MATLAB for the impatient

August 23, 2004

The online documentation for MATLAB includes an excellent but rather long *Getting Started* guide. There are also plenty of excellent books about MATLAB that you can buy or borrow. But the best way to learn MATLAB, at least at the beginning, is by learning the essentials and diving in. Here I'm trying to help with the first half of that formula.

This document was written about version 6.5 of MATLAB. Version 7.0 changes a few things, particularly about inline functions, but nothing too serious.

1 Introduction

MATLAB is a software package for computation in engineering, science, and applied mathematics. It offers a powerful programming language, excellent graphics, and a wide range of expert knowledge. MATLAB is published by and a trademark of The MathWorks, Inc.

The focus in MATLAB is on computation, not mathematics. Hence symbolic expressions and manipulations are not possible (except through a clever interface to Maple). All results are not only numerical but inexact, thanks to the rounding errors inherent in computer arithmetic. The limitation to numerical computation can be seen as a drawback, but it is a source of strength too: MATLAB is much preferred to Maple, Mathematica, and the like when it comes to numerics.

On the other hand, compared to other numerically oriented languages like C++ and FORTRAN, MATLAB is much easier to use and comes with a huge standard library. The unfavorable comparison here is a gap in execution speed. This gap is not always dramatic, and it can often be narrowed or closed with good MATLAB programming, but MATLAB is not the usual tool for high-performance computing.

The MATLAB niche is numerical computation on workstations for nonexperts in computation. This is a huge niche—one way to tell is to look at the number of MATLAB-related books on mathworks.com. Even for hard-core supercomputer users, MATLAB can be a valuable environment in which to explore and fine-tune algorithms before more laborious coding in C.

*Department of Mathematical Sciences, Ewing Hall, University of Delaware, Newark, DE 19716; driscoll@math.udel.edu.

1.1 The fifty-cent tour

When you start MATLAB, you get a multipaneled **desktop** and perhaps a few other new windows as well. The layout and behavior of the desktop and its components are highly customizable.

The component that is the heart of MATLAB is called the Command Window. Here you can give MATLAB commands typed at the prompt, >>. Unlike FORTRAN and other compiled computer languages, MATLAB is an **interpreted** environment—you give a command, and MATLAB tries to execute it right away before asking for another.

In the default desktop you can also see the Launch Pad. The Launch Pad is a window into the impressive breadth of MATLAB. Individual **toolboxes** add capability in specific methods or specialties. Often these represent a great deal of expert knowledge. Most have friendly demonstrations that hint at their capabilities, and it's easy and interesting to spend a day on these. You may notice that many toolboxes are related to electrical engineering, which is a large share of MATLAB's clientele. Another major item, not exactly a toolbox, is Simulink, which is a control-oriented interface to MATLAB's dynamic simulation facilities.

Notice at the top of the desktop that MATLAB has a notion of **current directory**. In general MATLAB can only "see" files in the current directory (folder) and on its **path**, which can be customized. Commands for working with the directory and path include `cd`, `what`, `addpath`, and `pathedit` (in addition to widgets and menu items). You can add files to a folder on the path and thereby add commands to MATLAB; we will return to this subject in section 3.

You can also see a tab for the **workspace** next to the Launch Pad. The workspace shows you what variables are currently defined and some information about their contents. (At startup it is, naturally, empty.) This represents another break from compiled environments: variables created in the workspace persist for you to examine and modify; output does not need to be immediately dumped into a file.

In this document I will often give the names of commands that can be used at the prompt. In many—maybe most—cases these have equivalents among the menus, buttons, and other graphical widgets. Take some time to explore these widgets. This is one way to become familiar with the possibilities in MATLAB.

1.2 Help

MATLAB is huge. You can't learn everything about it, or even always remember things you have done before. It is essential that you become familiar with the online help.

There are two levels of help:

- If you need quick help on the syntax of a command, use `help`. For example, `help plot` tells you all the ways in which you can use the `plot` command. Typing `help` by itself gives you a list of categories that themselves yield lists of commands.
- Use `helpdesk` or the menu/graphical equivalent to get into the Help Browser. This includes HTML and PDF forms of all MATLAB manuals and guides, including toolbox manuals. The *MATLAB: Getting Started* and the *MATLAB: Using MATLAB* manuals are excellent places to start. The *MATLAB Function Reference* is an essential resource.

