MATLAB on Athena (AC-71)

Table of Contents || Revision history || Copyright information

Getting Started

MATLAB (MATrix LABoratory) is an interactive program for scientific and engineering numeric calculation. Applications include:

- matrix manipulation
- finding the roots of polynomials
- digital signal processing
- x-y and polar plotting
- 3-dimensional graphics

This document assumes a fundamental working knowledge of matrix mathematics. If you are interested in learning more about MATLAB but do not understand matrix mathematics, you may want to attend the Athena minicourse which begins at a more basic level.

Starting a Session

Before using MATLAB for the first time, we recommend that you create a subdirectory in your home directory named **matlab**. Use this directory for your personal MATLAB function files. See **Creating Your Own Functions and Scripts** for details. Only once, before you ever use MATLAB at all, do the following:

```
athena% mkdir ~/matlab
```

More information on the **mkdir** command can be found in the **Creating a Directory** section of **Working on Athena**.

To actually use MATLAB, do the following:

```
athena% add matlab
athena% matlab
```

You can also start the application from the Dash menubar. Select MATLAB on the Analysis and Plotting submenu of the Numerical/Math menu.

A new xterm window pops up (you are prompted for its location and size in the usual manner), and all appropriate environment variables are set in that window alone. The new xterm displays the MATLAB header and >> prompt. A graphics window also pops up as MATLAB starts up, but then disappears almost immediately; you have no control over this window and need not worry about its behavior. As you create graphics, they appear in separate windows. The graphics windows usually appear on top of the MATLAB xterm window and you will need to position the windows so that they don't overlap (this makes it easier to use them together).

MATLAB lives in the matlab locker on Athena. The main program and all of the functions for MATLAB are in specific subdirectories of the /mit/matlab directory. This locker is automatically attached whenever

you execute **add matlab** unless it has already been attached during your login session.

When you are finished using MATLAB, type **exit** at the >> prompt; any graphics windows disappear and the athena% system prompt returns in the xterm from which you started MATLAB. Typing **exit** at the athena% prompt makes this xterm go away.

Matlab 6 introduced the Matlab Desktop, a graphical interface with additional tools; because it runs more slowly than the traditonal command-line interface, it is not currently on by default but you can use it by typing **desktop** at the MATLAB prompt, or start MATLAB by typing **matlab -desktop**.

Running a MATLAB Demo

If you've never seen MATLAB in action before, type **demo** at the MATLAB >> prompt:

```
>> demo
```

This puts you into the MATLAB demo window. At the bottom left is a "Close" button, which closes the demonstration window. At the top, the menu bar has File, Window, and Help options. The File window allows you to open and close files, create new files, print demonstrations, and perform other file-related functions. Window allows you to switch between demo windows; Help uses Netscape to access MATLAB's graphical help capabilities.

A list of topics which have demonstrations will appear in the left-hand window; information on these topics appears in the upper right-hand window. In order to expand a topic in the left window, double-click on it and subtopics will appear below. When you click on one of these, a list of possible demonstrations to run in the lower right-hand window will appear. At this point, the button below this window changes to "Run (demonstration name)". Click on this to run the demonstration.

More Help With MATLAB

Online Help

MATLAB has a good built-in help facility, both from the command line as described below and through a graphical "Help Desk" which includes a full documentation set in Adobe Acrobat (PDF) format, a searchable command index, and some toolbox-specific help. To start the Help Desk, type **helpdesk** at your MATLAB prompt.

To use command line help, you may first want to type **more on** at your MATLAB prompt. This will cause MATLAB to pause between each screen of text. Therefore, you can control your reading, rather than having the entire file scroll by. (Use the space bar to advance a page, RETURN to advance a line, "q" to exit from the item being displayed. Type 'help more' for more details.)

Typing **help** with no arguments at the MATLAB >> prompt displays a list of all primary help topics along with a short description for each one. Typing **help *directoryname*** displays a list of .m files in the directory, along with a brief description of each one if one has been provided, while typing 'what directoryname' displays all of the MATLAB-related files in a directory.

Typing **help *function*** displays the syntax for the ***function*** (i.e. what arguments it expects) and a short description of what that function does. For example, if you type **help help** at the MATLAB prompt, it will

give you documentation on the help function.

If you think you are doing everything right, but MATLAB disagrees, try looking at **help** for the functions you are using. It may also be useful to verify the source of a function by typing 'which function -all'; this behaves similarly to the where command in UNIX, telling you what directory the function is located in.

If you don't know a function's name, you can try to find it using the 'lookfor' command (or the search feature in the Help Desk). This does a keyword search on the first comment line of each M-file accessible to MATLAB, and may take some time to finish. (The switch "-all" can be used to search through the entire first comment block, but this will lengthen the search.) For example:

```
>> lookfor cartesian
```

returns

```
CART2POL Transform Cartesian to polar coordinates.
CART2SPH Transform Cartesian to spherical coordinates.
POL2CART Transform polar to Cartesian coordinates.
SPH2CART Transform spherical to Cartesian coordinates.
```

When you write your own functions, you can also include **help** information for them. This can be very useful for other people using your function, or for your own use if you haven't used the function for a while. For more information on how to include **help** information for your own functions, see the section **Creating Your Own Functions**.

