

Appendix B

MATLAB Overview


This appendix provides a broad overview of a few choice features of MATLAB. It should be pointed out, however, that MATLAB is both a programming language as well as a software suite. Thus it is just as appropriate to refer to “MATLAB code” as to “open MATLAB.”

The choice of which features to include here primarily results from their usefulness in the 351L laboratory. Deeper help on these topics as well as the myriad of other functions in MATLAB can be obtained from a number of outside references, including those in the bibliography.

B.1 Basic MATLAB Interface

The MATLAB desktop consists of four basic windows:

- Command Window
- Command History
- Current Directory
- Workspace

A fifth important window—the Editor—can be opened by typing `edit` at the command prompt (`>>`). When running experiments with many windows open, it is sometimes helpful to dock the Editor into the MATLAB desktop. The Editor window has an icon similar to  that can be used to dock it with the main MATLAB desktop.

The Command Window is useful for executing files or making quick calculations. Lengthy processes are better suited to be written in a script—more on that later. The Command Window may be cleared by issuing the command `clc`.

The Command History logs all of the actions executed in the Command Window as well as sign-in and exit times from MATLAB. It is most useful for finding commands issued previously if there is a lot of output cluttering the Command Window. Also, you may select several lines from the command history and drag them into the Editor if you wish to save them into a script.

The Current Directory is familiar to Windows users. It should be noted, however, that MATLAB can only execute scripts from the Current Directory (and subdirectories) and the MATLAB root directory. Thus if you wrote code for Lab 1 and wanted to run it in your Lab 2 directory, it would be easiest to copy-paste the file.

The Workspace lists all of the variables (and their type) that are currently active. To clear the Workspace, use the command `clear all`. To delete a certain variable (call it `var`), use `clear var`. Also note that clearing the Command Window does not clear the Workspace and vice-versa.

B.1.1 Statements

Statements are entered in the Command Window at the command prompt: `>>`. The comment symbol in MATLAB is the percent sign: `%`. When MATLAB encounters this symbol, it ignores the remaining text on that line. Comments are useful for creating “headers” for scripts. It is recommended that you include some authorship information in every script you write.

Example Code:

```
>> % Kane Warrior
>> % Lab 1, Section 2
>> % Data Analysis and Plotting Script
>> % 17 July 2008
```

Results from statements issued are by default stored in the single variable named `ans`. This variable will show up in the Workspace and appear in the Command Window:

Example Code:

```
>> 3.45                                <Enter>
ans =
      3.4500
>>
```

Pressing the *Enter* key causes MATLAB to execute the statement in the command prompt.

Example Code:

```
>> sqrt(1.44)                          <Enter>
ans =
      1.2000
>>
```

If a statement is too long for a single line, one may issue a carriage return by typing three periods (`...`) and then the *Enter* key.

Example Code:

```
>> 2 + 6.35 + sqrt(36) ...              <Enter>
+ sqrt(49)                             <Enter>
ans =
      21.3500
>>
```

Note: From now on the explicit typing of `<Enter>` will be assumed to be understood and therefore omitted.

Multiple statements may be issued on one line by separating them with a comma. The statements are executed from left to right.

Example Code:

```
>> clear all
>> 5, ans + 1.1
ans =
      6.1000
>>
```

Variables are user-defined entities that allow the saving and recalling of information throughout a MATLAB session. Variables may be named according to certain conventions, the most important of which is that they must begin with a letter. All variables currently active appear in the Workspace. A quick list of all of the variable names can be produced in the Command Window with the **who** command. A more complete table with variable names, dimensions, sizes, and classes is obtained by using the plural form: **whos**.

Statements that are terminated with a semicolon (;) are executed, but the results are not shown in the Command Window. Any variables assigned are saved and may be displayed by typing that variable's name subsequently:

Example Code:

```
>> a = 25; b = sqrt(a) + 2.5;
>>
>> a, b
a =
      25
b =
      7.5000
>>
```

B.1.2 Help

MATLAB's biggest advantage over other programming languages is its library of functions. To aid users in finding a function, MATLAB has two *very* useful commands.

To see a list of functions that have a certain word in the title or description, use the **lookfor** command.

Example Code:

```
>> lookfor sinc
DIRIC   Dirichlet, or periodic sinc function
SINC    Sin(pi*x)/(pi*x) function.
INVSINC Desired amplitude and frequency response for invsinc
        filters
>>
```

To get explicit information on a single function whose name is known, the **help** function is extremely useful. It explains the syntax and purpose of every function in the MATLAB library.

Example Code:

```
>> help sqrt
SQRT    Square root.
```

```
SQRT(X) is the square root of the elements of X. Complex
results are produced if X is not positive.

See also sqrtm.

>>
```

B.1.3 M-Files

Files with extension `.m` are executable in MATLAB. There are two flavors of m-files: Script and Function. All m-files contain plain text, and as such can be edited with any text editor. MATLAB provides a customized text editor, as discussed above. However, you may read, write, and edit m-files with any plain text editor (such as Notepad, WinEdt, emacs, etc.).

Scripts

Scripts are helpful because they can lump dozens or even thousands of Command Line statements into a single action. The conventions for naming scripts are very similar to those for naming variables:

- **Do Not** begin filenames with numbers
- **Do Not** include punctuation anywhere in filenames
- **Do Not** use a filename that is already used for a MATLAB function or an existing variable

There are a few useful shortcut keys when writing and debugging scripts. The *F5* key saves and executes the entire active script. When dealing with long files, it is often useful to execute only the code you have just written. The *F9* key executes only the text that is currently highlighted.

