EEE 304

Lab Exercise 1

1. Introduction to MATLAB

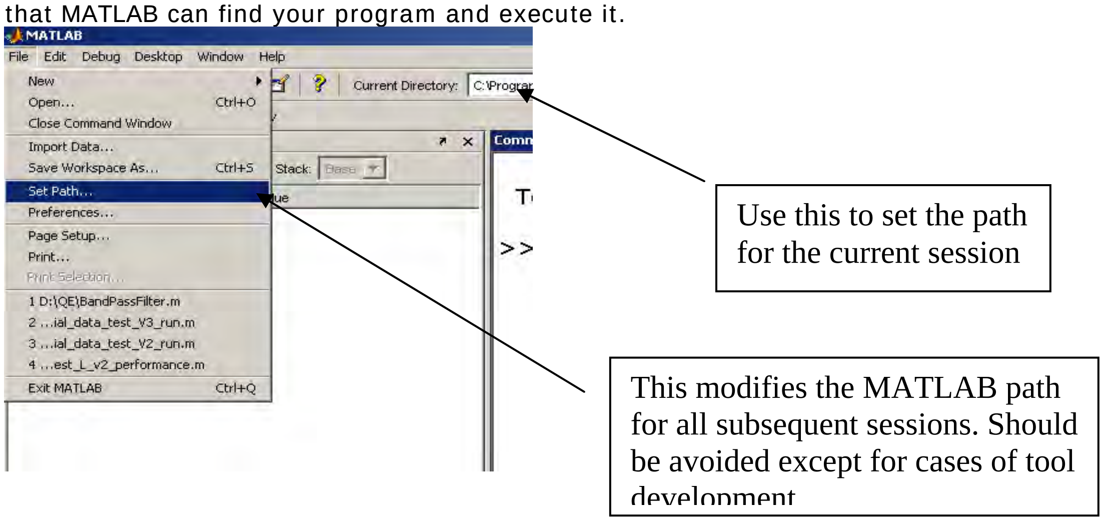
MATLAB is a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numeric computation. Using MATLAB, one can solve technical computing problems easier than with traditional programming languages, such as C, C++, and Fortran. The resulting code is vastly more efficient in terms of code development and debugging time, code reusability and maintainability. Although net speed of execution is inferior to that of an optimized low-level code, recent versions allow the auto-coding of MATLAB programs into executables whose speed is often comparable to the more traditional codes.

MATLAB is useful in a wide range of applications, including signal and image processing, communications, control design, test and measurement, financial modeling and analysis, and computational biology. Add-on toolboxes (collections of special-purpose MATLAB functions, available separately) extend the MATLAB environment to solve particular classes of problems in these application areas. MATLAB provides a number of features for integrating code with other languages and applications and documenting and sharing work. (See Mathworks website http://www.mathworks.com.)

The objective of this sequence of laboratory exercises is to explore a few fundamental problems arising in System Theory and, at the same time, develop expertise in problem solving with MATLAB.

2.1 MATLAB basics: Setting the path

In the MATLAB development environment, you should set the path where your program is so that MATLAB can find your program and execute it.



Use this to set the path for the current session

This modifies the MATLAB path for all subsequent sessions. Should be avoided except for cases of tool development

Fig 1 Setting the path of your program

2.2 MATLAB Commands

There are 2 ways to use MATLAB: One is to type the MATLAB Command in the "Command Window". For example , the MATLAB Command "bench" is to compare your computer execution speed with other computers by computing some task. By type "bench" and press the key "Enter", your will see some figures, tables about your computer execution speed. If you want to know more information about "bench", type " help bench" in "command window" and press the key "Enter". You will find more information about it.

The second way is to write a script with extension name "m", in which you can type more than one command. For example, you write 3 commands below in a script with the name 'eee304.m'.

```
x=1;
y=2;
z=x+y;
```

when you type the filename in the command window and press the key 'Enter', the file will be executed. These 3 commands will add the variables x and y and assign the result to the variable z.

In the following we will work with commands that are useful in the analysis of systems. For detail help, consult the MATLAB Manual, or use the built-in help command.

3.1 Create a LTI system by Transfer Function Models

Related Function: tf.

Arguments: numerator polynomial, denominator polynomial, sampling time (optional).

You can create the transfer function $h(s) = \frac{1}{s^2+1}$ by

```
h= tf([1 ],[1 0  1])
```

3.2 Step and Bode frequency response of LTI models

Related Function: bode, step

bode (SYS) draws the Bode plot of the LTI model SYS (created with tf or other system creation commands). The frequency range and number of points are chosen automatically (a manual selection is possible, see "help bode").

step Step response of LTI models.
step (SYS) plots the step response of the LTI model SYS. For multi-input models, independent step commands are applied to each input channel. The time range and number of points are chosen automatically.

3.3 Continuous to discrete time system conversion

Related Function : c2d(SYS,TS), TS is the sample time. Compare the step responses of a system and its discretization, for different sample times: e.g., step(SYS,c2d(SYS,TS));.

3.4 Frequency response of LTI system

Related Function : freqz, freqs (older functions)

[H,W] =freqz(B,A,N);

Returns the N-point complex frequency response vector H and the N-point frequency vector W in radians/sample of the DT filter:

$$h(e^{j\omega}) = \frac{b(1) + b(2)e^{-j\omega} + \cdots + b(m+1)e^{-jm\omega}}{a(1) + a(2)e^{-j\omega} + \cdots + a(n+1)e^{-jn\omega}}$$

where B=[b(1) b(2) ... b(m+1)] A=[a(1) a(2) ... a(n+1)]

See also bode, dbode (now obsolete)

3.5 The frequency analysis of a signal

Related Function : fft

FFT implements the Discrete Fourier transform evaluated at discrete frequencies.

3.6 Filtering

The response of a linear system to arbitrary inputs can be computed with the MATLAB command lsim.
LSIM(SYS,U,T) plots the time response of the LTI model SYS to the input signal described by U and T. The time vector T consists of regularly spaced time samples and U is a matrix with as many columns as inputs and whose i-th row specifies the input value at time T(i).
For example,

```
        t = 0:0.01:5;   u = sin(t);   lsim(sys,u,t)
```

simulates the response of a single-input model SYS to the input u(t)=sin(t) during 5 seconds.

For discrete-time models, U should be sampled at the same rate as SYS (T is then redundant and can be omitted or set to the empty matrix).

See also the signal processing Function : filter (older format)

A different approach is to use the SIMULINK GUI environment to perform the simulation (see more details below).