1.3 Basic commands and syntax

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

```
>> 2+2

ans =
     4

>> 4^2

ans =
    16

>> sin(pi/2)

ans =
     1

>> 1/0
Warning: Divide by zero.

ans =
   Inf

>> exp(i*pi)

ans =
  -1.0000 + 0.0000i
```

Notice some of the special expressions here: `pi` for π, `Inf` for ∞, and `i` for $\sqrt{-1}$.[1] Another special value is `NaN`, which stands for **not a number**. `NaN` is used to express an undefined value. For example,

```
>> Inf/Inf

ans =
   NaN
```

You can assign values to variables with alphanumeric names.

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

[1] `j` is also understood for $\sqrt{-1}$. Both names can be reassigned, however. It's often safer to use `1i` or `1j` to refer to the imaginary unit.

```
x =
    1.7321

>> days_since_birth = floor(now) - datenum(1969,05,06)

days_since_birth =
       12810

>> 3*z
??? Undefined function or variable 'z'.
```

Observe that *variables must have values before they can be used*. When an expression returns a single result that is not assigned to a variable, this result is assigned to `ans`, which can then be used like any other variable.

```
>> atan(x)

ans =
    1.0472

>> pi/ans

ans =
     3
```

In floating-point arithmetic, you should not expect "equal" values to have a difference of exactly zero. The built-in number `eps` tells you the maximum error in arithmetic on your particular machine.[2] For simple operations, the relative error should be less than this number. For instance,

```
>> eps

ans =
   2.2204e-16

>> exp(log(10)) - 10

ans =
   1.7764e-15

>> ans/10

ans =
   1.7764e-16
```

Here are a few other demonstration statements.

[2] Like other names, `eps` can be reassigned, but doing so has no effect on the roundoff precision.

```
>> % This is a comment.
>> x = rand(100,100);  % ; means "don't print out result"
>> s = 'Hello world';  % single quotes enclose a string
>> t = 1 + 2 + 3 + ...
4 + 5 + 6              % ... continues a line

t =
    21
```

Once variables have been defined, they exist in the **workspace**. You can see what's in the workspace from the desktop or by typing

```
>> who

Your variables are:

ans  s    t    x
```

1.4 Saving work

If you enter `save myfile` , all the variables in the workspace will be saved to a file called `myfile.mat` in the current directory. Later you can use `load myfile` to recover the variables.

If you right-click in the Command History window and select "Create M-File…", you can save all your typed commands to a text file. This can be very helpful for recreating what you have done. Also see section 3.1.

2 Arrays and matrices

MATLAB encourages and expects you to make heavy use of arrays, vectors, and matrices.

Some jargon: An **array** is a collection of numbers, called **elements** or **entries**, referenced by one or more indices running over different index sets. In MATLAB, the index sets are always sequential integers starting with 1. The **dimension** of the array is the number of indices needed to specify an element. The **size** of an array is a list of the sizes of the index sets.

A **matrix** is a two-dimensional array with special rules for addition, multiplication, and other operations. It represents a mathematical linear transformation. The two dimensions are called the **rows** and the **columns**. A **vector** is a matrix for which one dimension has only the index 1. A **row vector** has only one row and a **column vector** has only one column.

Although an array is much more general and less mathematical than a matrix, the terms are often used interchangeably. Indeed, MATLAB makes no formal distinction—not even between a scalar and a 1×1 matrix. The commands below are sorted according to the array/matrix distinction, but MATLAB will let you mix them freely. The idea (here as elsewhere) is that MATLAB keeps the language simple and natural. It's up to you to stay out of trouble.

2.1 Building arrays and matrices

The simplest way to construct a small array is by enclosing its elements 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. Information about size and dimension is stored with the array.[3]

```
>> size(A)

ans =
     3     3

>> ndims(A)

ans =
     2

>> size(b)

ans =
     3     1

>> ndims(b)

ans =
     2
```

Notice that there is really no such thing as a one-dimensional array in MATLAB. Even vectors are technically two-dimensional, with a trivial dimension. Table 1 lists more commands for obtaining information about an array.

Arrays can be built out of other arrays, as long as the sizes are compatible.

[3] Because of this, array sizes are not usually passed explicitly to functions as they are in FORTRAN.

Table 1: 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

```
>> [A b]

ans =
     1     2     3     0
     4     5     6     1
     7     8     9     0

>> [A;b]
??? Error using ==> vertcat
All rows in the bracketed expression must have the same
number of columns.

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

B =
     1     2     5
     3     4     6
```

One special array is the **empty matrix**, which is entered as `[]` .

