Chapter-III

Introduction:

In this chapter we shall consider some fundamental concepts of linear systems analysis and use the power of MATLAB to undertake analysis.

Basic discrete-time sequences:

1. The delta sequence:

The delta sequence plays an important role in the characterization of discrete-time linear time-invariant systems. The delta sequence, written as $\delta[n]$, is defined as

$$\delta[n] = \begin{cases} 1, & n = 0 \\ 0, & n \neq 0 \end{cases}$$

Practice:

```
>>n=-30:30;                    %specify index n
>>delta=(n= =0);               %define the delta sequence
>>stem(n,delta, 'filled')      %plot the delta sequence
```

2. The unit-step sequence:

The unit-step sequence, written as $u[n]$ is defined as

$$u[n] = \begin{cases} 1, & n \geq 0 \\ 0, & n < 0 \end{cases}$$

Practice:

```
>>n=-30:30;                    %specify index n
>>u_step=(n>=0);               %define the unit step sequence
>>stem(n, u_step, 'filled')    %plot the unit step sequence
```

Practice:

Provide a MATLAB code to sketch the discrete-time sequence $x[n]$ specified by

$$x[n] = 2\delta[n] + 3\delta[n-1] - 5\delta[n-3]$$

```
>>n=-30:30;                    %specify index n
```

```
>>xn=2*(n= =0)+3*((n-1)>=0)-5*((n-3)>=0);   %define the sequence x[n]
>>stem(n,xn; 'filled');grid                 %plot the sequence x[n]
```

3. The ramp sequence:

The ramp sequence, r[n], is defined as follows:

$$r[n] = \begin{cases} n, & n \geq 0 \\ 0, & n < 0 \end{cases}$$

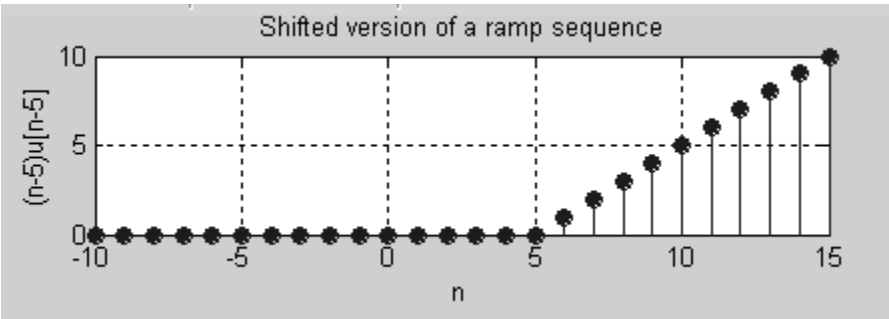
Practice:

```
>>n=-10:10;                                 %define index n
>>ramp=n.*(n>=0);                          %define a ramp
>>stem(n,ramp,'filled')                    %plot ramp
```

Practice:

Generate and plot a shifted version of a ramp sequence, r[n-5]

```
>>n=-5:15;                                  %define index n
>>x=(n-5).*((n-5)>=0);                     %define shifted version of ramp
>>stem(n,x,'filled');grid                  %plot the shifted version of ramp
```



Practice:

Define and sketch the discrete-time exponential sequence given by

$$x[n] = (0.8)^n u[n]$$

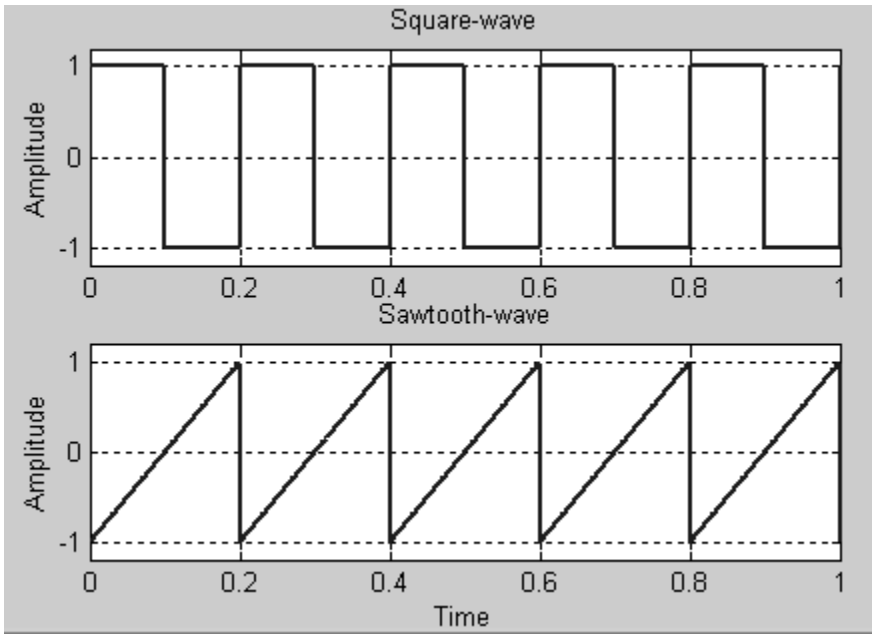
```
>>%exponential sequence
>>n=-30:30;                                 %specify index n
>>x=(0.8).^n.*(n>=0);                       %define the sequence x[n]
>>stem(n,x);grid                            %plot the exponential sequence
```

Square and sawtooth waves:

The MATLAB built-in functions **square** and **sawtooth** make it possible to generate a squarewave and sawtooth wave, respectively.

Practice:

```
t=(0:0.001:1);                          %time base
x=square(2*pi*5*t);                     %squarewave generator
subplot(2,1,1);plot(t,x,'LineWidth',2); grid   %plot squarewave
axis([0 1 –1.2 1.2]);                   %scale axes
title('Squarewave')                     %add title
ylabel('Amplitude');                    %label the vertical axis
%y=max(0,x);                            %squarewave ranging from 0 to 1
z=sawtooth(2*pi*5*t);                   %sawtooth wave
subplot(2,1,2); plot(t,z,'LineWidth',2);grid   %plot sawtooth wave
title('Sawtooth wave')                  %add title
ylabel('Amplitude')                     %label the vertical axis
xlabel('Time')                          %label the horizontal axis
axis([0 1 –1.2 1.2])                    %scale the axes
```

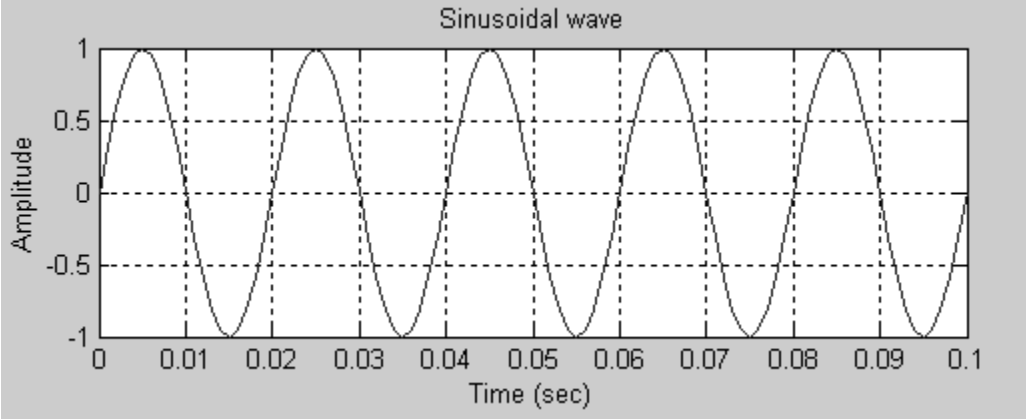


Practice:

Generate a 50 Hz sinusoidal signal.

```
>>Fs=1000;                  %sampling frequency
>>Ts=1/Fs;                  %sampling interval
>>t=0:Ts:0.1;               %sampling instants
>>x=sin(2*pi*50*t);         %signal vector
>>plot(t,x);grid            %plot the signal
>>xlabel('Time (sec)')      %add label to the horizontal axis
>>ylabel('Amplitude')       %add label to the vertical axis
```

>>title('Sinusoidal wave') %add title to the plot



Convolution:

- Convolution sum

The convolution sum of two sequences $x[n]$ and $h[n]$, written as $y[n]=x[n]*h[n]$, is defined by

$$y[n]=x[n]*h[n]=\sum_{k=-\infty}^{\infty}x[k]h[n-k]$$

MATLAB has a built-in function, **conv**, to perform convolution on finite-length sequences of values. This function assumes that the two sequences have been defined as vectors and it generates an output sequence that is also a vector. Convolving a sequence $x[n]$ of length N with a sequence $h[n]$ with length M results in a sequence of length L=N+M-1.

Syntax:

>>y=conv(x,h)

where x and h are finite sequences written in vector form

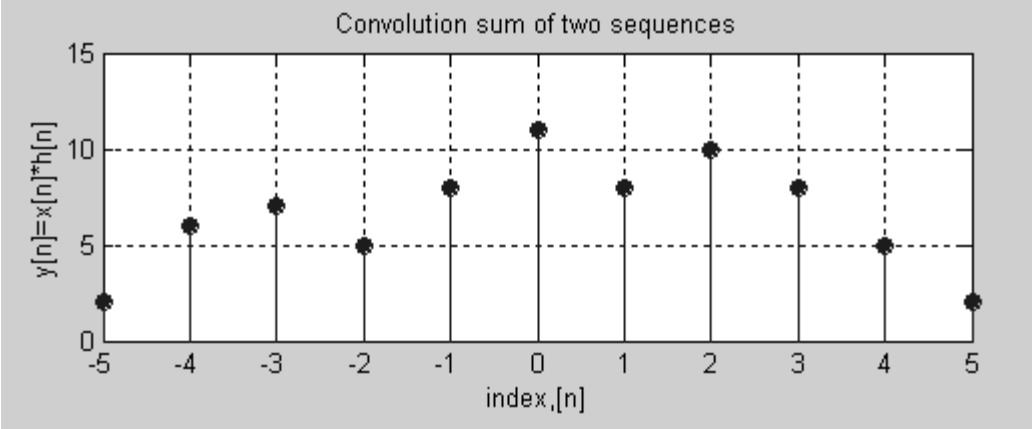
Practice:

Determine the convolution of the sequences $x[n]$ and $h[n]$ specified below.

```
>>x=[1  2  2  1  2]; nx=[-2:2];              %define sequence x[n] and its range
>>h=[2  2  -1  1  2  2  1]; nh=[-3:3];      %define sequence h[n] and its range
>>nmin=min(nx)+min(nh);                     %specify the lower bound of convolved sequences
>>nmax=max(nx)+max(nh);                     %specify the upper bound of convolved sequences
>>y=conv(x,h); n=[nmin:nmax];               %compute convolution and specify its range
>>stem(n,y,'filled'); grid                  %plot the resulting sequence y[n]
>>title('convolution of two sequences')     %add title to the plot
>>ylabel('y[n]=x[n]*h[n]')                  %label the y-axis
>>xlabel('index, [n]')                      %label the horizontal axis
```

>>[n' y'] <enter> % print index and sequence y[n] as column vectors

n	-5	-4	-3	-2	-1	0	1	2	3	4	5
y	2	6	7	5	8	11	8	10	8	5	2



- Numerical convolution:

The other form of convolution is known as the convolution integral. If $x(t)$ and $h(t)$ are two continuous-time signals, then the convolution integral is defined by

$$y(t) = x(t) * h(t) = \int_0^t x(\tau) h(t-\tau)\, d\tau$$

since computers have a hard time integrating, we shall consider evaluating $y(t)$ numerically. Let $t = kT$

$$\therefore y(kT) = \int_0^{kT} x(\tau) h(kT-\tau)\, d\tau \cong T\sum_{i=0}^{k} x(iT)\, h\big((k-i)T\big) = Tx(kT) * h(kT)$$

where T is the integration step size. This equation is the convolution sum and can be found using the **conv** function. Generally, the smaller the value of T, the higher the accuracy of the result.

Practice:

Evaluate the convolution of $x(t) = \sin(2t)$ with $h(t) = e^{-0.1t}$ from t=0 to 5 with T=0.1.

