Signal and Linear System Analysis - 2nd Edition
Gordon E. Carlson

MATLAB® Tutorial

Contents

Signal and Linear System Analysis
Gordon E. Carlson

MATLAB Tutorial

This tutorial provides basic MATLAB information and specific application information for the text "Signal and Linear System Analysis - 2nd Edition" by Gordon E. Carlson. The MATLAB User's and Reference Guides should be used to obtain greater breadth and depth of information.

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

1.0 Basic MATLAB Information

On initiation of MATLAB, the **Command Window** appears on the monitor screen. The values of all variables defined after initiation are stored in the **Workspace.**

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

1.1 Statements

Statements (commands, data entries, functions, mathmatical expressions, logical expressions,flow control) are typed at the prompt

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

If the statement is too long for a single line, it can be extended by typing three periods followed by pressing the **Enter** key. After the statement is complete, pressing the **Enter** key causes command, data entry, function, or expression execution. Results are stored as **ans** in the workspace and displayed in the **Command Window**. To minimize vertical space required in the window, click on **Options - -Numeric Format - - Compact**.

```
>>  3.45                    (Enter)
ans =
     3.4500
>>                          (Enter)
>>  sqrt(1.44)              (Enter)
```

```
ans =
      1.2000
>>                                  (Enter)
>>   2+6.35+sqrt(36) ...            (Enter)
     +sqrt(49)                      (Enter)
ans =
     21.3500
```

The statement **sqrt(1.44)** uses the MATLAB *function* **sqrt** to compute the square-root of **1.44**.

Previously entered statements in the **Command Window** are stored in a buffer. They can be recovered for reuse or for modification and reuse by using the **Up Arrow** key. The buffer is not cleared by either **clear all** or **Edit - - Clear Session**. It is only cleared by exiting and reentering MATLAB.

Data can be given a name, that is, stored as a variable. 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
```

Variable names are case sensitive. That is **mb** and **Mb** are two different variables. Other variable name rules are indicated in the MATLAB User's Guide.

If statements are each 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
```

Note that all entered and computed values remain in the **Workspace** and can be used or printed to the **Command Window**. The last value stored in **ans**, **21.3500**, has overwritten the first and second values stored, **3.4500**, and **1.2000**.

To determine which variables are stored in the **Workspace** and their size, we can use the command **whos**. If we do this now at the conclusion of Section 1.1, we see the following in the **Command Window.**

```
>>  whos
```

Name	Size	Elements	Bytes	Density	Complex
a	1 by 1	1	8	Full	No
ans	1 by 1	1	8	Full	No
b	1 by 1	1	8	Full	No
c	1 by 1	1	8	Full	No
d	1 by 1	1	8	Full	No

```
Grand total is 5 elements using 40 bytes
```

Throughout this tutorial, we assume that the **Workspace** is cleared only at the end of every section, except when we indicate otherwise.

1.2 Numeric Format

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

1.2.1 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**. 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.

```
>>  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 **j** has not been redefined in an earlier statement. This form of entry is required if the imaginary part is to be generated by a function. If **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.

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

`

1.2.2 Matrices

All numeric values are stored in matrices. The single values considered in the above sections are stored in (1x1) matrices. 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.

```
>>  a=[3  4                                          (Enter)
       2  1]                                         (Enter)
a =
    3   4
    2   1
>>                                                   (Enter)
>>  b=[1.5  -2.4  3.5  0.7;  -6.2  3.1  -5.5  4.1;   (Enter)
       1.1  2.2  -0.1  0]                            (Enter)
b =
    1.5000   -2.4000    3.5000   0.7000
   -6.2000    3.1000   -5.5000   4.1000
    1.1000    2.2000   -0.1000        0
```

Individual matrix entries, or a submatrix made up of original matrix entries, can be obtained by specifying row and column indices of the desired entries in two one-row matrices.

```
>>  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 construct a matrix from submatrices.

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

All submatrices in each specified row of submatrices must have the same number of rows. Also, the sum of the number of submatrix columns in all specified rows of submatrices must be equal.

1.2.3 Vectors

A row vector is an one-row matrix and a column vector is a one-column matrix. Therefore, they are entered like matrices.

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

We can also enter a row vector with equally spaced values by using the colon notation

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

Note that the middle number in the specification for the **d** vector specifies the interval between vector values. A row vector entered with the colon notation is particularly useful when we want to evaluate a mathematical function over an interval of independent variable values. For example, consider the evaluation of the function $y(x) = (x)^{1/2}$ over the interval $0.5 \pounds x \pounds 2.0$ at values of x that are multiples of 0.25.

```
>>  x=0.5:0.25:2.0;
>>  y=sqrt(x);
>>  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
```

We can use a vector to create a vector that contains only a subset of an original vector's values.

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

One use for this technique is to select every *n*th value of an independent variable and a function of this variable. For example, to select every third value of the independent variable **x** and the function **y = sqrt(x)**, we use the commands.

```
>>  k=1:3:7;  x1=x(k),  y1=y(k)
x1 =
    0.5000   1.2500   2.0000
y1 =
    0.7071   1.1180   1.4142
```

We can obtain a submatrix of a matrix by using colon notation for two row vectors that specify the row and column indices of the desired entries.

```
>>  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 =
    1.5000   -2.4000    3.5000   0.7000
   -6.2000    3.1000   -5.5000   4.1000
    1.1000    2.2000   -0.1000        0
>>
>>  n=m(1:2,2:4), o=m(:, 1:2), p=m(2, :)
n =
   -2.4000    3.5000   0.7000
    3.1000   -5.5000   4.1000
o =
    1.5000   -2.4000
   -6.2000    3.1000
    1.1000    2.2000
p =
   -6.2000   3.1000   -5.5000    4.1000
```

Note that a colon by itself indicates all rows or all columns.

1.2.4 Arrays

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 "Signal and Linear System Analysis - 2nd Edition".

1.2.5 Special Numbers

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 an undefined results that is given the notation **NaN**, which stands for *Not a Number*.

1.2.6 Polynomial Coefficients and Roots

An *n*th degree polynomial in the variable *x* is $a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_o$. We can define its coefficients by a one-dimensional row array (vector) of length n. To find the roots of the polynomial for this coefficient array, we can use the function **roots**. The results of this function is a one-dimensional column array containing the n roots. If we know the roots of a polynomial, we can place them in a column array and use the function **poly** to find the row array of polynomial coefficients (except for the multiplicative constant, a_n. Finally, if we want to multiply two polynomials, we use the row arrays of their coefficients **a** and **b** as inputs to the function **conv(a,b)**. The output of this function is the row array of the product-polynomial coefficients.

```
>>  a=[2  5  6  3];  b=[3  9  3];  r=roots(a)
r =
     -0.7500 + 0.9682i
     -0.7500 -  0.9382i
     -1.000
>>
>>  c=poly(r)
c =
     1.0000   2.5000   3.0000   1.5000
>>
>>  d=conv(a,b)
d =
     6   33   67   73   39   6
```

1.3 Character Strings

In addition to numbers, MATLAB can also store and use text. We require text, for example, when we want to label output data or plots. Text is stored in *character strings*, which we normally call just *strings*. We designate them by a single quote, or apostrophe, **'** at the beginning and at the end of the character string.

```
>>  'Signal and System Analysis'
ans =
Signal and System Analysis
>>
>>  title='            Sample Index',  n=1:6
title =
            Sample Index
n =
     1     2     3     4     5     6
```

Another use for a character string is in the flow control statements to be discussed in Section 1.7. In these statements, we can use **'t'** and **'f'** as true and false indicators to perform statement execution control.

