EE216 Laboratory Manual
Appendix A - MATLAB® Tutorial

Contents

MATLAB Tutorial

This tutorial provides basic MATLAB information and specific application information for the EE266 Linear Systems I Laboratory. The MATLAB User's and Reference Guides should be used to obtain greater breadth and depth of information.

The tutorial is designed so it will work with the MATLAB Professional Version, plus the Signal Processing, Control System, and Symbolic Math Toolboxes, or the MATLAB Student Edition.

1.0 Basic MATLAB Information (close all, look for, help)

On initiation of MATLAB, the **Command Window, Workspace, Launch Pad. Command History**, and **Current Directory** windows appear. All exercises will use the **Command Window.** The **Workspace** and **Command History** windows show what variables are defined and past commands. These lists can also be accessed through the **Command Window** at the **Command Prompt (>>)**.

The **Command Window** can be cleared by the **clc** command or by clicking on **Edit -Clear Command Window**. This does not clear the **Workspace**. The **Workspace** can be cleared by the command **clear all** or by **Edit -Clear Workspace**.

Immediately after initiation of MATLAB, click on **Edit -Clear Command Window** to eliminate unneeded information at the top of the monitor screen. Also click on **File - Preferences - Command Window - Numeric Display** to minimize the vertical space required in the **Command Window**. This increases the amount of information visible on the monitor screen and reduces the size of **Command Window** printouts. **Command Window** printout is accomplished by clicking on **File -Print**. This will only print what is currently displayed in the **Command Window**. The **Command Window** has a limited buffer size; if you fill the buffer, old information will be lost and cannot be recovered.

1.1 Statements

In the **Command Window,** all statements are typed at the Command Prompt.

Professional Version	Student Edition
>>	**EDU>>**

Statements preceded by **%** are comments (non-executable statements). Identify the specific laboratory experiment and section of the experiment in your comments:

```
>> % Name
>> % Lab 1 Section 2
>> % **********
```

Results are stored as the variable **ans** in the workspace and displayed in the **Command Window**.

```
>>       3.45                                    (Enter)
ans =
       3.4500
>>                                               (Enter)
```

After the statement is complete, pressing the **Enter** key causes the command to be executed.

```
>>      sqrt(1.44)                               (Enter)

ans =

       1.2000

>>                                               (Enter)
```

If a statement is too long for a single line, it can be extended by typing a space and three periods followed by pressing the **Enter** key.

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

Multiple statements, separated by commas, can be typed at one prompt. The statements are executed from left to right.

```
>> a=16, b=sqrt(a)
a =
       16
b =
       4
```

If statements are terminated with a semicolon (**;**), then they are executed but the result is not printed to the **Command Window**. The values entered or computed can be subsequently displayed by entering the variable values at a prompt.

```
>>      c=25;   d=sqrt(b)+2.5;
>>
>>      ans, a, b, c, d
ans =

       21.3500
```

a =
16
b =
4
c =
25
d =
4.5000

The statement **sqrt(1.44)** uses the MATLAB *function* **sqrt** to compute the square root of **1.44**. MATLAB has many built-in functions. Use the **lookfor** or **help** commands if you wish to search for a MATLAB function or to get help using the function. Help can also be accessed in **Help - MATLAB Help**.

>> **lookfor sinc**
(will display all functions with "sinc" in the name or function description)

>> **help sinc**
(will display the help file, or comments, giving instructions on how to use the function.)

Typing only **help** at the command prompt will list all the MATLAB directories that contain functions.

Previously entered statements in the **Command Window** are stored in a buffer. They can be recovered by using the **Up Arrow** key or by using the **Command Window History**. This buffer is not cleared by either **clear all** or **Edit – Clear Session** but is cleared with **Edit -Clear Command History**.

Data can be stored as variables. Variable names are case sensitive. That is, **mb** and **Mb** are two different variables. Variable names cannot start with a number and cannot contain punctuation or special characters.

Note that all entered and computed values remain in the **Workspace** and can be used or printed to the **Command Window**. Any variable can be overwritten later. Notice that **ans** is now**21.3500**, when before it had been **1.2000**

To determine which variables are stored in the **Workspace** and their size, we can use the command **whos**.

whos

Name	**Size**	**Bytes**	**Class**
a	**1x1**	**8**	**double array**
ans	**1x1**	**8**	**double array**
b	**1x1**	**8**	**double array**
c	**1x1**	**8**	**double array**
d	**1x1**	**8**	**double array**

Grand total is 5 elements using 40 bytes

1.2 M-Files

Files with the filename extension **.m** are executable files. We call such files *m-files* and they come in two different types: *script* and *function*. Function m-files will be discussed later. The functions that we have already defined and used are contained in function m-files in the MATLAB toolboxes.

