Little Bits of MATLAB

James H. McClellan
School of ECE
Georgia Tech

27-March-1997

ECE-2025
ECE-4270

• The default is version 3.5 of MATLAB; version 4.1 changes are indicated when necessary.

Contents

(Blank page)

Chapter 1

Introduction

The MATLAB software environment can take you by surprise. It is extremely easy to learn, and is best tackled by doing rather than by just reading. Try new things and see the results; more than likely, you can be doing very sophisticated operations within a matter of hours.

If there is one watchword, it would be *vector*—you must create expressions and programs that state the operation in a vector form. Previous experience in a high level language, such as FORTRAN or C, can actually be detrimental, because such languages force programming at a rather low level. Instead, it is best to think in the "language" of linear algebra. Soon, the syntax will become second nature, and MATLAB will become an essential part of your mathematical and signal processing experimentation.

The emphasis of this book is on the concepts involved in using MATLAB efficiently. Therefore, programming techniques will be stressed after a quick introduction to the basics of the MATLAB language.

1.1 New Features in Version 4

At the present time, version 4.1 os MATLAB is available for most platforms. The Student Version will be updated to version 4 very soon. Therefore, some material in this book needs updating. Since the book was prepared primarily for students who might be using the Student Version in their courses, the presentation reflects version 3.5. In all cases, an effort has been made to point out differences that are significant for those users who are presently using version 4.1. Fortunately the basics of MATLAB are identical between version 3.5 and version 4.1, so the reader can be confident that any MATLAB skills learned in the context of version 3.5 are still applicable to version 4.1. The main differences lie in the area of plotting, since version 4 has greatly enhanced graphics capabilities.

Comment

This presentation reflects one person's usage and experience, so the emphasis may be skewed. After five years of answering questions and teaching courses with considerable MATLAB usage, certain patterns have emerged. One theme that seems significant but eludes most students is the dictum to "vectorize." So this book is dedicated to that one goal— NO LOOPS!!!

Chapter 2

Essentials of Operating MATLAB

This chapter presents the basic operation of MATLAB. In reality, the best way to learn a software environment like MATLAB is simply to use it, and constantly try new things. An advantage to MATLAB is that the start-up time is minimal; quite likely you can be doing very sophisticated operations within a matter of hours.

If there is one watchword, it would be *vector*—you must create expressions and programs that state the operation in a vector form. Previous experience in a high level language, such as FORTRAN or C, can actually be detrimental, because such languages force programming at a rather low level. Instead, it is best to think in the "language" of matrices and linear algebra. Soon, the syntax will become second nature, and MATLAB will become an essential part of your mathematical and signal processing experimentation.

2.1 Overview of Basic Capabilities

MATLAB is an interactive mathematics program for performing scientific and engineering calculations. It is an excellent tool for doing the matrix manipulations commonly found in linear algebra. Four features stand out:

1. All computations are carried out in double-precision arithmetic to guarantee high accuracy. Variables can be *complex*-valued, if necessary.

2. There is a huge set of mathematical functions, e.g., linear equation solvers, eigenvalues, singular values, etc.

3. There is a rich set of *plotting* capabilities that are extremely useful when viewing vectors as signals. A variety of 2-D plotting functions also exist.

4. MATLAB is also a *programming* environment, so the user can extend its functional capabilities by writing new modules. One set, authored at Georgia Tech, contains a number of popular DSP signal modeling techniques, and DSP plotting formats. (On the HP worstations, these are found in a sub-directory called `gatech`; in the Vectra Lab, they can be found in the GT-MATLAB download area.)

Thus MATLAB provides a versatile interactive computing environment for mathematics and engineering. Applications such as DSP (digital signal processing) and control systems are two primary ones that will be treated in this book.

2.2 Quick Startup

This section is for those who only want to know the bare minimum before starting to run the software. Since the program contains on-line help and several demonstrations, it is possible to learn the syntax "on the fly" by just trying things that seem natural in a mathematical sense.

Some of the information on starting MATLAB may be site-dependent, so please consult your site-dependent source to find out which files must be loaded before you can begin. Assuming that all the software is present, running MATLAB is simple: you type `MATLAB` and after the program starts, it will display the prompt `>>`. All commands to MATLAB are typed after the prompt. When you are ready to quit, type `quit` or `exit` to return to the operating system. If it is necessary to abort a MATLAB function without quitting, try ctl-C.

Command Line Recall

MATLAB has a command line recall and editing facility. This is extremely valuable when similar commands must be executed over and over. The implementation of this feature has varied somewhat on different machines. With version 4 and the later revisions of version 3.5, the arrow keys are used. Furthermore, "emacs" commands are also aupported. On a Macintosh the cut and paste facility can also be used, and was the only mechanism in early revs of version 3.5.

Shell Escape

If a command is preceded with an exclamation point `!`, then it is taken to be a shell command to the operating system. For example, to invoke an editor.

Some of the often used commands are given in the following two tables.

General Commands	
`ˆC`	control-C aborts a function
`clear`	clear workspace of all variables
`demo`	run demos (menu driven)
`exit`	terminate (same as quit)
`help`	help on a specific topic
`length`	vector length (row or column)
`quit`	terminate (same as exit)
`size`	matrix row and column dimensions
`type` *filename*	list out an M-file
`who`	list names of variables in workspace
`whos`	list names and sizes of all variables

Special Characters: Punctuation	
`=`	assignment
`[ ]`	used to form vectors and matrices
	from scalars and other matrices
`( )`	arithmetic expression precedence grouping
`.`	decimal point
`..`	continue statement to next line
`,`	separator for multiple commands on one line
`,`	separator for row elements, subscripts & function args
`;`	SUPPRESS PRINTING of output when command ends with `;`
`;`	end rows of a matrix
`:`	subscripting, vector generation
`%`	comment delimiter, rest of line is comment
`!`	execute operating system command

2.2.1 Demonstrations

MATLAB has a built-in set of demonstration programs. You may wish to try some of them immediately, or you may wish to read the following summary of MATLAB features and then try the demos. In either case, simply type `demo` at the MATLAB prompt, and follow the menu.

```
>> demo

       ------- MATLAB Demonstrations -------

       1)  Introduction to basic MATLAB commands.
       2)  Some of the graphics capabilities.
       3)  Predict 1990 population.
       4)  A square wave is the sum of odd harmonics.
       5)  The convolution theorem.
       6)  Reduced row echelon form and eigenvalue movies.
       7)  Pretty 3-d mesh surfaces.
       8)  Ordinary differential equations.
       9)  Interesting plots!
       10) Benchmarks.
       11) Spectral analysis using FFTs.
       12) Design recursive digital filters.

           0)  Quit.

  Select a demo number:
```

2.2.2 Help

MATLAB has a very effective on-line help facility. It is absolutely essential that the user become accustomed to using the on-line help. To get a list of all MATLAB features simply type

```
>> help
```

The result will be several screenfuls, listing all features for which on-line help is available. A copy of the list is given in the Chapter 4 of this write-up.

A related operator is `type` which will list out the entire contents of an M-file. This is useful when you need to see which algorithm was used for a particular M-file, or when you are trying to learn the programming style of experts who have coded the primary functions inside MATLAB. You cannot `type` out the contents of a built-in function., such as `fft` because it is compiled from C code, not interpreted from an ASCII M-file.

In version 4, there is a searching capability called `lookfor`. This allows the user to search the entire database of MATLAB help to find all entries that contain a given string. For example, if you want all functions that refer to the FFT, enter `lookfor fft`.

$\Rightarrow$ **Note:** In Chapter 4 is a printout of the help message for many commonly used functions. To obtain a listing of a help message on a specific topic, such as the FFT, type

```
>> help fft
```

The result will be a short description of how to use the command.

```
>>help fft

FFT    FFT(X) is the discrete Fourier transform of vector X.  If the
   length of X is a power of two, a fast radix-2 fast-Fourier
   transform algorithm is used.  If the length of X is not a
   power of two, a slower non-power-of-two algorithm is employed.
   FFT(X,N) is the N-point FFT, padded with zeros if X has less
   than N points and truncated if it has more.
   If X is a matrix, the FFT operation is applied to each column.
   See also IFFT, FFT2, and IFFT2.
```

It is also possible to ask for help on language constructs such as `if` statements, or on syntax issues such as the colon operator `:`.

```
 >> help if

IF  Conditionally execute statements. The simple form is:
            IF variable, statements END
    The statements are executed if the variable has all
    non-zero elements. The variable is usually the result of
    expr rop expr where rop is ==, <, >, <=, >=, or ~=.
    For example:

             IF I == J,
                A(I,J) = 2;
             ELSEIF ABS(I-J) == 1,
                A(I,J) = -1;
             ELSE
                A(I,J) = 0;
             END
 >> %---------------------------------------------------

 >> help :

:   Colon. Used in subscripts, FOR iterations and possibly
    elsewhere.
    J:K  is the same as [J, J+1, ..., K]
    J:K  is empty if J > K.
    J:I:K  is the same as [J, J+I, J+2I, ..., K]
    J:I:K  is empty if I > 0 and J > K or if I < 0 and J < K.
    The colon notation can be used to pick out selected rows,
    columns and elements of vectors and matrices.
    A(:) is all the elements of A, regarded as a single
    column. On the left side of an assignment statement, A(:)
    fills A, preserving its shape from before.
    A(:,J) is the J-th column of A
    A(J:K) is A(J),A(J+1),...,A(K)
    A(:,J:K) is A(:,J),A(:,J+1),...,A(:,K) and so on.
    For the use of the colon in the FOR statement, See FOR.
```

Be advised that the help facility uses an environment variable called `MATLABPATH` to search through a set of directories; your present working directory should be on this path, so that files you create will be incorporated into the MATLAB help system.

2.3 Three Windows

1. Command
2. Graphics
3. Edit
4. Help (not really a separate window)

2.3.1 Differences in PC vs. UNIX vs. Mac

2.4 Data, Variables, and Expressions

The name MATLAB stands for *matrix laboratory*, and this is appropriate because the one and only data object in MATLAB is a rectangular numerical matrix with real or complex elements. Indeed, scalars (single numbers) are actually 1×1 matrices; and even character strings are treated as vectors of characters.

Most MATLAB statements are assignments of the form:

variable = expression

The expression is usually a function that takes inputs and returns outputs. There are no dimension statements or type declarations in MATLAB. Storage for data and variables is allocated automatically in response to MATLAB statements. There is, of course, limited storage on any computer. (About 300 Kbytes are available for storing data on a PC; much more on other systems.)

MATLAB is an *interpreted* language, i.e., expressions typed by the user are immediately evaluated by the MATLAB system, and the results are displayed to the user. The response to the user can be suppressed by placing a semicolon at the end of the MATLAB input statement.

2.4.1 Constructing Matrices

Matrices can be entered explicitly by typing a list of numbers, surrounded by square brackets `[ ]`. The elements within a row must be separated by spaces or commas, and each row terminated with a semicolon `;` or a carriage return. For example, the statement

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

results in the output

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

Since no array name was given, the result is returned in the generic variable `ans`, which always holds the result of the last unassigned MATLAB operation.

The equals sign `=` is used to assign names to scalars and matrices resulting from MATLAB statements. Thus typing `A=[1 4 7; 2 5 8; 3 6 9]` assigns the symbol `A` to the matrix on the right-hand side of the `=`. The matrix is saved in MATLAB's workspace and is subsequently referred to by the symbol `A`. The variable can be removed the workspace by doing `clear A`.

Since the end of a row can also be indicated by a carriage return, another way to enter the matrix `A` is

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

If a row of data is too long to fit on one line, when you reach the end of a line, you can type two periods `..` followed by carriage return and continue entering data on the next line.

The echo in MATLAB can be annoying, especially for very large matrices. A semicolon used at the end of a MATLAB statement will suppress typing of the result. Thus,

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

creates the same matrix as above, but does not type it out.

Sometimes we may wish to access a single matrix entry. This is done by explicitly referring to the element by its row and column index. Indexing starts at 1, so `A(3,2)` is the element in the 3rd row and 2nd column. For example, to assign the value 2.718×10^{-5} to the second element of the third row of the matrix `A`, we would type `A(3,2)=2.718e-05;`. Note the use of exponential notation (also found in FORTRAN and C) to specify this last constant.

2.4.2 The Data Workspace

At any time in a MATLAB session, you may obtain a summary of the entire workspace by typing `whos`. The resulting list gives the variable names and their dimensions., together with the amount of free memory.

```
 >> whos
              Name       Size       Total    Complex

                  A      3 by 3          9       No
                  j      1 by 1          2      Yes
                 pi      1 by 1          1       No
                  x      4 by 1          4       No
                  y      1 by 4          4       No

Grand total is (20 * 8) = 160 bytes,

leaving 314256 bytes of memory free.
```

To find out the size of a specific variable, type `size(A)`, which will return a row vector with two entries, the first being the number of rows in `A`, the second the number of columns. Once defined, a variable remains in the workspace until it is explicitly removed by the `clear` command, e.g.,

```
   >> clear A
```

Saving the Workspace

All the variables in the workspace can be saved to a disk file using the `save` command. The resulting file can be reloaded with the `load` command. For example, the command `save project` creates a file called `project.mat` that can be read by the command `load project`. The file created by `save` is a binary file, but it contains a header that allows it to be read by MATLAB on any machine—format conversions are made, if necessary.

Workspace Commands	
`clear`	remove all variables from the workspace
`load`	load variable(s) from `.mat` file or ASCII file
`pack`	compact memory (garbage collection?) equivalent to `save, clear, load`
`save`	save variable(s) to a `.mat` file
`who`	list names of variables in workspace
`whos`	list names and sizes of all variables

Garbage Collection

Any system such as MATLAB that maintains an environment with variables continually being created and destroyed must have a form of "garbage collection" to remove dead (or unused) space. Unfortunately, MATLAB has no automatic garbage collection mechanism. The function `clear` allows the user to manage his workspace and do his own house cleaning. Even that is not enough, since other temporary arrays might be created and destroyed whenever M-files are run. In place of garbage collection, there is a MATLAB function called `pack` which saves all the variables in the entire workspace, clears the workspace, and then loads the saved variables. This is time-consuming, but it is the best way to get some room to work if memory limits start to hinder your progress.

2.4.3 Row & Column Vectors

As is common terminology, a matrix with only one column is called a *column vector*. Column vectors can be entered as follows:

```
>> x=[1
2
3
4]
```

or

```
>> x = [1;2;3;4]
```

both of which result in the output

```
x =
     1
     2
     3
     4
```

As before, the vector `x` is saved by name for future use. A *row vector* is a matrix with only one row. We can obtain a row vector by entering it explicitly as described above, or by “transposing” a column vector, `y=x'`. For the vector `x` above, the transposed output will be

```
 y =
      1     2     3     4
```

Notice that we use two symbols, a period followed by a prime, `y=x.'`, to do the transpose, because the “prime” operator by itself takes the complex-conjugate transpose of a matrix; the “period” suppresses the conjugate operation.

The `i`-th element of a row vector `y` is, strictly speaking, `y(1,i)`, but we can simply refer to it as `y(i)`. Similarly, the `i`-th element of a column vector `x` can be picked with either `x(i,1)` or `x(i)`.

The length of a (row or column) vector can be obtained with the function `length(x)` which is short hand for `max(size(x))`.

2.4.4 Formatting Numbers

One last comment on numbers concerns the format for display of numbers. For example, the value of the constant π is obtained by typing the name `pi` in MATLAB. This results in

```
 >> pi
 ans =
     3.1416
```

Note that only 5 significant digits are displayed, but since MATLAB actually does double precision floating point arithmetic, we would expect that many more digits are actually used to represent each number. To see numbers with full precision, use the `long` format:

```
>> format long
>> pi
ans =
   3.14159265358979
```

After changing to `format long` in a MATLAB session, all numbers would be displayed with approximately 15 significant digits. To go back to the short format type `format short`. Other output formats are available, consult `help format` for more information. For example, `format compact` will make the output to the terminal single-spaced; and `format bank` is useful for balancing your checkbook.

2.4.5 Complex Numbers in Matrices

MATLAB handles complex numbers as easily as real numbers, but in order to enter them we need to create the basic imaginary number, i.e., $\sqrt{-1}$. For example, if you like to call it `J` you can make the assignment:

```
>> J=sqrt(-1)
J =
        0 + 1.0000i
```

Note that MATLAB always uses the symbol `i` when printing the imaginary base no matter what you call it, so to prevent confusion you may better off to go along with MATLAB and call it `i`. Furthermore, newer versions of MATLAB start up with the symbols, `i` and `j` already defined as $\sqrt{-1}$. Since MATLAB is case sensitive the symbols `J` and `j` represent different variables.

Now we can create complex numbers with statements like:

```
>> z = 3+4*J
z =
   3.0000 + 4.0000i
>> r = 0.9; theta = pi/3;
>> w = r*exp(J*theta)
w =
   0.4500 + 0.7794i
```

To enter a complex matrix, you might type each element as a complex expression, or you can add two real matrices as in the following:

```
>> E = [1 2; 3 4] + J*[5 6; 7 8]
E =
   1.0000 + 5.0000i   2.0000 + 6.0000i
   3.0000 + 7.0000i   4.0000 + 8.0000i
```

In version 4, complex constants are allowed, so we could also define `E` via `E = [ 1+5i 2+6i` 3+7i 4+8i];

For complex-valued matrices, some operators are sensitive to the imaginary part. For example, when matrix entries are complex, the "prime" operator gives the conjugate transpose; thus `E′` results in

```
>> E′
ans =
   1.0000 - 5.0000i   3.0000 - 7.0000i
   2.0000 - 6.0000i   4.0000 - 8.0000i
>> E.′
ans =
   1.0000 + 5.0000i   3.0000 + 7.0000i
   2.0000 + 6.0000i   4.0000 + 8.0000i
```

The transpose of a complex matrix (sans conjugate) is obtained by typing `E.′`; the "period" before the "prime" suppresses the conjugate operation.

2.4.6 The Colon : Operator

The colon symbol `:` represents one of the most useful operators in MATLAB. It is used for (1) creating vectors and matrices, (2) specifying submatrices as subscript ranges, and (3) in `for` iterations.

To learn more about the use of `:`, type `help :` and see section 4.4(??).