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.

```
>>  M='MATLAB Character String'
M =
MATLAB Character String
>>
>>  C=M(8:16)
C =
Character
```

1.4 Arithmetic and Logical Operations

There are a number of basic operations that we use to generate mathematical expressions. Some are arithmetic operations and some are logical operations.

1.4.1 Arithmetic Operations

The arithmetic operations of matrix addition, subtraction, multiplication, raise to a power, and transpose are specified by **+, -, *, ^,** and **'**, respectively. In addtion, **b/c** and **c\b** specify the multiplication of matrix **b** by the inverse of square matrix **c** on the right and the

inverse of square matrix **c** on the left, respectively. Note that the matrix sizes must be equal for addition and subtraction, conformable for multiplication, **/**, and **** operations, and square for the raise to a power operation.

```
>>  a=[1  2;  3   4],  b=[3  1;  7  8],  c=[2  4]
a =
     1   2
     3   4
b =
     3   2
     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
```

We define arithmetic operations on an array to be operations on an element by element basis. Since matrix addition and subtraction are element by element operations, then array addition and subtraction equal matrix addition and subtraction. Therefore, these operations are specified by **+** and **-**. Array multiplication, division, transposition, and raise to a power are specified by **.***, **./**, **.'**, and **.^**, respectively. Note that two arrays used in an arithmetic array operation must be the same size.

```
>>  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
```

An exception to the array or matrix size requirements occurs when we add, subtract, multiply or divide by a constant. In these cases the constant is added to, subtracted from, multiplied times, or divided into each array or matrix element.

```
>>  p=c+2,  q=c - 2,  r=2.*c,  s=c./2,  t=2*c,  u=c/2
p =
    4   6
q =
    0   2
r =
    4   8
s =
    1   2
t =
    4   8
u =
    1   2
```

Note that a matrix multiplication or division operation is an array operation when multiplication or division by a constant is performed.

1.4.2 Logical Operations

The logical operations "and", or, and "not" are specified by **&**, **|**, and **-**, respectively. These can be used in conjunction with the relational operations

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

to construct arrays of "zeros" and "ones" (0-1 arrays). The ones correspond to elements for which the logic operation is satisfied.

```
>>  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
```

1.5 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. Thus, they return an array having the same size as the input array with elements all modified in the same way. We have already defined and used five of the functions. These are

sqrt - square root
real - complex number real part
imag - complex number imaginary part
abs - complex number magnitude
angle - complex number angle

If a number is real, then **abs** produces the absolute value of the number. Other available mathematical functions that are of interest to us in signal and system analysis include

exp - exponential base e
log - logarithm base e
log 10 - logarithm base 10
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 ∞

The trigonometric functions all apply to angles expressed in radians.

1.6 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.

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

```
>>   f=0:2:4;  w=2*pi*f;
>>   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.

```
>>   t=-0.5:0.5:2.5;
>>   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
```

The notation **.*** is required for the first mulitplication since we want element by element multiplication of the two like-sized, one-dimensional arrays. We can use just ***** for the second multiplication since it is a constant times an array.

1.7 Flow Control

MATLAB has flow control statements that we can use to repetitively or selectively execute other statements. Each of these must be associated with an **end** statement.

1.7.1 For Statement

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

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

For statements can be nested

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

The size of array **y** was kept small, 3 x 4, so it could be printed in a reasonable amount of space. If we increase the size of array **y** to 200 x 200, then it requires approximately 8s to execute the statements. Execution time can be reduced to approximately 4.3s if array **y** is formed before computing values for it. This can be done by using the statement **y=zeros(200,200)** prior to the first **for** statement. Execution time is measured with function **tic** inserted before and function **toc** inserted after the statements for which execution time is desired.

1.7.2 While Statement

The **while** statement is like the **for** statement except that it executes a set of statements repetitively for a non-designated number of times. Execution stops when a logic expression is satisfied. The statements

```
>>  n=1;
```

```
>>  while 2*n<5000;   n=2*n;   end;
>>  n
n =
     4096
```

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

1.7.3 If Statement

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

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

If statements can be nested. Also, we can use the string variables **'t'** and **'f'** in an **==** or **~=** logic expression to execute statements controlled by an **if** statement.

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

If c = 't' and n ≠ 2, the result of the above statements is

```
c =
t
y =
     0
```

On the other hand, if c = 'f', then the results of the above statements is

```
c =
false
y =
     NaN
```

regardless of the value of n.

1.8 Other Functions and Commands

There are a large number of MATLAB functions and commands available in MATLAB and its toolboxes. We have already mentioned the functions **roots**, **poly**, and **conv** in Section 1.2.6. We mention several additional general functions here that are useful for our signal and system analyses. Other, more specific functions are briefly described in the text where they are first used. Some of the functions have variations that are not described here. These are invoked by changes in input and output specifications. See the Reference and User's Guides for more information.

The functions and commands described here and in the text do not encompass all functions and commands used. The MATLAB Reference Guide and the Toolbox User's Guides contain information on those not described here or in the text.

1.8.1 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.

```
>>  a=[1  0  2  3  0  4];   b=find(a)
b =
    1   3   4   6
>>  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. Returns 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**.

```
>>  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.

```
>>  e=[1  2  3],  es=sum(e),  f=[4;  5;  6],  fs=sum(f)
e =
     1   2   3
es =
     6
f =
     4
     5
     6
fs =
     15
>>
>>  g=[1  2  3;  4  5  6],  gs=sum(g)
g =
     1   2   3
     4   5   6
gs =
     5   7   9
```

The **sum** statement can be used with the **meshgrid** statement to evaluate a summation for a range of independent variable values. To illustrate this usage, we consider the evaluation of the summation $x(t) = \Sigma_{k=1}^{3} \sqrt{k}\, t^{\sqrt{1.2k}}$ for $0 \le t \le 0.8$s at $dt = 0.2$s intervals.

