | normalized | points | {pixels} |
            characters ]
      WindowButtonDownFcn
      WindowButtonMotionFcn
      WindowButtonUpFcn
      WindowStyle: [ {normal} | modal ]

      ButtonDownFcn
      Children
      Clipping: [ {on} | off ]
      CreateFcn
      DeleteFcn
      BusyAction: [ {queue} | cancel ]
      HandleVisibility: [ {on} | callback | off ]
      HitTest: [ {on} | off ]
      Interruptible: [ {on} | off ]
      Parent
      Selected: [ on | off ]
      SelectionHighlight: [ {on} | off ]
      Tag
      UIContextMenu
      UserData
      Visible: [ {on} | off ]

»set(gca)
      AmbientLightColor
      Box: [ on | {off} ]
      CameraPosition
      CameraPositionMode: [ {auto} | manual ]
      CameraTarget
```

```
CameraTargetMode: [ {auto} | manual ]
CameraUpVector
CameraUpVectorMode: [ {auto} | manual ]
CameraViewAngle
CameraViewAngleMode: [ {auto} | manual ]
CLim
CLimMode: [ {auto} | manual ]
Color
ColorOrder
DataAspectRatio
DataAspectRatioMode: [ {auto} | manual ]
DrawMode: [ {normal} | fast ]
FontAngle: [ {normal} | italic | oblique ]
FontName
FontSize
FontUnits: [ inches | centimeters | normalized | {points} | pixels ]
FontWeight: [ light | {normal} | demi | bold ]
GridLineStyle: [ - | -- | {:} | -. | none ]
Layer: [ top | {bottom} ]
LineStyleOrder
LineWidth
NextPlot: [ add | {replace} | replacechildren ]
PlotBoxAspectRatio
PlotBoxAspectRatioMode: [ {auto} | manual ]
Projection: [ {orthographic} | perspective ]
Position
TickLength
TickDir: [ {in} | out ]
TickDirMode: [ {auto} | manual ]
Title
Units: [ inches | centimeters | {normalized} | points | pixels |
      characters ]
View
XColor
XDir: [ {normal} | reverse ]
XGrid: [ on | {off} ]
XLabel
XAxisLocation: [ top | {bottom} ]
XLim
XLimMode: [ {auto} | manual ]
XScale: [ {linear} | log ]
XTick
XTickLabel
XTickLabelMode: [ {auto} | manual ]
XTickMode: [ {auto} | manual ]
YColor
YDir: [ {normal} | reverse ]
YGrid: [ on | {off} ]
YLabel
YAxisLocation: [ {left} | right ]
YLim
YLimMode: [ {auto} | manual ]
YScale: [ {linear} | log ]
YTick
YTickLabel
YTickLabelMode: [ {auto} | manual ]
YTickMode: [ {auto} | manual ]
ZColor
```

```
ZDir: [ {normal} | reverse ]
ZGrid: [ on | {off} ]
ZLabel
ZLim
ZLimMode: [ {auto} | manual ]
ZScale: [ {linear} | log ]
ZTick
ZTickLabel
ZTickLabelMode: [ {auto} | manual ]
ZTickMode: [ {auto} | manual ]

ButtonDownFcn
Children
Clipping: [ {on} | off ]
CreateFcn
DeleteFcn
BusyAction: [ {queue} | cancel ]
HandleVisibility: [ {on} | callback | off ]
HitTest: [ {on} | off ]
Interruptible: [ {on} | off ]
Parent
Selected: [ on | off ]
SelectionHighlight: [ {on} | off ]
Tag
UIContextMenu
UserData
Visible: [ {on} | off ]
```

There are two very useful primitive graphics commands, `line` and `patch`. The command, `line`, draws a straight line between any two points according to the scale of the current axes. The command, `patch`, draws a filled polygon according to the scale of the current axes.

```
»subplot(2,1,1)   ;   line(randn(10,1),randn(10,1))   ;
»subplot(2,1,2)   ;   patch(rand(10,1),rand(10,1),'r')   ;
```



A.4.13 Graphical User Interface Tools

There is a set of tools in MATLAB which a programmer can use to develop a graphical user interface. They allow the programmer to create windows, buttons, menus, etc... and link program execution to them. A discussion of these features is outside the scope of this tutorial.

A.4.14 Character Strings

String handling is often important and MATLAB has a complete set of string manipulation functions. The function, `char`, converts positive integers to the corresponding character according to the American Standard for Communication and Information Interchange, (ASCII).