An alternative to the bracket notation is the `cat` function. This is one way to construct arrays of more than two dimensions.

```
>> cat(3,A,A)

ans(:,:,1) =

     1     2     3
     4     5     6
     7     8     9


ans(:,:,2) =

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

Bracket constructions are suitable only for very small matrices. Large matrices are built more automatically. In many cases a matrix of interest has some structure or property that streamlines

the process; some useful functions are shown in Table 2. The array referencing and operations in the following sections are also handy.

Table 2: Commands for building matrices.

`eye`	identity matrix
`zeros`	all zeros
`ones`	all ones
`diag`	diagonal matrix (or, extract a diagonal)
`toeplitz`	constant on each diagonal
`triu`	upper triangle
`tril`	lower triangle
`rand , randn`	random entries
`linspace`	evenly spaced entries
`repmat`	duplicate vector across rows or columns

An especially important constructor 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
```

The format is `first:step:last` . The result is always a row vector, or the empty matrix if `last` $<$ `first`.

The typical C and FORTRAN array construction style of nested loops, perhaps with conditional statements (like `if i==j` for the diagonal), should eventually become your *last* resort in MATLAB.

2.2 Referencing elements

It is frequently necessary to access one or more of the elements of a matrix. Each dimension is given a single index or vector of indices. The result is a **block** extracted from the matrix. Some

examples using the definitions above:

```
>> A(2,3)

ans =
     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)  % special keyword

ans =
     2     5

>> B(:,3)      % "include all" syntax

ans =
     5
     6

>> b(:,[1 1 1 1])

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

The colon is often a useful way to construct these indices. There are some special syntaxes: `end` means the largest index in a dimension, and `:` is short for `1:end` —i.e. everything in that dimension. Note too from the last example that the result need not be a subset of the original array.

Vectors can be given a single subscript. In fact, *any* array can be accessed via a single subscript. Multidimensional arrays are actually stored linearly in memory, varying over the first dimension,

then the second, and so on. (Think of the columns of a table being stacked on top of each other.) In this sense the array is equivalent to a vector, and a single subscript will be interpreted in this context. (See `sub2ind` and `ind2sub` for more details.)

```
>> A

A =
     1     2     3
     4     5     6
     7     8     9

>> A(2)

ans =
     4

>> A(7)

ans =
     3

>> A([1 2 3 4])

ans =
     1     4     7     2

>> A([1;2;3;4])

ans =
     1
     4
     7
     2

>> A(:)

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

The output of this type of index is in the same shape as the index. The potentially ambiguous `A(:)` is always a column vector.

Subscript referencing can be used 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
```

An array is resized automatically if you delete elements or make assignments outside the current size. (Any new undefined elements are made zero.) This can be highly convenient, but it can also cause hard-to-find 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 array**, whose elements are 0 and 1 with interpretation as "false" and "true."[4] Using a logical array as an index returns those values where the index is 1 (in the single-index sense above).

```
>> B>3
```

[4] In a recent version of MATLAB the commands `false` and `true` were introduced for creating logical arrays.

```
ans =
     0     0     0
     0     1     1

>> B(ans)

ans =
     4
     6

>> b(b==0)

ans =
     0
     0

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

ans =
     0
     0
     0

>> b(logical([1 1 1]))   % every element

ans =
     0
     1
     0
```

It's worth noting that the `find` command, which locates nonzeros, effectively converts a logical index vector to a standard one. There is no one-step command for converting in the other direction.

Table 3: Relational operators.

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

2.3 Matrix operations

The arithmetic operators + , - ,*,^ are interpreted in a matrix sense. When appropriate, scalars are "expanded" to match a matrix.[5]

```
>> 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
```

[5]This gives scalar addition more of an array rather than a matrix interpretation.

```
    66    81    96
   102   126   150
```

The apostrophe ' produces the complex-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.

```
>> 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-16 *

    0.5551
         0
         0
```

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

Table 4: Functions from linear algebra.

`\`	solve linear system (or least squares)
`rank`	rank
`det`	determinant
`norm`	norm (2-norm, by default)
`expm`	matrix exponential
`lu`	LU factorization (Gaussian elimination)
`qr`	QR factorization
`chol`	Cholesky factorization
`eig`	eigenvalue decomposition
`svd`	singular value decomposition

2.4 Array operations

Array operations simply act identically on each element of an array. We have already seen some array operations, namely `+` and `-`, which are defined for arrays the same as for matrices. But the operators `*`, `'`, `^`, and `/` have different matrix interpretations. To get elementwise behavior appropriate for an array, 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

>> A*C

ans =
```

```
    23    11     2
    50    32     2
    77    53     2

>> A./A

ans =
     1     1     1
     1     1     1
     1     1     1

>> (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
```

There is no difference between `'` and `.'` for real-valued arrays. Most elementary functions, such as `sin`, `exp`, etc., act elementwise.

```
>> B

B =
     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
```

It's easy to forget that `exp(A)` is an array function. Use `expm(A)` to get the matrix exponential $I + A + A^2/2 + A^3/6 + \cdots$.

