Inessential MATLAB on Athena[1]

Version 1.02

/afs/sipb.mit.edu/project/doc/current/imatlab.dvi

The Student Information Processing Board
Jamie Morris
jemorris@mit.edu

June 29, 1997

[1]Copyright © 1996 Student Information Processing Board of the Massachusetts Institute of Technology

Contents

1 Introduction

1.1 What is MATLAB?

MATLAB (MATrix LABoratory) is an interactive system for matrix-based computation designed for scientific and engineering use. It is good for many forms of numeric computation and visualization. Besides dealing with explicit matrices in linear algebra, it can handle differential equations, polynomials, signal processing, and other applications. Results can be made available both numerically and as excellent graphics.

Some classes at MIT require the use of MATLAB; it is also useful for many that don't. It may not be most suitable tool for your needs; e.g. Maple and Mathematica are better for symbolic math (MATLAB's symbolic math draws on part of Maple), Xess for spreadsheet work.

This document is intended partially as a MATLAB tutorial and partially as a reference.

1.2 Getting into MATLAB

As you use MATLAB you will create data files, function files, and script files. While you can arrange for these files to be found by MATLAB no matter where you put them (see **4.3.3 Paths**), it is simplest to use the default behavior by putting all these files in a directory called **matlab** in your home directory. To create this directory, do (only once ever):

athena% **mkdir ~/matlab**

MATLAB will then use this directory for files unless you tell it not to. (See **4.3 M-files**.) If you created this directory *after* starting a MATLAB session, you will need to do **addpath('~/matlab')** in MATLAB that session to tell it the directory exists; future sessions will notice it upon startup.

To get a session of MATLAB, do

athena% **add matlab**
athena% **matlab &**

(if you have already done **add matlab** in that login session, you need not repeat it).

MATLAB may also be started from the menu bar; it is in the "Math/Plotting" menu.

When you start MATLAB, it creates a new **xterm**, with the MATLAB header and MATLAB >> prompt. As you create graphics in MATLAB, it will pop up additional windows for them.

1.3 Demos

The MATLAB demo is very instructive if you've never used MATLAB. To get it, do

>> **demo**

Also see **help demo** for a list of demostration facilities.

1.4 Help

MATLAB has several internal help features. To get a listing of help topics, do

>> **help**

To get more help on a specific topic,

>> **help** *topic*

In particular, **help general** will give you a table of contents listing many of the available functions by category. Topics also include every individual function, so you can do

>> **help** *function*

You may also search for functions whose descriptions contain a keyword. To use this, do the following and go for a water cooler break (the search will be slow):

>> **lookfor** *keyword*

The MATLAB on Athena Homepage, which you can get to with **add matlab** and then the URL **file://localhost/mit/matlab/www/home.html**, and has pointers to lots of information about MATLAB. Type **doc** at the MATLAB prompt to start up a web browser on this page.

Complete documentation on MATLAB, including the **MATLAB New Features Guide**, **MATLAB Release Notes**, and the **SIMULINK User's Guide**, are available for reference in the Athena Consulting Office (11-115) and in Barker and Hayden Libraries. The Barton catalog at **http://libraries.mit.edu/barton** lists several textbooks in various topics, which use Matlab as a tool for walking through their subject matter. There is also the Mathworks web site, **http://www.mathworks.com**.

2 The Basics

2.1 Fundamentals of Use

Your interaction with MATLAB is in the form of you giving MATLAB a statement, possibly followed by MATLAB giving you a result, then back to you giving a statement. A statement is usually of the form of either

>> *variable* = *expression*

or just

>> *expression*

The result (which is almost always a matrix — see **2.2 Building Matrices**) of evaluating *expression* is immediately displayed. To suppress display, end the line with a semicolon; this will greatly speed up calculations resulting in large matrices. If no variable is given in which to store the result of *expression*, it is automatically stored in the variable *ans*.

```
>> x = 3 + 2
x =
   5
>> y = 4 + 6;
>> y
y =
   10
>> 5 + 3
ans =
   8
```

To put more than one statement on a single line, separate them by commas or semicolons. To have a single statement span several lines, end each line before the last with a space followed by three or more periods.

```
>> a = 1 + 1, b = 2 + 2
a =
   2
b =
   4
>> c = 1 + 2 + 3 ...
>> + 4 + 5 + 6 + ...
>> 7 + 8 + 9 + 10
c =
   55
```

A function may have more than one output, and moreover, all the outputs may depend on how many outputs you request. For example, **[val vec] = eig(***matrix***)** gives a (diagonal) matrix of eigenvalues of *matrix* and a matrix of the eigenvectors of *matrix*; however, **v = eig(***matrix***)** gives a vector of eigenvalues — which is not the same as either of the former two outputs. See the **help** on a function for details; it is generally wise to look at the **help** before first using any function.

MATLAB is, by default, case-sensitive to names, so `myvar` and `myVAR` are distinct. Typing **casesen** toggles this sensitivity on and off.

Normal scalar arithmetic works; you can use MATLAB as a basic calculator. The arithmetic functions are +, -, ∗, /, and ^ (exponentiation). Use parentheses as usual to specify precedence. Also see **help arith**.

MATLAB always does complex math, though it will display purely real results intelligently by omitting the imaginary part. Both `i` and `j` are equal to the square root of -1. If you happen to overwrite both of these (by using them as variables for something else), you can recreate with, e.g.

```
>> myi = sqrt(-1)
```

or do **clear i** to remove your definitions of **i**, leaving it with its original meaning.

A complex number is specified in the form *1+2i* or *1-2i*, where there are no spaces inside the number. Spaces can cause complex numbers in vectors and matrices to seem like two numbers, which will produce errors or at least undesired results.

```
>> x = (1+2i)^2
x =
   -3.0000 + 4.0000i
```

Mathematical functions will accept complex arguments and, in general, give complex answers (e.g. the trigonometric functions and arcfunctions are defined over the complex plane). Specifically complex manipulations are available in the form of **real** and **imag** (giving the real and imaginary parts of their argument), **conj** for complex conjugation, **abs** for absolute value, and **angle** for the phase (in radians; also see **help unwrap**).

See **help elfun** for a list of trigonometric, exponential, complex, and numeric elementary scalar functions. For more specialized functions see **help specfun**.

There are several commands to help with variable assignments; **who** will list the variables currently in use in this session, and **whos** lists the variables along with their sizes and whether they are purely real. Use **clear** *variable* to remove *variable* from those currently in use; **clear** by itself removes all variables.

The MATLAB-defined constant **eps** gives the machine unit roundoff, typically about 10^{-16};

this is useful in specifying tolerances. The constant **pi** exists. Some other predefined constants are **realmax** and **realmin** for largest and smallest (positive) floating point numbers, **inf** for infinity, **NaN** for Not-a-Number, and **version** for the MATLAB version number.

To halt a runaway computation, use Control-C. Use **more on**, **more off**, and **more(**n**)** to have MATLAB scroll as the UNIX `more`, turn off such scrolling, and set the number of lines per page (default 23). This is useful with calculations giving long results and with long **help** entries. Use **clc** to clear the command window, and **home** to send the cursor to the upper left without clearing.

Line editing in MATLAB will Do The Right Thing: you can use the up and down arrows to get at old command lines, left and right arrows to move in and edit these lines, etc. For more details see **help cedit** or **type cedit** in MATLAB.

2.2 Building Matrices

Everything in MATLAB is a matrix. A vector is a 1xN or Nx1 matrix; a scalar is a 1x1 matrix. The most straightforward way to make a vector or matrix is to write it out:

>> **v = [1 2 3 4 5]**

creates the obvious vector (the spaces may be replaced with commas); the statements

>> **m = [1 2 3; 4 5 6; 7 8 9]**

and

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

both create the obvious 3×3 matrix. Note that semicolons are used to separate the rows of the matrix in the one-line square bracket notation.