Creating Regular Matrices

The colon can be used to create vectors or matrices with regularly spaced elements. For example, it is often necessary to have a vector of integers running from 1 to N. When used in this way,

`j:k`	is the same as	`[j,j+1,j+2,...,k]`	if `j<k`
`j:i:k`	is the same as	`[j,j+i,j+2i,...,k]`	if `i>0` and `k>j`, or if `i<0 and k<j`

NOTE: the empty matrix `[ ]` is created if `i,j,` and `k` do not satisfy the correct ordering relations. The following example shows a 2×5 matrix created from regular submatrices. Notice that the regular submatrix generated with the colon operator is always a row vector.

```
>> D=[(0:.1:.4);(4:-1:0)]
D =
        0    0.1000    0.2000    0.3000    0.4000
   4.0000    3.0000    2.0000    1.0000         0
```

Selecting Subscripts

The colon is also useful for picking out selected rows, columns, and elements of vectors and matrices. When used by itself, as in `D(:,j)`, we get the entire `j`-th column of the matrix `D`; likewise, `D(i,:)` is its `i`-th row. In order to see the values of the third through fifth columns in all the rows of the matrix `D` above simply type:

```
>> D(:,3:5)
ans =
    0.2000    0.3000    0.4000
    2.0000    1.0000         0
```

Running Loops

In order to designate the range of a `for` loop, the colon operator is used to generate a regular vector of “indices” needed in the loop. In the most common case, we use `1:n`, so we can step through the integers `1, 2, ... n`. Thus, the usual loop is

```
   for i = 1:n
      some operation depending on i;
   end
```

Storage Order of Matrix Elements

For the most part, we are not concerned with the internal storage order of the matrices in MATLAB. But there is one common situation where the colon operator is used and the storage order makes a difference. When a matrix `A` is written as `A(:)` the meaning is to take all the elements of `A` and put them in one large column vector. Since the storage of the matrix `A` is column dominant, the columns are stacked on top of one another when creating `A(:)`.

```
>> A = [3:5;0:2]
A =
     3     4     5
     0     1     2
>> A(:)    %---illustrate the storage order for matrices
ans =
     3
     0
     4
     1
     5
     2
```

2.4.7 Special Constants

MATLAB contains a number of predefined constants that come in handy. These include not only π, ∞ and $\sqrt{-1}$, but also the floating point tolerance `eps`, and the IEEE value `NaN` which stands for “not a number”—the result of a illegitimate floating-point computation.

```
>> [i j]    %--assumed at start up
ans =
        0 + 1.0000i        0 + 1.0000i
>> format long
>> [pi exp(1)]
ans =
   3.14159265358979   2.71828182845905

>> eps
eps =
     2.220446049250313e-16

>> [0/0 1/0]
Warning: Divide by zero
ans =
   NaN     Inf
```

Special Values	
`ans`	result of last evaluation (when not assigned)
`clock`	wall clock time as `[year month day hour minute seconds]`
`computer`	type of computer
`date`	string with the date in `dd-mmm-yy` format
`eps`	floating-point precision
`i,j`	$\sqrt{-1}$ (initially defined at startup)
`Inf`	∞
`NaN`	IEEE standard representation: "not-a-number"
`pi`	π

2.4.8 Text Strings

Text strings in MATLAB are just arrays of characters, but they are identified internally as having the type "string". The characters can be forced to take on their numerical ASCII values by applying `abs` to the string. Conversely, the string nature can be restored with the function `setstr`. String operations must be done via matrix operations. Thus, string concatenation of `'abcd'` and `'qwerty'` is done by joining together the two sub-strings into a larger array: `[ 'asdf' 'qwertry' ]`.

A string can be evaluated with `eval`, the resulting being to execute the command specified by the string. Thus `eval('help fft')` will ask for help on the FFT function. The related function `feval` will invoke the function given in a string.

Some functions exist for converting numbers to strings: `num2str` and `int2str` Printing formatted text strings can present some problems if the user is not familiar with the C language. The functions `sprintf` and `fprintf` are borrowed from C, but the quoting of arguments is a little different—single quotes are used instead of double quotes. Also the number of formats is restricted.

Text and Strings	
`abs`	convert string to ASCII values
`eval`	evaluate text macro
`feval`	evaluate function given by string
`fprintf`	formatted i/o as in C
`hex2num`	convert hex string to number
`int2str`	convert integer to string
`isstr`	detect string variables
`num2str`	convert number to string
`setstr`	set flag indicating matrix is a string
`strcmp`	compare string variables
`sprintf`	convert number to string à là C

2.5 Plotting Graphs

A most important feature of MATLAB is its plotting capabilities. Graphical output of signals is a necessary part of signal processing. MATLAB has built-in functions for making the following types of plots: linear x–y, loglog, semilog, polar, mesh, contour, and bar charts. A separate window (or screen) is devoted to graphics display. Once the graph is on the screen you can add labels to it with `title`, `xlabel` for the x-axis, `ylabel` for the y-axis, and `text` for annotation. A grid can also be overlayed (`grid`). There are a variety of other commands for controlling the screen, scaling, etc. All of these plot commands are documented by the `help` facility. If you type `help plot`, you will get a description of the basic x–y plotting function and a list of related plotting functions. The demos also contain some interesting plots; type `demo` and follow the menu.

Graph Annotation	
`grid`	overlay a grid on a plot
`ginput`	graphics input: get `(x,y)` position from mouse
`gtext`	mouse-positioned text
`text`	annotation, positioned at `(x,y)`
`title`	make title from text string
`xlabel`	label the x-axis
`ylabel`	label the y-axis

2.5.1 1-D Plotting

The simplest use of `plot` is

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

If `y` is a *real* vector, this command will simply plot the sequence of elements of `y` using the integers `1:length(y)` to label the horizontal axis. The data points will be connected by straight (solid) lines. If `y` is a real matrix, each column will be drawn as a separate curve on the same graph, but with a different line type. If `y` is a *complex* vector or matrix, then the plot will be the imaginary part (as the y-axis) versus the real part (as the x-axis). In effect, this is nearly a polar plot.

Polar Coordinates

In some applications of complex numbers, polar plots are necessary, e.g., to plot the roots of a z-transform polynomial $H(z)$

```
     Hroo = roots(H);
     polar( angle(Hroo), abs(Hroo), 'x' ), grid
```

Each root position will be marked with an `'x'`. Note that using `plot` on a complex vector will generate an x–y plot that is similar to a polar plot, but the `grid` function will not draw a polar grid as its does for the `polar` plot.

1-D Plotting Functions	
`bar`	bar graph (for histogram)
`errorbar`	add error bars to plot
`loglog`	log y vs. log x plot
`polar`	polar coordinates
`plot`	x–y plot, lines or points (`x, o, +, *, .`)
`semilogx`	linear y vs. log x plot
`semilogy`	log y vs. linear x plot
`stairs`	bar graph without internal lines

2.5.2 Horizontal Scaling and Labeling

An extremely important maneuver is to control the labeling of the horizontal axis. The following example shows how to create a horizontal “time” axis with labels running from $t = -5$ to $t = +5$.

```
 >> t = -5.0:0.1:5.0;  %--- creates 101 regularly-spaced points
 >> ssss = sin(pi*t);
 >> plot( t, sss )
```

This sequence of commands creates a vector (`sss`) of 101 samples of the signal $s(t) = \sin(\pi t)$ at a sampling interval $\Delta t = 0.1$. The vector `t`, used as the first argument to `plot`, will determine the horizontal axis scaling. Try it and see how it looks. If a third argument were given as `plot(t,s,'x')`, the sample points would be marked with the symbol `x`, and not connected by lines.

Multiple Plots Per Graph

You can also put multiple plots on the same graph by using a matrix, by including more x–y vector pairs; or by using the `hold` command followed by more `plot` commands. Thus the following three sequences of commands will generate a plot containing three curves:

```
y = rand(10,3);
y1 = y(:,1);  y2 = y(:,2);  y3 = y(:,3);
x = 1:10;
%---------
   plot(y)  %---plot 3 columns
%---------
   plot(x, y1, x, y2, x, y3 );   %---plot 3 x-y pairs
```

```
%---------
   plot(y1)
   hold on      %---freeze the graphics window
   plot(y2)
   plot(y3)
   hold off     %--- un-freeze
```

2.5.3 Two-Dimensional Plotting

Mesh plots and contour plots are available for displaying 2-D data held in arrays. Since MATLAB assumes that the fundamental 2-D data item is a matrix, the convention for viewing a 2-D array is that the (1,1) point is in the upper left-hand corner. This is inconsistent with the signal matrix convention that is used in columnwise data analysis—where the “origin” is in the lower right-hand corner. To transform a `contour` plot into the expected DSP form, apply the `rot90` function to the data prior to plotting.

2-D & 3-D Plotting Functions	
`contour`	contour plot
`mesh`	3-D mesh plot
`meshdom`	2-D x–y domain for `mesh( )`
`rot90`	rotate by 90 degrees, for `contour( )`

2.5.4 Multiple Plots Per Page

MATLAB has limited ability to create a display with more than one plot; only 1, 2 or 4 plots per page are possible. Grouping plots together is accomplished with the `subplot` command, which sets up the configuration of the graphics window. `subplot` takes one argument `’ijk’` which is a 3 digit integer: the graphic window is partitioned into an `i`-by-`j` matrix of small window tiles and the `k`-th of these is selected for the next plot command. The numbering of the plot in the 2×2 case is demonstrated by the following example:

```
>> t = -0.5:0.1:0.5;
>> sss = sin(pi*t);
>> subplot(221)
>> plot(1:9, ’o’), title(’FIRST’)
>> subplot(222)
>> plot(3:-1:-7), title(’SECOND’)
>> subplot(223)
>> plot(t, sss),   title(’THIRD’)
>> subplot(224)
>> plot(t, sss.ˆ2, ’x’),  title(’FOURTH’)
```

produces the four plots shown in Figure 2.1. To reset the graphics window back to one plot per page, use `subplot` with no argument.

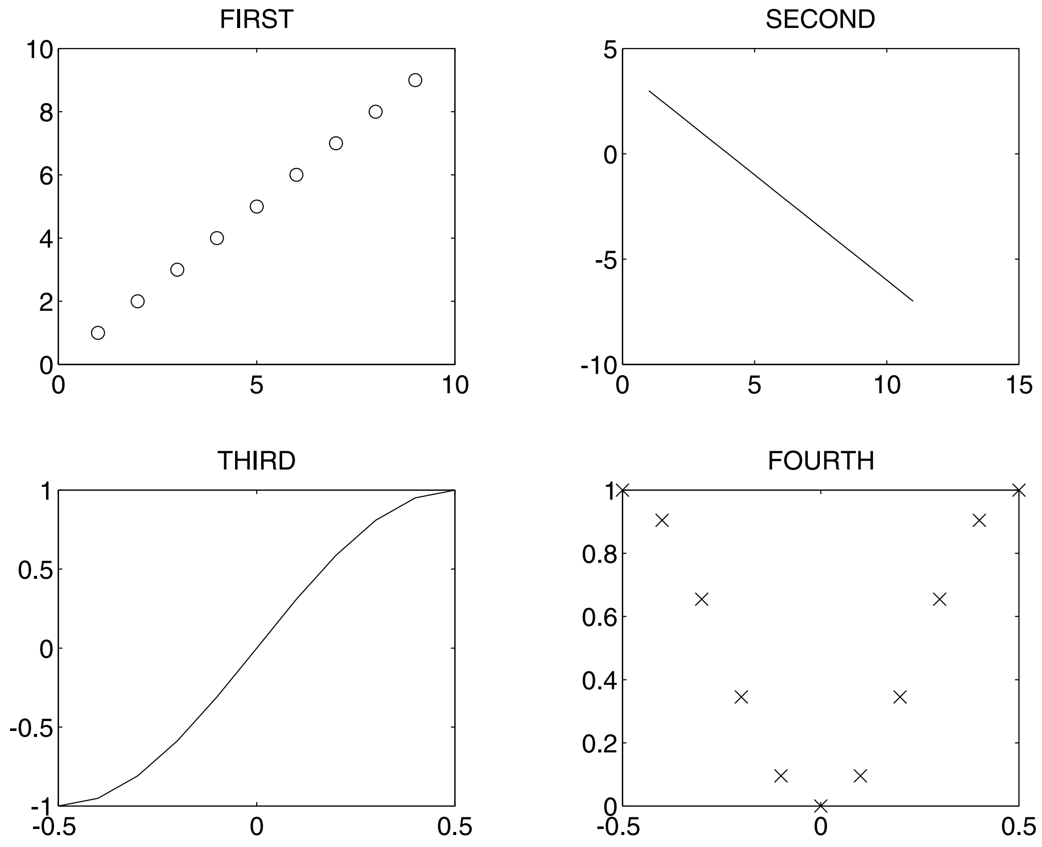

Figure 2.1: Using `subplot` to put multiple plot on one page.

2.5.5 Controlling the Graphics

Several commands can be used to control the behavior of the graphics window. The most useful of these is `hold` which inhibits the "clear screen" that is done prior to every plot command. Thus it is possible to overlay several plots. The automatic scaling is done for the first plot, so futher plots must adhere to that scaling. The `axis` command allows the user to specify the scaling of the graphics window.

Graphics Window Control	
`axis`	freeze the axis limits; also manual scaling
`clg`	clear graphics screen
`hold`	hold plot (for overlaying multiple plots)
`pause`	wait between plots in an M-file
`shg`	show graphics screen
`subplot`	2 or 4 plots per page

Graphic hardcopy is usually site-dependent, but it is important to realize that a post-processing program is needed to actually produce the graphics output. This program is called `gpp` (graphics post processor). The input to `gpp` is created within MATLAB using the `meta` command, which turns the plots on the screen into disk files that can be used by `gpp`. Consult your site manager for details on making hardcopy of your plots.

Graphics Hard Copy	
`!gpp` *filename*	graphics post processor (OS shell command) makes hard copy from a metafile
`meta` *filename*	make graphics metafile from present screen plot
`print`	send graph screen to printer *(system dependent)*
`print -deps`	produce Encapsulated POSTSCRIPT file
`prtsc`	print screen dump *(system dependent)*

Note: the Student version only permits screen dumps.

2.6 User Interface

MATLAB is most often used as a self-contained system, but it does not have to be isolated from other programs and external sources of data. In signal processing, it is especially important to be able to bring data into MATLAB from external sources, such as an A/D system. The FFT and plotting capabilities alone make MATLAB a good system for viewing signals and making sonograms.

2.6.1 Disk Files

The interface into the file system from MATLAB is rather simple. The user is always located in a current directory, which can be changed via the `chdir` command. The user can obtain a list of all files with `dir`; just the M-files and the `*.mat` files are listed by the `what` command. Files can be deleted (`del`) and they can be typed out to the terminal (`type`).

Since MATLAB is based on the concept of a workspace, it is essential that the user be able to save and restore the entire workspace; the commands `save` *filename* and `load` *filename* will accomplish this.

Disk File Operations	
`chdir`	change working directory
`delete`	delete a file
`diary`	write diary of the session to disk file
`dir`	directory of files on disk
`fprintf`	formatted printf à là C
`load`	load variable from `.mat` file or ASCII file
`pack`	compact memory (garbage collection?) equivalent to `save, clear, load`
`save`	save variable to a `.mat` file
`!translate`	read & write other data formats
`type`	list function or file
`what`	show M-files and `.mat` files on disk

Memory Limitations

In the MATLAB environment, variables are continually being created and destroyed, so there is a need for some sort of "garbage collection" to remove dead (or unused) space. In PC-MATLAB, it is easy to run out of space. Unfortunately, MATLAB has no automatic garbage collection mechanism. The closest approximation is a MATLAB function called `pack` that wil `save` all the variables in the entire workspace, then `clear` the entire workspace, and finally `load` the saved variables. This is time-consuming, but it is the only way to get some room to work when memory limits start to hinder your progress.

2.6.2 Importing and Exporting Data

MATLAB has its own binary file format that is used in the `.mat` files. Each file contains a small header, followed by all the array values in double-precision binary format. There are utilities for converting from other common formats (e.g., ASCII) into the `.mat` file form. The `load` will actually read ASCII files in which each row contains the same number of data items. The `!translate` program will exchange data between the `.mat` file format and others. And if those are insufficient, some C functions (`mload.c` and `msave.c`) are available, so the user can write a customized translation program between the `.mat` file form and any other specialized format.

Version 4 Changes

In version 4, file I/O is greatly enhanced by functions such as `fread` and `fwrite` which operate similar to the C functions with the same names. The functions allow direct binary reading and writing of disk files. This is especially useful for importing data from A-to-D converters, such as 12-bit speech signals and 16-bit sound, as well as 8-bit image data from scanners.

2.6.3 Diary: Recording a User's Session

Since MATLAB is an interpreted language, the normal mode of operation is for the user to issue commands, one at a time, from the keyboard. These input commands are echoed in the command window; most of the output is displayed in the same command window. Once the workspace becomes cluttered with many variables after a long sequence of commands, it is hard to keep track of everything. One aid is the `diary` command, which allows the user to direct all command window input and output to a file. Thus a record can be made of an entire MATLAB session. This can be useful in debugging, or in producing final reports as might be needed for homework problems or projects.

Chapter 3

Vectors, Vectors, Everywhere

This is the heart and soul of MATLAB. It is certainly the case that an appreciation of matrices and vectors will make MATLAB seem logically consistent.

3.1 Matrix and Array Operations

A full range of numerical operations on matrices and vectors is built into MATLAB. There are two categories of array operations: (1) those that obey the laws of matrix algebra, and (2) element-wise operations that are applied individually over the entire array.

3.1.1 Matrix Operations

The basic matrix operations are addition of matrices (denoted `+`), subtraction (`-`), multiplication (`*`), and conjugate transpose (`′`). For example, the statement

```
>> D = A*B + C′
```

says to form the new matrix `D` by multiplying the matrix `A` times the matrix `B` and add the result to the conjugate transpose of the matrix `C`. Of course, this will only work if the dimensions of `A`, `B`, and `C` are consistent for the operations specified. Recall that matrix multiplication is only defined when the number of columns in `A` equals the number of rows in `B`. If not, MATLAB will complain and try to tell you what is wrong. Scalars can multiply any matrix or vector.

In addition to the standard operations above, MATLAB supports two forms of "matrix division": left and right inverses. These operations can be used to solve sets of linear equations, using either the left inverse operator `\` or the right inverse operator `/`. Thus,

`x = A\b` is a solution to `A*x = b`

`x = b/A` is a solution to `x*A = b`

If the matrix `A` is a nonsingular square matrix, "left division", denoted `A\b`, corresponds to left multiplication of `b` by the inverse of `A`. That is, an equivalent MATLAB expression would be `inv(A)*b`. Similarly `b/A` which denotes "right division" is equivalent to `b*inv(A)` when `A` is invertible. It is worth pointing out that, in both cases, MATLAB actually computes the matrix division result without explicitly forming the matrix inverse.

Matrix Operators		**Elementwise Array Ops**	
`+`	addition	`+`	*same*
`-`	subtraction	`-`	*same*
`*`	multiplication	`.*`	pointwise multiply
`/`	right (pseudo)-inverse	`./`	right division
`\`	left (pseudo)-inverse	`.\`	left division
`^`	matrix power: $\mathbf{A}^n$	`.^`	element powers
`'`	conjugate transpose	`.'`	transpose (no conjugate)

3.1.2 Simultaneous Linear Equations

A very common operation needed in all fields is the solution of a set of simultaneous linear equations. Since this problem can be expressed as a matrix equation:

$$\mathbf{Ax} = \mathbf{b} \qquad \Longrightarrow \qquad \mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

the solution is obtained either by inverting $\mathbf{A}$, to get $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, or, more generally, by using the left inverse of $\mathbf{A}$. In MATLAB the answer is obtained with the left inverse, `x = A\b`, which has the advantage that all of the special situations will be taken care of (e.g., over-determined, under-determined, singular, etc.)