Functions

There are thousands of functions in the MATLAB library. However, one may find it necessary to write a new function—often a collection of other functions—to suit one’s specific needs. Just like MATLAB native functions, user-defined functions have a strict yet simple syntax. To define a new function, there are four essential components that must be the first executable line in the m-file:

- The MATLAB command `function`
- Output variables
- The user-defined function name
- Input variables

Output variables are contained in [square brackets] and separated by commas. Input variables are contained in (parentheses) and also separated by commas. Comments may appear above the line containing the `function` command, and these will appear when the `help` command is issued in relation to this new function.

Example Code:

```
% Kane Warrior
% 17 July 2008
```

```

%
% VECTOR function
% INPUTS:
% x      A complex number
%
% OUTPUTS:
% m      Magnitude of x
% theta  Angle of x in degrees counterclockwise from positive
%        Real axis
%
function [m,theta] = vector(x)

m=abs(x);                %Use the native function abs.
theta=angle(x)*180/pi;    %Use the native function angle and
                        % convert from radians to
                        % degrees.

```

B.1.4 Storage and Retrieval Commands

Data can be stored as variables, which can then be saved to memory. The syntax is: `save filename <variables>`. This saves the variables listed in `<variables>`—separated by spaces—as a file called `filename.mat` in the current directory. If no variables are specified, all of the variables in the Workspace are saved.

Upon clearing the Workspace, the variables can be restored by issuing `load filename`. This will restore the variables saved in `filename.mat` to the workspace and overwrite any existing variables with the same name.

B.1.5 Numeric Format

There are several types of numbers recognized by MATLAB. So far we have only dealt with rationals. We will, however, need more general numbers (such as non-repeating decimals and complex numbers).

The default setting in MATLAB is for numbers to be truncated after four digits to the right of the decimal and nine to the left. After this, the number will be presented in scientific notation. If this is not convenient, the command `format long` will always give 15 digits to the right of the decimal. To revert, use `format short`.

Floating Point Numbers

Floating point numbers represent real numbers. They are how computers interpret what we know as 2, $1/2$, and $\log 2$. Floating point numbers come in various *precisions*, the most common of which are single and double. The default for MATLAB is double precision. This should not become an issue in this course, so we will leave it at that.

As an example, let us issue the following commands:

Example Code:

```

>> s = 0.5555;
>> l = 0.55555

```

```

1 =
    0.5555
>> format long
>> s = 1
ans =
   -4.9999999999999449e-005

```

Complex Numbers

One feature that sets MATLAB apart from other languages is how it deals with complex numbers. Upon startup, the variables `i` and `j` are predefined to both be $\sqrt{-1}$. Using these variables, we can define complex numbers. MATLAB, in turn, saves them in memory as a single variable with a real and a complex part. For example, we create a complex variable below.

Example Code:

```

>> c = 3 + 1j
c =
    3.0000 + 1.0000i

```

The Workspace resulting from the previous section and this action is shown in Figure B.1.

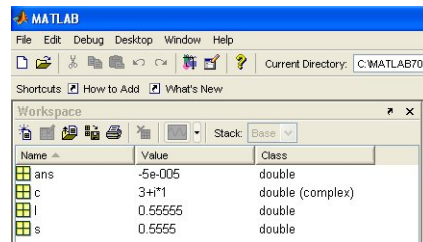


Figure B.1: Example workspace showing floating point real and complex numbers

The built-in functions listed in Table B.1 are particularly useful when dealing with complex numbers.

| <i>Function</i> | <i>Example</i> |
|-----------------------|---------------------|
| <code>real(c)</code> | 3 |
| <code>imag(c)</code> | 1 |
| <code>abs(c)</code> | 3.1623 |
| <code>angle(c)</code> | 0.3218 ^a |

Table B.1: Common built-in MATLAB functions dealing with complex arguments. Here the examples use $c = 3 + i$.

^aAs with all built-in MATLAB angular functions, the default is always radians.

One must be careful because the variable names `i` and `j` can be reassigned by the user to be something *other* than $\sqrt{-1}$. Say, for example, that on lines 10–20 in a script there is a `for` loop that uses `i` as its index and runs up to `i = 100`. Then on line 25, the script tries to execute complex

variable assignment `x = 4+5*i;`. Of course, unless the variable `i` was cleared between lines 20 and 25, the script will create `x = 504` and not the desired `x = 4.0000 + 5.0000*i`. The moral is to be very careful when using `i` and `j` as non-native variables.

Arrays

Arrays are the heart and soul of MATLAB (MATrix LABoratory). Arrays are groups of numbers that travel around together. They may be grouped so they stay in the right order, so that they perform some mathematical operation, or any other reason. We are all familiar with one- and two-dimensional arrays. We already know them as *vectors* and *matrices*, respectively. However, MATLAB can handle higher-dimensional arrays; for these it is easier to think of every two-dimensional object as being a matrix stacked among others.

Arrays are defined using square brackets: `[]`. Entries in a row may be separated by either a comma or whitespace (space bar). A row is terminated and a new one begun with either a semicolon or a line break within the brackets.

Example Code:

```
>> rvec = [1 2 3 4]
rvec =
     1     2     3     4
>> cvec = [11; 12; 13; 14]
cvec =
    11
    12
    13
    14
>> TwoDimArray = [1 2 3
4 5 6
7, 8, 9]
TwoDimArray =
     1     2     3
     4     5     6
     7     8     9
```

In MATLAB, array dimensions are always listed with rows first and columns second. For example, if we wanted to call out the 6 from the variable `TwoDimArray`, we would enter

Example Code:

```
>> TwoDimArray(2,3)
ans =
     6
```

This is because the 6 lies in the second row and the third column. Some helpful array creation and basic modification commands are listed in Table B.2.

