Appendix E

MATLAB

E.1 Introduction

MATLAB is a software tool that is useful for solving a wide variety of problems arising in engineering applications. The MATLAB software is a product of a company called The MathWorks. Extensive information on this software (including detailed guides and manuals) is available, free of charge, from the company's web site (`http://www.mathworks.com`). A number of helpful books on MATLAB are also available [1, 2]. In this appendix, a reasonably detailed introduction to MATLAB is also provided.

E.2 Octave

Although MATLAB is very powerful, it is a commercial software product. Therefore, MATLAB is not free. Fortunately, an open-source MATLAB-like software package is available called Octave. Octave is available for download from its official web site `http://www.octave.org`. This software is included in several major Linux distributions (e.g., Fedora Core and Red Hat). As well, Octave is also available for the Cygwin environment under Microsoft Windows. (For more details about Cygwin, see `http://www.cygwin.org`.)

E.3 Invoking MATLAB

On most UNIX systems, the MATLAB software is started by invoking the `matlab` command.

E.3.1 UNIX

The MATLAB software is invoked using a command line of the form:

`matlab` *[options]*

The `matlab` command supports a number of options including the following:

`-help` or `-h`
: Display information on MATLAB options.

`-nodisplay`
: Disable all graphical output. The MATLAB desktop will not be started.

`-nojvm`
: Disable all Java support by not starting the Java virtual machine. In particular, the MATLAB desktop will not be started.

 Copyright © 2013 Michael D. Adams

Table E.1: Keys for Command Line Editing

Key	Action
Up arrow	Recall previous line
Down arrow	Recall next line
Left arrow	Move left one character
Right arrow	Move right one character
Home	Move to beginning of line
End	Move to end of line
Ctrl-C	Cancel current line

`-nodesktop`
Disable the MATLAB desktop. Use the current terminal for commands.

`-display` *displayname*
Specify the X display to use for graphics output.

Like most UNIX programs, MATLAB uses the X Windows System for rendering graphics output. The `DISPLAY` environment variable provides the default display setting for graphics output. If necessary, one can override the default display setting by explicitly specifying the display to use via the `-display` option.

When running MATLAB remotely, it may be necessary to disable the desktop (with the `-nodesktop` option). When the desktop is enabled, MATLAB tends to require an obscene amount of network bandwidth (much more than what a standard DSL link can provide). With the desktop disabled, one should be able to use graphics functionality when running MATLAB remotely without too much trouble with a standard high-speed Internet connection. (Forget about using graphics functionality over a dialup link, however. The X Windows protocol requires far too much bandwidth for that.)

E.3.2 Microsoft Windows

Unfortunately, the author has not used MATLAB under Microsoft Windows. So, he cannot comment on the specifics of running MATLAB under this operating system.

E.4 Command Line Editor

In MATLAB, several keys are quite useful for editing purposes, as listed in Table E.1. For example, the arrow keys can be used to perform editing in the usual way.

E.5 MATLAB Basics

Arguably, one of the most helpful commands in MATLAB is the `help` command. This command can be used to obtain information on many of the operators, functions, and commands available in MATLAB. For example, to find information on the `help` command, one can type:

```
help help
```

In a similar vein, the `doc` command can be used to obtain detailed documentation on many of the functions and commands in MATLAB. For example, to display documentation on the `doc` command, one can type:

```
doc doc
```

Table E.2: Predefined Variables

Variable	Description
`pi`	π
`i`	$\sqrt{-1}$
`j`	$\sqrt{-1}$
`nan`	not-a-number (NaN)
`inf`	infinity
`ans`	last expression evaluated that was not assigned to variable
`date`	date
`clock`	wall clock
`realmin`	smallest usable positive real number
`realmax`	largest usable positive real number
`bitmax`	largest usable positive integer

Table E.3: Operators

Symbol	Description
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	right division
`\`	left division
`^`	exponentiation
`'`	conjugate transpose
`.*`	element-wise multiplication
`./`	element-wise division
`.^`	element-wise exponentiation
`.'`	transpose

E.5.1 Identifiers

Identifiers (i.e., variable/function names) are case sensitive and may consist of uppercase and lowercase letters, underscores, and digits, but the first character cannot be a digit. Although an identifier can be arbitrarily long, only the first n characters are significant, where n depends on the particular version of MATLAB being used. Any characters after the first n are simply ignored. (The `namelengthmax` function can be used to find the precise value of n.) Several variables are automatically predefined by MATLAB as listed in Table E.2. You can assign a new value to a predefined variable, but its original value will be lost.

E.5.2 Basic Functionality

In MATLAB, comments begin with a percent sign character and continue to the end of line. Some of the operators supported by MATLAB are listed in Table E.3. Some math functions provided by MATLAB are listed in Tables E.4, E.5, E.6, E.7, and E.8.

Beware of the `sinc` function in MATLAB. In MATLAB, the `sinc` function is defined as $\operatorname{sinc} x = \frac{sin(\pi x)}{\pi x}$. This is different from the definition of the sinc function employed in this document.

Example E.1. Below are some examples of some very basic calculations done using MATLAB.

```
a = [1 2 3; 4 5 6; 7 8 9] % 3 x 3 array
b = [1 2 3
     4 5 6
```

 Copyright © 2013 Michael D. Adams

Table E.4: Elementary Math Functions

Name	Description
abs	magnitude of complex number
angle	principal argument of complex number
imag	imaginary part of complex number
real	real part of complex number
conj	conjugate of complex number
round	round to nearest integer
fix	round towards zero
floor	round towards $-\infty$
ceil	round towards ∞
sign	signum function
rem	remainder (with same sign as dividend)
mod	remainder (with same sign as divisor)

Table E.5: Other Math-Related Functions

Name	Description
min	minimum value
max	maximum value
mean	mean value
std	standard deviation
median	median value
sum	sum of elements
prod	product of elements
cumsum	cumulative sum of elements
cumprod	cumulative product of elements
polyval	evaluate polynomial
cart2pol	Cartesian-to-polar coordinate conversion
pol2cart	polar-to-Cartesian coordinate conversion

Table E.6: Exponential and Logarithmic Functions

Name	Description
exp	exponential function
log	natural logarithmic function
log10	base-10 logarithmic function
sqrt	square root function

Table E.7: Trigonometric Functions

Name	Description
sin	sine function
cos	cosine function
tan	tangent function
asin	arcsine function
acos	arccosine function
atan	arctangent function
atan2	two-argument form of arctangent function