```
>>  t=0:0.2:0.8;  k=1:3;  c=sqrt(k);  n=sqrt(1.2*k);
>>  tm=meshgrid(t,1:size(k,2));
>>  cm=meshgrid(c,1:size(t,2)).';
>>  nm=meshgrid(n,1:size(t,2)).';
>>  x=sum(cm.*tm.^nm);
>>  tm,  cm, nm,  x
tm =
         0   0.2000   0.4000   0.6000   0.8000
         0   0.2000   0.4000   0.6000   0.8000
         0   0.2000   0.4000   0.6000   0.8000

cm =
    1.0000   1.0000   1.0000   1.0000   1.0000
    1.4142   1.4142   1.4142   1.4142   1.4142
    1.7321   1.7321   1.7321   1.7321   1.7321

nm =
    1.0954   1.0954   1.0954   1.0954   1.0954
    1.5492   1.5492   1.5492   1.5492   1.5492
    1.8974   1.8974   1.8974   1.7984   1.7984
x =
         0   0.3701   1.0130   1.8695   2.9182
```

The **meshgrid** statements permit us to set up equal size arrays with the number of columns equal to the number of values of **t** (note the transpose required to produce **cm** and **nm**). The sum of the arithmetic operations on these arrays for each column is the desired summation values at the corresponding value of **t**. Note that we have evaluated the same summation as we did earlier in Section 1.7.1 using the **for** statement. More statements were required but less execution time is needed since the full array capability of MATLAB is being used. The difference in executions time is insignificant when only three values are computed for the summation. We used only three so that we could print the meshgrid arrays in reasonable space. If we compute 10000 values instead of 3, the execution time using **meshgrid** and **sum** is approximately 1.2s; whereas, the execution time using **for** is approximately 7s.

1.8.2 Data Storage and Retrieval Commands

diary ***filename*** - Saves the contents of the **Command Window**, up to the command **diary off**, in the file *filename* in the MATLAB directory.

save ***filename variables*** - Saves the *variables* from the **Workspace** in the file *filename*.mat in the MATLAB directory.

load *filename* - Loads the file *filename*.mat from the MATLAB directory.

1.8.3 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. By itself, it provides only a default scaling and size. The plotting functions **plct** and **pldt** provide more flexibility in plot characteristics. They are discussed in Section 2.1.2 of this tutorial.

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**'.

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.

figure - Opens a new **Figure Window**.

print *filename -deps 2* - Prints the figure in the current **Figure Window** to file *filename*.eps in Encapsulated Post Script 2 format (See the MATLAB Reference Guide for other possible formats).

1.9 M-Files

Files with the filename extension **.m** stored in the MATLAB directory, or toolbox subdirectories of the MATLAB directory, are executable files. We call such files *m-files* and they come in two different types: *script* and *function*. The functions that we have already defined and used are contained in function m-files in the MATLAB toolboxes.

1.9.1 Script M-Files

We have shown how we executed a series of statements to evaluate one mathematical expression. When we press the **Enter** key after typing a statement, or a sequence of statements following a **for**, **while**, or **if** statement, the statement, or sequence of statements is executed. The result is stored in the **Workspace** and printed in the **Command Window**. However if the statement, or sequence of statements, is terminated with a

semicolon (;), then the print to **Command Window** is suppressed.

Since the statements are executed as they are entered in the **Command Window**, then we must reenter them if we want to repeat the computations. Such a repeat of computations is necessary if we want to investigate the effect of changing parameters. To avoid having to reenter all the statements, we can generate the statements with a text editor or a word processor. We then store them in a file in the MATLAB directory with the filename extension **.m**. We call such files *script "m-files"* and we can execute the statements in them by typing the file name in the **Command Window** and pressing the **Enter** key. For example, we can store the statements

```
% plsig.m  -  Plots two signals

    t=0:0.01:10;          % Defines time interval
    x=3*sin(0.4*pi*t - 0.1)+4*cos(0.85*pi*t+0.3);
    y=3*sin(0.4*pi*t+0.3)+4*cos(0.85*pi*t - 0.4);
    plot (t,x);  xlabel('t');  ylabel('x');
    figure;
    plot(t,y);  xlabel('t');  ylabel('y');
```

in file **plsig.m** in the MATLAB directory. These statements generate the plots of the two signals $x(t)$ and $y(t)$ for $0 \le t \le 10$. The statements or portion of statements preceded by **%** are comment statements only and are not executed. The two plots are generated when we type

```
>>  plsig          (Enter)
```

in the **Command Window**. The two signals are plotted as Figure No. 1 and Figure No. 2 in two **Figure Windows**. At the completion of execution, Figure No. 2 appears in the **Figure Window**. To view Figure No. 1, click on **Windows - - Figure No. 1**. To make a hard copy of a figure, click on **File - - Print** while the figure is in the **Figure Window**. To erase a figure, click on **File - - Close** while the figure is in the **Figure Window**. To return to the **Command Window** click on **Windows - - Command** or begin typing a new statement.

If we want to make changes to the script m-file **plsig.m** we can type

```
>>  !          (Enter)
```

in the **Command Window** to change to the DOS Window. We are then in the MATLAB directory and can edit the **plsig.m** m-file as desired. To return to the **Command Window** we type **Exit** followed by pressing the **Enter** key.

1.9.2. 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.

Many of the functions we have previously defined, such as **y = sin(x)**, are built-in MATLAB functions. However, others are contained in function m-files in MATLAB toolboxes. For example, the function m-file **mean.m** contains the statements

```
function  y=mean(x)
% MEAN   Average or mean value
%        For vectors, MEAN(X) is the mean
%        value of the elements in x
%        For matrices, MEAN(X) is a row vector
%        containing the mean value of each column
%        See also MEDIAN, STD, MIN, MAX
%        Copyright © 1984-94 by Mathworks, Inc.

[m,n] = size(x)
if m==1
    m=n;
end
y = sum(x)/m;
```

We can create our own functions in the MATLAB directory for computations that we want to perform more than once. For example, we can convert the script m-file **plsig.m** into a function m-file with signal frequency, signal phase, and computation time interval inputs. In this way, we can change the signal frequencies and phases and the computation time interval without having to edit the m-file. Also, since the two signals have the same form and only differ in frequency and phase values, we can generate them with a second function m-file, **ssig.m**.

M-file **ssig.m**

```
function  s=ssig(t,f1,P1,f2,P2)
%   ssig.m - Sum of sinusoids signals
```

```
    s=3*sin(2*pi*f1*t+P2)+4*cos(2*pi*f2*t+P2);
```

M-file **plsigf.m**

```
function  plsigf(t1,t2,f1,f2,Px1,Px2,Py1,Py2)
%  plsigf.m - Plots two sum of sinusoids signals

    t=t1:0.01:t2;        % defines time interval
    x=ssig(t,f1,Px1,f2,Px2);
    y=ssig(t,f1,Py1,f2,Py2);
    plot (t,x);   xlabel('t');   ylabel('x');
    figure;
    plot (t,y);   xlabel('t');   ylabel('y');
```

To create the same plots as we did in Section 1.9.1, we type