It can be convenient to enter a complex-valued matrix or vector in the form

>> **m = [1 2; 3 4] + i * [4 3; 2 1]**

instead of the equivalent

>> **m = [1+4i 2+3i; 3+2i 4+i]**

There are a number of useful functions to create a matrix from scratch. Specifically, **zeros(**M**,** N**)** and **ones(**M**,** N**)** create $M \times N$ all-zero and all-one matrices respectively, **eye(**N**)** the size-N identity matrix, **rand(**M**,** N**)** a uniformly random matrix, **randn(**M**,** N**)** a normally distributed one, and **hilb(N)** a Hilbert matrix, among others. Some more specialized matrix generators are listed by **help specfun**. Those that take arguments (M, N) will create a square matrix if given only (N).

The function **size** takes a matrix and returns the vector [*width height*]; this can be used with other functions to build matrices the same size as existing ones, e.g.

>> **emptym = zeros(size(m))**

The **length** function gives the maximum of *width* and *height*.

Matrices and vectors may be built out of smaller ones;

```
>> va = [1 2 3]; vb = [4 5 6]; vc = [7 8 9];
>> m = [va; vb; vc]
```

produces the same 3×3 matrix as before. Similarly, matrices can be built in blocks:

```
>> ma = [1 2; 3 4]
ma =
   1 2
   3 4
>> mb = [5 6; 7 8];
>> mwide = [ma mb]
mwide =
   1 2 5 6
   3 4 7 8
>> mtall = [ma ; mb]
mtall =
   1 2
   3 4
   5 6
   7 8
```

Since the size of a matrix can be changed at any time (and can be zero), things like this are valid:

```
>> x = [ ];
>> x = [x 1]
x =
   1
>> x = [x 2]
x =
   1 2
>> x = [x 3 x]
x =
   1 2 3 1 2
```

Note that resizing a matrix does take time; doing it ten thousand times in a loop can be noticeable. A way to avoid this is to start with a matrix of zeros (or ones, as appropriate) of the eventual largest size, and then to use only part of the matrix at a time.

There are operations specifically to let you get a new object from an existing one; **triu** and **tril** take a matrix and produce its upper and lower triangular matrices. **Diag** takes a matrix and produces the vector of its diagonal entries, or takes a vector and produces a diagonal matrix with those entries. **fliplr** and **flipud** take a matrix and flip it left-right or up-down; **rot90** takes a matrix and rotates it counterclockwise 90 degrees (an optional integer second argument specifies repetition). The operator **'** takes the conjugate transpose of a matrix; the operator **.'** takes the transpose. The **reshape** function rearranges matrix elements into a new shape.

```
>> [1 2+i]'
ans =
   1
   2-i
```

Elements of a matrix are referenced as $m(row, column)$ (or $v(n)$ for a vector); this can be used to access elements, or to build a matrix. Indexing starts at one, not at zero, so it looks like

(1, 1)	(1, 2)	...
(2, 1)	(2, 2)	...
...	...	...

You could, painfully, build a matrix like

```
>> m(1, 1) = 1; m(1, 2) = 2; m(2, 1) = 3; m(2, 2) = 4
m =
   1 2
   3 4
```

but this is more useful used in conjuction with **for** and **while** statements (see **2.4 Flow**). On the other hand, submatrices can often be done this way quickly:

```
>> m = [1 2 3; 4 5 6; 7 8 9]; s = m([1 2], [2 3])
s =
   2 3
   5 6
```

The colon operator is critical to efficient use of MATLAB. The expression $a : b$ evaluates to a row vector with values starting at a and incrementing by one as long as they are no greater than b. The expression $a : s : b$ uses increments of s instead of one. A colon by itself as a matrix index, e.g. $m(1, :)$ or $m(:, 5)$, yields the entire row or column, while a colon used as the *only* index to a matrix turns it into a vector by stringing the columns together.

Also see **help colon**.

```
>> v = 1:5
v =
   1 2 3 4 5
>> w = 5:-1:1
w =
   5 4 3 2 1
>> m = [1 2 3; 4 5 6; 7 8 9];
>> s = m(1:2, 2:3)
s =
   2 3
   5 6
>> t = m(1:2, 3:-1:2)
t =
   3 2
   6 5
>> m(1:2, :)
ans =
   1 2 3
   4 5 6
>> m(:)'
ans =
   1 4 7 2 5 8 3 6 9
```

In a similar fashion, **linspace(low, high, n)** gives a linearly evenly spaced vector of n points from *low* to *high*; n is 100 if omitted. Further, **logspace(low, high, n)** gives a logarithmically evenly spaced points between 10^{low} and 10^{high} (n defaults to 50); however, if *high* is **pi**, the high end is π itself and not 10^{π}.

Finally, matrices can be loaded from a disk file created via MATLAB or some other application (e.g. a text editor); this is the fastest method for large matrices. If a file named *base.extension* contains a series of rows of numbers (no other formatting) defining a matrix, **load base.extension** will assign that matrix to the variable *base*. E.g. **load values.txt** takes data from the file **values.txt** and puts it into **values**. Don't use a **.mat** extension, since that will make **load** assume it is a MATLAB-saved file. Since **load** is frequently used for very large matrices, it does not echo the input.

2.3 Matrix and Element-wise Operations

Some operations are intended for matrices in particular. These include the conjugate and non-conjugate transpose operators **'** and **.'**, the matrix multiplication operator $*$, and the left and right matrix "division" operators \ and /. For instance, if **A** is a matrix and **x** and **b** are vectors, then the lines

```
>> A'
>> A^2
>> A * x = b
>> x = A\b
>> x * A = b
>> x = A/b
```

respectively take the conjugate transpose of A, take the square of A, give a typical matrix equation involving matrix multiplication, give the solution for that equation, give another matrix equation, and give the solution for the second equation.

Such solutions to matrix equations are solved exactly (with Gaussian elimination) if the matrix is square; others are solved in a least-squares sense (with Householder orthogonalization). Also see **help slash**.

To get inner and outer products of vectors, remember their formal definitions. The inner product is given by **x' * y = y' * x**, and the outer products by **x * y'** and **y * x' = (x * y')'**.
MATLAB understands multiplication and division between a matrix and a scalar in the normal sense;

```
>> 10 * [1 2; 3 4]
ans =
   10 20
   30 40
```

If you want to take two matrices (or vectors) and multiply or divide them element by element, or if you want to exponentiate each element of a matrix, place a period before the operator. These are *array operations* as opposed to matrix operations.

```
>> [1 2; 3 4].^2
ans =
   1 4
   9 16
```

Exponentiation also has both matrix and array forms. If x and y are scalars and A and B are matrices, $y\hat{}x$, $A\hat{}x$, and $x\hat{}A$ have their usual mathematical meanings. Array exponentiation is available with $A.\hat{}x$ to raise each element to a power, and $A.\hat{}B$ to raise each element of A to the power of the corresponding element of B.

MATLAB also has a large number of matrix functions to implement common mathematical operations, such as finding eigenvalues and eigenvectors. For instance, **kron** will give the (Kronecker) tensor product. See **help matfun**.