In the general case where `A` is not square, or square but not invertible, the "pseudo-inverse" of `A` is used to solve the linear equations. (NOTE: when the solution of `Ax = b` is not unique, the MATLAB function `x = pinv(A)*b` gives a different answer from `x = A\b`.) See help on `pinv`, `/` or `\` and your favorite linear algebra text for a discussion of the pseudo-inverse. As a final note, the left inverse operator is extremely useful for least-squares approximation problems in which the linear equations are over-determined. These situations arise in signal modeling (e.g., Prony's method, linear prediction, etc.).

3.1.3 Element-by-Element Array Operations

Array operations, as opposed to *matrix* operations, are element-by-element arithmetic operations. Instead of the usual matrix operation symbols `+ - * / \`, the "period" operator is concatenated with each symbol, `.* .\ ./` , to represent element-by-element multiplication, left division, and right division, respectively. Whenever element-by-element operations are used, the size of the matrices involved must be identical; otherwise, MATLAB will complain. For example, suppose

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

Then the pointwise multiplication and division operators produce:

```
>> z = x.*y
z =
     4    10    18

>> x.\y, x./y
ans =
    4.0000    2.5000    2.0000
ans =
    0.2500    0.4000    0.5000
```

The array operation of raising to a power, `.^`, can take three forms. If both `x` and `y` are matrices (vectors) with the same dimensions, the result `z = y.^x` is a matrix (vector) of the same dimension with entries $z_i = y_i^{x_i}$, e.g., for `x` and `y` above,

```
>> z = y.^x
z =
     4    25   216
```

When one of the operands is a scalar, it is used over the entire matrix. The resulting matrix has the same dimensions as the matrix operand. If the exponent is a scalar, we have

```
>> z = x.^3
z =
     1     8    27
```

or, when the base is a scalar, we have a convenient way to generate a geometric sequence, e.g., 2^{-n}:

```
>> n = 0:-1:-6
n =
     0    -1    -2    -3    -4    -5    -6

>> z = 2 .^n
z =
    1.0000    0.5000    0.2500    0.1250    0.0625    0.0312    0.0156
>> z = 2..^n
z =
    1.0000    0.5000    0.2500    0.1250    0.0625    0.0312    0.0156
>> z = 2.^n
??? Error using ==> ^
Matrix must be square.
```

The last statement, `z = 2.^n` shows that the "point" in these "point operations" might be misinterpreted when the numerical scalar, such as 2, could have a decimal point. The number must contain a decimal point, or be separated from the `.^` operator with a space or grouped in parentheses to avoid the confusion. This ambiguity is no longer present in version 4, `2.^n` is interpreted correctly as `2 .^n`.

3.1.4 Relational and Logical Operators

MATLAB has a full complement of built-in relational (greater than, less than, etc.) and logical (AND, OR, NOT) operations. The symbols are obvious, except for logical NOT which is ~.

Relational Operators			**Logical Ops**	
<	<=	==	&	(Logical AND)
>	>=	~=	|	(Logical OR)
			~	(Logical NOT)

These are applied to vectors or arrays in a pointwise fashion. They return a value that is either `TRUE` or `FALSE`. MATLAB follows the conventions of the C language with respect to `TRUE` and `FALSE`, because `FALSE` is zero and `TRUE` is any non-zero value with a preferred value of 1. Use the help facility (`help relop` or `help logop`) to find out more details about how these work.

Numerous relational and logical functions are available to deduce properties of arrays and variables. Special variable types can be detected, strings compared and logical conditions can be applied across entire arrays. For example, `all` will compute the logical AND of each column of an array, and then return a row vector of 1's and 0's, indicating `TRUE` and `FALSE`.

Relational and Logical Functions	
`any`	logical OR over each column of an array
`all`	logical AND (column-wise)
`find`	list array indices where condition is TRUE
`exist`	check if variables exist
`isnan`	detect IEEE not-a-number NaN
`finite`	detect infinities
`isempty`	detect empty matrix
`isstr`	detect string variable
`strcmp`	compare string variables

An important use of the relational operators is to control an action over an entire vector, when programming in MATLAB. For example, the expression `x>5` when applied to a vector `x` will return another vector of the same size as `x` with entries equal to 1 when `x[i]` is greater than 5 and 0 otherwise. Then the relational function `find(y)` can be used to extract those indices where `y` is non-zero. For example, `x(find(x>5))` will return a (smaller) vector containing all the values in `x` which are greater than 5.

3.1.5 Scalar Math Functions

MATLAB also has many elementary math functions (sin, cos, log, etc.). In general, these are element-wise functions. (The matrix versions have different names, e.g., the matrix exponential, `expm(A)`.) The sinusoidal functions are very useful for generating discrete-time signals. For example, to generate one period of the sine-wave sequence $y[n] = \sin(2\pi n/8)$, type

```
>> n = 0:7;
>> y = sin(2*pi.*n/8)
y =
   0    0.7071    1.0000    0.7071    0.0000   -0.7071   -1.0000   -0.7071
```

Trigonometric Functions					
`sin`	`cos`	`tan`	`sinh`	`cosh`	`tanh`
`asin`	`acos`	`atan`	`asinh`	`acosh`	`atanh`
		`atan2`	(4-quadrant arctangent)		

Most math functions have the names that you would expect, so look for them with the `help` facility. In addition to the trigonometric and hyperbolic functions, there are functions that manipulate complex numbers: `abs` takes the magnitude, and `angle` computes the phase. The magnitude function is called `abs` because it is a generalization of the absolute value function defined for real numbers. In fact, all the math functions work correctly on complex numbers. For example, the logarithm of a complex number is defined as:

$$\log(z) = \log(|z|) + j\angle z$$

The following example shows that MATLAB implements the logarithm correctly for this case.

```
>> log(1-j), log(2)/2, -pi/4
ans =
    0.3466 - 0.7854i
ans =
    0.3466
ans =
   -0.7854
```

Elementary Math Functions			
`exp`	`imag`	`abs`	(complex mag)
`log`	`real`	`angle`	(phase)
`log10`		`conj`	(conjugate)
`sqrt`			

Special Functions

The last class includes a number of special math functions that find use in statistics, and in filter design.

Special Functions	
`bessel`	Bessel function
`ellipj`	complete elliptic integral of the first kind
`ellipk`	Jacobian elliptic functions
`erf`	error function
`gamma`	complete and incomplete gamma functions
`inverf`	inverse error function

3.1.6 Quantization Functions

Another class of math functions are those associated with quantization: rounding, truncation, remainders, etc. These are useful when creating indices. The remainder function can be modified slightly to create the circular indexing needed in DSP. As it stands, `rem` is not a modulo-N function because it returns an answer that has the same sign as the input. Included in this class is the `rat` function that approximates a real number with a nearby rational.

Quantization Functions	
`ceil`	round towards $+\infty$
`fix`	round towards 0 (truncation)
`floor`	round towards $-\infty$
`rat`	rational approximation to real number
`rem`	remainder (signed)
`round`	round to nearest integer
`sign`	signum function

3.1.7 Vector Math Functions

MATLAB also has a class of functions that act on the columns of a matrix. In other words, the columns of the matrix are treated as a group of vectors. This convention is used quite a bit in DSP, when a group of signals needs to be processed by the same algorithm. See Section 3.1.1 for a definition of a *signal matrix*.

The column-wise functions include the `mean`, `sum`, `prod`, `max`, `min`, etc. The style for these functions is to treat each column of a matrix as an individual vector, and apply the function over each vector. For example, when `sum` is applied to an $M \times N$ matrix, the result is an N-element row vector, a $1 \times N$ matrix, where the i^{th} entry in the vector is the sum of all the elements in the i^{th} column of the matrix.

Columnwise Data Analysis			
`corrcoef`	correlation coefficients	`max`	maximum
`cov`	covariance matrix	`mean`	mean or average
`cplxpair`	re-order into complex pairs	`median`	median
`cumprod`	cumulative product of elements	`min`	minimum
`cumsum`	cumulative sum of elements	`prod`	product of all elements
`diff`	1st difference (approx derivative)	`std`	standard deviation
`hist`	histogram (no plotting)	`sum`	sum of all elements
`sort`	sort vector (can also return index re-ordering)		

3.1.8 Matrix Functions & Decompositions

Since MATLAB was originally created as a linear algebra package, it has many sophisticated matrix operations and decompositions. The most well-known of these are the determinant (`det`), the trace (`trace`), the inverse (`inv`), the characteristic polynomial (`poly`), and the eigenvector–eigenvalue decomposition (`eig`). Other useful linear algebra operations include triangular factorization, orthogonal factorization, and the singular value decomposition, `svd()`. See the `help` facility for how to use them.

Matrix Manipulation	
`diag`	construct diagonal matrix from a vector
`fliplr`	flip rows of matrix left-to-right
`flipud`	flip cols of matrix up-and-down
`reshape`	change size of rows and columns
`rot90`	rotate 90 degrees (not transpose)
`size`	return matrix dimensions
`tril`	extract lower triangular part
`triu`	extract upper triangular part
`.'`	matrix transpose operator
`:`	general re-arrangement

Matrix Functions	
`det`	determinant of a square matrix
`expm`	matrix exponential (`expm1`, `expm2`, `expm3`)
`funm`	matrix function (user specified)
`kron`	Kronecker product of matrices
`logm`	matrix logarithm
`poly`	characteristic polynomial
`sqrtm`	matrix square root
`trace`	sum of the diagonal elements in a matrix

Matrix Condition Numbers	
`cond`	condition number in 2-norm
`norm`	matrix norms: 1-norm, 2-norm, F-norm, ∞-norm
`rank`	rank of a matrix
`rcond`	reciprocal of condition number

Matrix Decomposition and Factorization	
`balance`	improve condition number
`backsub`	back substitution (Gaussian elimination)
`cdf2rdf`	convert complex-diagonal to real-diagonal
`chol`	Cholesky factorization
`eig`	eigenvalues and eigenvectors
`hess`	Hessenberg form
`inv`	matrix inverse
`lu`	LU factors for Gaussian elimination
`nnls`	non-negative least-squares approximation
`null`	null space
`orth`	orthogonalization of a matrix
`pinv`	pseudo-inverse
`qr`	orthogonal-triangular decomposition
`qz`	QZ algorithm
`rref`	reduced row echelon form
`rsf2csf`	convert real-schur to complex-schur
`schur`	Schur decomposition
`svd`	singular value decomposition

3.1.9 Outer Products

These can be used to replicate a row or a column to produce a matrix, e.g., when `x` is a row vector, `ones(5,1)*x` will give a matrix with 5 identical rows.

3.1.10 SubMatrices

Build up more complex matrices from smaller ones

```
>> A = [ 1 2; 3 4;];
>> B = [ 1 1 1 ; 2 0 3;];
>> C = [ A, B ]
>>  C = [ 1  2  1  1  1
          3  4  2  0  3 ]
```

Can also use the function `kron` to repeat and scale submatrices:

```
>> A = [ 1 2; 3 4;];
>> B = [ 1 1 1 ; 2 0 3;];
>> C = kron(A,B)
>>  C = [ 1  1  1  2  2  2
          2  0  3  4  0  6
          3  3  3  8  8  8
          6  0  9  16  0 24  ]
```

Notice the scaled copies of `B` that make up `C`.

3.1.11 Empty Matrices

The empty matrix is a legitimate construct in MATLAB. Some operations naturally produce the empty set as an answer, so MATLAB uses the notation `[  ]` to denote it.

3.1.12 Special Matrices

All of these functions are matrix constructors. Among the most often used are the matrices consisting of all ones, `ones()`, and all zeros, `zeros()`. The Toeplitz matrix and Vandermonde matrix forms arise quite often in DSP.

Special Matrices Known by Name			
`compan`	companion matrix	`linspace`	linearly spaced vectors
`diag`	diagonal matrix from a vector	`logspace`	logarithmically spaced vectors
`eye`	identity matrix	`magic`	magic square
`gallery`	menu of various esoteric forms	`meshdom`	domain for mesh plots
`hadamard`	Hadamard matrix	`ones`	matrix of all ones
`hankel`	Hankel matrix	`rand`	matrix with random entries
`hilb`	Hilbert matrix	`toeplitz`	Toeplitz matrix
`ident`	same as `eye`	`vander`	Vandermonde matrix
`invhilb`	inverse of Hilbert matrix	`zeros`	matrix of all zeros

3.1.13 Advanced Numerical Functions

There are also a number of functions for numerical solution of problems: numerical integration, numerical solution of differential equations, spline fitting for interpolation, and numerical computation of the zeros of a non-linear function.

Interpolation	
`spline`	cubic spline
`table1`	1-D table look-up
`table2`	2-D table look-up

Differential Equation Solution, Integration	
`ode23`	2nd/3rd order Runge-Kutta method
`ode45`	4th/5th order Runge-Kutte-Fehlberg method
`quad, quad8`	numerical function integration

Non-Linear Equations and Optimization	
`fmin`	min of function of one variable
`fmins`	min of multivariable function (unconstrained)
`fsolve`	solution to system of nonlinear equations (zeros of a multi-variable function)
`fzero`	zero of a function of one variable

MATLAB also has functions for finding roots and doing other manipulations of polynomials These are discussed in Section 3.2.

Chapter 4

Programming in MATLAB

One significant value of the MATLAB environment it that it is extensible, because the user can create new functions and add them to the environment. Including new functions in the environment is as simple as creating an ASCII file and putting it in a directory that is on the path specified by the environment variable `MATLABPATH`. These functions are called "M-files" because the file name must have a `.m` extension. Indeed, many of the functions supplied in the standard toolboxes are actually M-files. Since each M-file is an ASCII file, an excellent way to learn MATLAB programming is to view some of the existing M-files, which can be done using the `type` command.

There are two type of M-files which serve different purposes:

1. *Script:* a sequence of often repeated commands

2. *Function:* a transformation from input variables to outputs.

The function M-file contributes a new verb (or action) to the MATLAB language. Since it transforms inputs to outputs, it also matches the concept of a function in mathematics or a system in engineering. The script M-file, onf the other hand, is convenient for saving demos and plots with labels and annotation.

This chapter presents a style of programming that should help improve your MATLAB programs. For more ideas and tips, study some of the functions provided in this appendix, or some of the M-files found in the toolboxes of MATLAB. Copying the style of other programmers is always an efficient way to improve your own knowledge of a computer language. In the discussion below, we present the most important points involved in writing good MATLAB code assuming that your are both an experienced programmer and have progressed beyond the novice level as a MATLAB user.

4.1 Editing ASCII M-files

M-files are regular ASCII text files with a "`.m`" extension. They can be created with any text editor, e.g., `emacs`. In the course of creating and debugging an M-file, you need to go back and forth between MATLAB and the editor. The exact procedure for doing this is system-dependent.

The specific editor used is a matter of personal taste and also depends on the operating system. For the Macintosh version of MATLAB an editor is built into the MATLAB environment; for DOS and UNIX external editing programs must be used. With the advent of "windowed" operating systems, the editor is merely run in a separate window. Previously, in version 3.5 of MATLAB running under DOS, a "shell escape" was needed to switch from MATLAB to the editor without killing MATLAB.

Suppose that you want to edit (or create) the file `myfunc.m` using the generic editor in DOS. Using the `!` feature, you simply type

```
>> !edit myfunc.m
```

This starts the edit program under DOS and suspends MATLAB. When you are finished editing, save the M-file and exit the editor as you normally do. This will re-activate MATLAB so that you can test your M-file. Since DOS imposes a severe memory limit, you may have to use a small editor so that both MATLAB and the editor can be resident in the memory simultaneously.

On the Macintosh, the editor is coupled into the MATLAB program. One feature that is available is the ability to "save and execute" the edit window. Under version 3.5, this was done with `cmd-G`; in version 4, use `cmd-E`, or pull down the File Menu when the edit window is active.

4.2 Creating Your Own Scripts

Whenever you have a long sequence of MATLAB commands that you want to execute repeatedly, you should create a *script* M-file. A script file simply contains the list of commands, and, whenever the script filename is invoked, they will be executed as though you typed them in at the keyboard, one after another. An ideal use for a script is saving a sequence of plotting commands with labeling information, such as would be needed in a demo. In the following case, the sequence of commands would generate a signal, take its FFT and display the spectrum and the signal together as a two-panel `subplot`.

```
%--
%-- Demonstrate the spectrum of a sinusoid
%--
format compact
clear          %-- remove everything else from workspace
F_samp = 2000;
T_interval = 0.1;
F_sin = 23.45      %-- echo the sinusoid's frequency (Hertz)
tt = 0 : (1/F_samp) : T_interval;
xx = cos( 2*pi*F_sin*tt );
%
subplot(212)
plot( tt, xx ), xlabel('TIME  (sec)' )
title('SINE WAVE')
%
Nfft = 512;
XX = fft( xx, Nfft);
ff = F_samp*( [0:Nfft-1]/Nfft );
jkl = 0:Nfft/2;            %-- only show positive freqs
%
subplot(211)
plot( ff(jkl), abs(XX(jkl)) ), xlabel('FREQUENCY  (Hz)')
title('SPECTRUM of SINUSOID'), ylabel('MAGNITUDE')
```

Note the use of semicolons to suppress printing of intermediate results except for `F_sin`. If the foregoing example is saved into a file called `showfft.m`, then it can be invoked from the command line via:

```
>> showfft
```

When the commands are saved in a file, it would be easy to make minor changes to show other cases. For example, to change the frequency we only need to edit one line of the file to set `F_sin` to another value. Suppose that we want a sine wave with exactly four periods over the specified time interval. Then we would define `F_sin` as `F_sin = 4/T_interval`.

A second example will illustrate how a script differs from a function (next section). Suppose we wish to create a vector that is clipped—all elements larger than a certain threshold are set equal to the threshold limit. A script for doing this is as follows:

```
Lx = length(x);          %-- operate on vector x
for n=1:Lx               %-- Loop to do the center clip
   if( abs(x(n)) > thresh )      %-- need a threshold
      x(n) = sign(x(n)*thresh;
   end
end
```

If these commands were contained in a file called `clipit.m` on the MATLAB path, then typing `>> clipit` would cause MATLAB to first determine the length of the vector `x` using the function `length`, then use a `for` loop to find the elements in `x` that have to be saturated. Doing things this way assumes that the vector named `x` already exists in the MATLAB workspace, and that a threshold has been defined using the name `thresh`. Furthermore, the the vector `x` is modified by the script. In other words, `x` and `thresh` are global variables in the workspace, as are the auxiliary variables `n` and `Lx`. (Warning: if we had used `i` as the loop index, then start-up definition of `i` as the $\sqrt{-1}$ would have been clobbered.) This sort of script is not necessarily a good way to do this problem because it is so inflexible—if we want to clip a different vector `y` we must edit `clipit.m`.