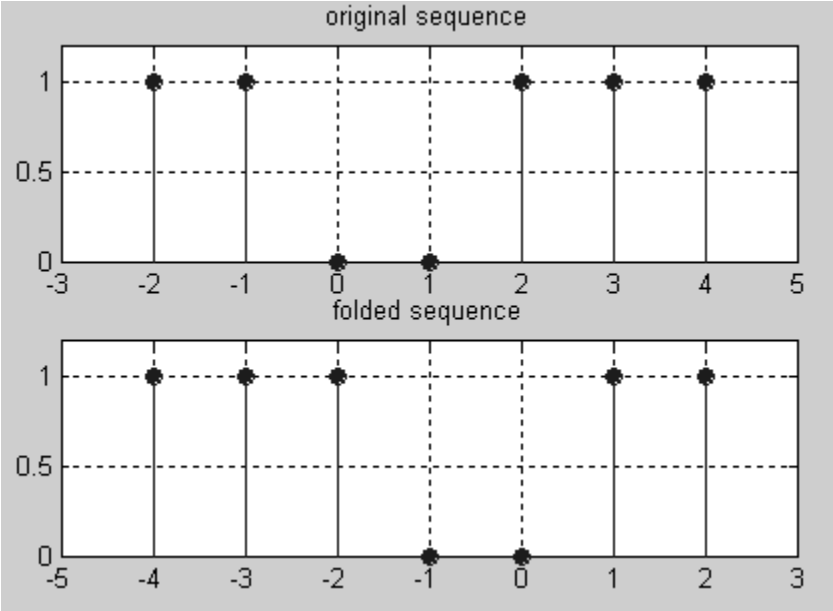
```
>>t=0:0.1:5;                %define t=kT
>>x=sin(2*t);               %define x(kt)
>>h=exp(-0.1*t);            %define h(kT)
>>0.1*conv(x,h);            %evaluate convolution
```

Sequence folding:

A number of signal processing operations involve taking the mirror image of a sequence about the vertical axis passing through the origin. The **flplr** function flips a sequence left to right. The following script illustrates the process of folding via MATLAB.

```
n=-2:4;
x=[1 1 0 0 1 1 1 ];
y=fliplr(x);
m=-fliplr(n);
subplot(2,1,1);stem(n,x,'filled');grid
title('original sequence')
axis([-3 5 0 1.2])
subplot(2,1,2);stem(m,y,'filled');grid
title('folded sequence')
axis([-5 3 0 1.2])
```

The following plots show the original sequence and its folded version.



Ordinary Differential Equations (ODE):

MATLAB has built-in routines for solving ordinary differential equations, namely, **ode23**, and **ode45**. The function **ode45** is more accurate, but a bit slower than **ode23**.

In order to solve a differential equation using MATLAB, the following items are required:

- You need to make a separate function file that contains the differential equation or system of differential equations.
- Specify the time range over which the solution is desired
- Specify the initial conditions

Practice:

Use MATLAB to solve the following ordinary differential equation

$$\begin{cases} \dfrac{dy}{dt} + 3ty = 0 \\ y(0) = 2 \end{cases}$$

- Create the function

```
function  ydot=yprime(t,y)
ydot=-3*t*y;
```

- Time range: we want the solution, say, over the interval [0 6]

- Initial conditions: initial=2

Typing the following at the command prompt will solve the ODE.

```
>>initial =3;
>>[t,y]=ode23('yprime', [0 6],initial)
>>plot(t,y,'LineWidth',2)
```

We shall now consider the solution of second-order differential equations. Higher order ordinary differential equations require conversion to two or more first order ordinary differential equations.

Practice:

Solve the following second order ordinary differential equation:

$$\begin{cases} \dfrac{d^2y}{dt^2} + 3t\dfrac{dy}{dt} + 7y = 0 \\ y(0) = 0, \qquad y'(0) = 1 \end{cases}$$

To convert to first-order differential equations, we proceed as follows:

$$y_1 = y$$
$$y_2 = y'$$
$$\therefore y_2' = -3ty_2 - 7y_1$$

This yields a system of two first-order differential equations

$$\begin{cases} y_1' = y_2 \\ y_2' = -3ty_2 - 7y_1 \end{cases}$$

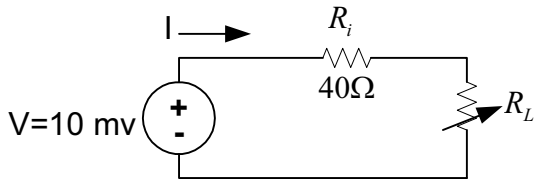
```
function  ydot= yprime(t,y)
ydot= [y(2); -7*y(1)-3*t*y(2)];
```

Typing the following at the command prompt will solve the ODE.

```
>>initial=[0; 1];
>>[t,y]= ode45('yprime',[0 6],initial)
>>plot(t,y,'LineWidth',3);grid
```

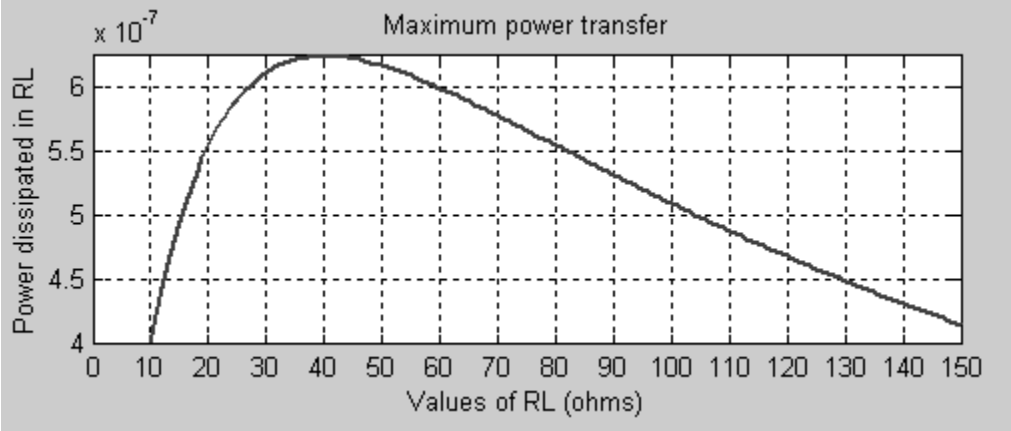
Practice:

For the voltage divider depicted below, sketch the power dissipated in RL as RL varies from 10 ohms to 150 ohms. Deduce the value of RL yielding a maximum power transfer.



$$I = \frac{V}{R_i + R_L} \Rightarrow P = R_L \left(\frac{V}{R_i + R_L} \right)^2 = V^2 \frac{R_L}{\left(R_i + R_L \right)^2}$$

```
>>close all; clear all; clc;
>>v=0.010;
>>RI=40;
>>RL=10:1:150;
for i=1:length(RL)
        P(i)=(V^2*RL(i))/(RI+RL(i))^2;
end
plot(RL,P,'LineWidth',2,'Color',[0.2 0.3 0.1]);grid
set(gca, 'Xtick',0:10:150);
axis([ 0 max(RL)  min(P)  max()])
xlabel('Values of RL (ohms)')
ylabel('Power dissipated in RL')
title('Maximum power transfer')
k=find(P= =max(P));
fprintf('\n');
disp([Value of RL for max. power transfer: ', num2str(RL(k)),'  ohms'])
fprint('\n');
```



Transfer function:

A transfer function $H(s)$ characterizes fully a linear time-invariant system. A transfer function can be entered into MATLAB in a number of ways:

- We can use the command **tf(num,den),** where **num** and **den** are vectors of coefficients of the numerator and denominator polynomials, respectively.

Practice:

$$H(s)=\frac{s+1}{s^2+5s+6}$$

```
>>num=[1 1];                    %specify the numerator of H(s)
>>den=[1 5 6];                  %specify the denominator of H(s)
>>sys=tf(num,den);              %specify the transfer function of the system
>>sys                           %print the transfer function on the screen
```

- An alternative approach is to use the command **zpk(zeros, poles, gain),** where **zeros**, **poles** and **gain** are vectors of zeros, poles and gain of the transfer function.

Practice:

$$H(s)=\frac{5(s+3)}{(s+4)(s+11)}$$

```
>>zeros=[-3];                   %specify the zeros of H(s)
>>poles=[-4 –11];               %specify the poles of H(s)
>>gain=5;                       %specify the gain
>>sys=zpk(zeros,poles,gain);    %specify the transfer function H(s)
>>sys                           %print the transfer function on the screen
```

Step response and impulse response:

- Continuous-time systems:

We shall now take a look at the step response of a linear time-invariant system (LTI) specified by its transfer function. The step response is simply the output of an LTI system driven by a unit step function. The command **step** provides the step response of continuous-time LTI systems.

Practice:

A linear time-invariant system is modeled by its transfer function given by

$$H(s) = \frac{\text{num(s)}}{\text{den(s)}} = \frac{s+1}{s^2+3s+3}$$

Find the step response.

```
>>%Step response of a continuous-time LTI system
>>num=[1 1];                 %specify the numerator of H(s)
>>den=[1 3 3];               %specify the denominator of H(s)
>>sys=tf(num,den)            %LTI system model created with transfer function (tf)
>>step(sys)                  %plot the step response of given system
```

Alternatively, a similar result can be obtained through the following steps:

```
>>%Step response of a continuous-time LTI system
>>num=[1 1];                 %specify the numerator of H(s)
>>den=[1 3 3];               %specify the denominator of H(s)
>>sys=tf(num,den);           %LTI system model
>>t=0:0.01:10;               %specify time vector
>>step(sys,t)                %plot the step response of given system
```

A third alternative involves the determination of the response before plotting

```
>>%Step response of a continuous-time LTI system
>>t=0:0.01:10;               %specify a time vector
>>sys=tf([1 1],[1 3 3]);     %specify model of system
>>[y, t]=step(sys,t);        %compute the step response
>>plot(t,y);grid;            %plot the step response
```

The inverse Lap lace transform of the transfer function H(s) is the impulse response of the system. The command **impulse** determines the impulse response of an LTI system.

Practice:

Determine the impulse response of a system governed by the transfer function given by

$$H(s) = \frac{\text{num(s)}}{\text{den(s)}} = \frac{s+1}{s^2+3s+3}$$

```
>>%Impulse response of a continuous-time LTI system
>>num=[1 1];                 %specify the numerator of H(s)
>>den=[1 3 3];               %specify the denominator of H(s)
```

```
>>sys=tf(num,den);          %specify model of system
>>impulse(sys)              %plot the impulse response of the system
```

Alternatively, the same result can be obtained through the following steps:

```
>>%Impulse response of a continuous-time LTI system
>>num=[1 1];                %specify the numerator of H(s)
>>den=[1 3 3];              %specify the denominator of H(s)
>>sys=tf(num,den);          %specify model of system
>>t=0:0.01:10;              %define a time vector
>>impulse(sys,t);           %plot the impulse response
```

A third alternative involves the determination of the impulse response before plotting the result.

```
>>%Impulse response of a continuous-time LTI system
>>t=0:0.01:10;              %specify time vector
>>sys=tf([1 1],[1 3 3]);    %model of the system
>>[y,t]=impulse(sys,t);     %compute the impulse response
>>plot(t,y)                 %plot the impulse response
```

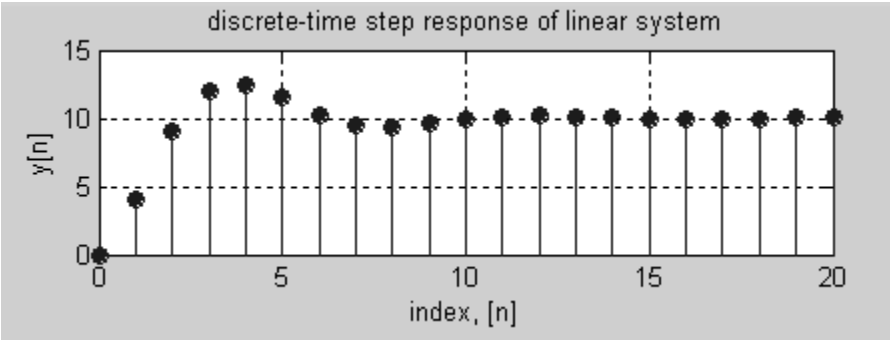
- Discrete-time systems

The discrete counterparts of the commands **step** and **impulse** are **dstep** and **dimpulse**. We shall illustrate these commands with few examples:

Practice:

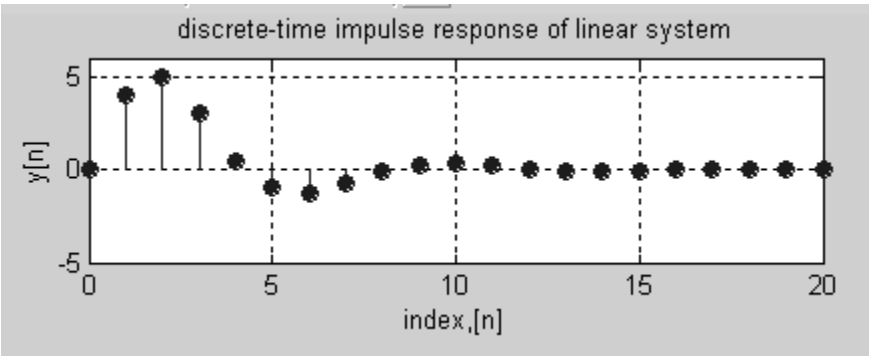
$$H(z)=\frac{4z+1}{z^2-z+0.5}$$

```
>>%Step response of a discrete-time linear system
>>n=0:20;                   %specify discrete-time vector
>>num=[4 1]                 %specify the numerator of H(z)
>>den=[1 –1 0.5];           %specify the denominator of H(z)
>>y=dstep(num,den,n);       %compute the step response
>>stem(n,y,'filled')        %sketch the step response
>>title('discrete-time step response of linear system')
>>xlabel('index,[n]')
>>ylabel('y[n]')
```



Practice:

```
%Impulse response of a discrete-time linear system
>>n=0:20;                    %specify discrete-time vector
>>num=[4 1];                 %specify the numerator of H(z)
>>den=[1 –1 0.5];            %specify the denominator of H(z)
>>y=dimpulse(num,den,n);     %compute the discrete-time impulse response
>>stem(n,y,'filled')         %plot impulse response
>>title('discrete-time impulse response of linear system')
>>ylabel('y[n]')
>>xlabel('index,[n]')
```



Pole-zero plots:

The command **pzmap** displays the poles and zeros of the continuous- or discrete-time linear system in the complex plane. The multiplicity of poles and zeros is not specified by **pzmap**. The poles are depicted as crosses (x's), and zeros by (0's).