If you apply a function that operates on scalars to a matrix or vector, or if you apply a function that operates on vectors to a matrix, MATLAB performs the operation *element-wise*. Scalar functions will be applied to each element of the matrix, and the result will be a matrix of the same size. Vector functions will be applied to each column of the matrix, and the result will be a row vector of the same width. (Use the transpose operators to effect row-by-row application.) See **help funm** if you want to use the matrix and not the array version of a function. Lastly, functions defined strictly on the real line are applied separately to the real and imaginary parts of a complex number.

```
>> sin([0 (pi/6) (pi/2) pi])
ans =
   0 .5 1 0
>> max([1 10; 20 2])
ans =
   20 10
>> max(max([1 10; 20 2]))
ans =
   20
>> round(1.7+3.2i)
ans =
   2 + 3i
```

Certain functions are particularly useful for this. The functions **max**, **min**, **median**, **mean**, **std**, **sum**, and **prod** take a vector and return its maximum, minimum, median, arithmetic mean, standard deviation, element sum, and element product (respectively). Applied to a matrix, they return a row vector of the result on each column. **sort** sorts a vector (or each column of a matrix) in ascending order. Also see **help datafun**.

Applying operations element-wise is a powerful feature of MATLAB and using it is the fastest and best way to accomplish most things.

2.4 Flow

2.4.1 Relations

MATLAB provides these logical operators:

&	and
|	or
~	not
xor	exclusive or

and these boolean relations:

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

The result of applying any of these to a scalar or pair of scalars (as appropriate) is always a 1 (True) or a 0 (False). They operate element-wise on matrices; there are no "full matrix" versions. To test whether a relation holds over an entire matrix, or whether it holds anywhere in the matrix, the commands **any** and **all** are useful. Each takes a vector; **any** ORs the elements (true iff any are true) and **all** ANDs the elements (true iff all are true). They operate column-by-column on a matrix to produce a vector. So, for instance,

```
>> all(all(A == B))
>> any(any(A ~= B))
```

both test for whether the matrices A and B are entirely equal, the former returning 1 if so (because every element in every column matches), the latter returning 1 if not (because some element in some column did not match). The function **find** will return the indices of the non-zero (True) elements of a matrix; this is useful for selective indexing.

There are also a number of boolean tests for various characteristics; see **help ops** for most of them.

2.4.2 Control

You can do normal programming things in MATLAB. The explicit looping operators are **while** and **for**. The syntax of **while** is

while *relation*
  statements
end

The indentation is customary but not required. As long as *relation* holds true, *statements* will be executed repeatedly. The **end** is mandatory. When *relation* evaluates to False (0), execution resumes after **end**.

for *variable* = *matrix*
  statements
end

causes *variable* to assume the value of each column of *matrix* in succession, and execute *statements* for each, then resume after the mandatory **end**. Frequently *matrix* will be a vector, e.g.

for n = 1:5

will run **n** from 1 to 5.

If MATLAB encounters the command **break** inside a **for** or **while** loop, it immediately jumps to the **end**. If there are nested loops, only the innermost is broken out of.

Conditional control and branching are accomplished with **if**, whose most general syntax is

if *relation*
  statements
elseif *another relation*
  more statements
elseif *a third relation*
  a third set of statements
else
  last statements
end

There can be any number of **elseif**s, and the final **else** is optional; the **end** is mandatory as usual. The first *relation* of an **if** or **elseif** that evaluates to True (anything non-zero) is used; its *statements* are executed and control then passes to after the **end**. If no *relation* is True, the **else** *statements*, if any, are executed.

All control structures can be nested. Control structures are relatively slow and inefficient in MATLAB; what MATLAB is good at is matrices. Instead of applying **for** over a vector, you can usually apply your function(s) to the vector directly so it operates element-wise; this will be much faster. Instead of **for** with **if**, to test over the elements of a vector, you can and should use selective indexing (see below).

2.4.3 Selective Indexing

The syntax $M(row,\ column)$ to examine or change a specific element of matrix M has already been explained (**2.2 Building Matrices**). This can be generalized by replacing *row* and/or *column* with vectors of the same height and/or width (respectively) as M with entries all 1 (True) or 0 (False). This specifies a matrix with entries corresponding to those in M whose indices are specified as True in the vectors.

```
>> x = [1 2 3 4 5];
>> x([1 0 1 0 1])
ans =
  1 3 5
>> m = [1 2 3; 4 5 6; 7 8 9];
>> m([1 0 1], [1 1 0])
ans =
  1 2
  7 8
>> m(3, [1 0 1])
ans =
  7 9
```

One rarely enters the indexing vectors by hand this way. Since a relation operator applied to a matrix results in a matrix of 1s and 0s, it is naturally useful to index a matrix of the same size.

```
>> x = 3:7
x =
  3 4 5 6 7
>> big = x > 4
big =
  0 0 1 1 1
>> x(big)
ans =
  5 6 7
>> x(x > 3 & x < 6)
ans =
  4 5
```

This is even more useful when specifying elements, or specifying *changes* to elements, rather than merely viewing them.

```
>> disp(x)
  3 4 5 6 7
>> x(x > 4) = x(x > 4) + 10
x =
  3 4 15 16 17
>> x = 1:5;
>> x(x > 2) = x(x < 4) + 10
x =
  1 2 11 12 13
>> x = 1:5; y = x.^2
y =
  1 4 9 16 25
>> x(y > 8) = x(y > 8) + y(y > 8)
x =
  1 2 12 20 30
```

Let us examine the second example in more detail. We make x [1 2 3 4 5]. The relation $x > 2$ is then the vector [0 0 1 1 1]; therefore $x(x > 2)$ is the last three elements of x, which currently hold [3 4 5]. On the other side, $x < 4$ is [1 1 1 0 0], so $x(x < 4)$ is the first three elements of x, which currently hold [1 2 3]. To this matrix is added the scalar 10; adding a scalar to a matrix is always elementwise, so the right-hand side becomes [11 12 13]. Since this is a 1x3 vector and the left-hand side is also a 1x3 vector, the assignment is valid, and the left-hand side (i.e. the last three elements of x) becomes [11 12 13]. MATLAB then prints the entirety of the vector x, including the parts that have not changed: [1 2 11 12 13].

The first and third examples, $x(x > 2) = x(x > 2) + 10$ and $x(y > 8) = x(y > 8) + y(y > 8)$, are perhaps less general but more typical. They could each be replaced with a **for** loop over the vectors with an **if** to perform the test. The selective indexing method is *much, much faster*, because MATLAB is specifically built to do this sort of thing. The code is also more readable. Therefore, you should use MATLAB in this form whenever possible. Between selective indexing and the colon operator, most procedures can be “vectorized.”

3 Graphics

Also see **help graphics**.

3.1 2-D Graphics

The **plot** command suffices for basic 2-D graphics. If x and y are equal-length vectors, **plot(x, y)** will plot the values of y on the y-axis versus those of x on the x-axis. If the first argument is omitted, it will be taken to be the indices of y — that is, 1:n where n is the length of y. Since graphs usually begin at zero, and MATLAB indexes from one, this is rarely wanted. If y has n elements, **plot(0:**$n-1$**, y)** may be more appropriate. A frequent usage is the form **x = 0:.01:1, y = sin(2 * pi *x), plot(x, y)**. Parametrics, e.g. **t = 0:.01:1, x = sin(t), y = cos(t), plot(x, y)** work well, since the arguments just expand into the same vector.

To change the style of the lines, put in a third argument, in single quotes. E.g. **plot(x, y, '-')** specifies a solid line (the default). `'--'` is a dashed line, `':'` dotted, `'-.'` alternating dots and dashes. To plot only the data points and not draw lines between them, put `'.'` (points), `'+'` (crosses), `'*'` (stars), `'o'` (circles), or `'x'` (x's).