Elementwise operators are often useful in functional expressions. Consider evaluating a Taylor approximation to $\sin(t)$:

```
>> t = (0:0.25:1)*pi/2

t =
         0    0.3927    0.7854    1.1781    1.5708

>> t - t.^3/6 + t.^5/120

ans =
         0    0.3827    0.7071    0.9245    1.0045
```

This is easier and better than writing a loop for the calculation.

Another kind of array operation works 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,1)

ans =
     9     7     0

>> sum(C,2)

ans =
     3
     6
     7
```

Other functions that behave this way include

Table 5: "Parallel" functions.

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

2.5 Sparse matrices

It's natural to think of a matrix as a complete rectangular table of numbers. However, many real-world matrices are both extremely large and very **sparse**, meaning that most entries are zero.[6] In such cases it's wasteful or impossible to store every entry. Instead one could for example keep a list of nonzero entries and their locations. MATLAB has a `sparse` data type for this purpose. The `sparse` and `full` commands convert back and forth.

```
>> A = vander(1:3);
>> sparse(A)

ans =
   (1,1)        1
   (2,1)        4
   (3,1)        9
   (1,2)        1
   (2,2)        2
   (3,2)        3
   (1,3)        1
   (2,3)        1
   (3,3)        1

>> full(ans)
ans =

     1     1     1
     4     2     1
     9     3     1
```

You can dissect a sparse matrix using `find`, which returns the indices and values of the nonzeros.

"Sparsifying" a standard full matrix is usually *not* the right way to create a sparse matrix—the point is to avoid creating very large full matrices, even temporarily. One alternative is to give `sparse` the raw data required by the format.

```
>> sparse(1:4,8:-2:2,[2 3 5 7])
```

[6] This is often the result of a "nearest neighbor interaction" that the matrix is modeling. For instance, the PageRank algorithm used by Google starts with an adjacency matrix in which a_{ij} is nonzero if page j links to page i. Obviously any page links to a tiny fraction of the more than 3 billion and counting public pages!

```
ans =
   (4,2)        7
   (3,4)        5
   (2,6)        3
   (1,8)        2
```

Alternatively, you can create an empty sparse matrix with space to hold a specified number of nonzeros, and then fill in using standard subscript assignments. Another useful sparse building command is `spdiags`, which builds along the diagonals of the matrix.

```
>> M = ones(6,1)*[-20 Inf 10]

M =
   -20   Inf    10
   -20   Inf    10
   -20   Inf    10
   -20   Inf    10
   -20   Inf    10
   -20   Inf    10

>> full( spdiags( M,[-2 0 1],6,6 ) )

ans =
   Inf    10     0     0     0     0
     0   Inf    10     0     0     0
   -20     0   Inf    10     0     0
     0   -20     0   Inf    10     0
     0     0   -20     0   Inf    10
     0     0     0   -20     0   Inf
```

The `nnz` command tells how many nonzeros are in a given sparse matrix. Since it's impractical to view directly all the entries (even just the nonzeros) of a realistically sized sparse matrix, the `spy` command helps by producing a plot in which the locations of nonzeros are shown. For instance, `spy(bucky)` shows the pattern of bonds among the 60 carbon atoms in a buckyball.

MATLAB has a lot of ability to work intelligently with sparse matrices. The arithmetic operators `+`, `-`, `*`, and `^` use sparse-aware algorithms and produce sparse results when applied to sparse inputs. The backslash `\` uses sparse-appropriate linear system algorithms automatically as well. There are also functions for the iterative solution of linear equations, eigenvalues, and singular values.

3 Scripts and functions

An **M-file** is a regular text file containing MATLAB commands and saved with the filename extension `.m`. There are two types, **scripts** and **functions**. MATLAB comes with a pretty good editor that is tightly integrated into the environment. Start it using `open` or `edit`. However, you are free to use any text editor. An M-file should be saved in the **path** in order to be executed. The path is just a list of directories (folders) in which MATLAB will look for files. Use `editpath` or menus to see and change the path.

There is no need to compile either type of M-file. Simply type in the name of the file (without the extension) in order to run it. Changes that are saved to disk will be included in the next call to the function or script.

One important type of statement in an M-file is a **comment**, which is indicated by a percent sign `%`. Any text on the same line after a percent sign is ignored (unless `%` appears as part of a string in quotes).

3.1 Using scripts effectively

A script is mostly useful as a "driver" for a multistep task. The commands in a script are literally interpreted as though they were typed at the prompt. Scripts are useful for creating and revising a long, complex sequence of commands and for reproducing or continuing your work at a later time. As a rule of thumb, scripts should be at the top level—they can call functions, but functions should not call them.

3.2 Functions

Functions are the main way to extend the capabilities of MATLAB. Each function must start with a line such as

