

MATLAB Tutorial

This tutorial is available as a supplement to the textbook *Fundamentals of Signals and Systems Using the Web and Matlab*, 3rd edition by Edward Kamen and Bonnie Heck, published by Prentice Hall. The tutorial covers basic MATLAB commands that are used in introductory signals and systems analysis. It is meant to serve as a quick way to learn MATLAB and a quick reference to the commands that are used in this textbook. For more detailed information, the reader should consult the official MATLAB documentation. An easy way to learn MATLAB is to sit down at a computer and follow along with the examples given in this tutorial and the examples given in the textbook.

The tutorial is designed for students using either the professional version of MATLAB (ver. 7.01) with the Control Systems Toolbox (ver. 6.1), the Signal Processing Toolbox (ver. 6.2.1), and the Symbolic Math Toolbox (ver. 3.1) or using the Student Edition of MATLAB (ver. 7.0).

The topics covered in this tutorial are:

1. MATLAB Basics	2
A. Definition of Variables	2
B. Definition of Matrices	4
C. General Information	6
D. M-files	6
2. Fourier Analysis	9
3. Continuous Time System Analysis	11
A. Transfer Function Representation	11
B. Time Simulations	15
C. Frequency Response Plots	17
D. Analog Filter Design	18
E. Control Design	19
F. State Space Representation	20
4. Discrete-Time System Analysis	22
A. Convolution	22
B. Transfer Function Representation	22
C. Time Simulations	23
D. Frequency Response Plots	25
E. Digital Filter Design	25
F. Digital Control Design	27
G. State Space Representation	29
5. Plotting	30
6. Loading and Saving Data	32

There are several predefined variables which can be used at any time, in the same manner as user-defined variables:

```
i      sqrt(-1)
j      sqrt(-1)
pi     3.1416...
```

For example,

```
y = 2*(1+4*j)
```

```
yields:      y =
              2.0000 + 8.0000i
```

There are also a number of predefined functions that can be used when defining a variable. Some common functions that are used in this text are:

```
abs      magnitude of a number (absolute value for real numbers)
angle    angle of a complex number, in radians
cos      cosine function, assumes argument is in radians
sin      sine function, assumes argument is in radians
exp      exponential function
```

For example, with `y` defined as above,

```
yields:      c = abs(y)
              8.2462
```

```
yields:      c = angle(y)
              1.3258
```

With `a=3` as defined previously,

```
yields:      c = cos(a)
              -0.9900
```

```
yields:      c = exp(a)
              20.0855
```

Note that `exp` can be used on complex numbers. For example, with $y = 2+8i$ as defined above,

$$\begin{aligned} c &= \exp(y) \\ \text{yields: } c &= -1.0751 + 7.3104i \end{aligned}$$

which can be verified by using Euler's formula:

$$c = e^2 \cos(8) + je^2 \sin(8)$$

B. Definition of Matrices

MATLAB is based on matrix and vector algebra; even scalars are treated as 1x1 matrices. Therefore, vector and matrix operations are as simple as common calculator operations.

Vectors can be defined in two ways. The first method is used for arbitrary elements:

$$v = [1 \ 3 \ 5 \ 7];$$

creates a 1x4 vector with elements 1, 3, 5 and 7. Note that commas could have been used in place of spaces to separate the elements. Additional elements can be added to the vector:

$$v(5) = 8;$$

yields the vector $v = [1 \ 3 \ 5 \ 7 \ 8]$. Previously defined vectors can be used to define a new vector. For example, with v defined above

$$\begin{aligned} a &= [9 \ 10]; \\ b &= [v \ a]; \end{aligned}$$

creates the vector $b = [1 \ 3 \ 5 \ 7 \ 8 \ 9 \ 10]$.

The second method is used for creating vectors with equally spaced elements:

$$t = 0:.1:10;$$

creates a 1x101 vector with the elements 0, .1, .2, .3,...,10. Note that the middle number defines the increment. If only two numbers are given, then the increment is set to a default of 1:

$$k = 0:10;$$

creates a 1x11 vector with the elements 0, 1, 2, ..., 10.

Matrices are defined by entering the elements row by row:

```
M = [ 1 2 4 ; 3 6 8 ] ;
```

creates the matrix

$$M = \begin{bmatrix} 1 & 2 & 4 \\ 3 & 6 & 8 \end{bmatrix}$$

There are a number of special matrices that can be defined:

```
null matrix:          M = [ ] ;
```

```
nxm matrix of zeros:  M = zeros(n,m) ;
```

```
nxm matrix of ones:   M = ones(n,m) ;
```

```
nxn identity matrix:  M = eye(n) ;
```

A particular element of a matrix can be assigned:

```
M(1,2) = 5 ;
```

places the number 5 in the first row, second column.

In this text, matrices are used only in Chapter 12; however, vectors are used throughout the text. Operations and functions that were defined for scalars in the previous section can also be used on vectors and matrices. For example,

```
a = [ 1 2 3 ] ;  
b = [ 4 5 6 ] ;  
c = a + b
```

yields: c =

```
5 7 9
```

Functions are applied element by element. For example,

```
t = 0:10 ;  
x = cos(2*t) ;
```

creates a vector x with elements equal to $\cos(2t)$ for $t = 0, 1, 2, \dots, 10$.

Operations that need to be performed element-by-element can be accomplished by preceding the operation by a ".". For example, to obtain a vector x that contains the elements of $x(t) = t\cos(t)$ at specific points in time, you cannot simply multiply the vector t with the vector $\cos(t)$. Instead you multiply their elements together:

```
t = 0:10;  
x = t.*cos(t);
```

C. General Information

MATLAB is case sensitive so "a" and "A" are two different names.