To specify a color, give the appropriate letter in single quotes as a third argument. To specify both color and line style, give both (in that order) as a single single-quoted argument, with no spaces; e.g. `'b:'` is a blue dashed line. Available colors are white (w), black (k), red (r), green (g), blue (b), cyan (c), magenta (m), and yellow (y).

To put multiple pictures in the same graph window, simply give all their arguments to a single **plot**. E.g. **plot(x1, y1, x2, y2)** plots $x1$ versus $y1$ and $x2$ versus $y2$. **plot(x1, y1, 'b:', x2, y2)** does the same, but with the former as a dotted blue curve.

For multiple nonoverlapping plots in one window, use **subplot(**x, y, n**)**, where x and y are each 1 or 2, to break the figure window into an x by y set of subgraphs, selecting the nth as the current subplot. MATLAB will rotate the current subplot clockwise by default (see **3.2 Handles** and use **axes** to override this; subplot 1 is in the upper left). Use **subplot** with no arguments to regain the single-plot figure window.

For multiple figure windows, **figure** n will make the nth figure window current, creating it if necessary; **figure** by itself creates the next figure window. Use **gcf** to see which figure is current.

Use **clf** and **cla** to clear the current figure or axes.

Replacing **plot** with **loglog**, **semilogx**, or **semilogy** produces a plot that is logarithmic in both axes, the x axis, or the y axis, respectively (non-logarithmic axes remaining linear). For other kinds of plots, use **help** on **polar**, **bar**, **hist**, **quiver**, **compass**, **feather**, **rose**, **stairs**, **fill**.

Both **xlabel** and **ylabel** take a text string (in single quotes) and use it to label the appropriate axis. Similarly **title** uses its text string argument as the graph's title. To place text arbitrarily, use **gtext** to place via mouse and **text** to place via specifying coordinates — **help** for details. Use **grid** to get gridlines. Stylized versions are available as **stext**, **sxlabel**, **stitle**, etc (see **help**).

You can *find* the coordinates of a series of points with **ginput** by clicking with the mouse.

MATLAB will auto-scale the axes by default, using the minimum and maximum values of the data given to it for each axis. To examine or change this behavior, use **axis**. **axis(**$xmin, xmax, ymin, ymax$**)** sets the limits. **v = axis** returns in v the current limits, in the same order. **axis(axis)** prevents the graph from rescaling when you make a new plot, **axis auto** returns to auto-scaling, **axis square** uses the same scaling for both axes, **axis equal** uses the same scaling and ticmarks for both, and **axis on** and **axis off** turn scaling and ticmarks on and off. See **help axis** and **help hold** for details.

Given a function, **fplot('***function name***',** [*xmin xmax*]**)** will plot the function in the given range; this can offer a more convenient way to view a MATLAB-defined or M-file function than constructing vectors of values by hand. See **help fplot**. (Remember that MATLAB isn't big on symbolic things like this, though; a problem set asking you to plot a function probably wants it evaluated on a vector of points.)

Also see **help plotxy**.

3.2 Handles

Every feature of a figure on display can be edited as needed. Simply click on "Figure Properties" in the "Edit" menu to go into the property editor, and you will see all of the objects you can change. These same objects can also be edited from the Matlab prompt as is descripbed below.

A *handle* is a special MATLAB object attached to part of a plot that lets you find and change the properties of that part. The boolean **ishandle** tests whether something is a handle.

A plot always has at least two parts with handles, the figure and the axis. The former has general characteristics of the entire window; the latter has characteristics of the axes. The special constant **gcf** (for **g**et **c**urrent **f**igure) is a handle to the current plot's figure; **gca** (**g**et **c**urrent **a**xis) is a handle to the current plot's axis; and **gco** (**o**bject) is a handle to the last piece of the plot you clicked on with the mouse. Various graphics commands return handles; generally a handle is returned by any function that creates or adds something to a plot. **figure**, **axes**, and **text** all do this. See the help entries for specific other functions for their return values.

Objects with handles have *properties*. Each property has a name, which is a text string. The value of a property of an object is given by **get(***handle***,** *propertyname***)**; if the property name is omitted, all properties are listed (with their names). You change these values with **set**, whose most general format is **set(***vector of handles***,** *property name***,** *property value***,** *another property name***,** *value for that property***,** *a third property name***,** *value for third property***, ...)**. For each handle in the vector (a single handle is fine), each given property is set to the specified value. Omitting the property values will display the possible values for the specified property, with curly braces {} around the current value. Also omitting the property names gives such a list for all the object's properties.

Some objects have *children*, which are objects under them; you can get handles to these objects from their parent by **get**ting the 'Children' property. You can tell a parent that you want its children to have certain defaults unless they override; **set** on the parent the property of 'Default' followed by the name fo the object type (given by the 'Type' property) followed by the defaulting property name (e.g. 'DefaultTextColor').

Three special property values can be given for any property. Giving 'default' resets the property to its default value as affected by any parental defaults; giving 'factory' sets to the default value ignoring any parental defaults. The 'remove' value removes the default value.

To return all properties (except position) to the factory defaults use **reset**. An object can be destroyed with **delete(***handle***)**. The **findobj** function can be used to seek for objects with given property values.

A typical strategy is to get handles to the top objects, look through their children for the object you want, look through its properties for the one you want, see what possible values it has, and set the one you want.

3.3 Saving Graphs and Producing Hardcopy

The figure's file menu will let you save it and all of its attributes and properties under a filename of your choosing. The format is Matlab's **.fig** format. For important figures, it is recommended that you do this.

The **print** command, by itself, sends the current graphics figure to the printer. See **help printopt** about setting default options in an M-file.

MATLAB will print in portrait orientation (vertical longest, in the middle of the page, 4/3 aspect ratio) unless you specify otherwise with **orient** prior to printing. **orient landscape** changes to landscape (horizontal longest), **portrait** to the default, and **tall** to full-page portrait; with no argument, **orient** returns the current orientation.

To save the picture, **print** *filename* will save the current graphics figure; if *filename* does not have an appropriate suffix (e.g. **.ps**, **.eps**, **.jet**), the default mode will be used and the default suffix added. Specify **-append** to append to an existing file instead of overwriting it (the default). To save a non-current figure, specify **-f***number*; to save in a specific format, specify **-d***device*. See **help print**.

Other formats for printing or displaying the figure are available from the "Export" dialog in the "File" menu. Among these are the PDF and PNG formats.

3.4 3-D

Also see **help graph3d**.

3.4.1 Line Plots

A one-dimensional curve in 3-D is produced by **plot3**, which is completely analogous to **plot**; **plot3(x, y, z)** plots the curve, **plot3(x, y, z, 'b:')** makes it a blue dotted line, etc. As with **plot**, including more than one triplet (or quadruplet) of values in a single call overlays the plots. There exists **zlabel** exactly like **xlabel** and **ylabel**, and **axis** actually has six arguments, the last two being for z.

3.4.2 Surface Plots

Both **surf** and **mesh** plot a two-dimensional curve in 3-D; **surf** facets the surface, while **mesh** plots the facet outlines only. They are identical otherwise, including in taking arguments. There are variants; **surfc** and **meshc** include a contour plot (see **help contour**), **surfl** provides lighting, **meshz** provides a reference plane, and **waterfall** gives a mesh without column lines.

The most general argument to **surf** is a vector of four matrices, e.g. **surf(X, Y, Z, C)**. The first three matrices give the 3-D coordinates of each point on the surface, and the last gives its color as per the current colormap. Omitting the last, e.g. **surf(X, Y, Z)**, uses **C** = **Z**, i.e. color is proportional to height.

