MATLAB Survival Guide

MATLAB is...

A fancy calculator. You can use MATLAB just like a calculator: Type in a command for some operations to be performed on some data, and MATLAB comes back with the answer. That is to say, MATLAB is an *interpreted* environment. Of course, you get a calculator that does advanced linear algebra, special functions, etc.

A graphics package. MATLAB has a great deal of power for creating 2-D and 3-D graphs. A wide variety of prepackaged graph types are provided. Moreover, everything that is rendered is an *object* within a class hierarchy, and you can modify any of these objects' properties. MATLAB also includes a full set of user interface widgets, so you can create or use graphical interfaces to packages.

A programming language. You can write functions of MATLAB statements just as with any computer language. The MATLAB language has some useful features, such as natural mathematical syntax, strong support of matrices, advanced data types, and automatic dynamic memory allocation. The programs that you write are executable on any platform running MATLAB.

A numerical laboratory. MATLAB is a contraction of "matrix laboratory," which reflects a certain attitude. In MATLAB it's easy to formulate a hypothesis, conduct experiments, examine results, and refine ideas in a feedback loop—in short, to practice *experimental mathematics.* The emphasis is on efficiency of algorithmic development and mathematical understanding.

MATLAB isn't...

A maximum-performance tool. If you must get the very most out of your machine, MATLAB probably isn't the way to go. Finely tuned numerical libraries for low-level programming are likely to be superior. That isn't to say that MATLAB is useless for practical problems. An occasional compromise is to link in C or Fortran routines to a MATLAB function.

A symbolic manipulator. Unlike Maple, Mathematica, and other symbolic mathematics packages, MATLAB doesn't do abstract mathematical transformations. Every variable must always have a concrete value, and all calculations are performed in a fixed precision.

Getting help

The best way to get started is to read the user's guide that comes with MATLAB, if it is available to you. Otherwise, there are many books that have MATLAB instruction as a primary or secondary goal. Visit `http://www.mathworks.com` for a current list.

Much help is available online. The most extensive is through the command `helpdesk`, which points a web browser to the top of a large, searchable documentation tree. For specific help on using built-in features and functions, you can use `help` or the more convenient `helpwin`. Most useful for beginners are

```
>> helpwin general
>> helpwin lang
>> helpwin ops
```

If you need to find a command related to some keyword, `lookfor` may help. For instance,

```
>> lookfor memory
CLEAR  Clear variables and functions from memory.
INMEM List functions in memory.
MEMORY Help for memory limitations.
PACK   Consolidate workspace memory.
MOVIEIN Initialize movie frame memory.
```

Running the built-in `demo` will give you ideas about what's available.

You will start by knowing how to do just a few things. As you get more comfortable, you will seek out shortcuts or additional capabilities that you "know" must be there. The best way to learn MATLAB is to dig in and start using it!

Simple use

If you type in a valid expression and press Enter, MATLAB will immediately execute it and return the result.

```
>> 2+2
ans =
     4

>> sin(pi/2)
ans =
     1

>> 1/0
Warning: Divide by zero.
ans =
   Inf

>> exp(i*pi)
ans =
  -1.0000 + 0.0000i
```

The `>>` symbol is the *prompt*, and when you see it, MATLAB is waiting for you to enter something. Notice that expressions involving the imaginary unit `i` (or `j`) return complex results.

You can assign values to variables.

```
>> x = sqrt(3)
x =
    1.7321
```

```
>> atan(x)
ans =
    1.0472

>> pi/ans
ans =
     3
```

When an expression returns a single result that is not assigned to a variable, this result is assigned to the variable `ans`, which can then be used like any other variable.

Here are a few other demonstration statements.

```
>> % This is a comment.
>> x = rand(100,100);  % don't print out x
>> s = 'Hello world';  % a string variable
>> t = 1 + 2 + 3 + ...
4 + 5 + 6              % continuation of a line

t =
    21
```

Matrices

Matrices are the heart and soul of MATLAB. Fundamentally, MATLAB sees (almost) everything as a matrix, with vectors, scalars, and even strings as special cases.

Constructing matrices

The simplest way to construct a matrix is by enclosing its entries in square brackets.