```
function [out1,out2] = myfun(in1,in2,in3)
```

The variables `in1`, etc. are **input arguments**, and `out1` etc. are **output arguments**. You can have as many as you like of each type (including zero) and call them whatever you want. The name `myfun` should match the name of the disk file.

Here is a function that implements (badly, it turns out) the quadratic formula.

```
function [x1,x2] = quadform(a,b,c)

d = sqrt(b^2 - 4*a*c);
x1 = (-b + d) / (2*a);
x2 = (-b - d) / (2*a);
```

From MATLAB you could call

```
>> [r1,r2] = quadform(1,-2,1)

r1 =
     1

r2 =
     1
```

One of the most important features of a function is its **local workspace**. Any arguments or other variables created while the function executes are available only to the executing function statements. Conversely, variables in the command-line workspace (called the **base workspace**) are normally not visible to the function. If during the function execution, more functions are called, each of those calls also sets up a private workspace. These restrictions are called **scoping**, and they make it possible to write complex programs without worrying about name clashes. The values of the input arguments are copies of the original data, so any changes you make to them will not change anything outside the function's scope.[7] In general, the only communication between a function and its caller is through the input and output arguments. You can always see the variables defined in the current workspace by typing `who` or `whos` .

A single M-file may hold more than one function definition. A new function header line in a file ends the **primary function** and starts a new **subfunction**. As a silly example, consider

```
function [x1,x2] = quadform(a,b,c)

d = discrim(a,b,c);
x1 = (-b + d) / (2*a);
x2 = (-b - d) / (2*a);

function D = discrim(a,b,c)
D = sqrt(b^2 - 4*a*c);
```

A subfunction has its own workspace; thus, changes made to `a` inside `discrim` would not propagate into the rest of `quadform`. In addition, the subfunctions themselves have a limited scope. In the example the subfunction `discrim` is available *only* to the primary function `quadform` , not to the command line.[8]

Another important aspect of function M-files is that most of the functions built into MATLAB (except core math functions) are themselves M-files that you can read and copy. This is an excellent way to learn good programming practice.

To debug a program that doesn't work, you can set **breakpoints** in one or more functions. (See the Breakpoints menu in the Editor.) When MATLAB reaches a breakpoint, it halts and lets you

[7] MATLAB does avoid copying (i.e., "passes by reference") if the function never alters the data.
[8] MATLAB uses directory listings to see what functions are available at the prompt.

inspect and modify all the variables currently in scope—in fact, you can do anything at all from the command line. You can then continue execution normally or step by step. It's also possible to set non-specific breakpoints for error and warning conditions. See `help debug` for all the details.

3.3 Inline functions

Sometimes you may need a quick function definition that is temporary—you don't care if the function is around tomorrow. You can avoid writing M-files for such functions.

```
>> sincos = inline('sin(x) + cos(x)')

sincos =
     Inline function:
     sincos(x) = sin(x) + cos(x)

>> sincos([0 pi/4 pi/2 3*pi/4 pi])

ans =
    1.0000    1.4142    1.0000    0.0000   -1.0000
```

You can also define functions of more than one variable, and name the variables explicitly:

```
>> w = inline('cos(x - c*t)','x','t','c')

w =
     Inline function:
     w(x,t,c) = cos(x - c*t)

>> w(pi,1,pi/2)

ans =
   6.1232e-17
```

One use of inline functions is described in section 3.7.

3.4 Conditionals: if and switch

Often a function needs to branch based on runtime conditions. MATLAB offers structures for this similar to those in most languages.

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
```

The 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. As a condition is evaluated from left to right, it may become obvious before the end that truth or falsity is assured. At that point, 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.