Copyright © 2013 Michael D. Adams

Table E.8: Radix Conversion Functions

Name	Description
`dec2bin`	convert decimal to binary
`bin2dec`	convert binary to decimal
`dec2hex`	convert decimal to hexadecimal
`hex2dec`	convert hexadecimal to decimal
`dec2base`	convert decimal to arbitrary radix
`base2dec`	convert arbitrary radix to decimal

```
    7 8 9] % 3 x 3 array
a - b
x = [1; 3; -1] % 3-dimensional column vector
y = x .* x + 3 * x + 2
y = a * x

t = 5;
s = t ^ 2 + 3 * t - 7;

z = 3 + 4 * j; % complex number in Cartesian form
z = 20 * exp(j * 10); % complex number in polar form
```

□

The `disp` function prints a single string. For example, the following code fragment prints "Hello, world" (followed by a newline character):

```
disp('Hello, world');
```

The `sprintf` function provides very sophisticated string formatting capabilities, and is often useful in conjunction with the `disp` function. The use of the `sprintf` function is illustrated by the following code fragment:

```
name = 'Jane Doe';
id = '06020997';
mark = 91.345678912;
disp(sprintf('The student %s (ID %s) received a grade of %4.2f%%.', ...
  name, id, mark));
```

The `sprintf` function is very similar in spirit to the function of the same name used in the C programming language.

E.6 Arrays

Frequently, it is necessary to determine the dimensions of an array (i.e., matrix or vector). For this purpose, MATLAB provides two very useful functions as listed in Table E.9. The function `size` can be used to determine the number of rows and/or columns in an array:

- `size(a)` returns a row vector with the number of rows and columns in `a` as elements (in that order);
- `size(a, 1)` returns the number of rows in `a`; and
- `size(a, 2)` returns the number of columns in `a`.

The function `length` is used to find the maximum of the two array dimensions. That is, `length(a)` is equivalent to `max(size(a))`. Usually, the `length` function is used in conjunction with arrays that are known to be row/column vectors.

Example E.2. Suppose that `a = [1 2 3 4; 5 6 7 8]`. Then, `size(a)` returns `[2 4]`, `size(a, 1)` returns `2`, `size(a, 2)` returns `4`, and `length(a)` returns `4`. □

 Copyright © 2013 Michael D. Adams

Table E.9: Array Size Functions

Name	Description
`size`	query array dimensions
`length`	query vector/array dimension

E.6.1 Arrays with Equally-Spaced Elements

Often, it is necessary to specify a vector with equally-spaced elements. As a result, MATLAB provides a compact means for specifying such a vector. In particular, an expression of the following form is employed:

start : *step* : *end*

The above expression is equivalent to a row vector with its first element equal to *start* and each of the subsequent elements increasing in value by *step* until the value would exceed *end*. Note that *step* may be negative.

Example E.3. Here, we give some examples of abbreviated forms of vectors.

Abbreviated Form	Long Form
`1 : 4`	`[1 2 3 4]`
`0 : 0.2 : 1`	`[0 0.2 0.4 0.6 0.8 1]`
`1 : -1 : -2`	`[1 0 -1 -2]`
`0 : 10 : 25`	`[0 10 20]`
`-1.5 : -1 : -4`	`[-1.5 -2.5 -3.5]`

□

E.6.2 Array Subscripting

Suppose that we have an array `a`. We can access elements of the array by specifying the rows and columns in which the elements are contained. In particular, `a(`*rowspec*`,` *colspec*`)` is the array consisting of the elements of `a` that are in the rows specified by *rowspec* and columns specified by *colspec*. Here, *rowspec* is either a vector containing row indices or the special value ":" which means "all rows". Similarly, *colspec* is either a vector containing column indices or the special value ":" which means "all columns". We can also access elements of the array `a` by specifying a 1-dimensional element index, where elements in the array are numbered in column-major order. That is, `a(`*indspec*`)` is the vector of elements of `a` that have the indices specified by *indspec*. Here, *indspec* is either a vector containing element indices or the special value ":" which means "all elements".

Example E.4. Suppose that `a` is a 10×10 matrix and `x` is 10×1 vector. Then, we have the results listed in the table below.

Expression	Meaning
`a(1, :)`	first row of `a`
`a(:, 1)`	first column of `a`
`a(1 : 50)`	first 50 elements of `a` arranged in a row vector
`a(1 : 10)`	first 10 elements of `a` arranged in a row vector (i.e., the first column of `a`)
`a(1 : 2 : 10, :)`	odd-indexed rows of `a`
`a(:, 2 : 2 : 10)`	even-indexed columns of `a`
`a(1 : 5, :)`	rows 1 to 5 of `a`
`a(:, 6 : 10)`	columns 6 to 10 of `a`
`a(1 : 2, 9 : 10)`	submatrix consisting of elements that are in rows 1,2 and also in columns 9,10
`x(1 : 3)`	first three elements of `x` (arranged as a row or column vector to match `x`)

□

Copyright © 2013 Michael D. Adams

Table E.10: Special Matrix/Vector Functions

Name	Description
`eye`	identity matrix
`ones`	matrix of ones
`zeros`	matrix of zeros
`diag`	diagonal matrix
`rand`	random matrix
`linspace`	vector with linearly spaced elements
`logspace`	vector with logarithmically spaced elements

Table E.11: Basic Array Manipulation Functions

Name	Description
`rot90`	rotate array by 90 degrees
`fliplr`	flip array horizontally
`flipud`	flip array vertically
`reshape`	change array dimensions

E.6.3 Other Array Functions

Certain types of matrices tend to be used frequently in code. For this reason, MATLAB provides functions for generating some common forms of matrices. These functions are listed in Table E.10.

MATLAB provides functions for performing some common operations to matrices. These are listed in Table E.11.

E.7 Scripts

Instead of interactively entering MATLAB code for immediate execution, code can be placed in a file and then executed. Normally, MATLAB code is placed in what are called M-files. The term “M file” originates from the fact that these files use a name ending in the suffix “.m”. To create an M-file script, one simply creates a file with a name ending in the suffix “.m”. Then, the code in the M-file can be invoked by using a command with the same name as the M-file but without the “.m” extension. For reasons that will become apparent shortly, the base name of the M-file (i.e., the name without the “.m” extension) must be a valid MATLAB identifier. For example, `2foobar.m` is not a valid M-file name since “2foobar” is not a valid MATLAB identifier. (Recall that MATLAB identifiers cannot start with a digit such as “2”.) Also, in order for MATLAB to find an M-file, the file must be in one of the directories listed in the MATLAB search path. We will explain how to query and change the MATLAB search path later in this section. Before doing so, however, we provide a few examples of M-file scripts below.