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

>> b = [0;1;0]
b =
     0
     1
     0
```

Separate columns by spaces (or commas) and rows by semicolons (or new lines). A matrix which has just one column is a *column vector*. You can also create a *row vector*. Sometimes, the two types of vectors are interchangeable, but often they are not, so take care.

Brackets can be nested, so long as the result makes sense.

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

You may occasionally need the *empty matrix*, which is input as `[]`. Bracket constructions are suitable only for very small matrices. For larger ones, there are many useful functions, as shown in Table 1.

Table 1: Commands for building matrices.

`eye`	identity matrix
`zeros`	all zeros
`ones`	all ones
`diag`	diagonal matrix
`triu`	upper triangle
`tril`	lower triangle
`rand`, `randn`	random entries
`linspace`	evenly spaced entries

An especially important construct is the *colon operator*.

```
>> 1:8
ans =
     1     2     3     4     5     6     7     8

>> 0:2:10
ans =
     0     2     4     6     8    10

>> 1:-.5:-1
ans =
    1.0000    0.5000         0   -0.5000   -1.0000
```

An *array* is subtly different than a matrix. Both are collections of related numbers, but a matrix is given a special mathematical interpretation. Arrays in MATLAB can have any number of dimensions, while matrices have two. Formally, there is no way to distinguish a matrix from a 2-D array in MATLAB. In fact, the term "matrix" is often used in either case. Confusing the two types in practice, however, is liable to lead to disaster.

Accessing elements

Table 2 lists several useful commands for obtaining information about a matrix or array.

It is frequently necessary to access one or some of the elements of a matrix. Using the definitions above, we have

```
>> A(2,3)
ans =
```

Table 2: Matrix information commands.

`size`	size in each dimension
`length`	size of longest dimension (esp. for vectors)
`ndims`	number of dimensions
`find`	indices of nonzero elements

```
     6

>> b(2)        % b is a vector
ans =
     1

>> b([1 3])    % multiple elements
ans =
     0
     0

>> A(1:2,2:3)  % a submatrix
ans =
     2     3
     5     6

>> B(1,2:end)
ans =
     2     5

>> B(:,3)
ans =
     5
     6
```

Each *subscript* can be a scalar or a vector (possibly constructed using the colon operator), and the result is a submatrix. The first element in each dimension is always indexed as `1`. The special keyword `end` stands for the last possible element in a dimension, and the colon by itself means "everything in that dimension."

Vectors can be given a single subscript. In fact, any array can be accessed via a single subscript. Multi-dimensional arrays are actually stored linearly in memory, varying over the first dimension, then the second, and so on. (Think of the columns of a matrix being stacked on top of each other.) Thus the array is equivalent to a vector. A single subscript will be interpreted as an index into this vector. Thus `A(6)` is our example would return `8`. (See `sub2ind` and `ind2sub` for more details.) A matrix `A` can be explicitly converted into column vector form by `A(:)`.

Subscript referencing can be on either side of assignments.

```
>> B(1,:) = -A(1,:)
B =
```

```
    -1    -2    -3
     3     4     6
>> C = rand(2,5)
C =
    0.8125    0.4054    0.4909    0.5909    0.5943
    0.2176    0.5699    0.1294    0.8985    0.3020
>> C(:,4) = []   % delete elements
C =
    0.8125    0.4054    0.4909    0.5943
    0.2176    0.5699    0.1294    0.3020
>> C(2,:) = 0    % expand the scalar into the submatrix
C =
    0.8125    0.4054    0.4909    0.5943
         0         0         0         0
>> C(3,1) = 3    % create a new row to make space
C =
    0.8125    0.4054    0.4909    0.5943
         0         0         0         0
    3.0000         0         0         0
```

Observe that a matrix can be resized automatically if you delete elements or make assignments outside the current size. This can be highly convenient, but it can also cause subtle mistakes.

A different kind of indexing is *logical indexing.* Logical indices usually arise from a *relational operator* (see Table 3). The result of applying a relational operator is a logical matrix, which is essentially a matrix of

Table 3: Relational operators.

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

zeros and ones. Using a logical matrix as an index to another matrix returns those values where the logical matrix has a one. Here we are using the single-subscript interpretation of the matrix.

```
>> B>0
ans =
     0     0     0
     1     1     1

>> B(ans)
ans =
     3
     4
     6

>> b(b==0)
```

```
ans =
     0
     0

>> A(A~=round(A))
ans =
     []

>> b([1 1 1])   % first element, three copies
ans =
     0
     0
     0

>> b(logical([1 1 1]))   % every element
ans =
     0
     1
     0
```

Notice that when you use a single subscript on a matrix, you lose the shape information about that matrix, because the result is a submatrix of a column vector.

Matrix operations

The arithmetic operators `+, -, *, ^` are interpreted in a matrix sense. When appropriate, scalars can be "expanded" to match a matrix.