3.7 Filter design

Design a Butterworth filter (lowpass, highpass or bandstop)
Function: butter

EXAMPLE:

```
[B,A] = butter(N,Wn); % the analog Butterworth filter

H=tf([B] ,[A]); % create the sys by tf

Bode(H);% bode plot

SYSD =c2d(H,1);% conversion

Noise=randn(1,1024); % signal
AF_noise=abs(fft(noise)); % amplitude-frequency
Filitered_noise= filter(B,A,noise);
%plot
```

3.8 Sample code

```
% MATLAB Code Sample
% EEE304
% sampling parameter
Fs=1000;% sampling frequency 1000Hz

% noise
% the length of noise is 1024 points
noise=randn(1,1024);

%the frequency range
frequency=[0:1023]*1000/2/1024;

% amplitude-freqency of noise
AF_noise=abs(fft(noise));

% plot
figure
plot(frequency,AF_noise)
xlabel('Hz')
ylabel('Amplitude')
```

```
% bandpass filter
Wn=[50 200]; % pass band is 50~200Hz

%fir fliter
n    = 50; %order n
Wn = Wn/(Fs/2); %cutoff frequency  [300 6000]/sampling frquency/2
b = fir1(n,Wn);
a=[1];

y = filter(b,a,noise);
figure
plot(abs(fft(y)))
xlabel('Hz')
ylabel('Amplitude')
```

4.1 Introduction to SIMULINK

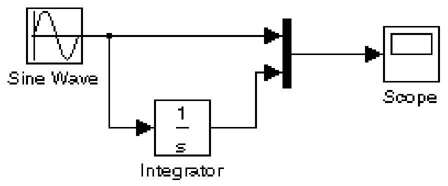
SIMULINK is a platform for multidomain simulation and Model-Based Design of dynamic systems. It provides an interactive graphical environment and a customizable set of block libraries that allows for an accurate design, simulation, implementation, and testing of control, signal processing, communications, and other time-varying system applications.

Add-on products extend the SIMULINK environment with tools for specific modeling and design tasks and for code generation, algorithm implementation, test, and verification.

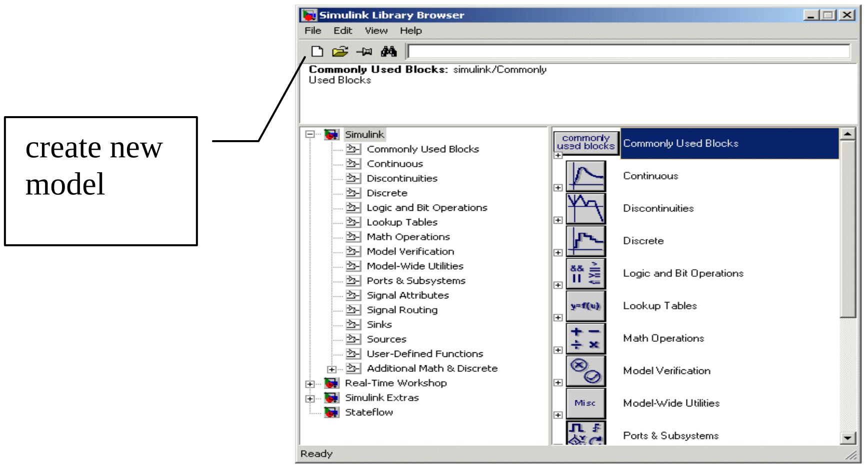
SIMULINK is integrated with MATLAB, providing immediate access to an extensive range of tools for algorithm development, data visualization, data analysis and access, and numerical computation.

4.2 Using SIMULINK to build a Model

This example shows how to build a model using some of the model-building commands and actions. The model integrates a sine wave and displays the result along with the sine wave. The block diagram of the model looks like this.



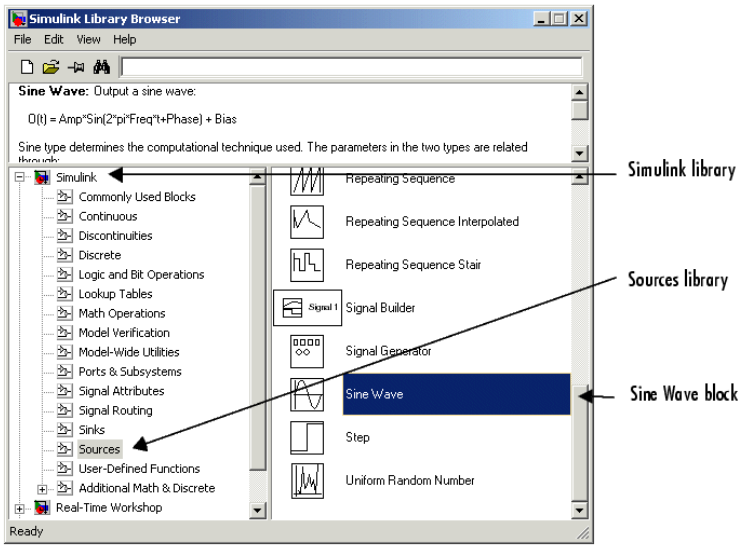
To create the model, first enter SIMULINK in the MATLAB Command Window. On Microsoft Windows, the SIMULINK Library Browser appears.



You then need to copy blocks into new model file (.mdl) from the following SIMULINK block libraries:

- Sources library (the Sine Wave block)
- Sinks library (the Scope block)
- Continuous library (the Integrator block)
- Signal Routing library (the Mux block)

Copying is performed wither by drag-and-drop, or with the standard Windows commands (ctrl-C, ctrl-V). You can copy a Sine Wave block from the Sources library, using the Library Browser (Windows only) or the Sources library window (UNIX and Windows). To copy the Sine Wave block from the Library Browser, first expand the Library Browser tree to display the blocks in the Sources library. Do this by clicking the Sources node to display the Sources library blocks. Finally, click the Sine Wave node to select the Sine Wave block. Here is how the Library Browser should look after you have done this.


Sine Wave: Output a sine wave:

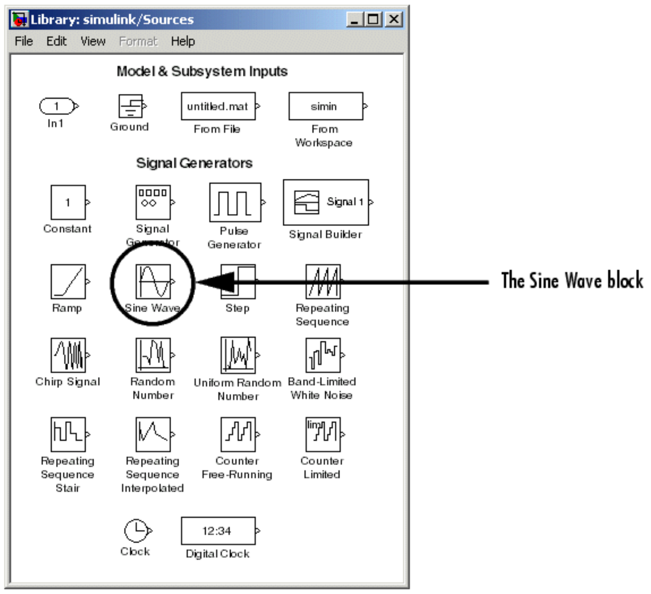
O(t) = Amp*Sin(2*pi*Freq*t+Phase) + Bias

Sine type determines the computational technique used. The parameters in the two types are related through:

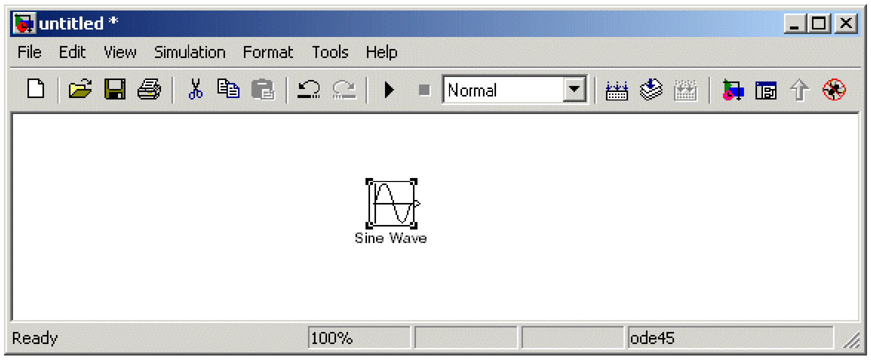
Now drag a copy of the Sine Wave block from the browser and drop it in the model window.