Example E.5. Suppose that we have an M-file script called `hello.m` with the following contents:

```
% Print a greeting.
disp('Hello, World.');
```

Then, the code in this file can be executed by simply typing the following in MATLAB:

```
hello
```

That is, we invoke the code in the M-file by using the base name of the file. (It is tacitly assumed that the file `hello.m` has been placed in one of the directories listed in the MATLAB search path.) □

Example E.6. In order to save some typing, we can create a file called `main.m` containing the following:

```
a = [
  0.9501 0.8913 0.8214 0.9218;
  0.2311 0.7621 0.4447 0.7382;
```

 Copyright © 2013 Michael D. Adams

```
    0.6068 0.4565 0.6154 0.1763;
    0.4860 0.0185 0.7919 0.4057;
];
y0 = a * [1 2 3 4].';
y1 = a * [-1 2.5 3 4].';
y3 = a * [41 -22 3 4].';
```

Then, to invoke the code in the above file, we simply type the following in our MATLAB session:

```
main
```

(Again, it is tacitly assumed that the file `main.m` has been placed in one of the directories listed in the MATLAB search path.) □

Generally, one should avoid giving a script file a name that is associated with a previously defined variable or function, as this leads to the potential for naming conflicts. For example, it would be a bad idea to name a script file as `sin.m`, since the `sin` function is already built into MATLAB.

Clearly, MATLAB needs a means to locate M-file scripts, since there are usually many directories in which a user might choose to place a script. For this purpose, the MATLAB search path is used. The MATLAB search path is a list of directories in which MATLAB looks for M-files. In order for the code in an M-file to be successfully located by MATLAB and executed, the M-file must be stored in a directory listed in the MATLAB search path. The MATLAB search path can be queried with the `path` command:

```
path
```

This command will output all of the directories in the MATLAB search path (i.e., all of the directories in which MATLAB will look for M-file scripts).

A new directory can be added to the MATLAB search path with the `addpath` command:

`addpath` *dirname*

This adds the directory *dirname* to the MATLAB search path.

A few other commands are also sometimes useful in the context of M-file scripts. These commands are described below.

The working directory for MATLAB can be changed using the `cd` command:

`cd` *dirname*

The working directory is changed to the directory named *dirname*.

The current working directory can be queried with the `pwd` command:

```
pwd
```

This command will display the current working directory.

E.8 Relational and Logical Operators

The relational and logical operators provided by MATLAB are listed in Tables E.12 and E.13, respectively. Some functions that are also quite useful in relational and logical expressions are listed in Table E.14. As far as boolean expressions are concerned, MATLAB considers any nonzero number to be true and zero to be false.

Example E.7. Suppose that `a = [1 2 3 4 5]` and `b = [5 4 3 2 1]`. Then, we have that

Expression	Value
`a > b`	`[0 0 0 1 1]`
`a == b`	`[0 0 1 0 0]`
`a < b`	`[1 1 0 0 0]`
`a >= 2 & a <= 3`	`[0 1 1 0 0]`
`a < 2 | a > 4`	`[1 0 0 0 1]`
`~a`	`[0 0 0 0 0]`

Copyright © 2013 Michael D. Adams

Table E.12: Relational Operators

Symbol	Description
`<`	less than
`<=`	less than or equal
`>`	greater than
`>=`	greater than or equal
`==`	equal
`~=`	not equal

Table E.13: Logical Operators

Symbol	Description
`&`	and
`|`	or
`~`	not

Table E.14: Relational and Logical Functions

Name	Description
`any`	any element nonzero
`all`	all elements nonzero
`find`	find nonzero elements
`exist`	check if variables exist
`isnan`	detect NaNs
`finite`	detect infinities
`isempty`	detect empty matrices
`isstr`	detect strings
`strcmp`	compare strings

□

Example E.8. The following code fragment illustrates how one might use some of the relational/logical functions such as `all`, `any`, and `find`.

```
a = [1 2 3; 4 5 6; 7 8 9];
if all(a > 0)
    disp('All matrix elements are positive.');
end
if all(a < 0)
    disp('All matrix elements are negative.');
end
if ~any(a == 0)
    disp('All matrix elements are nonzero.');
end
if all(real(a) == a)
    disp('All matrix elements are real.');
end
i = find(a >= 8);
disp('The following matrix elements are greater than or equal to 8:');
disp(a(i));
```

□

 Copyright © 2013 Michael D. Adams

Table E.15: Operator Precedence

Operators	Precedence Level
`()`	highest
`.' ' .^ ^`	
`+ - ~`	
`.* * ./ .\ / \`	
`+ -`	
`:`	
`< <= > >= == ~=`	
`&`	
`|`	lowest

Unary `+` and `-` have higher precedence than the binary forms.
In older versions of MATLAB, `&` and `|` have the same precedence.

E.9 Operator Precedence

The operator precedence is shown in Table E.15.

E.10 Control Flow

In the sections that follow, we introduce the conditional execution and looping constructs available in MATLAB.

E.10.1 If-Elseif-Else

The **`if`**-**`elseif`**-**`else`** construct allows groups of statements to be conditionally executed, and has a number of variants. The simplest variant (i.e., the **`if`** variant) has the form:

`if` *expression*
    statements
`end`

If the expression *expression* is true, then the statements *statements* are executed. The next simplest variant (i.e., the **`if`**-**`else`** variant) has the form:

`if` $expression_1$
    $statements_1$
`else`
    $statements_2$
`end`

If the expression $expression_1$ is true, then the statements $statements_1$ are executed. Otherwise, the statements $statements_2$ are executed. Finally, the most general variant has the form:

`if` $expression_1$
    $statements_1$
`elseif` $expression_2$
    $statements_2$
        ⋮
`elseif` $expression_{n-1}$
    $statements_{n-1}$
`else`
    $statements_n$
`end`

Copyright © 2013 Michael D. Adams

Note that the **elseif** and **else** clauses are optional.

Example E.9. The code fragment below provides a simple example of the use of an **if**-**elseif**-**else** construct. The code tests the sign of the variable x and prints an appropriate message.

```
if x > 0
    disp('x is positive');
elseif x < 0
    disp('x is negative');
else
    disp('x is neither positive nor negative');
end
```

□

E.10.2 Switch

The **switch** construct provides another means to conditionally execute groups of statements. The general form of this construct is as follows:

switch *expression*
case $test\_expression_1$
    $statements_1$
case $test\_expression_2$
    $statements_2$
        ⋮
case $test\_expression_{n-1}$
    $statements_{n-1}$
otherwise
    $statements_n$
end

The switch expression *expression* is compared to each of the test expressions $test\_expression_1$, $test\_expression_2$, ..., $test\_expression_{n-1}$ in order. The first test expression, say $test\_expression_k$, matching the expression *expression* has its corresponding statements $statements_k$ executed. If none of the test expressions match the switch expression and an **otherwise** clause is present, the statements $statements_n$ in this clause are executed. The switch expression must be either a scalar or string.