The **X** and **Y** matrices can be replaced by vectors of the width and height of **Z**; both or neither must be replaced. A **X** matrix is then used with the **x** vector as each row, and a **Y** matrix with the **y** vector as each column. If **X** and **Y** are omitted entirely, the vectors 1:width and 1:height are used instead.

To evaluate a function that can be performed elementwise on a matrix over a grid of points on the plane and plot it at even intervals, use **meshgrid** to create **X** and **Y** matrices from vectors with **[X, Y] = meshgrid(x, y)** (all the rows of **X** are **x**, all the columns of **Y** are **y**). Then e.g.

Z = X.*Y and **mesh(Z)** will display (in this case) the coordinate products, but at evenly spaced intervals as the **x** and **y** vectors did not directly reach **mesh**.

Portions of a surface plot may be hidden behind others. One method of dealing with this is to change the viewpoint; see **help view**. Alternatively, since MATLAB does not plot values of NaN and Inf, you can make "cutouts" in a plot by setting the color value on some region of facets to one of those.

Other features of 3-D plots can be set with **caxis**, **colormap**, **shading**, and **view**, and various **color** functions, for all of which see the **help** entries, as well as the commands described above for 2-D plots.

3.5 Animation

MATLAB provides simple animation capabilities with three movie functions: **moviein**, **getframe**, and **movie**. A movie consists of a series of frames, a frame being a special MATLAB object that holds a copy of the current picture, based on the current axis. A movie is a matrix whose columns are frames.

Because of the way MATLAB handles matrix resizing, you will suffer if you do not preallocate the movie matrix. Use *mov* = **moviein(***nframes***)** to make *mov* a matrix capable of holding *nframes* frames of the current axis and contents. If you will be getting frames (see below) for a specific object, and perhaps in a specific position rectangle, allocate the matrix as *mov* = **moviein(***nframes***,** *handle***,** *rectangle***)** or *mov* = **moviein(***nframes***,** *handle***)**.

Build your movie matrix out of columns provided by **getframe**. By itself **getframe** gets a frame of the current axis and contents. As **getframe(***handle***)** (or **getframe(***handle***,** *rectangle***)**) it gets a frame from the object specified by *handle* (in the *rectangle* region, from the lower left corner, in the 'Units' property units). The calls to **getframe** to build a movie should be consistent with each other and the **moviein** preallocation call.

The function **movie(***handle***,** *mov***,** *n***)** plays the movie *mov* in the object given by *handle* (or in the current figure if *handle* is omitted) *n* times (once if *n* is omitted). A negative *n* plays the movie back and forth each time. A vector *n* gives the repetitions in the first element and the frames to play, in order, in succeeding elements. An optional extra argument specifies the frames per second (defaulting to 12), limited by machine capabilities. An optional further extra argument specifies the location to play at, from the lower left corner, in the 'Units' property units.

4 The Session

4.1 Recording and Retrieving

To store and retrieve the state of your session, **save** *filename* and **load** *filename* write and read *filename***.mat**. The file matlab.mat will be used if *filename* is omitted. Default **.mat** files are not human-readable; see **help save** and **help load** for other uses, including ASCII formats and partial-state operations.

If you want to keep a record of your session, **diary** *filename* will store in *filename* (using the file "diary" if *filename* is omitted) everything that appears in the MATLAB xterm afterwards. (Graphics are not recorded.) **diary off** will turn off this recording until restored with **diary on**. (Or plain **diary** to toggle between these.) The diary is plain text, so your favorite text editor can be used on it.

4.2 Shell Escapes

MATLAB provides ways to deal with the shell from within MATLAB. **pwd** and **cd** give the current directory path, **cd** *directory* changes the current directory. **dir** and **ls** give a directory listing; **what** lists the M-files. **type** *filename* shows the contents of text file *filename* or *filename***.m**. **delete** *filename* will *remove* the file — it acts as the UNIX command "rm," not the UNIX "delete." To send an arbitrary command to your shell, prefix it with "!";

>> **!emacs &**

will start up an emacs in the background just as from an athena prompt. This is frequently helpful for managing M-files (see **4.3 M-files**). The command

[status result] = unix('*unix command string***')**

also executes an arbitrary command in the operating system, returning the command status (zero for success, nonzero for failure) and standard output.

4.3 M-files

An m-file is a plain text file (created with your favorite text editor) in a format MATLAB understands. They must end in **.m** or MATLAB won't accept them. Experience shows (but documentation is silent on) that m-file names may not have periods in them other than the **.m**, nor may they have any hyphens (apparently MATLAB wants to think they're minuses). Underscores are acceptable.

Any m-files in any directory in your *path* (see **4.3.3 Paths**) can be used, as can m-files in the current directory. Use **clash** to test if your m-file name will collide with an existing one.

M-files can reference each other and themselves recursively.

To see what an m-file says, **type** *file***.m**. This works for MATLAB's built-in m-files as well as for those you create. Use **what** *directoryname* or **help** *directoryname* to see what m-files (and mat-files and mex-files) are available; **what** by itself looks in the current directory. **exist('***thingy***')** will return zero if *thingy* is undefined, and nonzero otherwise; the exact value specifies the nature of *thingy* (see **help exist**). See **4.3.3 Paths** about where MATLAB will look for m-files.

In theory, MATLAB only loads in an m-file script or function the first time you refer to it; it caches the contents and refers to the contents thereafter. This would mean changes in the file would not affect MATLAB's actions. In practice, MATLAB (at least on Athena) does read the m-file each time, and uses the cache only if it can no longer find the m-file in question. To force MATLAB to reread the file, use **clear(***filename***)** to remove the cached meaning.

If all you want to do is supply some data, see **2.2 Building Matrices**.

4.3.1 Script and Data Files

A script is a list of commands that you might give to MATLAB directly. Typing *script* in MATLAB will execute, in order, the commands in the `script.m` file. As usual, the result of each will be printed out unless it ends with a semicolon; putting a semicolon at the end of the script invocation will *not* suppress these results.

To see the commands themselves as well as their results, use **echo on** and **echo off**.

Variables in a script file are automatically global; that is, if there is a variable of the same name in the workplace, it is what will be used and/or changed by the script.

4.3.2 Function Files

You may define your own functions in m-files named *functionname*.**m**.

```
function s = squares(high)
%SQUARES Make a vector of squares.
%   squares(high) returns (1:high)
temp = 1:high;
s = temp.*temp;
```

This (in **squares.m**) may then be used as any function;

>> **sq = squares(5)**
```
sq =
  1 4 9 16 25
```

You will note that **squares.m** puts semicolons after each line; if it did not, every time it was called it would echo the result of each line of computation, which is extremely annoying.

A more sophisticated version of such a function could be

```
function [y x] = powers(high, p)
%POWERS Table of some power of n versus n.
%    powers(high, p) returns [1:high] to the pth (elementwise)
%    powers(high) returns the square
if nargin < 2, p = 2; end;
y = [1:high].^p;
if nargout > 1, x = 1:high;
end
```

>> [y x] = **powers(4, 3)**
```
y =
  1 8 27 64
x =
  1 2 3 4
```
>> **y = powers(4, 3)**
```
y =
  1 8 27 64
```
>> [y x] = **powers(4)**
```
y =
  1 4 9 16
x =
  1 2 3 4
```

This function would be placed in the file **powers.m**; a function must always be in an m-file of the same name (**.m**).

In this example, more than one argument can be given, but the second is optional; the special variable **nargin** is consulted to see how many arguments were passed to the function, and the second is given a default value if omitted. There may also be more than one output value, and the special value **nargout** is checked to see how many assignments should be carried out. Use **nargchk** to confirm that **nargin** is in a valid specified range.