Comment statements are preceded by a "%".

On-line help for MATLAB can be reached by typing `help` for the full menu or typing `help` followed by a particular function name or M-file name. For example, `help cos` gives help on the cosine function.

The number of digits displayed is not related to the accuracy. To change the format of the display, type `format short e` for scientific notation with 5 decimal places, `format long e` for scientific notation with 15 significant decimal places and `format bank` for placing two significant digits to the right of the decimal.

The commands `who` and `whos` give the names of the variables that have been defined in the workspace.

The command `length(x)` returns the length of a vector x and `size(x)` returns the dimension of the matrix x .

D. M-files

M-files are macros of MATLAB commands that are stored as ordinary text files with the extension "m", that is *filename.m*. An M-file can be either a function with input and output variables or it can be a list of commands. All of the MATLAB examples in this textbook are contained in M-files that are available from the textbook website.

The following describes the use of M-files on a PC version of MATLAB. MATLAB requires that the M-file must be stored in a directory that is specified in the MATLAB path list. For example, consider using MATLAB on a PC with a user-defined M-file stored in a directory called "\MATLAB\MFILES". Then to access that M-file, add that directory to the path by clicking "File" then clicking "Set Path" and following directions.

The M-files associated with this textbook should be downloaded from www.ece.gatech.edu/users/192/book/M-files.html and copied to a user-specified directory. That directory should be added to the path. The M-files that come with MATLAB are already in the path.

To create an M-file, click on “File” then click on “New” then click on “M-file” and an editor screen will appear. Type in your MATLAB commands and then save the file to a directory that is in the path. The M-file is run by typing the name of the M-file from the MATLAB command window (without the .m extension). For example, suppose an M-file named `example.m` is located in the path. Then typing `example` from the command prompt runs that M-file.

As example of an M-file that defines a function, create a file in your working directory named `yplusx.m` that contains the following commands:

```
function z = yplusx(y,x)
z = y + x;
```

The following commands typed from within MATLAB demonstrate how this M-file is used:

```
x = 2;
y = 3;
z = yplusx(y,x)
```

MATLAB M-files are most efficient when written in a way that utilizes matrix or vector operations. Loops and if statements are available, but should be used sparingly since they are computationally inefficient. An example of the use of the command `for` is

```
for k=1:10,
    x(k) = cos(k);
end
```

This creates a 1x10 vector `x` containing the cosine of the positive integers from 1 to 10. This operation is performed more efficiently with the commands

```
k = 1:10;
x = cos(k);
```

which utilizes a function of a vector instead of a for loop. An `if` statement can be used to define conditional statements. An example is

```
if(a <= 2),
    b = 1;
elseif(a >=4)
    b = 2;
else
    b = 3;
```

end

The allowable comparisons between expressions are \geq , \leq , $<$, $>$, $==$, and $\sim=$.

Several of the M-files written for this textbook employ a user-defined variable which is defined with the command `input`. For example, suppose that you want to run an M-file with different values of a variable `T`. The following command line within the M-file defines the value:

```
T = input('Input the value of T: ')
```

Whatever comment is between the quotation marks is displayed to the screen when the M-file is running, and the user must enter an appropriate value.

2. Fourier Analysis

Commands covered: `dft`
 `idft`
 `fft`
 `ifft`
 `contfft`

The `dft` command uses a straightforward method to compute the discrete Fourier transform. Define a vector `x` and compute the DFT using the command

```
X = dft(x)
```

The first element in `X` corresponds to the value of $X(0)$. The function `dft` is available from the MathWorks ftp site and is defined in Figure C.2 of the textbook.

The command `idft` uses a straightforward method to compute the inverse discrete Fourier transform. Define a vector `X` and compute the IDFT using the command

```
x = idft(X)
```

The first element of the resulting vector `x` is $x[0]$. The function `idft` is available at the MathWorks ftp site and is defined in Figure C.3 of the textbook.

For a more efficient but less obvious program, the discrete Fourier transform can be computed using the command `fft` which performs a Fast Fourier Transform of a sequence of numbers. To compute the FFT of a sequence $x[n]$ which is stored in the vector `x`, use the command

```
X = fft(x)
```

Used in this way, the command `fft` is interchangeable with the command `dft`. For more computational efficiency, the length of the vector `x` should be equal to an exponent of 2, that is 64, 128, 512, 1024, 2048, etc. The vector `x` can be padded with zeros to make it have an appropriate length. MATLAB does this automatically by using the following command where `N` is defined to be an exponent of 2:

```
X = fft(x, N);
```

The longer the length of `x`, the finer the grid will be for the FFT. Due to a wrap around effect, only the first $N/2$ points of the FFT have any meaning.

The `ifft` command computes the inverse Fourier transform:

```
x = ifft(X);
```

The FFT can be used to approximate the Fourier transform of a continuous-time signal as shown in Section 6.6 of the textbook. A continuous-time signal $x(t)$ is sampled with a period of T seconds, then the DFT is computed for the sampled signal. The resulting amplitude must be scaled and the corresponding frequency determined. An M-file that approximates the Fourier Transform of a sampled continuous-time signal is available from the ftp site and is given below:

```
function [X,w] = contfft(x,T);
[n,m] = size(x);
if n<m,
    x = x';
end
Xk = fft(x);
N = length(x);
n = 0:N-1;
n(1) = eps;
X = (1-exp(-j*2*pi*n/N))./(j*2*pi*n/N/T).*Xk.';
w = 2*pi*n/N/T;
```

The input is the sampled continuous-time signal x and the sampling time T . The outputs are the Fourier transform stored in the vector X and the corresponding frequency vector w .