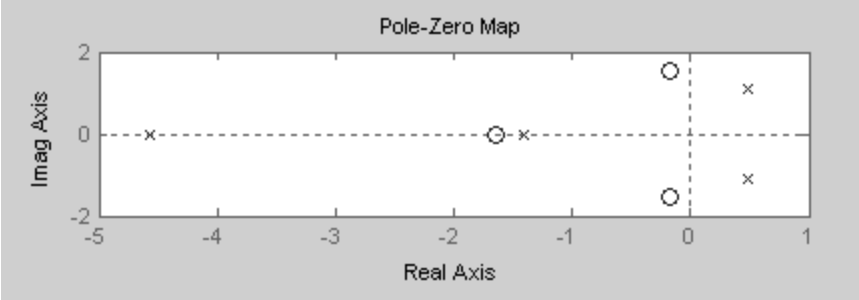
Practice:

Consider a linear time-invariant system modeled by the following transfer function:

$$H(s) = \frac{s^3 + 2s^2 + 3s + 4}{s^4 + 5s^3 + 2s^2 + 2s + 9}$$

Sketch the pole-zero plot of the system specified by H(s).

```
>>%Pole-zero plot of the system specified by H(s)
>>num=[1 2 3 4];             %specify the numerator of H(s)
>>den=[1 5 2 2 9];           %specify the denominator of H(s)
>>sys=tf(num,den);           %model of the system
>>pzmap(sys)                 %plot the pole-zero plot
>>[p,z]=pzmap(sys);          %return the system poles and zeros
```



The Z-Plane:

The command **zplane** displays the poles and zeros of a linear time-invariant system in the complex plane with a unit circle as a reference. The relative location of the poles and zeros, with respect to the unit circle, is an important aspect in the analysis and design of digital filters. Multiple order poles and zeros are indicated by the multiplicity number shown to the upper right of the zeros or poles.

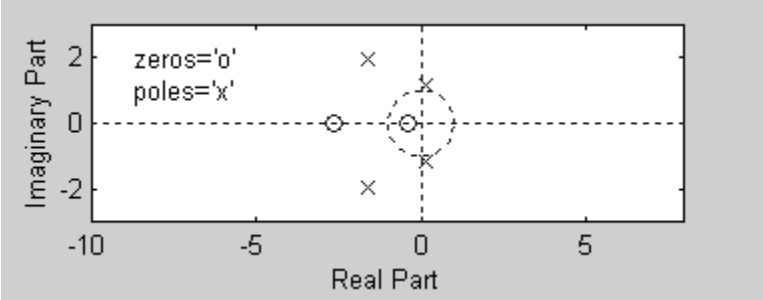
Practice:

A linear time-invariant discrete-time system is modeled by its transfer function given by

$$H(z)=\frac{z^2+3z+1}{z^4+3z^3+7z^2+3z+9}$$

Plot the zero-pole plot of the system.

```
>>num=[0 0 1 3 1];          %specify numerator of H(z)
>>den=[1 3 7 3 9];          %specify denominator of H(z)
>>zplane(num,den)           %plot the zero-pole plot
```



Practice:

An LTI system is described by the following difference equation:

$$y[n]+0.2y[n-1]+0.8y[n-2]=0.3x[n]+0.6x[n-1]+0.2x[n-2]$$

1. Find the poles and zeros of the system

2. Plot the pole-zero diagram
3. Is the system stable? (BIBO stability implies the convergence of the cumulative sum of the absolute value of the impulse response.)
4. Plot the magnitude and phase responses

```
>>b=[1 0.2 0.8];
>>a=[0.3 0.6 0.2];
>>zero=roots(b);
>>pole=roots(a);
>>zplane(b,a);
>>delta=[1 zeros(1,30)];
>>h=filter(b,a,delta);
>>cumsum(h)
>>N=512; Fs=8000;
>>freqz(b,a,N,Fs)
```

Stability design via Routh-Hurwitz:

It is possible to use MATLAB to find the gain, K, for the system so that it becomes stable. Following is an illustration of the design of gain of transfer function:

$$H(s) = \frac{K}{s^3 + 18s^2 + 77s + K}$$

```
>>K=1:2000;                  %specify the range of K
>>for m=1:length(K)          %number of K values to test
        denH=[1 18 77 K(m)]; %specify the denominator of H(s)
        poles=roots(denH);   %evaluate poles
        r=real(poles);       %vector containing real values
        if max(r)>=0         %test poles for real parts
        poles                %display poles with real part positive
        K=K(m)               %display the corresponding value of K
        break                %stop loop if rhp poles are found
        end                  %end if
end                          %end for
```

Partial-fraction expansion:

In the analysis of analog and discrete-time systems, one very useful method of solving the inverse Laplace transform and the inverse Z transform is to perform partial fraction expansion on the transfer function H(s) and H(z). MATLAB has built-in functions called **residue** and **residuez** that return the partial fraction expansions of the transfer function H(s) and H(z), respectively.

- ***Partial-fraction expansion on H(s)***
- simple poles:

$$H(s)=\frac{\text{num(s)}}{\text{den(s)}}=\frac{s^m+b_{m-1}s^{m-1}+...+b_1s+b_0}{s^n+a_{n-1}s^{n-1}+...+a_1s+a_0}=\frac{r(1)}{s-p(1)}+\frac{r(2)}{s-p(2)}+...+\frac{r(n)}{s-p(n)}+k(s)$$

where $p(k)$ are the poles of the transfer function, $r(k)$ are the coefficients of the partial fraction terms (called the residues of the poles) and k(s) is a remainder polynomial which is usually empty.

Syntax:

```
>>[r,p,k]=residue(num,den);          %perform partial fraction expansion of H(s)
>>[num,den]=residue(r,p,k);          %convert partial fraction back to polynomial form
```

vectors **num** and **den** specify the coefficients of the numerator and denominator polynomials in descending powers of s. The residues are stored in the column vector r, the corresponding pole locations in column vector p, and the remainder terms in row vector k.

Practice:

A linear time-invariant (LTI) system is governed by the transfer function H(s) given by

$$H(s)=\frac{s^2+2s+3}{s^3+5s^2+2s+7}$$

Use the residue command to obtain the partial fraction expansion of H(s)

```
>>%Partial-fraction expansion via MATLAB
>>num=[1 2 3];                      %specify numerator of H(s)
>>den=[1 5 2 7];                    %specify denominator of H(s)
>>[r,p,k]=residue(num,den)          %perform partial fraction expansion

r =

   0.6912
   0.1544 - 0.1645i
   0.1544 + 0.1645i


p =

  -4.8840
  -0.0580 + 1.1958i
  -0.0580 - 1.1958i


k =

     []
```

$$\therefore H(s)=\frac{0.6912}{s+4.8840}+\frac{0.1544-0.1645j}{s+0.0580-1.1958j}+\frac{0.1544+0.1645j}{s+0.0580+1.1958j}$$

- Multiple poles:

$$H(s)=\frac{\text{num(s)}}{\text{den(s)}}=\frac{r(1)}{s-p}+\frac{r(2)}{(s-p)^2}+\ldots+\frac{r(m)}{(s-p)^m}$$

Practice:

$$H(s)=\frac{8s+10}{(s+1)(s+2)^3}=\frac{B_1}{s+2}+\frac{B_2}{(s+2)^2}+\frac{B_3}{(s+2)^3}+\frac{A}{s+1}$$

```
>>num=[8 10];                    %specify numerator
>>den=con([1 1],[1 6 12 8]);     %specify denominator
>>[r,p,k]=residue(num,den)       %perform partial fraction expansion

r =

   -2.0000
   -2.0000
    6.0000
    2.0000


p =

   -2.0000
   -2.0000
   -2.0000
   -1.0000


k =

    []
```

$$\therefore H(s)=-\frac{2}{s+2}-\frac{2}{(s+2)^2}+\frac{6}{(s+2)^3}+\frac{2}{s+1}$$

- ***Partial fraction expansion on H(z)***
- Simple poles

$$H(z)=\frac{\text{num(z)}}{\text{den(z)}}=\frac{r(1)}{1-p(1)z^{-1}}+\frac{r(2)}{1-p(2)z^{-1}}+\ldots+\frac{r(n)}{1-p(n)z^{-1}}+k(1)+k(2)z^{-1}+\ldots$$

where **num** and **den** are the numerator and denominator polynomial coefficients, respectively, in ascending powers of z^{-1}, **r** and **p** are column vectors containing the residues and poles, respectively, **k** contains the direct terms in a row vector.

Syntax:

```
>>[r,p,k]=residuez(num,den);        %perform partial fraction expansion of H(z)
>>[num,den]=residuez(r,p,k);        %convert the partial fraction back to polynomial form
```

Practice:

A linear time-invariant system is governed by the transfer function H(z) given by

$$H(z)=\frac{6z^2-37z+53}{z^3+-10z^2+31z-30}=\frac{0+6z^{-1}-37z^{-2}+53z^{-3}}{1-10z^{-1}+31z^{-2}-30z^{-3}}$$

```
>>%Partial fraction expansion of H(z)
>>num=[0  6  -37  53];            %specify numerator of H(z)
>>den=[1 –10 31 –30];             %specify denominator of H(z)
>>[r,p,k]=residuez(num,den);      %perform partial fraction expansion of H(z)

r =

   3.0000
   2.0000
   1.0000


p =

   5.0000
   3.0000
   2.0000


k =

   []
```

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

- Multiple poles:

If p_1 is a multiple pole of order m, then H(z) has terms of the form:

$$H(z)=\frac{\text{num}(z)}{\text{den}(z)}=\frac{r(1)}{1-p_1z^{-1}}+\frac{r(2)}{\left(1-p_1z^{-1}\right)^2}+\ldots+\frac{r(m)}{\left(1-p_1z^{-1}\right)^m}$$

Practice:

Use the **residuez** command to find the partial fraction expansion of H(z).

$$H(z)=\frac{\text{num}(z)}{\text{den}(z)}=\frac{2+3z^{-1}+4z^{-2}}{1+3z^{-1}+3z^{-2}+z^{-3}}$$

```
>>num=[2  3  4];                 %specify the numerator of H(z)
>>den=[1  3  3  1];              %specify the denominator of H(z)
>>[r,p,k]=residuez(num,den)      %perform partial fraction expansion

r =

   4.0000 - 0.0000i
  -5.0001 + 0.0000i
   3.0000 - 0.0000i


p =

  -1.0000
  -1.0000 + 0.0000i
  -1.0000 - 0.0000i


k =

     []
```

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

Bode Plots:

If a system has a transfer function $H(s)$, its frequency response, $H(j\omega)$, is in general complex, and it's typically represented in two separate plots, one for its magnitude and the other for its phase, both as functions of frequency ω. These two plots together are called the Bode plot of the system. The command **bode** produces both the magnitude and the phase responses.

Syntax:

```
>>bode(sys)
```

Draws automatically the magnitude and phase plots of a system modeled by the transfer function H(s), specified by **sys**. The frequency is on a logarithmic scale, the phase is given in degrees, and the magnitude is given as the gain in decibels.

>>[mag,phase]=bode(sys,w)

This function returns the magnitude and phase of a system specified in terms of its transfer function H(s). The magnitude and phase are evaluated at a frequency vector w specified by the user. To obtain a plot with the magnitude expressed in decibels over a logarithmic frequency axis use semilogx(w,20*log10(mag)). Similarly, the phase (in degrees) over a logarithmic frequency axis is obtained via semilogx(w,phase). Finally, if it is desired to have a frequency axis in Hz, use w/(2*pi) instead of w.