Example E.10. The code fragment below examines the real variable n and prints some information concerning its value.

```
n = 5;
switch mod(n, 2)
case 0
    disp('number is even integer');
case 1
    disp('number is odd integer');
case {0.5, 1.5}
    disp('number is half integer');
otherwise
    disp('number is not an integer');
end
```

□

Example E.11. The code fragment below converts a mass specified in a variety of units to kilograms.

 Copyright © 2013 Michael D. Adams

```
x = 100; % input mass
units = 'lb'; % units for input mass
switch units
case {'g'}
    y = 0.001 * x;
case {'kg'}
    y = x;
case {'lb'}
    y = 0.4536 * x;
otherwise
    error('unknown units');
end
disp(sprintf('%f %s converts to %f kg', x, units, y));
```

□

E.10.3 For

The **for** construct allows a group of statements to be repeated a fixed number of times. This construct has the general form:

```
for variable = array
    statements
end
```

The statements *statements* are executed once for each value in the array *array*, where the variable *variable* is set to the corresponding array value each time.

Example E.12 (Degree to radian conversion)**.** The following code fragment outputs a table for converting angles from units of degrees to radians.

```
disp('Degrees    Radians');
for theta_degrees = -5 : 0.5 : 5
    theta_radians = theta_degrees * pi / 180;
    disp(sprintf('%7.1f    %7.4f', theta_degrees, theta_radians));
end
```

□

Example E.13. The code fragment below applies a linear transformation (represented by the matrix `a`) to each column of the matrix `[0 2 4 6; 1 3 5 7]`.

```
a = [1 0; 1 -1];
for v = [0 2 4 6; 1 3 5 7]
    disp(a * v);
end
```

□

E.10.4 While

The **while** construct allows a group of statements to be executed an indefinite number of times. This construct has the form:

```
while expression
    statements
end
```

The statements *statements* are executed repeatedly as long as the condition *expression* is true.

Example E.14. In the code fragment below, we compute the smallest machine-representable positive real number that, when added to one, is still greater than one. (Due to finite-precision effects, there is a lower bound on this quantity.) This quantity is essentially the built-in `eps` variable in MATLAB.

```
epsilon = 1;
while (1 + epsilon / 2) > 1
    epsilon = epsilon / 2;
end
disp(epsilon);
```

□

E.10.5 Break and Continue

Sometimes, it may necessary to prematurely break out of a loop or prematurely continue with its next iteration. This is accomplished via **`break`** and **`continue`**.

Example E.15. The two code fragments given below are equivalent. The first one employs a **`break`** statement, while the other does not.

```
% Code fragment 1
done = 0;
while 1
    if done
        break % Terminate (i.e., break out of) loop.
    end
    % Do something here.
    % If we are finished, set done to one.
end

% Code fragment 2
done = 0;
while ~done
    % Do something here.
    % If we are finished, set done to one.
end
```

□

Example E.16. The code fragment below gives an example of the use of the **`continue`** statement.

```
a = [1 0 3 2 0];
for i = a
    if i == 0
        % Skip over the processing of a zero element in the array.
        continue
    end
    % Process the nonzero array element.
    disp(i);
end
```

The above code will print only the nonzero elements of the array `a`. □

 Copyright © 2013 Michael D. Adams

E.11 Functions

MATLAB supports user-defined functions. To create a user-defined function, the code for the function is placed in an M-file. In this sense, user-defined functions are very similar to script files. For this reason, most of the material on script files in Section E.7 is also applicable here. There is, however, one key difference between a script and function file. A function file must include a **function** directive (whereas a script file must not). This directive is primarily used to indicate the number of input and output arguments for a function.

The first (non-comment) line in function file must contain the **function** directive. This directive has the form:

function [$argout_1, argout_2, \ldots, argout_n$] = $funcname$ ($argin_1, argin_2, \ldots, argin_m$)

This indicates that the function has the name *funcname*, the m input arguments $argin_1$, $argin_2$, $\ldots$, $argin_m$, and n output arguments $argout_1$, $argout_2$, $\ldots$, $argout_n$. The function name *funcname* should always be the same as the base name of the file in which the function is stored (i.e., the file name without the ".m" suffix). Immediately following the line containing the **function** directive, one should provide comments to be printed in response to a `help` inquiry for the function. The code in a function file executes until either the end of file is reached or a **return** statement is encountered.

In MATLAB all input arguments to a function are passed by value. For this reason, changes to the input arguments made inside of a function will not propagate to the caller. Also, any variables accessed/manipulated inside of a function are local in scope to that function.

Example E.17 (Sinc function)**.** Unfortunately, the `sinc` function provided by MATLAB is not quite the same as the sinc function as defined in these notes. In Listing E.1, we define a function `mysinc` that is compatible with the definition herein. The command `help mysinc` will result in MATLAB printing the first block of comments from the above function file.

Listing E.1: `mysinc.m`

```
function y = mysinc(x)
% mysinc - Compute the sinc function (as defined in these notes)
% mysinc(x) returns a matrix whose elements are the sinc of the
% elements of x

% Initialize the output array to all ones.
y = ones(size(x));
% Determine the indices of all nonzero elements in the input array.
i = find(x);
% Compute the sinc function for all nonzero elements.
% The zero elements are already covered, since the output
% array was initialized to all ones above.
y(i) = sin(x(i)) ./ (x(i));
return
```

□

Example E.18 (Factorial function)**.** The contents of the file named `myfactorial.m` is as given in Listing E.2. The code is invoked by calling the `myfactorial` function. For example, `myfactorial(4)` returns the value 24.

Listing E.2: `myfactorial.m`

```
function y = myfactorial(x)
% myfactorial - compute factorial

y = 1;
for n = 2 : x
        y = y * n;
end
return
```

Table E.16: Special Predefined Function Variables

Name	Description
`nargin`	number of input arguments
`nargout`	number of output arguments
`varargin`	variable-length input argument
`varargout`	variable-length output argument

□

In MATLAB, functions may take a variable number of input arguments and may return a variable number of output arguments. In order for a function to determine the number of input and output arguments and access these arguments, several variables are automatically defined upon entry to a function. These variables are listed in Table E.16. In what follows, we give some examples of functions that take a variable number of input arguments.

Example E.19 (Function with variable number of input arguments)**.** The contents of the file named `mysum.m` is as given in Listing E.3. The code is invoked by calling the `mysum` function. For example, `mysum(1)` returns the value 1, `mysum(1, 2)` returns the value 3, and `mysum(1, 2, 3)` returns the value 6.