The character % at the beginning of a line indicates a comment. The set of commented lines after the **function** line is returned by **help** *functionname*, so it is important that they exist and document the function's behavior. The first line is what **lookfor** examines, so it should be the name and a concise summary.

A function need not return anything; many plotting functions do not. For these the first line is of the form **function** *functionname*(*arguments*).

Encountering the **return** command causes the function to terminate and sends control back to what invoked the function.

Variables within a function exist only within the function; changing them does not affect any variables of the same name in the workspace, nor does referencing them reference variables of the same name in the workspace (or in other functions). This is not always the desired behavior. If you wish some variable to be used in more than one area, **global** *variable1 variable2...* in each such area. Any assignment to a variable declared **global** in one area affects the value of that variable in all areas in which it is declared **global**. **isglobal** will test if a given variable is global in an area; **who**(**s**) **global** will list the global variables.

If you wish your function to be able to take a *function* as argument, you should take its name and apply **feval('***functionname***',** *arguments for function***)**, which will evaluate the function of that name with those arguments.

Useful operations for functions include **pause**, which waits *argument* seconds, or for the user to strike any key given no argument (**pause off** and **pause on** change whether **pause** should be ignored); **keyboard**, which passes control to the keyboard (using a K>>prompt as indication) until **RETURN** is entered; and **error** and **input** for which see **5.4 Text**.

As with scripts, **echo** can be used to have the text of a function file be echoed when the function is invoked. **echo** *functionname* **on** and **echo** *functionname* **off** will turn echoing on and off for a single file, while **echo** *functionname* will toggle. **echo on all** and **echo off all** affect echoing of all function files.

4.3.3 Paths

MATLAB cannot search the entire filesystem for m-files. The places MATLAB looks, and the order in which it searches them, are defined by the *path*. The command **path** will list them from first to last as a vector of strings. On Athena, the initial path will consist of a set of directories in the matlab locker, and your **~/matlab** directory if it exists; the directory where you run MATLAB does *not* get put in the path, but the current directory always gets searched for m-files.

Giving **path** an argument of one or more strings will set the path to that list. Usually one merely wants to extend the current path and therefore does **path(path, '/mit/foo')** or **path('/mit/foo', path)** to retain the current path but add /mit/foo to its end or beginning, respectively. The function **addpath('***new path***')** simplifies this operation; the function **rmpath** simplifies the reverse. These are both in the **contrib/names** toolbox, which is primarily for name collision testing; see **help contrib/names** or just **help names**.

To see the source of a given function or m-file, **which** gives the path followed to find its argument, while **where** gives all paths with a file matching the argument. If you are seeking a function to do some job, but whose name you do not know, use **lookfor** to search for a keyword.

When MATLAB wants to know what you mean by a name, it first looks for a variable of that name, then for a built-in function, then for an m-file in the current directory, and finally for an m-file in the path. Use **clash** to see what m-files will collide with a particular name.

4.4 Numerical Formats

MATLAB always keeps track of numbers to double precision regardless of the output format. To alter the display of numbers, use **format**. The argument **short** gives five digits (the default) and **long** fifteen digits; a second argument of **e** to either switches to floating-point.

Further, **format hex** gives hexadecimal, **format +** gives signs *only*, **format bank** uses dollars and cents, and **format rat** uses ratios of small integers as approximations.

Finally, **format compact** and **format loose** can be set independantly of all the other formatting, and supress or allow, respectively, extra linefeeds in output.

5 Further MATLAB

5.1 Debugging

MATLAB has special functions for debugging m-file functions (but not scripts). You establish breakpoints within the m-file; when these are encountered control passes to the keyboard as with the **keyboard** command, giving a **K>>** prompt. You may then examine (or change) the variables with normal MATLAB commands.

Breakpoints are established with **dbstop**, in four ways. **dbstop in** *m-file* puts a breakpoint at the first executable line. **dbstop at** *line* **in** *m-file* puts one at the specified line number. **dbstop if error** causes a stop (in any m-file function) when a runtime error occurs; execution may not be continued after a runtime error. Finally, **dbstop if naninf** and **dbstop if infnan** cause stops (in any m-file function) when a value is noted to be Inf (infinite) or Not-a-Number (NaN). In all of these, the keywords **at**, **in**, and **if** are optional.

The command **dbtype** is similar to **type**, but it gives the line number of each line of the m-file, for setting breakpoints. It also takes as an optional third argument a vector of lines to display (instead of all of them).

A list of all breakpoints is available with **dbstatus**. Use **dbclear** to remove breakpoints; **dbclear at** *line* **in** *m-file*; **dbclear all in** *m-file*; **dbclear in** *m-file* (removes a breakpoint set at the first executable line); **dbclear all**; **dbclear if error**; and **dbclear if naninf** and **dbclear if infnan** (clear the breakpoints set by the corresponding forms of **dbstop**). As with **dbstop**, the keywords **at**, **in**, and **if** are optional.

Having finished exercising keyboard control at a breakpoint, use **dbcont** to resume execution after the breakpoint. Note that when the debugger prints out a line, that is the *next* line to be executed; it has not yet been executed.

Instead of letting execution continue to the next breakpoint, you can use **dbstep** *number* to execute the next *number* lines (one if *number* is omitted). This lets you examine the state everywhere without making every line a breakpoint. If the m-file being debugged is about to itself call an m-file function, **dbstep in** steps the debugger into the latter line by line instead of executing it as a single line.

To track which m-files called which, use **dbstack**. To change the scope of the local workspace — that is, to have the variables you refer to at the **K>>** prompt be those of the m-file function or those of the function caller — user **dbdown** and **dbup** respectively.

The command **dbquit** aborts execution of the m-file (returning nothing) and leaves debugging mode.

5.2 Efficiency Testing, Times and Dates

Most of the time spent in an algorithm is consumed by floating-point operations; integer operations are relatively instantaneous. The special function **flops** tracks the number of floating-point operations performed. It begins at zero, and **flops(0)** will reset it. Therefore, **flops(0);** *operation(s)***;** **flops** gives a reasonable measure of the effort spent in *operation(s)*.

Alternatively, you may wish to time your operations. Use **tic** to start a stopwatch and **toc** to read it. So **tic;** *operation(s)***; toc** gives the number of seconds during *operation(s)*. However, since your computer was probably doing other things during that time, this is not a very reliable indication of the operations' efficiency; repeat it and see.

A better method of measuring time consumption is given by **cputime**, which is a non-resettable counter giving the CPU time used by MATLAB since it started:
cpu = cputime; *operation(s)***; disp(cputime - cpu)**.

If you simply wish to see the time, **clock** offers a wall clock giving a [*year month day hour minute second*] vector, and **etime** gives the time elapsed between two such vectors. A day-month-year string is returned by **date**.

5.3 Polynomials

A polynomial is represented by a vector of the coefficients, constant last. E.g. $2x^3 + 8x^2 - 7x$ becomes **[2 8 -7 0]**.

Polynomial multiplication is performed by **conv(p, q)** (which convolves the vectors). Division is available with **[q, r] = deconv(denom, numer)**, which sets the quotient q and remainder r such that $denom = q * numer + r$. Polynomials of equal degree can be added with $+$, but polynomials of unequal degree cannot be add directly; the lesser must be padded with zeros in front.

polyfit(x, y, n) finds the polynomial p in x of degree n that best fits $y = p(x)$ in the least-squares sense. **polyval(p, x)** evaluates the polynomial p at the point x, elementwise, if x is a matrix. These functions can also give error estimates; see the **help** entries. **polyvalm(p, m)** performs polynomial evaluation on a matrix (i.e. as a matrix, not elementwise). For some other curve-fitting capabilities see **curvefit**.

