1
Fundamentals

MATLAB works with essentially only one kind of object, a rectangular numerical matrix with possible complex elements. In some situations, special meaning is attached to 1-by-1 matrices, which are scalars, and to matrices with only one row or column, which are vectors. Operations and commands in MATLAB are intended to be natural, in a matrix sense, not unlike how they might be indicated on paper.

Let's begin by looking at how matrices can be entered into MATLAB. If you're a "never-ever" user, you may find it useful to invoke MATLAB at this time and follow along.

1.1 Entering Simple Matrices

Matrices can be introduced into MATLAB in several different ways:

- Entered by an explicit list of elements,
- Generated by built-in statements and functions,
- Created in M-files,
- Loaded from external data files.

The MATLAB language contains no dimension statements for type declarations. Storage is allocated automatically, up to the amount available on any particular computer.

The easiest method of entering small matrices is to use an explicit list. The explicit list of elements is separated by blanks or commas, is surrounded by brackets, [and], and uses the semicolon ; to indicate the ends of the rows. For example, entering the statement

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

results in the output

```
A =

     1     2     3
     4     5     6
     7     8     9
```

The matrix A is saved for later use.

Large matrices can be spread across several input lines, with carriage returns replacing the semicolons. Although hardly necessary for a matrix of this size, the above matrix could also have been produced by three lines of input,

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

Matrices can be entered from disk files with names ending in ".m". If a file named gena.m contains three lines of text,

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

Then the statement gena reads the file and generates A.

The load command can read matrices generated by earlier MATLAB sessions or matrices in ASCII form exported from other programs. More on this later.

1.2 Matrix Elements

Matrix elements can be any MATLAB expressions; for example,

```
x = [ -1.3 sqrt(3) (1+2+3)*4/5 ]
```

results in

```
x =

   -1.3000    1.7321    4.8000
```

Individual matrix elements can be referenced with indices inside parentheses, (and). Continuing our example,

```
x(5) = abs(x(1))
```

produces

```
x =

     -1.3000    1.7321    4.8000         0    1.3000
```

Notice that the size of x is automatically increased to accommodate the new element and that the undefined intervening elements are set to zero.

Big matrices can be constructed using little matrices as elements. For example, we could attach another row to our matrix A with

```
r = [10 11 12];
A = [A; r]
```

which results in

```
A =

     1     2     3
     4     5     6
     7     8     9
    10    11    12
```

Little matrices can be extracted from big matrices using :. For example,

```
A = A(1:3, :);
```

takes the first three rows and all the columns of the current A to give us back the original A. (More on : later.)

1.3 Statements and Variables

MATLAB is an expression language. Expressions typed by the user are interpreted and evaluated by the MATLAB system. MATLAB statements are frequently of the form

```
variable = expression
```

or simply

```
expression
```

Expressions are composed from operators and other special characters, from functions, and from variable name. Evaluation of the expression produces a matrix, which is then displayed on the screen and assigned to the variable for future use. If the variable name and the = sign are omitted. A variable with the name ans, which stands for "answer," is automatically created. For example, typing the expression

```
1900 / 81
```

produces

```
ans =

   23.4568
```

A statement is normally terminated with the carriage return or enter key. However, if the last character of a statement is a semicolon, ;, the printing is suppressed, but the assignment is still carried out. This is useful in M-files (more later) and in situations where the result is large enough to make the individual numbers uninteresting. For example,

```
p = conv(r, r);
```

convolves the numbers in r with themselves, but does not display the result.

If the expression is so complicated that the statement will not fit on one line, an ellipsis consisting of three or more periods, …, followed by the carriage return, can be used to indicate that the statement continues on the next line. For example,

```
s = 1 - 1/2 + 1/3 -1/4 + 1/5 - 1/6 + 1/7 ...
      - 1/8 + 1/9 - 1/10 + 1/11 - 1/12;
```

evaluates the partial sum of the series, assigns the sum to the variable s, but does not print anything. The blank spaces around the =, +, and – signs are optional, but are included here to improve readability.

Variable and function names are formed by a letter, followed by any number of letters and digits (or underscores). Only the first 19 characters of a name are remembered.

MATLAB is case-sensitive; it normally distinguishes between upper- and lower-case letters, so a and A are not the same variable. All the function names must be in lower case; inv(A) would invert A, but INV(A) references an undefined function. However, the command casesen makes MATLAB insensitive to the case of letters. In this mode, a and A refer to the same matrix, and INV(a) inverts it.

1.4 Getting Workspace Information

The example statements entered up to this point have created variables that are stored in the MATLAB workspace. Executing

```
who
```

Lists the variables in the workspace:

Your variables are:

```
A         ans       p        r         s         x
```

This shows that six variables have been generated by our examples, including ans, More detailed information showing the size of each of the current variable is obtained with whos which, for our example so far, produces

```
Name      Size           Bytes  Class

  A         3x3             72  double array
  ans       1x1              8  double array
  p         1x5             40  double array
  r         1x3             24  double array
  s         1x1              8  double array
  x         1x5             40  double array
```

Grand total is 24 elements using 192 bytes. Each element of a real matrix requires 8 bytes of memory, so our 3-by-3 matrix A uses 72 bytes and all of our variables use a total of 192 bytes. The amount of remaining free memory depends upon the total amount available in the system and will vary from computer. On computers with virtual memory, an unlimited amount may be available.

The variable ans, along with an unlisted variable eps, have special meaning to MATLAB. They are permanent variables that cannot be cleared.

The variable eps is a tolerance for determining such things as near singularity and rank. Its initial value is the distance from 1.0 to the next largest floating point number. For the IEEE arithmetic used on many personal computers and workstations,

$$\texttt{eps = } 2^{-52}$$

which is approximately 2.2204e-016. The variable eps may be reset to any other value, including zero.

1.5 Numbers and Arithmetic Expressions

Conventional decimal notation, with optional decimal point and leading minus sign, is used for number. A power-of-ten scale factor can be included as a suffix. Here are some examples of legal numbers:

```
3          -99              0.0001
9.6397238  1.60210E-20      6.02252e23
```

On computers using IEEE floating point arithmetic, the relative accuracy of numbers is eps, which is approximately 16 significant decimal digits. The range is roughly 10^{-308} to 10^{308}.

Expressions can be built up using the usual arithmetic operators and precedence rules:

```
+   addition
-   subtraction
*   multiplication
/   right division
\   left division
^   power
```

The operations on matrices described later make it convenient to have the two symbols for division. The scalar expressions 1/4 and 4\1 have the same numerical value, namely 0.25. Parentheses are used in the standard way to alter the usual precedence of arithmetic operations.

Most ordinary elementary mathematical functions found on a good scientific calculator are built-in functions, for example, abs, sqrt, log, and sin. More can be added easily with M-files.

A number of built-in functions simply return commonly used special values. The function pi returns π, precalculated by the program as 4*atan(1). A more provocative way to generate π is

```
imag(log(-1))
```

The function Inf, which stands for infinity, is found in very few calculator systems or computer languages. On some computers, it is made possible by the IEEE arithmetic implemented in a math coprocessor. On other computers, floating point software is used to simulate a coprocessor. One way to generate the value returned Inf is

```
s = 1/0
```

which results in

```
Warning: Divide by zero.

s =

   Inf
```

On machines with IEEE arithmetic, division by zero does not lead to an error condition or termination of execution. It does produce a warning message and a special value that can behave in a sensible manner in subsequent computation.

The variable NaN is an IEEE number related to Inf but has different properties. It stands for “Not a Number” and is produced by calculations such as Inf/Inf or 0/0.

1.6 Complex Numbers and Matrices

Complex numbers are allowed in all operations and functions in MATLAB. Complex numbers are entered using the special functions i and j. Some of us might use

```
z = 3 + 4*i
```

while others might prefer

```
z = 3 + 4*j
```

Another example is

```
w = r*exp(i*theta)
```

There are at least two convenient ways to enter complex matrices. They are illustrated by the statements

```
A = [1 2; 3 4] + i*[5 6; 7 8]
```

and

```
A = [1+5*i  2+6*i;  3+7*i  4+8*i]
```

which produce the same result. When complex numbers are entered as matrix elements within brackets, it is important to avoid any blank spaces, because an expression like 1 + 5*i, with blanks surrounding the + sign, represents two separated numbers. (The same is true of real numbers; a blank before the exponent part in
1.23 e-4 causes an error.)

A built-in function name may be used as variable name, in which case the original built-in function becomes unavailable within the current workspace (or local M-life function) until the variable is cleared. If you use i and j as variables, and overwrite their values, a new complex unit can be generated and used in the usual way:

```
ii = sqrt(-1)
z = 3 + 4*ii
```

1.7 Output Format

The result of any MATLAB assignment statement is displayed on the screen, as well as assigned to the specified variable or to ans if no variable is given. The numeric display format can be controlled using the format command. Format affects only how matrices are displayed not how they are computed or saved (MATLAB performs all computation in double precision).

If all the elements of a matrix are exact integers, the matrix is displayed in a format without any decimal points. For example,

```
x = [ -1 0 1 ]
```

always results in

```
x =

    -1    0    1
```

If at least one of the elements of a matrix is not an exact integer, there are several possible output formats. The default format, called the short format, shows approximately 5 significant decimal digits. The other formats show more significant digits or use scientific notation. As an example, suppose

```
x = [ 4/3 1.2345e-6 ]
```

The formats, and the resulting output for this vector, are:

```
format short

x =

    1.3333   0.0000

format short e

x =
  1.3333e+000  1.2345e-006

format long

x =

   1.33333333333333   0.00000123450000

format long e

x =

     1.333333333333333e+000    1.234500000000000e-006

format hex

x =

   3ff5555555555555   3eb4b6231abfd271

format +

x =

++
```

For the long formats, the last significant digit may appear to be incorrect, but the output is actually an accurate decimal representation of the binary number stored in the computer. With the short and long formats, if the largest element of a matrix is larger than 1000 or smaller than 0.001, a common scale factor is applied to the entire matrix when it is displayed. For example,

```
   x = 1.e20*x
```

multiplies x by 10^{20} and results in the display

```
   x =
      1.0e+020 *

        1.3333    0.0000
```

The + format is a compact way of displaying large matrices. The symbols +, -, and blank are displayed for positive, negative, and zero elements.