4.3 Creating Your Own Functions

A better way to do the clip operation is to create a *function* M-file that takes two input arguments (a signal vector and a scalar threshold) and returns an output signal vector. We will show that most functions can be written according to a standard format. In order to continue with the clip example, use your editor to create an ASCII file `clip.m` that contains the following statements:

Eight lines of comments at the beginning of the function will be the response to `help clip`

First step is to figure out matrix dimensions of x

Input could be row or column vector

Since x is local, we can change it without affecting the workspace

create output vector

```
function  y = clip( x, Limit )
%CLIP    saturate mag of x[n] at Limit
%     when |x[n]| > Limit, make |x[n]| = Limit
%
%  usage:  Y = clip( X, Limit )
%
%      X  - input signal vector
%  Limit  - limiting value
%      Y  - output vector after clipping

[nrows ncols] = size(x);

if( ncols ~= 1 & nrows ~= 1 )      %-- NEITHER
   error('CLIP: input not a vector')
end
Lx = max([nrows ncols]);        %-- Length

for n=1:Lx          %-- Loop over entire vector
   if( abs(x(n)) > Limit )
      x(n) = sign(x(n))*Limit;    %-- saturate
   end
end
y = x;              %-- copy to output vector
```

Preserve the sign of x[n]

We can break down the M-file `clip.m` into four elements:

1. *Definition of Input/Output:* Function M-files must have the word `function` as the very first thing in the file. The information that follows `function` on the same line is a declaration of how the function is to be called and what arguments are to be passed. The name of the function should match the name of the M-file; if there is a conflict, it is the name of the M-file which is known to the MATLAB command environment.

   Input arguments are listed inside the parentheses following the function name. Each input is a matrix. The output argument (a matrix) is on the left side of the equals sign. Multiple output arguments are also possible, e.g., notice how the function `size(x)` in `clip.m` returns the number of rows and number of columns into separate output variables. Square brackets surround the list of output arguments. Finally, observe that there is no explicit return of the outputs; instead, MATLAB returns whatever value is contained in the output matrix when the function completes. The MATLAB function `return` just leaves the function, it does not take an argument. For `clip` the last line of the function assigns the clipped vector to `y`.

   The essential difference between the function M-file and the script M-file is dummy variables versus permanent variables. The following statement creates a clipped vector `wclipped` from the input vector `win`.

   ```
   >> wclipped = clip(win, 0.9999);
   ```

   The arrays `win` and `wclipped` are permanent variables in the workspace. The temporary arrays created inside `clip` (i.e., `y`, `nrows`, `ncols`, `Lx` and `i`) exist only while `clip` runs; then they are deleted. Furthermore, these variable names are local to `clip.m` so the name `x` could also be used in the workspace as a permanent name. These ideas should be familiar to anyone with experience using a high-level computer language like C, FORTRAN or PASCAL.

2. *Self-Documentation:* A line beginning with the `%` sign is a comment line. The first group of these in a function are used by MATLAB's help facility to make M-files automatically self-documenting. That is, you can now type `help clip` and the comment lines from *your* M-file

will appear on the screen as help information!! The format suggested in `clip.m` follows the convention of giving the function name, its calling sequence, definition of arguments and a brief explanation.

3. *Size and Error Checking:* The function should determine the size of each vector/matrix that it will operate on. This information does not have to be passed as a separate input argument, but can be extracted on the fly with the `size` function. In the case of the `clip` function, we want to restrict the function to operating on vectors, but we would like to permit either a row ($1 \times L$) or a column ($L \times 1$). Therefore, one of the variables `nrows` or `ncols` must be equal to one; if not we terminate the function with the bail out function `error` which prints a message to the command line and quits the function.

4. *Actual Function Operations:* In the case of the `clip` function, the actual clipping is done by a `for` loop which examines each element of the `x` vector for its size versus the threshold `Limit`. In the case of negative numbers the clipped value must be set to `-Limit`, hence the multiplication by `sign(x(n))`. This assumes that `Limit` is passed in as a positive number, a fact that might also be tested in the error checking phase.

4.3.1 Programming Primitives

In order to support a rich programming environment, MATLAB contains a number of operations that are essential for program flow: `if`, `else`, `for`, `while`, etc. When the operation must differentiate between `TRUE` and `FALSE`, the convention is the same as used in the C language: `FALSE` is the numerical value zero, and `TRUE` is any non-zero value, preferably one. The logical operators: `&` (AND), `|` (OR), and `~` (NOT), and relational operators: `>` (GREATER THAN), `<` (LESS THAN), and `==` (EQUAL TO) together with `any` and `all` permit compound logical statements to be written for `if` tests. See Chapter 3, section 3.1.4(?) for more details. Thus, writing structured programs is virtually identical to methods learned for C or FORTRAN.

Control Flow	
`break`	break out of `for` and `while` loops
`end`	terminate `if`, `for` or `while`
`for`	repeat according to a row vector
`if`	conditional
`else`	used with `if`
`elseif`	”
`pause`	pause until key is pressed
`return`	return from `function`
`while`	do while a condition is `TRUE`

Example: The condition $1 < x[n] \leq 2$ is expressed as a conjunction of two inequalities:

```
if( xn>1 & xn<=2 )
```

where `xn` is the MATLAB array containing the values of $x[n]$.

4.3.2 Avoid FOR Loops

Since MATLAB is an interpreted language, certain common programming habits are intrinsically inefficient. The primary one is the use of `for` loops to perform simple operations over an entire matrix or vector. *Whenever possible*, you should try to find a vector function (or the composition of a few vector functions) that will accomplish the same result rather than writing a loop. For example, if the operation were summing up all the elements in a matrix, the difference between calling `sum` and writing a loop that looks like FORTRAN code can be astounding—the loop is unbelievably slow due to the interpretative nature of MATLAB. Consider the following three methods for matrix summation:

Double Loop needed to index all matrix entries

```
[Nrows, Ncols] = size(x);
xsum = 0.0;
for m = 1:Nrows
   for n = 1:Ncols
      xsum = xsum + x(m,n);
   end
end
```

`sum` *acts on a matrix to give the sum down each column*

```
xsum = sum( sum(x) );
```

`x(:)` *is a vector of all elements in the matrix*

```
xsum = sum( x(:) );
```

The last two methods rely on the built-in function `sum` which has different characteristics depending on whether its argument is a matrix or a vector (called "operator overloading"). To get the third (and most efficient) method, the matrix `x` is converted to a column vector with the colon operator. Then one call to `sum` will suffice.

Repeating Rows or Columns

Often it is necessary to form a matrix by replicating a value in the rows or columns. If the matrix is to have all the same values, then functions such `ones(M,N)` and `zeros(M,N)` can be used. But when you want to replicate a column vector **x** to create a matrix that has eleven identical columns, you can avoid a loop by using the outer-product matrix multiply operation. The following MATLAB code fragment will do the job:

```
X = x * ones(1,11)
```

If `x` is a length L columns vector, then the matrix `X` formed by the outer product is $L \times 11$.

4.3.3 Vectorizing Logical Operations

The `clip` function offers a different opportunity for vectorization. The `for` loop in that function contains a logical test and might not seem like a candidate for vector operations. However, the logical operators in MATLAB apply to matrices. For example, a greater than test applied to a 3×3 matrix returns a 3×3 matrix of ones and zeros.

```
>> x = [ 1  2 -3; 3 -2  1; 4  0  -1]
>> x = [ 1  2 -3
         3 -2  1
         4  0 -1 ]
>> mx = x>0         %-- check the greater than condition
>> mx = [ 1  1  0
          1  0  1
          1  0  0 ]
>> y = mx .* x      %-- multiply by masking matrix
>> y = [ 1  2  0
         3  0  1
         4  0  0 ]
```

The zeros mark where the condition was false; the ones denote true. Thus, when we multiply `x` by the masking matrix `mx`, we get a result that has all negative elements set to zero. Note that the entire matrix has been processed without using a loop.

Since the saturation done in `clip.m` requires that we change the large values in `x`, we can implement the `for` loop with three array multipications. This leads to a vectorized saturation operator that works for matrices as well as vectors:

```
y = x.*(abs(x)<=Limit) + Limit*(x>Limit) - Limit*(x<-Limit);
```

Three different masking matrices are needed to represent the three cases of negative saturation, positive saturation, and no action. The additions correspond to the logical OR of these cases. The number of arithmetic operations needed to carry out this statement is $3N$ multiplications and $2N$ additions where N is the total number of elements in `x`. On the other hand, the statement is interpreted only once.

Exercise

Write a variation of the `clip` function with three input arguments: the matrix `x`, and two thresholds, and upper limit and a lower limit.

Creating an Impulse

Another simple example is given by the following trick for creating an impulse signal vector:

```
nn = [-20:80];
impulse = (nn==0);
```

This result could be plotted with `comb( nn, impulse )`, or `stem( nn, impulse)` in version 4. In some sense, this code fragment is perfect because it captures the essence of the mathematical formula which defines the impulse as only existing when $n = 0$.

$$\delta[n] = \begin{cases} 1 & n = 0 \\ 0 & n \neq 0 \end{cases}$$

The Find Function

An alternative to masking is to use the `find` function. This is not necessarily more efficient; it just gives a different approach. The `find` function will determine the list of indices in a vector where a condition is true. For example, `find( x>Limit );` will return the list of indices where the vector is greater than the `Limit` value. Thus we can do saturation as follows:

```
    y = x;
    jkl = find(y>Limit;
    y( find(y>Limit ) ) = Limit*ones(jkl);
    jkl = find(y<-Limit);
    y( jkl ) = -Limit*ones(jkl);
```

The `ones` function is needed to create a vector on the right-hand side that is the same size as the number of elements in `jkl`. In version 4.0 this would be unecessary since a scalar assigned to a vector is now assigned to each element of the vector.

Seek to Vectorize

The dictum to "avoid `for` loops" is not always an easy path to follow, because it means the algorithm must be cast in a vector form. We have seen that this is not particularly easy when the loop contains a logical test, but such loops can still be "vectorized" if masks are created for all possible conditions. This does result in extra arithmetic operations, but they will be done efficiently by the internal vector routines of MATLAB, so the final result should still be much faster than an interpreted `for` loop.

4.3.4 Composition of Functions

MATLAB supports the paradigm of "functional programming" which in the language of system theory is equivalent to cascading systems. Consider the following equation which can be implemented with one line of MATLAB code.

$$\sum_{n=1}^{L} \log\left(|x_n|\right)$$

Here is the MATLAB equivalent:

```
sum( log( abs(x) ) )
```

4.3.5 Programming Style

"May your functions be short and your variable names long." Each function should have a single purpose. This will lead to short simple modules that can be composed together with other functions to produce more complex operations. Avoid the temptation to build super functions with many options and a plethora of outputs.

MATLAB supports long variable names—up to 32 characters. Take advantage of this feature to give variables descriptive names. In this way, the number of comments littering the code can be drastically reduced. Comments should be limited to help information and documentation of tricks used in the code.

4.4 The COLON Operator

One essential part of MATLAB that is needed to avoid `for` loops is the colon notation for selecting parts of matrices (see section 2.3.6(?) for more info). The help for `:` is given below:

```
>>help :
:   Colon. Used in subscripts, FOR iterations and possibly elsewhere.
```

```
J:K  is the same as [J, J+1, ..., K]
J:K  is empty if J > K.
J:I:K  is the same as [J, J+I, J+2I, ..., K]
J:I:K  is empty if I > 0 and J > K or if I < 0 and J < K.
The colon notation can be used to pick out selected rows,
columns and elements of vectors and matrices.
A(:) is all the elements of A, regarded as a single
column. On the left side of an assignment statement, A(:)
fills A, preserving its shape from before.
A(:,J) is the J-th column of A
A(J:K) is A(J),A(J+1),...,A(K)
A(:,J:K) is A(:,J),A(:,J+1),...,A(:,K) and so on.
For the use of the colon in the FOR statement, See FOR.
```

The colon notation works from the idea that an index range can be generated by giving a start, a skip, and then the end. Therefore, a regularly spaced vector of integers is obtained via

```
iii = start:skip:end
```

Without the `skip` parameter, the increment is 1. Obviously, this sort of counting is similar to the notation used in FORTRAN DO loops. However, in MATLAB you can take it one step further by combining it with a matrix. If you start with the matrix `A`, then `A(2,3)` is the scalar element located at the 2nd row, and 3rd column of `A`. But you can also pull out a 4×3 sub-matrix via `A(2:5,1:3)`. If you want an entire row, the colon serves as a wild card: i.e., `A(2,:)` is the 2nd row. You can even flip a vector by just indexing backwards: `x(L:-1:1)`. Finally, it is sometimes necessary to just work with all the values in a matrix, so `A(:)` gives a column vector that is just the columns of `A` concatenated together. More general "reshaping" of the matrix `A` can be accomplished with the `reshape(A,M,N)` function. For example, a 5×4 matrix `B` can be reshaped into a 2×10 matrix via: `Bnew = reshape(B,2,10)`

4.5 Debugging an M-file

Since MATLAB is an interactive environment, debugging can be done by examining variables in the workspace. However, the big drawback in version 3.5 was the lack of a break point facility to stop the execution of a function at a given line and then examine variables; version 4 contains a debugger with support for break points. In version 3.5, stepping through a function M-file is not possible. Rather the user must mimic the behavior of a function, or a script, by re-executing, by hand, the lines that make up the function. Obviously, this suggests further that writing short functions is a good thing—they are easier to debug.

Programming and M-files	
`casesen`	set case sensitivity
`disp`	display matrix or text
`echo`	enable command echoing, for debugging
`error`	display error message to command screen
`etime`	elapsed time measurement
`eval`	interpret text in variable as a command
`exist`	check if variables exist
`function`	define a function (with args)
`getenv`	get environment string
`global`	define a global variable
`input`	get numbers from keyboard
`keyboard`	call keyboard as M-file
`menu`	select item from menu
`nargin`	number of input args to a function
`nargout`	number of output args from a function
`startup`	startup M-file (site dependent)

There are some support functions that will aid in debugging: `echo`, `disp`, `keyboard`, and `error`. The `echo` function will type out the result of all lines of code in a function, so that a dump of all possible information is done. `disp` is a print function which can be inserted to display a matrix within a function; `error` will test a condition, and then print a message and stop the function if the condition is true. The function `keyboard` comes the closest to being a breakpoint operator, because it pauses the program and returns control to the user at the keyboard. Then the user can enter MATLAB commands, for example to display some arrays or check numerical values. The keyboard mode is terminated in the same way as MATLAB, so you must careful not to hit the exit key twice (i.e., `ctl-D` in UNIX, `ctl-Z` in DOS, or `cmd-Q` on the Mac). Upon termination, the execution of the suspended function is resumed.

Command Window Manipulation	
`clc`	clear command screen
`format`	set output display format precision
`home`	home cursor

4.5.1 Version 4 Debugger

Need some info here

4.5.2 Timing Loops and Functions

One objective of algorithm testing in MATLAB is correctness, but another is efficiency. Efficiency can be measured by timing an operation or by counting the number of operations. MATLAB provides functions for both: `etime` to compute the elapsed time between two lines in a program; and `flops` to count the cumulative number of floating point operations done in all the functions used by MATLAB. On a time-shared system, elapsed time measurements may be highly variable depending on system load, number of users, etc.

The “flop” count is cumulative, but it can be reset to zero by using `flops(0)`. Thus the following code will count the number of flops in an FFT.

```
N = 512;
x = rand(N,1) + sqrt(-1)*rand(N,1);
flops(0);
X = fft(x,N);
numOfops = flops
```

The count for the FFT will include all multiplications and additions (real) including some operations needed to set up the table of exponentials used by `fft`.

Timing Measurements	
`clock`	wall clock time
`etime`	elapsed time
`flops`	count of floating-point operations

4.6 Programming Tips

This section presents a few programming tips that should help improve your MATLAB programs. For more ideas and tips, study some of the functions provided in this appendix, or some of the M-files in the toolboxes of MATLAB. Copying the style of other programmers is always an efficient way to improve your own knowledge of a computer language. In the hints below, we discuss some of the most important points involved in writing good MATLAB code. These comments assume that your are both an experienced programmer and at least an intermediate user of MATLAB.

4.6.1 Avoid FOR Loops

There is temptation among experienced programmers to use MATLAB in the same fashion as a high-level language like FORTRAN or C. However, this leads to very inefficient programming whenever `for` loops are used to do operations over the elements of a vector, e.g., summing the elements in a vector. Instead, you must look for the MATLAB functions that will do the same operation with a function call—in the case of summing, there is a MATLAB function called `sum`.

An alternative strategy that also avoids `for` loops is to use vector operations. In the sum example, the trick is to recognize that the sum of the elements in a row vector can be obtained by multiplying by a column vector of all ones. In effect, the inner product operation computes the sum.

The primary reason for introducing these tricks is that a `for` loop is extremely inefficient in MATLAB which is an interpreted language. Macro operations such as matrix multiplies are about as fast as micro operations such as incrementing an index, because the overhead of interpreting the code is present in both cases. The bottom line is that `for` loops should only be used as a last resort, and then probably only for control operations, not for computational reasons. More than likely, 90% of the `for` loops used in ordinary MATLAB programs can be replaced with equivalent, and faster, vector code.

4.6.2 Vectorize

The process of converting a `for` loop into a matrix-vector operation could be referred to a "vectorizing." Sometimes vectorizing appears to give a very inefficient answer in that more computation is done than in the `for` loop. Nonetheless, the resulting program will run much faster because one simple operation is applied repeatedly to the vector.

Repeating Rows or Columns

Often it is necessary to form a matrix by repeating one or more values throughout. If the matrix is to have all the same values, then functions such `ones(M,N)` and `zeros(M,N)` can be used. But suppose that you have a row vector **x** and you want to create a matrix that has 10 rows each of which is a copy of **x**. It might seem that this calls for a loop, but not so. Instead, the outer-product matrix multiply operation can be used. The following MATLAB code fragment will do the job:

```
X = ones(10,1) * x
```

If `x` is a length L row vector, then the matrix `X` formed by the outer product is $10 \times L$.

Vector Logicals

One area where slow programs are born lies in conditionals. Seemingly, conditional tests would never vectorize, but even that observation is not really true. Within MATLAB the comparison functions such as greater than, equal to, etc. all have the ability to operate on vectors or matrices. Thus the following MATLAB code

```
[1 2 3 4 5 6] < 4
```

will return the answer `[1 1 1 0 0 0]`, where `0` stands for `FALSE` and `1` represents `TRUE`.