Hardcopy Documentation

This document only discusses a portion of the complete capabilities of MATLAB. If you don't see something here, it doesn't mean MATLAB can't do it. If you have questions about MATLAB's capabilities, or need further help with using it, we recommend that you refer to the following MATLAB vendor documentation:

- **Using MATLAB**
- **Using MATLAB Graphics**

Complete documentation sets, including the **MATLAB New Features Guide**, **MATLAB Release Notes**, and the **SIMULINK User's Guide**, are available for reference in Barker and Hayden Libraries, and in the Athena Consulting Office in the basement of the Student Center (W20).

You may purchase your own copies of the documention from the vendor; contact:

```
The MathWorks, Inc.
24 Prime Parkway
Natick, MA  01760
(508) 653-1415
FAX:  (508) 653-2997
E-mail:  info@mathworks.com
```

Various MATLAB **Toolbox Guides** are also available from the vendor. If you are purchasing copies of the documentation on an MIT Purchase Order, reference our MATLAB site license number, 3021.

Matrix/Vector Operations

Creating and Working with Matrices

The most straightforward way to initialize a matrix is to type a command of the form:

```
variable = [value1-1  value1-2  value1-3 ... ;
value2-1  value2-2 ...]
```

where each *value* may be rational or complex numbers. Within the square brackets that are used to form vectors and matrices, you can use a semicolon to end a row. For example:

```
>> x = [1 2 3 4; 0 9 3 8]
x =
   1     2     3     4
   0     9     3     8
>>
```

(Note: You can also use the semicolon after an expression or statement to suppress printing or to separate statements.)

To use complex numbers, enter them in the form a+bi, without spaces. You can also initialize matrices with these:

```
>> x = [4+5i  2  4;  1  3+i  7]
x =
   4.0000 + 5.0000i   2.0000             4.0000
   1.0000             3.0000 + 1.0000i   7.0000
```

Vectors and scalars are initialized the same way as matrices. It is important to note that MATLAB indexes matrices in the following manner:

```
(1,1)  (1,2)  (1,3)  (1,4)
(2,1)  (2,2)  (2,3)  (2,4)
```

This means that the first element always has index **(1,1)**, not **(0,0)**.

If **x** is already defined as a vector of the form **[val1 val2 val3 val4...]** then you can define a new variable as a subset of **x** by using the index of the specific value in vector **x**. For example, if x is defined as **[2 4 1 7]**, then:

```
>> z = x(3)
z =
    1
```

You can specify a value in matrix **y** the same way:

```
>> y =  [ 1 2 3 4 5; 3 4 5 6 7]
y =
     1     2     3     4     5
     3     4     5     6     7

>> z  =  y(2,1)
z =
     3
```

You can also specify a range of numbers in a defined vector or matrix using the colon operator. Colon notation is used to specify a range of numbers, or to access selected elements of a matrix. J:K steps from J to K in increments of 1. For example:

```
>> z  =  (1:5)
z =
     1     2     3     4     5
```

J:D:K steps from J to K in increments of D. For example:

```
>> z = (1:3:7)
z =
     1     4     7
>> z = (14:-2:5)
z =
     14    12    10    8     6
```

MATLAB has a variety of built-in functions to make it easier for you to construct matrices without having to enumerate all the elements. (The following examples show both vectors and matrices.)

The **ones** function creates a matrix whose elements are all ones. Typing **ones(*m*,*n*)** creates an *m* row by *n* column matrix of ones. To create a ones matrix that is the same size as an existing matric, you can use **ones(size(*X*))**. This does not affect the input argument. For example (this definition of **x** applies to subsequent examples in this section):

```
>>  x = [1 2 3 4; 0 9 3 8]
x =
     1     2     3     4
     0     9     3     8

>>  y  =  ones(size(x))
y =
     1     1     1     1
     1     1     1     1
```

The **zeros** function is similar to the **ones** function. Typing **zeros(*m*,*n*)** creates an *m*-by-*n* matrix of zeros, and **zeros(size(x))** will create a two-by-four matrix of zeros, if **x** is defined the same way as above.

The **max** and **min** functions return the largest and smallest values in a vector. For example (this definition of **z** applies to the following series of examples):

```
>> z  =  [1  2  -9  3  -3  -5]
z =
   1     2    -9     3    -3    -5

>>  max(z)
ans =
     3
```

If called with a matrix as its argument, **max** returns a row vector in which each element is the maximum value of each column of the input matrix. The **max** function can also return a second value: the index of the maximum value in the vector or row of the maximum value down a column. To get this, assign the result of the call to **max** to a two element vector instead of just a single variable.

For example:

```
>> [a b] = max(z)
a =
     3
b =
     4
>> [a b] = max (x)
a =
     1     9     3     8
b =
     1     2     1     2
```

where **a** is the maximum value of the vector and **b** is the index of that value. The MATLAB function **min** is exactly parallel to **max**:

```
>>  min(z)
ans =
     -9
```

sum and **prod** are two more useful functions for matrices. If z is a vector, **sum(z)** is the sum of all the elements of the vector **z**:

```
>>  sum(z)
ans =
   -11
```

For matrices, **sum** sums the columns. Note that the semicolon following the statement suppresses the output. For example:

```
>> w = magic(3);
>> w

w =

     8     1     6
     3     5     7
     4     9     2

>> sum(w)

ans =

    15    15    15

>> sum(sum(w))

ans =

    45
```

Similarly, **prod(z)** is the product of all the elements of **z**.

```
>>  prod(z)
ans  =
      -810
```

For matrices, **prod(y)** takes the product down the columns.

```
>> prod(w)
ans =
      96    45    84
```

Often, it is useful to define a vector as a subunit of a previously defined vector. To do this, you can use the colon operator. For example, using the **z** defined above,

```
>>  z
z =
     1     2    -9     3    -3    -5

>>  y  =  z(2:5)
y =
     2    -9     3    -3
```

where **(2:5)** represents the sequence of index values to be taken from the larger vector.