Listing E.3: `mysum.m`

```
function y = mysum(a, b, c)
% mysum - calculate the sum (of one to three quantities)

if nargin == 1
    % function called with one argument
    y = a;
elseif nargin == 2
    % function called with two arguments
    y = a + b;
elseif nargin == 3
    % function called with three arguments
    y = a + b + c;
else
    error('invalid number of arguments');
end
return
```

□

Perhaps, we would like to write a function similar to the one in the previous example, except we would like to be able to handle an arbitrary number of input arguments (possibly many more than three). This can be accomplished by using the special predefined `varargin` variable.

Example E.20 (Variable number of input arguments)**.** Suppose that we would like to write a function that returns the sum of its input arguments, but allows the number of input arguments to be arbitrary. We can achieve this functionality with the code in Listing E.4. The code is invoked by calling the `mysum2` function. For example, `mysum2(1)` returns the value 1, `mysum2(1, 2, 3)` returns the value 6, and `mysum2(1, 1, 1, 1, 1, 1, 1, 1)` returns the value 8.

Listing E.4: `mysum2.m`

```
function y = mysum2(varargin)
% mysum2 - Compute the sum of the input arguments

if nargin == 0
    y = [];
```

 Copyright © 2013 Michael D. Adams

Table E.17: Basic Plotting Functions

Name	Description
`plot`	linear x-y plot
`loglog`	log log x-y plot
`semilogx`	semi-log x-y plot (x-axis logarithmic)
`semilogy`	semi-log x-y plot (y-axis logarithmic)
`polar`	polar plot
`bar`	bar chart
`stem`	stem plot
`pcolor`	pseudocolor (checkerboard) plot

Table E.18: Other Graphing Functions/Commands

Name	Description
`axis`	control axis scaling and appearance
`hold`	hold current plot
`subplot`	multiple axes in single figure
`figure`	create figure

```
    return
end
y = varargin{1};
for i = 2 : nargin
    y = y + varargin{i};
end
return
```

□

E.12 Graphing

MATLAB has a very rich set of graphing capabilities. Herein, we will try to illustrate some of these capabilities by way of examples.

Some of the basic plotting functions are listed in Table E.17 and several other graphing-related functions/commands are given in Table E.18.

When generating plots, it is sometimes desirable to be able to specify line styles, line colors, and marker styles. The supported line styles, line colors, and marker styles are listed in Tables E.19, E.20, and E.21, respectively.

Example E.21 (Simple plot)**.** Suppose that we want to plot the function $\sin t$ for t in the interval $[-4\pi, 4\pi]$. This can be accomplished with the following code:

```
t = linspace(-4 * pi, 4 * pi, 500);
y = sin(t);
plot(t, y);
```

The resulting plot is shown in Figure E.1. □

Often, we need to add annotations to plots (e.g., a title, axis labels, etc.). This is accomplished using the functions listed in Table E.22. Sometimes, we wish to use special symbols (such as Greek letters) in annotations. Fortunately, numerous symbols are available in MATLAB as listed in Table E.23.

Example E.22 (Annotated plot)**.** Here, we plot the function $\alpha(\omega) = |\omega|^2 \sin \omega$. Special symbols are used in the axis labels (in order to display Greek letters).

Copyright © 2013 Michael D. Adams

Table E.19: Line Styles

Symbol	Line Style
`-`	solid
`:`	dotted
`-.`	dash dot
`--`	dashed

Table E.20: Line Colors

Symbol	Line Color
`b`	blue
`g`	green
`r`	red
`c`	cyan
`m`	magenta
`y`	yellow
`k`	black
`w`	white

Table E.21: Marker Styles

Symbol	Marker Style
`.`	point
`o`	circle
`x`	cross
`+`	plus sign
`*`	asterisk
`s`	square
`d`	diamond
`v`	triangle (down)
`^`	triangle (up)
`<`	triangle (left)
`>`	triangle (right)
`p`	pentagram
`h`	hexagram

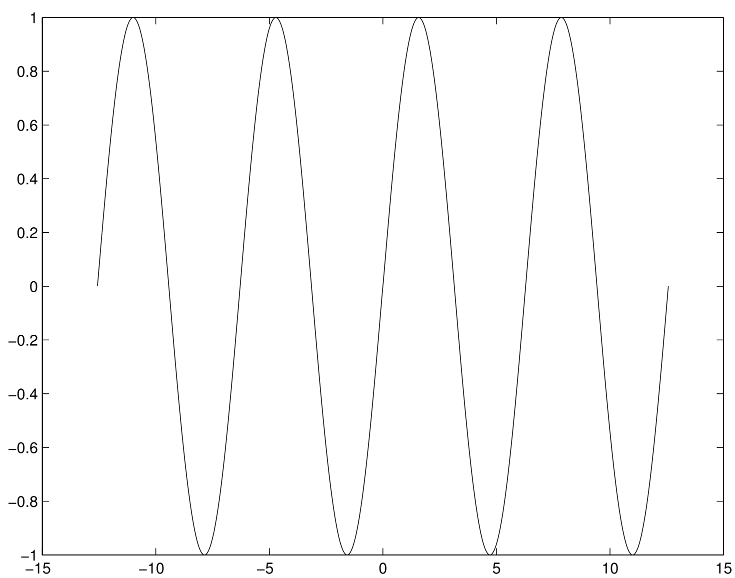

Figure E.1: Plot from example.

Table E.22: Graph Annotation Functions

Name	Description
`title`	graph title
`xlabel`	x-axis label
`ylabel`	y-axis label
`grid`	grid lines
`text`	arbitrarily-positioned text
`gtext`	mouse-positioned text

Copyright © 2013 Michael D. Adams

Table E.23: Special Symbols for Annotation Text

String	Symbol
`\alpha`	α
`\beta`	β
`\delta`	δ
`\gamma`	γ
`\omega`	ω
`\theta`	θ
`\Delta`	Δ
`\Omega`	Ω