To copy the Sine Wave block from the Sources library window, open the Sources window by double-clicking the Sources icon in the SIMULINK library window. (On Windows, you can open the SIMULINK library window by right-clicking the SIMULINK node in the Library Browser and then clicking the resulting Open Library button.)

SIMULINK displays the Sources library window.

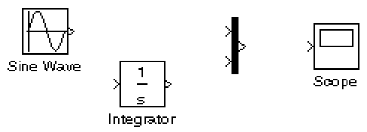


Now drag the Sine Wave block from the Sources window to your model window.

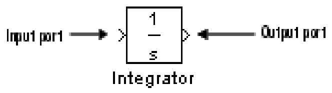


Copy the rest of the blocks in a similar manner from their respective libraries into the model window. You can move a block from one place in the model window to another by dragging the block. You can move a block a short distance by selecting the block, then pressing the arrow keys.

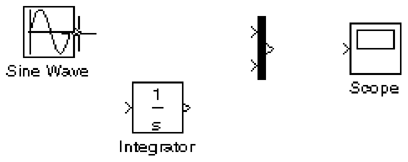
With all the blocks copied into the model window, the model should look something like this.



If you examine the blocks, you see an angle bracket on the right of the Sine Wave block and two on the left of the Mux block. The > symbol pointing out of a block is an output port; if the symbol points to a block, it is an input port. A signal travels out of an output port and into an input port of another block through a connecting line. When the blocks are connected, the port symbols disappear.

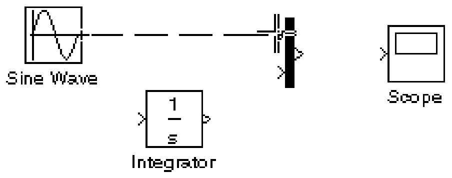


Now it's time to connect the blocks. Connect the Sine Wave block to the top input port of the Mux block. Position the pointer over the output port on the right side of the Sine Wave block. Notice that the cursor shape changes to crosshairs.

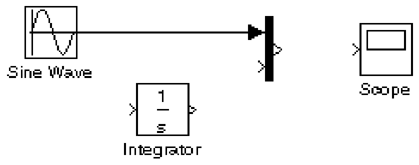


Hold down the mouse button and move the cursor to the top input port of the Mux block.

Notice that the line is dashed while the mouse button is down and that the cursor shape changes to double-lined crosshairs as it approaches the Mux block.

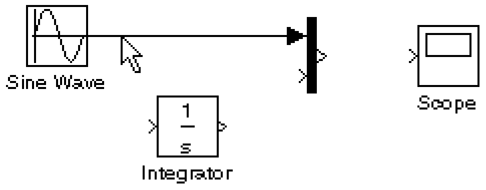


Now release the mouse button. The blocks are connected. You can also connect the line to the block by releasing the mouse button while the pointer is over the block. If you do, the line is connected to the input port closest to the cursor's position.

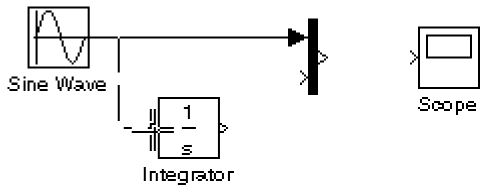


If you look again at the model at the beginning of this section (see Building a Model), you'll notice that most of the lines connect output ports of blocks to input ports of other blocks. However, one line connects a line to the input port of another block. This line, called a branch line, connects the Sine Wave output to the Integrator block, and carries the same signal that passes from the Sine Wave block to the Mux block.

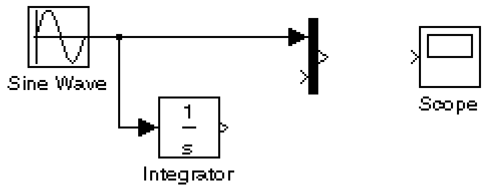
Drawing a branch line is slightly different from drawing the line you just drew. To weld a connection to an existing line, follow these steps:
1. First, position the pointer on the line between the Sine Wave and the Mux block.



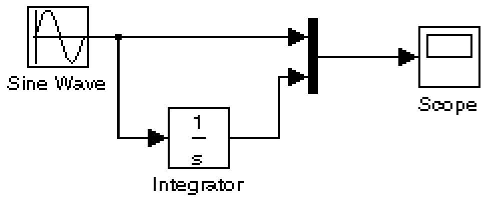
2. Press and hold down the Ctrl key (or click the right mouse button). Press the mouse button, then drag the pointer to the Integrator block's input port or over the Integrator block itself.



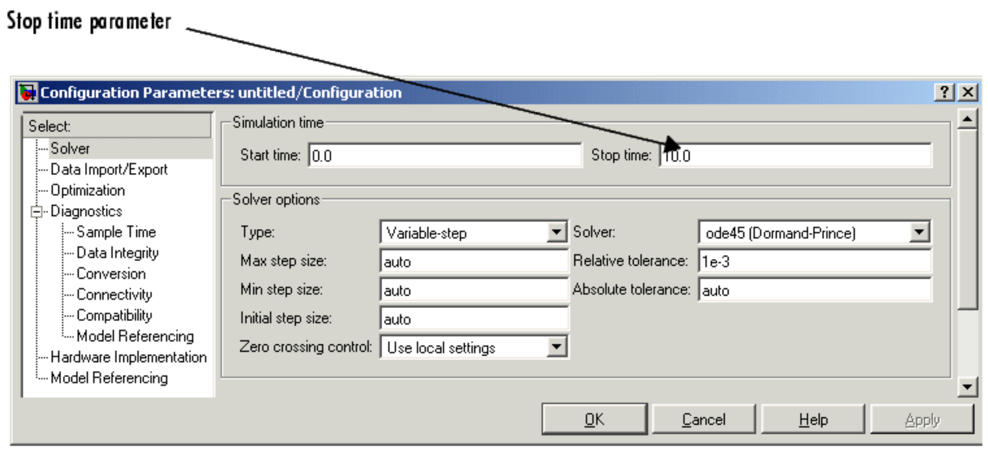
3.Release the mouse button. SIMULINK draws a line between the starting point and the Integrator block's input port.
Note that an alternative procedure, not using the ctrl key, is to begin from the input port of the integrator and connect to a point on the sine wave signal line. However, if the final point is not exactly on the signal line the connection is not made and the input line remains dashed.



Finish making block connections. When you're done, your model should look something like this.



Now set up SIMULINK to run the simulation for 10 seconds. First, open the Configuration Parameters dialog box by choosing Configuration Parameters from the Simulation menu. On the dialog box that appears, notice that the Stop time is set to 10.0 (its default value).