>>[mag,phase,w]=bode(sys);

This function returns the magnitude, phase, and a vector w that contains the values of the frequency in rad/s where the frequency response has been calculated.

Note: *In order to plot the magnitude and phase you need to remove singleton dimensions from mag and phase. This is done using the* ***squeeze*** *function or* ***reshape*** *function. This is carried out as follows:*

```
>>[mag, phase]=bode(sys, w);
>>mag=squeeze(mag);
>>phase=squeeze(phase);
>>semilogx(w,20*log10(mag))
```

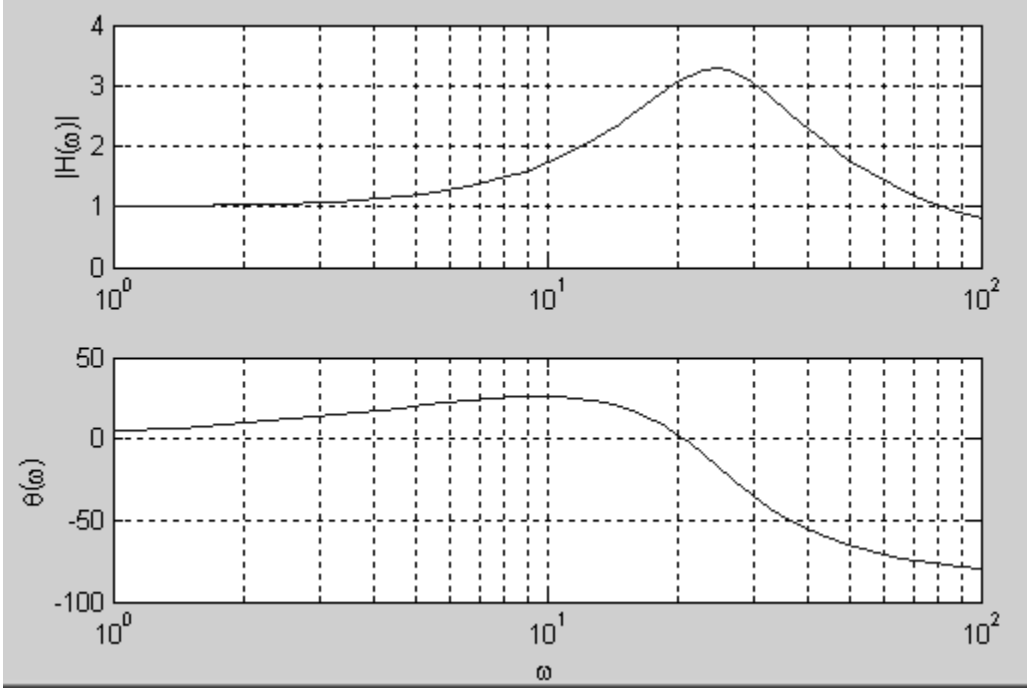
Practice:

Consider a linear time-invariant system governed by its transfer function given by

$$H(s)=\frac{\text{num}(\text{s})}{\text{den}(\text{s})}=\frac{78(s+8)}{s^2+25s+625}$$

Draw the bode plot.

```
>>%Bode plot with MATLAB
>>num=78*[1 8];                          %specify the numerator of H(s)
>>den=[1 25 625];                        %specify the denominator of H(s)
>>H=tf(num,den);                         %specify the transfer function model
>>[mag,phase,w]=bode(H)                  %compute magnitude and phase
>>%The output arguments of mag and phase are 3-D arrays. We need to alter the size
>>%of both via the reshape function, before plotting can be made possible.
>>subplot(2,1,1);semilogx(w,reshape(mag,length(w),1));grid
>>ylabel('|H(\omega)|')
>>subplot(2,1,2);semilogx(w,reshape(phase,length(w),1));grid
>>ylabel('\theta(\omega)')
>>xlabel('\omega')
```



Alternatively, the same results can be obtained as follows:

```
>>%Bode plot with MATLAB
>>num=78*[1 8];                    %specify the numerator of H(s)
>>den=[1 25 625];                  %specify the denominator of H(s)
>>H=tf(num,den);                   %model of the system
>>w=1:0.02:100;                    %specify a frequency vector
>>[mag,phase]=bode(H,w);           %compute magnitude and phase
>>mag=reshape(mag,size(w));        %reshape mag into a column vector
>>phase=reshape(phase,size(w));    %reshape phase into a column vector
>>subplot(2,1,1);semilogx(w,20*log10(mag));grid
>>subplot(2,1,2);semilogx(w,phase);grid
```

Practice:

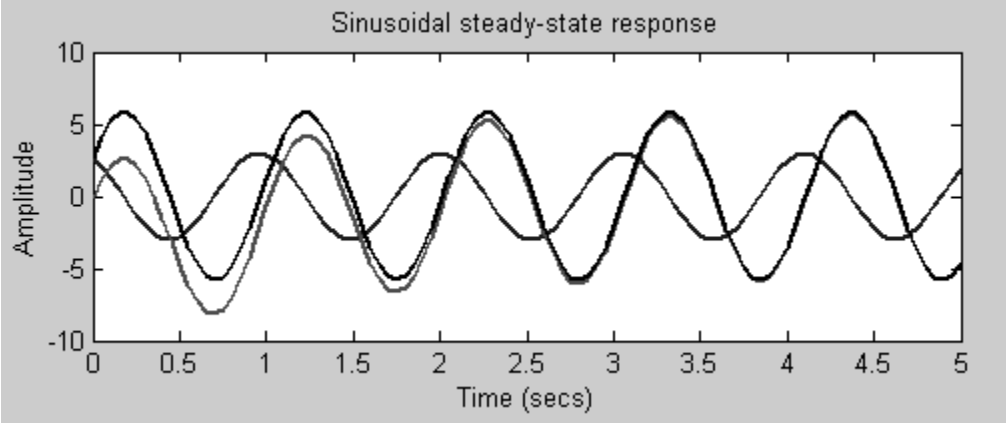
1. Simulate the response of the system

$$H(s)=\frac{10(s+6)}{s^2+5s+4}$$

to the sinusoidal input $x(t)=3\cos(6t+30^\circ)$ over the interval $0\le t\le 5$, assuming the system is in zero state.

2. Find the frequency response $H(j\omega)$ at ω=6 rad/s
3. Calculate the steady-state output, $y_{ss}(t)$
4. Sketch x(t), y(t) and yss(t) in a single plot

```
t=[0:0.05:5]';
x=3*cos(6*t+30*pi/180);
y=lsim(num,den,x,t);
[mag,phase]=bode(num,den,6);
yss=3*mag*cos(6*t+(30+phase)*pi/180);
subplot(2,1,1);plot(t,x,t,y,'r',t,yss,'k','LineWidth',2);
set(gca,'XTick',[0:0.5:5])
title('Sinusoidal steady-state response')
xlabel('Time (secs)')
ylabel('Amplitude')
```



Interconnection of systems:

Systems may be interconnected in various ways to form larger systems. Conversely, one might break down larger systems into smaller ones to facilitate design and analysis. There are three major ways in which systems can be interconnected, namely,

- Series interconnection (or cascade)
- Parallel interconnection
- Feedback interconnection (positive feedback or negative feedback)

The MATLAB commands, **series**, **parallel**, **feedback,** and **cloop,** provide the overall transfer function of systems that are connected in series, parallel, feedback, and a closed-loop (unity feedback) system description given an open-loop system, respectively.

Practice:

Two systems are governed by the following transfer functions:

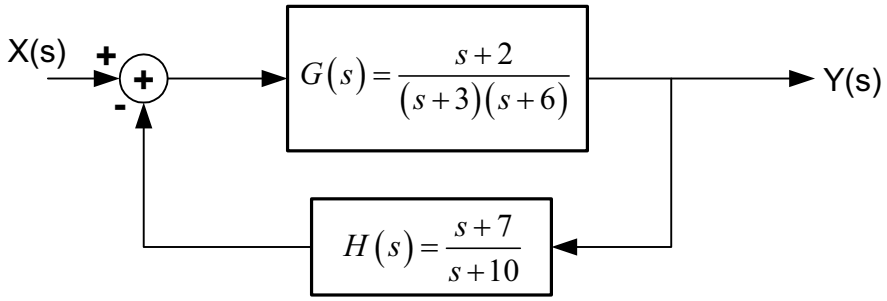
$$H_1(s) = \frac{s}{s^2 + 2s + 3}, \quad H_2(s) = \frac{s+3}{s^2 + 5s + 1}$$

1. Find the overall transfer function of the cascaded systems
2. Repeat part (1) if the systems are connected in parallel
3. Repeat part (1) for a feedback connection

```
>>%Systems connected in cascade
>>num1=[1 0];                  %specify numerator of system 1
>>den1=[1 2 3];                %specify denominator of system 1
>>sys1=tf(num1,den1);          %specify system 1
>>num2=[1 3];                  %specify numerator of system 2
>>den2=[1 5 1];                %specify denominator of system 2
>>sys2=tf(num2,den2);          %specify system 2
>>sys=series(sys1,sy2)         %specify the overall cascaded system

Transfer function:
        s^2 + 3 s
--------------------------------
s^4 + 7 s^3 + 14 s^2 + 17 s + 3


>>sys=parallel(sys1,sys2)      %specify overall system in parallel

Transfer function:
  2 s^3 + 10 s^2 + 10 s + 9
--------------------------------
s^4 + 7 s^3 + 14 s^2 + 17 s + 3


>>sign=-1                      %sign=-1 (negative feedback), sign=+1 (positive feedback)
>>sys=feedback(sys1,sys2,sign)%specify model for the closed-loop feedback system

Transfer function:
      s^3 + 5 s^2 + s
--------------------------------
s^4 + 7 s^3 + 15 s^2 + 20 s + 3
```

Practice:

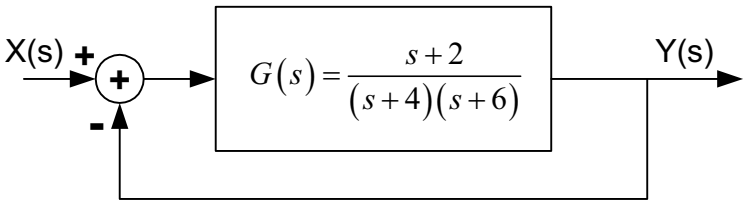
Find the overall transfer function of the system shown below.




```
>>numg=[1 2];
>>deng=conv([1 3], [1 6]);
>>numh=[1 7];
>>denh=[1 10];
>>[num, den]=feedback(numg,deng,numh,denh);
>>sys=tf(num,den)


Transfer function:
    s^2 + 12 s + 20
--------------------------
s^3 + 20 s^2 + 117 s + 194
```

Practice:

Find the transfer function of the feedback system depicted below




```
>>numg=[1 2];
>>deng=conv([1 4], [1 6]);
>>[num,den]=cloop(numg,deng);
>>sys=tf(num,den)

Transfer function:
    s + 2
---------------
s^2 + 11 s + 26
```

Frequency response plots:

- Continuous-time LTI models

The commands **freqs** and **freqz** provide the frequency response in the s-domain and frequency response in the z-domain, respectively. Several MATLAB commands are used in conjunction with frequency plots, namely, **abs**, **angle**, and **unwrap**. The **abs** and **angle** extract the magnitude and phase response, respectively, and **unwrap** removes jumps of size 2π prior to plotting the phase.

We begin with the frequency response of continuous-time linear time-invariant systems. The system function of a continuous-time system can be written in the form:

$$H(s)=\frac{\text{num(s)}}{\text{den(s)}}=\frac{s^m+b_{m-1}s^{m-1}+...+b_1s+b_0}{s^n+a_{n-1}s^{n-1}+...+a_1s+a_0}$$

Syntax:

>>[H,w]=freqs(num,den);

Given the transfer function, $H(s)$, specified by its numerator and denominator coefficients **num** and **den** in vector form. The function **freqs** returns the complex frequency response **H**, and a set of 200 frequencies **w** (in rad/s) where the frequency response has been calculated.

>>H=freqs(num,den,w);

This function returns the complex frequency response **H** of an LTI system specified in terms of its transfer function H(s). The complex frequency response is evaluated at a frequency vector **w** specified by the user.

>>freqs(num,den);

Draws automatically the magnitude (in dB) and phase (in degrees) plots of a system modeled by the transfer function H(s).