One final command, format compact, suppresses many of the line-feeds that appear between matrix displays and allows more information to be shown on the screen.

1.8 The HELP Facility

A HELP facility is available, providing online information on most MATLAB topics. To get a list of HELP topics, type

```
help
```

To get HELP on a specific topic, type help topic. For example,

```
help eig
```

provides HELP information on the use of the eigenvalue function.

```
help []
```

tell how to use brackets to enter matrices, and

```
help help
```

is self-referential, but works just fine.

1.9 Quitting and Saving the Workspace

To quit MATLAB, type quit or exit. Termination of a MATLAB session causes the variables in the workspace to be lost. Before quitting, the workspace may be saved for later use by typing

```
save
```

This saves all variables in a file on disk named matlab.mat . The next time MATLAB is invoked, the workspace may be restored from matlab.mat by executing

```
load
```

save and load may be used with other file names, or to save only selected variables. The command save temp stores the current variables in the file named temp.mat. The command

```
save temp X
```

save only variable X, while

```
save temp X Y Z
```

save X,Y, and Z

```
load temp
```

retrieves all the variables from the file named temp.mat. load and save are also capable of importing and exporting ASCII data files.

1.10 Functions

Much of MATLAB's power is derived from its extensive set of functions. MATLAB has a large number of functions. Some of the functions are intrinsic, or "built-in" to the MATLAB processor itself. Others are available in the library of external M-files distributed with MATLAB (the MATLAB toolbox). And some have been added by individual users, or groups of users, for more specialized applications. It is transparent to the user whether a function is intrinsic or contained in an M-file. This is an important feature of MATLAB; a user can create his own new functions, and they act just like the intrinsic functions built into MATLAB. More on M-files in a later section.

Up until now, we have only seen functions with one input argument and one output argument. MATLAB functions can also be used with multiple arguments. For example,

```
x = sqrt(log(z))
```

shows the nested use of two simple functions. There are MATLAB functions that use two or more input arguments. For example,

```
theta = atan2(y, x)
```

Of course, each one of the arguments could have been an expression.

Some functions return two or more output values. The output values are surrounded by brackets, [and], and separated by commas:

```
 [V, D] = eig(A)
 [y, I] = max(X)
```

The first function here returns two matrices, V and D, the eigenvectors and eigenvalues, respectively, of matrix A. The second example, using max, returns the maximum value y and the index i of the maximum value in vector X.

Functions that permit multiple output arguments can return fewer output arguments. For example, max with one output argument,

```
max(X)
```

returns just the maximum value.

The input or right-hand arguments to a function are never modified by MATLAB. The output, if any, of a function are always returned in left-hand arguments.

2
Matrix Operations

Matrix operations are fundamental to MATLAB; wherever possible they are indicated the way they would be in a textbook or on paper, subject only to the character set limitations of the computer.

2.1 Transpose

The special character prime ‘ (apostrophe) denotes the transpose of a matrix. The statements

```
A = [ 1 2 3; 4 5 6; 7 8 0]
B = A'
```

result in

```
A =

    1     2     3
    4     5     6
    7     8     0

B =

    1     4     7
    2     5     8
    3     6     0
```

and

```
x = [ -1 0 2]'
```

produces

```
x =

   -1
```

```
        0
        2
```

The prime ’ transposes in a formal matrix sense; if Z is a complex matrix, then Z’ is its complex conjugate transpose. This can sometimes lead to unexpected results if used carelessly with complex data. For an unconjugated transpose, use Z.‘ or conj(Z’).

2.2 Addition and Subtraction

Addition and subtraction of matrices are denoted by + and -. The operations are defined whenever the matrices have the same dimensions. For example, with the above matrices, A + x is not correct because A is 3-by-3 and x is 3-by-1. However,

```
C = A + B
```

is acceptable, and results in

```
C =

     2     6    10
     6    10    14
    10    14     0
```

Addition and subtraction are also defined if one of the operands is a scalar, which is a 1-by-1 matrix. In this case, the scalar is added to or subtracted from all the elements of the other operand. For example

```
y = x - 1
```

gives

```
y =

    -2
    -1
     1
```

2.3 Matrix Multiplication

Multiplication of matrices is denoted by *. The operation is defined whenever the "inner" dimensions of the two operands are the same; that is X*Y is permitted if the second dimension of x is the same as the first dimension of Y. For example, the above x and y are both 3-by-1, so the expression x*y is NOT defined and results in an error message. However, several other vector products are defined, and are very useful. The most common is the inner product, also called the dot product or scalar product. This is

```
x'*y
```

which results in

```
ans =

     4
```

Of course, y'*x would give the same result. There are two outer products, which are transposes of each other.

```
x*y'

ans =

     2     1    -1
     0     0     0
    -4    -2     2

y*x'

ans =

     2     0    -4
     1     0    -2
    -1     0     2
```

An element-by-element product is described in the next section. (There is no special provision in MATLAB for computing vector cross products. However, anyone needing cross products can easily write an M-file to compute them.)

Matrix-vector products are special cases of general matrix-matrix products. For our example A and x,

```
b = A*x
```

is allowed and results in the output

```
b =

     5
     8
    -7
```

Naturally, a scalar can multiply, or be multiplied by, any matrix.

```
pi*x

ans =

   -3.1416
         0
    6.2832
```

2.4 Matrix Division

There are two “matrix division” symbols in MATLAB, \ and /. If A is a nonsingular square matrix, then A\B and B/A correspond formally to left and right multiplication of B by the inverse of A, that is inv(A)*B and B*inv(A), but the result is obtained directly without the computation of the inverse. In general,

```
X = A\B  is a solution to  A*X = B

X = B/A  is a solution to   X*A = B
```

Left division, A\B, is defined whenever B has as many rows as A. If A is square, it is factored using Gaussian elimination. The factors are used to solve the equations A*X(:, j) = B(:, j) where B(:, j) denotes the j-th column of B. The result is a matrix X with the same dimensions as B. If A is nearly singular (according to the LINPACK condition estimator, RECOND), a warning message is displayed.

If A is not square, it is factored using Householder orthogonalization with column pivoting. The factors are used to solve the under- or overdetermined equations in a least squares sense. The result is an m-by-n matrix X where m is the number of columns of A and n is the number of columns of B. Each column of X has at most k nonzero components where k is the effective rank of A.

Right division, B/A, is defined in terms of left division by B/A = (A'\B')'.

For example, since our vector b was computed as A*x, the statement

```
z = A\b
```

result in

```
 z =

     -1
      0
      2
```

Sometimes, the use of \ and / to compute least squares solutions to over- or underdetermined systems of equations can cause surprise. It is possible to “divide” one vector by another. For example, with the above vector x and y,

```
s = x\y
```

produces

```
 s =

     0.8000
```

This is because s = 0.8 is the value of the scalar which solves the overdetermined equation xs = y in a least squares sense. We invite the reader to explain why

```
s = y/x
```

gives

```
 s =

          0         0   -1.0000
          0         0   -0.5000
          0         0    0.5000
```

2.5 Matrix Powers

The expression A^p raises A to the p-th power and is defined if A is a square matrix and p is a scalar. If p is an integer greater than one, the power is computered by repeated multiplication. For other values of p, the calculation involves eigenvalues and eigenvectors, such as [V, D] = eigen(A), then

```
A^p = V*D.^p/V
```

If P is a matrix, and a a scalar, a^P is raised to the matrix power P using eigenvalues and eigenvectors. X^P, where both X and P are matrices, is an error.

2.6 Elementary Matrix Functions

In MATLAB, expressions like exp(A) and sqrt(A) are regarded as array operations, defined on the individual elements of A. MATLAB can also calculate matrix transcendental functions, such as the matrix exponential and matrix logarithm. These special functions are defined only for square matrices, are rather difficult and expensive to compute, and sometimes have subtle mathematical properties.

A transcendental mathematical function is interpreted as a matrix function if an “m” is appended to the function name, as in expm(A) and sqrtm(A). As MATLAB is distributed, three of these functions are defined: expm, logm, and sqrtm.

However, the list can be extended by adding more M-file, or using funm. Other elementary matrix functions include: poly, det, trace, kron.

3
Array Operations

We use the term array operations to refer to element-by-element arithmetic operations, instead of the usual linear algebraic matrix operations denoted by the symbols *, /, \, ^, and ‘. Preceding an operator with a period **.** indicates an array or element-by-element operation.

3.1 Array Addition and Subtraction

For addition and subtraction, the array operations and the matrix operations are the same, so + and - can be regarded as either matrix or array operations.