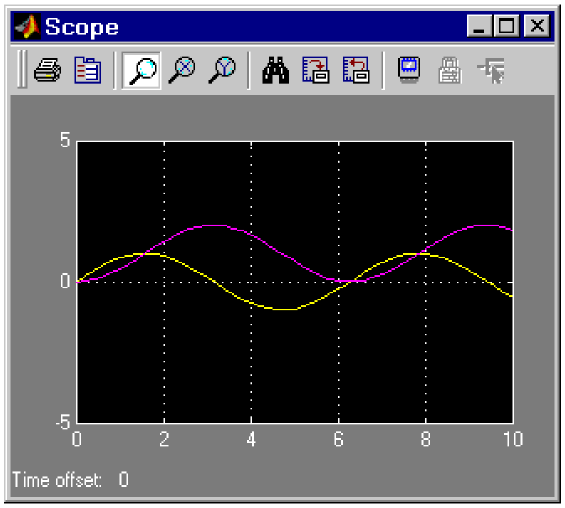


Start time: 0.0
Stop time: 10.0

Type: Variable-step
Solver: ode45 (Dormand-Prince)
Max step size: auto
Relative tolerance: 1e-3
Min step size: auto
Absolute tolerance: auto
Initial step size: auto
Zero crossing control: Use local settings

Close the Configuration Parameters dialog box by clicking the OK button. SIMULINK applies the parameters and closes the dialog box.

Now double-click the Scope block to open its display window. Finally, choose Start from the Simulation menu and watch the simulation output on the Scope.



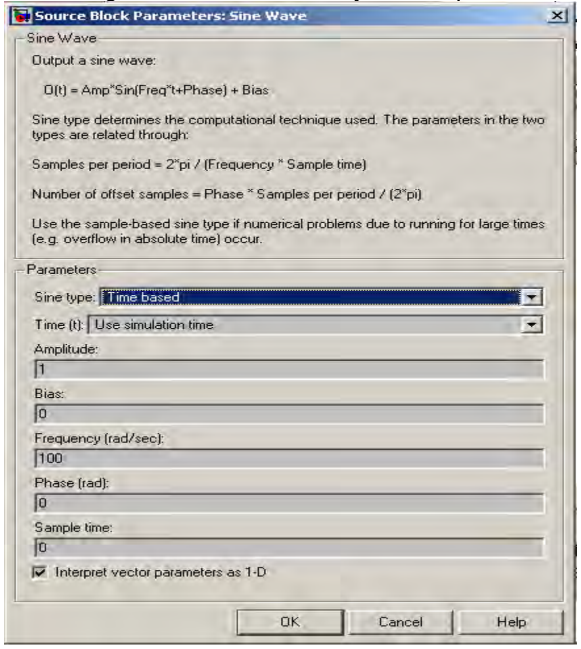
The simulation stops when it reaches the stop time specified in the Configuration Parameters dialog box or when you choose Stop from the Simulation menu or click the Stop button on the model window's toolbar (Windows only).

To save this model, choose Save from the File menu and enter a filename and location. That file contains the description of the model.

To terminate SIMULINK and MATLAB, choose Exit MATLAB (on a Microsoft Windows system) or Quit MATLAB (on a UNIX system). You can also enter quit in the MATLAB Command Window. If you want to leave SIMULINK but not terminate MATLAB, just close all SIMULINK windows.

Parameter setting

For the Sine Wave block, there are some adjustable parameters. By double mouse click on the component, there will be a dialog shown. You can adjust the parameters in this dialog.



Source Block Parameters: Sine Wave

Sine Wave

Output a sine wave:

O(t) = Amp*Sin(Freq*t+Phase) + Bias

Sine type determines the computational technique used. The parameters in the two types are related through:

Samples per period = 2*pi / (Frequency * Sample time)

Number of offset samples = Phase * Samples per period / (2*pi)

Use the sample-based sine type if numerical problems due to running for large times (e.g. overflow in absolute time) occur.

Parameters

Sine type: Time based

Time (t): Use simulation time

Amplitude: 1

Bias: 0

Frequency (rad/sec): 100

Phase (rad): 0

Sample time: 0

☑ Interpret vector parameters as 1-D

ASSIGNMENT

Design analog low-pass filters with the following specifications.

- Low-pass filter with cutoff frequency 8KHz
- Use 1ms as your time unit

1. Design a 1st order low-pass filter as a benchmark.

2. Design a Butterworth filter of order 5 using MATLAB.

Plot and compare

- frequency responses (bode)
- step responses (step)
- Use SIMULINK to compute the responses of the filters to the signal

  x(t) = sin 300Hz + sin 31.413kHz, for 1s time interval.

  Compare the filtered signal with the sin 300Hz and discuss your observations.

3. Design discrete time filters of order 6 by

- using the c2d command to convert the CT filter
- using the butter command with the DT option

Use sampling rates 100kHz and 10kHz. Obtain the filter response for the previous test signal x(t) and compare with the continuous time case. Discuss your observations.

4. Compile your results and comments in a brief lab report. Include the final MATLAB codes and/or SIMULINK models that you used to produce your results.

Notes: For SIMULINK comparisons, you may use a MUX block and a scope. Make sure that in the scope “parameters” the data history is NOT limited. Save the scope results by printing to a suitable printer (e.g., Adobe PDF). The SIMULINK transfer function block will accept arguments which are variables defined in the workspace, e.g., num1, den1, where [num1,den1] = butter(6,4).

EEE 304

Lab Exercise 2: Sampling and Reconstruction

Sampling Theory:

Suppose we sample a signal, such as the one in Figure 1, which is band-limited to frequencies between 0 Hz and fn Hz. If we want to reconstruct the signal later, we must ensure that the sample rate, fs, is strictly greater than 2*fn. A signal sampled at fs = 2*fn is said to be Nyquist sampled, and fs = 2*fn is called the Nyquist frequency.

Note that information may be lost if a signal is sampled exactly at the Nyquist frequency. For example, the sine wave in Figure 1 has a frequency of 1/2 Hz. The Nyquist frequency is therefore 1 Hz. If we sample the sine wave at a rate of 1 Hz, say at t=0, t=1, t=2, and so on, all the sample values will be 0. The signal will look is if it were identically 0, and no reconstruction method will be able to recreate the 1/2 Hz sine wave. This is why fs must be strictly greater than 2*fn.

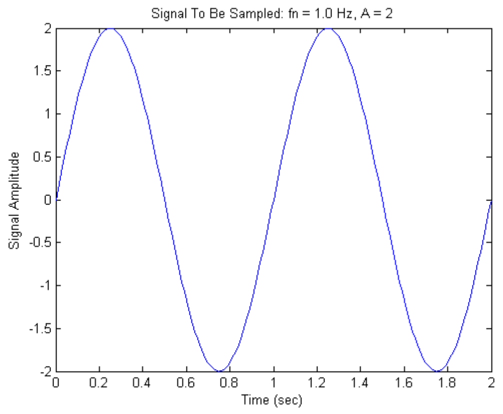


Figure 1: Sine Wave Signal.

Reconstruction Theory:

The reconstruction is implemented by applying the low pass filter to the sampled signal.

Frequency-magnitude character of signal:

The frequency-magnitude of discrete signal is described by the Discrete Fourier Transform (DFT). In Matlab, the DFT is done be the command '**fft**'.
Use the '**help fft**' in Matlab to learn how to use it.
The relationship between the analog frequency, ω, and digital frequency, f, is defined below

$$\omega = 2\pi f/f_s$$

where f_s is the sampling frequency.