Practice:

Plot the s-domain frequency response of a system governed by the following transfer function:

$$H(s)=\frac{10}{s^2+11s+10}$$

```
>>%The transfer function must be entered as vectors of descending powers of s
>>num=10;                                    %specify numerator of H(s)
>>den=[1 11 10];                            %specify denominator of H(s)
>>w=-15:0.05:15;                            %specify a frequency vector
>>H=freqs(num,den,w);                       %provide the frequency response
>>mag=abs(H);                               %compute the magnitude response
>>phase=angle(H)*180/pi;                    %compute the phase response in degrees
>>line_3dB=0.707*ones(size(w))*max(mag);    %draw the 3-dB line
>>subplot(2,1,1);plot(w,mag,w,line_3dB,'ro'); %plot magnitude response and 3-dB line
```

>>subplot(2,1,2);plot(w,phase); %plot the phase response

- Discrete-time LTI models:

We shall now consider the frequency response of discrete-time systems, which is also known as the discrete-time Fourier transform (DTFT). In general, the transfer function of a discrete-time system can be written in the form,

$$H(z) = \frac{\text{num}(\text{z})}{\text{den}(\text{z})} = \frac{b_1 + b_2 z^{-1} + ... + b_{nb+1} z^{-nb}}{a_1 + a_2 z^{-1} + ... + a_{na+1} z^{-na}}$$

Syntax:

>>[H,w]=freqz(num,den,n,'whole');

This function returns the complex frequency response **H**, and a frequency vector **w** (in radians per sample) containing the **n** frequency points spaced around the **whole** unit circle ($0 \le \omega < 2\pi$). If n is not specified, it defaults to 512.

>>[H,w]=freqz(num,den,n);

Evaluates the complex frequency response **H** at **n** equally spaced points around the upper half of the unit circle. The frequency vector w has values ranging from 0 to π radians per sample. When you don't specify the integer n, or you specify it as the empty vector [ ], the frequency response is computed using the default value of 512 samples.

>>H=freqz(num,den,w);

Returns the frequency response vector H calculated at the frequencies (in radians per sample) supplied by the vector w. The vector w can have an arbitrary length.

>>freqz(num,den);

Returns the magnitude response (in dB) and the **unwrapped** phase response (in degrees) against the normalized frequency from 0 to 1.

>>[H,f]=freqz(num,den,n, Fs);

Returns the frequency response vector **H** and the corresponding frequency vector **f** for the digital filter whose transfer function is determined by the (real or complex) numerator and denominator polynomials represented in the vector **num** and **den**, respectively. The frequency vector **f** is computed in units of Hz. The frequency vector f has values ranging from 0 to Fs/2.

>>[H,f]=freqz(num,den,n,'whole',Fs);

Uses n points around the whole unit circle (from 0 to 2π , or from 0 to Fs), to calculate the frequency response. The frequency vector **f** has length n and has values ranging from 0 to Fs Hz.

>>H=freqz(num,den,f,Fs);

Returns the frequency response vector **H** calculated at the frequencies (in Hz) supplied in the vector **f**. The vector **f** can have an arbitrary length.

>>[h,f,units]=freqz(num,den,n,'whole',Fs);

Returns the optional string argument **units**, specifying the units for the frequency vector f. The string returned in **units** is Hz.

Fourier synthesis:

Fourier series are often used to model periodic signals. By truncating the Fourier series, signals can be approximated accurately enough for applications. The computation and study of Fourier series is known as harmonic analysis. The process of expanding a periodic signal in Fourier series is termed **analysis**, while the process of reconstructing a waveform from its Fourier series is termed Fourier **synthesis**. It is well known that around a discontinuity of a function, the partial sum of the Fourier series exhibit oscillatory behavior (ringing) known as the **Gibbs phenomenon**. The **Gibbs phenomenon** is caused by the difficulty to represent sharp discontinuities with smooth trigonometric functions.

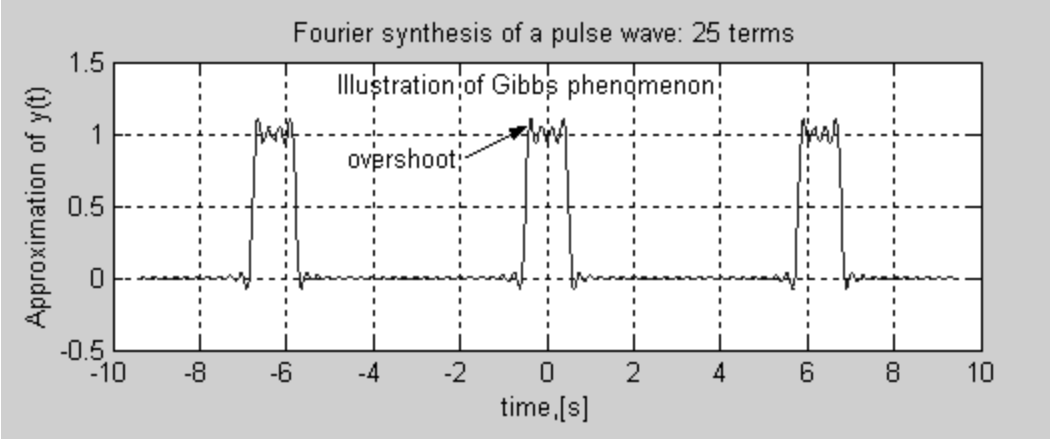
Practice:

The Fourier series of a pulse waveform is given by

$$y(t) = \frac{1}{2\pi} + \sum_{n=1}^{\infty} \frac{2}{n\pi} \sin\left(\tfrac{1}{2}n\right)\cos(nt)$$

Write a script file to synthesize an approximation of the pulse wave. Experiment with the number of terms in the partial sum and describe your observations as the number of terms increases. The error caused by using only a finite number of terms (truncated series) is called the truncation error.

```
t=-3*pi:0.02:3*pi;                                  %specify time span
N=input('Enter the number of terms: ')              %input the number of terms
x=zeros(size(t));                                   %initialization
for n=1:N                                           %time index
x=x+(2/(n*pi))*sin(0.5*n)*cos(n*t);                 %form the partial sum
end                                                 %end of loop
y=x+1/(2*pi);                                       %add dc component
plot(t,y);grid                                      %plot the partial sum
s=int2str(N);                                       %convert N into a string
title(['Fourier synthesis of a pulse wave: ',s,' terms'])    %add title to the resulting plot
xlabel('time,[s]')                                  %label the horizontal axis
xlabel('Approximation of y(t)')                     %label the vertical axis
```

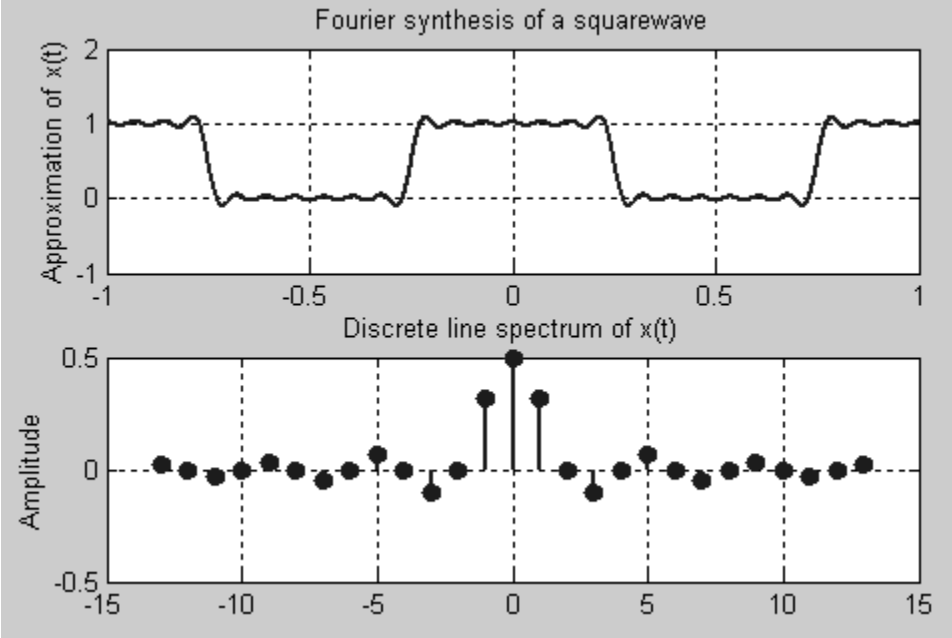


Practice:

Write a script file to synthesize a square wave whose complex Fourier series coefficients are given by

$$x(t) = \sum_{n=-\infty}^{\infty} c_n e^{jn\frac{2\pi}{T}t} \rightarrow c_n = \frac{1}{2}\sin\left(\frac{n}{2}\right)$$

```
N=input('Enter the number of terms: ');
t=[-1:0.001:1];
n=-N:N;
cn=1/2*sinc(n/2);
B=exp(j*2*pi*t.'*n);
C=ones(length(t),1)*cn;
P=B.*C;
xtrunc=(sum(P.')).';
subplot(2,1,1);plot(t,real(xtrunc),'LineWidth',2);grid
title('Fourier synthesis of a squarewave')
ylabel('Approximation of x(t)')
subplot(2,1,2);h=stem(n,cn,'filled');grid
set(h,'LineWidth',2);
title('Discrete line spectrum of x(t)')
ylabel('Amplitude spectrum')
```



The Discrete Time Fourier Transform:

The discrete time Fourier transform of a sequence x[n] is defined by

$$X\left(e^{j\omega}\right) = X\left(\omega\right) = \sum_{n=-\infty}^{\infty} x\left[x\right] e^{-j\omega n}$$

Practice:

1. Generate the following signal

$$x\left[n\right] = 1 + \cos\left(\frac{25\pi n}{100}\right), \qquad 0 \le n \le 99$$

2. Compute the DTFT of x[n] for $\omega = 0:0.01:2*\pi$
3. Plot the real part, the imaginary part, the amplitude and phase of $X\left(\omega\right)$

```
>>x=1+cos(25*pi/100)*(0:99));                %generate signal
>>w=0:0.01:2*pi;                            %frequency vector
>>X=exp(-j*w(:)*(0:99))*x(:);               %dtft
>>figure(1);
>>plot(w,real(X),w, imag(x));               %plot real and imaginary components
>>xlabel('\omega');
>>title('Re(\itX\rm(\omega)), Im(\it\X\rm(\omega))')
>>figure(2);
>>plot(w,abs(X));                           %plot magnitude
>>figure(3);
>>plot(w, angle(X));                        %plot phase
```

The Discrete Fourier Transform (DFT):

In a variety of applications we want to interpret data as signals of different frequencies. One way to accomplish this task is through the Discrete Fourier Transform (DFT). The DFT is a well-known tool to analyze the spectral content of a time series. The DFT is a very computationally intensive procedure. In 1965, **Cooley and Tukey** published a numerical algorithm to evaluate the DFT with a significant reduction in the amount of calculation required. The **Fast Fourier Transform**, or **FFT**, is a general name for a class of algorithms that allow the Discrete Fourier Transform of a sampled signal to be obtained rapidly and efficiently. Keep in mind that the FFT is *not a new transform*, but a computationally efficient method to compute the DFT. A common use of the FFT is to find the frequency components buried in a noisy time domain signal. The **fft** command of MATLAB allows the computation of the Discrete Fourier Transform (DFT) of an N element vector x representing N samples of a discrete time signal. The function **ifft** defines the inverse discrete Fourier transform. For any x[n], the value **ifft(fft(x))** equals x to within roundoff errors.

Syntax:

```
>>X=fft(x);                %compute the DFT of x
>>x=ifft(X);               %compute the inverse DFT of X
```

note*: For the fastest possible FFTs, you will want to pad your data with enough zeros to make its length a power of two. The* ***fft*** *function does this automatically if you provide a second argument specifying the overall length of the FFT as follows:*

```
>>X=fft(x,N);               %compute an N-point DFT
```

The next highest power of 2 greater than or equal to the length of the given sequence x[n] can be found using the MATLAB command **nextpow2**.

Practice:

Find the next highest power of 2 of 235-point sequence x[n]

```
>> N=235;                  %length of the sequence x[n]
>> NFFT=2^nextpow2(N)      %make the length a power of 2

NFFT =

  256
```

The result of **fft** is generally an array of complex numbers. The commands **abs** and **angle** can be used to compute the magnitude and angle of the complex values respectively.
For plotting a two-sided transform, the command **fftshift** will rearrange the output of **fft** to move the zero component to the center of the spectrum, as the following code demonstrates.