We can also make an array by concatenating smaller arrays. Special attention must be paid to the dimensions in order to make a valid concatenation.

Example Code:

```
>> h = [1 2 3], k = [4; 7], m = [5 6; 8 9]
h =
```

| <i>Function</i> | <i>Description</i> | <i>Example</i> |
|----------------------------------|--|---|
| <code>ones(m,n)</code> | Create an m by n array filled with ones | <code>ones(2,3) =</code> $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ |
| <code>zeros(m,n)</code> | Same as <code>ones</code> , but filled with zeros. | <code>zeros(3,2)=</code> $\begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$ |
| <code>transpose(A) or A.'</code> | Transpose the array A | <code>rvec.' =</code> $\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$ |
| <code>size(A)</code> | Return the dimensions of A in an array | <code>size(cvec) =</code> 4 1 |
| <code>length(A)</code> | Returns the maximal dimension of A | <code>length(rvec) =</code> 4 |
| <code>:</code> | Used to span indices, "to" | <code>TwoDimArray(1:2,2:3)</code> $= \begin{bmatrix} 2 & 3 \\ 5 & 6 \end{bmatrix}$ |

Table B.2: Common built-in MATLAB functions useful for defining and manipulating arrays. The examples use the variables created in Section B.1.5.

| | | |
|------------------------------------|----------------|----------------|
| <code>1</code> | <code>2</code> | <code>3</code> |
| <code>k =</code> | | |
| <code>4</code> | | |
| <code>7</code> | | |
| <code>m =</code> | | |
| <code>5</code> | <code>6</code> | |
| <code>8</code> | <code>9</code> | |
| <code>>> n = [h; k m]</code> | | |
| <code>n =</code> | | |
| <code>1</code> | <code>2</code> | <code>3</code> |
| <code>4</code> | <code>5</code> | <code>6</code> |
| <code>7</code> | <code>8</code> | <code>9</code> |

Vectors, as we have seen, are one-dimensional examples of arrays. To create vectors with equally spaced elements, the colon (`:`) notation is helpful. The syntax is *start value* : *step size* : *end value*. The default step size is 1.

| | | |
|---|----------------|----------------|
| <code>>> c = 2:5, d = 9:-1:3</code> | | |
| <code>c =</code> | | |
| <code>2</code> | <code>3</code> | <code>4</code> |
| <code>d =</code> | | |
| <code>9</code> | <code>8</code> | <code>7</code> |
| | <code>6</code> | <code>5</code> |
| | <code>4</code> | <code>3</code> |

A useful trick is to use a vector to call indices from another vector. For example, using the above variable `c`, we can call the second through the fifth values from `d` in the following way:

Example Code:

```
>> d(c)
ans =
     8     7     6     5
```

The following are more complex examples of arrays and indices.

Example Code:

```
>> m = [1.5 -2.4 3.5 0.7; -6.2 3.1 -5.5 4.1; 1.1 2.2 -0.1 0]
m =
     1.5     -2.4     3.5     0.7
    -6.2     3.1    -5.5     4.1
     1.1     2.2    -0.1     0
>> n = m(1:2,2:4), o = m(:,1:2), p = m(2,:)
n =
    -2.4     3.5     0.7
     3.1    -5.5     4.1
o =
     1.5     -2.4
    -6.2     3.1
     1.1     2.2
p =
    -6.2     3.1    -5.5     4.1
```

Strings

Strings (or sometimes character strings) are very much like vectors. The difference is that the entries are characters instead of numbers. Strings are defined by single quotation marks.

Example Code:

```
>> str = 'Signal and System Analysis';
>> disp(str)
Signals and System Analysis
>> str(1:6)
Signal
```

Strings are useful when we wish to load several files that have been given sequential names (like `data1.dat`, `data2.dat`, etc.). In this case, a `for` loop may be created that executes commands on the string `data` concatenated with the index 1, 2, etc.

Logicals

When MATLAB performs a logical operation (see Section B.2.2), the output is always binary in nature. That is, the output is an array of 1's and 0's. This output is then dual-purpose:

It contains numbers in indexed positions (like arrays) but it also contains “true/false” values in indexed positions. In this sense, logical arrays are very handy because they serve dual purposes and may be processed very quickly.

There is a key difference between arrays and logicals regarding indexing. When using an array to call the index, the value of the elements of the array is important. When using logicals, only the position of the “true” elements is important.

Example Code:

```
>> a = 2:1:5, b = a>3
a =
     2     3     4     5
b =
     0     0     1     1
>> a(b)
ans =
     4     5
>> a([3:4])
ans =
     4     5
>> notlogical = [0 0 1 1];
>> a(notlogical)
??? Subscript indices must either be real positive integers
    or logicals.
```

The last line is an error because MATLAB thinks we are looking to create an array with the first two elements being the 0th element of **a** and the third and fourth elements as the 1st element of **a**. When MATLAB looks for the 0th element of **a**, it gets confused because indices can only be positive integers (1, 2, 3, ...).

Cells

Cells are convenient ways to store objects of several different types. They are created and their indices are called with {curly braces} and act very much like arrays. However, a one-dimensional cell may have an entire matrix at position 1 but a single character at position 2. Cells do not have a clear analog like arrays do to matrices, so it may be more convenient to simply think of them as “multimedia” arrays.

Example Code:

```
>> M = [1 2; 3 4];
>> str = 'This is my matrix';
>> cell1 = {M str}
cell1 =
    [2x2 double]    'This is my matrix'
>> cell1{1}
ans =
     1     2
     3     4
>>
```

Special Numbers

In addition to the imaginary numbers `i` and `j`, MATLAB has two other special predefined variables. They are `pi` and `Inf`. The variable `pi` is MATLAB's closest estimate at the irrational number π . The variable `Inf` represents infinity and results from actions like dividing by zero or that result in a number larger than MATLAB can handle.