Therefore, ω =2π corresponds to f=f_s. For 100 points fft, theses 100 points occupy the range, [0, 2 π] evenly. The frequency interval between 2 consecutive points is f_s/(100-1). The sample code shows a way to plot the frequency-magnitude.

```
%%%%%
%EEE 304 Lab2
% sample code
clear all
close all
clc


%%%%%%%%%%%%%%%%
% creat the sine wave
%%%%%%%%%%%%%%%%
fs=1000;                    %sample frequency1000Hz
t=0:1/fs:0.2;               %the duration: 0.2second
fn=50;                      %singla frequency 50Hz
y=sin(2*pi*fn*t);           % the sine wave
figure                      % create a figure
plot(t,y);                  % plot the wave
title('the sine wave');     % title of the figure
xlabel('Time (second)');    % the label of x-axis
ylabel('Magnitude');        % the label of y-axis


%%%%%%%%%%%%%%
%plot the frequency response
%%%%%%%%%%%%%%
freq_y=fft(y);              % implement FFT on y
mag_freq_y=abs(freq_y) ; % the magnitude of FFT
df=fs/(length(y)-1);        % 2pi corresponds to fs
freq=0:df:fs;               % the frequency range is from 0 to fs
figure
plot(freq,mag_freq_y)
title('frequency response');   % title of the figure
xlabel('Frequency (Hz)');      % the label of x-axis
ylabel('Magnitude');           % the label of y-axis
```

Some Derivations: Continuous Fourier Transform (CFT) interpretation of the Discrete Fourier Transform (DFT), the Fast Fourier Transform (FFT), and the Discrete Time Fourier Transform (DTFT).

The DTFT is a discrete-time transform defined for an infinite sequence of numbers, as follows.

$$X_{DTFT}(e^{j\Omega}) = \sum_{n=-\infty}^{\infty} x(n)e^{-j\Omega n}$$

The DTFT is periodic with period 2π.
The DFT is a discrete-time transform defined for a finite sequence of N numbers, as follows.

$$X_{DFT}(k) = \sum_{n=0}^{N-1} x(n) e^{-j\frac{2\pi}{N}kn}$$

The term FFT is used to signify a specific method to compute the DFT in an efficient manner. While a transform in its own right, the DFT/FFT can be interpreted in terms of the CT-Fourier transform for a sampled signal *x(t)*. Let us consider a signal *x(t)*, sampled at the time instants *nT*, *n* = 0,1,…*N*-1.

1. Periodic Case

We assume that the signal is periodic both in discrete and continuous time and that the period is NT.

The CFT is found by computing the FS expansion first:

$$X(jw) = 2\pi \sum_k a_k \delta(w - kw_0)\ ; \quad a_k = \frac{1}{NT} \int_{<NT>} x(t) e^{-jkw_0 t} dt\ ; \quad w_0 = \frac{2\pi}{NT}$$

The DTFT is also found by computing the DFS expansion first:

$$X_{DTFT}(e^{j\Omega}) = 2\pi \sum_k b_k \delta(\Omega - k\Omega_0)\ ; \quad b_k = \frac{1}{N} \sum_{<N>} x(n) e^{-jk\Omega_0 n}\ ; \quad \Omega_0 = \frac{2\pi}{N}$$

$$Notice: x(n) \leftrightarrow x(nT), b_k = \frac{1}{NT} \sum_{<N>} x(nT) e^{-jkw_0 nT} T \cong a_k$$

The last approximation is the discretization of the FS integral.

The DFT is: $X_{DFT}(k) = \sum_{<N>} x(n) e^{-jk\Omega_0 n} = Nb_k$

If, in addition, the signal is bandlimited and its maximum frequency is smaller than π/T, then there is no aliasing and $a_k = b_k$.

2. Aperiodic Case

Suppose that the signal is 0 outside the sampled interval [0, (N-1)T]. The Fourier transform of this signal is

$$X(jw) = \int_{-\infty}^{\infty} x(t) e^{-jwt} dt = \int_0^{(N-1)T} x(t) e^{-jwt} dt$$

First, in a naïve approach, let us approximate the integral by an Euler discretization, whereby the integrand is taken as piecewise constant.

$$X(jw) = \int_0^{(N-1)T} x(t) e^{-jwt} dt \cong \sum_{n=0}^{N-1} x(nT) e^{-jwnT} T$$

Let us consider now a sampled version of the Fourier transform at the frequencies

$w = kw_0 = k\frac{2\pi}{TN}$, where k=0,1,…,N-1. Then,

$$X(jkw_0) \cong \sum_{n=0}^{N-1} x(nT) e^{-j\frac{k2\pi}{NT}nT} T = T \sum_{n=0}^{N-1} x(nT) e^{-j\frac{2\pi}{N}kn}$$

Thus,

$$F\{x\}|_{w=kw_0} \cong FFT\{x(nT)\}T$$

This approximation is valid as long as the integrand approximation by a piecewise constant function is reasonable. This implies that k must be small and *x(t)* should not change significantly inside any sampling interval of length *T*.

In a more precise formulation, the sampled signal has a Fourier transform

$$X_s(jw) = \frac{1}{2\pi} X(jw) * \frac{2\pi}{T} \sum_k \delta(w - kw_s) = \frac{1}{T} \sum_k X(j(w - kw_s))$$

where, as usual, $w_s = \dfrac{2\pi}{T}$. This is the familiar form of the sampled signal Fourier transform as a summation of shifted replicas of the signal Fourier transform. Of course, aliasing effects will occur if *X(jw)* extends beyond half the sampling frequency. On the other hand, using the Fourier transform definition on the sampled signal we find

$$X_s(jw) = F\left\{x(t)\sum_k \delta(t - nT)\right\} = \int \left[x(t)\sum_n \delta(t - nT)\right] e^{-jwt} dt$$

$$= \int \left[\sum_n x(nT)\delta(t - nT)e^{-jwnT}\right] dt = \sum_n x(nT)e^{-jwnT} \int \delta(t - nT) dt$$

$$= \sum_n x(nT)e^{-jwnT}$$

Again, evaluating this transform at a discrete set of frequencies $w = kw_0 = k\dfrac{2\pi}{TN}$, we get

$$X_s\left(jk\frac{2\pi}{TN}\right) = \sum_n x(nT)e^{-j\frac{2\pi}{N}kn} = FFT\{x(nT)\}$$

In other words, the FFT of the sampled signal is equal to the Fourier transform of the sampled signal evaluated at the frequencies kw_0, in the interval [0, w_s]. It will also be approximately equal to *X(jw)/T*, (at the corresponding discrete frequencies in the interval [0, w_s/2]) as long as any aliasing effects are small.

Finally, in the aperiodic case, a comparison of the DFT and DTFT definitions

$$X_{DTFT}(e^{j\Omega}) = \sum_{n=-\infty}^{\infty} x(n)e^{-j\Omega n} \quad \text{and} \quad X_{DFT}(k) = \sum_{n=0}^{N-1} x(n)e^{-j\frac{2\pi}{N}kn}$$

yields that $X_{DTFT}(e^{j\Omega})\Big|_{\Omega=2\pi k/N} = X_{DFT}(k)$, as long as x(n) is zero outside [0,N-1]

Notes:

There will be some aliasing due to the fact that a finite duration signal cannot be bandlimited. This implies that X(jw) is only approximately equal to the replicas in Xs(jw). The symmetry of the FFT is the reason why only the first half of the FFT points are shown in the plots.)

