Table D.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). This is due to the fact that, when the desktop is enabled, MATLAB tends to require a relatively large amount of network bandwidth, which can be problematic over lower-speed network connections.

D.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.

D.4 Command Line Editor

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

D.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
```

D.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 or an underscore. 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 query the precise

Table D.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

Table D.3: Operators

Symbol	Description
`+`	unary plus (i.e., identity) and binary plus (i.e., addition)
`-`	unary minus (i.e., negation) and binary minus (i.e., subtraction)
`*`	multiplication
`/`	right division
`\`	left division
`^`	exponentiation
`'`	conjugate transpose
`.*`	element-wise multiplication
`./`	element-wise division
`.^`	element-wise exponentiation
`.'`	transpose

value of n.) Several variables are automatically predefined by MATLAB as listed in Table D.2. A new value can be assigned to a predefined variable, causing its original value to be lost.

D.5.2 Basic Functionality

Generally, it is desirable to include comments in code to explain how the code works. 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 D.3. As a matter of terminology, an operator that takes one operand is said to be **unary**, while an operator that takes two operands is said to be **binary**. Note that the plus (`+`) and minus (`-`) operators have both unary and binary forms. For example, the expression "`-x`" employs the unary minus (i.e., negation) operator (where the minus operator has the single operand `x`), while the expression "`x - y`" employs the binary minus (i.e., subtraction) operator (where the operator has two operands `x` and `y`).

Some math functions provided by MATLAB are listed in Tables D.4, D.5, D.6, D.7, D.8, and D.9. Note that the `sinc` function in MATLAB does not compute the sinc function as defined (by (3.20)) herein. Instead, this MATLAB function computes the normalized sinc function (as defined by (3.21)).

Example D.1. Some examples of very basic calculations performed using MATLAB are as follows:

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


Table D.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 D.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 D.6: Exponential and logarithmic functions

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

Table D.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

Table D.8: Other math functions

Name	Description
`sinc`	normalized sinc function (as defined in (3.21))

Table D.9: 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

```
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.

D.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 D.10. 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 D.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`. ■

Table D.10: Array size functions

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

Table D.11: 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]`

D.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 D.3. In Table D.11, some examples of abbreviated forms of vectors are given. ■

D.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 D.4. Suppose that `a` is a 10×10 matrix and `x` is 10×1 vector. Some examples of array subscripting involving `a` and `x` are given in Table D.12. ■

D.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 D.13.

MATLAB provides functions for performing some common operations to matrices. These are listed in Table D.14.

D.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

Table D.12: Array subscripting examples

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`)

Table D.13: 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 D.14: Basic array manipulation functions

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

(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 D.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 D.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;
  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*

Table D.15: Examples of expressions involving relational operators

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]`

Table D.16: Relational operators

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

Table D.17: Logical operators

Symbol	Description
&	element-wise and
|	element-wise or
~	not
&&	short-circuit and
||	short-circuit or

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.

D.8 Relational and Logical Operators

The relational and logical operators provided by MATLAB are listed in Tables D.16 and D.17, respectively. Some functions that are also quite useful in relational and logical expressions are listed in Table D.18. As far as boolean expressions are concerned, MATLAB considers any nonzero number to be true and zero to be false.

Example D.7. Suppose that `a = [1 2 3 4 5]` and `b = [5 4 3 2 1]`. Some examples of expressions using `a` and `b` in conjunction with relational operators are given in Table D.15. ■

Example D.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
```


Table D.18: Relational and logical functions

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

Table D.19: Operator precedence

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

In older versions of MATLAB, `&` and `|` have the same precedence.

```
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));
```

■

D.9 Operator Precedence

When an expression involves more than one operator (such as "`-x ^ 2 + 2 * x - 1`"), the order in which operators are applied becomes important. This ordering is more formally referred to as operator precedence. The precedence of operators in MATLAB is shown in Table D.19. Note that the unary and binary minus operators have different precedence, and the unary and binary plus operators have different precedence. As one would probably expect, the multiplication operator (`*`) has higher precedence than the addition operator (binary `+`), and the exponentiation operator (`^`) has higher precedence than both the addition and multiplication operators.

D.10 Control Flow

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

D.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

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

Example D.9. The following code fragment 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
```