3.2 Array Multiplication and Division

Array, or element-by-element, multiplication is denoted by **.***. If A and B have the same dimension, then A.*B denotes the array whose elements are simply the products of the individual elements of A and B. For example, if

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

then

```
z = x .* y
```

results in

```
 z =

      4    10    18
```

The expression A./ B and A.\B give the quotients of the individual elements.
So

```
z = x ./ y
```

results in

```
z =

    0.2500    0.4000    0.5000
```

3.3 Array Powers

Element-by-element powers are denoted by .^. Here are several examples, using the above vectors x and y. Typing

```
z = x .^ y
```

results in

```
z =

     1    32   729
```

The exponent can be a scalar.

```
z = x .^ 2
z =

     1     4     9
```

Or, the base can be a scalar.

```
z = 2 .^ [x y]

z =

     2     4     8    16    32    64
```

3.4 Relational Operations

There are six relational operators for comparing two matrices of equal dimensions.

```
<  less than
<= less than or equal
```

```
>  greater than
>= greater than or equal
== equal
~= not equal
```

The comparison is done between the pairs of corresponding elements; the result is a matrix of ones and zeros, with one representing TRUE and zero FALSE. For example,

```
2+2 == 4
```

is simply 1.

Relational operators can show the patterns of matrix elements satisfying various conditions. For example, here is the magic square of order 6.

```
A = magic(6)

A =

   35     1     6    26    19    24
    3    32     7    21    23    25
   31     9     2    22    27    20
    8    28    33    17    10    15
   30     5    34    12    14    16
    4    36    29    13    18    11
```

A magic square of order n is an n-by-n matrix constructed from the integers from 1 through n^2 with equal row and column sums. If you stare at this particular matrix long enough, you might notice that the elements that are divisible by 3 occur on every third diagonal. To display this curiosity, we type

```
P = (rem(A, 3) == 0)
```

The double = is the test-for-equality operator, rem (A, 3) is a matrix of remainders, 0 is expanded to a matrix of zeros, and p becomes a matrix of ones and zeros.

```
P =
     0     0     1     0     0     1
     1     0     0     1     0     0
     0     1     0     0     1     0
     0     0     1     0     0     1
     1     0     0     1     0     0
     0     1     0     0     1     0
```

To see the pattern a little more clearly, format + prints matrices in a compact form, with a + where there is a positive element, a – where there is a negative element, and a blank space for zeros.

```
format +

P =

  +  +
+  +
 +  +
  +  +
+  +
 +  +
```

The function find is helpful with relational operators, finding nonzero elements in a 0-1 matrix, and hence the data elements that satisfy some relational condition. For example, if Y is a vector, find (Y < 3.0) returns a vector containing the indices of the elements in Y that are less than 3.0.
The statements

```
i = find(Y > 3.0);
Y(i) = 10*ones(1,length(i));
```

replace with 10.0 all elements in Y greater than 3.0. It will work even if Y is a matrix because a matrix can be referenced as a long column vector with a single subscript.

The relation X == NaN returns zeros everywhere. But it is sometimes necessary to test for NaNs. So a function isnan (X) is provided that returns ones where the elements of X are NaNs and zeros elsewhere. Also useful is isfinite (x), which returns ones for $-\infty < x < \infty$.

3.5 Logical Operations

There are three logical operators that work elementwise and are usually used on 0-1 matrices.

```
&  AND
|  OR
~  NOT
```

The & and | operators compare two scalars, or two matrices of equal dimensions. For matrices, they work elementwise; if A and B are 0-1 matrices, then A & B is another 0-1 matrix representing the logical AND of the corresponding elements of A and B. The logical operators regard anything nonzero as TRUE. They return ones where TRUE and zeros where FALSE.

NOT, or logical complement, is a unary operator. The expression ~A returns zeros where A is nonzero and ones where A is zero. Thus the two expressions

```
P | (~P)
P & (~P)
```

return all ones and all zeros, respectively.

The functions any and all are useful in conjunction with logical operators. If x is a 0-1 vector, any(x) returns 1 if any of the elements of x are nonzero, and returns 0 otherwise. The function all(x) returns a 1 only if all of the elements of x are nonzero. These functions are particularly useful in if statements,

```
if all(A < .5)
   do something
end
```

Because an if wants to respond to a single condition, not a vector of possibly conflicting suggestions.

For matrix arguments, any and all work columnwise to return a row vector with the result for each column. Applying the function twice, as in any (any(A)), always reduces the matrix to a scalar condition.

Here is a summary of the relational and logical functions in MATLAB: any, all, find, exist, isnan, finite, isempty, isstr, and strcmp.

3.6 Elementary Math Functions

A set of elementary mathematical functions is applied on an element-by-element basis to array.　For example,

```
    A = [1 2 3; 4 5 6]
    B = fix(pi*A)
    C = cos(pi*B)
```

produces

```
    A =

         1     2     3
         4     5     6
    B =

         3     6     9
        12    15    18
    C =

        -1     1    -1
         1    -1     1
```

Available functions include the trigonometric functions: sin, cos, tan, asin, acos, atan, atan2, sinh, cosh, tanh, asinh, acosh, atanh and the usual elementary functions: abs, angle, sqrt, real, imag, conj, round, fix, floor, ceil, sign, rem, exp, log, log10.

3.7 Special Math Functions

A number of special functions provide more advanced capabilities: Bessel, gamma, rat, erf, invertf, ellipk, ellipj.　Like the elementary functions, they operate element-by-element when given matrix inputs.

4
Vectors and Matrix Manipulation

The subscripting abilities of MATLAB allow manipulation of rows, columns, individual elements, and subportions of matrices. Central to subscripting are vectors, which are generated using “Colon Notation.” Vectors and subscripting are common MATLAB operations and make it possible to achieve fairly complex data manipulation effects.

4.1 Generating Vectors

The colon, :, is an important character in MATLAB. The statement

```
x = 1:5
```

generates a row vector containing the numbers from 1 to 5 with unit increment. It produces

```
 x =

      1    2    3    4    5
```

Increments other than one can be used.

```
y = 0 : pi/4 : pi
```

results in

```
 y =

         0   0.7854   1.5708   2.3562   3.1416
```

Negative increments are possible.

```
z = 6 : -1 : 1
```

gives

```
z =

     6     5     4     3     2     1
```

The colon notation allows the easy generation of tables. To get a vertical tabular form, transpose the row vector obtained from the colon notation, compute a column of function values, then form a matrix from the two columns. For example

```
x = (0.0: 0.2: 3.0)';
y = exp(-x) .* sin(x);
[x y]
```

produces

```
ans =

         0         0
    0.2000    0.1627
    0.4000    0.2610
    0.6000    0.3099
    0.8000    0.3223
    1.0000    0.3096
    1.2000    0.2807
    1.4000    0.2430
    1.6000    0.2018
    1.8000    0.1610
    2.0000    0.1231
    2.2000    0.0896
    2.4000    0.0613
    2.6000    0.0383
    2.8000    0.0204
    3.0000    0.0070
```

Other vector generation functions include 1inspace, which allows the number of points rather than the increment, to be specified,

```
k = linspace(-pi, pi, 4)
```

```
k =

   -3.1416   -1.0472    1.0472    3.1416
```

and logspace, which generates logarithmically evenly spaced vectors.

4.2 Subscripting

Individual matrix elements may be referenced by enclosing their subscripts in parentheses. An expression used as a subscript is rounded to the nearest integer. For example, given a matrix A:

```
A =

     1     2     3
     4     5     6
     7     8     9
```

The statement

```
A(3,3) = A(1,3) + A(3,1)
```

results in

```
A =

     1     2     3
     4     5     6
     7     8    10
```

A subscript can be vector. If X and V are vectors, then X(V) is [X(V(1)), X(V(2)),…, X(V(n))]. For matrices, vector subscripts allow access to contiguous and noncontiguous submatrices. For example, suppose that A is a 10-by-10 matrix. Then

```
A(1:5,3)
```

specifies the 5-by-1 submatrix, or column vector, that consists of the first five elements in the third column of A. Similarly,

```
A(1:5,7:10)
```

is the 5-by-4 submatrix of elements from the first five rows and the last four columns.

Using the colon by itself in place of a subscript denotes all of the corresponding row or column. For example,

```
A(:,3)
```

is the third column and

```
A(1:5,:)
```

is the first five rows.

Fairly sophisticated effects are obtained using submatrix referencing on both side of an assignment statement. For example,

```
A(:,[3 5 10]) = B(:,1:3)
```

replaces the third, fifth, and tenth columns of A with the first three columns of B.

In general, if v and w are vectors with integer components, then

```
A(v,w)
```

is the matrix obtained by taking the elements of A with row subscripts from v and column subscripts from w. So

```
A(:,n:-1:1)
```

reverses the columns of A and

```
v = 2:2:n;
w = [3 1 4 1 6];
A(v,w)
```

is legal, but probably of questionable utility.

One more feature in this vein is useful in advanced fiddling, A (:) . On the right-hand side of an assignment statement, A (:) denotes all the elements of A strung out in a long column vector. So

```
A = [1 2; 3 4; 5 6]
b = A(:)
```

results in

```
A =

     1     2
     3     4
     5     6
b =

     1
     3
     5
     2
     4
     6
```

On the left-hand side of an assignment statement, A(:) can be used to reshape or resize a matrix. To do this, A must already exist. Then A (:) denotes a matrix with the same dimensions as A, but with new contents from the right-hand side. For example, the above A has three rows and two columns, so

```
A(:) = 11:16
```

reshapes the six-element row vector into a 3-by-2 matrix.

```
A =
    11    14
    12    15
    13    16
```

4.3 Subscripting with 0-1 Vectors

The 0-1 vectors usually created from relational operations can be used to reference submatrices. Suppose A is an m-by-n matrix and L is a length m vector of zeros and ones. Then

```
A(L,:)
```

specifies the rows of A where the elements of L are nonzero.

Here is how outliers, those elements greater than 3 standard deviations, could be removed from a vector:

```
x = x(x <= 3*std(x));
```

Similarly,

```
L = X(:,3) > 100;
X = X(L,:);
```

replaces X with those rows of X whose third column is greater than 100.