Another simple example is given by the following trick for creating an impulse signal vector:

```
nn = [-20:80];   impulse = (nn==0);
```

This result could be plotted with `comb( nn, impulse )`. In some sense, this code fragment is perfect because it captures the essence of the mathematical formula which defines the impulse as only existing when $n = 0$.

Vectorize a CLIP function

To show an example of vectorizing at work, consider writing an M-file that will clip an input signal to given upper and lower limits. The code from a conventional language would look like the following in MATLAB:

```
  function  y = clip( x, lo, hi )
  % CLIP ---  threshold large and small elements in matrix x
  %  ==========> SLOWEST POSSIBLE VERSION <================
  %
  [M,N] = size(x);
  for m = 1:M
    for n = 1:N
      if x(m,n) > hi
         x(m,n) = hi;
      elseif x(m,n) < lo
         x(m,n) = lo;
end, end, end
```

The problem with this first version is the doubly nested `for` which is used to traverse all the elements of the matrix. In order to make a faster version, we must drop the loop altogether and use the vector nature of logicals. Furthermore, we can exploit the fact that `TRUE` and `FALSE` have numerical values to use them as masks (via multiplication) to select parts of the matrix `x`. Note that `( [x<=hi] + [x>hi] )` is a matrix of all ones.

```
function  y = clip( x, lo, hi )
% ===========> FAST VERSION <=============
%  (uses matrix Logicals to replace Loops)
y = (x .* [x<=hi])  +  (hi .* [x>hi]);
y = (y .* [x>=lo])  +  (lo .* [x<lo]);
```

If you count the number of arithmetic operations done in the second version, you will find that it is much greater than the count for the first version. To see this, use a very large matrix for `x`, and time the two functions with `etime` and `flops`. Even though you can generate cases where the second version requires 10 times as many operations, it will still run much faster—maybe 10 times faster!

4.6.3 The COLON Operator

One essential part of MATLAB that is needed to avoid `for` loops is the colon notation for selecting parts of matrices. The help for `:` is given below:

```
>>help :
:   Colon. Used in subscripts, FOR iterations and possibly elsewhere.
    J:K  is the same as [J, J+1, ..., K]
    J:K  is empty if J > K.
    J:I:K  is the same as [J, J+I, J+2I, ..., K]
    J:I:K  is empty if I > 0 and J > K or if I < 0 and J < K.
    The colon notation can be used to pick out selected rows,
    columns and elements of vectors and matrices.
    A(:) is all the elements of A, regarded as a single
    column. On the left side of an assignment statement, A(:)
    fills A, preserving its shape from before.
    A(:,J) is the J-th column of A
    A(J:K) is A(J),A(J+1),...,A(K)
    A(:,J:K) is A(:,J),A(:,J+1),...,A(:,K) and so on.
    For the use of the colon in the FOR statement, See FOR.
```

The colon notation works from the idea that an index range can be generated by giving a start, a skip, and then the end. Therefore, a regularly spaced vector of integers is obtained via

```
iii = start:skip:end
```

Without the `skip` parameter, the increment is 1. Obviously, this sort of counting is similar to the notation used in FORTRAN DO loops. However, in MATLAB you can take it one step further by combining it with a matrix. If you start with the matrix `A`, then `A(2,3)` is the scalar element located at the 2nd row, and 3rd column of `A`. But you can also pull out a 4×3 sub-matrix via `A(2:5,1:3)`. If you want an entire row, the colon serves as a wild card: i.e., `A(2,:)` is the 2nd row. You can even flip a vector by just indexing backwards: `x(L:-1:1)`. Finally, it is sometimes necessary to just work with all the values in a matrix, so `A(:)` gives a column vector that is just the columns of `A` concatenated together. More general "reshaping" of the matrix `A` can be accomplished with the `reshape(A,M,N)` function.

4.6.4 Matrix Operations

The default notation in MATLAB is matrix. Therefore, some confusion can arise when trying to do point-wise operations. Take the example of multiplying two matrices `A` and `B`. If the two matrices have compatible dimensions, then `A*B` is well defined. But suppose that both are 5×8 matrices and that we want to multiply them together element-by-element. In fact, we can't do matrix multiplication between two 5×8 matrices. To obtain point-wise multiplication we use the "point-star" operator `A .* B`. In general, when "point" is used with another arithmetic operator it modifies that operator's usual matrix definition to a point-wise one. Thus we have `./` and `.^` for point-wise division and exponentiation. For example, `xx = (0.9) .^ [0:49]` generates an exponential of the form a^n, for $n = 0, 1, 2, \ldots, 49$.

4.6.5 Signal Matrix Convention

Often it is necessary to operate on a group of signals all at once. For example, when computing the FFT on sections of a signal, it is convenient to put each section of the signal into one column of a matrix and then invoke the `fft` function to operate on the entire matrix. The result is that the 1-D FFT is computed down each column of the matrix. Another example along the same lines is the `sum` function which when applied to a matrix returns a vector answer—each element of the vector result is a column sum from the matrix. What would `sum( sum(A) )` compute for the matrix `A`?

This convention is not universal within MATLAB. For example, the `filter` function which is another workhorse DSP function will only process one vector at a time.

4.6.6 Polynomials

Another convention that is used in MATLAB, and is needed for DSP is the representation for polynomials. For the z-transform we often work with expressions of the form

$$H(z) = \frac{B(z)}{A(z)} = \frac{\sum_{\ell=0}^{M} b_\ell z^{-\ell}}{\sum_{k=0}^{N} a_k z^{-k}}$$

In MATLAB the polynomials $B(z)$ and $A(z)$ are represented by vectors `b` and `a` containing their coefficients. Thus `a = [1  -1.5  0.99]` represents the polynomial $A(z) = 1 - 1.5z^{-1} + 0.99z^{-2}$. From the vector form, we can extract roots via the M-file `roots(a)`, and also perform a partial fraction expansion with `residuez`. In addition, the signal processing functions `filter` and `freqz` both operate on the rational system function $H(z)$ in terms of the numerator and denominator coefficients: $\{b_\ell\}$ and $\{a_k\}$.

```
yout = filter(b, a, xin)

[H, W] = freqz(b, a, Nfreqs)
```

4.6.7 Self-Documentation via HELP

MATLAB has a very convenient mechanism for incorporating help into the system, even for user-written M-files. The comment lines at the beginning of any function are used as the help for that function. Therefore, it behooves the programmer to pay attention to documentation and to provide a few introductory comments with each M-file. For example, if you type `help freqz`, then the response is:

```
>> help freqz

 FREQZ  Z-transform digital filter frequency response.  When N is an integer,
    [H,W] = FREQZ(B,A,N) returns the N-point frequency vector W and the
    N-point complex frequency response vector G of the filter B/A:
                                 -1                 -nb
         jw  B(z)   b(1) + b(2)z + .... + b(nb+1)z
      H(e) = ---- = ----------------------------
                                 -1                 -na
         A(z)    1   + a(2)z + .... + a(na+1)z
    given numerator and denominator coefficients in vectors B and A. The
    frequency response is evaluated at N points equally spaced around the
    upper half of the unit circle. To plot magnitude and phase of a filter:
              [h,w] = freqz(b,a,n);
              mag = abs(h);  phase = angle(h);
              semilogy(w,mag), plot(w,phase)
    FREQZ(B,A,N,'whole') uses N points around the whole unit circle.
    FREQZ(B,A,W) returns the frequency response at frequencies designated
    in vector W, normally between 0 and pi. (See LOGSPACE to generate W).
    See also YULEWALK, FILTER, FFT, INVFREQZ, and FREQS.
```

You can also list the entire file `freqz.m` (by doing `type freqz`) to see that the help response consists of the initial comments in the file. If the M-file is a built-in, help is still available, e.g., for the `filter` function:

```
>> help filter

FILTER  Digital filter.
    Y = FILTER(B, A, X) filters the data in vector X with the
    filter described by vectors A and B to create the filtered
    data Y.  The filter is a "Direct Form II Transposed"
    implementation of the standard difference equation:

    y(n) = b(1)*x(n) + b(2)*x(n-1) + ... + b(nb+1)*x(n-nb)
                     - a(2)*y(n-1) - ... - a(na+1)*y(n-na)

    [Y,Zf] = FILTER(B,A,X,Zi) gives access to initial and final
    conditions, Zi and Zf, of the delays.
    See also FILTFILT.
```

Chapter 5

Signal Processing in MATLAB

Although MATLAB was not created originally as a signal processing package, it has many of the attributes that would be found in an ideal interactive DSP environment. Many of the basic MATLAB primitives naturally support DSP operations, because most DSP functions are based on mathematical operations found in linear algebra. However, since there is no data type in MATLAB that is called a signal, or a transform, it is necessary to establish some conventions for using vectors as signals. Remember that MATLAB has only arrays of numbers, so it is up to the user to assign a meaning, such as a "signal" to those arrays.

The DSP application clearly demonstrates the power of MATLAB when it is extended via new M-files. Most of the functions in the *Signal Processing Toolbox* are provided in the form of functions. The user can easily add some more, but the toolbox already provides many DSP functions for filter design, spectrum analysis, etc.

5.1 MATLAB Signal Processing Tool Box

These new files developed for signal processing are not the only ones available. The basic MATLAB release contains 67 signal processing functions that are listed below (using the `help` facility of MATLAB).

```
MatLab:Signal_Toolbox

  abcdchk   cheb1ap   decimate  fir1      invfreqs  nextpow2  series    xcorr2
  bartlett  cheb1ord  denf      fir2      invfreqz  numf      specplot  xcov
  bilinear  cheb2ap   detrend   freqs     kaiser    polystab  spectrum  yulewalk
  blackman  cheb2ord  dftmtx    freqz     kratio    prony     square    zp2ss
  boxcar    chebwin   ellip     grpdelay  lp2bp     rceps     ss2zp
  buttap    cheby1    ellipap   hamming   lp2bs     rcunwrap  tf2ss
  butter    cheby2    ellipord  hanning   lp2hp     readme2   triang
  buttord   conv2     fftfilt   hilbert   lp2lp     remez     vratio
  cceps     convmtx   filtfilt  interp    nargchk   remezdd   xcorr
```

Many of these functions are devoted to digital filter design—specifically the order determination for different types of IIR filters, and the design of FIR filters via windows and Chebyshev approximation (the `remez` function). Numerous windows are available for both filter design and spectrum analysis: Kaiser, Hamming, rectangular, etc. There are also functions for computing the complex cepstrum, and for convolution and correlation.

There are a variety of functions for doing Fourier analysis. For example, there is an `fft()` function, which takes a DFT of a vector. *It is not automatically padded out to the next power of two.* You can specify the DFT length by using a second argument `fft(x,n)`; then zero-padding will be done if the `length(x)` is less than `n`. If the input `x` is a matrix, then the 1-D FFT is performed on each of the columns; likewise, for the inverse FFT function `ifft(x,n)`. A 2-D version of the FFT is also included: `fft2` is part of the basic MATLAB toolbox.

The tables in the following sections summarize all the functions available in the Signal Processing Toolbox.[1] Try `help` on the individual function names for further information.

5.1.1 Version 3 of SP Toolbox

Need up-to-date information on new functionality

5.1.2 Signal Vector Convention

When studying DSP problems via MATLAB, we need a representation for signals and systems in terms of MATLAB's basic data object—the matrix. Obviously, we can represent a finite-length sequence of signal samples as a vector, either a row or a column. In some instances, it does not matter which you choose, but for consistency, it is best to always represent signals as column vectors. *If any* MATLAB *function doesn't accept this form you can always try the transpose.*

5.1.3 Signal Matrix Convention

The primary reason for standardizing on column vectors relates to processing multiple signals. For a group of signals, the convention is to use a matrix in which each column is an individual signal. Such a matrix will be called a *signal matrix*. Most of the DSP functions will accept a signal matrix as input and perform their operation on each column of the matrix. For example, when computing the FFT on sections of a signal, it is convenient to put each section of the signal into one column of a matrix and then invoke the `fft` function to operate on the entire matrix. The result is that the 1-D FFT is computed down each column of the matrix. Another example along the same lines is the `sum` function which when applied to a matrix returns a vector answer—each element of the vector result is a column sum from the matrix. What would `sum( sum(A) )` compute for the matrix `A`?

This convention is not universal within MATLAB. For example, the `filter` function which is another workhorse DSP function will only process one vector at a time.

5.2 Polynomials in Signals & Systems

Transforms are basic tools in the analysis of signals and systems, because they reduce many problems to algebraic operations on polynomials. MATLAB has numerous functions for dealing with polynomials. Polynomials are actually represented as vectors, where the vector elements are the polynomial coefficients in descending powers of the variable. Thus the polynomial $A(z) = 1 - 1.5z^{-1} + 0.99z^{-2}$ is represented by a vector `a = [1  -1.5  0.99]`. (This is still consistent with the z-transform which uses polynomials in z^{-1}; thus the last vector element is the coefficient of the highest negative power of z.) There are a number of polynomial functions: `roots` for finding roots, `poly` for reconstructing a polynomial from its roots, `conv` for multiplying polynomials, `deconv` for

[1]Ref: *Signal Processing Toolbox*, The Math Works, Inc.

dividing polynomials, `polyval` and `polyvalm` for evaluating polynomials, and `residue` and `residuez` for partial fraction expansions.

Polynomials	
`conv`	equivalent to poly multiplication
`deconv`	equivalent to poly division
	can return remainder, as well as quotient
`poly`	make polynomial from its roots, or characteristic polynomial of a matrix
`polyfit`	polynomial curve fitting (Least-Squares)
`polystab`	stabilization (flip roots inside UC)
`polyval`	polynomial evaluation
`polyvalm`	matrix polynomial evaluation
`residue`	partial-fraction expansion is s
`residuez`	partial-fractions for z-transform
`roots`	extracted via companion matrix method

There is a simple correspondence between the representation of a sequence as a vector and the representation of a polynomial as a vector. Since the z-transform of a finite-length sequence is just a polynomial

$$X(z) = \sum_{n=0}^{N-1} x[n]z^{-n}$$

we see that the same vector represents both the sequence *and* the coefficients of its z-transform polynomial, because the z-transform is written in ascending powers of z^{-1}.

System functions for discrete-time systems are rational functions of the form

$$H(z) = \frac{\sum_{\ell=0}^{M} b_\ell z^{-\ell}}{\sum_{k=0}^{N} a_k z^{-k}}$$

MATLAB represents such rational functions as two vectors of polynomial coefficients, one for the numerator and one for the denominator.

Unfortunately, two sources of confusion arise due to notational differences. The first is that MATLAB indexes its rows and columns beginning at 1, instead of 0. You simply have to adjust your thinking to take care of this. The other is due to the fact that in representing polynomials as vectors, the vector elements correspond to the polynomial coefficients in *descending* order. This is, in fact, what we want since it allows sequences to be indexed in *ascending* order in vectors. The only place this really causes a problem is in partial fraction expansions. For this reason there is a special function called `residuez` to make partial fraction expansions in the form

$$H(z) = \sum_{k=1}^{N} \frac{A_k}{1 - z_k z^{-1}}$$

On the other hand, if you are making a partial fraction expansion of a rational Laplace transform:

$$H(s) = \frac{\sum_{\ell=0}^{M} b_\ell s^{M-\ell}}{\sum_{k=0}^{N} a_k s^{N-k}} = \sum_{k=1}^{N} \frac{A_k}{s - s_k}$$

you would need to use the function `residue`. For example,

$$\frac{1}{s^2 + 5s + 6} = \frac{1}{s+2} + \frac{-1}{s+3}$$

which is verifed by the following MATLAB evaluation:

```
» a = [ 1 5 6 ];
» [r, p, k] = residue;
» r = [ 1 -1 ]
» p = [ 2 3 ]
» k = [ ]
```

In addition, the signal processing functions `filter` and `freqz` both operate on the rational system function $H(z)$ in terms of the numerator and denominator coefficients: $\{b_\ell\}$ and $\{a_k\}$.

```
yout = filter(b, a, xin)

[H, W] = freqz(b, a, Nfreqs)
```

5.3 Important Functions for DSP

In the course of doing basic DSP problems, there are two crucial functions that must be mastered. The first relates to the computation of the frequency response for a signal or a system. This can be accomplished with the MATLAB function `freqz`, which uses the `fft` function to do most of its computations. Understanding how these two functions work is the point of many of the exercises in basic DSP.

The second basic function is `filter` which performs the IIR or FIR filtering of a signal. Obviously, `filter` is used quite often to demonstrate the processing of signals. It can also be used as a signal generator when the input is an impulse or white noise.

5.3.1 Spectrum Analysis: fft & freqz

In spectrum analysis, the objective is to compute a version of the Fourier transform. For discrete-time signals, the transform of interest in the DTFT (discrete-time Fourier transform). However, the DTFT is a function of a continuous frequency variable, ω, so it is not possible to compute all of the DTFT. Most homework problems require a derivation of the closed form of the DTFT, and we would like to use MATLAB to confirm that such analytical methods are correct.

From the computational point of view, we can only calculate samples of the Fourier transform, using a finite transform known as the DFT (discrete Fourier transform). The DFT is usually computed using the FFT (fast Fourier transform) algorithm, which is an extremely efficient method whenever the length of the transform is a highly composite integer, e.g., a power of 2. The `fft` function in

MATLAB will compute the DFT via the FFT algorithm. The length of the transform is either inherited from the length of the input vector, or specified directly by the user. The help screen provides a concise statement of the function's capability:

```
>>help fft

FFT    FFT(X) is the discrete Fourier transform of vector X.  If the
   length of X is a power of two, a fast radix-2 fast-Fourier
   transform algorithm is used.  If the length of X is not a
   power of two, a slower non-power-of-two algorithm is employed.
   FFT(X,N) is the N-point FFT, padded with zeros if X has less
   than N points and truncated if it has more.
   If X is a matrix, the FFT operation is applied to each column.
   See also IFFT, FFT2, and IFFT2.
```

A related function is `fftshift` which moves the zero frequency point to the middle of the vector. This is very convenient when plotting

```
>>help fftshift

FFTSHIFT Move zeroth lag to center of spectrum.
   Shift FFT.  For vectors FFTSHIFT(X) returns a vector with the
   left and right halves swapped.  For matrices, FFTSHIFT(X) swaps
   the first and third quadrants and the second and fourth quadrants.
   FFTSHIFT is useful for FFT processing, moving the zeroth lag to
   the center of the spectrum.
```

To get an approximation to the DTFT, it is possible to use the FFT, but there are some pitfalls, especially when computing a frequency response. Consider the following two cases:

1. DTFT of a *finite-length* signal, $x[n]$, containing L_x points.
2. *Frequency response:* DTFT of an infinite-length signal (or impulse response) that has a rational z-transform.

In the first case, the `fft` function is directly usable—the length of the FFT (N_{FFT}) is merely chosen to be equal to the number of frequency samples desired. This does not have to be the same as the signal length, because `fft` will zero-pad when $N_{\text{FFT}} > L_x$. However, in the (uncommon) case where the user truly wants $N_{\text{FFT}} < L_x$, the signal must be time-aliased before calling the `fft` function.