Furthermore, operations like $\frac{0}{0}$ and $\sin \infty$ produce undefined results and are represented by `NaN`, which stands for “not a number.”

B.2 Operations

The operations in the above sections have been minimal and intuitive. Here we look at some of the more advanced operations MATLAB can perform.

B.2.1 Arithmetic Operations

MATLAB defines all arithmetic operations in matrix terms. Note that a single number may be either a scalar or a one-by-one matrix, but its operations are all treated as matrix ones. Issuing `help arith` and `help slash` will list the available arithmetic operations.

As one would expect, the *order* of operations is often important. We have seen that MATLAB executes commands from left to right. However, within a command, MATLAB follows the familiar *PEMDAS* (parentheses, exponents, multiplication and division, then addition and subtraction) order.

It is also important to observe that the operators “slash” `/` and “backslash” `\` represent two different matrix operations. For example, `B/A` is right division and represents BA^{-1} whereas `A\B` is left division and represents $A^{-1}B$. We know that these may be two *very* different quantities, so it is important to differentiate between them.

Example Code:

```
>> a = [1 2; 3 4]; b = [3 1; 7 8]; c = [2 4];
>> d = a + b, e = c*a, f = a^2, g = c'
d =
     4     3
    10    12
e =
    14    20
f =
     7     10
    15     22
g =
     2
     4
>> h = a\b, k = b/a
h =
    1.0000    6.0000
    1.0000   -2.5000
k =
   -4.5000    2.5000
   -2.0000    3.0000
```

To use an operator on an element-by-element basis rather than its matrix math sense, use the modifier of a period (.) prior to the operator. This tells MATLAB to perform the operation among entries with the same location in their respective arrays.

Example Code:

```
>> m = a.*b, n = b./a, o = b.^a
m =
     3     2
    21    32
n =
    3.0000    0.5000
    2.3333    2.0000
o =
     3     1
    343  4096
```

B.2.2 Logical Operators

The logical operations AND, OR, and NOT are specified by ampersand (&), pipe (|), and tilde (~) respectively. They can be used in conjunction with the relation operators listed in Table B.3 to construct logical arrays as in the examples below. As usual, “true” is represented by 1 and “false” by 0.

| <i>Relation</i> | <i>Symbol</i> |
|--------------------------|---------------|
| Less than | < |
| Less than or equal to | <= |
| Greater than | > |
| Greater than or equal to | >= |
| Equals exactly | == |
| Does not equal | ~= |

Table B.3: Relation operator symbols

Example Code:

```
>> a = [1 3 2; 4 6 5], b = a>2 & a<=5
a =
     1     3     2
     4     6     5
b =
     0     1     0
     1     0     1
>> c = [1 5 3 4 7 8], d = c>4
c =
     1     5     3     4     7     8
d =
     0     1     0     0     1     1
```

Further, logical arithmetic operations may be mixed in a single command. Priority is given to the arithmetic operations first, however.

Example Code:

```
>> 5 + 2 > 6 * 8
ans =
    0
>> 5 + 2 > 6
ans =
    1
>> 5 + (2>6) * 8
ans =
    5
```

B.2.3 Mathematical Operations

Another of MATLAB's strengths is its vast library of built-in mathematical functions. These range from utterly simple to quite complicated. A sampling of commonly useful math operations is listed in Tables B.4a and B.4b.

B.2.4 Data Analysis Operations

One may use MATLAB to perform analysis on sets of data collected elsewhere. There are several built-in MATLAB functions that calculate certain statistics or ease the process of data analysis. Table B.5 contains a sample of these.

B.3 Flow Control

Like many other programming languages, MATLAB uses flow control statements to repetitively or selectively execute statements. Once activated, all statements are executed until the **end** command.

B.3.1 for Statement

The **for** statement executes commands a fixed number of times. It is equivalent to the **do** command found in other languages. For example, to evaluate the summation

$$x(t) = \sum_{k=1}^3 t^{\sqrt{1.2k}} \sqrt{k}$$

for times between 0 and 0.8 seconds at intervals of 0.2 seconds, we would execute:

Example Code:

```
t = 0 : 0.2 : 0.8;
x = zeros(size(t));
for k = 1 : 3
    x = x + sqrt(k)*t.^sqrt(1.2*k);
end
```

| <i>Command</i> | <i>Description</i> |
|---------------------------|---------------------------------------|
| <code>exp(x)</code> | Base e exponent of x |
| <code>log(x)</code> | Base e logarithm of x |
| <code>log10(x)</code> | Base 10 logarithm of x |
| <code>complex(a,b)</code> | Construct the complex number $a + bi$ |
| <code>conj(x)</code> | Complex conjugate of x |
| <code>round(x)</code> | Round x to the nearest integer |
| <code>floor(x)</code> | Round x toward $-\infty$ |
| <code>ceil(x)</code> | Round x toward ∞ |

(a) Commonly used built-in mathematical functions

| <i>Command</i> | <i>Description</i> |
|-------------------------|--|
| <code>sin(x)</code> | $\sin x$ |
| <code>cos(x)</code> | $\cos x$ |
| <code>tan(x)</code> | $\tan x$ |
| <code>asin(x)</code> | $\arcsin x$ |
| <code>acos(x)</code> | $\arccos x$ |
| <code>atan(x)</code> | $\arctan x$ |
| <code>atan2(y,x)</code> | $\arctan(yi + x)$ Restricted to $-\pi$ to π |

(b) Commonly used built-in trigonometry functions

Table B.4

```
>> x
x =
    0    0.3701    1.0130    1.8695    2.9182
```

In the above example, we *initialized* the variable `x` to be all zeros before we executed the `for` loop. This is good programming practice because in larger or nested loops, it is faster for the system to assign values to already existing elements, rather than resize the variable on every iteration.