3. Continuous Time System Analysis

A. Transfer Function Representation

Commands covered: `tf`
 `zpk`
 `tf2zp`
 `zp2tf`
 `feedback`
 `parallel`
 `series`

Transfer functions are created in MATLAB from the vectors that contain the coefficients of the numerator and the denominator. Given a continuous-time transfer function

$$H(s) = \frac{B(s)}{A(s)}$$

where $B(s) = b_M s^M + b_{M-1} s^{M-1} + \dots + b_0$ and $A(s) = s^N + a_{N-1} s^{N-1} + \dots + a_0$. Store the coefficients of $B(s)$ and $A(s)$ in the vectors `num = [bM bM-1 ... b0]` and `den = [1 aN-1 ... a0]`. In this text, the names of the vectors are generally chosen to be `num` and `den`, but any other name could be used. Then, type

```
sys = tf(num,den).
```

Where `sys` is the desired name of the system

For example,

$$H(s) = \frac{2s+3}{s^3+4s^2+5}$$

is defined by

```
num = [2 3];  
den = [1 4 0 5];  
H = tf(num,den);
```

Note that all coefficients must be included in the vector, even zero coefficients.

A transfer function may also be defined in terms of its zeros, poles and gain:

$$H(s) = \frac{k(s-z_1)(s-z_2)\dots(s-z_m)}{(s-p_1)(s-p_2)\dots(s-p_n)}$$

using the command

```
sys = zpk(z,p,k);
```

where z is a vector of the zeros, p is a vector containing the poles, and K is the gain. For example,

$$H(s) = \frac{2(s-4)(s+1)}{(s+20)(s+10)(s-5)}$$

```
z = [4 -1];
den = [-20 -10 5];
k = 2;
H = zpk(z,p,k);
```

To find the zeros, poles and gain of a transfer function from the vectors `num` and `den` which contain the coefficients of the numerator and denominator polynomials, type

```
[z,p,k] = tf2zp(num,den)
```

The zeros are stored in `z`, the poles are stored in `p`, and the gain is stored in `k`. To find the numerator and denominator polynomials from `z`, `p`, and `k`, type

```
[num,den] = zp2tf(z,p,k)
```

The overall transfer function of individual systems in parallel, series or feedback can be found using MATLAB. Consider block diagram reduction of the different configurations shown in Figure 1. The transfer functions are stored in `G` and `H`.

To reduce the general feedback system to a single transfer function, $G_{cl}(s) = G(s)/(1+G(s)H(s))$ type

```
G_cl = feedback(G,H);
```

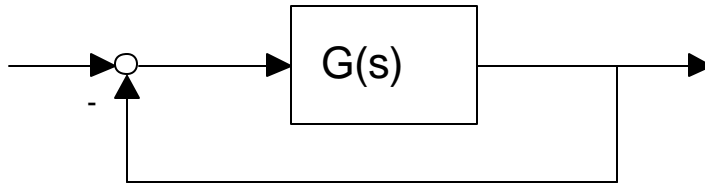
Where `G` and `H` are user-defined transfer functions. The default is negative feedback. For a unity feedback system, define `H` as `H=tf(1,1)`. To reduce the series system to a single transfer function, $G_s(s) = G(s)H(s)$ type

```
G_s = series(G,H);
```

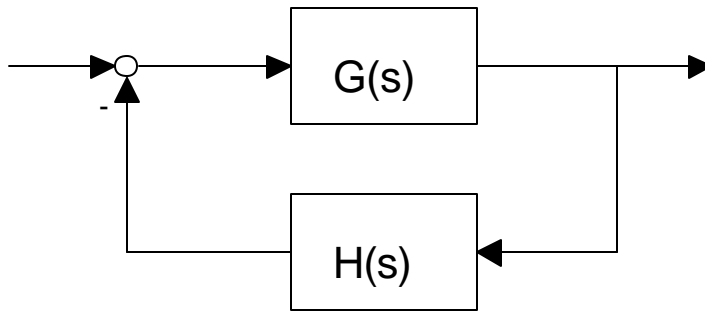
To reduce the parallel system to a single transfer function, $G_p(s) = G(s) + H(s)$ type

```
G_p = parallel(G,H);
```

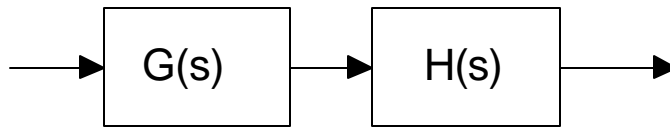
(Parallel is not available in the Student Version.)