The `if` / `elseif` construct is fine when only a few options are present. When a large number of options are possible, it's customary to use `switch` instead. For instance:

```
switch units
  case 'length'
    disp('meters')
  case 'volume'
    disp('liters')
  case 'time'
    disp('seconds')
  otherwise
    disp('I give up')
end
```

The switch expression can be a string or a number. The first matching `case` has its commands executed.[9] If `otherwise` is present, it gives a default option if no case matches.

3.5 Loops: for and while

Many programs require iteration, or repetitive execution of a block of statements. Again, MATLAB is similar to other languages here.

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

[9] Execution does not "fall through" as in C.

```
>> f = [1 1];
>> for n = 3:10
     f(n) = f(n-1) + f(n-2);
   end
```

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 10, with an execution of the body after each assignment. But remember that `3:10` is really just a row vector. In fact, you can use *any* row vector in a `for` loop, not just one created by a colon. For example,

```
>> x = 1:100; s = 0;
>> for j = find(isprime(x))
     s = s + x(j);
   end
```

This finds the sum of all primes less than 100 (though it's far from the fastest or shortest way).

A warning: If you are using complex numbers, you might want to avoid using `i` as the loop index. Once assigned a value by the loop, `i` will no longer equal $\sqrt{-1}$. However, you can always use `1i` for the imaginary unit.

It is sometimes necessary to repeat statements based on a condition rather than a fixed number of times. This is done with `while`.

```
while x > 1
  x = x/2;
end
```

The condition is evaluated before the body is executed, so it is possible to get zero iterations. It's often a good idea to limit the number of repetitions, to avoid infinite loops (as could happen above if `x==Inf`). This can be done using `break`.

```
n = 0;
while x > 1
  x = x/2;
  n = n+1;
  if n > 50, break, end
end
```

A `break` immediately jumps execution to the first statement after the loop.

3.6 Function handles

In many cases you need to use a function as an argument to another function. For example, the function `fzero` finds a root of a scalar function of one variable. So we could say

```
>> fzero(@sin,3)

ans =
    3.1416
```

Note the new syntax: The expression `@demo` is called a **function handle** and it gives `fzero` a way to accept the `sin` function as an input argument.

Why can't we just use `fzero(sin,1)`? MATLAB interprets the first argument as a *immediate call* to the `sin` function, which itself returns an error since it lacks an input. (Try it and see.) By creating the function handle you give `fzero` the power to control when `sin` is invoked, and the function itself becomes an input argument.

If you need to find the root of a more complicated function, or a function with a parameter, then you can write an M-file function and pass a handle to it. Say we have

```
function f = demo(x,a)
exp(x) - a*x;
```

Then we can use

```
>> fzero(@demo,1,[],3)

ans =
    0.6191

>> fzero(@demo,1,[],4)

ans =
    0.3574
```

Here we used the empty matrix `[]` as a placeholder—the online help tells you that the third argument otherwise has a particular meaning. `fzero` is set up so that every additional argument past the third is passed along verbatim to your function, in this case `demo`.

3.7 Function functions

Functions like `fzero` that accept other functions as arguments are called "function functions" in MATLAB lingo. The idea behind such functions is sometimes called **black-box programming**. The idea is that finding the root of a scalar function $f(x)$ is a universal procedure that has to be written well only once and can be used for many different definitions of f. In other words, you can separate (and recycle) the algorithms and applications of a particular numerical method. MATLAB comes with a large number of function functions, primarily for rootfinding, optimization, graphing, and differential equations. (See `help funfun` for a list.)

Sooner or later you will probably have to write function functions of your own. Say you want to use the bisection method of root finding. Here is a crude version.

```
function x = bisect(f,a,b)

fa = feval(f,a);
fb = feval(f,b);
while (b-a) > 1e-8
  m = (a+b)/2;
  fm = feval(f,m);
  if fa*fm < 0
    b = m;  fb = fm;
  else
    a = m;  fa = fm;
  end
end
x = (a+b)/2;
```

Note how `feval` is used to evaluate the generic input function `f`. The syntax `f(a)` would produce an error in most cases. Now we can call

```
>> bisect(@sin,3,4)

ans =
    3.1416
```

To see how to use optional extra parameters in `bisect` as we did with `fzero`, see section 5.2.

4 Graphics

Graphical display is one of MATLAB's greatest strengths—and most complicated subjects. The basics are quite simple, but you also get complete control over practically every aspect of each graph, and with that power comes complexity.