You can use subarrays in MATLAB, as well. A(J,:) specifies the Jth row of the matrix A. A(:,K) specifies the Kth column of the matrix A. Combining these, A(J-range,K-range) specifies a submatrix. So, you could specify a subset of **y** using the colon:

```
>> y = [1 2 3 4 5; 3 4 5 6 7]
y =
     1     2     3     4     5
     3     4     5     6     7
>> z = y(1:2,2:3)
z =
     2     3
     4     5
```

The **size** function returns a two-element vector giving the dimensions of the matrix with which it was called. For example:

```
>>  x = [1 2 3 4; 0 9 3 8]
x =
     1     2     3     4
     0     9     3     8

>>  y  =  size(x)
y =
    2     4
```

You can also define the result to be two separate values (as shown in the **max** example):

```
>>  [m  n]  =  size(x)
m =
        2
n =
        4
```

The **length** operator returns the length of a vector. If **z** is defined as in the above examples,

```
>>  length(x)
ans =
          4
```

For matrices, **length** is the length or the width, whichever is greater, i.e., **length(x)** is equivalent to **max (size(x)**).

Basic Arithmetic

MATLAB uses a straight-forward notation for basic scalar arithmetic. The following table summarizes simple MATLAB notation:

```
+    addition
-    subtraction
*    multiplication
/    division
^    exponentiation
```

All of these work for two scalars, including complex scalars. You can also add, subtract, multiply or divide all the elements of a vector or matrix by a scalar. For example, if **x** is a matrix or vector, then **x+1** adds one to each element **x**, and **x/2** divides each element of **x** by 2. **x^2** does not square each element of **x**, but **x.^2** does. Matrix and vector exponentiation are discussed later.

The special operator ' (prime or apostrophe) takes the transposition of a matrix. For example:

```
>> a = [1 2 3]

a =

     1     2     3

>> a'

ans =

     1
     2
     3
```

For a matrix with complex entries, this takes the complex conjugate transpose. To take the transpose without the conjugate, use **.'**.

```
>> b = [1 2 3] + i*[-4 3 2]

     1.0000 - 4.0000i     2.0000 + 3.0000i     3.0000 + 2.0000i

>> b'

ans =
     1.0000 + 4.0000i
     2.0000 - 3.0000i
     3.0000 - 2.0000i

>> b.'

ans =
     1.0000i - 4.0000i
     2.0000  + 3.0000i
     3.0000  + 2.0000i
```

Element-Wise Operations

You often may want to perform an operation on each element of a vector while doing a computation. For example, you may want to add two vectors by adding all of the corresponding elements. The addition (+) and subtraction (-) operators are defined to work on matrices as well as scalars. For example, if x = [1 2 3] and y = [5 6 2], then

```
>>  w  =  x+y
w =
    6     8     5
```

Multiplying two matrices element by element is a little different. The ***** symbol is defined as matrix multiplication when used on two matrices. Use **.*** to specify element-wise multiplication. So, using the x and y from above,

```
>>  w  =  x .* y
w =
   5    12     6
```

You can perform exponentiation on a vector similarly. Typing **x .^ 2** squares each element of x.

```
>>  w  =  x .^ 2
w =
```

```
    1     4     9
```

Finally, you cannot use / to divide two matrices element-wise, since / and \ are reserved for left and right matrix ``division." Instead, you must use the **./** function. For example:

```
>>  w  =  y ./ x
w =
   5.0000    3.0000    0.6667
```

The **abs** operator returns the magnitude of its argument. If applied to a vector, it returns a vector of the magnitudes of the elements. For example, if x = [2 -4 3-4i -3i]:

```
>>  y  =  abs(x)
y =
   2   4   5   3
```

The **angle** operator returns the phase angle (i.e., the "argument") of its operand in radians. The **angle** operator can also work element-wise across a vector. For example:

```
>>  phase  =  angle(x)
phase=
      0   -3.1416   -0.9273   -1.5708
```

The **sqrt** function computes the square root of its argument. If its argument is a matrix or vector, it computes the square root of each element. For example:

```
>>  x = [4   -9    i    2-2i];
>>  y  =  sqrt(x)
y =
   2.0000   0 + 3.0000i   0.7071 + 0.7071i   1.5538 - 0.6436i
```

MATLAB also has operators for taking the real part, imaginary part, or complex conjugate of a complex number. These functions are **real**, **imag** and **conj**, respectively. They are defined to work element-wise on any matrix or vector.

MATLAB has several operators that round fractional numbers to integers. The **round** function rounds its elements to the nearest integer. The **fix** function rounds its elements to the nearest integer towards zero, e.g. rounds ``down" for positive numbers, and ``up" for negative numbers. The **floor** function rounds its elements to the nearest integer towards negative infinity, e.g. ``down." The **ceil** (short for ceiling) function rounds its elements to the nearest integer towards positive infinity, e.g. ``up."

round	rounds to nearest integer
fix	rounds to nearest integer towards zero
floor	rounds down (towards negative infinity)
ceil	rounds up (towards positive infinity)

All of these commands are defined to work element-wise on matrices and vectors. If you apply one of them to a complex number, it will round both the real and imaginary part in the manner indicated. For example:

```
>>  ceil(3.1+2.4i)
ans=
    4.0000 + 3.0000i
```