1.2.1 Script M-Files.

Open the **Editor/Debugger** using **File-New-M-file**. This can also be accomplished using the **new file** icon on the toolbar. Begin with the commands **clear all**, **close all** and **clc**. You should begin all script M-files with these commands to clear out all previously used variables, close open figure windows and refresh the command window. Use comments containing your name, experiment number and what section of the lab you are working on. It is always good programming practice to comment your scripts so that others can more easily read your code. A template is provided that shows how to structure your comments (ask your instructor for its location). Save your m-file with **File - Save** using an appropriate file name (e.g., exp1.m). **Do not** begin an M-file name with numbers or punctuation. **Do not** use punctuation in a file name.

NOTE*:* MATLAB function names are reserved file names. If you name an M-file with the same file name as a function, MATLAB will execute your file instead of the function when it is called. **Do not** name an M-file with a reserved file name!

To execute a script M-file, type the script name at the command prompt and press enter.

>> **exp1** **(Enter)**

If your script contains errors, MATLAB will display an error message below the prompt, along with the line number that generated the error. If your script does not contain errors, you will see another prompt when the script has executed. You can also save and run an M-file from the Editor/Debugger window using the F5 function key.

1.2.2 Storage and Retrieval Commands

save *filename variables* - Saves the *variables* from the **Workspace** in the file *filename*.mat in the MATLAB/work directory. If *variables* is left off, then all variables in the **Workspace** are saved. **load *filename***

- Loads file *filename*.mat from the MATLAB/work directory.

1.3 Numeric Format

So far we have only entered and used real numbers. We can also enter and use other types of numbers.

1.3.1 Complex Numbers

A complex number consists of a real part and an imaginary part. We can use **i** or **j** as an indicator for the imaginary part. A complex number printed to the **Command Window** always uses the indicator **i**. MATLAB will recognize **i** or **j** as imaginary indicators. The real part, imaginary part, amplitude, and angle in radians of a complex number are given by the functions **real**, **imag**, **abs**, and **angle**, respectively. **(editor)** indicates lines written in the editor. >> indicates the beginning of the line(s) of output that would result (when run) from the line(s) just shown entered in the editor.

```
(editor)   a=3 - 4j,   b=real(a), c=imag(a), d=abs(a), e=angle(a)
>> a =
       3.0000 -4.0000i
b =
       3
c =
     - 4
d =
       5
e =
     - 0.9273
```

We can also enter a complex number as **a=3 -j*4**, where ***** indicates multiplication.

If either **i** or **j** has been redefined in an earlier statement, we must again define it to be **j = sqrt (- 1)** before using it to generate a complex number.

```
(editor) f=4; g=9; h=sqrt(f)+j*sqrt(g)
>>h =
      2.0000+3.0000i
```

1.3.2 Matrices and Vectors

All numeric values are stored in matrices. The single values considered in the above sections are stored in (1x1) matrices. Single row or single column matrices are called vectors. We enter a (*n* x *m*) matrix (*n* rows, *m* columns) by entering the individual matrix term values row by row within square brackets. We can enter more than one row on a single statement line if we use semicolons between rows.

```
(editor)        a=[3   4
                     2       1]
>>a =
        3       4
        2       1
```

(editor) b=[1.5 - 2.4 3.5 0.7; - 6.2 3.1 -5.5 4.1; 1.1 2.2 - 0.1 0]

>> b =

1.5000	**- 2.4000**	**3.500**	**0.7000**
- 6.2000	**3.1000**	**- 5.5000**	**4.1000**
1.1000	**2.2000**	**- 0.1000**	**0**

Values stored in a matrix can be referenced by an index. All MATLAB indices begin with 1, and cannot be 0 or negative. You can specify values in a matrix using a single index, or by specifying its row and column indices.

(editor) b(1)

>> ans =

1.5000

(editor) e=b(2, 3), f=b([2 3], [1 3]), g=b(2, [3 4])

>> e =

- 5.5000

f =

- 6.2000 - 5.5000
1.1000 - 0.1000

g =

- 5.5000 4.1000

We can also create a matrix by concatenating (joining) several vectors. The vectors used must have the correct number of rows and columns to make the resulting matrix proper.