```
>>   plsig(0,10,0.2,0.425,-0.1,0.3,0.3,-0.4)
```

in the **Command Window** and press the **Enter** key. With different input variable values, we can generate other sum of sinusoid plots.

2.0 Specific Application Information

In Section 2, we first present several function m-files that we have created specifically for signal and system analyses. These functions are used in examples contained in "Signal and Linear System Analysis" and are useful for end-of-chapter problems and other signal and system analysis tasks. The function m-files are available in this John Wiley and Sons FTP Archive location as "Signal and System Analysis Functions - Carlson " so that they can be downloaded and used.

MATLAB is used in 76 examples in the text, "Signal and Linear System Analysis". In addition, 102 of the end-of-chapter problems are designed specifically for solutions using MATLAB. Many of the remaining end-of-chapter problems can also be solved using MATLAB. The script m-files, function m-files, and data files required for the text examples and problems are available in this John Wiley & Sons FTP Archive location as "Text Example and Data Files - Carlson". The script and function m-files are identified by example number. For example, **e66.m** and **e66a.m** are script m-files used in Example 6.6 and **e94a.m** and **e94be.m** are script m-files used in Example 9.4. The data are stored in mat-files which can be loaded into the MATLAB **Workspace** with the **load** command. The filename indicates either the type of data or the example or problem in which it is used. For example, **sentence.mat** contains data for a spoken sentence, **e1613h.mat** contains data used in Example 16.13, and **p1544.mat** contains data used in Problem 15.44.

2.1 Signal and System Analysis Functions

We have created sixteen function m-files for use in signal and system analyses. These functions encompass:

1. step and ramp signal functions,
2. continuous-time and discrete-time signal plotting functions,
3. continuous-time Fourier series and Fourier transform functions,
4. discrete-time Fourier series and Fourier transform functions,
5. straight-line approximate Bode plot functions,
6. a pole-zero plotting function, and
7. a Butterworth or Chebyshev filter-order-selection 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 Functions

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 Plotting Functions

The plotting functions that we have created contain several input variables. These variables permit us to specify variables plotted, plot scale and grid, plot size and location in the **Figure Window**, and line weight and type. For simplicity, we refer here to plotting continuous-time and discrete-time signals. Actually, the plotting functions are valid for any continuous-variable or discrete-variable function of any independent variable (such as time, frequency, distance, etc.).

2.1.2.1 Continuous-Time Signals

We plot a continuous-time signal by connecting the signal values with straight time segments. A smooth curve is produced if the time increment between the available signal values is small with respect to the time axis length plotted. The continuous-time signal plotting function is

plct(x,y,xg,yg,p,aw,lw,lt,axs)

where the variables **x** and **y** are plotted in the horizontal and vertical directions, respectively.

The vector **xg** specifies the grid line locations along the horizontal axis. The horizontal axis variable is plotted from **xg(1)** to **xg(size(x,2))**. For equal grid line spacing along the horizontal axis, we can use **xg=x1:dx:x2** where **dx** is the grid line spacing. The vector **yg** specifies the same information for the vertical axis.

The vector **p** is a (1x4) vector having the elements

p(1) = horizontal location of plot lower left corner

p(2) = vertical location of plot lower left corner

p(3) = plot horizontal size

p(4) = plot vertical size

All locations and sizes are specified in terms of fractions of the **Figure Window** dimensions. The default figure window aspect ratio is 3 by 4. This means that a fractional distance or size specified in the vertical direction is 3/4 as large as the same fractional distance or size specified in the horizontal direction.

The variables **aw** and **lw** are used to specify the line width for the axes and plotted curve respectively. Good initial choices for the values of these variables are **aw=1** and **lw=1.5**, respectively. The variable **lt** specifies the line type used for the plotted curve. Choices include solid **'-'** and dashed **'--'** lines. See the Reference Guide for all available choices. We use the final input variable **axs** to specify whether we want a linear, **axs=0**, semilog, **axs=1**, or loglog, **axs=2,** plot.

The grid lines are dotted and labelled with values in a Times font. These default specifications can be changed by editing the **plct.m** file. Also, the grid lines can be turned off by replacing **grid on** with **grid off** in the **plct.m** file.

2.1.2.2 Discrete-Time Signals

We plot a discrete-time signal as vertical lines at the signal sample location. The vertical lines are topped with a specified symbol. The discrete-time signal plotting function is

pldt(n,y,T,xg,yg,p,aw,lw,st,lt)

where the variables **y**, **xg**, **yg**, **p**, **aw**, **lw** and **lt** are the same as used in **plct**. In place of the horizontal axis variable **x** in **plct**, we have the signal sample index vector **n** in **pldt**. We also specify the spacing between the signal samples with the variable **T**. Another additional input variable in **pldt** is the symbol type used on top of the vertical signal sample lines. The symbol size is set to **2*lw** in the first script statment in **pldt**. It can be changed by editing this statement (see the help statements in **pldt**).

Before using the plotting function **pldt**, we must create the function **stemm(x,y,symtype,linetype)** by modifying the function **stem(x,y,linetype)** located in directory **c:\matlab\toolbox\matlab\plotxy**. The required modifications are shown in the help statements in **pldt**. They change the function **stem** so that different top-of-line symbols can be specified. The function **stemm** is located in file **stemm.m** available in "Signal and System Analysis Functions - Carlson" in this FTP Archive.

2.1.2.3 Multiple Plots

We can plot two or more functions of the same independent variable on the same set of axes by using **plct** and/or **pldt**. All that is required is that we remain in the same **Figure Window**, use the same vectors for **xg**, **yg**, and **p** in all plotting statements, and include the statements **hold on** and **hold off** after the first and last plotting statements, respectively.

If we want to construct two or more different plots in the same **Figure Window**, then we must specify a different location and size vector for each plot. These vectors must be chosen so that the plots do not overlap.

Finally, if we want to place our plots on different pages when we print them, then we must place them in different **Figure Windows**. This is accomplished by placing the command **figure** between the plot statements. We did this in the **plsig.m** script that we showed in Section 1.9.1.

2.1.2.4 Plotting Examples

The plotting function **plct** is used in many MATLAB examples in "Signal and Linear System Analysis". Example 2.1 is the first example in which **plct** is used. The script given there illustrates the script required to produce continuous-variable plots. Both multiple

plots on one set of axes and multiple, different-axes plots on one page are illustrated in this example. Also illustrated is the use of the **text** function to label individual curves. With this function, we specify the text for each label and the corresponding label location, in axis coordinates. Note that the location specified is the left center of the text printed. Example 6.17 is the first example in which we set **axs** to something other than **0** so that a non-linear axis scale is produced. In this example, **axs=1** and a logarithmic horizontal scale is produced. Note that the horizontal separation of grid lines specified by the **a** vector cannot use the simple colon notation. This is true since the grid separation increment is not constant.

Example 2.10 is the first example in which the discrete-variable plotting function is used. The plots in this example illustrate the use of several different top-of-line symbols to distinguish the different signal plots. In producing these plots, **pldt** was edited to change the first statement from **ss=2*lw** to **ss=6*lw**. This produces top-of-line symbols that are large enough to permit us to distinguish different symbol types.

2.1.3 Continuous-Time Fourier Series and Fourier Transform 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.

2.1.3.1 Fourier Series Coefficients

The function

[Xn,no,fo]=ctfsc(t,x,plt,ev,a,bm,ba,YL1,YL2,YL3)

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

If desired, we can plot the Fourier series coefficient amplitudes and angles as a function of frequency f. The plot is enabled by setting **plt=1**. If the signal is real and even, then the series coefficients are real and even. In this case, the Fourier series coefficients can be plotted rather that their amplitudes and angles, if desired. This plotting format is used if we set variable **ev** equal to one. Vectors **a**, **bm**, and **ba** specify the plotting interval and grid line spacing for the horizontal (frequency) axis, the coefficient amplitude or total coefficient axis, and the coefficient angle axis, respectively. We use the final three input variables to specify the vertical axis labels for the total coefficient, coefficient amplitude, and coefficient angle axes, respectively. For example, if the signal is y(t), then **YL1='Xn'**, **YL2='|Xn|'**, and **YL3='arg(Xn)'**. No input variables are provided to specify plot locations or sizes. If these must be changed, then we need to edit the appropriate **p** vector in the **pldt** statements in m-file **ctfsc.m**.

The first example in "Signal and Linear System Analysis" using the function **ctfsc** is Example 5.3. No coefficient plots are made so all variables after **plt** are arbitrarily set to zero. The first example for which coefficient plots are made is Example 5.8.

2.1.3.2 Truncated Fourier Series

The second Fourier series related function is