4.1 2D plots

The most fundamental plotting command is `plot`. Normally it uses line segments to connect points given by two vectors of x and y coordinates. Here is a simple example.

```
>> t = pi*(0:0.02:2);
>> plot(t,sin(t))
```

A line object is drawn in the current axes of the current figure (these things are created if necessary). The line may appear to be a smooth, continuous curve. However, it's really just a game of "connect the dots," as you can see by entering

```
>> plot(t,sin(t),'o-')
```

Now a circle is drawn at each of the points that are being connected. Just as `t` and `sin(t)` are really vectors, not functions, curves in MATLAB are really joined line segments. You may zoom in to see this by clicking on the magnifying glass icon in the figure and drawing a rectangle.[10]

If you now say

```
>> plot(t,cos(t),'r')
```

you will get a red curve representing $\cos(t)$. The curve you drew earlier is erased. To add curves, rather than replacing them, use `hold` .

```
>> plot(t,sin(t),'b')
>> hold on
>> plot(t,cos(t),'r')
```

You can also do multiple curves in one go, if you use column vectors:

```
>> t = (0:0.01:1)';
>> plot(t,[t t.^2 t.^3])
```

Other useful 2D plotting commands are given in Table 6. See a bunch more by typing `help graph2d` .

Table 6: 2D plotting commands

`figure`	Open a new figure window.
`subplot`	Multiple axes in one figure.
`semilogx` , `semilogy` , `loglog`	Logarithmic axis scaling.
`axis` , `xlim` , `ylim`	Axes limits.
`legend`	Legend for multiple curves.
`print`	Send to printer.

4.2 3D plots

Plots of surfaces and such for functions $f(x, y)$ also operate on the "connect the dots" principle, but the details are more difficult. The first step is to create a grid of points in the xy plane. These are the points where f is evaluated to get the "dots" in 3D.

Here is a typical example:

[10] A significant difference from Maple and some other packages is that if the viewpoint is changed to zoom in, the "dots" are *not* recomputed to give a smooth curve.

```
>> x = pi*(0:0.02:1);
>> y = 2*x;
>> [X,Y] = meshgrid(x,y);
>> surf(X,Y,sin(X.^2+Y))
```

Once a 3D plot has been made, you can use the rotation button in the figure window to manipulate the 3D viewpoint. There are additional menus that give you much more control of the view, too.

The `surf` plot begins by using `meshgrid` to make an x-y grid that is stored in the arrays `X` and `Y`. To see the grid graphically, use

```
plot(X(:),Y(:),'k.')
```

With the grid so defined, the expression `sin(X.^2+Y)` is actually an array of values of $f(x,y) = \sin(x^2 + y)$ on the grid. (This array could be assigned to another variable if you wish.) Finally, `surf` makes a solid-looking surface in which color and apparent height describe the given values of f. An alternative command `mesh` is similar to `surf` but makes a "wireframe" surface. The most common 3D plotting commands are shown in Table 7.

Table 7: 3D plotting commands

`surf` , `mesh` , `waterfall`	Surfaces in 3D.
`colorbar`	Show color scaling.
`plot3`	Curves in space.
`pcolor`	Top view of a colored surface.
`contour` , `contourf`	Contour plot.

4.3 Annotation

A bare graph with no labels or title is rarely useful. The last step before printing or saving is usually to label the axes and maybe give a title. For example,

```
>> t = 2*pi*(0:0.01:1);
>> plot(t,sin(t))
>> xlabel('time')
>> ylabel('amplitude')
>> title('Simple Harmonic Oscillator')
```

You can use the `legend` command or menu item to create an editable legend describing multiple curves. By clicking on the "A" button in the figure window, you can add text comments anywhere on the graph. You can also use the arrow button to draw arrows on the graph. This combination may be superior to a legend in some cases.

4.4 Auto function plots

When plotting a mathematical function, you must pick the evaluation points of the plot before calling `plot` or `surf` . This extra step can be skipped by using special alternative plotting commands for mathematical expressions.

```
>> ezplot('exp(3*sin(x)-cos(2*x))',[0 4])
>> ezsurf('1/(1+x^2+2*y^2)',[-3 3],[-3 3])
>> ezcontour('x^2-y^2',[-1 1],[-1 1])
```

5 Other data structures

Not long ago, MATLAB viewed every variable as a matrix or array. While this point of view is ideal for simplicity, it is too limited for some tasks. A few additional data types are provided.