(editor) h=[1 2 3], k=[4; 7], m=[5 6; 8 9]

>> h =

1 2 3

k =

4
7

m =

5 6
8 9

(editor) n=[h; k m]

>> n =

1 2 3
4 5 6
7 8 9

1.3.3 Vectors

Single-row or single-column matrices are called vectors. Therefore, they are entered like matrices.

```
(editor) a=[3 5 9], b=[3; 5; 9]
>> a =
        3    5    9
b =
        3
        5
        9
```

Colon (**:**) notation can be used to enter a set of equally spaced, sequential numbers.
Variable = start : step : end;

```
(editor) c=2:5, d=3:2:9
>> c =
     2    3    4    5
d =
     3    5    7    9
```

This example evaluates the function y = sqrt (x) at samples from 0.5 to 2.0 in steps of 0.25.

```
(editor) x=0.5:0.25:2.0;
(editor) y=sqrt(x);
(editor) x, y
>> x =
        0.5000    0.7500    1.0000    1.2500    1.5000    1.7500    2.0000
y =
        0.7071    0.8660    1.0000    1.1180    1.2247    1.3229    1.4142
```

Vectors also use indices to reference values. Multiple indices will return multiple values.

```
(editor) f=[10   5   4   7   9   0]
(editor) g=[2 5 6]; h=f(g)
        >> f =
        10    5    4    7    9    0
h =
         5    9    0
```

Here are more examples of indices, combined with colon notation. Note that a colon by itself indicates all rows or all columns. How does each output relate to the original matrix?

```
(editor) m=[1.5 - 2.4   3.5 0.7; - 6.2   3.1 - 5.5   4.1;
(editor)        1.1 2.2 - 0.1   0]
>> m =
        1.5000   - 2.4000        3.5000          0.7000
        - 6.2000         3.1000   - 5.5000        4.1000
        1.1000           2.2000   - 0.1000        0

(editor) n=m(1:2,2:4), o=m(:, 1:2), p=m(2, :)
>> n =

      - 2.4000    3.5000    0.7000
        3.1000         -    4.1000
                 5.5000
o =
        1.5000         -
                 2.4000
      - 6.2000    3.1000
        1.1000    2.2000
p =
      - 6.2000    3.1000    - 5.5000   4.1000
```

1.3.4 Array

Matrices or vectors can also be interpreted as two-dimensional or one-dimensional arrays, respectively. This is the interpretation that we use in most of our MATLAB applications in EE 216.

1.3.5 Special Number Notation

Two special numbers are provided in MATLAB. They are *p* and *infinity* and are given the notation **pi** and **Inf**, respectively. In addition, operations such as *0/0* or *sin(infinity)* produce undefined results that is given the notation **NaN**, which stands for *Not a Number*.

1.4 Character Strings

In addition to numbers, MATLAB can also store and use text. Text is stored in *character strings*. We designate a string (including single characters) with single quotes (‘ ’).

```
(editor) 'Signal and System Analysis'
>> ans =
Signal and System Analysis
```

The character strings are stored in arrays with one character corresponding to one array element. Therefore, we can select a portion of a character string to use or print.

(editor) M='MATLAB Character String'
>>M =
MATLAB Character String

(editor) C=M(8:16)
>> C =
Character

1.5 Arithmetic Operations

MATLAB defines all arithmetic operations in matrix terms. Use the command **help arith** and **help slash** to list the arithmetic operators. **Note: c\b** specifies matrix division (b multiplied by the inverse of c on the left). Matrix sizes must be appropriate (conformable) for all arithmetic matrix operations. Examples are provided below.

```
(editor) a=[1 2; 3 4]; b=[3 1; 7 8]; c=[2 4];
(editor) 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
(editor)   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 in a matrix math sense, we use the modifier '**.**' It is implicit that scalar multiplication is performed on each element.

```
(editor)   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
```

All rules of matrix operations apply when using arithmetic operators.

1.6 Function M-Files

Function m-files are like script m-files except that variable values may be passed into or out of function m-files. Also, variables defined and manipulated only inside the file do not appear in the **Workspace**. The first line of a function m-file starts with the word function and defines the function name and input and output variables. For example:

```
function [z,w] = abcd(x,y)
```

is the first line of the function m-file **abcd.m**. The input variables of this function are **x** and **y** and the output variables are **z** and **w**. Each input and output is an array or matrix. If the array is (1x1), then the variable has a single real or complex value. A function m-file name must match the function name for it to execute. Most will contain the function definition, help file and function statements. Here is an example of a function m-mile for **example.m**,

```
function [x,y,z] = example(w)
%EXAMPLE        An example of a function.   This function
%               computes the square root, the square and
%               the mean of the values in a matrix w.
x = sqrt(w); y = w.^2; z = mean(w);
```

An example of calling this function is shown below.