4.4 Empty Matrices

The statement

```
x = [ ]
```

assigns a matrix of dimension zero-by-zero to x. Subsequent use of this matrix will NOT lead to an error condition; it will propagate empty matrices. This is different from the statement

```
clear x
```

which removes x from the list of current variables. Empty matrices do exist in the workspace; they just have zero size. The function exist can be used to test for the existence of a matrix (or a file for that matter), while isempty indicates if a matrix is empty.

It is possible to generate empty vector. If n is less than 1, then 1:n contains no elements and so

```
x = 1:n
```

is a complicated way of creating an empty x.

More importantly, an efficient way of removing rows and columns of a matrix is to assign them to an empty matrix. So

```
A(:,[2 4]) =[ ]
```

deletes columns 2 and 4 of A.

Certain matrix functions will return mathematically plausible values if given empty matrices. These include det, cond, prod, sum, and possibly others. For example, prod, det, and sum return 1, 1, and 0, respectively, when given null matrix arguments.

4.5 Special Matrices

A collection of functions generates special matrices that arise in linear algebra and signal processing: company, diag, gallery, hadamard, hankel, hilb, invhilb, magic, pascal, toeplitz, vander.

For example, generate a companion matrix associated with the polynomial $x^3 - 7x + 6$

```
p = [1 0 -7 6];
a = compan(p)

a =

     0     7    -6
     1     0     0
     0     1     0
```

The eigenvalues of a are the roots of the polynomial.

```
eig(a)

ans =

  -3.0000
   2.0000
   1.0000
```

A Toeplitz matrix with diagonal disagreement is

```
c = [ 1 2 3 4 5];
r = [1.5 2.5 3.5 4.5 5.5];
t = toeplitz(c, r)
t =

   1.0000    2.5000    3.5000    4.5000    5.5000
   2.0000    1.0000    2.5000    3.5000    4.5000
   3.0000    2.0000    1.0000    2.5000    3.5000
   4.0000    3.0000    2.0000    1.0000    2.5000
   5.0000    4.0000    3.0000    2.0000    1.0000
```

Other functions generate less interesting, but more useful, utility matrices, such as zeros, ones, rand, eye, linspace, logspace, and meshdom.

These functions include eye (size(A)), which returns an identity matrix of the same size as A. The catchy name is used because I and i are often used as subscripts or as sqrt(-1).

The group also includes zeros and ones, which generate constant matrices of various sizes, and rand, which generates matrices of uniformly or normally distributes random elements. For example, to generate a random 4-by-3 matrix

```
A = rand(4,3)

A =

   0.9501    0.8913    0.8214
```

```
    0.2311    0.7621    0.4447
    0.6068    0.4565    0.6154
    0.4860    0.0185    0.7919
```

4.6 Building Larger Matrices

Large matrices can be formed from small matrices by surrounding the small matrices with brackets, [and]. For example,

```
C = [A  eye(4); ones(size(A))  A^2]
```

creates one such larger matrix, assuming that A has four rows. The smaller matrices in this type of construction must be dimensionally consistent or an error message results.

9.7 Matrix Manipulation

Several functions will rotate, flip, reshape, or extract portions of a matrix: rot90, fliplr, flipud, diag, tril, triu, reshape, .’, and :.

For example, to reshape a 3-by-4 matrix into a 2-by-6 matrix:

```
a =

     1     4     7    10
     2     5     8    11
     3     6     9    12
b =

     1     3     5     7     9    11
     2     4     6     8    10    12
```

The three functions diag, triu, and tril provide access to diagonal, upper triangular, and lower triangular portions of matrices. For example,

```
tril(rand(4,3))
```

produces

```
ans =

   0.9218        0        0
   0.7382   0.9169        0
   0.1763   0.4103   0.8132
   0.4057   0.8936   0.0099
```

Also very useful are size and length. The function size returns a two-element vector containing the row and column dimensions of a matrix. If a variable is known to be a vector, length returns the length of the vector, or max (size (V)).

5
Data Analysis

This section presents an introduction to data analysis using MATLAB and describes some elementary statistical tools. More powerful techniques are available using the linear algebra and signal processing functions.

5.1 Column-oriented Analysis

Matrices are, of course, used to hold all data, but this leaves a choice of orientation for multivariate data. By convention, the different variables in a set of data are put in columns, allowing observations to vary down through the rows. A data set consisting of 50 samples of 13 variables would be stored in a matrix of size 50-by-13.

Starting an example, the ever-present Longley econometric data consist of the variables
1. GNP deflator
2. GNP
3. Unemployment
4. Armed Forces
5. Population
6. Year
7. Employment

In general, there are several methods for getting data into MATLAB. Assuming that the data are not already in a machine-readable form, the easiest way to enter the data is using a text editor or word processor. If we create a file called longley.m that contains a statement to assign data to ldata. We can execute the command longley. This accesses Longley.m and creates the matrix called 1data (or any other name of your choice) in the workspace. You could try entering this matrix interactively, but the chances are remote that you would get it correct the first time. If you were to make a mistake resulting in an error message, there would be no way to correct it without starting over. (Unless your version of MATLAB has provisions for editing previous lines).

If there are more observations than will fit across the screen, rows can be continued to the next line using the ellipsis … consisting of three periods. The matrix could also be entered in blocks of columns and the whole thing pieced together at the end.

For data entered in this columnwise fashion, a group of functions provides basic data analysis capabilities: max, min, mean, median, std, sort, sum, prod, cumsum, cumprod, diff, hist, corrcoef, cov, cplxpair.

For vector arguments, these functions don’t care whether the vectors are oriented in a row or column direction. For array arguments, the functions operate in a column-oriented fashion on the data in the arrays. This means, for example, that if max is applied to an array, the result is a row vector containing the maximum values over each column.

Thus if

```
A =

    9    8    4
    1    6    5
    3    2    7
```

then

```
m = max(A)
mv = mean(A)
s = sort(A)
```

results in

```
m =

    9    8    7
mv =

   4.3333   5.3333   5.3333
s =

    1    2    4
    3    6    5
    9    8    7
```

More functions can be added to this list using M-files, but when doing so, care must be exercised to handle the row vector case. If you are writing your own column-oriented M-files, you should look at how this is accomplished in other M-files, for example mean.m and diff.m.

5.2 Regression and Curve Fitting

Before attempting to fit a curve to data, the data should be normalized. Normalization can improve the accuracy of the final results. One way to normalize is to remove the mean

```
[n, p] = size(X);
e = ones(n,1);
X = X – e*mean(X);
```

and to normalize to unit standard deviation

```
X = X ./ (e*std(X));
```

6
Function Functions

A class of functions in MATLAB works not with numerical matrices, but with mathematical functions. These function functions include:

- Numerical integration
- Nonlinear equations and optimization
- Differential equation solution

Mathematical functions are represented in MATLAB by function M-files. For example, the function

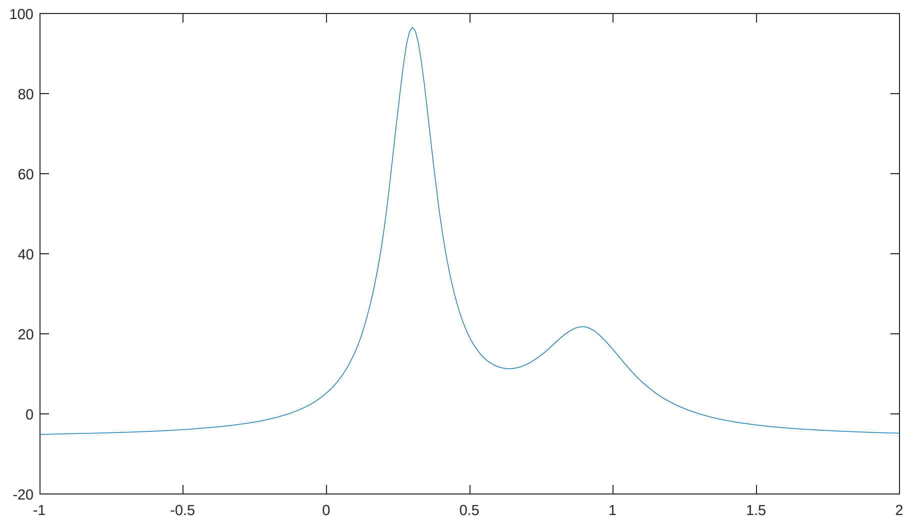
$$humps(x) = \frac{1}{(x-.3)^2+.01} + \frac{1}{(x-.9)^2+.04} - 6$$

is made available to MATLAB by creating an M-file called humps.m:

```
function y = humps(x)
y = 1 ./ ((x-.3).^2 + .01) + 1 ./ ((x-.9).^2 + .04) - 6;
```

A graph of the function is:

```
x = -1 :.01:2;
plot(x,humps(x))
```



6.1 Numerical Integration

The area beneath humps(x) can be determined by numerically integrating humps(x), a process referred to as quadrature. To integrate humps from 0 to 1:

```
q = quad('humps',0,1)  or  q = quad(@humps,0,1)

q =

   29.8583
```

The two MATLAB functions for quadrature are: quad, quad8.

Notice that the first argument to quad is a quoted string containing the name of a function. This is why we call quad a function function—it is a function that operates on other functions.