5.1 Strings

As we have seen, a string in MATLAB is enclosed in single forward quotes. In fact a string is really just a row vector of character codes. Because of this, strings can be concatenated using matrix concatenation.

```
>> str = 'Hello world';
>> str(1:5)

ans =

Hello

>> double(str)

ans =
    72   101   108   108   111    32   119   111   114   108   100

>> char(ans)

ans =

Hello world

>> ['Hello',' ','world']

ans =

Hello world
```

You can convert a string such as '3.14' into its numerical meaning (not its character codes) by using `eval` or `str2num` on it. Conversely, you can convert a number to string representation using `num2str` or the much more powerful `sprintf` (see below). If you want a quote character within a string, use two quotes, as in `'It''s Cleve''s fault'` .

Multiple strings can be stored as rows in an array. However, arrays have to have to be rectangular (have the same number of columns in each row), so strings may have to be padded with extra blanks at the end. The function `str2mat` does this.

```
>> str2mat('Goodbye','cruel','world')

ans =

Goodbye
cruel
world

>> size(ans)

ans =
     3     8
```

There are lots of string handling functions. See `help strfun` . Here are a few:

```
>> upper(str)

ans =

HELLO WORLD

>> strcmp(str,'Hello world')

ans =

     1

>> findstr('world',str)

ans =

     7
```

Formatted output

For the best control over conversion of numbers to strings, use `sprintf` or `fprintf` . These are closely based on the C function *printf*, with the important vectorization enhancement that format

specifiers are "recycled" through all the elements of a vector or matrix (in the usual row-first order).

For example, here's a script that prints out successive Taylor approximations for $e^{1/4}$.

```
x=0.25;  n=1:8;  c=1./cumprod([1 n]);
for k=1:9,  T(k)=polyval(c(k:-1:1),x);  end
fprintf('\n     T_n(x)       |T_n(x)-exp(x)|\n');
fprintf('-----------------------------------\n');
fprintf('%15.12f      %8.3e\n', [T;abs(T-exp(x))] )
```

```
     T_n(x)       |T_n(x)-exp(x)|
-----------------------------------
 1.000000000000      2.840e-01
 1.250000000000      3.403e-02
 1.281250000000      2.775e-03
 1.283854166667      1.713e-04
 1.284016927083      8.490e-06
 1.284025065104      3.516e-07
 1.284025404188      1.250e-08
 1.284025416299      3.892e-10
 1.284025416677      1.078e-11
```

5.2 Cell arrays

Collecting objects of different sizes is a recurring task. For instance, suppose you want to tabulate the Chebyshev polynomials: $1, x, 2x^2 - 1, 4x^3 - 3x$, etc. In MATLAB one expresses a polynomial as a vector (highest degree first) of its coefficients. The number of coefficients needed grows with the degree of the polynomial. Although you can put all the Chebyshev coefficients into a triangular array, this is an inconvenient complication.

Cell arrays are used to gather dissimilar objects into one variable. They are indexed like regular numeric arrays, but their elements can be absolutely anything. A cell array is created or referenced using curly braces {} rather than parentheses.

```
>> str = { 'Goodbye', 'cruel', 'world' }

str =

    'Goodbye'    'cruel'    'world'

>> str{2}

ans =

cruel
```

```
>> T = cell(1,9);
>> T(1:2) = { [1], [1 0] };
>> for n = 2:8,  T{n+1} = [2*T{n} 0] - [0 0 T{n-1}];  end
>> T

T =
  Columns 1 through 5

    [1]    [1x2 double]    [1x3 double]    [1x4 double]    [1x5 double]

  Columns 6 through 9

    [1x6 double]    [1x7 double]    [1x8 double]    [1x9 double]

>> T{4}

ans =
     4     0    -3     0
```

Cell arrays can have any size and dimension, and their elements do not need to be of the same size or type. Cell arrays may even be nested. Because of their generality, cell arrays are mostly just containers; they do not support any sort of arithmetic.

One special cell syntax is quite useful. The idiom `C{:}` for cell array `C` is interpreted as a comma-separated list of the elements of `C`, just as if they had been typed. For example,

```
>> str2mat(str{:})  % same as str2mat('Goodbye','cruel','world')

ans =
Goodbye
cruel
world
```

The special cell array `varargin` is used to pass optional arguments into functions. For example, consider these modifications to the bisection algorithm on page 26:

```
function x = bisect(f,a,b,varargin)

fa = feval(f,a,varargin{:});
fb = feval(f,b,varargin{:});
...
```

If arguments beyond the first three are passed in to `bisect`, they are passed along to `f`. Naturally, in other contexts you are free to look at the elements of `varargin` and interpret them yourself. There is also a `varargout` for optional outputs.