5.3.2 A DTFT Function

We need two functions for computing the DTFT. The MATLAB function `freqz` will suffice for the infinite signal case, but a new function will be needed to compute the DTFT of a finite-length signal. It should be called `dtft(h,N)`, and is essentially a layer that calls `fft(h,N)`.

```
function [H,W] = dtft( h, N )
%DTFT   calculate DTFT at N equally spaced frequencies
```

```
%   usage:   H = dtft( h, N )
%      h: finite-length input vector, whose length is L
%      N: number of frequencies for evaluation over [-pi,pi)
%             ==> constraint: N >= L
%
%      H: DTFT values (complex)
%      W: (2nd output) vector of freqs where DTFT is computed
%
N = fix(N);
L = length(h);  h = h(:);  %<-- for vectors ONLY !!!
if( N < L )
   error('DTFT: # data samples cannot exceed # freq samples')
end
W = (2*pi/N) * [ 0:(N-1) ]';
mid = ceil(N/2) + 1;
W(mid:N) = W(mid:N) - 2*pi;   % <--- move [pi,2pi) to [-pi,0)
W = fftshift(W);
H = fftshift( fft( h, N ) );  %<--- move negative freq components
```

Note that you don't have to input the signal length L, because it can be obtained by finding the length of the vector `h`. Furthermore, since the DTFT is periodic, the region from $\omega = \pi$ to 2π is actually the negative frequency region, so the transform values just need to be re-ordered. This is accomplished with the MATLAB function `fftshift` which exchanges the upper and lower halves of a vector. If `Hrot = fftshift(H)` is applied to the DTFT vector, then the $[-\pi, \pi]$ plot can be produced by noting that `Hrot(1)` is the frequency sample for $\omega = -\pi$.

When plotting in the transform domain it would be best to make a two-panel `subplot` as shown in Fig. 5.1. The MATLAB program that produces Fig. 5.1 is given below:

```
%--- example of calculating and plotting a DTFT
%---
format compact, subplot(111)
a = 0.88 * exp(sqrt(-1)*2*pi/5);
nn = 0:40;  xn = a.^nn;
[X,W] = dtft( xn, 128 );
subplot(211), plot( W/2/pi, abs(X) ); grid, title('MAGNITUDE RESPONSE')
   xlabel('NORMALIZED FREQUENCY'), ylabel('| H(w) |')
subplot(212), plot( W/2/pi, 180/pi*angle(X) ); grid
   xlabel('NORMALIZED FREQUENCY'), ylabel('DEGREES')
   title('PHASE RESPONSE')
```

5.3.3 Frequency Response: Rational Form

The frequency response calculation is needed quite often, particularly when doing IIR filter design or when working with exponential signals. To handle this case, the function `freqz(b,a,n)` should be used. The first two arguments to `freqz` are vectors containing the coefficients of the numerator (`b`) and denominator (`a`) polynomials in the rational form. The third argument is the number of frequency samples desired over the range $0 \leq \omega < \pi$. Thus, `freqz` is actually evaluating the

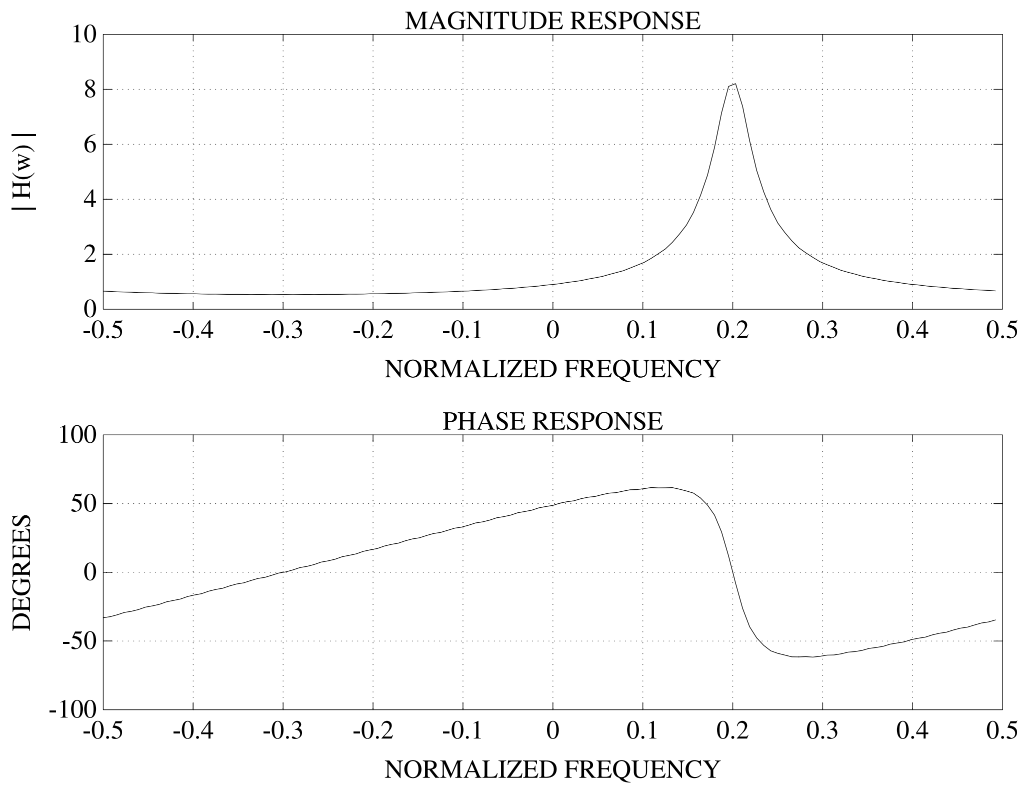

Figure 5.1: Two-panel frequency domain plot made via `subplot`.

rational DTFT:

$$H(e^{j\omega}) = \frac{\displaystyle\sum_{\ell=0}^{N_b} b_\ell e^{-j\omega\ell}}{\displaystyle\sum_{k=0}^{N_a} a_k e^{-j\omega k}}$$

Actually, there are three versions of the `freqz` function:

```
[ H, W ] = freqz( b, a, n )
[ H, W ] = freqz( b, a, n, 'whole' )
[ H, W ] = freqz( b, a, W )
```

The first line is the case described above with one extension: two outputs can be returned—the frequency response vector `H`, and a list of the radian frequencies `W` created from the uniform sampling of the interval $[0, \pi)$ by the `n` frequency samples. The second version has a fourth argument that changes the frequency interval to the "whole" unit circle $[0, 2\pi)$. The last case allows the user to input the vector frequencies where the frequency response is to be evaluated, so options like logarithmic spacing (`logspace`) could be done.

The group delay function (`grpdelay`) follows the same syntax as `freqz`. To plot magnitude and phase and group delay of a filter:

```
  [H, W] = freqz(b, a, n);  %-- n equally spaced freqs
  Hmag   = abs(H);
Hphase = angle(H);
  Hgroup = grpdelay(b, a, n);
  semilogy(W, Hmag)      %----- log y vs. linear x
plot(W, Hphase)
plot(W, Hgroup)
```

In version 3 of the Signal Processing Toolbox, the `freqz` function has an option so that it plots the magnitude and phase when called with no output arguments.

The following table summarizes the all functions available for spectrum analysis. The function `freqs` is a continuous-time version of `freqz`. In addition to the `fft` and `freqz` functions, there is also a statistically based spectrum estimator called `spectrum`.

Fourier & Spectrum Analysis	
`dftmtx`	discrete Fourier transform matrix coefficients
`fft`	Fast Fourier Transform (FFT)
`fftshift`	swap halves of vectors—put DC in the middle
`freqs`	Laplace transform frequency response $B(s)/A(s)$
`freqz`	z-transform frequency response $B(z)/A(z)$
`grpdelay`	group delay
`ifft`	inverse FFT
`specplot`	plot output of spectrum function
`spectrum`	Welch method of power spectrum estimation

5.3.4 Windows

Windows are needed in spectrum analysis and in filter design. The following table shows the ones in the signal processing tool box; most are named for the person who first published the particular window.

Windows by Name		
`boxcar`	rectangular window—all ones	
`bartlett`	nearly the same as `triang`	
`blackman`	`hamming`	`kaiser`
`chebwin`	`hanning`	`triang`

5.3.5 filter

The filtering of data is a primary activity in DSP. The implementation of an function to do filtering offers many possiblities in MATLAB, but the most basic is the `filter` function which will implement either an FIR or IIR digital filter. Since `filter` is a built-in function it is quite efficient; in fact, the `conv` function is based on `filter`. However, if a long FIR is to be implemented, it is better to use an FFT-based method such as `fftfilt` to compute the convolution. Note: `filter` is not aware of signal matrices; it can only filter one vector per call.

Filter Analysis and Implementation	
`fftfilt`	overlap-add filter implementation
`filter`	direct-form IIR filter implementation
`filtfilt`	zero-phase version of `filter`

`y = filter(b, a, x)` filters the data in vector `x` with the filter described by coefficient vectors `a` and `b` to create the filtered data vector `y`. The operation performed internally by `filter` is described in the *time-domain* by a set of difference equations (executed in the order shown from top to bottom): *(give Fig for TDF-2)*

$$\begin{aligned}
y[n] &= b_1 x[n] + v_1[n-1] \\
v_1[n] &= b_2 x[n] + v_2[n-1] - a_2 y[n] \\
\vdots &= \quad \vdots \qquad\qquad \vdots \\
v_{\ell-1}[n] &= b_\ell x[n] + v_\ell[n-1] - a_\ell y[n] \\
v_\ell[n] &= b_{\ell+1} x[n] - a_{\ell+1} y[n]
\end{aligned}$$

where $\ell = \max(na, nb)$. This is the *Transposed Direct Form II* structure. The input-output description of this filtering operation in the z-transform domain is a rational transfer function:

$$Y(z) = \frac{b(1) + b(2)z^{-1} + \cdots + b(nb+1)z^{-nb}}{1 + a(2)z^{-1} + \cdots + a(na+1)z^{-na}} X(z)$$

If $a(1) \neq 1$, the filter coefficients are normalized by $a(1)$. If $a(1) = 0$, the input is in error.

When used with two left hand arguments, `filter` returns the final values of the states:

```
[y,vfinal] = filter( b, a, x )
```

When used with an extra right hand argument, initial conditions for the states are specified:

```
y = filter( b, a, x, vinit )
```

The size of the initial/final condition vector is $\max(na, nb)$.

`y = filtfilt(b,a,x)` performs zero-phase digital filtering by processing the input data in both the forward and reverse directions, see problem 5.39 in [Oppenheim & Schafer]. After filtering in the forward direction, the filtered sequence is reversed and run back through the filter. The resulting sequence has precisely zero phase distortion and double the filter order. Care is taken to minimize startup and ending transients by matching initial conditions. See the M-file for details.

Example: Perform the convolution of `h` and `x` by setting the `a` vector equal to 1. The input signal `x` must be zero padded to get the entire output of the convolution.

```
y = filter( h, 1, [x zeros(1, length(h)-1)] );
```

NOTE: this is how the `conv` function actually works.

5.3.6 Filter Design

The design of frequency-selective digital filters is a primary activity in DSP. MATLAB offers many functions to perform the classic design methods for both finite impulse response (FIR) and infinite impulse response (IIR) filters.

Band Edge Frequency Scaling

The cutoff frequencies in all MATLAB filter design functions are normalized to π. Thus a vector such as `ff = [ 0 0.4 0.6 1]` specifies band edge frequencies at $\omega = \{0, 0.4\pi, 0.6\pi, \pi\}$.

Filter Design Steps

The user must do three things:

1. Make up the filter specifications in terms of band edges and desired values.
2. Select the filter order using a MATLAB to predict the order from the specs. For example, `ellipord` will predict the order of an elliptic filter.
3. Do the actual design. Call a function like `ellip` which in turn will first call `ellipap` to generate the analog prototype and then call `bilinear` to transform the filter from analog to digital. Since functions exist for the design of analog prototypes, MATLAB can also design the classical analog filters.

Give a reference.

Need a Detailed Example

Need a Filter Design Exercise

IIR Filter Design

For the most part, IIR digital filter design relies on the transformation of “classic” analog filter, e.g., Chebyshev, Elliptic, etc. The filter design process involves several steps: first the filter order must be estimated from the given specs, then the analog filter is determined, next the analog filter is transformed to a digital filter via the bilinear transformation, and finally the digital filter might have to be converted from a lowpass design to another form such as bandpass.

IIR Digital Filter Design	
`butter`	Butterworth digital filter design
`buttord`	Butterworth filter order selection
`cheby1`	Chebyshev type I
`cheb1ord`	estimate Chebyshev filter order
`cheby2`	Chebyshev type II
`cheb2ord`	estimate Chebyshev filter order
`ellip`	elliptic (Cauer) filter design
`ellipord`	estimate elliptic filter order
`prony`	Prony's time-domain IIR filter design
`yulewalk`	Yule-Walker filter design

Analog Lowpass Filter Prototypes	
`buttap`	Butterworth filter prototype
`cheb1ap`	Chebyshev type I filter proto (passband ripple)
`cheb2ap`	Chebyshev type II filter proto (stopband ripple)
`ellipap`	elliptic filter prototype

Filter Transformations (IIR)	
`bilinear`	bilinear transformation with optional pre-warping
`lp2bp`	lowpass to bandpass analog filter transformation
`lp2bs`	lowpass to bandstop
`lp2hp`	lowpass to highpass
`lp2lp`	lowpass to lowpass
`ss2zp`	state-space to zero-pole (rational) conversion
`tf2ss`	transfer function to state-space
`zp2ss`	zero-pole to state-space

FIR Filter Design

The Kaiser window produces reasonably good FIR filters, and can be used in `fir1` and `fir2` via

```
h = fir1( L, omega_cutoff, kaiser(L+1,beta) )
```

NOTE: the length of the FIR filter returned from `fir1` is NOT `L`; it is actually `L+1`. The parameter `L` is actually the order of the polynomial that represents the FIR filter. For optimal Chebyshev approximation of FIR linear phase digital filters, use the Remez exchange algorithm, `remez`.

FIR Filter Design	
`fir1`	window based FIR design—lowpass, highpass, bandpass, bandstop
`fir2`	window based FIR design—arbitrary response
`remez`	Parks-McClellan optimal FIR filter design (Remez exchange algorithm)
`remezord`	(version 3 SP only) predict P-M FIR filter order

Need an example

5.3.7 Statistical Signal Processing Functions

Many practical signal processing simulations must use models of additive noise. The `rand` matrix generator can produce such signals. The probability distribution of the noise generator can be either unifrom or Gaussian. (In version 4, separate functions are provided for Gaussian (`randn`) and uniform (`rand`) random distributions. Related functions for processing random signals are listed in the table below. The `spectrum` function implements the method of averaging modified periodograms to obtain a well-behaved power spectrum estimate of a noisy signal.

Statistical Functions on Signals	
`detrend`	linear trend removal
`hist`	histogram
`mean`	average value
`median`	middle value
`rand`	Uniform (or Gaussian) random noise `rand('normal')` sets `rand` to Gaussian
`rands` `specplot`	(version 4 only) Gaussian random noise plot output of spectrum function
`spectrum`	Welch method of power spectrum estimation
`std`	standard deviation

Correlation and Convolution	
`conv`	convolution
`corrcoef`	correlation coefficients
`cov`	covariance matrix (sum of outer products)
`deconv`	deconvolution (polynomial division)
`xcorr`	cross-correlation function
`xcov`	covariance function

5.3.8 Advanced DSP Functions

Signal Modeling	
`invfreqs`	analog filter fit to frequency response
`invfreqz`	discrete filter fit to frequency response
`prony`	Prony's discrete filter fit to time response

Decimation and Interpolation	
`decimate`	Lowpass FIR decimation
`interp`	Lowpass interpolation
`spline`	cubic spline interpolation

2-D Signal Processing	
`conv2`	2-D convolution
`fft2`	2-D FFT
`fttshift`	swap quadrants—put origin in the middle
`ifft2`	inverse 2-D FFT
`xcorr2`	2-D cross-correlation

Cepstrum Processing	
`cceps`	complex cepstrum
`hilbert`	Hilbert transform
`rceps`	real cepstrum & min phase reconstruction
`unwrap`	unwrap phase

5.3.9 Miscellaneous Utility Functions

These functions do not fit any classification, so they are lumped together as miscellaneous odds and ends.

Signal Processing Utilities	
`abs`	magnitude of complex signal (absolute value)
`angle`	phase angle of complex signal
`convmtx`	convolution matrix
`cplxpair`	order vector into complex pairs
`cumsum`	cumulative sum (i.e., convolve with $u[n]$)
`diff`	first difference (i.e., convolve with $\delta[n] - \delta[n-1]$)
`sawtooth`	generate a triangle wave function
`square`	generate a square wave function

Chapter 6

Control Toolbox

6.1 Feedback System

Need block diagram of feedback system with transfer function

$$\frac{G(s)}{1 + KG(s)H(s)} \tag{6.1}$$

This does depend on where the gain K is located and on the specific forward gain and loop gain.

6.1.1 Transfer Functions

6.1.2 Partial Fractions

6.2 State-Space Representation

A single-input, single-output system (SISO)

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{A}\mathbf{x} + \mathbf{b}u \\ y &= \mathbf{C}\mathbf{x} + \mathbf{d}u \end{aligned}$$

Therefore, in MATLAB we must define the four matrices **A**, **b**, **C** and **d** to specify the system.

Dimension of **x** is same as number of poles.

Multi-Input, Multi-Output

supported with an arg that tells which input to use. Does it have to be single output?

6.2.1 Transfer Functions from State-Space

6.2.2 Conversion Between Forms

Various utilities such `ss2tf`, `tf2ss`,

6.2.3 Digital Control

A single-input, single-output system (SISO)

$$\begin{aligned} \mathbf{x}[n+1] &= \mathbf{A}\mathbf{x}[n] + \mathbf{b}u[n] \\ y[n] &= \mathbf{C}\mathbf{x}[n] + \mathbf{d}u[n] \end{aligned}$$

6.3 Time-Domain Response: Continuous-Time

Need a simulation.

6.3.1 Simulated Response of Differential Equation

Simulation choice of delta t. ODE solvers

6.3.2 Step Response

There is a functions called `step`

6.3.3 Impulse Response

Find step response of $sH(s)$. May have to attach an additional state variable

6.3.4 Ramp Response

Again, use `step` but apply to $H(s)/s$

6.4 Time-Domain Response: Discrete-Time

6.4.1 Digital Filtering

Refer to DSP for explanation of `filter`. Plotting with comb or stem

6.4.2 Step Response

Is there a `step` function? NO you probably `cumsum` the impulse response.