Practice:

```
>>x=[1  2  3  4];           %define a vector x
```

```
>>y=fftshift(x)                    %swap the first and second halves of x

y =

    3     4     1     2
```

Practice:

Find the 4-point DFT of the sequence x[0]=1, x[1]=2, x[2]=3, x[3]=4, and x[n]=0 for all other n.

```
>>n=0:3;                          %index
>>xn=[1 2 3 4];                   %sequence x[n]
>>Xk=fft(xn)                      %compute the FFT of x[n]

10.0000
 -2.0000 - 2.0000i
 -2.0000
 -2.0000 + 2.0000i
```

Practice:

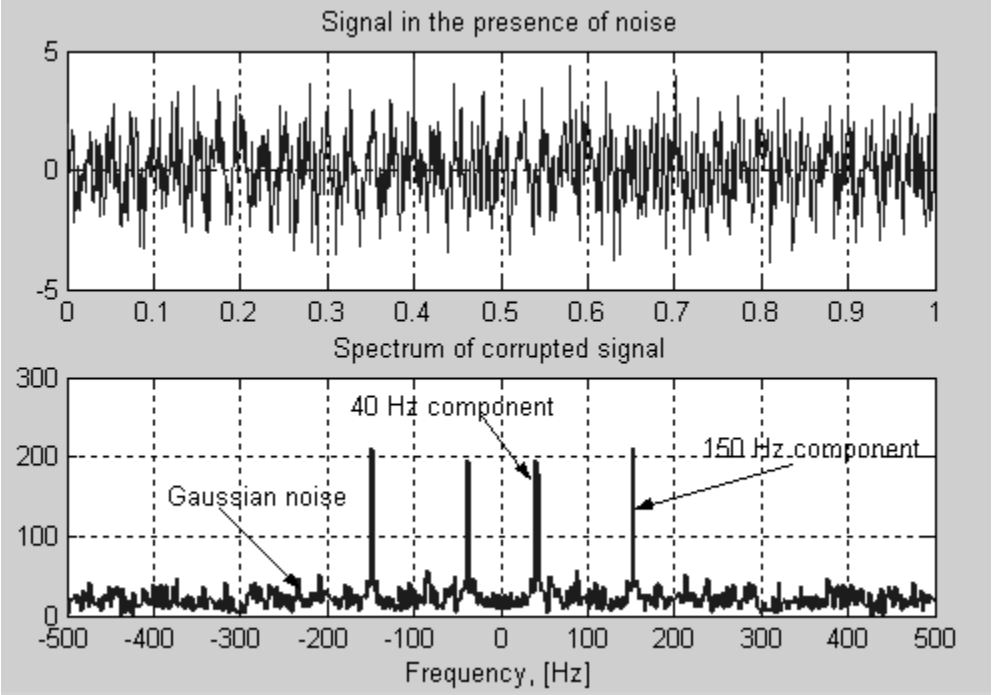
We form a signal containing 40 Hz and 150 Hz frequency components and corrupt it with gaussian noise having zero mean and unity standard deviation.

1. Sketch the corrupted signal in the time-domain
2. Determine the spectrum of the corrupted signal and sketch it

```
>>Fs=1000;                                %specify the sampling frequency
>>t=0:1/Fs:1;                             %define a time base
>>sig=cos(2*pi*40*t)+cos(2*pi*150*t);%signal component
>>noise=randn(size(sig));                 %noise component
>>signal=sig+noise;                       %corrupted signal
>>subplot(2,1,1);plot(t,signal);grid      %plot corrupted signal
>>title('Signal in the presence of noise')%add title
>>spec=fft(signal,512);                   %compute the spectrum of corrupted signal
>>mag=fftshift(abs(spec));                %center the spectrum around dc
>>f=Fs*(-255:256)/512;                    %frequency vector
>>subplot(2,1,2);plot(f,mag);grid         %plot spectrum
>>title('Spectrum of corrupted signal')   %add title
>>xlabel('Frequency,[Hz]')                %label the horizontal axis
```

Note:

The DFT results in an evaluation of the spectrum at evenly spaced frequencies. For an N-sample record and sampling frequency Fs, the frequency resolution (spacing) is $\Delta f = F_s / N$.



Obviously, it is difficult to identify the frequency components by looking at the original signal in the time domain. On the other hand, the identification of the spectral components is straightforward in the frequency domain.

Timing:

The functions **tic** and **toc** provide a means of finding the time taken to execute a segment of code. The statement **tic** starts the timing and **toc** gives the elapsed time since the last **tic**. The timing will vary depending on the model of computer being used and its current load.

Practice:

```
>> x=[1 2 3 4 5 6 7 8];
>> tic, fft(x),toc

elapsed_time =

   0.0900
```

***Note**: A sampled record is of finite length and is therefore usually a truncated version of the actual signal. In essence it is obtained by multiplying the signal by a rectangular pulse of height 1 over the time occupied by the record.*

Practice:

Obtain the amplitude spectrum of the signal $x(t) = \sin(2\pi f_1 t) + 0.06\sin(2\pi f_2 t)$ using 64 samples 125 μs apart; apply a Hamming window to the data set.

```
Ts=0.000125;                                  %sampling interval
F=[1062.5  1625];                             %frequency tones of x(t)
t=(0:63)*Ts;                                  %time base
x=sin(2*pi*F(1)*t)+0.06*sin(2*pi*F(2)*t);     %signal
wham=hamming(64);                             %hamming window
xham=wham'.*x;                                %windowed signal
subplot(2,1,1);stem(t,xham)                   %plot windowed signal
xlabel(Time (seconds)'); ylabel('xham(n)')    %label axes
f=(0:63)/(64*Ts);                             %frequency vector
subplot(2,1,2); stem(f,abs(fft(xham)));       %plot spectrum
xlabel('Frequency (Hz)'); ylabel('FFTmag');   %label axes
```

Zero-padding:

Zero padding refers to the operation of extending a sequence of length M to length N>M by appending N-M zero samples to the given sequence. This operation may be performed on any sequence x[n], prior to computing its transform with a DFT.

Practice:

```
>>x=[1  2  3  4  5];                  %sequence x[n]
>>xp=[1  2  3  4  5  0  0  0];        %padded sequence
```

The Discrete-Time Fourier Series(DTFS):

The DTFS is a frequency domain representation for periodic discrete-time sequences.
The built-in function **fft** may be used to evaluate the DTFS. If x is an N-point vector containing x[n] for the single period $0 \le n \le N-1$, then the DTFS of x[n] can be computed by

```
>>X=fft(x)/N
```

where the N-point vector X contains the DTFS coefficients, X[k] for $0 \le k \le N-1$. MATLAB assumes the summations in the DTFS formula run from 0 to N-1, so the first elements of x and X correspond to x[0] and X[0], respectively, while the last elements correspond to x[N-1] and X[N-1]. Similarly, given DTFS coefficients in a vector X, the command **ifft** may be used to recover the original sequence x[n].

```
>>x=ifft(X)*N
```

returns a vector x that represents one period for the time vector.

Practice:

Determine the DTFS coefficients for the following signal.

$$x[n] = 1 + \sin\left(\frac{\pi}{12}n + \frac{3\pi}{8}\right); \qquad N = 24 \text{ (period)}$$

```
>>x=ones(1,24)+sin([0:23]*pi/12+3*pi/8);      %specify the sequence x[n]
```

```
>>X=fft(x)/24                    %compute the DTFS of x[n]
```

Circular convolution:

Practice:

```
%-----------------------------------------
%Assume x and y are real data signals
%whose length is less than N
%-----------------------------------------
N=128;              %length of circular convolution
X=fft(x,N);         %DFT of x[n]
Y=fft(y,N);         %DFT of y[n]
z=real(ifft(X.*Y)); %N-point circular convolution of x[n] and y[n]
%-----------------------------------------
```

DTMF (multi-tone dialing procedure):

Dual-tone-multi-frequency (DTMF) also known as touch-tone is the standard in the telecommunication systems used in encoding digits for transmission between the customer and the central office. In the DTMF scheme, a telephone is equipped with a keypad as depicted in Figure 1. The A, B, C, and D keys are usually not present on a regular telephone keypad. In DTMF there are 16 distinct tones. Each tone consists of two simultaneous frequencies mixed together (added amplitudes), for example, pressing '1' will send a tone made of 1209 Hz and 697 Hz to the other end of the line. The frequencies were selected to avoid harmonics (no frequency is a multiple of another, the difference between any two frequencies does not equal any of the frequencies, likewise for the sum of two frequencies). At the central office, the telephone switch decodes the tones. Decoding can be done using a bank of eight bandpass filters, one for each of the eight possible tones. If two of these tones are detected, it is assumed that the key associated with both of these tones was pressed. DTMF "Touch" tones are defined in CCITT volume VI.

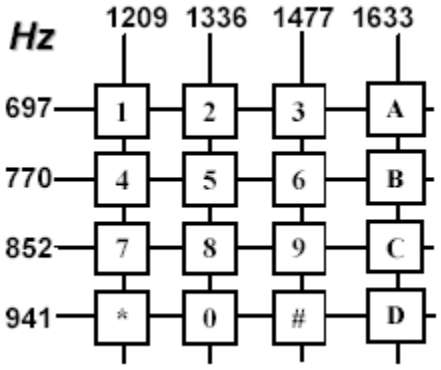


Figure 1: Touch-Tone Telephone Pad

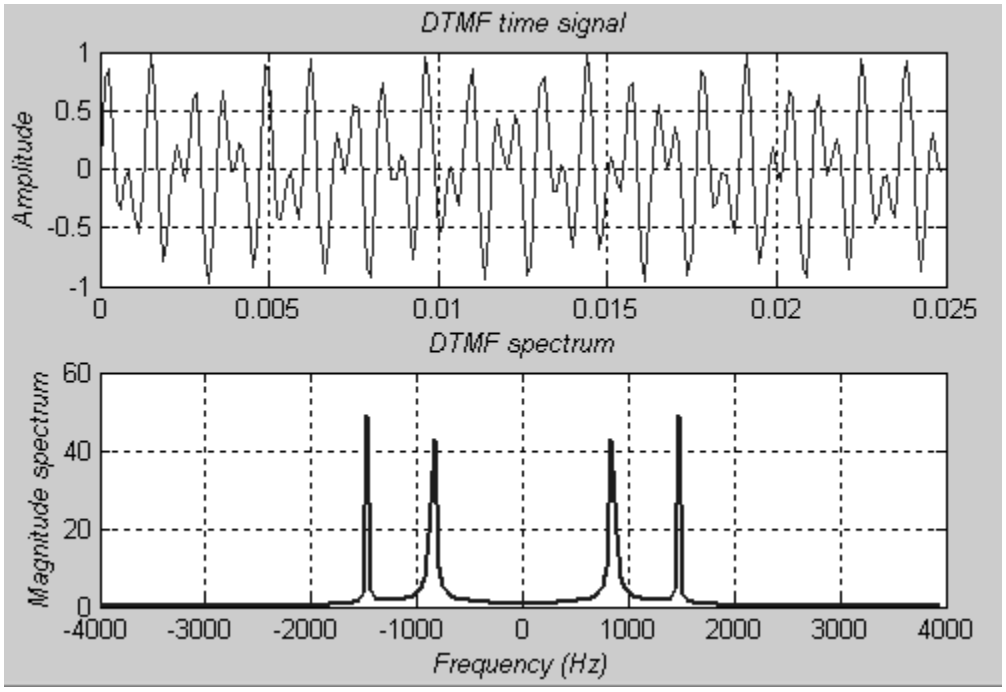
Telephone tones:

A telephone keypad has 12 buttons, a microphone and a speaker. The phone is connected to the central office through a single pair of wires. The phone company office supplies –48 volts at 80 ma to power the telephone. When the phone company wants to ring your phone, they generate a

20 Hz, 90 volt RMS signal that pulses on for 2 seconds, then off for 4 seconds. Each time a user presses a button on their phone, the phone turns on a pair of oscillators. The frequency of these oscillators change, based on the button you pressed as shown in Figure 1.