■

D.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 D.10. The following code fragment 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 D.11. The following code fragment converts a mass specified in a variety of units to kilograms:

```
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));
```

■

D.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 D.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 D.13. The following code fragment 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
```
■

D.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 D.14. Suppose that we would like to compute the smallest machine-representable positive real number that, when added to one, is still greater than one. This quantity is sometimes referred to as **machine epsilon**. Due to finite-precision effects, there is a lower bound on this quantity. Although machine epsilon is available via the built-in `eps` variable in MATLAB, we can compute its value explicitly using the following code:

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

D.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 D.15. The following two code fragments are equivalent, where the first employs a **`break`** statement and the second 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 D.16. The following code fragment 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. ■

D.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 D.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 the 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 body of a function extends from the **function** directive to a corresponding **end** directive or, if no such **end** directive is present, the end of the file. The code in a function executes until either the end of the function 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 D.17 (Sinc function)**.** As noted earlier, the sinc function provided by MATLAB computes the normalized sinc function (as defined by (3.21)), not the sinc function (as defined by (3.20)). In Listing D.1, we define a function mysinc that computes the sinc function. The command help mysinc will result in MATLAB printing the first block of comments from the above function file.

Listing D.1: mysinc.m

```
1 function y = mysinc(x)
2     % mysinc - Compute the sinc function.
3     % mysinc(x) returns a matrix whose elements are the sinc of the
4     % elements of x
5
6     % Initialize the output array to all ones.
```


Table D.20: 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

```
    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));
end
```

■

Example D.18 (Factorial function)**.** Suppose that we would like to write a function called `myfactorial` that takes a single integer argument n and returns $n!$. We can achieve this functionality with the code in Listing D.2, which is placed in a file named `myfactorial.m`. The code is invoked by calling the `myfactorial` function. For example, `myfactorial(4)` returns the value 24.

Listing D.2: `myfactorial.m`

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

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

■

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 D.20. In what follows, we give some examples of functions that take a variable number of input arguments.

Example D.19 (Function with variable number of input arguments)**.** Suppose that we would like to write a function called `mysum` that takes one, two, or three arguments and returns their sum. We can achieve this functionality with the code in Listing D.3, which is placed in a file named `mysum.m`. 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 D.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
```


Table D.21: 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

```
        y = a + b + c;
    else
        error('invalid number of arguments');
    end
end
```

■

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 D.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 D.4, which is placed in a file named `mysum2.m`. 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 D.4: `mysum2.m`

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

    if nargin == 0
        y = [];
        return
    end
    y = varargin{1};
    for i = 2 : nargin
        y = y + varargin{i};
    end
end
```

■

D.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 D.21 and several other graphing-related functions/commands are given in Table D.22.

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 D.23, D.24, and D.25, respectively.

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

Table D.22: 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

Table D.23: Line styles

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

Table D.24: Line colors

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

Table D.25: 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

Table D.26: 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

```
% Select the sample points for the function to be plotted.
t = linspace(-4 * pi, 4 * pi, 500);

% Sample the function to be plotted at the sample points.
y = sin(t);