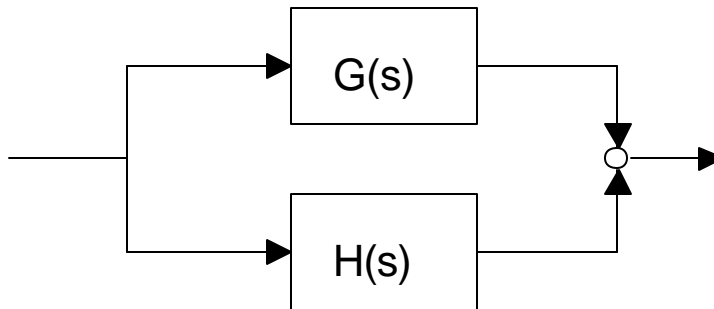
unity feedback



feedback



series



parallel

B. Time Simulations

Commands covered: `residue`
 `step`
 `impulse`
 `lsim`

The analytical method to find the time response of a system requires taking the inverse Laplace Transform of the output $Y(s)$. MATLAB aides in this process by computing the partial fraction expansion of $Y(s)$ using the command `residue`. Store the numerator and denominator coefficients of $Y(s)$ in `num` and `den`, then type

```
[r,p,k] = residue(num,den)
```

The residues are stored in `r`, the corresponding poles are stored in `p`, and the gain is stored in `k`. Once the partial fraction expansion is known, an analytical expression for $y(t)$ can be computed by hand.

A numerical method to find the response of a system to a particular input is available in MATLAB. To plot the step response, of a transfer function defined as H , type

```
step(H)
```

To plot the impulse response, type

```
impulse(H)
```

For the response to an arbitrary input, use the command `lsim`. Create a vector `t` which contains the time values in seconds at which you want MATLAB to calculate the response. Typically, this is done by entering

```
t = a:b:c;
```

where `a` is the starting time, `b` is the time step and `c` is the end time. For smooth plots, choose `b` so that there are at least 300 elements in `t` (increase as necessary). Define the input `x` as a function of time, for example, a ramp is defined as $x = t$. Then plot the response by typing

```
lsim(H,x,t);
```

To customize the commands, the time vector can be defined explicitly and the step response can be saved to a vector. Simulating the response for five to six time constants generally is sufficient to show

the behavior of the system. For a stable system, a time constant is calculated as $1/\text{Re}(-p)$ where p is the pole that has the largest real part (i.e., is closest to the origin).

For example, consider a transfer function defined by

$$H(s) = \frac{2}{s+2}$$

The step response y is calculated and plotted from the following commands:

```
num = 2; den = [1 2];  
H = tf(num,den);  
t = 0:3/300:3; % for a time constant of 1/2  
y = step(H,t);  
plot(t,y)
```

For the impulse response, simply replace the word `step` with `impulse`. For the response to an arbitrary input stored in `x`, type

```
y = lsim(H,x,t);  
plot(t,y)
```


C. Frequency Response Plots

Commands covered:

```
freqs
bode
logspace
log10
semilogx
unwrap
```

To compute the frequency response $H(\omega)$ of a transfer function, store the numerator and denominator of the transfer function in the vectors `num` and `den`. Define a vector `w` that contains the frequencies for which $H(\omega)$ is to be computed, for example `w = a:b:c` where `a` is the lowest frequency, `c` is the highest frequency and `b` is the increment in frequency. The command

```
Hw = freqs(num,den,w)
```

returns a complex vector `H` that contains the value of $H(\omega)$ for each frequency in `w`.

To draw a Bode plot of a transfer function that has been stored in the system `H`, type

```
bode(H)
```

For example,

```
numH = [2 1 3];
denH = [1 2 4 0];
H = tf(numH,denH);
bode(H);
```

where `numH` and `denH` hold the numerator and denominator of the transfer function `H`.

To customize the plot, first define the vector `w` that contains the frequencies at which the Bode plot will be calculated. Since `w` should be defined on a log scale, the command `logspace` is used. For example, to make a Bode plot ranging in frequencies from 10^{-1} to 10^2 , define `w` by

```
w = logspace(-1,2);
```

The magnitude and phase information for the Bode plot can then be found by executing:

```
[mag,phase] = bode(H,w);
```

Where the phase is in degrees. To plot the magnitude in decibels, convert `mag` using the following command:

```
magdb = 20*log10(mag);
```

To plot the results on a semilog scale where the y-axis is linear and the x-axis is logarithmic, type

```
semilogx(w, magdb)
```

for the log-magnitude plot and type

```
semilogx(w, phase)
```

for the phase plot. The phase plot may contain jumps of $\pm 2\pi$ which may not be desired. To remove these jumps, use the command `unwrap` prior to plotting the phase.

```
semilogx(w, unwrap(phase))
```

D. Analog Filter Design

Commands covered:

- `buttap`
- `cheblap`
- `zp2tf`
- `lp2lp`
- `lp2bp`
- `lp2hp`
- `lp2bs`

MATLAB contains commands for various analog filter designs, including those for designing a Butterworth filter and a Type I Chebyshev filter. The commands `buttap` and `cheblap` are used to design lowpass Butterworth and Type I Chebyshev filters, respectively, with cutoff frequencies of 1 rad/sec. For an n-pole Butterworth filter, type

```
[z, p, k] = buttap(n)
```

where the zeros of the filter are stored in `z`, the poles are stored in `p` and the gain of the filter is in `k`. For an n-pole Type I Chebyshev filter with R_p decibels of ripple in the passband, type

```
[z, p, k] = cheblap(n, Rp)
```