6.4.3 Impulse Response

Do this one directly with `filter`

6.5 Frequency Response

`freqz` for digital? `bode` for continuous.

6.5.1 Bode Plots

`bode` is a lot like `freqz`

6.6 Nyquist Plots

Explain polar notation. Say a wee bit about phase margin? i.e., why are you making a Nyquist plot? axis('square') to see true unit circle.

6.7 Root Locus

Need numerator and denom of $G(s)H(s)$ in

$$\frac{G(s)}{1 + KG(s)H(s)} \tag{6.2}$$

Then `rlocus` will vary K and plot roots. Sometimes its choice of K may be flaky Also, it probably does lots of rooting internally.

6.7.1 Stability Tests

Just root. But maybe the toolbox has some built in.

6.8 Linear Quadratic State Estimation

This should be very brief. Just indicate that the capability exists.

Then do we make lists of the rest of it?

6.9 Optimization Toolbox

6.9.1 Least-Squares Inverse

6.9.2 FMINS

Doing an `feval` to pass function definition.

6.9.3 Non-Linear Minimization

6.9.4 Non-negative Least-Squares

6.9.5 Linear Programming

Chapter 7

Symbolic Toolbox

Nothing here (27-March-94)

Chapter 8

Quick Reference Guide

8.1 Summary of Available Help Screens

The help messages obtained using the `help` command for many commomly used functions are included in section 4.3. To see the help message for any particular function, such as the FFT, type

```
>> help fft
```

For a listing of the functions for which help is available, type

```
>> help
```

The following list describes the built-in MATLAB functions and M-files for which help is available (not including any site-specific functions that may also be available):

```
Matlab built-in functions:

help     &        clock    exist    hess     memory   rcond    string
[        |        conj     exit     hold     mesh     real     subplot
]        ~        contour  exp      home     meta     relop    sum
(        abs      cos      expm     ident    min      rem      svd
)        all      cumprod  eye      if       nan      return   tan
.        ans      cumsum   feval    imag     nargin   round    text
,        any      dc2sc    fft      inf      norm     save     title
;        acos     delete   filter   input    ones     sc2dc    type
%        asin     det      find     inv      pack     schur    what
!        atan     diag     finite   isnan    pause    script   while
:        atan2    diary    fix      isstr    pi       semilogx who
'        axis     dir      floor    keyboard plot     semilogy xlabel
+        balance  disp     flops    load     polar    setstr   ylabel
-        break    echo     for      log      polyline shg      zeros
*        casesen  eig      format   loglog   polymark sign
\        ceil     else     fprintf  logop    prod     sin
/        chdir    elseif   function ltifr    prtsc    size
^        chol     end      getenv   ltitr    qr       sort
<        clc      eps      ginput   lu       quit     sprintf
>        clear    error    global   magic    qz       sqrt
```

```
=        clg      eval     grid     max      rand     startup

Strike any key to continue  ...
```

Continuing, produces the following list of M-files.

```
Strike any key to continue ...Directory of M-files in \matlab

bench    demo     demolist info     intro    matlab   readme

acosh    cosh     fitfun   hex2num  log10    nersolv  quadstp  std
angle    cov      fliplr   hilb     logm     nestop   quiver   strcmp
asinh    cplxpair flipud   hist     logspace nnls     rank     subspace
atanh    date     fmin     humps    mean     null     rat      table1
backsub  dec2hex  fmins    ieee     median   num2str  ratmovie table2
bar      deconv   fminstep ifft     membrane ode23    reshape  tanh
bessel   diff     fplot    ifft2    menu     ode45    resi2    toeplitz
bessela  eigmovie fsolve   int2str  meshdom  orth     residue  trace
besselh  ellipj   funm     interp1  mkpp     pascal   roots    tril
besseln  ellipk   fzero    interp2  nebroyuf pinv     roots1   triu
blanks   erf      gallery  interp3  nechdcmp poly     rose     unmkpp
cdf2rdf  errorbar gamma    interp4  neconest polyder  rot90    unwrap
clabel   etime    gammac   inverf   nefdjac  polyfit  rref     vander
compan   expm1    gammai   invhilb  nefn     polyval  rrefmovi vdpol
compass  expm2    gradient isempty  neinck   polyvalm rsf2csf  why
computer expm3    gtext    kron     nelnsrch ppval    sinh
cond     feather  hadamard laguer   nemodel  quad     spline
conv     fft2     hankel   length   neqrdcmp quad8    sqrtm
corrcoef fftshift hex2dec  linspace neqrsolv quad8stp stairs
```

And continuing further displays a list of M-files that are especially useful for signal processing.

```
Directory of M-files in \matlab\signal

abcdchk  cheb1ap  decimate fir1     invfreqs nextpow2 series   xcorr2
bartlett cheb1ord denf     fir2     invfreqz numf     specplot xcov
bilinear cheb2ap  detrend  freqs    kaiser   polystab spectrum yulewalk
blackman cheb2ord dftmtx   freqz    kratio   prony    square   zp2ss
boxcar   chebwin  ellip    grpdelay lp2bp    rceps    ss2zp
buttap   cheby1   ellipap  hamming  lp2bs    rcunwrap tf2ss
butter   cheby2   ellipord hanning  lp2hp    readme2  triang
buttord  conv2    fftfilt  hilbert  lp2lp    remez    vratio
cceps    convmtx  filtfilt interp   nargchk  remezdd  xcorr
```

There will also be a list of the M-files in your local area. The list in your local area can also be obtained by typing `what`.

8.2 Summary of Frequently Used Commands

`A'`	complex conjugate transpose of A
`A.'`	transpose of A
`A+B`	matrix sum
`A*B`	matrix product
`A/B`	right inverse, same as AB^{-1}, if B is invertible
`A\B`	left inverse, same as $A^{-1}B$, if A is invertible
`x = A\b`	solve $Ax = b$
	use least-squares pseudo-inverse, if necessary
`A^n`	$A * A * A * \ldots * A$, `n`-fold matrix product
`abs(x)`	returns the magnitude of the elements of x
`bar()`	draws a bar graph
`contour(A)`	contour plot
`demo`	run the built-in MATLAB demos
`det(A)`	determinant of A
`diag(A)`	extract diagonal elements of A and put them in a vector
`eig(A)`	eigenvalues of A, output in a column vector
`[V,D] = eig(A)`	eigen-analysis of A
	diagonal elements of `D` are the eigenvalues
	column vectors of `V` are the eigenvectors
`exp(x)`	is the exponential of the elements of x, e to the x
`eye(n)`	`n` $\times$ `n` identity matrix
`fft(A,n)`	1-D FFT of each column of A, length = `n`
`filter(b,a,x)`	filter $x[n]$ using pole-zero system $H(z) = B(z)/A(z)$
`format compact`	single space between display lines
`format long`	display answers to 16 decimal places, `short` uses 5 places
`freqz(b,a,n)`	frequency response of system $H(z) = B(z)/A(z)$
`hamming(n)`	Hamming window of length `n`
`help` *name*	ask for help on the function called *name*
`hold on`	hold the plot in order to add more curves
	`hold off` releases the plot
`imag(x)`	returns the imaginary-parts of the elements of x
`inv(A)`	inverse of A
`kaiser(n,alpha)`	Kaiser window
`length(x)`	returns the length of the vector x
`mesh(A)`	3-D mesh plot, using the matrix entries as amplitude
`norm(x)`	2-norm of the vector $\mathbf{x}$, i.e., $\mathbf{x}^H\mathbf{x}$
	also works for a matrix input
`ones(m,n)`	`m` $\times$ `n` matrix of ones, `ones(n)` gives `n` $\times$ `n` matrix
`plot(x,y)`	x–y plot, `plot(y)` takes the x-axis to be indices
`polar(ang,rad)`	polar plot given vectors of angles and radii
`rand(m,n)`	`m` $\times$ `n` matrix of (uniform) random numbers
`rank(A)`	rank of matrix A
`real(x)`	returns the real-parts of the elements of x
`remez()`	Remez exchange method for optimal FIR
	linear phase filter design

`roots(c)`	compute roots of polynomial whose coefficients are c
`round(x)`	rounds the elements of x to the nearest integers
`size(A)`	returns the number of rows and columns in A
`sqrt(x)`	returns the square roots of the elements of x
`svd(A)`	singular value decomposition $A = U^H \Sigma V$
`title(`*string*`)`	title the plot
`triang(n)`	Triangular window
`xlabel(`*string*`)`	label the x-axis
`ylabel(`*string*`)`	label the y-axis
`zeros(m,n)`	`m` × `n` matrix of zeros, `ones(n)` gives `n` × `n` matrix

8.3 Help Screens for Frequently Used Commands

```
>> help ’

’       Matrix transpose. X’ is the complex conjugate transpose of X.
        X.’ is the non-conjugate transpose.
        Quote. ’ANY TEXT’ is a vector whose components are the
        ASCII codes for the characters. A quote within the text
        is indicated by two quotes.

>> help +

+       Addition.
        X + Y adds matrices X and Y.  X and Y must have the same
        dimensions unless one is a scalar. A scalar can be added
        to anything.

>> help *

*       Multiplication.
        X*Y is the matrix product of X and Y.  Any scalar
        (1-by-1 matrix) may multiply anything. Otherwise, the number
        of columns of X must equal the number of rows of Y.

        X.*Y denotes element-by-element multiplication.  X and Y
        must have the same dimensions unless one is a scalar.
        A scalar can be multiplied into anything.

>> help /

/       Right division.
        B/A is the matrix division of A into B, which is roughly the
        same as B*INV(A) , except it is computed in a different way.
        More precisely, B/A = (A’\B’)’. See \.

        B./A denotes element-by-element division.  A and B
        must have the same dimensions unless one is a scalar.
        A scalar can be divided with anything.

        WARNING:  3./A is NOT the same as 3 ./A because in the first
        case the dot is sucked up by the 3 as a decimal point causing
        matrix division, while in the second case the dot is
        associated with the / for elementwise division.

>> help \

\       Left division.
        A\B is the matrix division of A into B, which is roughly the
        same as INV(A)*B , except it is computed in a different way.
        If A is an N-by-N matrix and B is a column vector with N
        components, or a matrix with several such columns, then
        X = A\B is the solution to the equation A*X = B computed by
        Gaussian elimination. A warning message is printed if A is
        badly scaled or nearly singular.  A\EYE(A) produces the
        inverse of A.  If A is an M-by-N matrix with M < or > N and B
        is a column vector with M components, or a matrix with several
        such columns, then X = A\B is the solution in the least
        squares sense to the under- or overdetermined system of
        equations A*X = B. The effective rank, K, of A is determined
        from the QR decomposition with pivoting. A solution X is
        computed which has at most K nonzero components per column. If
        K < N this will usually not be the same solution as PINV(A)*B.
        A\EYE(A) produces a generalized inverse of A.
        See / for the meaning of .\.

>> help ˆ

ˆ       Powers.
        Z = Xˆy is X to the y power if y is a scalar. If y is an
        integer greater than one, the power is computed by repeated
        multiplication. For other values of y the calculation
        involves eigenvalues and eigenvectors.
        Z = xˆY is x to the Y power, if Y is a matrix and x is a
        scalar, computed using eigenvalues and eigenvectors.
        Z = XˆY, where both X and Y are matrices, is an error.

        Z = X.ˆY denotes element-by-element powers.  X and Y
        must have the same dimensions unless one is a scalar.
        A scalar can operate into anything.

>> help abs

ABS     ABS(X) is the absolute value of the elements of X. When
```

```
        X is complex, ABS(X) is the complex modulus (magnitude) of
        the elements of X. See also ANGLE.

>> help bar

 BAR    Bar graph.
        BAR(Y) draws a bar graph of the elements of vector Y.
        BAR(X,Y) draws a bar graph of the elements in vector Y at
        the locations specified in X.  The X-values must be in
        ascending order.  If the X-values are not evenly spaced, the
        interval chosen is not symmetric about each data point.
        Instead, the bars are drawn midway between adjacent X-values.
        The endpoints simply adopt the internal intervals for the
        external ones needed.
        [XX,YY] = BAR(X,Y) does not draw a graph, but returns vectors
        X and Y such that PLOT(XX,YY) is the bar chart.
        See also STAIRS and HIST.

>> help contour

CONTOUR Contour plot.
        CONTOUR(Z) is a contour plot of matrix Z treating the values
        in Z as heights above a plane.  Element Z(1,1) is displayed
        in the upper left corner of the contour.
        CONTOUR(Z,N) draws N contour levels, overriding the default
        automatic value.
        CONTOUR(Z,V) draws LENGTH(V) contour lines at the locations
        specified in vector V.
        CONTOUR(Z,N,X,Y), where X and Y are vectors, specifies the
        X- and Y-axes used on the plot. CONTOUR(Z,V,X,Y) works too.
        CONTOUR(...,’linetype’) draws with the color and linetype
        specified, as in the PLOT command.
        C = CONTOUR(...) returns a two row matrix of contour lines.
        Each contiguous drawing segment contains the value of the
        contour, the number of (x,y) drawing pairs, and the pairs
        themselves.  The segments are appended end-to-end as
            C = [level#1 x1 x2 x3 ... level#2 x2 x2 x3 ...;
                 #pairs1 y1 y2 y3 ... #pairs2 y2 y2 y3 ...]
        See also CLABEL, MESH, GRADIENT, and QUIVER.

>> help demo

        DEMO brings up a menu of the available demonstrations.

>> help det

DET     DET(X) is the determinant of the square matrix X.
```

```
>> help diag

DIAG    If V is a row or column vector with N components,
        DIAG(V,K) is a square matrix of order N+ABS(K) with the
        elements of V on the K-th diagonal. K = 0 is the main
        diagonal, K > 0 is above the main diagonal and K < 0 is
        below the main diagonal. DIAG(V) simply puts V on the
        main diagonal. For example,
        DIAG(-M:M) + DIAG(ONES(2*M,1),1) + DIAG(ONES(2*M,1),-1)
        produces a tridiagonal matrix of order 2*M+1.

        If X is a matrix, DIAG(X,K) is a column vector formed from
        the elements of the K-th diagonal of X. DIAG(X) is the main
        diagonal of X. DIAG(DIAG(X)) is a diagonal matrix.

>> help eig

EIG     Eigenvalues and eigenvectors.
        EIG(X) is a vector containing the eigenvalues of a square
        matrix X.
        [V,D] = EIG(X) produces a diagonal matrix D of
        eigenvalues and a full matrix V whose columns are the
        corresponding eigenvectors so that X*V = V*D.

        [V,D] = EIG(X,’nobalance’) performs the computation with
        balancing disabled, which gives better results for certain
        ill-conditioned problems.

        Generalized eigenvalues and eigenvectors.
        EIG(A,B) is a vector containing the generalized eigenvalues
        of square matrices A and B.
        [V,D] = EIG(A,B) produces a diagonal matrix D of general-
        ized eigenvalues and a full matrix V whose columns are the
        corresponding eigenvectors so that A*V = B*V*D.
>> help exp

EXP     EXP(X) is the exponential of the elements of X, e to the X.

>> help eye

EYE     Identity matrix. EYE(N) is the N-by-N identity matrix.
        EYE(M,N) is an M-by-N matrix with 1’s on the diagonal and
        zeros elsewhere. EYE(A) is the same size as A.

>> help fft
```

```
FFT     FFT(X) is the discrete Fourier transform of vector X.  If the
        length of X is a power of two, a fast radix-2 fast-Fourier
        transform algorithm is used.  If the length of X is not a
        power of two, a slower non-power-of-two algorithm is employed.
        FFT(X,N) is the N-point FFT, padded with zeros if X has less
        than N points and truncated if it has more.
        If X is a matrix, the FFT operation is applied to each column.
        See also IFFT, FFT2, and IFFT2.

>> help filter

FILTER  Digital filter.
        Y = FILTER(B, A, X) filters the data in vector X with the
        filter described by vectors A and B to create the filtered
        data Y.  The filter is a ‘‘Direct Form II Transposed’’
        implementation of the standard difference equation:

        y(n) = b(1)*x(n) + b(2)*x(n-1) + ... + b(nb+1)*x(n-nb)
                         - a(2)*y(n-1) - ... - a(na+1)*y(n-na)

        [Y,Zf] = FILTER(B,A,X,Zi) gives access to initial and final
        conditions, Zi and Zf, of the delays.
        See also FILTFILT.

>> help format

FORMAT  Set output format. All computations are done in double
        precision. FORMAT may be used to switch between different
        display formats as follows:
          FORMAT         Default. Same as SHORT.
          FORMAT SHORT   Scaled fixed point format with 5 digits.
          FORMAT LONG    Scaled fixed point format with 15 digits.
          FORMAT SHORT E Floating point format with 5 digits.
          FORMAT LONG E  Floating point format with 15 digits.
          FORMAT HEX     Hexadecimal format.
          FORMAT +       Compact format. The symbols +, - and blank
                         are printed for positive, negative and zero
                         elements. Imaginary parts are ignored.
          FORMAT BANK    Fixed format for dollars and cents.
          FORMAT COMPACT Suppress extra line-feeds.
          FORMAT LOOSE   Puts the extra line-feeds back in.

>> help freqz

 FREQZ  Z-transform digital filter frequency response.  When N is an
        integer, [H,W] = FREQZ(B,A,N) returns the N-point frequency
        vector W and the N-point complex frequency response vector G
        of the filter B/A:
                                    -1                 -nb
             jw  B(z)   b(1) + b(2)z + .... + b(nb+1)z
          H(e) = ---- = ----------------------------
                                    -1                 -na
                 A(z)    1   + a(2)z + .... + a(na+1)z
        given numerator and denominator coefficients in vectors B and
        A. The frequency response is evaluated at N points equally
        spaced around the upper half of the unit circle. To plot
        magnitude and phase of a filter:
                  [h,w] = freqz(b,a,n);
                  mag = abs(h);  phase = angle(h);
                  semilogy(w,mag), plot(w,phase)
        FREQZ(B,A,N,'whole') uses N points around the whole unit circle.
        FREQZ(B,A,W) returns the frequency response at frequencies
        designated in vector W, normally between 0 and pi. (See
        LOGSPACE to generate W).  See also YULEWALK, FILTER, FFT,
        INVFREQZ, and FREQS.

>> help hamming

 HAMMING   HAMMING(N) returns the N-point Hamming window.

>> help help

HELP    HELP lists all help topics, starting with the built-in
        functions and continuing with the M-files in the various
        directories on disk.
        HELP topic gives help on the topic.
        HELP HELP obviously prints this message.
        HELP word where 'word' is a filename, displays the first
        comment lines in the M-file 'word.m'.

>> help hold

HOLD    Hold the current graph on the screen. Subsequent PLOT com-
        mands will add to the plot, using the already established
        axis limits, and retaining the previously plotted curves.
        HOLD ON turns on holding, HOLD OFF turns it off, and HOLD,
        by itself, toggles the HOLD state.

>> help imag

IMAG    IMAG(X) is the imaginary part of X.
        See I or J to enter complex numbers.
        Imaginary numbers are not entered into MATLAB using the
        letters I or J as might be expected. This is because I and
        J are often used as indices. To enter imaginary numbers,
        SQRT(-1) is used.
        For example, 3+2i is entered as 3+2*sqrt(-1). Alternatively,
        this could be done as i = SQRT(-1); 3+2*i.

>> help inv

INV     INV(X) is the inverse of the square matrix X. A warning
        message is printed if X is badly scaled or nearly
        singular.

>> help kaiser

 KAISER  KAISER(N,beta) returns the BETA-valued N-point Kaiser window.

>> help length

LENGTH  LENGTH(X) returns the length of vector X. It is equivalent
        to MAX(SIZE(X)).

>> help mesh

MESH    Mesh surface. MESH(Z) produces a 3-dimensional mesh surface
        ‘‘picture’’ of matrix Z using the values in Z as heights above
        a plane. See MESHDOM to plot functions of two variables.

        MESH(Z,M) specifies a view-point. The two-element vector M =
        [AZ EL] contains AZ, the azimuth or horizontal rotation, and
        EL, the vertical elevation (both in degrees). Azimuth revolves
        about the z-axis, with positive values indicating
        counter-clockwise rotation of the view-point (clockwise
        rotation of the object). Positive values of elevation
        correspond to moving above the object; negative values move
        below. Here are some examples:

                EL = 90 is directly overhead.
                M = [0 0] looks directly up the first column
                          of Z, from the Z(m,1) element.
                AZ = 180 is behind the matrix.
                M = [-37.5 30] is the default.

        MESH(Z,S) and MESH(Z,M,S) control the scale factors used to
        set the X, Y and Z axes. The vector S is defined as S = [sx sy sz],
        where the three scalars, relative to each other, set the size
        of the object in each of the three dimensions. See also CONTOUR.

>> help norm
```

```
NORM    For matrices..
         NORM(X) is the largest singular value of X
         NORM(X,1) is the 1-norm of X, the largest column sum,
                        MAX(SUM(ABS(REAL(X))+ABS(IMAG(X)))
         NORM(X,2) is the same as NORM(X)
         NORM(X,inf) is the infinity norm of X, the largest row sum.
                        MAX(SUM(ABS(REAL(X'))+ABS(IMAG(X')))
         NORM(X,'fro') is the F-norm, SQRT(SUM(DIAG(X'*X)))
        For vectors..
         NORM(V,P) = SUM(ABS(V)^P)^(1/P)
         NORM(V) = NORM(V,2)
         NORM(V,inf) = MAX(ABS(V))
         NORM(V,-inf) = MIN(ABS(V))

>> help ones

ONES    All ones. ONES(N) is an N-by-N matrix of ones. ONES(M,N)
        is an M-by-N matrix of ones. ONES(A) is the same size as
        A and all ones.

>> help plot

PLOT    Plot vectors or matrices. PLOT(X,Y) plots vector X versus
        vector Y. If X or Y is a matrix, then the vector is plotted
        versus the rows or columns of the matrix, whichever lines
        up. PLOT(X1,Y1,X2,Y2) is another way of producing multiple
        lines on the plot. PLOT(X1,Y1,':',X2,Y2,'+') uses a
        dotted line for the first curve and the point symbol +
        for the second curve. Other line and point types are:

           solid   -      point  .      red       r
           dashed  --     plus   +      green     g
           dotted  :      star   *      blue      b
           dashdot -.     circle o      white     w
                          x-mark x      invisible i
                                        arbitrary c1, c15, etc.

        PLOT(Y) plots the columns of Y versus their index. PLOT(Y)
        is equivalent to PLOT(real(Y),imag(Y)) if Y is complex.
        In all other uses of PLOT, the imaginary part is ignored.
        See SEMI, LOGLOG, POLAR, GRID, SHG, CLC, CLG, TITLE, XLABEL
        YLABEL, AXIS, HOLD, MESH, CONTOUR, SUBPLOT.

>> help polar

POLAR   POLAR(THETA, RHO) makes a plot using polar coordinates of
```

```
        the angle THETA, in radians, versus the radius RHO.
        See GRID for polar grid lines and PLOT for how to obtain
        multiple lines and different line-types.

>> help rand

RAND    Random numbers and matrices. RAND(N) is an N-by-N matrix
        with random entries. RAND(M,N) is an M-by-N matrix with
        random entries. RAND(A) is the same size as A. RAND with
        no arguments is a scalar whose value changes each time
        it is referenced.

        Ordinarily, random numbers are uniformly distributed in
        the interval (0.0,1.0). RAND('normal') switches to a
        normal distribution with mean 0.0 and variance 1.0.
        RAND('uniform') switches back to the uniform distribution.
        RAND('dist') returns a string containing the current
        distribution, either 'uniform' or 'normal'.

        RAND('seed') returns the current value of the seed for the
        generator. RAND('seed',n) sets the seed to n.
        RAND('seed',0) resets the seed to 0, its value when MATLAB
        is first entered.

>> help rank

 RANK   Rank.  K = RANK(X) is the number of singular values  of   X
        that are larger than MAX(SIZE(X)) * NORM(X) * EPS.
        K = RANK(X,tol) is the number of singular values of  X that
        are larger than tol .

>> help real

REAL    REAL(X) is the real part of X. See also IMAG and ABS.

>> help remez

 REMEZ  Parks-McClellan optimal equirriple FIR filter design.
        B = REMEZ(N,F,M) designs an N'th order FIR digital filter,
        with the frequency response specified by vectors F and M,
        and returns the filter coefficients in length N+1 vector B.
        Vectors F and M specify the frequency and magnitude
        breakpoints for the filter such that PLOT(F,M) would show a
        plot of the desired frequency response. The elements of M must
        appear in equal-valued pairs.  The frequencies in F must be
        between 0.0 < F < 1.0, with 1.0 corresponding to half the
        sample rate. They must be in increasing order, start with 0.0,
        and end with 1.0.
        B = REMEZ(N,F,M,W) uses vector W to specify weighting in each
        of the pass or stop bands in vectors F and M.
        B = REMEZ(N,F,M,FTYPE) or B = REMEZ(N,F,M,W,FTYPE), where
        FTYPE is the string 'Hilbert' or 'differentiator', designs
        Hilbert transformers or differentiators, respectively.  For
        the Hilbert case, the lowest frequency should not be 0.

        See also FIR1, FIR2, BUTTER, CHEBY1, CHEBY2, YULEWALK, FREQZ
        and FILTER.

>> help roots

 ROOTS  Find polynomial roots. ROOTS(C) computes the roots of the
        polynomial whose coefficients are the elements of the
        vector C. If C has N+1 components, the polynomial is
        C(1)*X^N + ... + C(N)*X + C(N+1). See ROOTS1 and POLY.

>> help round

ROUND   ROUND(X) rounds the elements of X to the nearest
        integers.

>> help size

SIZE    If X is an M-by-N matrix, then SIZE(X) is [M, N].
        SIZE can also be used with a multiple assignment,
                [M, N] = SIZE(X).

>> help sqrt

SQRT    SQRT(X) is the square root of X. Complex results are
        produced if X is not positive. See also SQRTM.

>> help svd

SVD     Singular value decomposition. [U,S,V] = SVD(X) produces a
        diagonal matrix S , of the same dimension as X and with
        nonnegative diagonal elements in decreasing order, and
        unitary matrices U and V so that X = U*S*V'.
        By itself, SVD(X) returns a vector containing the singular
        values.
        [U,S,V] = SVD(X,0) produces the ``economy size''
        decomposition. If X is m-by-n with m > n, then only the
        first n columns of U are computed and S is n-by-n.

>> help title
```

```
TITLE   TITLE('text') writes the text at the top of the current
        plot as a title.

>> help triang

 TRIANG   TRIANG(N) returns the N-point triangular window.

>> help xlabel

XLABEL  XLABEL('text') writes the text on the current plot beneath
        the x-axis.

>> help ylabel

YLABEL  YLABEL('text') writes the text on the current plot beside
        the y-axis.

>> help  zeros

ZEROS   All zeros. ZEROS(N) is an N-by-N matrix of zeros.
        ZEROS(M,N) is an M-by-N matrix of zeros. ZEROS(A) is the
        same size as A and all zeros.
```