More Derivations: *The Fourier transform of the signal under consideration.*
In this experiment we defined the signal y(t) = sin(w_0t), for t in [0,T_{max}]. Consequently,

$$F\{\sin(w_0 t)\,pulse_{[0,T_{\max}]}(t)\} = \frac{1}{2\pi}\frac{\pi}{j}\left[\delta(w-w_0)-\delta(w+w_0)\right] * \left[\frac{2\sin(wT_{\max}/2)}{w}e^{-jwT_{\max}/2}\right]$$

$$= \frac{1}{j}\left[\frac{\sin((w-w_0)T_{\max}/2)}{w-w_0}e^{-j(w-w_0)T_{\max}/2} - \frac{\sin((w+w_0)T_{\max}/2)}{w+w_0}e^{-j(w+w_0)T_{\max}/2}\right]$$

This expression is used as a basis to compare the FFT-computed transform. The latter will include aliasing effects, attaining its theoretical value as the sampling interval approaches zero.

ASSIGNMENT

Experiment 1.

Implement sampling and reconstruction in Simulink with the following specifications.

- Zero-Order-Hold (ZOH) sampling
- x(t) = sin 70Hz for 1s time interval, amplitude=1
- Sampling frequencies: 80Hz, 400Hz, and 1000Hz.

1. Use SIMULINK to sample the x(t) with ZOH sampling method with different sampling frequencies. Show the frequency magnitude character of sampled signals with FFT. Compare and discuss your observations.

2. Design a discrete Butterworth filter of order 6 using MATLAB for reconstruction. Specify the proper cut-off frequency of the filter. Show the frequency magnitude character of reconstructed signals with FFT. Compare and discuss your observations.

Hint: the SIMULINK blocks below may be needed for the assignment.

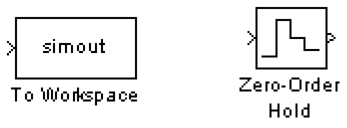


Fig2 SIMULINK blocks

'To Workspace' component writes its input to the workspace. The block writes its output to an array or structure that has the name specified by the block's Variable name parameter. Array or structure contains not only the input, but also some info about input. For example, the block's Variable name is simout, you can use the command below to access the input value.

```
Data=simout.signals.values;
```

Experiment 2.

Here we create a sinusoid with frequency 1 kHz and listen to the sound. We use a high sampling frequency of 44.1 kHz that represents a fairly good approximation of the continuous time signal.

Define the sampling frequency and the time vector. (8000 points / 44 kHz ~ 0.2 s worth of data) Plot the output and look at the first 0.004s. Play the sound.	`fs=44100; no_pts=8192;` `t=([1:no_pts]'-1)/fs;` `y1=sin(2*pi*1000*t);` `plot(t,y1);axis([0,.004,-1.2,1.2])` `disp('original');sound(y1,fs);`
Check the frequency domain signal. fr is the frequency vector and f1 is the magnitude of *F*{y1}.	`fr=([1:no_pts]'-1)/no_pts*fs; %in Hz` `fr=fr(1:no_pts/2);` `f1=abs(fft(y1));f1=f1(1:no_pts/2)/fs;`
F is the continuous time Fourier. (See derivation notes.) Compare the analytical continuous-time Fourier transform with its FFT computation. Notice the small amount of aliasing due to the fact that the truncated sinusoid is not bandlimited.	`frp=fr*2*pi;tmax=max(t);` `F1=1/j*sin((frp-1000*2*pi)*tmax/2)` `.*exp(j*(frp-1000*2*pi)*tmax/2)` `./(frp-1000*2*pi);` `F2=1/j*sin((frp+1000*2*pi)*tmax/2)` `.*exp(j*(frp+1000*2*pi)*tmax/2)` `./(frp+1000*2*pi);` `F=abs(F1-F2);` `loglog(fr,F,fr,f1);pause`

Experiment 3.

Now we sample the same sinusoid (1 kHz) at a lower sampling rate to observe the aliasing effects.

Sample the sin at 44/4 kHz. Compare the two output and play the sound. This may still sound OK because of the internal filtering of the soundcard.	`a=4; t_a=([1:no_pts/a]'-1)/fs*a;` `y_a=sin(2*pi*1000*t_a);` `plot(t,y1,t_a,y_a);` `axis([0,0.004,-1.2,1.2]);` `sound(y_a,fs/a);`
Check the frequency domain signal. Notice that the replicas of *F*{y_a} are now in the audible range (not shown). Also, the aliasing effects are in general more pronounced.	`fr_a=([1:no_pts/a]'-1)/no_pts*a*fs/a;` `fr_a=fr_a(1:no_pts/a/2);` `f_a=abs(fft(y_a));` `f_a=f_a(1:no_pts/a/2)/fs*a;` `loglog(fr,F,fr,f1,fr_a,f_a);`
Use the interp1 function to get the ZOH version of the signal at 44 kHz. The ZOH version is how the signal would sound with a 44/a kHz D/A converter. Its difference from the original is significant in both the time and frequency domain and its reproduction is quite poor.	`y_n = interp1(t_a,y_a,t,'nearest','extrap');` `f_n=abs(fft(y_n));f_n=f_n(1:no_pts/2)/fs;` `plot(t,y1,t,y_n);` `disp('original');sound(y1,fs);` `disp('nearest');sound(y_n,fs);` `loglog(fr,F,fr,f1,fr,f_n);`

Experiment 4.

Here we use different digital filters to “interpolate” the values of the low rate signal and convert it to a high rate signal. This does not overcome any aliasing effects that occurred at sampling but improves the reproduction by reducing the undesirable properties of the low rate ZOH. (It does require a better D/A converter!) In the following table we concentrate on the implementation of a digital filter by approximating the impulse response of an ideal low-pass.

Generate an “Upsampled” version of the low-rate signal at 44 kHz.	`y_u=y1*0;k=1:a:no_pts;y_u(k)=y_a;` `f_u=abs(fft(y_u));`

The upsampled signal contains several replicas of the original Fourier transform within the sampling frequency and it is not an approximation of the original signal.	`f_u=f_u(1:no_pts/2)/fs*a;`
	`plot(t,y1,t,y_u);axis([0,0.004,-1.2,1.2])`

However, the original signal can be "recovered" after low-pass filtering. Here, the filter is defined in terms of its impulse response and the output is computed as a convolution sum. Notice that the impulse response HS is 2000 points long yielding an 1000 points delay in the sound reproduction. This is significant but not necessarily impractical.	`hs=sin(3000*2*pi*t)/pi./t; i=1000:-1:2;`
	`HS=[hs(i);3000*2;hs(2:1000)]*a/fs;`
	`y_f=conv(HS,y_u);`
	`y_f=y_f(1000:999+no_pts);`
	`f_f=abs(fft(y_f));f_f=f_f(1:no_pts/2)/fs;`
	`plot(t,y1,t,y_f);axis([0,0.004,-1.2,1.2])`
	`disp('original');sound(y1,fs);`
	`disp('filtered-upsampled');sound(y_f,fs);`
	`loglog(fr,F,fr,f1,fr,f_f);`

Notes: Given a discrete-time signal *y(n)*, the upsampled signal (by *a*) is defined as the signal *z(n)* such that

$$z(n)=\begin{cases} y(k) & if\ \ n=ak \\ 0 & otherwise \end{cases}$$

Equivalently, we may think of *z(n)* as the signal produced by sampling the continuous time signal *y(t)* with the impulse train $p_u(t)=\sum b_n\delta(t-n\tfrac{T}{a})$, where b_n = 1, if *n/a* is an integer, and 0 otherwise.
Clearly, the upsampling impulse train is identical with the original delta train $p(t)=\sum\delta(t-nT)$ and, therefore, the continuous-time sampled signals *y(t)p(t)* and $y(t)p_u(t)$ are the same, and have the same Fourier transforms. The latter, however, has a smaller sampling time (larger sampling frequency), implying that the FFT of the discrete-time sequence will contain several replicas of *F{y}*. The upsampling process does not create any new information about the time signal but allows for the implementation of better low-pass filters in discrete time.