6.2 Nonlinear Equations and Optimization

The function functions for nonlinear equations and optimization include: fminbnd, fminsearch, fsolve, fzero. Continuing our example, the location of the minimum of humps(x) in the region from 0.5 to 1 is computed with fminbnd:

```
xm = fminbnd(@humps,.5,1)

xm =

    0.6370
```

Its value at the minimum is:

```
y = humps(xm)

y =

   11.2528
```

The location of the minimum of humps(x) can also be computed with fminsearch:

```
xm = fminsearch(@humps,0.5)

xm =

   0.6370
```

From looking at the graph, it is apparent that humps has two zeros. The location of the zero near x = 0 is:

```
xz1 = fzero(@humps, 0)

xz1 =

  -0.1316
```

The zero near x = 1 is at:

```
xz1 = fzero(@humps, 1)

xz1 =

   1.2995
```

6.3 Differential Equation solution

MATLAB’s functions for solving ordinary differential equations are: ode23, ode45.

Consider the second order differential equation known as the Van der Pol equation

$$\ddot{x} + (x^2 - 1)\dot{x} + x = 0$$

We can rewrite this as a system of coupled first order differential equations

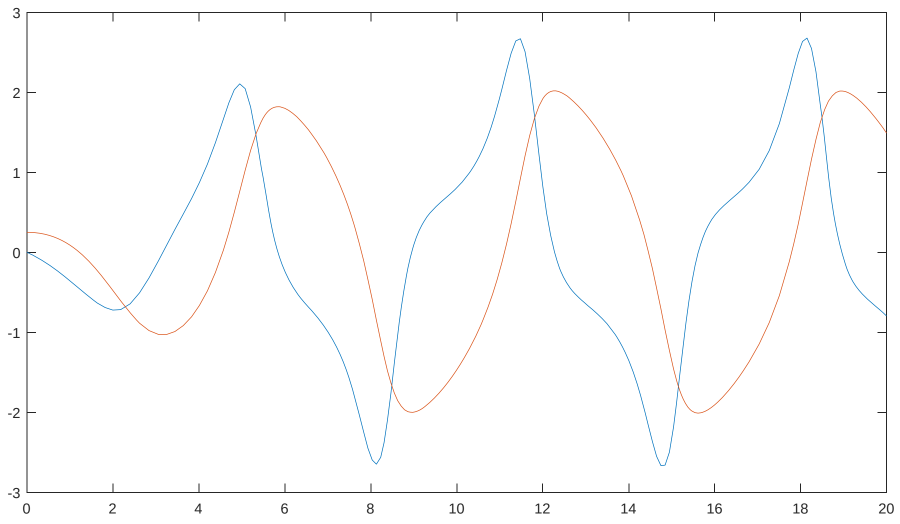
$$\dot{x}_1 = x_1(1 - x_2^2) - x_2$$
$$\dot{x}_2 = x_1$$

The first step toward simulating this system is to create a function M-file containing these differential equations. We can call it vdpol.m:

```
function xdot = vdpol(t,x)
xdot = zeros(2,1);
xdot(1) = x(1) .* (1 - x(2).^2) - x(2);
xdot(2) = x(1);
```

To simulate the differential equation defined in vdpol over the interval $0 \le t \le 20$ invoke ode45

```
t0 = 0; tf = 20;
x0 = [ 0 0.25]'; % Initial conditions
[t,x] = ode45(@vdpol,[t0 tf],x0);
plot(t,x)
```



7
Graphing

Scientific and engineering data are examined graphically in MATLAB using “graph paper” commands to create plots on the screen. There are many different types of “graph paper” from which to choose: plot, loglog, semilogx, semilogy, polar, mesh, contour, bar, and stairs.

Once a graph is on the screen, the graph may be labeled, titled, or have grid lines drawn in: title, xlabel, ylabel, text, gtext, and grid.

There are commands for manual axis scaling and graph control: axis, hold, shg, clg, subplot, and ginput.

And there are commands for sending hardcopy to a printer: print, prtsc, and meta.

7.1 X-Y Plots

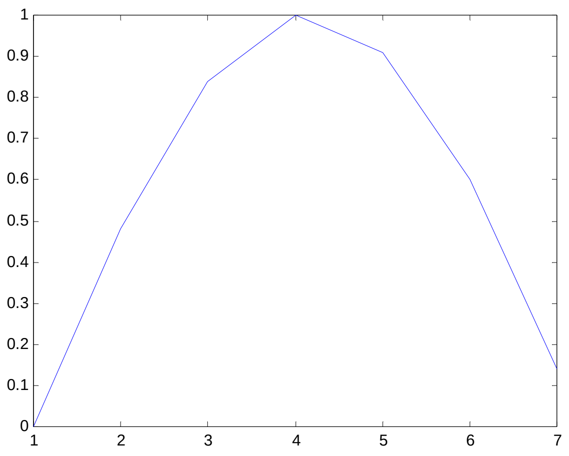
The plot command creates linear x-y plots.   Once plot is mastered, logarithmic and polar plots are created by substituting the words loglog, semilogx, semilogy, or polar for plot.   All five commands are used the same way; they only affect how the axis is scaled and how the data are displayed.

7.2 Basic Form

If Y is a vector plot(Y) produces a linear plot of the elements of Y versus the index of the elements of Y.   For example, to plot the numbers {0., .48, .84, 1., .91, .6, .14}, enter them into a vector and execute plot:

```
    Y = [0.  .48  .84  1.  .91  .6  .14];
    plot(Y)
```

This results in a graph on your screen:



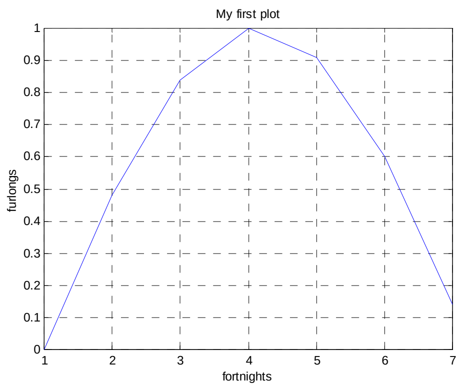
Notice that the data are auto-scaled and that X- and Y-axes are drawn.

At this point, depending upon the exact hardware you are using, the screen full of commands that you have typed in may have vanished to make way for the graph display. MATLAB has two displays, a graph display and a command display. Some hardware configurations allow both to be seen simultaneously, while others allow only one to be seen at a time. If the command display is no longer there, it can be brought back by pressing any key on the keyboard.

Once the command display has been brought back, a graph title, X and Y labels, and grid lines can be put on the plot by successively entering the commands

```
title('My first plot')
xlabel('fortnights')
ylabel('furlongs')
grid
```

This results in:

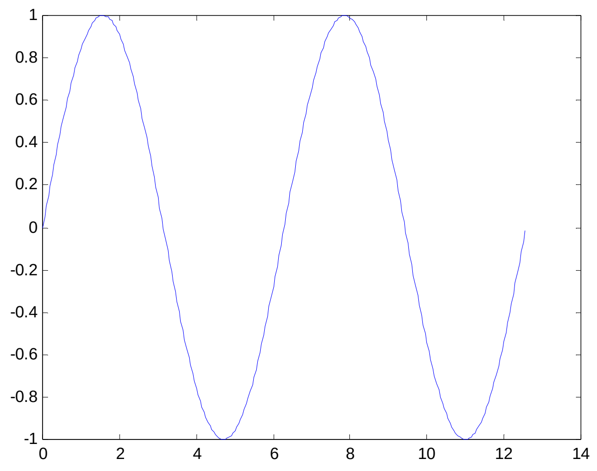


The function gtext ('text') allows a mouse or arrow keys to position a cross-hair on the graph, at which point the text will be placed when any key is pressed.

If X and Y are vectors of the same length, the command plot(X, Y) draws an x-y plot of the elements of X versus the elements of Y. For example,

```
    t = 0:.05:4*pi;
    y = sin(t);
    plot(t,y)
```

results in



7.3 Multiple Lines

There are two ways to plot multiple lines on a single graph. The first is to give plot two arguments, as in plot (X, Y), where either X, Y, or both, are matrices. Then:

1. If Y is a matrix, and X a vector, plot (X, Y) successively plots the rows or columns of Y versus the vector X, using a different line type for each. The row or column "direction" of Y is selected that has the same number of elements as vector X. If Y is a square matrix, the column direction is arbitrarily used.
2. If X is a matrix, and Y a vector, then the above rules are applied except the family of lines from X are plotted versus vector Y.
3. If X and Y are both matrices of the same size, plot (X, Y) plots the columns of X versus the columns of Y.
4. If no X is specified, as in plot (Y), where Y is a matrix, then lines are plotted for

each column of Y versus the row index.

The second, and easier, way to plot multiple lines on a single graph is to use plot with multiple arguments:

```
Plot(X1, Y1, X2, Y2,…, Xn, Yn)
```

The variables X1, Y1, X2, Y2, etc. are pairs of vectors. Each x-y pair is graphed, generating multiple lines on the plot. Multiple arguments have the benefit of allowing vectors of differing lengths to be displayed on the same graph. As before, each pair uses a different line type.

7.4 Line and Mark Styles

7.4.1 Type

The linetypes used on a graph may be controlled if the defaults are not satisfactory. Point plots using various symbols may also be selected. For example,

```
plot(X, Y, ‘x’)
```

Draws a point plot using x-mark symbols while

```
plot(X1, Y1, ’:’, X2, Y2, ’+’)
```

uses a dotted line for the first curve and the plus symbol + for the second curve. Other line and point types are:

```
LINE TYPES       POINT TYPES
solid  -         point   .
dashed --        plus    +
dotted :         star    *
dashdot -.       circle  o
                 x-mark  x
```

7.4.2 Color

On systems that support color, line- and mark-colors may be specified in a manner similar to line- and mark-types. For example, the statements

```
plot(X, Y, ‘t’)
plot(X, Y, ‘+g’)
```

use a red line on the first graph and green +-marks on the second. Other colors are:

red	r
green	g
blue	b
white	w
invisible	i

If your hard-copy device does not support color, the various colors on the interactive display are mapped to different linetypes for output.