Nested for loops involve executing entire sub-loops on every iteration of outer loops. In the example below, `m` starts at 1, and `n` runs from 1 to 4 before `m` advances to 2; Then the process repeats.

Example Code:

```
y = zeros(3,4);
for m = 1 : 3
    for n = 1 : 4
        y(m,n) = m + n;
    end
end
```

| <i>Function</i> | <i>Description</i> | <i>Example</i> |
|---------------------------|--|---------------------------------|
| <code>find(a)</code> | Returns indices of nonzero elements | <code>find(a) = 1 3 4 6</code> |
| <code>find(a>b)</code> | Returns indices for which logical expression is true | <code>find(a>2) = 4 6</code> |
| <code>max(a,[],d)</code> | Returns the maximum elements along the direction specified | <code>max(a) = 4</code> |
| <code>min(a,[],d)</code> | Returns the minimum elements along the direction specified | <code>min(a) = 0</code> |
| <code>mean(a,d)</code> | The average of the elements | <code>mean(a) = 1.6667</code> |
| <code>sum(a,d)</code> | The sum of the elements | <code>sum(a) = 10</code> |

Table B.5: Commonly used data analysis functions. All relevant examples operate on the vector $a = (1, 0, 2, 3, 0, 4)$. The optional arguments `[]`, `d` can be used to specify the dimension on which the function is to act. In all cases `d = 1` operates on columns and returns a row, and `d = 2` is the opposite.

```
>> y
y =
     2     3     4     5
     3     4     5     6
     4     5     6     7
```

It is often important to remember to place the semicolon after each line inside of the loops. If omitted, MATLAB will display that line every time it is iterated. This can significantly slow the execution time of a script.

B.3.2 while Statement

The **while** statement is similar to the **for** statement except that the loop terminates when a logical condition is satisfied. To find the largest power of 2 that is less than 5,000, we might use

Example Code:

```
n = 1;
while n*2 < 5000
    n = n*2;
end

>> n
n =
    4096
```

Care should be taken when using **while** loops because there is a possibility that poorly stated logical conditions may never be satisfied; this results in an endless loop.

Important: Should you find yourself in an infinite loop, press Control-c in the Command Window to terminate the current process.

Another common use of **while** statements is to perform tasks while a certain variable counts down towards zero. Here it is appropriate to introduce a concept called “machine zero.” MATLAB systems have a least counting element, it is the smallest quantity that the system can differentiate between numbers. On most modern installations, it is around 10^{-16} . So if a **while** loop is set to run until the counter reaches zero, it will actually run until the counter reaches $\pm 10^{-16}$.

This may be a problem if the counter only asymptotically approaches zero. It may take many iterations to reach the machine zero. In this case, it is useful to define a number that is “close enough” to zero for the loop to terminate. This value, called the stopping point, represents a trade-off between absolute accuracy and execution speed.

B.3.3 if Statement

The **if** statement allows us to selectively execute statements based on the outcome of a logical expression. Further, they can be nested within other loops as well as other **if** statements.

Example Code:

```
for k = 1 : 4
    if k == 1
        x(k) = 3*k;
    else
        if k == 2 | k == 4
            x(k) = k/2;
        else
            x(k) = 2*k;
        end
    end
end

>> x
x =
     3     1     6     2
```

In addition, **if** statements are often used to issue warnings or check for data errors before performing intense calculations.

Example Code:

```
c = 't'; n = 2;
if c == 'f'
    c = 'false';
    y = NaN;
end
d = 0.1 : 0.1 : 0.4;
if c == 't'
    if n == 2
        y = 100*d(n);
    elseif n == 1
        y = 10*d(n);
    end
end
```



```

        else
            y = 0;
        end
    end
end

```

Which of course yields $c = t$ and $y = 20$.

When used with `if` statements, the `break` command is very useful. If a loop is to be executed a variable number of times, `if` statements can be used to check the status of certain criteria, and then the `break` command can be issued to halt the loop.

B.4 Plotting

Plotting—in two dimensions or three—is an extremely useful tool in the analysis and presentation of data. Even if they are never destined to be published, plots often give a “big picture” idea of a data set that is more valuable than numerical or statistical analysis alone.

As such, MATLAB has a wealth of graphical display functions from the urbane to the imaginative. The most basic is the `plot(x,y)` command. This places a connected plot of the data pairs $x(k), y(k)$ for all k in the current figure window. The following three lines of code generate the plot in Figure B.2.

Example Code:

```

x = linspace(0, 2*pi);
y = sin(x);
plot(x,y);

```

Of course, the plot in Fig. B.2 is presentable and the ease with which it was generated is nice. But it is lacking somewhat in presentation (what are we looking at?) and functionality (why are we looking at it?). As such, MATLAB provides many more commands to create detailed graphs and dress them up nicely.

In particular, the `plot` command itself has many optional arguments. Issuing `help plot` details all of the different line and marker styles available.

We now expand on the three-line example above to incorporate some of the commands detailed in Table B.6. The resulting graph is shown in Figure B.3.