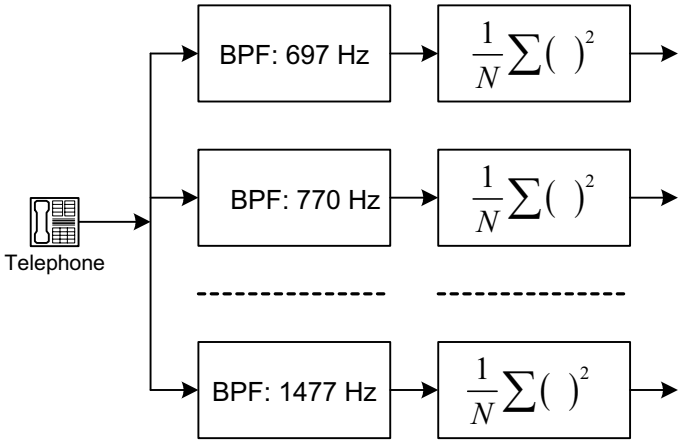
Practice:

```
function [s,t]=tone(test,N)
%-------------------------------------
%  freq(Hz)   1209   1336   1477
%  697          1      2      3
%  770          4      5      6
%  852          7      8      9
%  941          *      0      #
%-------------------------------------
%Example: [s,t]=tone('test',200)
Fs=8000;                    %sampling frequency
dt=1/Fs;                    %time interval
t=(0:(N-1))*dt;             %time base
fr=[697 770 852 941];     %row frequencies
fc=[1209 1336 1477];      %column frequencies
if     test=='1',r=1;c=1;
elseif  test=='2',r=1;c=2;
elseif  test=='3',r=1;c=3;
elseif  test=='4',r=2;c=1;
elseif  test=='5',r=2;c=2;
elseif  test=='6',r=2;c=3;
elseif  test=='7',r=3;c=1;
elseif  test=='8',r=3;c=2;
elseif  test=='9',r=3;c=3;
elseif  test=='*',r=4;c=1;
elseif  test=='0',r=4;c=2;
elseif  test=='#',r=4;c=3;
else disp('test not valid'); return;
end
s=[sin(fr(r)*2*pi*t)+sin(fc(c)*2*pi*t)]/2;
subplot(2,1,1);plot(t,s);grid
title('\itDTMF time signal')
ylabel('\itAmplitude');
spec=fftshift(abs(fft(s,N)));
f=[-N/2: N/2-1]*Fs/N;
subplot(2,1,2);plot(f,spec,'LineWidth',2);grid
title('\itDTMF spectrum')
xlabel('\itFrequency (Hz)')
ylabel('\itMagnitude spectrum')
return
```



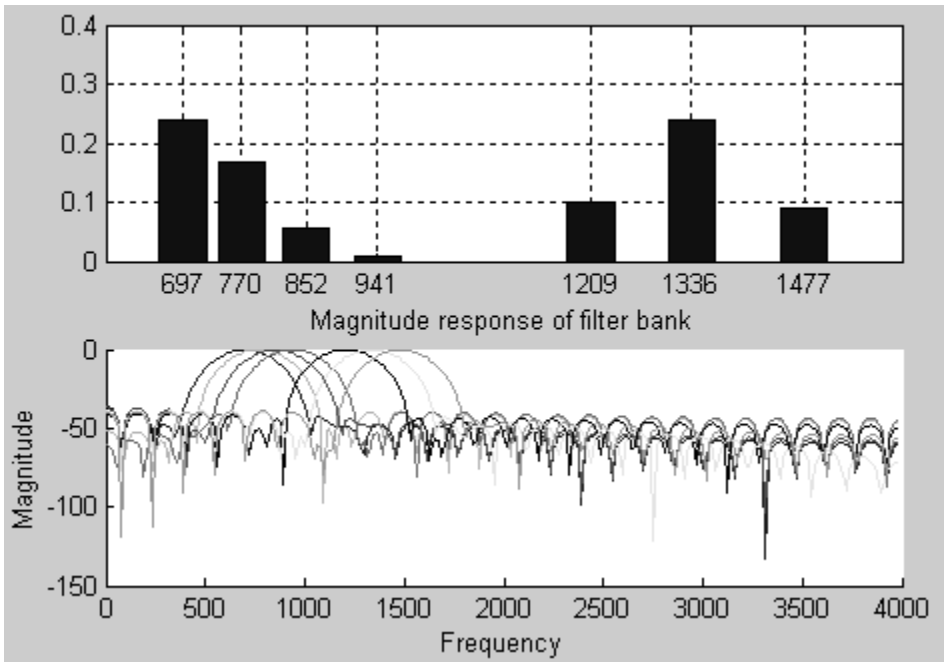
DTMF Decoder:

In this programming exercise we will implement a decoder to indicate which DTMF tone is being generated. This can be done in a variety of ways. We will use a simple FIR filter bank. The filter bank consists of seven bandpass filters, each passing only one of the seven possible DTMF frequencies. When the input to the filter bank is a DTMF signal, the outputs from two of the bandpass filters would be larger than the rest. If we detect which two are the largest, then we know the corresponding frequencies. These frequencies are then used to determine the DTMF code. A good measure of the output levels is the average power of the outputs.




```
clear all, clc;
fs=8000;
```

```
ft=[697  770  852  941  1209  1336  1477];
Nf=50;
N=input("Enter the value of N: ');
T=(N-1)/fs;
disp(' Observation time');
disp(T);
t=[0: (N-1)]/fs;
x=sin(2*pi*t*ft(1)+sin(2*pi*t*ft(6));
for I=1:7
        f1=ft(i)-1;
        f2=ft(i)-1;
        b(i,:)=fir1(Nf,[f1 f2]*2/fs);
        y(i,:)=filter(b(i,:),1,x);
        p(i)=1/N*sum(y(i,:).^2);
end
subplot(2,1,1); bar(ft,p);
subplot(2,1,2); hold on
for i=1:7
        [H,f]=freqz(b(i,:),1,256,fs);
        c=['b','c','m','k','g','y'];
        plot(f,20*log10(abs(H), c(i));
        title('Magnitude response of filter bank')
        xlabel('Frequency')
        ylabel('Magnitude')
end
hold off
```



Speech Processing:

In this section, we look at some basic speech processing within the MATLAB environment. Toward this end, we recommend the following minimum hardware:

- 16-bit sound card
- Microphone (for recording), with better than –60 dBm sensitivity
- Headphones or a set of loudspeakers

There are various ways to get sound into MATLAB as vectors of data. A "**.wav**" file produced by any windows editing/recording utility can be read into a MATLAB array using the **wavread** function. To playback a sound as samples in a vector, use the **sound** function.

Note: *The "cool edit 2000*" (http://www.syntrillium.com) *editing/recording utility can be used to obtain your own speech. Be sure to avoid clipping. Clipping will occur if you speak too loudly or hold the microphone too close to your mouth*.

Syntax:

>>[y,Fs,bits]=wavread('africa.wav');

Returns a vector **y** that contains the audio samples (digitized human voice), **Fs** is the sampling frequency, and **bits** is the number of bits used to quantize the samples. This information is all contained in the header of the **.wav** file, and **wavread** strips it out before assigning the audio data to **y**. The file **africa.wav** is a short recording of speech. The range of data values in **y** should lie between –1 and 1, with saturation of the sound equipment occurring for values outside this range.

Make sure to use the semi-colon at the end of the statement; the file might have over hundreds of thousands samples in length.
If you have a sound card, you can listen to the sounds as follows:

>>sound(y,Fs);

This function plays back the sound stored in vector **y**. The second argument of the **sound** command tells MATLAB how fast to send the samples of **y** to the speakers (Try different values for **Fs** if you want to hear the sound played back at different rates). The default value of **Fs** is 8192 Hz.

Practice:

```
>>t=440;                    %signal frequency in Hz
>>Fs=48000;                 %sampling frequency in Hz
>>tspan=0.4;                %signal duration in seconds
>>t=0;1/Fs:tspan;           %time base
>>x=sin(2*pi*f*t); grid     %signal x(t)
>>sound(x, Fs);             %play sound
```

Practice:

```
>>f=440;                    %signal frequency in Hz
>>Fs=48000;                 %sampling frequency in Hz
```

```
>>tspan=0.4;                 %signal duration in seconds
>>t=0:1/Fs:tspan;            %time base
>>x=sign(sin(2*pi*f*t));     %square wave
>>sound(x,Fs);               %play sound
```

Wave (.wav) Files:

The WAVE file format is an uncompressed audio file format that contains samples of an audio signal. WAVE files take up a lot of memory because of the fact that they are uncompressed. A WAVE file contains both sampled audio data along with other useful information about the file itself, such as the sampling rate, etc. Since WAVE files are binary, you cannot just open them up in a text editor and expect to be able to read it

Practice:

```
>>N=400;
>>h=ones(N,1);
>>h=h/sum(h);
>>[x,Fs]=wavread('*.wav');
>>y=conv(x,h);
>>soundsc(y,Fs);
```

Practice:

1. Design a low-pass Butterworth filter with a passband from 0 to 0.3*pi, and a stopband from 0.5*pi to pi. Allow at most 1 dB loss in the passband, and at least 25 dB loss in the stopband.
2. Read the file **messy.wav**, filter the signal using the designed filter, and play it over the speaker.

```
>>[x,Fs]=wavread('c:\messy.wav');    %load the wave file
>>[N,Wn]=buttord(0.3,0.5,1,20);      %estimate filter order and cutoff frequency
>>[num,den]=butter(N,Wn);            %design the Butterworth filter
>>y=filter(num,den,x);               %generate a filtered output
>>sound(y,Fs);                       %play the sound
```

The spectrogram:

A spectrogram is a visual representation of the frequencies that make up a particular sound signal. A complex signal such as the voice can be viewed as the sum of simple sinusoidal signals. In a spectrogram, the horizontal dimension corresponds to time, and the vertical dimension corresponds to frequency (or pitch). The relative intensity of the sound at any particular time and frequency is indicated by colors. Each horizontal line on the graph represents a harmonic in the signal and their relative strengths can be read by changes in color and brightness. Colors represent a range of signal intensities. Blue represents the lowest level and red represents the highest level. In MATLAB, we can use the **specgram** function to display the spectrogram of a sound signal.

Syntax:

```
>>[y,f,t]=specgram(x,nfft,Fs, window, noverlap)
```

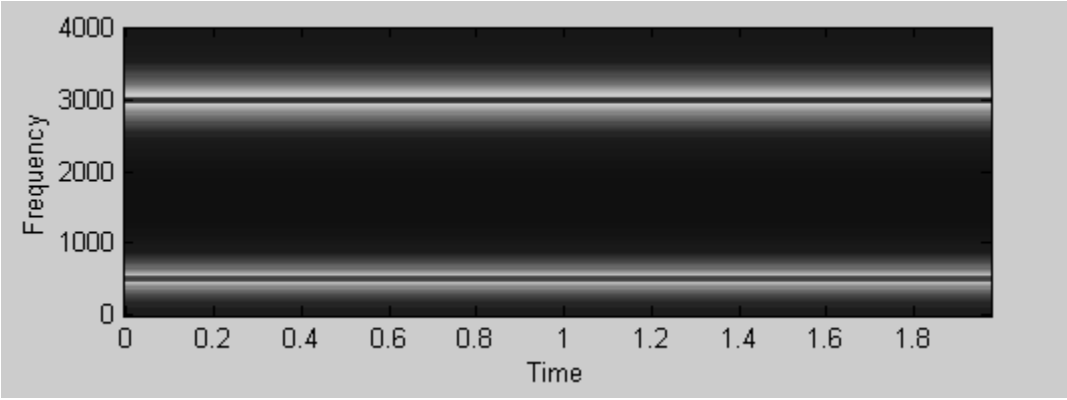
where **x** is the time-varying signal, **nfft** is the number of samples used in the DFT, **Fs** is the sampling rate, **window** specifies the time window being used for FFT, and the **noverlap** specifies the amount of sample overlap. This function returns spectrogram in matrix **y**, the **f** and **t** are vectors representing frequency and time respectively.

Practice:

Use the **specgram** command to display the spectrogram of a dual-tone given by

$$x(t) = 3\cos(2\pi 500t) + 7\cos(2\pi 3000t)$$

```
>>Fsamp=8000;                                  %sampling rate
>>inc=1/Fsamp;                                 %time increment
>>dur=2;                                       %time duration
>>t=0:inc:dur;                                 %time vector
>>x=3*cos(2*pi*500*t)+7*cos(2*pi*3000*t);      %dual-tone
>>specgram(x, 256,Fsamp);                      %spectrogram
```



Chirp signals:

A chirped sinusoidal signal is a sinewave of increasing frequency over some prescribed period. In other words, a chirped sinusoid begins at one frequency and smoothly moves to another frequency over a time interval. Chirp signals have been used in radar development since the 1960's because of their advantages over single frequency signals.

A typical chirped sinusoidal signal can be written as

$$c(t) = \cos\left(2\pi\mu t^2 + 2\pi f_0 t + \Psi\right)$$

where f_0 is the initial frequency, t is time, and μ is a constant.

The instantaneous frequency of the chirp is found by taking the time derivative of the phase:

$$f_i(t) = \frac{1}{2\pi}\frac{d}{dt}\left(2\pi\mu t^2 + 2\pi f_0 t + \Psi\right) = 2\mu t + f_0$$

which clearly reflects a linear frequency variation with time. Since the linear variation of the frequency can produce an audible sound similar to a siren or a bird chirp; the linear-FM signals are also called "chirp" signals or simply "chirps"

Practice:

Use the following parameters to define a chirp signal:

- $f_{\min} = 200$ Hz (lower frequency)
- $f_{\max} = 2000$ Hz (upper frequency)
- $T = 2$ Sec (time range)

Specify μ and f_0 to define the signal $c(t)$ so that it sweeps the specified frequency range.