You can find the roots of a polynomial with **roots** and its derivative with **polyder**. The characteristic polynomial of a matrix M (i.e. det(M - lambda)) is given by **poly(M)**. See **help polyfun** for further polynomial and interpolation functions.

5.4 Text

Text strings are given in single quotes, e.g. **'a test string'**. A string is a row vector of characters.

```
>> s = 'a string';
>> s(5)
ans =
   r
```

Normal vector manipulation carries over into strings; resultant vectors will frequently be of numbers, especially if the manipulation wouldn't make sense as a string. This is done with normal ASCII converson. Both **real(vector)** and **abs(vector)** force ASCII decimal representation, while **setstr(vector)** forces character representation if possible. You can use **isstr** to test whether a vector is a string. (And **isletter** and **isspace** to test individual characters.)

```
>> s - 1
ans =
   96 31 114 115 113 104 109 102
>> [s ' which is longer than it was']
ans =
   a string which is longer than it was
```

The **help strfun** entry gives a nice summary of string functions.

A string (like anything else) can be displayed with **disp**; this can be useful in M-files. Prompted input is available with **input**. In the form **input('give me an answer: ')**, it prints its string argument without a carriage return and waits for the user to respond; the response is evaluated in the workspace and the result returned. In the form **input('give me an answer: ', 's')**, the response is not evaluated, but is returned as a literal string. Advanced prompted input is available with **menu**, and beyond that with **uimenu** and **uicontrol**.

To execute a string as a MATLAB expression, use **eval**. Given a single string argument, it attempts to evaluate it as if it had been entered in the current workspace. The results are returned in a vector. If **eval** is given a second string argument, and the execution of the first generates an error, it will attempt to evaluate the second.

The function **error** takes a single string argument, displays it, and exits from any m-file it occurs within (unless the argument is an empty string, in which case nothing happens).

The last error message generated is recalled by **lasterr**. It is reset to an empty string by **lasterr('')**. It is most often used when giving **eval** two arguments, as the second can examine its contents to see how the first failed.

There are a number of other useful string functions. Case conversion is done by **upper** and **lower**. Sequences of spaces are generated by **blanks** (the transpose then acts as a sequence of newlines), and trailing spaces are removed by **deblank**. Comparison is done with **strcmp**, searching with **findstr**, search and replace with **strrep**, and parsing by delimiters with **strtok**.

Conversions between strings and integer, real, and hex representations of numbers are available with **num2str**, **int2str**, **str2num**, **hex2num**, **hex2dec**, and **dec2hex**. Matrix conversations are handled by **str2mat** and **mat2str**. For more general conversions, see **help** for **sprintf** and **sscanf**; for use with files see **help** for **fprintf** and **fscanf**.

5.5 Sparse Matrices

MATLAB has two matrix storage modes. *Full* storage is the default and stores the value of each element; it is appropriate for matrices with many nonzero elements. *Sparse* storage is used only when you explicitly invoke it (though it propagates somewhat thereafter); it stores only the values of the nonzero elements (and their integer indices, of course). It is therefore appropriate for matrices with few nonzero elements, i.e. sparse matrices.

The advantages of sparse matrices are size and speed. A 5000 by 5000 matrix in full storage requires space for 25 million complex numbers even if only 50,000 are nonzero. The same 50,000-nonzero element matrix, in sparse storage, would store 50,000 complex numbers and 50,000 pairs of integer indices, less than .5% the space (less than one megabyte, instead of 200 meg). Similarly, solving $Ma = b$ where M is such a matrix would take most of a day in full storage, and less than half a minute in sparse. Obviously if you are dealing with large, sparse matrices, you should take advantage of these features.

Some sparsity functions are useful for other purposes, as sparsity is connected to graph and tree theory.

See **help sparse** for a description of sparse storage mode, and **help sparfun** for a list of the sparse matrix functions. The **MATLAB User's Guide** (see **1.4 Help**) has (as usual) a much fuller description of the nature, uses, and manipulation of sparse matrices.

5.6 Calculus

5.6.1 Differential Equations

For exact (symbolic) differentiations and solutions, see **5.8 Symbolic Math**.

The function **ode23** is used to solve ordinary differential equations with 2nd and 3rd order Runge-Kutta formulas. The variant **ode23p** also plots the results. The function **ode45** uses 4th and 5th order Runge-Kutta formulas and so is slower but more accurate; it otherwise acts as **ode23**. A simple demo is available with **odedemo**.

The basic format is **[t, y] = ode23**$('mysystem', t_0, t_f, y_0)$. Here t is the single independent variable and y is a vector (or scalar) of the dependent variables. The scalars t_0 and t_f specify the initial and final times; y_0 is a vector with the initial state of the dependent variables. The system of equations to be solved is in the m-file *mysystem*.**m**.

(An optional fifth argument can specify the tolerance, which defaults to 10^{-3}; an optional boolean sixth argument specifies whether to give status displays while integrating, defaulting to 0 (False).)

The results are a vector t of times and a matrix y in which the nth column gives the values of the nth dependent variable (y_n) at those positions. Thus **plot(t, y)** suffices for basic display. (See **ode23p** for automatic phase plane plots.)

The m-file defines a function with the same name as the system of equations, with one output (a vector of derivatives) and two inputs (a scalar time, and a vector holding the variables whose derivatives are returned). It is important to realize that y, $\dot{y}$, and $\ddot{y}$ are independent variables as far as MATLAB is concerned until you define their relationships. This means that the y vector will typically contain some intermediate derivatives of your basic dependent variables. The output vector is defined to be the derivative vector of the input vector; the process by which you specify this defines the system.

As an example, consider the system of equations

$$\begin{aligned} \ddot{x} + \dot{y} - 2\dot{x} &= 3 && (1) \\ \dot{y} &= 3x + 5 && (2) \\ & && (3) \end{aligned}$$

Here x and y are our basic dependent variables, and t is our independent variable (since we used time derivatives). Since the system is second-order in x, we must consider $\dot{x}$ as a necessary part of the state of the system and include it in our vectors of dependent variables. Thus we want a function that takes a scalar t and a vector holding (in whatever order) $x, y, \dot{x}$, and returns a vector holding *in the same order* $\dot{x}, \dot{y}, \ddot{x}$, i.e. the derivative of the input vector.

```
function der = mysystem(t, state)
% MYSYSTEM  to solve the system x'' + y' - 2x' = 3, y' = 3x + 5.
%       mysystem(t, [x(t) y(t) x'(t)])
%       returns     [x'(t) y'(t) x''(t)]
%       Suitable for use with ode23, ode45.
```

```
% First assign the input values to variables with names that
% make what they are easier to remember.
x = state(1);
y = state(2);
xdot = state(3);

% Now calculate the output values.
% The first, the derivative of x, is trivial since we require
% it as an input value.  We'll put a line to do it anyway as a reminder.
xdot = xdot;

% We can get the second derivative of x in terms of x' and y' from
% our first equation, but we need y' first.  We'll use the second for that.
ydot = 3*x + 5;

% Now we can get the second derivative of x.
xdoubledot = 3 - 2*xdot - ydot;

% Construct the vector of derivatives in the same order as
% the inputs.  We took them as x, y, x', so we must return x', y', x''.
der = [xdot, ydot, xdoubledot];
```

With this function in the file **mysystem.m**, we can solve the system over any stretch of time with any initial values:

```
>> t0 = 0;
>> tf = 10;
>> x0 = 0;
>> y0 = 1;
>> xdot0 = -1;
>> state0 = [x0 y0 xdot0];
>> [t state] = ode23('mysystem', t0, tf, state0);
>> x = state(:, 1); y = state(:, 2); xdot = state(:, 3);
>> figure(1); plot(t, x); title('x(t)')
>> figure(2); plot(t, y); title('y(t)')
>> figure(3); plot(t, xdot); title('xdot(t)')
>> figure(4); plot(x, y); title('phase plot')
```

While we have considered the independent variable to be time, it is obvious no such interpretation is required.