% Clear the current figure.
clf

% Plot the data.
plot(t, y);
```

The resulting plot is shown in Figure D.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 D.26. Sometimes, we wish to use special symbols (such as Greek letters) in annotations. Fortunately, numerous symbols are available in MATLAB as listed in Table D.27.

Example D.22 (Annotated plot)**.** Suppose that we would like to plot the function $\alpha(\omega) = |\omega|^2 \sin \omega$, using special symbols in the axis labels to display Greek letters. This can be accomplished with the following code:

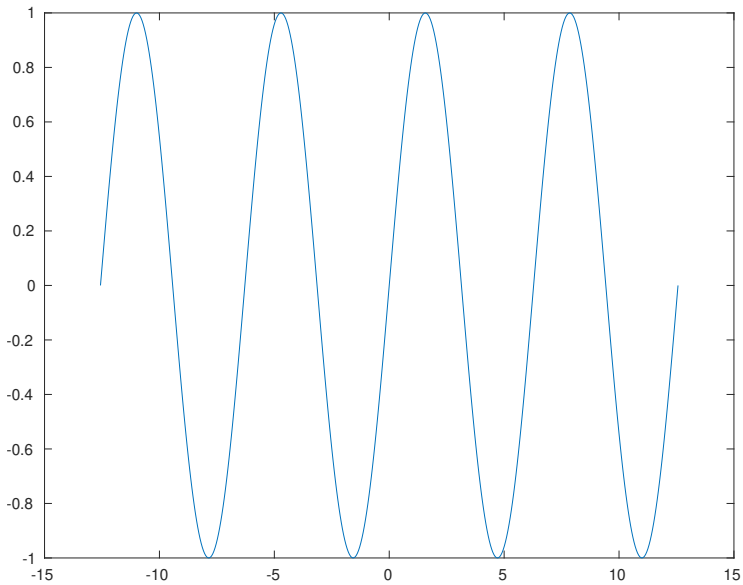

Figure D.1: Plot from example.

Table D.27: Special symbols for annotation text

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

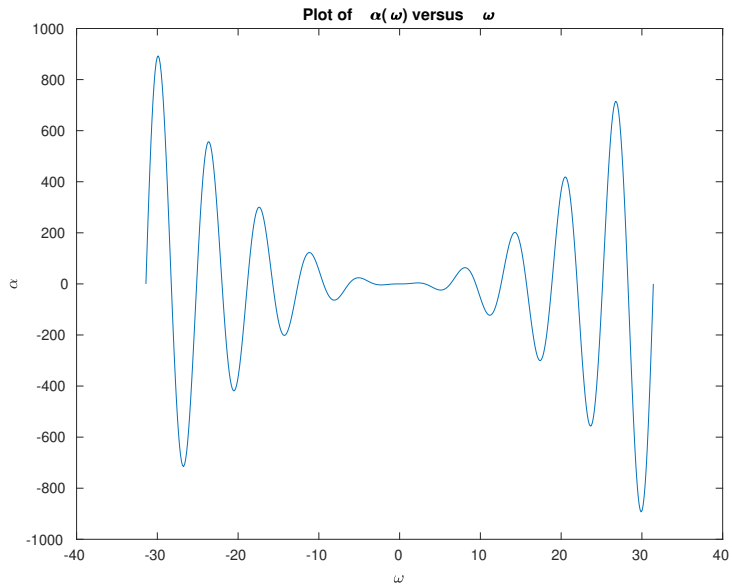

Figure D.2: Plot from example.

```
w = linspace(-10 * pi, 10 * pi, 500);
a = abs(w) .^ 2 .* sin(w);

% Clear the current figure.
clf
% Plot the data.
plot(w, a);

% Set the title, x-axis label, and y-axis label for the plot.
title('Plot of \alpha(\omega) versus \omega');
xlabel('\omega');
ylabel('\alpha');
```

The resulting plot is shown in Figure D.2. ■

Example D.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);

% Clear the current figure.
clf

% Plot sin(t) versus t using a dashed red line.
plot(t, sin(t), 'r--');

% Prevent subsequent plots from erasing the current one.
hold on

% Plot cos(t) versus t using a solid blue line.
plot(t, cos(t), 'b-');

```


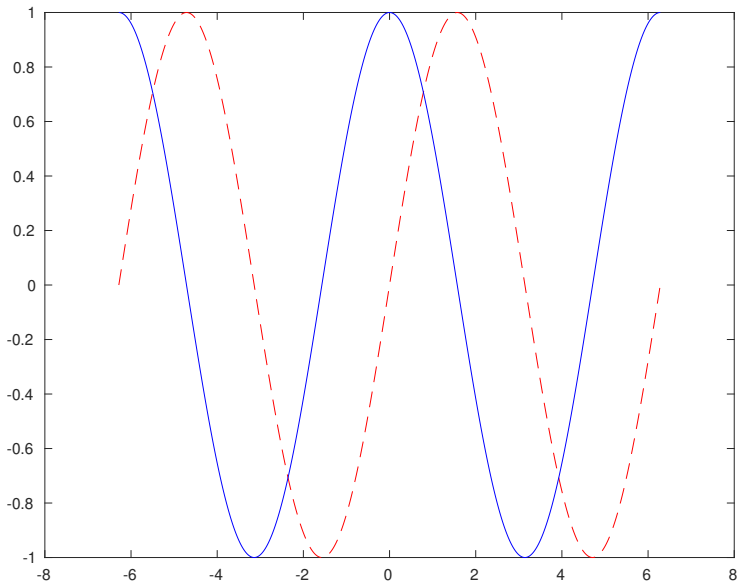

Figure D.3: Plot from example.

```
% Allow subsequent plots to erase the current one.
hold off
```