```
>> w=[1 2 3 4 5]
>> [x,y,z]=example(w);
>> x,y,z
x =
```

```
1.0000    1.4142    1.7321    2.0000    2.2361
 y =
1     4     9    16    25
 z =
   3
```

Many functions, such as **z = mean(x)**, are built-in MATLAB functions. However, others are contained in function m-files in MATLAB toolboxes. A function m-file (or a script m-file) can be executed as long as it is in the MATLAB path. (See the MATLAB help files or **File-Set Path**. Any directory or folder listed is in the MATLAB path.)

1.7 Logical Operations

The logical operations AND, OR , and NOT, are specified by **&**, |, and **~** , (ampersand, pipe, and tilde) respectively. These can be used in conjunction with the relational operations (<, <=, >, >=, ==, ~=) to construct arrays of zeros and ones (0 - 1 arrays). The ones correspond to elements for which the logic operation is true.

```
(editor) 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
(editor) c=[1    5    3    4    7    8],  d=c>4
>> c =
        [1    5    3    4    7    8],
d =
         0    1    0    0    1    1
```

1.8 Mathematical Functions

MATLAB contains a set of built-in mathematical functions. All of these functions are applied to arrays on an element-by-element basis. A partial list is given below.

sqrt	-	square root
real	-	complex number real part
imag	-	complex number imaginary part
abs	-	complex number magnitude or absolute value of a real number
angle	-	complex number angle
exp	-	exponential base e
log	-	logarithm base e
Log10	-	logarithm base 10

The trigonometric functions all apply to angles expressed in radians.

sin	-	sine
cos	-	cosine
tan	-	tangent
asin	-	arcsine
acos	-	arccosine
atan	-	arctangent
atan2	-	four quadrant arctangent
round	-	round to nearest integer
floor	-	round toward $-\infty$
ceil	-	round toward ∞

1.8.1 Mathematical Expressions

We can combine arithmetic operations, 0 -1 arrays generated by logical operations, and mathematical functions into mathematical expressions. Often, these expressions take the form of equations, although they may also be used in flow control statements. The arithmetic operations follow the usual precedence rules. Many mathematical expressions require parentheses to construct the desired sequence of operations within the precedence rules.

```
(editor) t=0.1;
(editor) x=2^t*sqrt(t) -sin(2*t)/3
>> x =
        0.2727

(editor) y=2^(t*sqrt(t)) -   sin(2*t)/3
>> y =
        0.9559
```

We can evaluate a mathematical expression for a set of independent variable values by expressing the independent variable as a one-dimensional array (vector) and using array operations.

```
(editor) f=0:2:4; w=2*pi*f;
(editor) X=(3   -j*0.1*w)./(1.5+j*0.2*w)
>> X =
        2.0000    0.1566 - 1.1002i      - 0.2956   -   0. 6850i
```

One important use of a 0 -1 array for signal and system analysis is in the representation of a piecewise-defined signal with a mathematical expression. Given,

$$x = \begin{cases} t+1, & 0 \le t < 1 \\ 2, & 1 \le t \le 2 \end{cases}$$

the script would be,

```
(editor) t= 0:0.5:2;
(editor) x=(t+1).*(t>=0&t<1)+2*(t>=1&t<=2)
>> x =
    0 1.0000    1.5000    2.0000    2.0000    2.0000    0
```

1.9 Flow Control

MATLAB has flow control statements that we can use to repetitively or selectively execute other statements. The flow control statement affects all statements between itself and an associated **end** statement.

1.9.1 For Statement

The **for** statement permits us to execute the same set of statements repetitively for a designated number of times. It is equivalent to FOR or DO statements found in other computer languages. For example, to evaluate the summation $x(t) = \sum_{k=1}^{3} t^{\sqrt{1.2k}} \sqrt{k}$ for $0 \le t \le 0.8s$ at dt=0.2s intervals and print the results, we can use the statements

```
(editor) t=0:0.2:0.8; x=zeros(size(t));
(editor) for k=1:3;
(editor) x=x+sqrt(k)*t.^sqrt(1.2*k);
(editor) end;
>> x
x =
          0    0.3701    1.0130    1.8695 2.9182
```

For statements can be nested

```
(editor)    for m=1:3;
(editor)     for n=1:4;
(editor)        y(m,n)=m+n;
(editor)        end;
(editor)        end;
>>    y
y =
      2   3   4   5
      3   4   5   6
      4   5   6   7
```

When using a **for** loop to evaluate large arrays, it is best to define the output variable before the loop executes. This allows the loop to execute faster, since MATLAB does not have to keep re-sizing the array.