7.5 Imaginary and Complex Data

When the arguments to plot are complex (have nonzero imaginary parts), the imaginary part is ignored except when plot is given a single complex argument.
For this special case, the result is a shortcut to a plot of the real part versus the imaginary part. Thus plot (Z), when Z is a complex vector of matrix, is equivalent to plot (real (Z), imag (Z)).

To plot multiple lines in the complex plane, there is no shortcut, and the real and imaginary parts must be taken explicitly.

7.6 Logarithmic, Polar, and Bar Plots

The use of loglog, semilogx, semilogy, and polar is identical to the use of plot. These command allow data to be plotted on different types of “graph paper,” i.e., in different coordinate systems:

- polar (theta, rho) is a plot in polar coordinates of the angle theta, in radians, versus the radius rho. Subsequent use of the grid command draws polar grid lines.
- loglog is a plot using $\log_{10}$-$\log_{10}$ scales.
- semilogx is a plot using semi-log scales. The x-axis is $\log_{10}$ while the y-axis is linear.
- semilogy is a plot using semi-log scales. The y-axis is $\log_{10}$ while the x-axis is linear.

bar (x) displays a bar chart of the elements of vector x. bar does not accept multiple arguments. Similar, but missing the vertical lines, is stairs, which produces a stairstep plot useful for graphing sampled data systems.

7.7 3-D Mesh Surface and Contour Plots

The statement mesh (Z) creates a three dimensional perspective plot of the elements in matrix Z. A mesh surface is defined by the Z coordinates of points above a rectangular grid in the x-y plane. The plot is formed by joining adjacent points with straight lines.

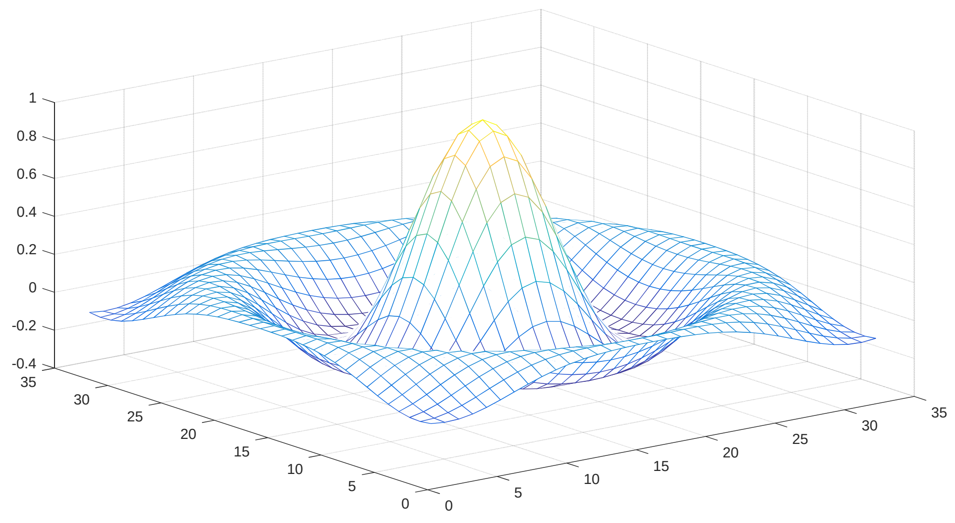
mesh can be used to visualize large matrices that are otherwise too large to print out in numerical form. It can also be used to graph functions of two variables.

The first step in displaying a function z = f (x, y) of two variables is to generate special X and Y matrices that consist of repeated rows and columns, respectively, over the domain of the function. The function can then be evaluated directly and graphed.

Consider the sin(r)/r or sinc function that results in the “sombrero” surface that everybody likes to show. One way to create this is:

```
x = -8: .5: 8;
y = x';
X = ones(size(y))*x;
Y = y*ones(size(x));
R = sqrt(X.^2 + Y.^2) + eps;
Z = sin(R)./R;
mesh(Z)
```

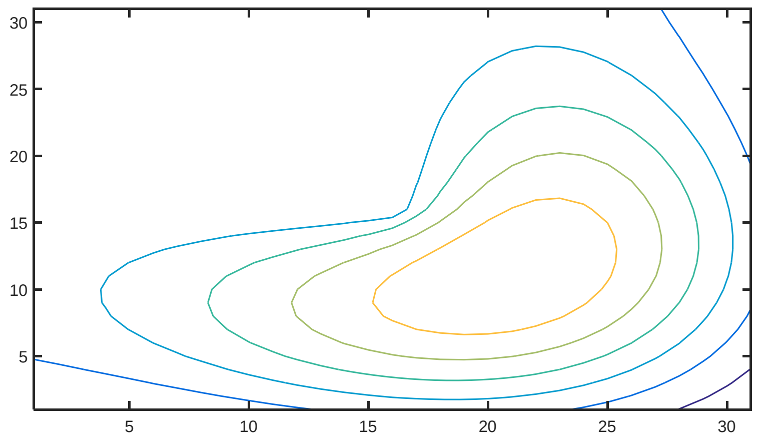
The first command defined the x-domain over which the function is to be evaluated. The third statement creates a matrix X of repeated rows. After generating corresponding Y, a matrix R is created containing the distance from the center of the matrix, which is the origin. Forming the sinc function and applying mesh results in



What does an identity matrix look like as a mesh surface? Try mesh (eye (14)). For an easier method of generating the special X and Y matrices required to evaluate functions of two variables, see meshdom.

A contour plot is an alternative to mesh plot for viewing the contents of a matrix. A contour plot of the L-shaped membrance is

```
z = membrane(1, 15, 9, 2);
contour(z)
```



7.8 Screen Control

MATLAB conceptually has two displays, a graph window and a command window.

Your particular hardware configuration may allow both to be seen simultaneously in different windows, or it may allow only one to be seen at a time. Several commands are available to switch back and forth between the windows, and /or erase the windows, as required:

shg	show graph window
any key	bring back command window
clc	clear command window
clg	clear graph window
home	home command cursor

For example, if during your MATLAB session only the command display is on the screen, typing shg will recall the last plot that was drawn on the graphing display.

By default, hardware configurations that cannot display both the command screen and the graph screen simultaneously will pause in graphic mode after a plot is drawn and wait for a key to be pressed.

It is possible to split the graph window into multiple partitions, in order to show several plots at the same time. The statement subplot (mnp) breaks the graph window into an m-by-n grid and uses the p-the box for the subsequent plot. For example,

```
subplot(211), plot(abs(y))
subplot(212), plot(angle(y))
```

breaks the screen in two, plots the magnitude of a complex vector in the top half, and plots the phase in the bottom half. The command subplot (111), or just subplot, returns to the default single whole-screen window.

7.9 Manual Axis Scaling

In certain situations, it may be desirable to override the automatic axis scaling feature of the plot command and to manually select the plotting limits. Executing axis, by itself, freezes the current axis scaling for subsequent plots. Typing axis again resumes auto-scaling. axis returns a 4-element row vector containing the [x_min, x_max, y_min, y_max] from the last plot. axis(V), where V is a 4-element vector, sets the axis scaling to the prescribed limits.

A second use of axis is to control the aspect ratio of the plot on the screen. axis ('square') sets the plot region on the screen to be square. With a square aspect ratio, a line with slope 1 is at a true 45 degrees, not skewed by the irregular shape of the screen. Also, circles, like plot (sin(t), cos(t)), look like circles instead of ovals. axis ('normal') sets the aspect ratio back to normal.

The hold command holds the current graph on the screen. Subsequent plot commands will add to the plot, using the already established axis limits and retaining the previously plotted curves. hold remains in effect until issued again.

7.10 Hard copy

The prtscr command provides printing capabilities.

prtscr	prtsc initiates a graph window screen dump, like Shift-Prtsc, and allows it to be done in an M-file or in a for loop. In general, this results in a low-resolution plot, because the pixels on the screen are transferred to pixels on a printer.

The easiest way to obtain graphics hard copy is to hold the Shift key down and to press the PrtSc key. This sends a screen dump of the picture in the graph window to the printer.

8
Control Flow

MATLAB has control flow statements like those found in most computer languages. The control flow statements carry MATLAB beyond the level of a simple desk calculator, allowing it to be used as a complete high-level matrix language.

8.1 FOR Loops

MATLAB has its own version of the "DO" or "FOR" loop found in computer languages. It allows a statement, or group of statements, to be repeated a fixed, predetermined number of times. For example

```
for i = 1:n, x(i) = 0, end
```

assigns 0 to the first n elements of x. If n is less than 1, the construction is still legal, but the inner statement will not be executed. If x does not yet exist, or has fewer than n elements, then additional space will be allocated automatically.

The loops can be nested and are usually indented for readability.

```
for i = 1:m
  for j = 1:n
    A(i,j) = 1/(i+j-1);
  end
end
A
```

The semicolon terminating the inner statement suppresses repeated printing, while the A after the loops displays the final result.

An important point: Each for must be matched with an end. If you simply type

```
for i = 1:n, x(i) = 0
```

The system will patiently wait for you to enter the remaining statements in the body of the loop. Nothing will happen until you type the end.