[xfs,Xnn]=ctfs(t,x,Xn,no,fo,N,plt,os,a,b,p,YL)

This function first selects the 2N + 1 Fourier series coefficients, **Xnn**, centered on X_0, 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. This fact is illustrated in Example 5.3, which is the first example using **ctfs** in "Signal and Linear System Analysis".

The input variables **Xn**, **no**, and **fo** are the output variables obtained from function **ctfsc**. The remaining input variables, **x**, **plt**, **os**, **a**, **b**, **p**, and **YL,** are used if we want to plot the truncated Fourier series. The plot is enabled if variable **plt** is set equal to one. Vectors **a** and **b** specify the plotting interval and grid line spacing for the horizontal (time) axis, and the vertical axis, respectively. Vector **p** specifies the plot location and size in the same way that it does for the function **plct**. The input variable **YL** specifies the vertical axis label. Suppose that, for comparison purposes, we want to display a dashed line plot of the original signal from which the Fourier series expansion was computed. To do this, we set **os=1** and vector **x** equal to the original signal (see Example 5.3).

2.1.3.3 Fourier Transform

The function

[f,X,N,no]=ctft(t,x,dfm,plt,ev,a,bm,pm,ba,pa,YL1,YL2,YL3)

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**, then the computed transform will have some distortion since it is the transform of a 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 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**.

If desired, we can plot the amplitude and angle of the Fourier transform, or, if the signal is real and even, the Fourier transform. The plot is enabled by setting **plt=1**. Variables **ev**, **YL1**, **YL2**, and **YL3**, and vectors **a**, **bm**, and **ba** perform the same functions as they do in the Fourier series coefficient function (see Section 2.1.3.1). We specifiy the amplitude (or transform) and angle plot locations and sizes with input vectors **pm** and **pa** respectively.

The function **ctft** is first used in Example 5.11 in "Signal and Linear System Analysis". This example illustrates the script required to use **ctft**.

2.1.3.4 Inverse Fourier Transform

The function

[t,x,N]=ctift(f,X,dtm,plt,cc,a,bm,pm,ba,pa,YL1,YL2,YL3)

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 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.

The remaining input variables are all associated with the plot that is generated if **plt=1**. These are the same variables used in the Fourier transform (See Section 2.1.3.3) except that **cc** replaces **ev**. The variable **cc** performs the same plot type selection function that **ev** does. We set it equal to one when the transform values have the complex conjugate symmetry X(-f)=X*(f) because the inverse transform is real in this case.

The function **ctift** is discussed prior to Example 5.11 and first used in Example 5.12 in "Signal and Linear System Analysis". In Example 5.12, a transform generated by **ctft** is inverted to show that the transformed signal is recovered. Also, the result of truncating the transform is shown.

2.1.4 Discrete-Time Fourier Series and Fourier Transform Functions

We have also developed four functions to be used in performing Fourier analyses of discrete-time signals. The first function computes discrete-time Fourier series coefficients corresponding to an interval of a discrete-time signal. The second function computes samples of the discrete-time Fourier series representation of a discrete-time signal over a specified time interval. The last two functions compute the discrete-time Fourier transform and inverse discrete-time Fourier transform, respectively. All four functions are equivalent to those developed for continuous-time signals and discussed in Section 2.1.3. We will only discuss differences here.

2.1.4.1 Discrete-Time Fourier Series Coefficients

We use the function

[Xm,mo,ro]=dtfsc(n,x,T,plt,nf,ev,a,bm,ba,YL1,YL2,YL3)

to compute discrete-time Fourier Series expansion coefficients **Xm**. These coefficients correspond to the portion of the discrete-time-signal **x** within the expansion interval specified by the signal sample index vector **n**. There are **ns=size(n,2)** discrete-time Fourier series coefficients computed and the normalized frequency between them is **ro=1/ns**. Series coefficients are computed for both positive and negative frequencies and the coefficient X_0 corresponding to r = 0 is stored in **Xm(mo)**. The values of **ro** and **mo** are function outputs along with **Xm**.

The discrete-time Fourier transform interval can begin anywhere between **-10*ns** and **10*ns**. These limits can be changed, if desired, by making the script changes specified in the help statements in the function m-file **dtfsc.m**.

The discrete-time Fourier series coefficient amplitudes and angles or, if possible, the coefficients themselves can be plotted by setting the input variable **plt** equal to 1. We can generate the plots as a function of normalized frequency, r, by setting input variable **nf** equal to one. If we set **nf** equal to any other value, plots versus the frequency f are produced. The input variable **T** specifies the discrete-time signal sample spacing. It is only used to determine the correct spacing between the discrete-time Fourier series coefficient when they are plotted versus the frequency **f**.

All remaining input variables for the function **dtfsc** control plot characteristics and are the same as those we defined for function **ctfsc** (see Section 2.1.3.1). The first example in "Signal and Linear System Analysis" that uses the function **dtfsc** is Example 12.3. We make no coefficient plots in this example so all variables after the plot variable are arbitrarily set to zero. The first example in which we make coefficient plots is Example 12.7.

2.1.4.2 Discrete-Time Fourier Series

The function

[xfs,Xmm]=dtfs(n,x,T,Xm,mo,ro,N,plt,os,a,b,p,YL)

computes the discrete-time signal values **xfs** that correspond to the discrete-time Fourier series coefficients **Xm**. Computations are made for the time interval specified by the input sample index vector **n** and the input sample spacing **T**. This interval does not need to be the same as the expansion interval used to generate the coefficients. This fact is illustrated in Example 12.3 in "Signal and Linear Systems Analysis", which is the first example that uses **dtfs**.

The input variables **Xm**, **mo**, and **ro** are the output variables obtained from function **dtfsc**. All discrete-time Fourier series coefficients are used to obtain the discrete-time Fourier series values if the input variable **N** is selected so that **2*N+1** is greater than or equal to **ns=size(Xm,2)**. Otherwise, the set of coefficients is truncated to include only **2*N+1** coefficients centered on **r=0**. In this case, the discrete-time Fourier series is only an approximation to the original discrete-time signal (see discussion and plot following Example 12.3).

All the remaining input variables are used only if we plot the discrete-time Fourier series. They are the same as those defined for **ctfs** (see Section 2.1.3.2).

2.1.4.3 Discrete-Time Fourier Transform

The function

[f,X,N,no]=dtft(n,x,T,dfm,ev,nf,a,bm,pm,ba,pa,YL1,YL2,YL3)

computes the discrete-time Fourier transform of the portion of the discrete-time signal **x** that is contained in the interval specified by the sample index vector **n**. The signal portion must begin at **n(1)<=0** and end at **n(size(n,2))>0**. The spacing between signal sample values is specified by the input variable **T**. The input variable **nf** specifies if the plot is to be generated in terms of normalized frequency (**nf** equal to one) or frequency (**nf** equal to any other value).

The remaining input variables and all output variables are identical to those for the function **ctft**. Refer to Section 2.1.3.3 (replacing **dt** with **T**) for a discussion of them. The function **dtft** is first used in Example 12.13.

2.1.4.4 Inverse Discrete-Time Fourier Transform

The last function that we have created for the discrete-time Fourier analysis of discrete-time signals is

[n,x,T]=dtift(f,X,plt,cc,seq,a,bm,pm,ba,pa,YL1,YL2,YL3)

This function computes the inverse **x** of the discrete-time Fourier transform **X**. The input vector **f** specifies the frequency interval and sample spacing for the input discrete-time Fourier transform vector **X**. These input vectors are the outputs of function **dtft** and encompass a frequency interval of length **1/T**. The transform and inverse transform both contain **ns=size(X,2)** samples. The internal function variable **df** is the transform sample spacing. The sample index and the spacing between the inverse transform samples (discrete-time signal) are the output vector **n** and the output variable **T**, respectively.

The input variables in addition to **f** and **X** are the same for those for the function **ctift** except that **dtm** is replaced by **seq**. The variable **dtm** is not needed since the signal sample spacing is not selectable but is **T=1/(ns*df)**. A sequence is plotted (versus n) if **seq=1**; otherwise, a discrete-time signal (versus t) is plotted. All other input variables select plot characteristics. They are defined in Sections 2.1.3.1, 2.1.3.3, and 2.1.3.4.

The function **dtift** is first used in Example 12.13 in "Signal and Linear System Analysis". The example illustrates that the discrete-time signal is recovered from its discrete-time Fourier transform.

2.1.5 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-time approximations and the straight-time 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.

The first example that uses the Bode plot functions in "Signal and Linear System Analysis" is Example 6.17. The frequency response in this example contains three numerator factors and three denominator factors.

2.1.5.1 Transfer Function Parameter Computation

The function

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

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

$$H(s)=[n(1)s^{\alpha}+...+n(\alpha)s+n(\alpha+1)] / [d(1)s^{\beta}+...+d(\beta)s+d(\beta+1)]$$

or frequency response

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