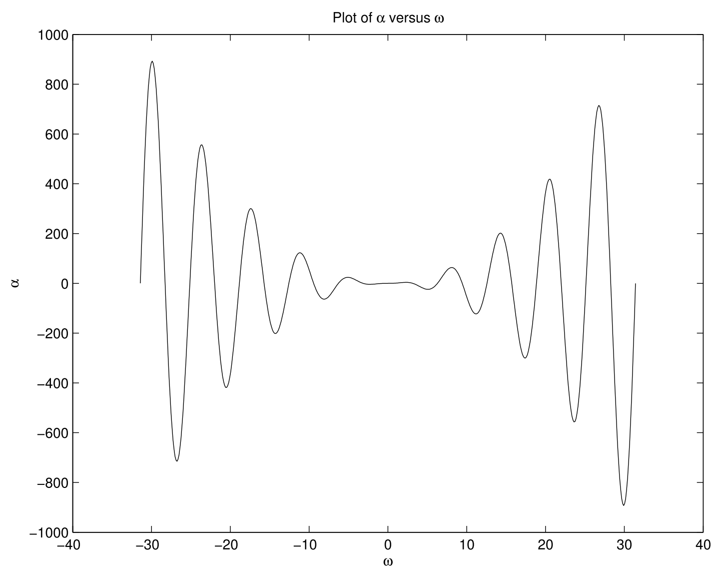

Figure E.2: Plot from example.

```
w = linspace(-10 * pi, 10 * pi, 500);
a = abs(w) .^ 2 .* sin(w);
plot(w, a);
xlabel('\omega');
ylabel('\alpha');
title('Plot of \alpha versus \omega');
```

The resulting plot is shown in Figure E.2. □

Example E.23 (Multiple plots on single axes)**.** Suppose that we want to plot both $\sin t$ and $\cos t$ on the same axes for t in the interval $[-2\pi, 2\pi]$. This can be accomplished with the following code:

```
t = linspace(-2 * pi, 2 * pi, 500);
plot(t, sin(t), ':'); % plot using dotted line
hold on % cause the second plot not to erase the first
plot(t, cos(t), '-'); % plot using solid line
hold off % allow the next plot to erase previous plots
```

The resulting plot is shown in Figure E.3. □

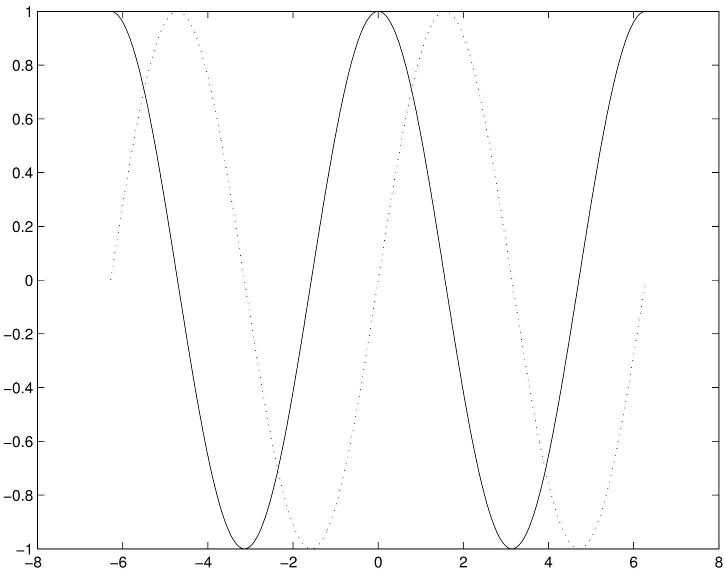

Figure E.3: Plot from example.

Example E.24 (Multiple axes on same figure)**.** We would like to plot the functions $\cos t$, $\sin t$, $\arccos t$, and $\arcsin t$ using four separate plots on the same figure. This can be done as follows:

```
t = linspace(-pi, pi, 500);
subplot(2, 2, 1); % select first subplot in 2-by-2 layout
plot(t, cos(t));
title('cos(t)')
subplot(2, 2, 2); % select second subplot in 2-by-2 layout
plot(t, sin(t));
title('sin(t)');
t = linspace(-1, 1, 500);
subplot(2, 2, 3); % select third subplot in 2-by-2 layout
plot(t, acos(t));
title('acos(t)')
subplot(2, 2, 4); % select fourth subplot in 2-by-2 layout
plot(t, asin(t));
title('asin(t)')
```

The resulting graphs are shown in Figure E.4. □

E.13 Printing

To print copies of figures, the `print` command is employed. This command has various forms, but one of the simplest is as follows:

`print -d`*device filename*

Here, *device* specifies the output format and *filename* is the name of the file in which to save the output. For example, for a PostScript printer, the `ps` format should be used. For more details on additional options, type `help print`.

 Copyright © 2013 Michael D. Adams

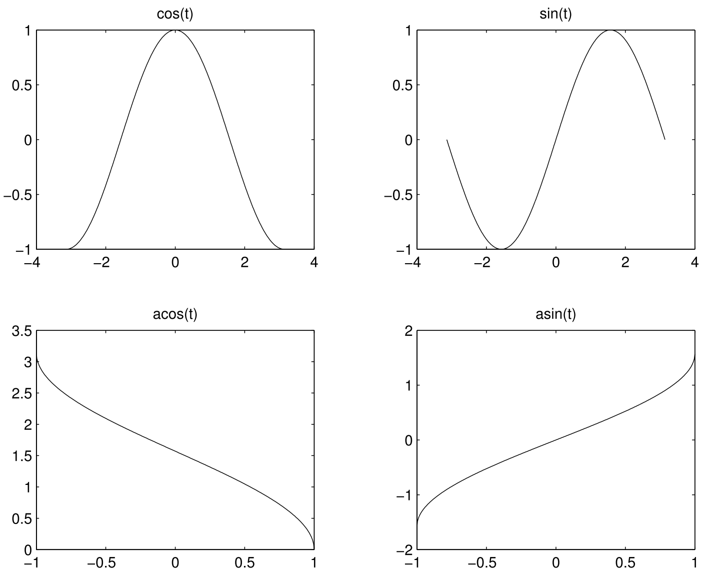

Figure E.4: Plot from example.

E.14 Symbolic Math Toolbox

Symbolic computation is sometimes quite helpful in solving engineering problems. For example, a very complicated formula or equation involving several variables might need to be simplified without assuming specific values for the variables in the formula/equation. The Symbolic Math Toolbox provides MATLAB with such symbolic computation capabilities.

E.14.1 Symbolic Objects

The Symbolic Math Toolbox defines a new data type called a **symbolic object**. The toolbox uses symbolic objects to represent symbolic variables and expressions. In essence, a symbolic object can have as its value any valid mathematical expression.

Symbolic objects can be used in many of the same ways as non-symbolic objects. One must, however, keep in mind that performing a computation symbolically is quite different than performing it non-symbolically. Generally speaking, symbolic computation is much slower than non-symbolic computation.