For another example, assume

```
t =

    -1
     0
     1
     3
     5
```

And that we want to generate a Vandermonde matrix, a matrix whose columns are powers of elements of t.

```
A =

     1    -1     1    -1     1
     0     0     0     0     1
     1     1     1     1     1
    81    27     9     3     1
   625   125    25     5     1
```

Here is the most obvious double loop.

```
n = max(size(t));
for j = 1:n
   for i = 1:n
      A(i,j) = t(i)^(n-j);
   end
end
```

But the following single loop with vector operations is significantly faster and also illustrates the fact that for loops can go backwards.

```
A(:,n) = ones(n,1);
for j = n-1:-1:1
   A(:,j) = t .* A(:,j+1);
end
```

The general form of a for loop is

```
for v = expression
  statements
end
```

The expression is actually a matrix, because that's all there is in MATLAB. The columns of the matrix are assigned one by one to the variable v and then the statements are executed. A clearer way of accomplishing almost the same thing is

```
E = expression;
[m,n] = size(E);
for j = 1:n
  v = E(:,j);
  statements
end
```

Usually, the expression is something like m:n, or m:i:n, which is a matrix with only one row, and so its columns are simply scalars. In this special case, the for loop is like the "FOR" or "DO" loops of other languages.

8.2 WHILE Loops

MATLAB also has its version of the "WHILE" loop, which allows a statement, or a group of statements, to be repeated an indefinite number of times, under control of a logical condition. Here is a simple problem to illustrate a while loop. What is the first integer n for which n! (that is n factorial) is a 100 digit number? The following while loop will find it. If you don't already know the answer, you can run this yourself.

```
n = 1;
while prod(1:n) < 1.e100, n = n+1; end
n
```

A more practical computation illustrating while is the calculation of the exponential of a matrix, called expm (A) in MATLAB. One possible definition of the exponential function is the power series,

```
expm(A) = I + A + A^2/2! + A^3/3! + …
```

It is reasonable to use this for the actual computation as long as the elements of A are not too large. The idea is to sum as many terms of this series as are needed to produce a result that would not change if more terms are added in finite precision machine arithmetic.

In the following loop, A is the given matrix, E will become the desired exponential, F is an individual term in the series and k is the index of that term. The indented statements will be repeated until F is so small that adding it to E doesn't change E.

```
E = zeros(size(A));
F = eye(size(A));
k = 1;
while norm(E+F-E,1) > 0
   E = E + F;
   F = A*F/k;
   k = k+1;
end
```

If we want to compute the array or element-by-element exponential exp (A) instead, we just have to change the initialization of F from eye (A) to ones (A) and change the matrix multiplication A*F to array multiplication A·*F.

The general form of a while loop is

```
while expression
   statements
end
```

The statements are executed repeatedly as long as all of the elements in the expression matrix are nonzero. The expression matrix is almost always a 1-by-1 relational expression, so nonzero corresponds to TRUE. When the expression matrix is not scalar, it can be reduced using the functions any and all.

8.3 IF and BREAK Statements

Here are a couple of examples illustrating MATLAB's if statements. The first shows how a computation might be broken down into three cases, depending upon the sign and parity of n.

```
if n < 0
  disp(‘n is negative.’)
elseif rem(n,2) == 0
  disp(‘n is even.’)
else
  disp(‘n is odd.’)
end
```

The second example involves a fascinating problem from number theory. Take any positive integer. If it is even, divide it by 2; if it is odd, multiply it by 3 and add 1. Repeat this process until your integer becomes a one. The fascinating unsolved problem is: Is there any integer for which the process does not terminate? Our MATLAB program illustrates while and if statements. It also shows the input function, which prompts for keyboard input, and the break statement, which provides an exit jump out of loops.

```
% Classic “3n+1” problem from number theory.
while 1
  n = input(‘Enter n, negative quits. ‘);
  if n <= 0, break, end
  while n > 1
    if rem(n,2) == 0
      n = n/2
    else
      n = 3*n+1
    end
  end
end
```

Can you make this run forever?

9
M-files: Scripts and Functions

MATLAB is usually used in a command-driven mode; when single-line commands are entered, MATLAB processes them immediately and display the results. MATLAB is also capable of executing sequences of commands that are stored in files. Together, these two modes form an interpretive environment.

Disk files that contain MATLAB statements are called M-files because they have a file type of ".m" as the last part of the filename (the ".m" is optional on the macintosh). For example, a file named bessel.m might contain MATLAB statements that evaluate Bessel functions.

An M-file consists of a sequence of normal MATLAB statements, possibly including references to other M-files. An M-file can call itself recursively.

One use of M-files is to automate long sequences of commands. Such files are called script files or just scripts. A second type of M-file provides extensibility to MATLAB. Called function files, they allow new functions to be added to the existing functions. Much of the power of MATLAB derives from this ability to create new functions that solve user-specific problems.

Both types of M-files, scripts and functions are ordinary ASCII text files, and are created using an editor or word processor of your choice.

9.1 Script Files

When a script is invoked, MATLAB simply executes the commands found in the file, instead of waiting for input from the keyboard. The statements in a script file operate globally on the data in the workspace. Scripts are useful for performing analyses, solving problems, or doing designs that require such long sequences of commands that they become cumbersome to do interactively.

As an example, suppose the MATLAB commands

```
% An M-file to calculate Fibonnaci numbers
f = [1 1]; i = 1;
while f(i) + f(i+1) < 1000
```

```
f(i+2) = f(i) + f(i+1);
i = i + 1;
end
plot(f)
```

are contained in a file called fibno.m. Typing the statement fibno causes MATLAB to execute the commands, calculate the first 16 Fibonnaci numbers, and create a plot. After execution of the file is complete, the variables f and i remain in the workspace.

The demos supplied with MATLAB are good examples of using scripts to perform more complicated tasks. The script named startup.m is executed automatically when MATLAB is invoked. Physical constants, engineering conversion factors, or anything else you would like predefined in your workspace may be put in these files. On multi-user or networked systems, a script called matlab.m is reserved for use by the system manager. It can be used to implement system-wide definitions and messages.

9.2 Function Files

If the first line of an M-file contains the word "function", the file is a function file. A function differs from a script in that arguments may be passed, and that variables defined and manipulated inside the file are local to the function and do not operate globally on the workspace. Function files are useful for extending MATLAB, that is, creating new MATLAB functions using the MATLAB language itself.

Here is a simple example. The file mean.m on your disk contains the statements:

```
function y = mean(x)
% MEAN    Average or mean value. For vectors,
%         MEAN(x) returns the mean value. For
%         matrices, MEAN(x) is a row vector
%         containing the mean value of each column.
[m,n] = size(x);
if m == 1
  m = n; % Handle isolated row vector.
end
y = sum(x) / m;
```

The existence of this disk file defines a new function called mean. The new function mean is used just like any other MATLAB function. For example, if Z is a vector of the integers from 1 to 99,

```
Z = 1:99;
```

The mean value is found by typing

```
mean(Z)
```

which results in

```
ans =

    50
```

Let’s examine some of the details of mean.m:

- The first line declares the function name, the input arguments, and the output arguments. Without this line, the file would be a script file, instead of a function file.
- The % symbol indicates that the rest of a line is a comment and should be ignored.
- The first few lines document the M-file and will be displayed if help mean is typed.
- The variables m, n, and y are local to mean and will not exist in the workspace after mean has finished. (Or, if they previously existed, they will be unchanged.)
- It was not necessary to put our integers from 1 to 99 in a variable with the name x. In fact, we used mean with a variable called Z. The vector Z that contained the integers from 1 to 99 was passed or copied into mean where it became a local variable named x.

A slightly more complicated version of mean called stat calculates standard deviation too:

```
function [mean,stdev] = stat(x)
[m,n] = size(x);
if m == 1
  m = n; % Handle isolated row vector.
end
mean = sum(x) /m;
stdev = sqrt(sum(x.^2) / m – mean.^2);
```

stat illustrates that it is possible to return multiple output arguments.

A function that calculates the rank of a matrix uses multiple input arguments:

```
function r = rank(x,tol)
% rank of a matrix
s = svd(x);
if (nargin == 1)
   tol = max(size(x)) * s(1) * eps;
end
r = sum(s > tol);
```

This example demonstrates the use of the permanent variable nargin to find the number of input arguments. The variable nargout, although not used here, contain the number of output arguments.

Some helpful hints:
When an M-function file is invoked for the first time during a MATLAB session, it is compiled and placed into memory. It is then available for subsequent use without recompilation. It remains in memory for the duration of the session, unless you run low on free memory, in which case it may be cleared automatically.

The what command shows a directory listing of the M-files that are available in the current directory on your disk, type lists M-files, and ! is used to invoke your editor, allowing you to create or modify M-files.

In general, if you input the name of something to MATLAB, for example by typing whoopie, the MATLAB interpreter goes through the following steps:

1. Looks to see if whoopie is a variable.
2. Looks in the current directory for a file named whoopie.m.
3. Checks if whoopie is a built-in function.
4. Looks in the directories specified by the environment symbol MATLABPATH for a file named whoopie.m.

Thus MATLAB first tries to use whoopie as a variable, if it exists before typing to use whoopie as a function.

9.3 Anonymous Functions and Function Handles

An anonymous function is a function that is not stored in a program file, but is associated with a variable whose data type is function_handle. Anonymous functions can accept inputs and return outputs, just as standard functions do. However, they can contain only a single executable statement.

For example, create a handle to an anonymous function that finds the square of a number:

```
sqr = @(x) x.^2;
```

Variable sqr is a function handle. The @ operator creates the handle, and the parentheses () immediately after the @ operator include the function input arguments. This anonymous function accepts a single input x, and implicitly returns a single output, an array the same size as x that contains the squared values.