Chapter 9

MATLAB Commands by Function

General	
help	help facility
demo	run demonstrations
who	list variables in memory
what	list M-files on disk
size	row and column dimensions
length	vector length
clear	clear workspace
computer	type of computer
Ctrl-C	local abort
quit	terminate program
exit	same as quit

Matrix Operators		Array Operators	
`+`	addition	`+`	addition
`-`	subtraction	`-`	subtraction
`*`	multiplication	`.*`	multiplication
`/`	right division	`./`	right division
`\`	left division	`.\`	
`ˆ`	power	`.ˆ`	power
`′`	conjugate transpose	`.′`	transpose

Relational and Logical Operators			
`<`	less than	`&`	AND
`<=`	less than or equal	`|`	OR
`>`	greater than	`˜`	NOT
`>=`	greater than or equal		
`==`	equal		
`˜=`	not equal		

Special Characters	
`=`	assignment statement
`[`	used to form vectors and matrices
`]`	see `[`
`(`	arithmetic expression precedence
`)`	see `(`
`.`	decimal point
`...`	continue statement to the next line
`,`	separate subscripts and function arguments
`;`	end rows, suppress printing
`%`	comments
`:`	subscripting, vector generation
`!`	execute operating system command

Special Values	
ans	answer when expression is not assigned
eps	floating point precision
pi	π
i,j	$\sqrt{-1}$
inf	∞
NaN	Not-a-Number
clock	wall clock
date	date
flops	floating point operation count
nargin	number of function input arguments
nargout	number of function output arguments

Text and Strings	
abs	convert string to ASCII value
eval	evaluate text macro
num2str	convert number to string
int2str	convert integer to string
setstr	set flag indicating matrix is a string
sprintf	convert number to a string
isstr	detect string variables
strcmp	compare string values
hex2num	convert hexadecimal string to number

Graph Paper	
plot	linear X-Y plot
loglog	loglog X-Y plot
semilogx	semi-log X-Y plot
semilogy	semi-log X-Y plot
polar	polar plot
mesh	3-dimensional plot
contour	contour plot
meshdom	domain for mesh plots
bar	bar charts
stairs	stairstep graphs
errorbar	add errorbars

Graph Annotation	
title	plot title
xlabel	x-axis label
ylabel	y-axis label
grid	draw grid lines
text	arbitrarily positioned text
gtext	mouse-positioned text
ginput	graphics input

Graph Window Control	
axis	manual axis scaling
hold	hold plot on screen
shg	show graph screen
clg	clear graph screen
subplot	split graph window

Command Window	
clc	clear command screen
home	home cursor
format	set output display format
disp	display matrix or text
fprintf	print formatted number
echo	enable command echoing

Control Flow	
if	conditionally execute statement
elseif	used with **if**
else	used with **if**
end	terminate **if**, **for**, **while**
for	repeat statements a number of times
while	do while
break	break out of **for** and **while** loops
return	return from functions
pause	pause until key press

Programming and M-files	
input	get numbers for keyboard
keyboard	call keyboard as M-file
error	display error message
function	define function
eval	interpret text in variables
feval	evaluate function given by string
echo	enable command echoing
exist	check if variables exist
casesen	set case sensitivity
global	define global variables
startup	startup M-file
getenv	get environment string
menu	select item from menu
etime	elapsed time

Disk Files	
chdir	change current directory
delete	delete file
diary	diary of the session
dir	directory of files on disk
load	load variables from file
save	save variables on file
type	list function or file
what	show M-files on disk
fprintf	write to a file
pack	compact memory via **save**

Relational and Logical Functions	
any	logical conditions
all	logical conditions
find	find array indices of logical values
exist	check if variables exist
isnan	detect NaN's
finite	detect infinities
isempty	detect empty matrices
isstr	detect string variables
strcmp	compare string variables

Trigonometric Functions	
sin	sine
cos	cosine
tan	tangent
asin	arcsine
acos	arccosine
atan	arctangent
atan2	four quadrant arctangent
sinh	hyberbolic sine
cosh	hyberbolic cosine
tanh	hyperbolic tangent
ashinh	hyperbolic arcsine
acosh	hyperbolic arccosine
atanh	hyperbolic arctangent

Elementary Math Functions	
abs	absolute value or complex magnitude
angle	phase angle
sqrt	square root
real	real part
imag	imaginary part
conj	complex-conjugate
round	round to nearest integer
fix	round towards zero
floor	round towards $-\infty$
ceil	round towards ∞
sign	signum function
rem	remainder or modulus
exp	exponential base e
log	natural logarithm
log10	log base 10

Special Functions	
bessel	Bessel function
gamma	complete and incomplete gamma functions
rat	rational approximation
erf	error function
inverf	inverse error function
ellipk	complete elliptic integral of the first kind
ellipj	Jacobian elliptic functions

Polynomials	
poly	characteristic polynomial
roots	polynomial roots - companion matrix method
roots1	polynomial roots - Laguerre's method
polyval	polynomial evaluation
polyvalm	matrix polynomial evaluation
conv	multiplication
deconv	division
residue	partial-fraction expansion
polyfit	polynomial curve fitting

Matrix Manipulation	
rot90	rotation
fliplr	flip matrix left-to-right
flipud	flip matrix up-and-down
diag	extract or create diagonal
tril	lower triangular part
triu	upper triangular part
reshape	reshape
.′	transposition
:	general rearrangement

Special Matrices	
compan	companion
diag	diagonal
eye	identity
gallery	esoteric
hadamard	Hadamard
hanke	Hankel
hilb	Hilbert
invhilb	inverse Hilbert
linspace	linearly spaced vectors
logspace	logarithmically spaced vectors
magic	magic square
meshdom	domain for mesh plots
ones	constant
rand	random elements
toeplitz	Toeplitz
vander	Vandermonde
zeros	zero

Matrix Condition	
cond	condition number in 2-norm
norm	1-norm, 2-norm, F-norm, ∞-norm
rank	rank
rcond	condition estimate

Decompositions and Factorizations	
balance	balanced form
backsub	backsubstitution
cdf2rdf	convert complex-diagonal to real-diagonal
chol	Cholesky factorizion
eig	eigenvalues
hess	Hessenberg form
inv	inverse
lu	factors from Gaussian elimination
nnis	nonnegative least-squares
null	null space
orth	orthogonalization
pinv	pseudoinverse
qr	orthogonal-triangular decomposition
qz	QZ algorithm
rref	reduced row echelon form
rsf2csf	convert real-schur to complex-schur
svd	singular value decomposition

Elementary Matrix Functions	
expm	matrix exponential
logm	matrix logarithm
sqrtm	matrix square root
funm	arbitrary matrix function
poly	characteristic polynomial
det	determinant
trace	trace
kron	Kronecker tensor product

Interpolation	
spline	cubic spline
table1	1-D table look-up
table2	2-D table look-up

Differential Equation Solution	
ode23	2nd/3rd order Runge-Kutta method
ode45	4th/5th order Runge-Kutta-Fehlberg method

Numerical Integration	
quad	numerical function integration
quad8	numerical function integration

Nonlinear Equations and Optimization	
fmin	minimum of a function of one variable
fmins	minimum of a multivariable function (unconstrained nonlinear optimiztion)
fsolve	solution to a system of nonlinear equations (zeros of a multivariable function)
fzero	zero of a function of one variable

Columnwise Data Analysis	
max	maximum value
min	minimum value
mean	mean value
median	median value
std	standard deviation
sort	sorting
sum	sum of elements
prod	product of elements
cumsum	cumulative sum of elements
cumprod	cumulative product of elements
diff	approximate derivatives
hist	histogram
corrcoef	correlation coefficients
cov	covariance matrix
cplxpair	reorder into complex pairs

Chapter 10

Signal Processing Functions by Group

Analog Lowpass Filter Prototypes	
buttap	Butterworth filter prototype
cheb1ap	Chebyshev type I filter prototype (passband ripple)
cheb2ap	Chebyshev type II filter prototype (stopband ripple)
ellipap	elliptic filter prototype

IIR Filter Design	
butter	Butterworth filter design
buttord	butterworth filter order selection
cheby1	Chebyshev type I filter design
cheby1ord	Chebyshev type I filter order selection
cheby2	Chebyshev type II filter design
cheby2ord	Chebyshev type II filter order selection
ellip	elliptic filter design
ellipord	elliptic filter order selection
prony	Prony’s time-domain IIR filter design
yulewalk	Yule-Walker filter design

FIR Filter Design	
fir1	window based FIR filter design - low, high, band, stop
fir2	window based FIR filter design - arbitrary response
remez	Parks-McClellan optimal FIR filter design

Filter Transformations	
bilinear	bilinear transformation with optional prewarping
lp2bp	lowpass to bandpass analog filter transformation
lp2bs	lowpass to bandstop analog filter transformation
lp2hp	lowpass to highpass analog filter transformation
lp2lp	lowpass to lowpass analog filter transformation
ss2zp	state-space to zero-pole conversion
tf2ss	transfer function to state-space conversion
zp2ss	zero-pole to state-space conversion

Filter Analysis/Implementation	
abs	magnitude
angle	phase angle
fftfilt	overlap-add filter implementation
filter	direct filter implementation
filtfilt	zero-phase verson of **filter**
freqs	Laplace transform frequency response
freqz	z-transform frequency response
grpdelay	group delay
unwrap	unwrap phase

Modeling	
invfreqs	analog filter fit to frequency response
invfreqz	discrete filter fit to frequency response
prony	Prony's discrete filter fit to time response

Spectral Analysis	
cceps	complex cepstrum
detrend	linear trend removal
fft	discrete Fourier transform
fftshift	swap halves of vectors
hilbert	Hilbert transform
ifft	inverse discrete Fourier transform
rceps	real cepstrum and minimum phase reconstruction
specplot	plot output of **spectrum** function
spectrum	Welch method of power spectrum estimation

Correlation/Convolution	
conv	convolution
corrcoef	correlation coefficients
cov	covariance matrix
deconv	deconvolution
xcorr	cross-correlation function
xcov	covariance function

2-D Signal Processing	
conv2	2-D convolution
fft2	2-D discrete Fourier transform
fftshift	swap quadrants of arrays
ifft2	inverse 2-D discrete Fourier transform
xcorr2	2-D cross-correlation

Windows	
bartlett	Bartlett window
blackman	Blackman window
boxcar	rectangular window
chebwin	Chebyshev window
hamming	Hamming window
hanning	Hanning window
kaiser	Kaiser window
triang	triangular window

Decimation/Interpolation	
decimate	lowpass FIR decimation
interp	lowpass interpolation
spline	cubic spline interpolation

Other	
convmtx	convolution matrix
cplxpair	order vector into complex conjugate pairs
dftmtx	discrete Fourier transform matrix
polystab	polynomial stabilization
square	square wave function