The resulting plot is shown in Figure D.3. ■

Example D.24 (Multiple axes on same figure)**.** Suppose that we would like to plot the functions $\cos t$, $\sin t$, $\arccos t$, and $\arcsin t$ using four separate plots in the same figure. This can be done using the following code:

```
% Select the sample points for the subsequent plots.
t = linspace(-pi, pi, 500);

% Clear the current figure.
clf

% Select the first subplot in a 2-by-2 layout.
subplot(2, 2, 1);
% Plot cos(t) versus t.
plot(t, cos(t));
title('cos(t)')

% Select the second subplot in a 2-by-2 layout.
subplot(2, 2, 2);
% Plot sin(t) versus t.
plot(t, sin(t));
title('sin(t)');

% Select the sample points for the subsequent plots.
t = linspace(-1, 1, 500);

% Select the third subplot in a 2-by-2 layout.
subplot(2, 2, 3);
% Plot acos(t) versus t.
```


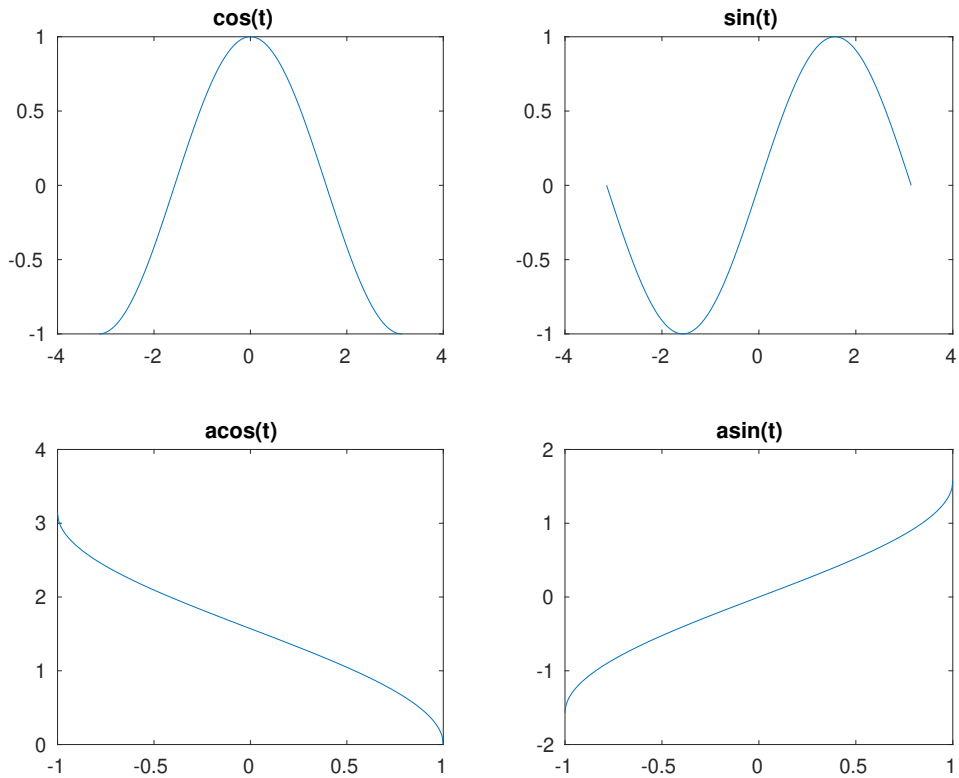

Figure D.4: Plot from example.

```
plot(t, acos(t));
title('acos(t)')

% Select the fourth subplot in a 2-by-2 layout.
subplot(2, 2, 4);
% Plot asin(t) versus t.
plot(t, asin(t));
title('asin(t)')
```

The resulting graphs are shown in Figure D.4. ■

D.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`.

D.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.

D.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, constants, 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.