MATLAB can also calculate the remainder of an integer division operation. If x = y * n + r, where n is an integer and r is less than n but is not negative, then **rem(x,y)** is r. For example:

```
>> x = [8  5  11];
>> y = [6  5  3];
>>  r  =  rem(x,y)
r=
  2     0     2
```

The standard trigonometric operations are all defined as element-wise operators. The operators **sin**, **cos** and **tan** calculate the sine, cosine and tangent of their arguments. The arguments to these functions are angles in radians. For example:

```
>> sin (pi/2)
ans =
      1
```

Note that the functions are also defined on complex arguments. For example, cos(x+iy) = cos(x)cosh(y) - i sin(x)sinh(y). The inverse trig functions (**acos**, **asin** and **atan**) are also defined to operate element-wise across matrices. Again, these are defined on complex numbers, which can lead to problems for the incautious user. The arctangent is defined to return angles between pi/2 and -pi/2.

In addition to the primary interval arctangent discussed above, MATLAB has a full four-quadrant arctangent operator, **atan2**. **atan2(***y*,*x***)** returns the angle between -pi and pi whose tangent is the real part of *y/x*. If *x* and *y* are vectors, **atan2(***y*,*x***)** divides *y* by *x* element-wise, then returns a vector in which each element is the four-quadrant arctangent of corresponding element of the *y/x* vector.

MATLAB also includes functions for exponentials and logarithms. The **exp** operator computes e to the power of its argument. This works element-wise, and on complex numbers. The **pi** function returns the floating point number nearest the value of pi. So, to generate the complex exponential with a frequency of pi/4, we could type:

```
>>  n  =  0:7;
>>  s  =  exp(i*(pi/4)*n)
s =

  Columns 1 through 4
   1.0000            0.7071 + 0.7071i   0.0000 + 1.0000i  -0.7071 + 0.7071i

  Columns 5 through 8
  -1.0000 + 0.0000i  -0.7071 - 0.7071i  -0.0000 - 1.0000i   0.7071 - 0.7071i
```

MATLAB also has natural and base-10 logarithms. The **log** function calculates natural logs, and **log10** calculates base-10 logs. Both operate element-wise for vectors. Both are defined for complex values.

Logical Operations

MATLAB allows you to perform boolean operations on vectors element-wise. For the purpose of boolean algebra, MATLAB regards any non-zero real element as true, and zero as false. MATLAB uses **&** for the boolean **and** operator, **|** for **or**, and **~** for **not**. For example:

```
>>  x  =  [1 0 2 4]  &  [0 0 1 i]
x =
     0     0     1     1
>>  x  =  [1 0 2 4]  |  [0 0 1 i]
x =
     1     0     1     1
```

In addition, you can run a cumulative boolean "or" or boolean "and" across all the elements of a matrix or vector. If *v* is a vector or matrix, **any(***v***)** returns true if any element of *v* is non-zero; **all(***v***)** returns true if all

the elements of *v* are non-zero.

You can also compare two vectors element-wise with any of six basic relational operators:

<	less than
>	greater than
==	equal to
~=	not equal to
<=	less than or equal to
>=	greater than or equal to

For example:

```
>>  x  =  [1 2 3 4 5]  <=  [5 4 3 2 1]
x =
      1    1    1    0    0
```

Relational operators are particularly important in programming control structures.

Control Structures

MATLAB includes several control structures to allow you to write programs. The **for** command allows you to make a command or series of commands be executed several times. It is functionally very similar to the **for** function in C. (Be careful not to use the variable i for an index; else, you may inadvertantly redefine sqrt(-1).) For example, typing

```
for s = 1:4
      s
end
```

causes MATLAB to make the variable **s** count from 1 to 4, and print its value for each step (the indentations in the **for** structure are optional). From the above example, MATLAB would return:

```
s =
     1
s =
     2
s =
     3
s =
     4
```

Every **for** command must have a matching **end** statement to indicate which commands should be executed several times. You can have nested **for** loops. For example:

```
for m = 1:3
    for n = 1:3
       x (m,n) = m + n*i;
    end
end
```

defines x to be the matrix:

```
x =
```

```
1.0000 + 1.0000i   1.0000 + 2.0000i   1.0000 + 3.0000i
2.0000 + 1.0000i   2.0000 + 2.0000i   2.0000 + 3.0000i
3.0000 + 1.0000i   3.0000 + 2.0000i   3.0000 + 3.0000i
```

(This is yet another way to define a matrix.)

The **if** command lets you have programs that make decisions about what commands to execute. The basic command looks like:

```
if a > 0
        x = a^2;
end
```

This command assigns x to be the value of "a" squared, if a is positive. Note that it has to have an **end** to indicate which commands are actually part of the **if**. In addition, you can define an **else** clause which is executed if the condition you gave the **if** is not true. The example above might be expanded thus:

```
if a > 0
       x = a^2;
else
       x = - a^2;
end
```

For this version, if you had already set **a** to be 2, then **x** would get the value 4, but if **a** was -3, **x** would be -9. Note that you only need one **end**, which comes after all the clauses of the **if**. Finally, you can expand the **if** to include several possible conditions. If the first condition isn't satisfied, it looks for the next, and so on, until it either finds an **else**, or finds the **end**. For example:

```
if a > 0
         x = a^2;
elseif a == 0,
         x = i;
else
         x = - a^2;
end
```

This command checks whether **a** is positive: if **a** is not positive, it checks whether **a** is zero; if **a** is not zero, it does the else clause. Thus, if **a** is positive, **x** well be **a** squared, if **a** is 0, **x** will be **i**, and if **a** is negative, then **x** will be the negative of **a** squared.