Experiment 5.

Equalizers are often used in audio equipment to amplify (or attenuate) frequency bands in an effort to improve the quality of reproduction of the original signal. Their objective is to cancel (or invert) the audio signal distortion caused by equipment limitations (amplifier, speaker, media) or the environment where the signal is reproduced.
In this experiment we use digital low-pass filters to implement a "frequency equalizer." The test signal is the standard Windows sound "Tada." The attached Matlab program loads the sound (make sure its copy is in the working directory) and filters it with three different low-pass filters. Then, by taking different linear combinations of the resulting signals we can amplify or attenuate the energy of the signal in the desired frequency band(s).

```
disp('Experiment 2')
%Exp.1: Signal generation, Fourier Transform
fs=44100; %sample sin at 44.1 kHz
no_pts=2*8192;
t=([1:no_pts]'-1)/fs;
y1=sin(2*pi*1000*t);
plot(t,y1);axis([0,0.004,-1.2,1.2]);title('The signal y(t)');pause
disp('original');sound(y1,fs);pause(no_pts/fs*2)
fr=([1:no_pts]'-1)/no_pts*fs;fr=fr(1:no_pts/2); %in Hz
f1=abs(fft(y1));f1=f1(1:no_pts/2)/fs;
frp=fr*2*pi;tmax=max(t);
F1=1/j*sin((frp-1000*2*pi)*tmax/2).*exp(j*(frp-1000*2*pi)*tmax/2)./(frp-
1000*2*pi);
F2=1/j*sin((frp+1000*2*pi)*tmax/2).*exp(j*(frp+1000*2*pi)*tmax/2)./(frp+1000*2*
pi);
F=abs(F1-F2);
loglog(fr,F,fr,f1);title('Frequency Domain magnitudes: F[y], FFT[y]');pause

disp('Experiment 3')
%Exp.2: Low-frequency sampling and reconstruction, Fourier Transform
a=4; %sample sin at 44.1/a kHz; use a = power of 2
t_a=([1:no_pts/a]'-1)/fs*a;
y_a=sin(2*pi*1000*t_a);
plot(t,y1,t_a,y_a);axis([0,0.004,-1.2,1.2]);
title('y(t) sampled at high and low rates');pause
disp('original');sound(y1,fs);pause(no_pts/fs*2)
disp('low-freq sampling');sound(y_a,fs/a);pause(no_pts/fs*2)
% this sounds OK because of the internal filtering
fr_a=([1:no_pts/a]'-1)/no_pts*a*fs/a;fr_a=fr_a(1:no_pts/a/2);
f_a=abs(fft(y_a));f_a=f_a(1:no_pts/a/2)/fs*a;
loglog(fr,F,fr,f1,fr_a,f_a);title('Magnitudes of F[y], FFT[y], FFT[y_a]');pause

%bypass internal filters and emulate 44.1/a kHz D/A
y_n = interp1(t_a,y_a,t,'nearest','extrap');
        % REM: nearest neighbor interpolation ~ shifted ZOH
f_n=abs(fft(y_n));f_n=f_n(1:no_pts/2)/fs;
plot(t,y1,t,y_n);axis([0,0.004,-1.2,1.2])
title('Original and low-rate ZOH signals');pause
disp('original');sound(y1,fs);pause(no_pts/fs*2)
disp('nearest (~ZOH)');sound(y_n,fs);pause(no_pts/fs*2)
loglog(fr,F,fr,f1,fr,f_n);title('Magnitudes of F[y], FFT[y],
FFT[y_{a,ZOH}]');pause

disp('Experiment 4')
%Exp.3: Digital filtering and reconstruction from the low-rate sampled signal
%linear interpolation
y_l = interp1(t_a,y_a,t,'linear','extrap');
f_l=abs(fft(y_l));f_l=f_l(1:no_pts/2)/fs;
plot(t,y1,t,y_l);axis([0,0.004,-1.2,1.2])
title('Original and lin.-interpolated low-rate signals');pause
disp('original');sound(y1,fs);pause(no_pts/fs*2)
disp('lin.interp.');sound(y_n,fs);pause(no_pts/fs*2)
loglog(fr,F,fr,f1,fr,f_l);title('Magnitudes of F[y], FFT[y], FFT[y_l]');pause

%upsampling
y_u=y1*0;k=1:a:no_pts;y_u(k)=y_a;
f_u=abs(fft(y_u));f_u=f_u(1:no_pts/2)/fs*a;
plot(t,y1,t,y_u);axis([0,0.004,-1.2,1.2])
title('Original and upsampled signals');pause
disp('original');sound(y1,fs);pause(no_pts/fs*2)
disp('upsampled');sound(y_u,fs);pause(no_pts/fs*2)
loglog(fr,F,fr,f1,fr,f_u);title('Magnitudes of F[y], FFT[y], FFT[y_u]');pause

%3kHz lowpass filter (1000*2-1 point approximation)
hs=sin(3000*2*pi*t)/pi./t; % this gives a division by zero warning
i=1000:-1:2;HS=[hs(i);3000*2;hs(2:1000)]*a/fs;  % set the correct value at 0
y_f=conv(HS,y_u);
%eliminate initial/final points (initialization) of 1000 samples = 2.27e-2 s
(significant)
y_f=y_f(1000:999+no_pts);
```

```
f_f=abs(fft(y_f));f_f=f_f(1:no_pts/2)/fs;
plot(t,y1,t,y_f);axis([0,0.004,-1.2,1.2])
title('Original and filtered upsampled signals');pause
disp('original');sound(y1,fs);pause(no_pts/fs*2)
disp('filtered-upsampled');sound(y_f,fs);pause(no_pts/fs*2)
loglog(fr,F,fr,f1,fr,f_f);title('Magnitudes of F[y], FFT[y], FFT[y_f]');pause

%use a butterworth filter
A=[ 1.0000e+000 -7.5419e+000  2.5836e+001 -5.2896e+001  7.1625e+001 -
6.6986e+001 ...
    4.3797e+001 -1.9759e+001  5.8842e+000 -1.0441e+000  8.3813e-002];
B=[  2.2721e-008  2.2721e-007  1.0225e-006  2.7265e-006  4.7714e-006  5.7257e-
006 ...
    4.7714e-006  2.7265e-006  1.0225e-006  2.2721e-007  2.2721e-008];
td=-3.7885e-4;
% comment the following two calls if the functions are not available
[B,A] = butter(10,2700/22050);
[m,p]=dbode(B,A,1/fs,2*pi*1000); td=p/180/2/1000; % find delay at 1kHz
y_b=dlsim(B*a,A,y_u);
f_b=abs(fft(y_b));f_b=f_b(1:no_pts/2)/fs;
plot(t,y1,t+td,y_b);axis([0,0.004,-1.2,1.2]);
title('Original and butterworth-filtered upsampled signals');pause
disp('original');sound(y1,fs);pause(no_pts/fs*2)
disp('b-filtered upsampled');sound(y_b,fs);pause(no_pts/fs*2)
loglog(fr,F,fr,f1,fr,f_b);title('Magnitudes of F[y], FFT[y], FFT[y_b]');pause



%Exp.5: Equalization
[y,f]= wavread('tada');
y=y(:,1); % keep only one channel
ly=length(y); ty=([1:ly]'-1)/f;
w=([1:ly]'-1)/ly*f;w=w(1:ly/2); %in Hz
fy=abs(fft(y));fy=fy(1:ly/2)/f; % FFT
    soundsc(y,f)
loglog(w,fy);title('Magnitude of F[y]');pause

i=2:500;i_=500:-1:2; L=500:499+ly; % define 3 1000-point lowpass filters
hs1=sin(700*2*pi*ty)/pi./ty;HS1=[hs1(i_);700*2;hs1(i)]/f;
hs2=sin(2000*2*pi*ty)/pi./ty;HS2=[hs2(i_);2000*2;hs2(i)]/f;
hs3=sin(5000*2*pi*ty)/pi./ty;HS3=[hs3(i_);5000*2;hs3(i)]/f;
z1=conv(HS1,y);z1=z1(L);
z2=conv(HS2,y);z2=z2(L);
z3=conv(HS3,y);z3=z3(L);
done=0;
while done ==0
    coef=input('enter equalization coefficients [low, mid-low, mid-hi, hi] ');
    if isempty(coef),coef=[1 1 1 1];end
    yx=[z1 z2-z1 z3-z2 y-z3]*coef';
    soundsc(yx,f)
    fx=abs(fft(yx));fx=fx(1:ly/2)/f;
    loglog(w,fx);title('Equalized signal');pause
    done=input('Done 1=y, [0]  ');
    if isempty(done);done=0;end
end
```