```
>> A+A
ans =
     2     4     6
     8    10    12
    14    16    18

>> ans-1
ans =
     1     3     5
     7     9    11
    13    15    17

>> 3*B
ans =
    -3    -6    -9
     9    12    18

>> A*b
ans =
```

```
     2
     5
     8

>> B*A
ans =
   -30   -36   -42
    61    74    87

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

>> A^2
ans =
    30    36    42
    66    81    96
   102   126   150
```

The apostrophe ′ produces the conjugate transpose of a matrix.

```
>> A*B'-(B*A')'
ans =
     0     0
     0     0
     0     0

>> b'*b
ans =
     1

>> b*b'
ans =
     0     0     0
     0     1     0
     0     0     0
```

A special operator, \ (backslash), is used to solve linear systems of equations. A forward slash has a similar interpretation.

```
>> C = [1 3 -1;2 4 0;6 0 1];
>> x = C\b
x =
   -0.1364
    0.3182
    0.8182
```

```
>> C*x - b
ans =
   1.0e-15 *
    0.1110
         0
         0
```

This example brings up an important point about floating-point arithmetic. Since most real numbers cannot be represented exactly, you should not expect "equal" values to have a difference of exactly zero. The built-in number `eps` tells you the smallest significant number relative to unity on your particular machine.

```
>> norm(ans)
ans =
   1.1102e-16

>> eps
ans =
   2.2204e-16
```

You should *never* do a comparison like `a==b` to detect equality of floating-point numbers; instead use something like `abs(a-b) < eps*max(a,b)`.

Some familiar functions from linear algebra are listed in Table 4; there are many others.

Table 4: Functions from linear algebra.

`rank`	rank
`det`	determinant
`norm`	norm (2-norm, by default)
`expm`	matrix exponential
`eig`	eigenvalue decomposition
`svd`	singular value decomposition
`chol`	Cholesky factorization
`lu`	LU decomposition

Array operations

Conceptually, array operations simply act identically on each element of an array(s). We have already seen some array operations, namely `+` and `-`. But `*,',^,/` have particular matrix interpretations. To get a componentwise behavior, precede the operator with a dot.

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

```
>> C
C =
     1     3    -1
     2     4     0
     6     0     1

>> A.*C
ans =
     1     6    -3
     8    20     0
    42     0     9

>> b./b

Warning: Divide by zero.
ans =
   NaN
     1
   NaN

>> (B+i)'
ans =
  -1.0000 - 1.0000i   3.0000 - 1.0000i
  -2.0000 - 1.0000i   4.0000 - 1.0000i
  -3.0000 - 1.0000i   6.0000 - 1.0000i

>> (B+i).'
ans =
  -1.0000 + 1.0000i   3.0000 + 1.0000i
  -2.0000 + 1.0000i   4.0000 + 1.0000i
  -3.0000 + 1.0000i   6.0000 + 1.0000i
```

Observe that `0/0` is undefined and returns `NaN`, which stands for *Not A Number*. There is no difference between `'` and `.'` for real arrays.

Most elementary functions, such as `sin`, `exp`, etc., act componentwise.

```
 >> B
B =
    -1    -2    -3
     3     4     6

>> abs(B)
ans =
     1     2     3
     3     4     6
```

```
>> cos(pi*B)
ans =
    -1     1    -1
    -1     1     1

>> exp(A)
ans =
   1.0e+03 *
    0.0027    0.0074    0.0201
    0.0546    0.1484    0.4034
    1.0966    2.9810    8.1031

>> expm(A)
ans =
   1.0e+06 *
    1.1189    1.3748    1.6307
    2.5339    3.1134    3.6929
    3.9489    4.8520    5.7552
```

Remember that `exp` and `sqrt` are quite different from `expm` and `sqrtm`. Be sure you use the right one!

Other array operations work in parallel along one dimension of the array, returning a result that is one dimension smaller.

```
>> C
C =
     1     3    -1
     2     4     0
     6     0     1
>> sum(C)
ans =
     9     7     0
>> sum(C,1)
ans =
     9     7     0
>> sum(C,2)
ans =
     3
     6
     7
```

Other functions that behave this way include

```
max   sum    mean   any
min   diff   median  all
sort  prod    std
```

Flow control

Flow control statements include branching and looping structures. These work much as in other languages.

if

Here is an example illustrating most of the features of `if`.