Example Code:

```

x = linspace(0, 2*pi);
y1 = sin(x);
y2 = cos(x);
ax = [0 2*pi -1.5 1.5];
figure;
subplot(2,1,1)
plot(x,y1,'o')
axis(ax)
title('Basic Sinusoids')
ylabel('y-axis')
legend('Sin(x)')
subplot(2,1,2)

```

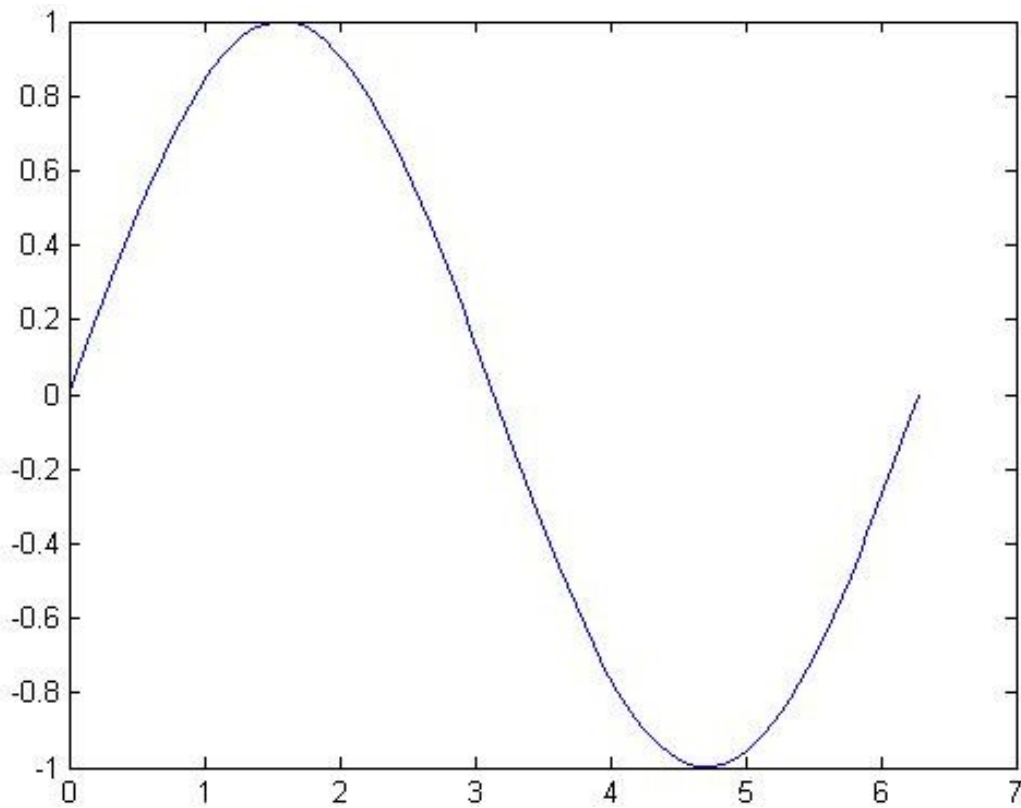


Figure B.2: Simple plot generated from three lines of code.

| <i>Command</i> | <i>Description</i> |
|---------------------------------------|---|
| <code>figure(a)</code> | Makes <code>a</code> the active figure window |
| <code>hold on</code> | Places subsequent plot commands on same axes instead of replacing them |
| <code>hold off</code> | Releases <code>hold on</code> command |
| <code>title('text')</code> | Places <code>text</code> as a title above the plot |
| <code>xlabel('text')</code> | Labels x-axis of plot with <code>text</code> |
| <code>ylabel('text')</code> | Labels y-axis of plot with vertically rotated <code>text</code> . |
| <code>text(x,y,'text')</code> | Places <code>text</code> on the plot at point (x,y) |
| <code>legend('First','Second')</code> | Places a legend that labels two plots on the same axes |
| <code>axis([x1 x2 y1 y2])</code> | Scales the x- and y-axes to fit between x_1 , x_2 , y_1 , and y_2 |
| <code>subplot(m,n,id)</code> | Splits the figure window into an m by n array of plots with plot number <code>id</code> as the active plot ^a |

Table B.6: Common plot editing commands

^aIn a `subplot` array, the plots are numbered starting at 1 in the upper left and count across the first line, then across the second line, etc.

```

plot(x,y2,'+-')
axis(ax)
xlabel('x-axis')
ylabel('y-axis')
legend('Cos(x)')