The third major control structure is the **while** command. The **while** command allows you to execute a group of commands until some condition is no longer true. These commands appear between the **while** and its matching **end** statement. For example, if you want to start x at 2 and keep squaring x until it is greater than one million, you would type:

```
x = 2
while x < 1000000
       x = x^2;
end
```

This runs until x is 4.2950e+09. Everything between the **while** line and the **end** is executed until the boolean condition on the **while** line is no longer true. You have to make sure this condition will eventually stop being true, or the command will never finish. If it is not initially true, no commands will be executed.

The pause and keyboard commands can be useful in functions that you write yourself. The **pause** command causes MATLAB to wait for a key to be pressed before continuing. The **keyboard** command passes control to the keyboard, indicated by a the prompt K>>. You can examine or change variables, or issue any MATLAB command. Terminate keyboard mode by executing the command **return** at the K>> prompt.

Sometimes you will want to terminate a **for** or **while** loop early. You can use the **break** command to jump out of a **for** or **while** command. For example, you could rewrite the **while** example above using **break**:

```
while 1
        if x > 1000000
                break;
        end
        x = x^2;
end
```

Selective Indexing

Sometimes, you only want to perform an operation on certain elements of a vector, such as all the elements of the vector that are less than 0. One way to do this is a **for** loop that checks to see whether each element is less than zero, and if so, does the appropriate function. However, MATLAB includes another way to do this. If you say

```
>> x(x<0) = - x(x<0)
```

MATLAB changes all the negative elements of the vector x to be positive. The following sequence of commands illustrates this:

```
>> x = [-3 -2 0 2 4]
x =
    -3    -2    0    2    4
>> x(x<0) = - x(x<0)
x =
     3     2    0    2     4
```

Though this notation can be more confusing than a **for** loop, MATLAB is written such that this operation executes much, much faster than the equivalent **for** loop.

You can also perform operations on a vector conditionally based upon the value of the corresponding element of another vector. For example, if you want to divide two vectors element-wise, you have to worry about what happens if the denominator vector includes zeros. One way to deal with this is shown below.

```
>> x = [3 2 0 2 4]
 x =
      3    2    0    2    4
 >> y = [1 1 1 1 1]
 y =
      1    1    1    1    1
 >> q = zeros(1,length(y))
 q =
      0    0    0    0    0
 >> q(x~=0) = y(x~=0) ./ x(x~=0)
 q =
    0.3333    0.5000       0     0.5000    0.2500
```

You can perform this type of conditional indexing with any boolean operator discussed earlier, or even with boolean operators on the results of functions on elements of vectors. For example:

```
>> q((x<=3) & (q<=0.4)) = q((x<=3) & (q<=0.4)) + 14
q =
   14.3333   0.5000     14.0000     0.5000    0.2500
```

Polynomial Operations

Vectors can also be used to represent polynomials. If you want to represent an Nth-order polynomial, you use a length N+1 vector where the elements are the coefficients of the polynomial arranged in descending order of exponent. So, to define y = x^2 - 5x + 6, you would type:

```
>>  y  =  [1  -5  6];
```

The MATLAB **roots** function calculates the roots of a polynomial for you. If **y** is defined as above:

```
>>  roots(y)
ans =
     3
     2
```

MATLAB also has the **poly** function, which takes a vector and returns the polynomial whose roots are the elements of that vector.

You can multiply two polynomials using the **conv** function. The convolution of the coefficient vectors is equivalent to multiplying the polynomials. For example, if we define w = 3x^2 - 4x -1 (so the vector w = [3 -4 -1]) and use the **y** given above:

```
>>  z  =  conv(w,y)
z =

      3   -19    37   -19    -6
```

The **polyval** function returns the value of a polynomial at a specific point. For example:

```
>>  polyval(y,1)
ans =

     2
```

The **polyval** function also works element-wise across a vector of points, returning the vector where each element is the value of the polynomial at the corresponding element of the input vector.

Signal Processing Functions

MATLAB comes with several useful signal processing functions. Type **help signal** for a list of functions, and then see the help for the individual function such as **help fft**, **help filter**, or **help freqz**.

Libraries and Search Paths

MATLAB has several libraries that contain files of functions called M-files. An M-file consists of a sequence of MATLAB statements (see **Creating Your Own Functions**). To see the complete list of toolboxes (libraries), type the command **path** at the MATLAB >> prompt. For a description of a toolbox, type **help** ***toolboxname*** or **what** ***toolboxname*** at your prompt. To see the text of an M-file, type **type** ***filename***.

The list you get with the **path** command is also MATLAB's current directory search path. MATLAB's search rules are as follows; when you enter a name, MATLAB:

1. looks to see if the name is a variable.
2. looks for it as a built-in function.
3. searches in the current directory for the related .m file.
4. searches the directories specified by **path** for the .m file.

You can also use **path**, with appropriate arguments, to add or change directories in the search path. For details, type **help path** at the MATLAB >> prompt, or see **path** in the **MATLAB Reference Guide**.

Graphics

MATLAB supports several commands that allow you to display results of your computations graphically. Graphs are displayed in a graphics window that MATLAB creates when you give one of the plotting commands. The default graphics window starts up with a black background. To change this, type the command **whitebg** at the MATLAB >> prompt.