E.14.2 Creating Symbolic Objects

A symbolic object is created using either the `sym` function or `syms` directive. For example, a symbolic object `f` with the value ax^2+bx+c is created as follows:

```
f = sym('a * x ^ 2 + b * x + c');
```

Sometimes, it is necessary to create one or more symbolic objects, each with its value equal to its name. Obviously, this can be accomplished with the `sym` function. For example, we can write:

```
x = sym('x');
y = sym('y');
z = sym('z');
```

This common construct, however, has an abbreviated form. Instead of writing the above lines of code, we can simply say:

Copyright © 2013 Michael D. Adams

```
syms x y z;
```

E.14.3 Manipulating Symbolic Objects

Symbolic objects can often be used in the same way as non-symbolic objects. For example, we can do things like:

```
f = sym('t + 1');
g0 = f ^ 3 - 2 * f - 21;
g1 = cos(f) * sin(f / 3);
```

Symbolic objects are quite distinctive from other types of objects in MATLAB. For example, the following two lines of code have very different effects:

```
x = pi;
x = sym('pi');
```

It is important to understand the difference in what these two lines of code do.

To substitute some expression/variable for another variable, use the `subs` function. For example, to substitute $t+1$ for t in the expression t^2, we can use the following code:

```
f = sym('t ^ 2');
g = subs(f, 't', 't + 1')
```

To factor a symbolic expression, use the `factor` function. For example, suppose that we want to factor the polynomial t^2+3t+2. This could be accomplished with the following code:

```
f = sym('t ^ 2 + 3 * t + 2');
g = factor(f)
```

After executing the preceding code, `g` is associated with the expression $(t+2)(t+1)$. Note that the `factor` function will only produce factors with real roots.

To simplify a symbolic expression, use the `simplify` function. For example, suppose that we want to substitute $2t+1$ for t in the expression t^2-1 and simplify the result. This can be accomplished with the following code:

```
f = sym('t ^ 2 - 1');
g = simplify(subs(f, 't', '2 * t + 1'))
```

To expand an expression, use the `expand` function. For example, to compute $(t+1)^{10}$, we can use the following code:

```
f = sym('(t+1)^10');
g = expand(f)
```

To display an expression in a human-friendly format, use the `pretty` function. For example, to compute $\frac{1}{2}t^2+\frac{1}{3}(t+1)^{-4}$ in an expanded and beautified format, type:

```
f = sym('((1/2) * t^2 + (1/3) * (t+1))^(-4)');
pretty(expand(f))
```

Example E.25 (Sum of arithmetic series)**.** Consider a sum of the form

$$S(a,d,n) \triangleq \sum_{k=0}^{n-1}(a+kd)$$

(i.e., the sum of an arithmetic series). Suppose that we did not happen to know that this sum can be calculated as

$$S(a,d,n) = \tfrac{1}{2}n\left(2a+d(n-1)\right).$$

We could determine a general formula for the sum using the following code:

 Copyright © 2013 Michael D. Adams

Table E.24: Functions related to signal processing

Name	Description
besself	Bessel analog filter design
bode	bode plot
butter	Butterworth analog/digital filter design
freqs	frequency response of analog filter
impulse	compute impulse response
lsim	simulate system
step	compute step response
tf	generate system model from transfer function coefficients

```
f = sym('a + k * d');
simple(symsum(f, 'k', '0', 'n - 1'))
```

The code yields the result `1/2*n*(2*a+d*n-d)`. Clearly, this result is equivalent to the known expression for $S(a,d,n)$ given above. □

E.14.4 Plotting Symbolic Expressions

To plot a symbolic expression in one variable, use the `ezplot` function. For example, to plot the function $f(t) = 3t^2 - 4t + 2$, we can use the following code:

```
f = sym('3 * t ^ 2 - 4 * t + 2');
ezplot(f);
```

The range of the independent variable may optionally be specified. For example, to plot the expression with t in the interval $[-1,1]$, type:

```
ezplot(f, [-1 1]);
```

E.15 Signal Processing

MATLAB provides many functions that are helpful in the analysis of signals and systems. Some of the more useful functions are listed in Table E.24. In what follows, we provide several examples of how some of these functions can be used.

E.15.1 Frequency Responses

In signal processing, we often need to evaluate the frequency responses of LTI systems. Usually such systems have a frequency response that is rational. That is, the frequency response $H_F(\omega)$ is of the form

$$H_F(\omega) = \frac{\sum_{k=0}^{M-1} \alpha_k \omega^k}{\sum_{k=0}^{N-1} \beta_k \omega^k},$$

where α_k and β_k are complex constants. Fortunately, a function of this form can be evaluated quite easily in MATLAB using the `polyval` function.

Often, it is more convenient to characterize a LTI system by its transfer function rather than its frequency response. Most systems of interest have a transfer function that is rational. That is, the transfer function $H_{TF}(s)$ is of the form

$$H_{TF}(s) = \frac{\sum_{k=0}^{M-1} a_k s^k}{\sum_{k=0}^{N-1} b_k s^k},$$

Copyright © 2013 Michael D. Adams

where a_k and b_k are complex constants. Although we could compute the frequency response by using `polyval` to evaluate the transfer function at points on the imaginary axis, there is an easier way to accomplish this task. In particular, we can simply use the `freqs` function provided by MATLAB.

Example E.26 (Computing and plotting frequency responses with freqs)**.** Consider the LTI system with transfer function $H(s)$, where

$$H(s) = \frac{1}{s^2 + \sqrt{2}s + 1}.$$

(This system is a second-order Butterworth lowpass filter with a cutoff frequency of 1 rad/s.) Suppose that we would like to evaluate the frequency response at various values of ω. This is equivalent to evaluating the transfer function $H(s)$ at points on the imaginary axis. To this end, we can employ the `freqs` function in MATLAB. More specifically, we can calculate and plot the magnitude and phase responses of the above system with the code given below.

```
% Initialize the numerator and denominator coefficients of the transfer
% function.
tfnum = [0 0 1];
tfdenom = [1 sqrt(2) 1];

% Plot the magnitude and phase responses.
freqs(tfnum, tfdenom, linspace(0, 2, 500));
```

□

Example E.27 (Plotting frequency responses)**.** Suppose that we would like to have a function that behaves in a similar way to the MATLAB `freqs` function, but with a few differences in how plotting is performed. In particular, we would like the plots generated with the magnitude response as a unitless quantity and the phase response in unwrapped form. This can be accomplished with the code given in Listing E.5.