To find the numerator and denominator polynomials of the resulting filter from `z`, `p` and `k`, type

```
[b, a] = zp2tf(z, p, k)
```

where a contains the denominator coefficients and b contains the numerator coefficients. Frequency transformations from one lowpass filter to another with a different cutoff frequency, or from lowpass to highpass, or lowpass to bandstop or lowpass to bandpass can be performed in MATLAB. These transformations can be used with either the Butterworth filters or the Chebyshev filters. Suppose b and a store the numerator and denominator of a transfer function of a lowpass filter with cutoff frequency 1 rad/sec. To map to a lowpass filter with cutoff frequency ω_0 and numerator and denominator coefficients stored in b_1 and a_1 , type

$$[b_1, a_1] = lp2lp(b, a, \omega_0)$$

To map to a highpass filter with cutoff frequency ω_0 , type

$$[b_1, a_1] = lp2hp(b, a, \omega_0)$$

To map to a bandpass filter with bandwidth B_w centered at the frequency ω_0 , type

$$[b_1, a_1] = lp2bp(b, a, \omega_0, B_w)$$

To map to a bandstop filter with stopband bandwidth B_w centered about the frequency ω_0 , type

$$[b_1, a_1] = lp2bs(b, a, \omega_0, B_w)$$

E. Control Design

Commands covered: `rlocus`

Consider a feedback loop as shown in Figure 1 where $G(s)H(s) = KP(s)$ and K is a gain and $P(s)$ is a transfer function that contains the poles and zeros of the controller and of the plant. The root locus is a plot of the roots of the closed loop transfer function as the gain is varied. Suppose that the numerator and denominator coefficients of $P(s)$ are used to define the transfer function P . Then the following command computes and plots the root locus:

$$\text{Rlocus}(P)$$

To customize the plot for a specific range of K , say for K ranging from 0 to 100, then use the following commands:

```
K = 0:100;
r = rlocus(P,K);
plot(r, '.')
```

The graph contains dots at points in the complex plane that are closed loop poles for integer values of K ranging from 0 to 100. To get a finer grid of points, use a smaller increment when defining K , for

example, $K = 0:0.5:100$. The resulting matrix r contains the closed poles for all of the gains defined in the vector K . This is particularly useful to calculate the closed loop poles for one particular value of K . Note that if the root locus lies entirely on the real axis, then using `plot(r, '.')` gives inaccurate results.

F. State Space Representation

Commands Covered: `ss`
 `step`
 `lsim`
 `ss2tf`
 `tf2ss`
 `ss2ss`

The standard state space representation is used in MATLAB, i.e.,

$$\begin{aligned}\dot{x} &= Ax + Bu \\ y &= Cx + Du\end{aligned}$$

where x is $n \times 1$ vector, u is $m \times 1$, y is $p \times 1$, A is $n \times n$, B is $n \times m$, C is $p \times n$ and D is $p \times m$. The system is defined from the system matrices as

```
sys = ss(A,B,C,D);
```

The response of a system to various inputs can be found using the same commands that are used for transfer function representations: `step`, `impz`, and `lsim`. For example, the step response is obtained by typing:

```
[y,x,t] = step(sys);
```

The states are stored in x , the outputs in y and the time vector, which is automatically generated, is stored in t . The rows of x and y contain the states and outputs for the time points in t . Each column of x represents a state. For example, to plot the second state versus time, type

```
plot(t,x(:,2))
```

To find the response of an arbitrary input or to find the response to initial conditions, use `lsim`. Define a time vector t and an input matrix u with the same number of rows as in t and the number of columns equaling the number of inputs. An optional argument is the initial condition vector x_0 . The command is then given as

```
[y,x] = lsim(sys,u,t,x0);
```

You can find the transfer function H for a single-input/single-output (SISO) state space system using the command:

```
H = tf(sys);
```

Given a transformation matrix P , the `ss2ss` function will perform the similarity transform. Store the original state space model in `sys1` and the transformation matrix in `P`.

```
sys2 = ss2ss(sys1,P);
```

performs the similarity transform $z=Px$ resulting in a state space system that is defined as:

$$\begin{aligned}\dot{x} &= \bar{A}x + \bar{B}u \\ y &= \bar{C}x + \bar{D}u\end{aligned}$$

where $\bar{A} = PAP^{-1}$, $\bar{B} = PB$, $\bar{C} = CP^{-1}$, $\bar{D} = D$.

4. Discrete-Time System Analysis

A. Convolution

Commands covered: `conv`
 `deconv`

To perform discrete time convolution, $x[n]*h[n]$, define the vectors x and h with elements in the sequences $x[n]$ and $h[n]$. Then use the command

```
y = conv(x,h)
```

This command assumes that the first element in x and the first element in h correspond to $n=0$, so that the first element in the resulting output vector corresponds to $n=0$. If this is not the case, then the output vector will be computed correctly, but the index will have to be adjusted. For example,

```
x = [1 1 1 1 1];  
h = [0 1 2 3];  
y = conv(x,h);
```

yields $y = [0 1 3 6 6 6 5 3]$. If x is indexed as described above, then $y[0] = 0$, $y[1] = 1$, In general, total up the index of the first element in h and the index of the first element in x , this is the index of the first element in y . For example, if the first element in h corresponds to $n = -2$ and the first element in x corresponds to $n = -3$, then the first element in y corresponds to $n = -5$.

Care must be taken when computing the convolution of infinite duration signals. If the vector x has length q and the vector h has length r , then you must truncate the vector y to have length $\min(q,r)$.

The command `conv` can also be used to multiply polynomials: suppose that the coefficients of $a(s)$ are given in the vector a and the coefficients of $b(s)$ are given in the vector b , then the coefficients of the polynomial $a(s)b(s)$ can be found as the elements of the vector defined by $ab = \text{conv}(a,b)$.

The command `deconv` is the inverse procedure to the convolution. In this text, it is used as a means of dividing polynomials. Given $a(s)$ and $b(s)$ with coefficients stored in a and b , then the coefficients of $c(s) = b(s)/a(s)$ are found by using the command $c = \text{deconv}(b,a)$.