The following are some of the large number of new plot types that have been added to MATLAB 4.0:

- 3-D shaded color surface graphs
- 3-D contour plots
- 3-D line trajectories
- 3-D volumetric "slice" plots
- 3-D axes on mesh and surface plots
- Combination surface and contour plots
- Image display
- Lighting and rendering models

Plotting Individual Graphs

The **plot** command is the simplest way of graphing data. If *x* is a vector, **plot(***x***)** will plot the elements of *x* against their indices. The adjacent values of *x* will be connected by lines. For example, to plot the discrete-time sequence that is a sinusoid of frequency pi/6, you would type:

```
>>  n  =  0:11;
>>  y  =  sin((pi/6)*n);
>>  plot(n,y)
```

When plot gets two vectors for arguments, it creates a graph with the first argument as the abscissa values, and the second vector as ordinate values. In the example above, plot will use the values of **y** for the y-axis, and the values of **n** for the x-axis. If you typed:

```
>>  plot(y)
```

MATLAB would use the values of **y** for the y-axis and their indices for the x-axis. Notice that the first value graphed would have an abscissa value of one, and not zero. This is because MATLAB indexes vector elements beginning with one, not zero.

You can also change the type of line used to connect the points by including a third argument specifying line type. The format for this is **plot(***x*,*y*,*line-type***)**. The line types available are:

```
'-'    solid line (default)
'--'   dashed line
```

```
':'    dotted line
'-.'   line of alternating dots and dashes
```

Whichever character you chose to use must be enclosed by single quotes. For example, **plot(n,y,':')** would create the same graph as above, except that the points would be connected by a dotted line. The default line type is solid. In this case, it is misleading to connect the adjacent values by lines, since this is a graph of a discrete-time sequence. Instead, we should just put a mark to indicate each sample value. We can do this by using a different set of characters in place of the *line-type* argument. If we use a **'.'**, each sample is marked by a point. Using a **'+'** marks each sample with a + sign, **'*'** uses stars, **'o'** uses circles, and **'x'** uses x's. For example, the following command plots the values of **y** against their indices, marking each sample with a circle:

```
>>  plot(n,y,'o')
```

You can also plot several graphs on the same axis. For example, the following command plots **y1** versus **x1** and **y2** versus **x2** on the same axis using different line types for each graph:

```
plot(x1,y1,x2,y2)
```

You can also include a specific line or point type (from the list above) for each graph:

```
plot(x1,y1,'line-type1',x2,y2,'line-type2')
```

You can also create plots with either or both axes changed to log-scale. All of these functions follow the same conventions for arguments and line or point types as **plot**:

Command	X-Axis Scale	Y-Axis Scale
loglog	logarithmic	logarithmic
semilogy	linear	logarithmic
semilogx	logarithmic	linear

You can use additional MATLAB commands to title your graphs or put text labels on your axes. For example, the following command labels the current graph at the top with the text enclosed in single quotes:

```
>>  title('MATLAB Graph #1')
```

Similarly, the following commands label the x- and y-axes:

```
>>  xlabel('This is the x-axis')
>>  ylabel('This is the y-axis')
```

The **axis** command is used to control the limits and scaling of the current graph. Typing

```
a  =  axis
```

will assign a four-element vector to **a** that sets the ``minimum ranges" for the axes. The first element is the minimum x-value, the second is the maximum x-value for the current graph. The third and fourth elements are the minimum and maximum y-values, respectively. You can set the values of the axes by calling the axis function with a four-element vector for an argument. You might want to do this, for example, if you were going to plot several sets of data on the same graph and you knew that the range of one set of data was significantly larger than the other.

The elements of the vector you use to set the values of the axes should be your choices for the x- and y-axis limits, in the same order as specified above (**[x-min x-max y-min y-max]**). So, if you type

```
>>  axis([-10 10 -5 5])
```

you will rescale the axis in the graphics window so the x-axis goes at least from -10 to 10, and the y-axis from -5 to 5. MATLAB tends to resize axes whenever it produces a plot; see **help axis** for other options.

The **hold** command will keep the current plot and axes even if you plot another graph. The new graph will just be put on the current axes (as much as fits). Typing **hold** a second time will toggle the **hold** off again, so the screen will clear and rescale for the next graph.

Aspect Ratio Control

As with many computers, MATLAB's graphing system is not perfectly scaled. MATLAB does allow you to switch between a perfectly square aspect ratio and the normal aspect ratio. Typing **axis('square')** will give you a truly square aspect ratio, while **axis('normal')** would flip you back to MATLAB's usual aspect ratio.

Plotting Multiple Graphs

You can use the **subplot** command to split the screen into multiple windows, and then select one of the sub-windows as active for the next graph. The **subplot** function can divide the graphics window into a maximum of four quadrants; first the window splits horizontally, then it splits vertically. The format of the command is:

```
>> subplot(xyn)
```

or

```
>> subplot(i>x,y,n)
```

.