The numerator and the denominator coefficients are stored in the input vectors **n** and **d**, respectively. Function outputs 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 function where, for linear factors, **bf>0**, **bf=0**, and **bf<0** indicate LHP, Imaginary Axis, and RHP plane poles or zeros, respectively,

ex = a row-vector of factor exponents 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 jω factor. Quadratic factors that correspond to LHP, Imaginary Axis, and RHP poles are indicated with positive, zero, and negative damping ratios, respectively.

2.1.5.2 Straight-Line Data Computation

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

[am,ph]=slbode(w,c,bf,ex,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 2.1.5.1.

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.

2.1.6 Pole-Zero Plotting Function

There are pole-zero plotting functions available in the MATLAB Signal Processing and Control Systems toolboxes. We have created an additional poles-zero plotting function to provide additional flexibility in plot generation. The function is

. **[z,p,k]=pzp1(form,a,b,po,amv,grs,tl,gr)**

The input vectors **a** and **b** specify the transfer function numerator and denominator coefficients, respectively, if **form** is specified as **'tf'**. On the other hand, if **form** is specified as **'pz'**, then the vectors **a** and **b** specify the transfer function zero and pole locations, respectively.

The input vector **po** specifies the plot location and size (see help statements in file **pzpl.m**). All locations and sizes are specified in terms of fractions of the Figure Window dimensions. A square plot is produced even if the vector elements **po(3)** and **po(4)** are not equal. They should be set equal to produce an easily predictable plot size.

The input variable **amv** specifies the maximum absolute value plotted on both the horizontal and vertical (real and imaginary) axes. That is, plot scales extend from **-amv** to **+amv** on both axes.

The input variables **grs**, **tl**, and **gr** control the characteristics of the plot grid. Variable **grs** specifies the grid line or tick mark spacing. Dotted grid lines are produced if **gr** is set equal to one. Otherwise, only tick marks are produced. The variable **tl** is used to suppress the grid line or tick mark labels if it is set equal to zero. Otherwise, grid labeling is produced.

In addition to the pole-zero plot, function outputs include the zeros, vector **z**, the poles, vector **p**, and the transfer function multiplicative constant, variable **k**. The variable **k** is set equal to one if zeros and poles are supplied in the input.

The first example that uses function **pzpl** in "Signal and Linear System Analysis" is Example 7.20. Other examples in which it is used include Example 8.4 and 14.23. In Example 14.23, we add the unit circle to the pole-zero plot since the unit circle is an important feature in the z-plane.

2.1.7 Butterworth and Chebyshev Filter Order Selection Function

The last function created for signal and system analysis is

[n,wss]=abcord(type,pband,wc,ws,gs,dBwrmg,maxg,rdB)

We use this function to determine the Butterworth or Chebyshev filter order required to produce desired stopband gain characteristics. The low-pass filter (LPF) order is selected to produce filter stopband gain that is less than or equal to gain **gs** at frequency **ws**. The stopband frequency **ws** is larger than the cutoff frequency **wc**. For the high-pass filter (HPS), the frequency **ws** is in the stopband located at a frequency less than the cutoff frequency **wc**.

Two specified cutoff and stopband frequencies are required for band-pass (BPF) and band-reject (BRF) filters. Thus, for these filter types, **wc** and **ws** are two element row vectors. If the filter is a BPF, then the specified stopband frequencies are above and below the filter passband defined by the upper and lower cutoff frequencies. On the other hand, the specified stopband frequencies are in the stopband between the upper and lower cutoff frequencies for a BRF.

We specify whether we are using a Butterworth or Chebyshev filter by setting the input variable type equal to **'b'** or **'c'**, respectively. The filter passband characteristic is specified with the input variable **pband**. We use the values **'lp'**, **'hp'**, **'bp'**, or **'br'** for the input variable **pband** to specify a LPF, HPF, BPF, or BRF, respectively.

The stopband gain **gs** can be specified either as a numerical gain value or as gain in dB with respect to the maximum filter gain. We set the input variable **dBwrmg** equal to **'n'** or **'y'** to indicate whether we are using the first or second type of stopband gain specification, respectively. If we are using the first type (that is, numerical gain), then we set the input variable **maxg** equal to the filter maximum gain. Otherwise, we set it to an arbitrary number since it is not used.

The final input variable is **rdB**. We use this variable to specify the passband ripple in dB for a Chebyshev filter.

The outputs of the function **abcord** are the filter order **n** and the scaled stopband frequencies **wss**. The scaled stopband frequencies are those used in **abcord** to select the filter order from the normalized LPF prototype data.

The function **abcord** is first used in Examples 8.8, 8.9, and 8.10 in "Signal and Linear System Analysis". These examples illustrate low-pass and high-pass filter design. We also use it in Example 15.11 to produce a discrete-time filter design.

2.2 Techniques and Functions Used in Text Examples

In Section 1.0, we presented basic MATLAB statement and format characteristics, operations, and functions that we use in MATLAB script to perform analyses. In addition, in Section 2.1, we described sixteen function m-files that we created specifically for signal and system analyses. We use other MATLAB functions and various programming techniques in the 76 MATLAB examples in the text "Signal and Linear System Analysis". In this section, we discuss other functions used and those techniques which may not be directly obvious to the student. We reference the functions and techniques to the first example in which they are used to illustrate the script required to utilize them. The order in which the functions and techniques are discussed is the order in which they appear in the text. Thus, the discussion is organized by example number. All examples are not mentioned since many only use techniques and functions that have already been used in earlier examples. We have already mentioned the first example in which the created signal and system analysis functions are used. We will not repeat these references here.

The text is divided into three parts. These are: Part I - Fundamental Concepts, Part II - Continuous-Time Signals and Systems, and Part III - Discrete-Time Signals and Systems. We discuss examples in each part in a separate section. More examples are discussed in the earlier chapters than in the later chapters since the functions and techniques used are cumulative.

2.2.1 Part I - Fundamental Concepts

Part I of "Signal and Linear System Analysis" contains Chapters 1 and 2. MATLAB is first used in Chapter 2 where evaluation and plotting of both continuous-time and discrete-time signals are the first tasks performed.

In Example 2.5, we demonstrate how to use a 0-1 array to generate a one-dimensional array of constants, **hm**, having the same size as the variable array **t**. This permits us to use an array operation rather than a **for** loop to compute the signal values **y**.

Example 2.10 utilizes a large set of nested **for** loops to compute several signals. The script to do this can be greatly simplified and looks more like the defined signal expressions if we use the **sum** function. This requires setting up the appropriate variable arrays using **meshgrid** (see Section 1.8.1) so that we can use the array operations. It also makes the computation faster; however, since so few points are computed, the speed advantage is not very significant. For those who want to pursue the array operations approach, the appropriate script is included as file **e210ao.m** in the "Text Example and Data Files - Carlson" section in the FTP archive. The **meshgrid** function statements are annotated to aid understanding of the arrays created.

2.2.2 Part II - Continuous-Time Signals and Systems

Chapters 3 through 9 are contained in Part II of "Signal and Linear System Analysis". Example 3.6 illustrates the use of step and ramp signal generation functions, **us.m** and **ur.m**, in the generation of a piecewise defined signal.

In Example 3.11, we use the numerical integration function **quad8('e311x',0,0.5)** to integrate the signal defined by the functions **x=e311x(t)**. Integration is performed over the interval from **t=0** to **t=0.5**. The function **quad8** is a *function function*. It operates on a specified MATLAB function (**e311x** in this case) rather than on variable arrays. Iteration is used until the relative error is less than a specified value (default value = 0.001). If this relative error is not achieved in 10 iterations, a warning message is printed and the computation proceeds using the value obtained.

Example 4.4 also uses **quad 8**. In this case, the integrand function **e44ig** is a function of the variable **x** and **t**. Integration is performed over the variable **x** and a value of the variable **t**, **t(n)**, is used in the integration. This example shows that the value of the variable **t** is included by using six input variables where the sixth input variable is **t(n)**. The fourth and fifth variables must be specified if we want to specify the sixth variable. These variables are the specified relative error and a plot selecting variable, respectively. We set them to **0.001** and **0** to produce the default relative error value and no plot.

A simple method for performing numerical integration used the trapezoidal integration

rule. This is illustrated in Example 3.12.

The function **conv** is designed to perform the discrete-time convolution sum (see Example 11.11). However, by using eq. 4.27 in the text, we can use **conv** to compute an approximation to the continuous-time convolution integral. This is illustrated in Example 4.10. The function **conv** can also be used to find the polynomial coefficients of a product of two polynomials For example, the coefficients for the polynomial product

$$(2x^2 + 5x - 3)(4x^2 - 10x + 6) = 8x^4 - 50x^2 + 60x - 18$$

are obtained as follows:

```
>>  a=[2  5  -3];  b=[4  -10  6];  c=conv(a,b)
c =
     8   0   -50   60   -18
```

The **angle (A)** function finds the principal angle corresponding to each element of the array **A**. That is, it finds angles between $-p$ and $+p$ radians. Many times a plot of the angle is easier to interpret if we plot angles that extend to less than $-p$ radians, or greater than $+p$ radians. We can do this by removing the $+2p$ and $-2p$ jumps that occur in the principal angle plot. These jumps occur when the angle passes $-np$ and $+np$ in the negative and positive directions, respectively, where n is an odd integer. The MATLAB function **unwrap** removes these $\pm 2p$ jumps. The unwrap starts with the first value in the angle vector. Thus **unwrap** cannot correct for a net jump of $\pm m2p$ from $f =$ **f(1)** to $f = 0$, where m is an integer. we can remove this uncorrected net jump by adding or subtracting the multiple of $2p$ that will make the unwrapped phase angle equal to the principal phase angle at $f = 0$. The function **unwrap** is first used in Example 5.11 where $2p$ is subtracted so angles match at $f = 0$.

In Example 6.17, we introduce the function **h=freqs(n,d,w)** found in the Signal Processing and Signal and System toolboxes. This function produces the system frequency response values **h** for the frequency interval **w**, where **w** is specified in radians/second. The system parameter inputs are the numerator, **n**, and denominator, **d**, coefficients of the system transfer function (ratio of polynomials in s) or the system frequency response (ratio of polynomials in jw). If we want to generate frequency response values corresponding to the frequency interval vector **f** in Hertz, then we replace **w** by **2*π*f** in the input variables.

The first three MATLAB examples in Chapter 7 (Examples 7.5, 7.9, and 7.15) use the Symbolic Math Toolbox to find Laplace transform and inverse Laplace transform expressions. The Symbolic Math notation **Heaviside(t)** is introduced for a unit step function and used in all three examples. The Laplace transform **laplace** and inverse Laplace transform **invlaplace** functions often produce unnecessarily complicated results. These results can sometimes be simplified with the **simplify** command. We find that the **laplace** and **invlaplace** functions cannot compute Laplace transform and inverse Laplace

transform expressions for arbitrary Laplace transformable functions. Thus, their utility is limited.

Sometimes we want to evaluate a Symbolic-Math-produced mathematical expression for values of the independent variable. This is done with the **eval** function, as illustrated in Example 7.15. Evaluation is only possible if all terms and operations in the Symbolic Math expression correspond to defined terms and operations. Replacements can be made if necessary. For example, **Heaviside** is replace with **us** in Example 7.15 since a non-symbolic expression does not recognize the **Heaviside** function but does recognize the **us** function as a representation of a unit step. Likewise, all multiplications in Symbolic Math are represented by *****. Since we want array operations in Example 7.15, we replace ***** with **.***. The function that provides the substitution capability is **strrep** and its use is illustrated in Example 7.15.

We use partial fraction expansion in the determination of inverse Laplace transforms and inverse z-transforms of rational functions. The function **[r,p,k]=residue(n,d,)** computes the partial fraction expansion coefficients **r** corresponding to the poles **p** of a rational transform. The function inputs are the numerator and denominator coefficients specified by the one-dimensional row-arrays **n** and **d**. The partial fraction expansion coefficients and pole values are function outputs and are contained in one-dimensional column-arrays. If the rational function specified by **n** and **d** is proper, the output one-dimensional row-array **k** is empty. Otherwise it contains the coefficient of the powers of *s* or *z* that we obtain when we divide the rational function to produce a rational remainder. Example 7.25 is the first example in which we use the function **residue**.

We can use the function

[z,p k]=tf2zp(n,d)

found in the Control System Toolbox, to find the zeros **z**, poles **p**, and multiplicative constant **k** (called gain) corresponding to a rational Laplace transform or z-transform. The zero and pole values are placed in one-dimensional column arrays. The coefficients of the transform numerator and denominator are supplied by the input one-dimensional row-arrays **n** and **d**. The degree of the numerator must be less than or equal to the degree of the denominator.

The gain is the ratio of the coefficients of the highest degree numerator and denominator terms. Together, the gain, zeros, and poles specify the transform completely. Thus, the transform can be obtained from the zeros, poles, and gain with the inverse function

[n,d]=zp2tf(z,p,k)

The functions **tf2zp** and **zp2tf** are first used in Example 8.3.

The functions **tf2zp** and **zp2tf** can be used for multiple transforms having the same

denominator. In this case, **n** is a matrix with rows corresponding to the numerator coefficients for each transform and **z** is an array having columns that correspond to each of the numerator polynomials. The gain variable **k** is a one-dimensional column-array containing the gain factors corresponding to each transform.

The Signal Processing and Signal and System Toolboxes contain functions that produce Butterworth and Chebyshev filter designs. The designs are in terms of transfer function numerator coefficients **nu** and denominator coefficients **de**. The functions for continuous-time filter design are