Listing E.5: `myfreqs.m`

```
function [freqresp, omega] = myfreqs(tfnum, tfdenom, omega)
% The myfreqs function has essentially the same interface as the MATLAB freqs
% function, but performs plotting slightly differently.
% The magnitude response is plotted as a unitless quantity (not in decibels).
% The phase response is plotted with the phase unwrapped.

% If the frequencies have been specified as an input argument, then simply
% pass them through to the real freqs function.
if nargin >= 3
        [freqresp, omega] = freqs(tfnum, tfdenom, omega);
else
        [freqresp, omega] = freqs(tfnum, tfdenom);
end

% If no output arguments were specified, plot the frequency response.
if nargout == 0

    % Compute the magnitude response as a unitless quantity.
    magresp = abs(freqresp);

    % Compute the phase response with the phase unwrapped.
    phaseresp = unwrap(angle(freqresp));

    % On the first of two graphs, plot the magnitude response.
```


Copyright © 2013 Michael D. Adams

```
    subplot(2, 1, 1);
    plot(omega, magresp);
    title('Magnitude Response');
    xlabel('Frequency (rad/s)');
    ylabel('Magnitude (unitless)');

    % On the second of two graphs, plot the phase response.
    subplot(2, 1, 2);
    plot(omega, phaseresp);
    title('Phase Response');
    xlabel('Frequency (rad/s)');
    ylabel('Angle (rad)');

end
```

□

E.15.2 Impulse and Step Responses

Sometimes, we need to determine the response of a LTI system to a specific input. Two inputs of particular interest are the Dirac delta function $\delta(t)$ and unit-step function $u(t)$. Fortunately, it is quite easy to compute impulse and step responses, as illustrated by the example below.

Example E.28 (Computing impulse and step responses)**.** Consider the LTI system with the transfer function

$$H(s) = \frac{1}{s^2+\sqrt{2}s+1}.$$

Suppose that we wish to calculate and plot the impulse and step responses of this system. This can be accomplished with the code given below.

```
% Initialize the numerator and denominator coefficients of the transfer
% function.
tfnum = [0 0 1];
tfdenom = [1 sqrt(2) 1];

% Determine the system model associated with the given transfer function.
sys = tf(tfnum, tfdenom);

% Plot the impulse response.
subplot(2, 1, 1);
impulse(sys);

% Plot the step response.
subplot(2, 1, 2);
step(sys);
```

Executing the above code produces the plots shown in Figure E.5. □

E.15.3 Filter Design

Example E.29 (Butterworth filter design)**.** Suppose that we want to design a tenth-order Butterworth lowpass filter with a cutoff frequency of 100 rad/s. We can accomplish this using the code below.

Copyright © 2013 Michael D. Adams

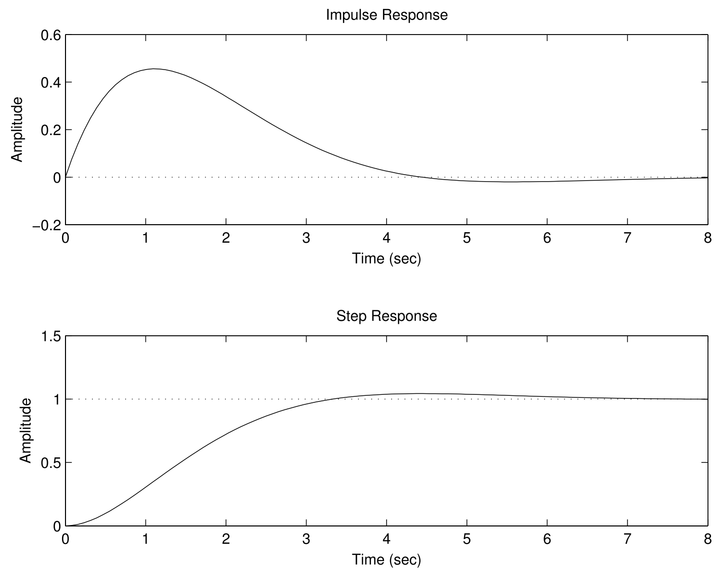

Figure E.5: Plot from example.

```
% Calculate the transfer function coefficients for a tenth-order Butterworth
% lowpass filter with a cutoff frequency of 100 rad/s.
[tfnum, tfdenom] = butter(10, 100, 's');

% Plot the frequency response of the filter.
freqs(tfnum, tfdenom);
```

This generates a filter with the frequency response shown in Figure E.6. □

Example E.30 (Bessel filter design)**.** Suppose that we want to design a tenth-order Bessel lowpass filter with a cutoff frequency of 100 rad/s. We can achieve this with the code given below.

```
% Calculate the transfer function coefficients for a tenth-order Bessel
% lowpass filter with a cutoff frequency of 100 rad/s.
[tfnum, tfdenom] = besself(10, 100);

% Plot the frequency response of the filter.
freqs(tfnum, tfdenom);
```

This generates a filter with the frequency response shown in Figure E.7. □

E.16 Miscellany

Some other functions that might be useful are listed in Table E.25.

 Copyright © 2013 Michael D. Adams

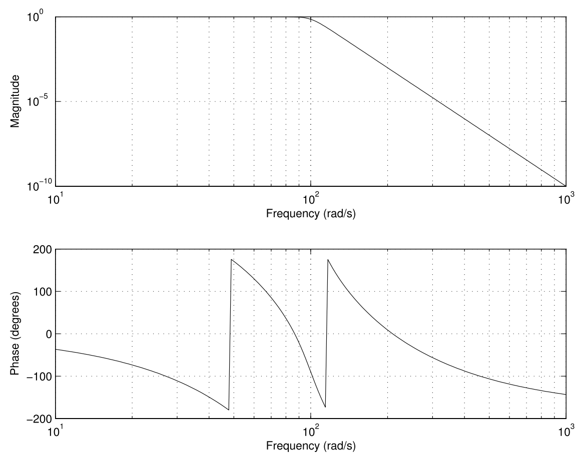

Figure E.6: Frequency response of the Butterworth lowpass filter.

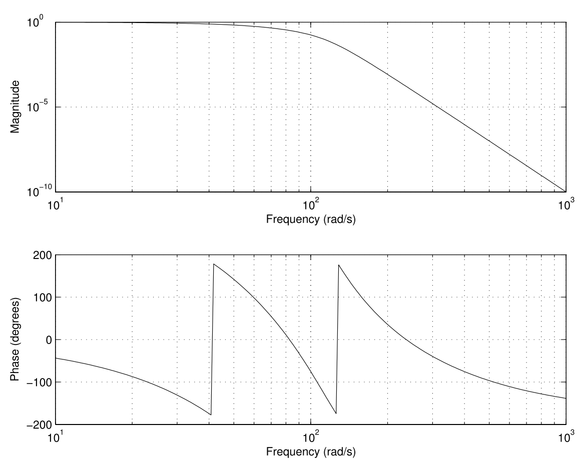

Figure E.7: Frequency response of the Bessel lowpass filter.

Table E.25: Miscellaneous Functions/Commands

Name	Description
`roots`	find roots of polynomial
`clear`	clear a variable
`diary`	log MATLAB session
`echo`	echo commands on execution (for debugging)
`quit`	quit MATLAB
`format`	output format for numbers

Copyright © 2013 Michael D. Adams