B. Transfer Function Representation

Commands Covered: `tf`

For a discrete-time transfer function, the coefficients are stored in descending powers of z or ascending powers of z^{-1} . For example,

$$H(z) = \frac{2z^2 + 3z + 4}{z^2 + 5z + 6} = \frac{2 + 3z^{-1} + 4z^{-2}}{1 + 5z^{-1} + 6z^{-2}} 1$$

then define the vectors as

```
num = [ 2 3 4 ];
den = [ 1 5 6 ];
```

Create the transfer function by typing

```
H = tf(num,den,Ts);
```

where Ts is the sampling time. If Ts = -1, the sampling time is undetermined.

C. Time Simulations

Commands Covered: `recur`
 `conv`
 `dstep`
 `dimpulse`
 `filter`

There are three methods to compute the response of a system described by the following recursive relationship

$$y[n] + \sum_{i=1}^N a_i y[n-i] = \sum_{i=0}^M b_i x[n-i]$$

The first method uses the command `recur` and is useful when there are nonzero initial conditions. This command is available from the book web site and a shortened version is given in Figure 2.5 of the textbook. The inputs to the function are the coefficients a_i and b_i stored in the vectors $a = [a_1 \ a_2 \ \dots \ a_N]$ and $b = [b_0 \ b_1 \ \dots \ b_M]$, the initial conditions on x and on y are stored in the vectors $x0 = [x[n_0-M], \ x[n_0-M+1], \ \dots, \ x[n_0-1]]$ and $y0 = [y[n_0-N], \ y[n_0-N+1], \ \dots, \ y[n_0-1]]$, and the time indices for which the solution needs to be calculated are stored in the vector n where n_0 represents the first element in this vector. To use `recur`, type

```
y = recur(a,b,n,x,x0,y0);
```

The output is a vector y with elements $y[n]$; the first element of y corresponds to the time index n_0 . For example, consider the system described by

$$y[n] - 0.6y[n-1] + 0.08y[n-2] = x[n-1]$$

where $x[n] = u[n]$ and with initial conditions $y[-1] = 2$, $y[-2] = 1$, and $x[-1] = x[-2] = 0$. To compute the response $y[n]$ for $n = 0, 1, \dots, 10$, type

```
a = [-0.6 0.08]; b = [0 1];
x0 = 0; y0 = [1 2];
n = 0:10;
x = ones(1,11);
y = recur(a,b,n,x,x0,y0);
```

The vector `y` contains the values of $y[n]$ for $n = 0, 1, \dots, 10$.

The second method to compute the response uses convolution and is useful when the initial conditions on y are zero. This method involves first finding the impulse response of the system, $h[n]$, and then convolving $h[n]$ with $x[n]$ as discussed in Section 4.A. For example, consider the system described above with zero initial conditions, that is, $y[-1]=y[-2]=0$. The impulse response for this system is $h[n] = 5[(0.4)^n - (0.2)^n]u[n]$. The commands to compute $y[n]$ are

```
n = 0:10;
x = ones(1,11);
h = 5*(0.4).^n - 5*(.02).^n;
y = conv(x,h);
y = y(1:length(n));
```

The vector `y` contains the values of $y[n]$ for $n = 0, 1, \dots, 10$. Note that the vector was truncated to `length(n)` because both $x[n]$ and $h[n]$ are infinite duration signals. See the comments in Section 4.A regarding the convolution of infinite duration signals.

The third method of solving for the response requires that the transfer function of the system be known. The commands `dstep` and `dimpulse` compute the unit step response and the unit impulse response, respectively while the command `filter` computes the response to initial conditions and to arbitrary inputs. The denominator coefficients are stored as $den = [1 \ a_1 \ a_2 \ \dots \ a_N]$ and the numerator coefficients are stored as $num = [b_0 \ b_1 \ \dots \ b_M, \ 0 \ \dots \ 0]$ where there are $N-M$ zeros padded on the end of the coefficients. For example, consider the system given above with initial conditions $y[-1] = y[-2] = 0$. To compute the step response for $n=0$ to $n=10$, type the commands

```
n = 0:10;
num = [0 1 0]; den = [1 -0.6 0.08];
y = dstep(num,den,length(n));
```

The response can then be plotted using the `stem` plot. To compute the impulse response, simply replace `dstep` with `dimpulse` in the above commands.

To compute the response to an arbitrary input, store the input sequence in the vector `x`. The command


```
y = filter(num,den,x);
```

is used to compute the system response. If the system has nonzero initial conditions, the initial conditions can be stored in a vector `v0`. For a first order system where $N=M=1$, define `zi = [b1*x[-1]-a*y[-1]]`. For a second order system where $N=M=2$, define `zi = [b1*x[-1]+b2*x[-2]-a1*y[-1]-a2*y[-2], b1*x[-1]-a2*y[-1]]`. To compute the response with nonzero initial conditions, type

```
y = filter(num,den,x,zi);
```

For example, consider the previous system with the initial conditions $y[-1] = 2$ and $y[-2] = 1$ and input $x[n] = u[n]$. Type the following commands to compute $y[n]$.

```
n = 0:10; x = ones(1,11);
num = [0 1 0]; den = [1 -0.6 0.08];
zi = [0.6*2-0.08*1, -0.08*2];
y = filter(num,den,x,zi);
```