```
(editor) t=0:0.5:10; y=zeros(size(t));
(editor) for q = 1:length(t);
(editor)         y = t+q;
(editor) end;
```

1.9.2 While Statement

```
(editor)   if n==2;
(editor)   y=10*d(n);
(editor)   else;
(editor)   y=0;
(editor)   end;
```

The **while** statement is like the **for** statement except that execution stops when a logic expression is satisfied. The statements

```
(editor) n=1;
(editor) while 2*n<5000; n=2*n; end;
>> n
n =
      4096
```

compute the largest power of 2 that is less than 5000.

1.9.3 If Statement

The **if** statement permits us to execute statements selectively depending on the outcome of a logic expression.

```
(editor)   for k=1:4;
(editor)       if k==1; x(k)=3*k;
(editor)           else if k==2|k==4; x(k)=k/2;
(editor)               else; x(k)=2*k;
(editor)               end;
(editor)           end;
(editor)       end;
>> x
x =
     3   1   6   2
```

If statements can be nested.

```
(editor)   c='t'; n=2;
(editor)   if c=='f'; c='false'; y=Nan; end;
(editor)   d=0.1:0.1:0.4;
(editor)   if c=='t';
```

```
(editor)     if n==2;
(editor)        y=10*d(n);
(editor)     else
(editor)        y=0
(editor)  end;
>>   c,  y
c =
        t
y =
        2
```

1.10 Other Functions and Commands

The functions and commands described here and in other experiments do not encompass all functions and commands available. To find out about other functions use the **lookfor** and **help** commands.

help - lists the directories in the MATLAB path which contain functions
also lists the toolboxes
help *directory* – lists functions in that directory
help *function* – displays help file for that function
lookfor *word* – searches for functions with *word* in the name or description

1.10.1 Numeric Functions

find(A) Returns a one-dimensional row-array containing the indices of non-zero elements of a one-dimensional array. It can be used with 0 - 1 arrays to find indices of elements that have other values.

```
(editor) a=[1   0   2   3   0   4];b=find(a)
>>b =
        1   3   4   6
(editor) n=find(a>2)
>>n =
        4 6
```

size(A,i) Returns the number of rows in **A** if **i=1** or the number of columns in **A** if **i=2**. If **i** is not included, then a row vector containing both the number of rows and the number of columns is returned.

zeros(m,n) Returns an (*m* x *n*) array of zeros. A modification is**zeros(size(A))** which returns an array of zeros having size equal to the size of **A**.

max(A) Returns the value of the largest element in a one-dimensional array. **max(max(A))** returns the value of the largest element in a two-dimensional array.

min(A) Like **max(A)** except that it returns smallest values.

mean(A) Returns the mean, or average value, of all elements in a one-dimensional array and a one-dimensional row-array containing the mean values of the elements in the columns of a two-dimensional array.

meshgrid(A,1:n) Returns an array having n rows where each row is the one-dimensional array **A**.