In this command, x is the number of vertical divisions, y is the number of horizontal divisions, and n is the window to select for the first plot. Both x and y must be less than or equal to two, and n must be less than or equal to x times y. For example, ***subplot(121)*** *will create two full-height, half-width windows for graphs, and select the first, e.g. left, window as active for the first graph. After that, unless you specifically indicate which window is active, MATLAB cycles through them with each successive plot. The order of that cycling is as follows:*

```
       |
   1   |   2
-------+-------
   3   |   4
       |
```

Typing ***subplot*** *with no arguments returns the graphics window to its original, single-window state.*

Output of Plots and Graphs

Sending Graphics to a Printer

*You can use the **print** command to get a hard copy of the current graphics window:*

```
>>  print -Pprintername
```

For example, the following command sends the current graph to the printer linus:

```
>>  print -Plinus
```

*It may take a minute or two for MATLAB to send the graphics file to the printer. The MATLAB default is to send printouts to PostScript printers. Other printer formats are supported; for details type **help print** at the >> prompt.*

*The default orientation for printing MATLAB graphics is portrait (longest paper dimension is vertical). Prior to issuing the **print** command, you can change the orientation with the **orient** commands, as follows:*

- ***orient landscape** - prints the current graphics window horizontally*
- ***orient tall** - prints the graphics window on the full page, in portrait orientation*
- ***orient portrait** - returns to the default orientation; the graphics window prints with an aspect ratio of 4/3 in the middle of the page*
- ***orient** - by itself, returns a string with the current paper orientation*

Sending Graphics to a File

*MATLAB graphics can be saved to a file. MATLAB supports several device file formats, with PostScript being the default. For details and the list of device options, type **help print** at the MATLAB prompt.*

*To save the current graphics window to a file, type the following command at the MATLAB prompt, where **device** is one of the options listed in **help print**:*

```
>> print -ddevice filename
```

Creating Your Own Functions and Scripts

Creating Your Own Functions

You can create your own functions within MATLAB using M-files. An M-file is an ASCII text file that has a filename ending with .m, such as stars.m. You can create and edit M-files with a text editor such as Emacs. There are two classes of M-files, functions and scripts.

*Function definitions are stored in files with the name function-name.m. The first line of a function definition must start with the word **function** and can be of the form:*

```
function  function-name(argument1, argument2,...)
```

*This specifies the name of the function and its input arguments. For instance, the first line of a function definition for **stars** from the M-file named stars.m would be **function stars(t)**.*

Functions Returning No Values

A common example of a function that doesn't return any value is one that draws a graph.

```
function stars(t)
%STARS(T)       draws stars with parameter t
n = t * 50;
plot(rand(1,n), rand(1,n),'.');
%that line plots n random points
title('My God, Look at all the Stars!');
%label the graph
```

*When **stars** is executed, it will plot a random sprinkling of points in your graphics window on a set of axes. The first line in the M-file defines this as a function named **stars** that takes one argument named **t**, and doesn't return anything. The next line is the help comment. To include help info for your own functions, just start the text line in the corresponding M-file with a **%**. Any comments preceded by a **%** and coming immediately after the first line in the M-file are returned by the **help** command from within MATLAB. For example:*

```
>>  help stars
STARS(T)       draws stars with parameter t
```

*A line of comments is also indicated by the **%** character. You can have as many of these as you want, just remember that the **help** command only prints the ones immediately following the function definition at the beginning of the M-file. Next, the function defines an internal variable named **n** to be fifty times **t**. This **n** is totally unrelated to any variable already defined in the main MATLAB workspace. Assigning this value will not alter any **n** you had already defined before calling the **stars** function. You can also see how we put a comment in the middle of the function indicating which command actually drew the stars.*

Sometimes, you will need to write several versions of a function before it works properly. When you modify a function definition in an M-file (with Emacs), you must type:

```
>> clear function-name
```

before running the modified function. If you don't, MATLAB won't look at the new version in the M-file and will simply run the old version.

Functions Returning One Value

Functions may return scalar or matrix values. The first line of such functions are of the form:

```
function variable = function-name(argument1,
argument2,...)
```

*where the variable will be set equal to the output value somewhere in the function definition. Here is an example where **y** is set to equal the function **fliplr**.*

```
function y = fliplr(x)
%FLIPLR(X) returns X with row preserved and columns flipped
%in the left/right direction.

%  X = 1 2 3     becomes  3 2 1
%      4 5 6              6 5 4

[m,n] = size(x);
y = x(:,n:-1:1);
```

Functions Returning More Than One Value

If you want to return more than one argument, you can do it by having the function return a vector of values. For example, the following function returns two vectors. The first vector is the prime factors less than 10 of the argument. The second indicates how many times each of these factors is used.

```
function [factors, times] = primefact(n)
%[FACTORS TIMES] = PRIMEFACT(N) find prime factors of n

primes = [2 3 5 7];

for i = 1:4
   temp = n;
     if ( rem(temp,primes(i)) == 0)
         factors = [factors primes(i)];
         times = [times 0];
             while (rem(temp,primes(i)) == 0)
               temp = temp/primes(i);
               times(length(times)) = times(length(times))+1;
             end
     end
end
```

*If you call this function with just one argument (for example, **a = primefact(10)**) the function returns a vector of prime factors, but not the vector indicating how many times each factor appears in the number. To get both vectors, you would call the function as follows:*

```
>> [a b] = primefact(180)
a =
     2    3    5
b =
     2    2    1
```

*This way, both vectors are returned: the primes in **a**, and the number of times each prime was used in **b**. From these results, you can see that 180=2*2*3*3*5.*

Functions Taking a Variable Number of Arguments

*Anytime you call a function, MATLAB defines a variable inside that function called **nargin**. **nargin** is the number of arguments with which the function was called. This allows you to write functions that behave differently when called with different numbers of arguments. If you specify the function **rhino** such that the first line of its M-file file reads:*

```
function hippo = rhino(a,b,c,d)
```

*MATLAB will allow you to call **rhino** with only two arguments, and treat your call to the program as if the last two arguments were optional, and not used. MATLAB sets **nargin = 2**, and executes the function. If the function tries to do something using the variables **c** or **d**, MATLAB generates an error message. MATLAB assumes you will use **nargin** to avoid referring to any optional arguments that were not supplied.*

Script M-files

A script M-file is simply a file of MATLAB commands as you would type them at the >> prompt. A script does not begin with a function line. Its contents are executed in sequence whenever you type their name.