D. Frequency Response Plots

Commands covered: `freqz`

The DTFT of a system can be calculated from the transfer function using `freqz`. Define the numerator and the denominator of the transfer function in `num` and `den`. The command

```
[H,Omega] = freqz(num,den,n,'whole');
```

computes the DTFT for n points equally spaced around the unit circle at the frequencies contained in the vector `Omega`. The magnitude of H is found from `abs(H)` and the phase of H is found from `angle(H)`. To customize the range for Ω , define a vector `Omega` of desired frequencies, for example `Omega = -pi:2*pi/300:pi` defines a vector of length 301 with values that range from $-\pi$ to π . To get the DTFT at these frequencies, type

```
H = freqz(num,den,Omega);
```

E. Digital Filter Design

Commands covered: `bilinear`
`butter`
`cheby1`
`hamming`
`hanning`

The analog prototype method of designing IIR filters can be done by first designing an analog filter with the desired characteristics as shown in Section 3.D, then mapping the filter to the discrete-time domain. Store the numerator and denominator of the analog filter, $H(s)$, in the vectors `num` and `den`, and let T be the sampling period. Then the numerator and denominator of the digital filter $H_d(z)$ is found from the following command

```
[numd,dend] = bilinear(num,den,1/T)
```

Alternately, the commands `butter` and `cheby1` automatically design the analog filter and then use the bilinear transformation to map the filter to the discrete-time domain. Lowpass, highpass, bandstop, and bandpass filters can be designed using this method. The digital cutoff frequencies must be specified; these should be normalized by π . To design a digital lowpass filter based on the analog Butterworth filter, use the commands:

```
[num,den] = butter(n,Omegac)
```

where `n` is the number of poles and `Omegac` is the normalized digital cutoff frequency, $\Omega_c = \omega_c T/\pi$. To design a highpass filter with cutoff frequency `Omegac`, use the commands

```
[num,den] = butter(n,Omegac,'high')
```

To design a bandpass filter with passband from `Omega1` to `Omega2`, define `Omega = [Omega1, Omega2]` and use the command

```
[num,den] = butter(n,Omega)
```

To design a bandstop filter with stopband from `Omega1` to `Omega2`, define `Omega = [Omega1, Omega2]` and use the command

```
[num,den] = butter(n,Omega,'stop')
```

The design for an n^{th} order Type I Chebyshev filter is accomplished using the same methods as for `butter` except that "butter" is replaced by "cheby1":

```
[num,den] = cheby1(n,Omegac); % for a lowpass filter
[num,den] = cheby1(n,Omegac,'high'); % for a highpass
filter
```

If `Omega` has two elements,

```
[num,den] = cheby1(n,Omega); % for a bandpass
[num,den] = cheby1(n,Omega,'stop'); % for a bandstop
```

The windows used in FIR filter design are given by

```
w = boxcar(N)      % rectangular window
w = hamming(N)
w = hanning(N)
```

These commands are used to truncate the infinite impulse response of an ideal digital filter with the result being an FIR filter with length N .

The Signal Processing Toolbox also provides commands for computing the FIR filter directly. To obtain an FIR filter with length N and cutoff frequency Ω_{c} (normalized by π) use the command

```
hd = fir1(N-1, Omega_c)
```

The vector `hd` contains the impulse response of the FIR where `hd(1)` is the value of $h_d[0]$. A length N highpass filter with normalized cutoff frequency Ω_{c} is designed by using the command

```
hd = fir1(N-1, Omega_c, 'high')
```

A bandpass with passband from Ω_{c1} to Ω_{c2} is obtained by typing

```
hd = fir1(N-1, Omega)
```

where $\Omega = [\Omega_{c1}, \Omega_{c2}]$. A bandstop filter with stopband from Ω_{c1} to Ω_{c2} is obtained by typing

```
hd = fir1(N-1, Omega, 'stop')
```

where $\Omega = [\Omega_{c1}, \Omega_{c2}]$. The `fir1` command uses the Hamming window by default. Other windows are obtained by adding an option of 'hanning' or 'boxcar' to the arguments; for example,

```
hd = fir1(N-1, Omega_c, 'high', boxcar(N))
```

creates a highpass FIR filter with cutoff frequency Ω_{c} using a rectangular window.

F. Digital Control Design

Commands covered: `bilinear`
 `c2dm`
 `hybrid`

An analog controller $G_c(s)$ can be mapped to a digital controller $G_d(z)$ using the bilinear transformation or the step response matching method. Store the numerator and denominator of $G_c(s)$ in `num` and

den. Then the numerator and denominator of $G_t(z)$ is found from the bilinear transformation using the commands

```
[numd,dend] = bilinear(num,den,1/T)
```

where T is the sampling frequency. To use the step invariant method, use the commands

```
[numd,dend] = c2dm(num,den,T,'zoh')
```

To simulate the response of a continuous-time plant with a digital controller, use the command `hybrid`, which is available at the book web site. Consider the block diagram in Figure 10.27. The numerator and denominator coefficients of the plant are stored in `NGp` and `DGp`; the numerator and denominator coefficients of the controller are stored in `NGd` and `DGd`; the reference input signal is stored in `r`; and the sampling time is stored in `T`. The increments in the time vector should be selected to be the sampling time divided by an integer, for example, `t = 0:b:Tend` where there is some integer m such that $bm=T$. The command is used as

```
[y,ud] = hybrid(NGp,DGp,NGd,DGd,T,t,r);
```