D.14.2 Creating Symbolic Objects

A symbolic variable is created using either the `sym` function or `syms` directive. For example, a symbolic variable `x` (whose value is simply itself) can be created using the `sym` function with the code:

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

The same result can be achieved in a less verbose manner using the `syms` directive as follows:

```
syms x;
```

Also, the `syms` directive allows multiple variables to be specified. So, a single invocation of this directive can be used to create multiple variables. For example, symbolic variables named `x`, `y`, and `z` can be created with the code:

```
syms x y z;
```

The `sym` function can also be used to create symbolic constants. For example, we can create a symbolic constant `p` that has the value π with the code:

```
p = sym(pi);
```

Note that the character string argument passed to the `sym` function can only contain a variable name or constant. It cannot be an arbitrary expression. For example, code like the following is not allowed:

```
f = sym('a * x ^ 2 + b * x + c');
  % ERROR: cannot pass arbitrary expression to sym function
```

From symbolic variables and constants, more complicated symbolic expressions can be constructed. For example, a symbolic expression `f` with the value `a * x ^ 2 + b * x + c` can be created with the code:

```
syms a b c x;
f = a * x ^ 2 + b * x + c;
```

As another example, a symbolic expression `f` with the value `pi * r ^ 2` can be created with the code:

```
syms r;
f = sym(pi) * r ^ 2;
```

D.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:

```
syms t;
f = 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:

```
syms t;
f = t ^ 2;
g = subs(f, t, t + 1)
```

After executing the preceding code, `g` is associated with the expression `(t + 1) ^ 2`.

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:

```
syms t;
f = t ^ 2 + 3 * t + 2;
g = factor(f)
```

After executing the preceding code, `g` is associated with the (row-vector) 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 `2 * t + 1` for `t` in the expression `t ^ 2 - 1` and simplify the result. This can be accomplished with the following code:

```
syms t;
f = t ^ 2 - 1;
g = simplify(subs(f, t, 2 * t + 1))
```

After executing the preceding code, `g` is associated with the expression `(2 * t + 1) ^ 2 - 1`.

To expand an expression, use the `expand` function. For example, to compute $(t+1)^5$, we can use the following code:

```
syms t;
f = (t + 1) ^ 5;
g = expand(f)
```

After executing the preceding code, `g` is associated with the expression:

```
t ^ 5 + 5 *t ^ 4 + 10 * t ^ 3 + 10 * t ^ 2 + 5 * t + 1
```

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

```
syms t;
f = ((1/2) * t^2 + (1/3) * (t+1))^(-4);
pretty(expand(f))
```

The output of the `pretty` function in this case might look something like:

```
                                1
-------------------------------------------------------
 8     7     6        5        4        3       2
t     t     t     11 t     65 t     22 t     4 t     4 t     1
-- + -- + -- + ----- + ----- + ----- + ---- + --- + --
16     6     3      27      162       81      27      81    81
```


Example D.25 (Sum of an arithmetic sequence)**.** 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 sequence). 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:

```
syms a d k n;
simplify(symsum(a + k * d, k, 0, n - 1))
```

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

Example D.26 (Derivatives)**.** Use MATLAB to compute each of the derivatives given below.
(a) $\frac{d}{dt}\left[e^t \cos(3t)\right]$; and
(b) $\frac{d}{d\omega}\left(\frac{\cos\omega}{\omega^2+1}\right)$.

Solution. (a) The derivative $\frac{d}{dt}\left[e^t \cos(3t)\right]$ can be evaluated with the code:

```
clear
syms t
pretty(simplify(diff(exp(t) * cos(3 * t), t)))
```

The result produced should resemble:

```
exp(t) (cos(3 t) - sin(3 t) 3)
```

(b) The derivative $\frac{d}{d\omega}\left(\frac{\cos\omega}{\omega^2+1}\right)$ can be evaluated with the code:

```
clear
syms omega
pretty(simplify(diff(cos(omega) / (omega ^ 2 + 1), omega)))
```

The result produced should resemble:

```
  sin(omega)    2 omega cos(omega)
- ---------- - ------------------
       2                  2     2
  omega  + 1       (omega  + 1)
```

■

Example D.27 (Definite and indefinite integrals)**.** Use MATLAB to compute each of the integrals given below.
(a) $\int_{t-1}^{t+1} (\tau+1)^2 d\tau$;
(b) $\int t\cos(2t)dt$;
(c) $\int_0^{\pi/3} \cos^2(t)dt$; and
(d) $\int \left(t^{-1}+t\right) dt$.

Solution. (a) The integral $\int_{t-1}^{t+1} (\tau+1)^2 d\tau$ can be evaluated with the code:

```
clear
syms t tau
pretty(simplify(int((tau + 1) ^ 2, tau, [(t - 1) (t + 1)])))
```


The result produced should resemble:

```
   2          8
2 t  + 4 t + -
              3
```

(b) The integral $\int t\cos(2t)dt$ can be evaluated with the code:

```
clear
syms t
pretty(simplify(int(t * cos(2 * t), t)))
```

The result produced should resemble:

```
cos(2 t)    t sin(2 t)
-------- + ----------
    4            2
```

(c) The integral $\int_0^{\pi/3}\cos^2(t)dt$ can be evaluated with the code:

```
clear
syms t pi
pretty(simplify(int(cos(t) ^ 2, t, [0 (pi / 3)])))
```

The result produced should resemble:

```
pi    sqrt(3)
-- + -------
 6       8
```

(d) The integral $\int\left(t^{-1}+t\right)dt$ can be evaluated with the code:

```
clear
syms t
pretty(simplify(int(t ^ (-1) + t, t)))
```

The result produced should resemble:

```
          2
         t
log(t) + --
          2
```

■

D.14.4 Plotting Symbolic Expressions

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

```
syms t;
ezplot(3 * t ^ 2 - 4 * t + 2);
```

The range of the independent variable may optionally be specified. For example, to plot the function $f(t)=3t^2-4t+2$ over the interval $[-1,1]$, we can use the code:

```
syms t;
ezplot(3 * t ^ 2 - 4 * t + 2, [-1 1]);
```


Table D.28: Some functions related to signal processing

Name	Description
`besself`	Bessel CT filter design
`bilinear`	bilinear transformation with optional frequency prewarping
`bode`	Bode frequency response of system
`butter`	Butterworth CT/DT filter design
`cfirpm`	complex and nonlinear-phase equiripple FIR filter design
`cheby1`	Chebyshev type-I CT/DT filter design
`cheby2`	Chebyshev type-II CT/DT filter design
`fir1`	linear-phase FIR DT filter design using windowing method
`fir2`	arbitrary-shape DT filter design using frequency-sampling method
`fircls`	linear-phase FIR filter design using constrained least-squares method
`firls`	linear-phase FIR filter design using least-squares error minimization
`firpm`	Parks-McClellan optimal equiripple FIR filter design
`freqs`	compute and optionally plot frequency response of CT system
`freqz`	compute and optionally plot frequency response of DT system
`fft`	compute forward DFT
`ellip`	elliptic or Cauer CT/DT filter design
`ifft`	compute inverse DFT
`impulse`	compute and plot impulse response of CT/DT system
`impz`	compute and plot impulse response of DT system
`lsim`	simulate time response of CT/DT system to arbitrary input
`maxflat`	generalized-Butterworth (i.e., maximally-flat) DT filter design
`mag2db`	convert magnitude to decibels
`step`	compute step response of CT/DT system
`stepz`	compute step response of DT system
`tf`	construct system model for CT/DT system

D.15 Signal Processing

MATLAB provides many functions that are helpful in the analysis of continuous-time (CT) and discrete-time (DT) signals and systems. Some of the more useful functions are listed in Table D.28. In what follows, we provide several examples to illustrate how some of these functions can be used.

D.15.1 Continuous-Time Signal Processing

In the sections that follow, we introduce some of the functionality in MATLAB that is useful for continuous-time signals and systems.

Most continuous-time LTI systems of practical interest are causal with rational transfer functions. For this reason, MATLAB has considerable functionality for working with such systems. Consider the rational transfer function

$$H(s) = \frac{\sum_{k=1}^{n} b_k s^{n-k}}{\sum_{k=1}^{m} a_k s^{m-k}} = \frac{b_1 s^{n-1} + b_2 s^{n-2} + \ldots + b_n}{a_1 s^{m-1} + a_2 s^{m-2} + \ldots + a_m}. \tag{D.1}$$

Typically, MATLAB represents such a transfer function using two vectors of coefficients, one for the $\{b_k\}$ and one for the $\{a_k\}$.

D.15.1.1 Frequency Responses

For a LTI system with a transfer function of the form of (D.1), the `freqs` function can be used to compute and optionally plot the frequency response of the system.