```
[nu,de]=butter(n,Wn,'ftype','s')
```

and

```
[nu,de]=cheby1(n,Rp,Wn,'ftype','s')
```

The input variables **n** and **Rp** are the filter order and the Chebyshev filter passband gain ripple in dB, respectively. The input **Wn** specifies the filter cutoff frequency, or frequencies. If a single value is supplied, then either a low-pass filter or a high-pass filter is designed. The filter is a low-pass filter if variable **'ftype'** is left out and a high-pass filter if **'ftype'** is set equal to **'high'**. If a two element row array is supplied for **Wn**, then either a band-pass filter or a band-reject filter is designed. The filter is band-pass if **'ftype'** is left out and band-reject if **'ftype'** is set equal to **'stop'**. Illustrations of using **butter** and **cheby1** in continuous-time filter design are shown in Examples 8.4, 8.5, 8.8, 8.9, and 8.10. Note that computer imprecision causes unwanted small coefficients in the numerator (See Example 8.5, 8.8, 8.9,and 8.10 for discussion).

If the input variable **'s'** is left out, then **butter** and **cheby1** design discrete-time filters instead of continuous-time filters. The technique used is the bilinear tranformation design technique discussed in Section 15.3 in “Signal and Linear System Analysis”. We illustrate this use of the filter design functions in Example 15.11.

We have only discussed the design of Type I Chebyshev filters. Type II filters have equal ripple in the stopband instead of the passband, and can be designed using the function **cheby2**.

In Example 9.2, we use the **meshgrid** - **sum** combination of functions so that we can avoid a **for** loop and use array operations to increase computation speed. This illustrates the use of this combination of functions which we first discussed in Section 1.7.1.

2.2.3 Part III - Discrete-Time Signals and Systems

Chapters 10 through 16 are contained in Part III of “Signal and Linear System Analysis”. Only a few examples in this part contain functions or techniques that we have not already discussed in Sections 2.2.1 and 2.2.2 or in the System and Signal Analysis

Functions section (Section 2.1).

Examples 13.3 and 13.5 illustrate the use of the **save** and **load** commands in saving data to a **mat** file and then reloading and using it later. If you do not recall the names of the variables saved in a **mat** file, you can load the file in a cleared **Workspace** and then use the **whos** command to display the **Workspace** contents.

In Examples 14.4, 14.6, and 14.10, we use Symbolic Math to perform z-transforms. We also evaluate Symbolic Math produced expressions for values of the independent variable. All of these operations parallel those that we used for continuous-time signals and Laplace transforms in Examples 7.5, 7.9, and 7.15. See the discussion for these examples in Section 2.2.2. One anomaly occurs in Examples 14.6. The anomaly is that the MATLAB symbolic z-tranform will not compute the transform of a step function (**Heaviside** function) if it is multiplied by a non-integer constant. This problem was avoided in Example 14.6 by first finding the transform of **K*Heaviside(t)** and then substituting the numerical value for *K*.

The final pair of function that we consider are the **fft** and **ifft** functions. These functions compute the Discrete Fourier Transform (DFT) and inverse Discrete Fourier Transform (IDFT) using the Fast Fourier Transform (FFT) algorithm. They are very important for signal and system analysis. The greatest computation speed is achieved if the number of signal or transform points used is a power of two. Computation speed is still quite good if the number of points factors into a large number of prime factors. The slowest computation speed occurs when the number of points is a prime number. For example, 0.05, 0.16, 0.71, and 10.77 seconds are required to compute the FFT of an exponential if 4096, 4095, 4097, and 4099 points are used, respectively. The time was measured using the **tic** and **toc** functions and 4096, 4095, 4097, and 4099 have 12, 5, 2, and 1 prime factors respectively. Note that 4096 is a power of 2 and 4099 is a prime number.

Computation speed is not usually a problem unless a large number of points is used or we need to compute many transforms. In these cases, we can usually change the number of points by a just a few points to obtain more prime factors, if the original number of points has only a few prime factors. The prime factors **R** of the number **N** are obtained by using the function **R=factor(N)**. This function is located in the Symbolic Math Toolbox.

The **fft** and **ifft** functions are first used in Example 16.2. They are also used in a number of additional examples following Example 16.2.