The outputs of the command are the system response, y , and the control signal that is input to the plant, ud . The M-file contains a loop which computes the discrete-time control and then simulates the continuous-time plant for T seconds with the constant control. The process repeats for the next T second interval. The commands for `hybrid` are given below:

```
function [Y,UD] = hybrid(Np,Dp,Nd,Dd,T,t,U);
[Ac,Bc,Cc,Dc]=tf2ss(Np,Dp);
[Ad,Bd,Cd,Dd]=tf2ss(Nd,Dd);
nsam = T/(t(2)-t(1)); % # of integration pts per
sample

% initialize
Y = 0;
UD = 0;
[ncr,ncc] = size(Ac);
xc0 = zeros(ncr,1);
[ndr,ndc] = size(Ad);
xdk = zeros(ndr,1);
kmax = fix(t(length(t))/T); % # of complete samples in t

for k = 0:kmax-1
    % calculate control and output of zoh
    ek = U(k*nsam+1) - Y(k*nsam+1);
    xd = Ad*xdk + Bd*ek;
    zoh = Cd*xdk + Dd*ek;
```

```

xdk = xd;
% integrate continuous-time plant with input
%   of zoh for T seconds
udi = zoh*ones(nsam+1,1);
ti = t(k*nsam+1:(k+1)*nsam+1);
[yi,xi] = lsim(Ac,Bc,Cc,Dc,udi,ti,xc0);
xc0 = xi(nsam+1,:);
% augment vectors
Y = [Y;yi(2:nsam+1)];
UD = [UD;udi(2:nsam+1)];
end

if(kmax*nsam+1 < length(t))
% compute tail of simulation from t(kmax*nsam)
%   to t_end
k = kmax;
% calculate control and output of zoh
ek = U(k*nsam+1) - Y(k*nsam+1);
xd = Ad*xdk + Bd*ek;
zoh = Cd*xdk + Dd*ek;
% integrate continuous-time plant with input of zoh
ti = t(k*nsam+1:length(t));
udi = zoh*ones(length(ti),1);
[yi,xi] = lsim(Ac,Bc,Cc,Dc,udi,ti,xc0);
% augment vectors
Y = [Y;yi(2:length(yi))];
UD = [UD;udi(2:length(udi))];
end

```

G. State Space Representation

Commands Covered: dlsim
 dstep
 dimpulse

Most of the commands for the continuous time state space representation also work for the discrete time state space. For example, `ss2tf`, `tf2ss`, and `ss2ss` for discrete time are used exactly the same way as for the continuous time case discussed in Section 3.F. There is a discrete time version of the command `lsim`, which is used as follows:

$$[y,x] = \text{dlsim}(A,B,C,D,u,n);$$

where the output is stored in `y`, the states are stored in `x`, the input is stored in `u` and the time index is stored in `n`.

5. Plotting

Commands covered:

```
plot
xlabel
ylabel
title
grid
axis
stem
subplot
```

The command most often used for plotting is `plot`, which creates linear plots of vectors and matrices; `plot(t,y)` plots the vector `t` on the x-axis versus vector `y` on the y-axis. There are options on the line type and the color of the plot which are obtained using `plot(t,y,'option')`. The linetype options are '-' solid line (default), '--' dashed line, '-.' dot dash line, ':' dotted line. The points in `y` can be left unconnected and delineated by a variety of symbols: + . * o x. The following colors are available options:

```
r    red
b    blue
g    green
w    white
k    black
```

For example, `plot(t,y,'--')` uses a dashed line, `plot(t,y,'*')` uses * at all the points defined in `t` and `y` without connecting the points, and `plot(t,y,'g')` uses a solid green line. The options can also be used together, for example, `plot(t,y,'g:')` plots a dotted green line.

To plot two or more graphs on the same set of axes, use the command `plot(t1,y1,t2,y2)`, which plots `y1` versus `t1` and `y2` versus `t2`.

To label your axes and give the plot a title, type

```
xlabel('time (sec)')
ylabel('step response')
title('My Plot')
```

Finally, add a grid to your plot to make it easier to read. Type

```
grid
```

The problem that you will encounter most often when plotting functions is that MATLAB will scale the axes in a way that is different than you want them to appear. You can easily override the autoscaling of the axes by using the `axis` command after the plotting command:

```
axis([xmin xmax ymin ymax]);
```

where `xmin`, `xmax`, `ymin`, and `ymax` are numbers corresponding to the limits you desire for the axes. To return to the automatic scaling, simply type `axis`.

For discrete-time signals, use the command `stem` which plots each point with a small open circle and a straight line. To plot $y[k]$ versus k , type

```
stem(k,y)
```

You can use `stem(k,y,'filled')` to get circles that are filled in. When using Version 3.0 of the Signal Processing Toolbox (or version 4.0 of the Student Version of MATLAB), the following must be done in order to get filled-in circles: The line in `stem.m`

```
h = plot(x,y,'o',xx(:),yy(:),linetype);
```

can be replaced with

```
h = plot(x,y,'.',xx(:),yy(:),linetype);  
set(h,'markersize',18);
```

to create closed circles.

To plot more than one graph on the screen, use the command `subplot(mnp)` which partitions the screen into an $m \times n$ grid where p determines the position of the particular graph counting the upper left corner as $p=1$. For example,

```
subplot(211),semilogx(w,magdb);  
subplot(212),semilogx(w,phase);
```

plots the bode plot with the log-magnitude plot on top and the phase plot below. Titles and labels can be inserted immediately after the appropriate `semilogx` command or `plot` command. To return to a full screen plot, type `subplot(111)`.

6. Loading and Saving Data

When using MATLAB, you may wish to leave the program but save the vectors and matrices you have defined. To save the file to the working directory, type

```
save filename
```

where "filename" is a name of your choice. To retrieve the data later, type

```
load filename
```