```

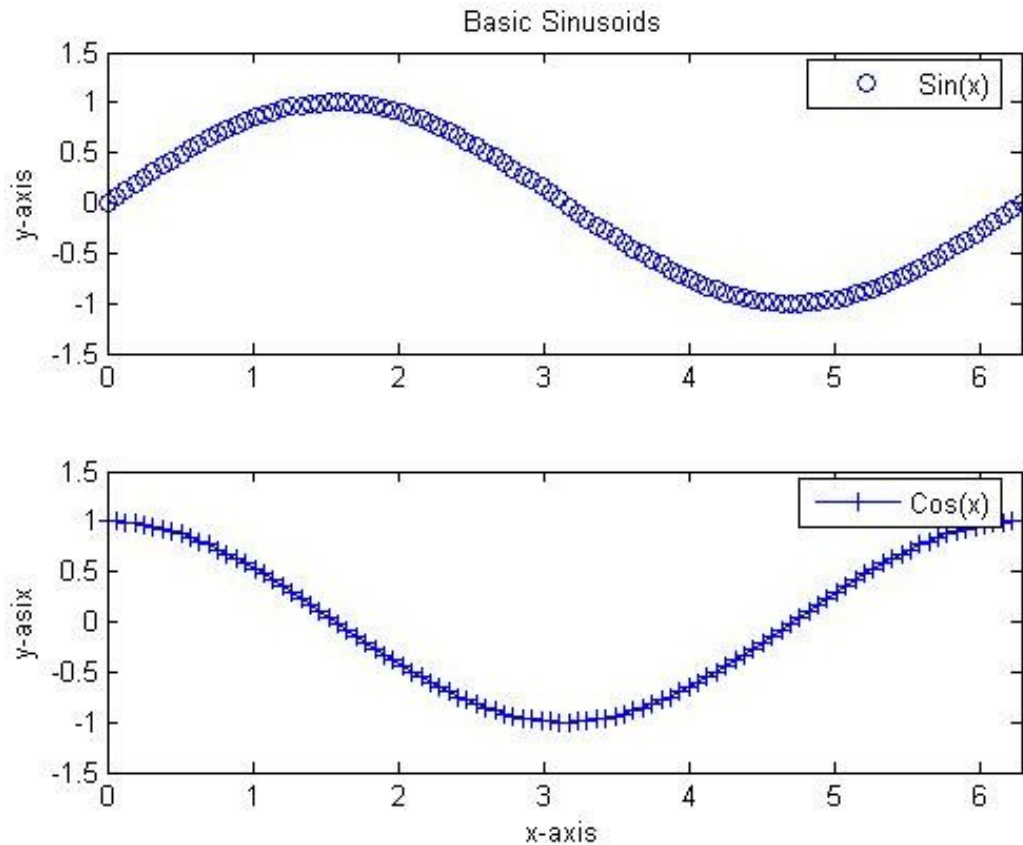


Figure B.3: Embellished plots using tools from Table B.6

B.5 Signals and Systems Applications¹

In this section, we present several function m-files that have been created specifically for signal and system analysis. On the laboratory computers, these m-files are located in

w:\matlab\toolbox\ee351L

These functions encompass

- Step and ramp signal functions

¹Section B.5 is taken directly from the first edition of this laboratory manual. It's author is unknown.

- Discrete- and continuous-time Fourier series functions
- Continuous-time Fourier transform function

All of the functions contain help statements at the beginning. These statements indicate the function's purpose, define input and output variables, and in a few cases provide useful modification suggestions and requirements.

B.5.1 Step and Ramp Functions

We use step and ramp functions often in signal and system analysis. Therefore, we have created the MATLAB functions `us` and `ur` to compute them. The syntax looks like

```
u = us(t)
r = ur(t)
```

The step function `us` returns $u = 0$ for $t < 0$ and $u = 1$ for $t \geq 0$. Likewise, `ur` returns $r = 0$ for $t < 0$ and $r = t$ for $t \geq 0$.

B.5.2 Continuous-Time Fourier Series Functions

We have created four functions for use in performing Fourier analyses of continuous-time signals. The first function computes Fourier series coefficients from sample values of a continuous-time signal. The second function computes samples of the truncated Fourier series approximation to a signal over a specified time interval. The last two functions compute the Fourier transform and the inverse Fourier transform, respectively.

Fourier Series Coefficients

The function

```
[Xn, f, ang, No, Fo] = ctfsc(t,x)
```

computes the Fourier series expansion coefficients `Xn` corresponding to the portion of the signal `x` within the expansion interval $t(1) - dt/2$ to $t(ns) + dt/2$. This interval has length `ns*dt` where `ns = size(t,2)` is the number of signal samples and `dt` is the time interval between samples.

There are `ns` Fourier series coefficients computed and the frequency interval between these is `Fo = 1/(ns*dt)`. Series coefficients are computed for both positive and negative frequencies and the coefficient X_0 corresponding to $f = 0$ is stored in `Xn(No)`. The values of both `No` and `Fo` are function outputs along with `Xn`.

The Fourier series expansion interval can begin anywhere between $-10*(ns*dt)$ to $10*(ns*dt)$. These limits can be changed, if required, as indicated in the help statements in the m-file.

Truncated Fourier Series

The second Fourier series function is

```
[xfs, Xnn] = ctfs(t,Xn,no,fo,N)
```

This function first selects the $2N+1$ Fourier series coefficients, `Xnn`, centered on X_0 . Then the function computes samples, `xfs`, of the truncated Fourier series approximation of x over the time interval specified by the input variable `t`. The time interval does not need to be the same as the expansion interval used to compute the Fourier series coefficients.

The input variables **Xn**, **no**, and **fo** are the output variables obtained from the function **ctfsc**. This function also plots the Fourier series. The original function can also be plotted on the same graph by using the **hold on** command after **ctfs** followed by the **plot** command.

Fourier Transform

MATLAB contains a built-in function for the Fourier transform called **fft**. Using this function requires knowledge of discrete-time concepts which students in this laboratory are not expected to know. We have therefore written a function which attempts to create the best approximation possible to a continuous-time Fourier transform.

The function

```
[f, X, N, no] = ctft(t,x,dfm)
```

computes the Fourier transform of the portion of the signal **x** contained in the interval $t(1) - dt/2$ to $t(ns) + dt/2$. This interval has length $ns*dt$ where $ns = \text{size}(x,2)$ is the number of signal samples and dt is the time interval between samples. The signal portion used must begin at $t(1) \leq 0$ and end at $t(ns) \geq 0$. If the signal is longer than the time interval $ns*dt$, then the computed transform will have some distortion since it is the transform of the truncated signal.

There are **N** Fourier transform samples computed, where **N** is the larger of $\text{ceil}(1/(dfm*dt))$ or $2*ns$. The variable **dfm** is the maximum spacing that we will allow between computed transform samples. Note that care must be exercised in selecting **dfm** and **dt** since **N** can become very large if they are chosen to be quite small. Transform samples are computed for both positive and negative frequencies with a spacing of $df = 1/(N*dt)$. The transform value at $f = 0$ is stored in **X(no)**. The frequencies at the transform sample points are stored in the output vector **f**.