MATLAB does not have built-in functionality to solve systems with multiple independent variables.

5.6.2 Numerical Integration

For exact (symbolic) integrations and solutions, see **5.8 Symbolic Math**.

There are three functions available for numerical integration. The functions **quad** and **quad8** use adaptive, recursive rules, the low order Simpson's and the higher order Newton Cotes 8 panel respectively; other than the rule used, they act identically. The basic format is **quad('***function***',** *start*, *finish*) to get the integral of *function* from *start* to *finish*. The *function* must take a single vector as input (of the length of *start* and *finish*) and return a single vector of output.

If the integration requires too much recursion (e.g. if the integral is singular), the value **Inf** is returned.

An optional fourth argument may be specified to give the tolerance (default 10^{-3}. An optional boolean fifth argument may then be given to specify whether the evalutions should be traced in a point plot automatically (False if 0, True otherwise). An empty matrix may be passed for either of these to keep the defaults.

If *function* takes more than one argument, the others may be specified as constant (for the integration) parameters by **quad('***function***',** *start*, *finish*, *tolerance*, *trace*, *first parameter*, *second parameter...*). This integrates *function*(**x,** *first parameter*, *second parameter...*) over $x = start$ to $finish$.

Alternatively, **trapz** may be used for trapezoidal numerical integration; the format is **trapz(x, y)**. Here x is a vector and y is a matrix of any number of columns, but as many rows as the length of x. Each column of y is considered to be a function over x and is integrated with the trapezoidal approximation. The result is a vector of the column integrations. If x is omitted, unit spacing between points is assumed.

5.7 Optimizing

The function **fmin('***function***',** *start*, *finish*) will attempt to return the minimum of $function$ (a scalar function of a scalar argument) in the interval [$start$, $finish$]. Of course, no numeric method can guarentee a minimum for a function that varies too rapidly.

An optional fourth argument may be given; this is a vector of options (see **help foptions**). The relevant options for **fmin** are the first, second, and fourteenth elements in the vector. If and only if element 1 is nonzero, the intermediate steps are displayed (default is 0). Element 2 gives the termination tolerance in the function argument (default 10^{-4}). Element 14 gives the maximum number of steps (default 500). The other elements of the options vector are ignored.

If *function* takes more than one argument, the others may be specified as constant parameters by **fmin('***function***',** *start*, *finish*, *options*, *first parameter*, *second parameter...*). This minimizes *function*(**x,** *first parameter*, *second parameter...*) over $x \in [start, finish]$.

A function of more than one variable may be similarly minimized with **fmins**, which uses a simplex search. The basic form is **fmins('***function***',** v**)**, which looks for a local minimum near the vector v. The function must still return a scalar.

A vector of options may be passed to **fmins** exactly as to **fmin**, with the same meanings and defaults. Parameters may also be passed to **fmins** as to **fmin**, except that there is a mandatory empty matrix argument between the options vector and the first parameter.

To maximize a function, minimize its negative.

The function **fzero('***function***', x)** seeks a zero of *function* with x as a starting point. The function must take and return a scalar; there is no vector-argument form of **fzero**. The value of the function at the returned point may not be exactly zero, but the function will change sign near that point. An optional third argument may be given to specify the convergence tolerance; an optional fourth boolean argument may be given to cause (if nonzero) intermediate printing.

See **help optim** for the contents of the Optimization Toolbox.

5.8 Symbolic Math

MATLAB obtains some basic symbolic math capabilities by using the Maple kernel. If you want to do a lot of symbolic math, you probably want to use Maple directly.

An expression is a text string (see **5.4 Text**). Isolated lower case characters of the alphabet (other than the imaginary constants i and j) are taken to be variables. Function names and operator symbols are usually interpreted correctly.

For many purposes, only one variable in an expression can be used as a free variable — for instance, when doing a single differentiation or integration. This is referred to as "the" symbolic variable of the expression. Use **symvar** on an expression to see what variable will be taken as the free variable by default; you will usually be able to override the default when using a symbolic function.

A symbolic matrix is a matrix of text characters, intepreted row by row. Each row begins with '[' and ends with ']'; the individual elements are substrings separated by commas.

Rather than give exhaustive descriptions here, we will list most of the symbolic functions with short descriptions based on their **help** entries, which you may see for full details:

sym	Create, access, or modify a symbolic matrix
symop	Symbolic operations; general arithmetic strings
symadd	Symbolic addition
symsub	Symbolic subtraction
symmul	Symbolic multiplication
symdiv	Symbolic division
sympow	Symbolic exponentation
inverse	Symbolic matrix inverse
diff	Symbolic differentiation
int	Symbolic integration
factor	Factor elements of symbolic matrix
simplify	Simplify elements of symbolic matrix
simple	Search for simplest form of a symbolic expression
expand	Expand elements of symbolic matrix
collect	Collect powers of free variables in symbolic expression
taylor	Taylor series expansion in symbolic variable
solve	Symbolic solution of algebraic equations
dsolve	Symbolic solution of ordinary differential equations

A special constant, **Digits**, determines the numeric accuracy of symbolic computations. Use **digits** to find the current accuracy and **digits(**d**)** to set the accuracy. The function **vpa**, for "variable precision arithmetic," numerically evaluates (to *Digits* precision) a symbolic expression.

A symbolic expression with a single free variable can be plotted with **ezplot(***function***)**.

5.9 SIMULINK

SIMULINK is a special interactive toolbox for simulating dynamic systems in block diagrams with a graphical (mouse-driven) interface. It can handle nonlinear systems of many variables and rates, in discrete or continuous time. The results can be further analyzed in the MATLAB workspace.

SIMULINK is available on DECstations, IBM RS/6000s, and Suns. It cannot be accessed from within a regular MATLAB session, but must be started by itself.

```
athena% add matlab
athena% simulink &
```

This will generate a MATLAB window named SIMULINK. All MATLAB functionality is present as are SIMULINK functions, for which full **help** information should be available. Use **simdemo** for demonstrations.

Copies of the **SIMULINK User's Guide** are available for reference in the Athena Consulting Office (11-115), and in Barker and Hayden Libraries.

5.10 Other Features

Type **help** by itself to list categories of MATLAB functions.

Among those that we have not gone into in detail are

matlab/datafun	Data analysis and Fourier transform functions.
matlab/sounds	Sound processing functions.
matlab/iofun	Low-level file I/O functions.
toolbox/uitools	User Interface Utilities.
toolbox/control	Control System Toolbox.
toolbox/ident	System Identification Toolbox.
toolbox/local	Local function library.
mutools/commands	Mu-Analysis and Synthesis Toolbox.: Commands directory
mutools/subs	Mu-Analysis and Synthesis Toolbox – Supplement
toolbox/optim	Optimization Toolbox.
toolbox/robust	Robust Control Toolbox.
toolbox/signal	Signal Processing Toolbox.
toolbox/splines	Spline Toolbox.
toolbox/stats	Statistics Toolbox.
toolbox/images	Image Processing Toolbox.
toolbox/symbolic	Symbolic Math Toolbox.
nnet/nnet	Neural Network Toolbox.
nnet/nndemos	Neural Network Demonstrations and Applications.
toolbox/pde	Partial Differential Equation Toolbox.