Find the square of a particular value (5) by passing the value to the function handle, just as you would pass an input argument to a standard function.

```
a = sqr(5)

a =

  25
```

Many MATLAB functions accept function handles as inputs so that you can evaluate functions over a range of values. You can create handles either for anonymous functions or for functions in program files. The benefit of using anonymous functions is that you do not have to edit and maintain a file for a function that requires only a brief definition. One example is to find the integral of the sqr function from 0 to 1 by passing the function handle to the integral function:

```
q = integral(sqr,0,1)

q =

  0.3333
```

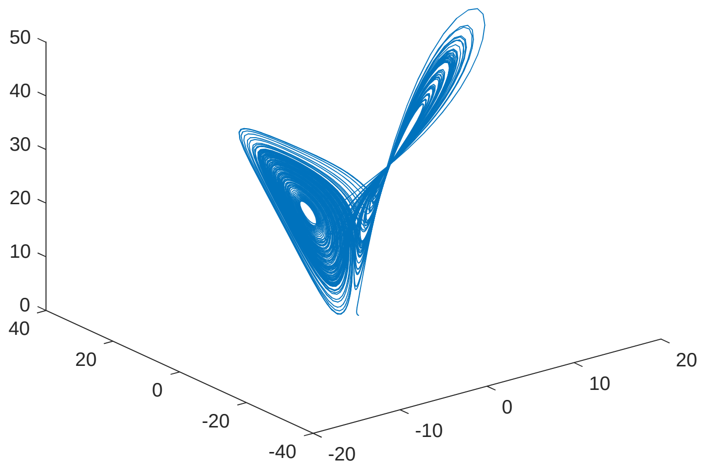
Another example is

```
humps = @(x) 1 ./ ((x-.3).^2 + .01) + 1 ./ ((x-.9).^2 + .04) - 6;
xz2 = fzero(humps, 1)

xz2 =

    1.2995
```

The other example is

```
sigma = 10; beta = 8/3; rho = 28;
f = @(t,x) [-sigma*x(1)+sigma*x(2); rho*x(1)-x(2)-x(1)*x(3); -beta*x(3)+x(1)*x(2)];
[t,x] = ode45(f,[0 100],[1 1 1]); % Runge-Kutta 4th/5th order ODE solver
plot3(x(:,1),x(:,2),x(:,3))
```

This results in



You can use an anonymous function to capture the problem-dependent parameters:

```
myfun = @(x,c) cos(c*x);  % The parameterized function
c = 2;   % The parameter
X = fzero(@(x) myfun(x,c),0.1)

X =

   0.7854
```

You do not need to create a variable in the workspace to store an anonymous function. Instead, you can create a temporary function handle within an expression, such as this call to the integral function:

```
q = integral(@(x) x.^2,0,1);
```

Function handles can store not only an expression, but also variables that the expression requires for evaluation. For example, create a function handle to an anonymous function that requires coefficients a, b, and c.

```
a = 1.3;
b = .2;
c = 30;
parabola = @(x) a*x.^2 + b*x + c;
```

Because a, b, and c are available at the time you create parabola, the function handle includes those values. The values persist within the function handle even if you clear the variables:

```
clear a b c
x = 1;
y = parabola(x)

y =

  31.5000
```

To supply different values for the coefficients, you must create a new function handle:

```
a = -3.9;
b = 52;
c = 0;
parabola = @(x) a*x.^2 + b*x + c;

x = 1;
y = parabola(1)
```

```
    y =

      48.1000
```

You can save function handles and their associated values in a MAT-file and load them in a subsequent MATLAB session using the save and load functions, such as

```
    save myfile.mat parabola
```

9.4 Echo, Input, Pause, Keyboard

Normally, while an M-file is executing, the commands in the file are not displayed on the screen. A command called echo causes M-files to be viewed as they execute, useful for debugging or for demonstrations.

The function input obtains input from the user. n = input (‘How many apples’) gives the user the prompt in the text string, waits, and then returns the number or expression input from the keyboard. One use of input is to build menu-driven M-files. The demo facility is an example of this.

Similar to input, but more powerful, is keyboard. This function invokes your computer keyboard as a script. Placed in M-files, this feature is useful debugging or for modifying variables during execution.

The pause command causes a procedure to stop and wait for the user to press any key before continuing. pause (n) pauses for n seconds before continuing.

It is also possible to define global variables, although we don’t recommend it.

9.5 Strings

Text strings are entered into MATLAB surrounded by single quotes. For example,

```
   s = 'Hello'
```

results in

```
    s =
    Hello
```

The text is stored in a vector, one character per element. In this case,

```
    size(s)

    ans =

         1    5
```

indicates that s has five elements. The characters are stored as their ASCII values and abs shows these values,

```
    abs(s)

    ans =

       72   101   108   108   111
```

The function setstr sets the vector to display as text instead of showing ASCII values. Also useful are disp, which simply displays the text in the variable, and the functions isstr and strcmp, which detect strings and compare strings, respectively.

Text variables can be concatenated into larger strings using brackets:

```
    s = [s, ' World']

    s =

    Hello World
```

Numeric values are converted to strings by sprintf, num2str, and int2str. The converted numeric values are often concatenated into larger strings to put titles on plots that include numeric values:

```
    f = 70; c = (f-32)/1.8;
    title([‘Room temperature is ‘, num2str(c), ’degree C’])
```

9.6 External Programs

It is possible, and often useful, to make your own external standalone programs act like new MATLAB functions. This can be done by writing M-files that

1. Save the variables on disk
2. Run the external programs (which read the data files, process them, and write the results back out to disk), and
3. Load the processed files back into the workspace.

For example, here is a hypothetical M-function that finds the solution to Garfield's equation using an external program called GAREQN

```
function y = garfield(a,b,g,r)
save gardata a b g r
!gareqn
load gardata
```

It requires that you have written a program (in Fortran or some other language) called GAREQN that reads a file called gardata.mat, processes it, and put the results back out to the same file. The utility subroutines described in the next section can be used to read and write MAT-files.

10
Disk Files

load and save are the MATLAB commands to store and retrieve from disk the contents of the MATLAB workspace. Other disk file related commands help manage the disk, allow external programs to be run, and provide data import/export capabilities.

10.1 Disk File Manipulation

The commands dir, type, delete, and chdir implement a set of generic operating system commands for manipulating files. For most of these commands, pathnames, wildcards, and drive designators may be used in the usual way.

The type command differs from the usual type commands in an important way: If no file type is given, .m is used by default. This makes it convenient for the most frequent use of type, to list M-files on the screen.

The diary command creates a diary of your MATLAB session in a disk file (graphics are not saved, however). The resulting ASCII file is suitable for inclusion into reports and other documents using any word processor.

For more details on these commands, use the online help facility.

10.2 Running External Programs

The exclamation point character ! is a shell escape and indicates that the rest of the input line should be issued as a command to the operating system. This is quite useful for invoking utilities or running other programs without quitting from MATLAB. For example

```
    !f77 simpleprog
```

invokes a Fortran compiler called F77 and

```
    !edt darwin.m
```

invokes an editor called edt on a file named darwin.m. After these programs complete, control is returned to MATLAB.

10.3 Importing and Exporting Data

Data from other programs and the outside world can be introduced into MATLAB by several different methods. Similarly, MATLAB data can be exported to the outside world. It is also possible to have your programs manipulate data directly in MAT-files, the file format MATLAB uses.

The best method depends upon how much data there are, whether the data are already in machine readable form, what the form is, etc. Here are some choices; select the one that looks appropriate.

1. Enter it as an explicit list of elements. If you have a small amount of data, say less than 10-15 elements, it is easy to type in the data explicitly using brackets, [and]. This method is awkward for larger amounts of data because you can't edit your input if you make a mistake.
2. Create it in an M-file. Use your text editor to create a script M-file that enters your data as an explicit list of elements. This method is good when your data are not already in computer-readable form and you have to type them in anyway. Essentially the same as method 1, it has the advantage of allowing you to use your editor to change the data or to fix mistakes. You can then just rerun the M-file to reenter the data.
3. Load it from an ASCII flat file. If the data are stored in ASCII form, with fixed length rows terminated with newlines (carriage returns), and with spaces separating the numbers, then the file is a so-called flat file. (ASCII flat files can be edited using a normal text editor.) Flat files can be read directly into MATLAB using the load command. The result is put into a variable whose name is the filename.
4. Write a program in Fortran or C to translate your data into MAT-file format.

Some methods of getting MATLAB data back to the outside world are:

1. For small matrices, use the diary command to create a diary file, and then list the variables on this file. You can use your text editor to manipulate the diary file at a later time. The output of diary includes the MATLAB commands used during the session, which is useful for inclusion into documents and reports.
2. Save a variable using the save command, with the -ascii option. For example,

```
A = rand(4,3);
save temp.dat A -ascii
```

creates an ASCII file called temp.dat that contains:

```
0.9501   0.8913   0.8214
0.2311   0.7621   0.4447
0.6068   0.4565   0.6154
0.4860   0.0185   0.7919
```

3. Write a program in Fortran or C to translate the MAT-file into your own special format.

You may prefer to have your external programs read or write the data directly in MAT-files used by the MATLAB load and save commands.

If you program in Fortran or C, there are some routines provided in the MATLAB Toolbox to help you interface your programs to MAT-files:

savemat. for	A subroutine call that writes MAT-files.
loadmat. for	A subroutine call that reads MAT-files.
testls1. for	An example of using savemat and loadmat.
test1s2.for	Another example of using savemat and loadmat.
loadmat.c	A C routine to load a matrix from a MAT-file.
savemat.c	A C routine to save a matrix to a MAT-file.
test1s.c	An example of using loadmat.c and savemat.c.

The implementation of the Fortan or C versions of these routines may differ from machine to machine.