Now we can plot the magnitude and phase of the Fourier transform using MATLAB's plot capabilities. We can use the **subplot** command to plot the amplitude and angle of the transform on the same page:

Example Code:

```
subplot(2,1,1)
plot(f,abs(X))
subplot(2,1,2)
plot(f,angle(X))
```

Due to complications arising from the fact that we are actually using discrete-time signals in our computations, the phase of the Fourier transform may sometimes be different from what is expected. If this happens, first try using the **unwrap** command before plotting the phase. If the problem still exists, the problem may be that an insufficient portion of the signal is begin included in the samples. Try extending the length of the samples to include a longer portion of the signal.

Inverse Fourier Transform

MATLAB contains a built-in function for the inverse Fourier transform called **ifft**. Using this function requires knowledge of discrete-time concepts which students in this laboratory are not expected to know. We have therefore written a function which attempts to create the best approximation possible to a continuous-time inverse Fourier transform.

The function

```
[t, x, n] = ctift(f,X,dtm)
```

computes the inverse, x , of the Fourier transform X . The input vector f specifies the frequency interval and sample spacing for the input Fourier transform vector X . These input vectors are outputs of the function `ctft`. The number of transform samples is $ns = \text{size}(X,2)$, and the number of inverse transform samples computed, N , is the larger of `ceil(1/(dtm*df))` or ns . The variable df is the spacing between transform samples, and the input variable dtm specifies the maximum spacing that we allow between computed inverse transform samples. As for the Fourier transform, we must exercise care in selecting the variables df and dtm so that N does not become excessively large.

If the magnitude of the transform is an even function of frequency and the phase of the transform is an odd function of frequency, then the inverse transform should be real. There will probably be some round-off error present, so in this case you should probably take the real portion of the inverse transform before plotting. The inverse transform can then be plotted using the standard MATLAB plot command:

Example Code:

```
x = real(x);  
plot(t,x)
```

B.5.3 Straight-Line Approximate Bode Plot Functions

Straight-line approximations to Bode amplitude response and Bode phase response plots are convenient in continuous-time system analyses. We have created the two functions `sysdat` and `slbode` to compute the parameters for the straight-line approximations and the straight-line approximation data, respectively. The location of the system zeros is not restricted. The functions are capable of finding the parameters and straight-line data for causal, stable systems (poles only in the LHP) and non-causal, stable systems (poles in both the LHP and RHP). Also, results for marginally stable systems can be obtained. These results are valid for input signals that do not contain poles located at single-order system poles on the imaginary axis.

Transfer Function Parameter Computation

The function

```
[rn, rd, imas, rhps, c, bf, ft, dr] = sysdat(n,d)
```

computes the transfer function, or frequency response, parameters for a system having the transfer function

$$H(s) = \frac{n(1)s^\alpha + \cdots + n(\alpha)s + n(\alpha + 1)}{d(1)s^\beta + \cdots + d(\beta)s + d(\beta + 1)}$$

or frequency response

$$H(j\omega) = \frac{n(1)(j\omega)^\alpha + \cdots + n(\alpha)j\omega + n(\alpha + 1)}{d(1)(j\omega)^\beta + \cdots + d(\beta)j\omega + d(\beta + 1)}$$

The numerator and the denominator coefficients are stored in the input vectors n and d , respectively. Function outputs are described in Table B.7.

A damping ratio only applies to a quadratic factor and is a number greater than zero but less than one. We set `dr(i)` equal to 10 for factor i as an indicator that factor i is a linear factor.

| <i>Output Variable</i> | <i>Description</i> |
|------------------------|--|
| rn | Roots of the numerator (system zeros) |
| rd | Roots of the denominator (system poles) |
| imas | Two-element row vector containing the number of zeros and the number of poles on the imaginary axis |
| rhps | Two-element row vector containing the number of zeros and the number of poles in the RHP |
| c | Constant multiplicative factor $c = n(i)/d(j)$, where i and j are the largest integers for which $n(i)$ and $d(j)$ are not equal to zero |
| bf | Row vector containing the break frequency for each factor where, for linear factors $\mathbf{bf} > 0$, $\mathbf{bf} = 0$, and $\mathbf{bf} < 0$ indicate LHP, imaginary axis, and RHP poles or zeros, respectively |
| ft | Row vector of factor types where +1 and +2 correspond to numerator linear and quadratic factors, and -1 and -2 correspond to denominator linear and quadratic factors |
| dr | Row vector of factor damping ratios |

Table B.7: Output argument descriptions for the custom function `sysdat`.

Quadratic factors that correspond to LHP, imaginary axis, and RHP poles are indicated with positive, zero, and negative damping ratios, respectively.

Note: If a factor is of order N , then the data for it will appear as data for N first order factors at the same break frequency. The data for each of these first-order factors is the same and the resulting straight-line approximation for the N^{th} -order factor is the sum of the straight-line approximations for the N first-order factors.

Straight-Line Data Computation

We compute the straight-line Bode plot approximation data with the function

```
[am, ph] = slbode(w,c,bf,ft,dr)
```

The input variables include the vector **w** defining the frequency interval and increment for which the data are computed. The other input variables are the frequency response factor parameters computed with function `sysdat`. These are defined in Table B.7.

The output variables are the vectors **am** and **ph**. These contain the data for the straight-line approximations to the amplitude-response and phase-response Bode plots, respectively.

Bibliography

- [1] Stephen J. Chapman. MATLAB Programming for Engineers, 2nd edition. Thomson Engineering, 2001
- [2] Mathworks website. <http://www.mathworks.com/>
- [3] EE351L Laboratory Manual, 1st edition.