Solution:

Because the time range for the frequency sweep is 2 seconds, we obtain a system of two equations and two unknowns,

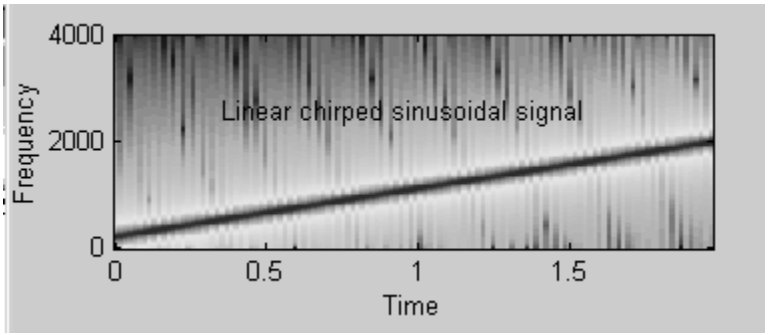
- $200 = 2\mu(0) + f_0 \Rightarrow f_0 = 200$ Hz
- $2000 = 2\mu(2) + f_0 \Rightarrow \mu = 450$ Hz/sec

Thus, the chirped sinusoidal signal is given by

$$c(t) = \cos\left(2\pi(450)t^2 + 2\pi(200)t + \Psi\right)$$

where the phase Ψ is arbitrary.

```
Fsamp=8000;
inc=1/Fsamp;
dur=2;
t=0:inc:dur;
psi=2*pi*(200*t+450*t.^2);
c=cos(psi);
subplot(2,1,1),specgram(c,256,8000);
sound(c,Fsamp);
```



MATLAB has a built-in function **chirp** to generate chirp signals. The command syntax is as follows:

>>y=chirp(t, f0, t1, f1)

Generates samples of a linear swept-frequency cosine signal **y** at the time instances defined in array **t**, where **f0** is the instantaneous frequency at time **0**, and **f1** is the instantaneous frequency at time **t1**. The frequencies **f0** and **f1** are both expressed in Hertz. If unspecified, **f0** is **0**, **t1** is **1**, and **f1** is **100**.

Practice:

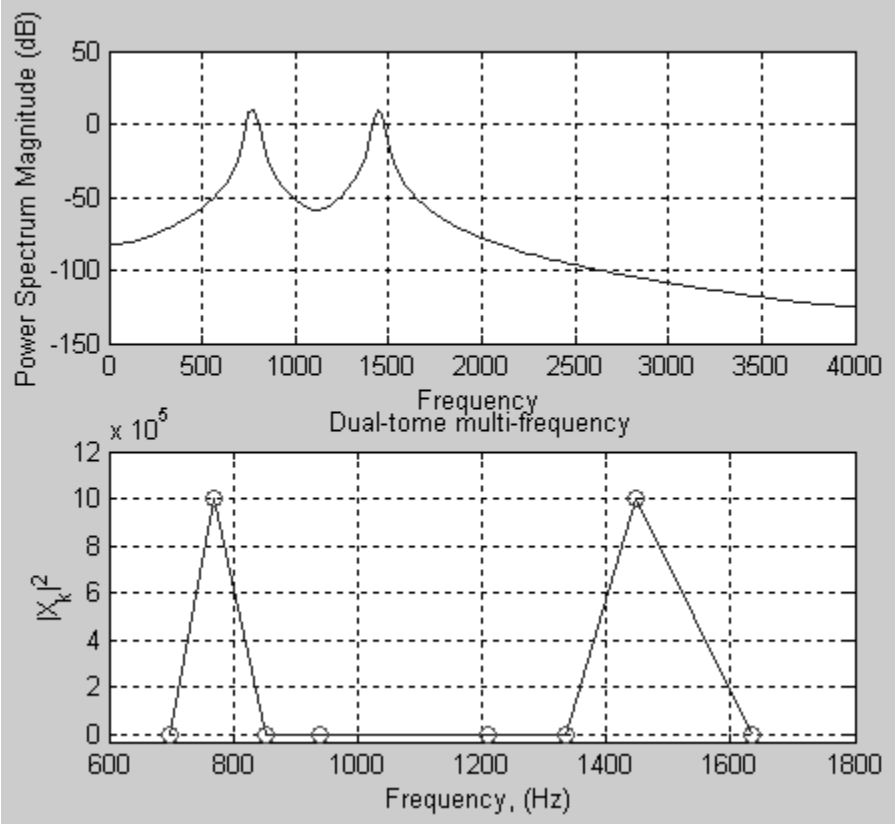
```
t=0: 0.001:2                    %2 secs at 1 kHz sample rate
y=chirp(t,0,1,150)              %start at dc, cross 150 Hz at t=1 sec
specgram(y,256,1e3,256,250);    %display the spectrogram
```

Goertzel Algorithm:

One method of calculating the DFT is the Goertzel algorithm. The computation of the DFT may be implemented using a first-order complex recursive structure, which consists of one complex addition and one complex multiply.

Practice:

```
Fs=8000;
f=[697  770  852  941  1209  1336  1447  1633];
t=[0:4000]'*1/Fs;
x=0.4*sin(2*pi*f(2)*t)+0.4*sin(2*pi*f(7)*t);
subplot(2,1,1); psd(x,[],Fs);
N=length(x);
omega=2*pi*f*(1/Fs);
B=1; x2=[];
for m=1:length(f)
        A=[1, -2*cos(omega(m)), 1];
        v=filter(B,A,x);
        x2(m)=v(N).^2+v(N-1).^2-2*cos(omega(m))*v(N)*v(N-1);
end
subplot(2,1,2); plot(f,x2,f,x2,'o'); xlabel('Frequency,(Hz)'); ylabel('|X_k|^2');
title('Dual-tone multi-frequency')
set(gca,'Ylim',[-3e4,7e5])
```



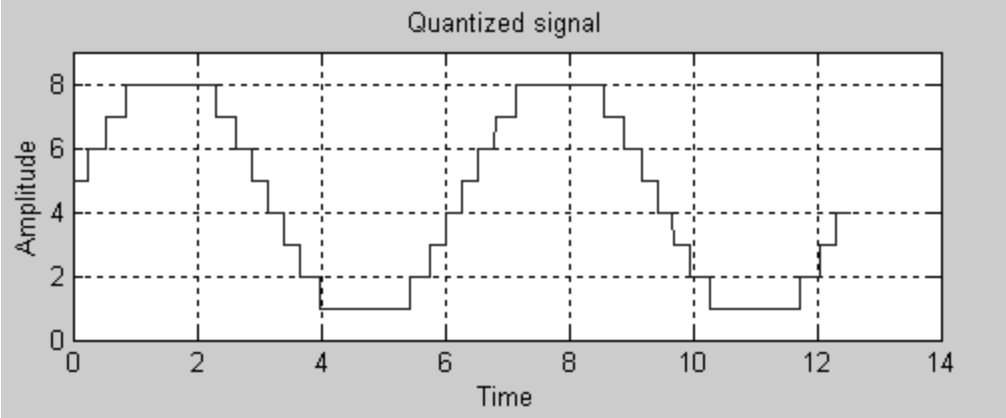
Quantization:

Quantization is the process of representing a value with reduced precision. For example, a five level uniform quantizer maps the input sequence values into five distinct output levels. A finite number of bits are required to represent the discrete value, as a result this representation lacks full precision and it is irreversible.

Practice:

Quantize the signal $x(t) = 5 + 4\sin(t)$ to integer values.

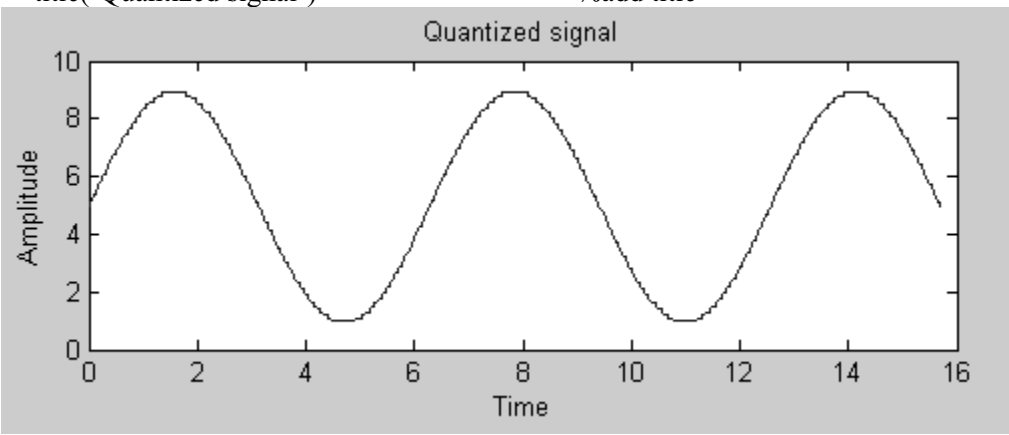
```
>>t=0:0.001: 5*pi;                          %time base
>>x=fix(5+4*sin(t));                        %quantize signal to integer values
>>plot(t,x, 'LineWidth',3);                 %plot quantized signal
>>xlabel('Time'); ylabel('Amplitude');      %label axes
>>title('Quantized signal')                 %add title
>>axis([0 14 0 max(x)+1])                   %scale axes
```



It is possible to adjust the quantization step size by pre-scaling and post-scaling the function values. The following yields functional values limited to one digit to the right of the decimal point.

Practice:

```
>>t=0:0.001:5*pi;                          %time base
>>x=fix(10*(5+4*sin(t)))/10;               %quantized signal
>>plot(t,x);                               %plot quantized signal
>>xlabel('Time'); ylabel('Amplitude')      %label axes
>>title('Quantized signal')                %add title
```



In digital communication, the data format is typically limited by the number of binary digits (bits). Common computer hardware utilizes 8 or 16 bits in the signal value representation. For example, a signal restricted to 8-bit format could consist of a sequence of integers lying between –256 and +255. MATLAB for constructing a single cycle of an 8-bit sinusoide at full amplitude scale might proceed as follows:

```
>>n=0:1000;
>>x=fix(255*sin(2*pi*n/1000));
>>plot(n,x)
```

Finally, to produce an audio signal, which is compatible with MATLAB **sound** function, we must restrict the signal values to real values lying between –1 and +1 inclusive. The following

MATLAB piece of code plays a quantization of a pure 1 kHz tone 3 seconds in duration, sampling rate of 8192 Hz.

Practice:

```
%5-bit quantization
n=0:(3*8192)-1;
y=((2^(5-1))-1)*sin(2*pi*1000/8192*n);
y=fix(y)/(2^(5-1));
sound(y,8192)
```

Zero-crossings:

Suppose we are given a signal $x(t) = \sin(2\pi 2t)$. We shall consider counting the number of zero-crossings.

Practice:

```
>>t=0:0.01:2;
>>x=sin(2*pi*2*t);
>>prod=x(1:length(x)-1).*x(2:length(x));
>>crossings=length(find(prod<0))

crossings=
                    7
```

Audio effects:

In this section we shall turn into some audio effects that will prove useful in digital signal processing. We will briefly cover

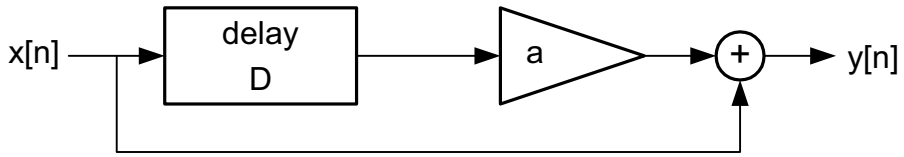
- Delay
- Echo
- Reverberation (reverb)
- Chorus

Delay:

A delay is one of the simplest effects, but very valuable if used properly. Simply stated, a delay takes an audio signal, and plays it back after the delay time. The delay time can range from several milliseconds to several seconds.

Echo effect:

A basic echo effect can be implemented according to the block diagram depicted below:



The difference equation describing this model is

$$y[n] = x[n] + ax[n - D]$$

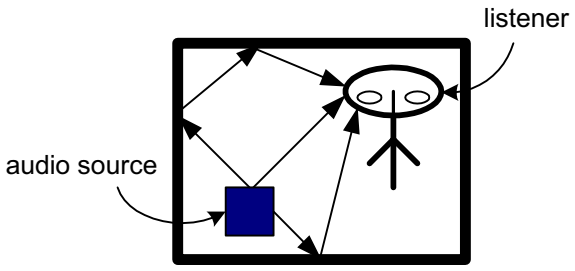
$$\therefore H(z) = 1 + az^{-D}$$

Practice:

```
clear all;
[x, fs,bits]=wavread('test.wav');
a=input('Enter the value of attenuation: ');
delay=0.3;
D=round(delay*fs);                          %number of samples in the delay
num=[1  zeros(1,D)  a];
den=[1];
x_delay=filter(num,den,x);
y=x+x_delay;
sound(y,fs)
```

Reverberation:

This is by far the most heavily used effect in music. Reverberation is the result of many reflections of a sound that take place in a room. A reflected sound wave will arrive a little later than the direct sound, and is typically little weaker. The series of delayed and attenuated sound waves is termed reverb.



Chorus:

Just as a chorus is a group of singers, the chorus effect can make a single instrument sound like there are actually several instruments being played.