*This also differs from a function M-file in that there are no arguments or output values. Also, the variables inside a script file are the same ones as in the main MATLAB workspace. If you type **n = 1000**, then execute a script that ends with the line **n = 2**, **n** would then be 2, and not 1000. To create a script file, just create a file (e.g. with Emacs) that contains the commands you want executed, and save it in ~/matlab. A script file does not need comments for the **help** command. The filename should still end in .m.*

Scripts are modified the same way functions are. Remember to type

```
clear script-name
```

before running a modified script, or MATLAB will run the old version.

Saving Your Work

*MATLAB's **diary** command records your session in a transcript file that you specify. Typing **diary filename** starts recording all the commands you type in **filename**, including "most of the resulting output" (according to the MATLAB manual). Graphs are not recorded, but almost all printed results are. Typing **diary off** turns the transcript off, and **diary on** turns it back on. The file is created in ASCII format, suitable for editing with Emacs and including in other reports. In fact, this function was used to generate almost all the examples in this document.*

If you want to save all your variables and their current values, type:

```
>> save filename
```

*before you exit MATLAB, and your work will be saved in a binary file named filename.mat. The default file name is matlab.mat in your current working directory; type **cd ~/matlab** at your MATLAB prompt to get to your home MATLAB directory. If you only want to save some of the variables, you can give the **save** command the names of the variables to save. If you type:*

```
>> save  filename  x  y
```

*MATLAB will save just the variables **x** and **y** in the file filename.mat.*

When you start up MATLAB at some future time, you can restore all your variables from the file filename.mat by typing:

```
>> load filename
```

*The command **load**, by itself, loads all the variables saved in matlab.mat.*

A file with the extension .mat is assumed to be binary MATLAB format. To retrieve the variables from filename.mat, the command is:

```
load filename
```

*See **help save** or **help load** for information on formats, including the ASCII text format.*

Interface Controls

Controlling the MATLAB Session

The following table briefly describes the interface controls available to you when using MATLAB:

clc	clear the command window
clear(x)	clear variable x
clf	clear the graphics window
diary	record transcript of session (specify a filename)
load	load previously saved workspace or session
save	save current workspace or session
who	list defined variables
whos	list defined variable with additional information

*MATLAB allows you to clear either the command (text) window, or the graphics window. The **clc** command clears the command window, and give you a fresh >> prompt. The **clf** command clears the graphics window and leaves it blank.*

*The **who** command displays the names of all your variables. The **whos** command gives you the names of all the variables, along with information about each variable's size, number of elements, number of bytes, density, and whether the variable is complex. For example:*

```
>> whos
    Name        Size       Elements     Bytes    Density   Complex

       A        3 by 3            9        72       Full      No
       B        3 by 3            9        72       Full      No
       C        3 by 3            9        72       Full      No
       I        3 by 3            9        72       Full      No
       X        3 by 3            9        72       Full      No
       a        1 by 9            9        72       Full      No
     ans        3 by 1            3        24       Full      No
       b        1 by 9            9        72       Full      No
       p        1 by 4            4        32       Full      No
       q        1 by 7            7        56       Full      No
       r        1 by 10          10        80       Full      No
Grand total is 87 elements using 696 byte
```

You can also run the whos command on a sparse matrix, producing output as in the following example:

```
>> whos
    Name        Size       Elements     Bytes    Density   Complex

       A    4253 by 4253      28831    362984     0.0016      No
```

*If you run the **who** command and see that you have variables you are no longer using, use the **clear** command to remove obsolete variables. For example, typing **clear z** would delete the variable z.*

Working With the Operating System

MATLAB also defines some basic file manipulation utilities, to work with the operating system without having to quit and restart MATLAB:

```
cd        change directory (same as chdir)
delete    delete file (same as rm)
dir       list (same as ls)
ls        list (same as dir)
type      display contents of a file (same as cat)
unix      execute operating system command and return result to MATLAB
!         precedes operating system command called within MATLAB
```

*The command **unix** can be used to execute any operating system command and return the result to MATLAB. For example:*

```
>> [s,date] = unix('date')

s =

     0


date =

Wed Oct 27 11:05:12 EDT 1993
```

*You can use any regular Unix command by preceding it by a **!**. Thus, typing **!date** gives you the current date and time. All pathnames and wildcards should work just as they do at your athena% prompt.*

Modeling Dynamic Systems (SIMULINK)

SIMULINK is a toolbox for the nonlinear simulation of dynamic systems, using a mouse-driven, block-diagram interface. It is an extension to the standard MATLAB, using Motif to create a number of new windows.

SIMULINK lets you create and edit block-diagram representations of dynamic systems. The model can then be analyzed either through Menu commands or from the MATLAB command line. Results from the analysis can be passed to the MATLAB workspace for further work.

To run SIMULINK, type:

```
>> simulink
```

To run some SIMULINK demonstration programs, type:

```
>> demo simulink
```

Regular MATLAB-style help is available for all SIMULINK functions, and the full MATLAB command set is also available from inside SIMULINK.

*The **SIMULINK User's Guide** is available for reference in Barker and Hayden Libraries, and in the Athena Consulting Office in the basement of the Student Center (W20). You may also purchase a copy from the vendor; see the section **For More Help about MATLAB**.*