```
(editor)   d= - 0.1:0.1:0.2;   dm=meshgrid(d,1:3
 >> dm =
        -0.1   0   0.1   0.2
        -0.1   0   0.1   0.2
        -0.1   0   0.1   0.2
```

sum(A) Returns the sum of the elements of **A** if **A** is a one-dimensional array. Returns a one-dimensional row-array that contains the sums of the columns of **A** if **A** is a two-dimensional array.

1.11 Plotting Functions and Commands

plot(x,y) Plots the variable **y** versus the variable **x** in the current **Figure Window**by connecting data values with straight lines. Two or more functions of the same independent variable can be plotted on the same set of axes. To do so, remain in the same **Figure Window**and use the statements **hold on** and **hold off** after the first and last **plot** statements, respectively. By itself, **plot** provides only a default scaling and size.

hold on Use after a plot command to draw multiple lines on one axis.

hold off Releases the **hold on** command.

xlabel('text') Labels the x-axis of a plot with the text specified by '**text**'.

ylabel('text') Labels the y-axis of a plot with the text specified by '**text**'.

title('text') Places the title specified by '**text**' above a plot.

text(x,y,'text') Adds the text specified by '**text**' to a plot at the location **(x, y)**, where **x** and **y** are the horizontal and vertical axis coordinates, respectively

Legend Creates a legend for multiple plots on one axis

Figure Opens a new **Figure Window**

subplot(m,n,ax) Splits the Figure Window into a matrix of MxN axes. Example **subplot(2,1,2)** specifies 2 rows of axes, in 1 column, and you are using the 2nd axis.

axis([ranges]) Sets scaling for the x-, y- and z-axes on the current plot according to to the vector ranges = [xmin xmax ymin ymax zmin zmax]. If only a 2-D is being used, ranges = [xmin xmax ymin ymax].

In some experiments, you will open a number of **Figure Windows** and generate a plot in each. To eliminate one of these plots, click on **File - Close** when you are in the corresponding **Figure Window**. To eliminate all plots, type the command **close all** in the **Command Window**.

2.0 Specific Application Information

In Section 2, several function m-files that have been created specifically for signal and system analysis are presented. On the laboratory computers, these m-files are located in **w:\matlab\toolbox\ee216**

2.1 Signal and System Analysis Functions

Two additional function m-files have been created for use in signal and system analyses in EE216. These functions encompass:

1. step and ramp signal functions
2. discrete- and continuous-time Fourier series functions
3. 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.

2.1.1 Step and Ramp Function

We use step and ramp functions often in signal and system analysis. Therefore, we have created the MATLAB functions **u = us(t)** and **r = ur(t)** to compute them. The values computed for **u** are **0** for **t<0** and **1** for **t>=0**. Likewise, the values computed for **r** are **0** for **t<0** and **t** for **t >=0**.

2.1.2 Continuous-Time Fourier Series Function

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.

2.1.2.1 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 X0 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 function m-file, **ctfsc.m**.

2.1.2.2 Truncated Fourier Series

The second Fourier series related function is

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

This function first selects the 2N + 1 Fourier series coefficients, **Xnn**, centered on X0, where N is specified with the input variable **N**. 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 time interval as the expansion interval used to compute the Fourier series coefficients.

The input variables **Xn**, **no**, and **fo** are the output variables obtained from 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.

2.1.3 Fourier Transform Functions

2.1.3.1 Fourier Transform

MATLAB contains a built-in function for the Fourier Transform called **fft(x)**. 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=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)<=0** and end at **t(ns)>0.** If the signal is longer than the time interval **ns*dt**, thenthe computed transform will have some distortion since it is the transform of the truncated signal.

There are **N** Fourier transform samples computed, where **N** equals the larger of **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.

```
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 being included in the samples. Try extending the length of the samples to include a longer portion of the signal.

2.1.3.2 Inverse Fourier Transform

MATLAB contains a built-in function for the inverse Fourier Transform called **ifft(x)**. 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=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.

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

2.1.4 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 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

2.1.4.1 Transfer Function Parameter Computation

The function

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

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

$$H(s)=[n(1)s^{a}+...+n(a)s+n(a+1)] / [d(1)s^{ß}+...+d(ß)s+d(ß+1)]$$

or frequency response

$$H_{\omega}(\omega)=[n(1)(j\omega)^{a}+...+n(a)(j\omega)+n(a+1)] / [d(1)(j\omega)^{ß}+...+d(ß)(j\omega)+d(ß+1)]$$

The numerator and the denominator coefficients are stored in the input vectors **n** and **d**, respectively. Function outputs include:

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 number of poles on the imaginary axis,

rhps = two-element row-vector containing the number of zeros and number of poles in the right-half plane,

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 = a row-vector that contains the break frequency for each factor where, for linear factors, **bf>0**, **bf=0**, and **bf<0** indicate LHP, Imaginary Axis, and RHP plane poles or zeros, respectively,

ft = a row-vector of factor types where +1 and +2 correspond to numerator linear and quadratic factors, respectively, and -1 and -2 correspond to denominator linear and quadratic factors, respectively,

dr = a row-vector of factor damping ratios.

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 or j? factor. 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 Nth order factor is the sum of the straight-line approximations for the N first-order factors.

2.1.4.2 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 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 Section 2.1.4.1.

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