```
»char([40:52])
ans =
()*+,-./01234

»char([[65:90] ; [97:122]])
ans =
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
```

A string is a vector of characters. Several MATLAB functions operate on strings. The function, `strcat` concatenates two strings.

```
»s1 = char([65:90]) ;
»s2 = char([97:122]) ;
»s3 = strcat(s1,s2)
s3 =
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

The function, `strcmp`, compares two strings. If they are the same it returns a logical 1, otherwise it returns a logical 0.

```
»strcmp(s1,s2)
ans =
     0

»strcmp('ABCDEFGHIJKLMNOPQRSTUVWXYZ',s1)
ans =
     1
```

The function, `findstr` finds one string within another. If the shorter of the two input strings is found within the longer, `findstr` returns the index of the first character in the longer string. If the shorter string is not found within the longer string, an empty matrix is returned.

```
»findstr(s1,'MNO')
ans =
    13

»findstr('tuv',s2)
ans =
    20

»findstr('tuv',s1)
ans =
     []
```

There are also functions to convert from strings to numbers and vice versa. The function, `num2str` converts a number to a string.

```
»n = 25.3 ;

»n
n =
   25.3000

»length(n)
ans =
     1

»ns = num2str(n) ;

»ns
ns =
25.3

»length(ns)
ans =
     4

»ns(2)
ans =
5
```

The function, `str2num`, converts a string to a number.

```
»ns = '368.92' ;

»length(ns)
ans =
     6

»n = str2num(ns) ;

»n
n =
  368.9200

»length(n)
```

```
ans =
     1
```

A.4.15 File Input/Output

Another important function of any programming language is to read data from a file and to store data in a file. In MATLAB a file is opened for reading or writing by the `fopen` command. The argument of the `fopen` command is a string specifying the file name and, optionally, the path to the file name and it returns an positive integer file identifier which is used in future references to that file. If the file should already exist and cannot be found, `fopen` returns a -1 to indicate that the file was not found. If a file is opened for write access it need not already exist, MATLAB will create it. When all interaction with the file is complete it is closed by the `fclose` command. The argument of `fclose` is the file identifier number.

Files can contain data encoded in different ways. Two very common formats are binary and text. The two commands, `fread` and `fwrite` read and write binary data to and from a file. The arguments of these functions are the file identification number, the number of data to be read and a string which is a precision indicator specifying how to read the data. The precision argument indicates how many bits represent a number. The available formats are

MATLAB	C or Fortran	Description
'char'	'char*1'	character, 8 bits
'uchar'	'unsigned char'	unsigned character, 8 bits
'schar'	'signed char'	signed character, 8 bits
'int8'	'integer*1'	integer, 8 bits.
'int16'	'integer*2'	integer, 16 bits.
'int32'	'integer*4'	integer, 32 bits.
'int64'	'integer*8'	integer, 64 bits
'uint8'	'integer*1'	unsigned integer, 8 bits.
'uint16'	'integer*2'	unsigned integer, 16 bits.
'uint32'	'integer*4'	unsigned integer, 32 bits.
'uint64'	'integer*8'	unsigned integer, 64 bits
'float32'	'real*4'	floating point, 32 bits.
'float64'	'real*8'	floating point, 64 bits.

If the file is a text file it can be read as text by the `fgetl` command, which reads in one line of text (up to the first line terminator) and returns that string without the terminator. The next invocation of `fgetl` reads in the next line of text in the same way.

The command, `save`, saves all the current variables to a disk file. If a file name is provided the data are put into a file of that name with a `.mat` extension. If a file name is not provided the data are put into a file named, `matlab.mat`. The command, `load`, retrieves variables saved with the `save` command. There are also options with each command which allow the saving and retrieval of a selected set of variables instead of all of them.

A.4.16 Time and Dates

MATLAB has a group of functions which return the current time or date or allow timing the execution speed of a program. They are in the group, `timefun`.

A.4.17 Data Types and Structures

MATLAB allows the creation of structures and cell arrays to handle large groups of disparate data as a unit, to convert data from one precision to another and other related functions. These features are described in the group, `datatypes`.