```
if isinf(x) | ~isreal(x)
  disp('Bad input!')
  y = NaN;
elseif (x == round(x)) & (x > 0)
  y = prod(1:x-1)
else
  y = gamma(x)
end
```

Conditions for `if` statements may involve the relational operators of Table 3 or functions such as `isinf` that return logical values. Numerical values can also be used, with nonzero meaning true, but `if x~=0` is better practice than `if x`. Individual conditions can be combined using

& (logical AND) | (logical OR) ~ (logical NOT)

Compound conditions can be "short-circuited." If, while evaluating the condition from left to right, it can be concluded at some point that the whole expression must be false, evaluation of the condition is halted. This makes it convenient to write things like

```
if (length(x) >= 3) & (x(3)==1)
```

that are otherwise awkward.

switch

`switch` statements technically provide no new functionality, but they are an alternative to `if` structures with many `elseif` clauses. For example:

```
switch answer(1)
case {'y','Y'}
  disp('OK, formatting hard drive.')
  % Put format command here
case {'n','N'}
  disp('Chicken!')
case 'Q'
  return
otherwise
  disp('Invalid entry, you get one more try.')
end
```

The `switch` line is given a scalar or string to work with. This value is compared with the options at each `case`, executing the block if a match is found. The optional `otherwise` block if no match is found. At most one block is executed, unlike the situation in C.

for

This illustrates the most common type of `for` loop:

```
f = [1 1];
for n = 3:100
  f(n) = f(n-1) + f(n-2);
end
disp(f(100))
```

Of course, you can have as many statements as you like in the body of the loop. The value of the index `n` will change from 3 to 100, with an execution of the body after each assignment.

Remember that `3:100` is really just a row vector. You can use *any* row vector in a `for` loop, not just one created by a colon. One handy use of this is with `find`, as in

```
for j = find(~isnan(x(:))')
  % Operate on x(j), which will never be NaN
end
```

But be sure it's a *row* vector.

Don't use `for` loops more than you really have to! In Fortran, you need a loop to do anything with an array. But MATLAB provides you with most of the array and matrix operations you ever need. Not only is it more aesthetic to write `A*B` than a triply nested loop, but `for` loops can slow down MATLAB tremendously.

A final warning: If you are using complex numbers, avoid using `i` as the loop index. At startup, `i` is the imaginary unit, but you can freely reassign it.

while

```
while (b-a) > tol
  m = mean([a b]);
  if f(a)*f(m) < 0
     b = m;
  else
     a = m;
  end
end
zero = mean([a b]);
```

The condition is checked before each execution of the body, so it is possible to never execute the body of the loop.

break

You can exit a loop prematurely. If `break` is encountered inside a `for` or `while` loop, execution is rerouted to the next statement beyond the `end`. Only the innermost loop is exited.

Sometimes it's just easiest to write a loop with a `break`, even though you could technically do without. But use it sparingly.

M–files: Functions and scripts

An *M–file* is a regular text file containing MATLAB commands, saved with the filename extension `.m`. There are two types, *scripts* and *functions.* The only syntactic difference is that functions begin with a function header, whereas scripts don't. But there are important practical differences:

Script	Function
no header	`function [a,b,...] = foo(x,y,...)`
base workspace	local workspace
interpreted	compiled (faster)
one per file	many in one file
no analysis tools	debugging & profiling available

Use a function whenever you want a module that performs a task multiple times on many different inputs. Scripts are often used as "drivers" that describe only the top level of a task.

On PC and Macintosh platforms, MATLAB has its own small but capable text editor for creating and editing M–files. You can start it graphically from the command window, or by the command `edit`. For Unix users, there is a good `matlab-mode.el` for Emacs maintained by Matt Wette.

Once an M–file has been saved to disk, it is invoked by entering the file's root name at the prompt. For example, if you saved a script as `mydriver.m`, then entering `mydriver` would cause the contents to be executed. There is a catch, however—the file must be in a directory (folder) on MATLAB's *path.* The path is simply a list of directories in which MATLAB looks whenever an unfamiliar command name is encountered. You can modify it using `path` or `editpath`.

Scripts

A script is simply a collection of valid MATLAB commands. When the script is invoked by typing its name at the prompt, the commands are read from the file one at a time and executed just as if they had been typed.

The usefulness of scripts is in gathering a set of commands that you might use multiple times, perhaps with small changes. They also let you recreate your work at a later date. Some people use MATLAB with an editor window open full-time; they enter all their commands into a script and periodically execute the script.

Functions

The biggest conceptual difference between a script and a function is in the *variable workspace,* the list of variables of which the function is aware. Commands entered at the prompt and in scripts all use the *base workspace,* which you can view using `who`. A function executes within a *local workspace* which is created just for it and which vanishes when the function terminates. The only variables a function knows about at its beginning are those given as input arguments, and the only variables that can pass out are its output arguments. These are named in the header, which must be the first line of the M–file:

```
function [out1,out2,...] = func(in1,in2,...)
```

Outputs are on the left, inputs on the right. They can have any names and you can have as many as you want. Except for the keyword `function`, this is just how you will invoke the function from the prompt, a script, or another function. (Of course, you will have to provide valid values for the input arguments.)

Here is a recursive sorting function.

```
function list = mergesort(list)
if length(list)==1
  return   % do nothing and quit
else
  n = floor(length(list)/2);
  l1 = mergesort(list(1:n));
  l2 = mergesort(list(n+1:end));
  list = merge(l1,l2);
end
```

You will have to supply the function `merge` yourself!

MATLAB has some very useful tools for debugging functions and tracking their efficiency. Try the help for `debug` and `profile`.

Graphics

The variety and power of graphics available in MATLAB is tremendous—as of this writing, there is a separate manual for graphics that ships with the software. Here we only attempt to show the most basic capabilities for 2-D graphics.

```
>> t = (0:.05:3)';
>> plot(t,sin(pi*t))       % basic plot
>> plot(t,t.^2,'r--')      % use a dashed red line
>> plot(t,bessel(0:3,t))   % one curve for each column
>> hold on                 % add to current plot
>> plot(t,bessely(0,t),'ko')
>> figure                  % new figure
>> semilogy(t,besseli(0:3,t),'s-')   % log scale in y
>> loglog(t,4*t.^2+1./t.^2)          % both log scales
```

If you want multiple "windows" within a figure, try `subplot`. To set the limits of the axes manually, use `axis`. To label title and axes, use `title`, `xlabel`, and `ylabel`. Also, `text` and `gtext` allow you to annotate with text anywhere. There is a `legend` command for graphs with multiple curves.

Every single object that is rendered has intrinsic properties that can be modified. You can do this by point-and-click if you start `guide`. See the manuals for what is going on.

Keep in mind that your screen is probably color and your printer probably isn't. To distinguish curves, create them with different linestyles and markers. See the help for `plot` and the examples above.

The `print` command can be used to print figures as hardcopy or as Postscript files. You can also print a figure using its window menu.

To get started on 3-D graphs, try `surf`, `mesh`, and `contour`.

Input and output

When MATLAB prints numerical values, it uses a certain format that you control with `format`. For instance,

```
>> format short   % the default
>> pi
ans =
    3.1416

>> format long
>> exp(1)
ans =
   2.71828182845905

>> format short e
>> exp(20+i)
ans =
  2.6214e+08 + 4.0825e+08i
```

For no-fuss display of matrices and strings, use `disp`. For trickier output use `sprintf` or `fprintf`, which work much like their C counterparts.

```
>> fprintf('%3i %12.4g\n',[1:6; exp((1:6).^2)])
  1        2.718
  2         54.6
  3         8103
  4    8.886e+06
  5      7.2e+10
  6    4.311e+15
```

To prompt the user for simple input, use `input`.

You can save matrices for later use by saying

```
>> save myfile A B C
```

The variables named are saved in a file `myfile.mat`. (use whatever name you like.) If you don't name any variables, the entire base workspace is saved. You can load the variables back in later via `load myfile`. You can also use `load` to load in simple text files of numbers. More sophisticated file input and output can be done with the C-like `fprintf` and `fscanf`.

To save the input and output in the command window during a session, use

```
>> diary log.txt
>> % Here go some commands and results.
>> diary off
```

Everything between the `diary` commands is saved in `log.txt`, whether typed by you or output by MATLAB.

As mentioned above, you can obtain hardcopy of your figures with `print`. You can also save them to a file with `print`:

```
>> print -dmfile myfig
```

This creates `myfig.m` (and possibly `myfig.mat`). By entering `myfig` as a command, you get a new figure that is a replica of the saved one.