EEE 304

Lab Exercise 3

Discrete-time Modulation and Demodulation

1. Download the file y.wav. Try to play this audio file.
2. load the sound file into Matlab by typing:

   [y,fs,nbits]=wavread('y.wav');

   This loads three variables into you Matlab workspace:

   y is the audio data,

   fs is the sample rate (11025 in this case), and

   nbits is the number of bits used to represent each sample in **y**.

ASSIGNMENT

1. Plot the frequency domain magnitude of the audio data, obtained by FFT. Discuss how to generate the frequency vector (Hz) for FFT.
2. Build a Simulink model to amplitude-modulate **y** by a cosine carrier using a suitable carrier frequency and then perform synchronous demodulation to restore the spectrum to its original shape. Show the waveforms of the modulated and de-modulated audio data.
3. Select an appropriate frequency for the carrier and justify your selection.
4. Design a low-pass filter with an appropriate cutoff frequency and order and justify your selections.
5. Plot the frequency–magnitude of modulated and de-modulated data and explain the modulation and de-modulation effects on the original spectrum.
6. If you need to save the processed data into a file, you can use the command **wavwrite(yr,fs,nbits,'yr.wav');**

   where yr is the audio data, fs is the sampling frequency, nbits

   yr is the audio data,

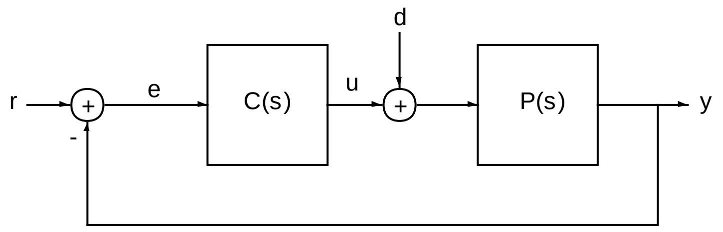
   fs is the sample rate (11025 in this case), and

   nbits is the number of bits used to represent each sample in **y**.

   'c:\yr.wav' is the directory and file name for the audio data.

Lab 4: Linear Feedback

Problems 1-3 will use the following diagram:



Problem 1:

Plant: $P(s) = \frac{1}{s}$ (e.g., a normalized description of car velocity with force as the input; this can represent a cruise control system.)

Controller: $C(s) = K\frac{Ts+1}{s}$ (Verify that this is a PI controller)

(a) Find the transfer function from the disturbance d to the output y. Show that if the input d is a step disturbance and $C(0) = \infty$, the effect of d on y approaches zero as $t \to \infty$. (*Hint: Assume K,T>0. Use the final value theorem.*) Demonstrate this result by choosing several different values of K>0 and T>0 and plotting the step response. (*Hint: Vary K,T by orders of magnitude.*)

(b) Design a controller (select values of T and K) such that the following specifications are met:
   a. Target crossover frequency = 1 (approximately equal to the closed-loop bandwidth).
   b. Target phase margin is 60°

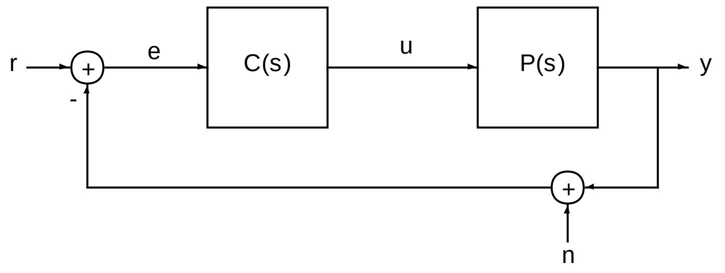
   (*Hint: K = 0.5, T =1.73.*)

(c) Using MATLAB plot the frequency and step responses from the reference input r to the output y and from the disturbance d to the output y.

(d) Compare the results in (c) with a proportional-only controller (C(s)=K) designed to have the same closed loop crossover frequency. (*Hint: K=1.*) This demonstrates the need for an integrator in the controller to compensate for constant disturbances.

Problem 2:

This problem explores the effect of sensor noise on the input to the plant. Plant inputs are typically provided by actuators, e.g., valves, motors, and other mechanical components. High-frequency inputs to these actuators cause more stress, leading to shorter actuator lifetime and higher probability of actuator failure. The sensors that measure the output and feed it back to the controller are prone to high-frequency measurement noise. This problem will show the trade-off between bandwidth and high-frequency noise attenuation. The following diagram shows how sensor noise enters into the closed-loop system:



Given the plant $P(s) = \frac{1}{s}$, design a controller as in Pr.1.b, but for a target crossover frequency of 10. Compare the frequency responses of the plant input to sensor noise for this controller and the controller of Pr.1, designed to yield crossover 1.

Problem 3:

Plant: $P(s) = \dfrac{1}{s(as+1)}$

This problem studies the effect of unmodeled lag on the closed-loop system behavior. The above transfer function represents a perturbation of the original system (studied in Pr.1). The term *1/as+1* can be viewed as a 1st order lag introduced by unmodeled actuator dynamics (engine; e.g., throttle to torque/force response) or sensor dynamics.

The objective of this problem is to demonstrate that when such lags alter the system significantly around the crossover frequency, the closed-loop may become unstable and the controller must be redesigned. Notice that the achievable performance may no longer be the same.

(a) Consider the cases where a = 0.1, 1, 10 and evaluate the closed-loop response (step, bode) for the same controller designed in Pr.1.b. Compare with the nominal response of Pr.1.d.

(b) For the case a = 10, redesign the controller to achieve 60^o phase margin at the highest possible frequency. (This is roughly 0.036, with a maximum angle contribution from the zero of 80^o.

Problem 4:

Constraints on the achievable bandwidth can also be imposed by other forms of lag, e.g. if the plant contains a delay e^{-sT_d}. Delays may exist because of physical reasons (transport delay, computational delay) or simply as approximations of higher order dynamics. The Pade approximation is used to write a delay as a rational transfer function for which the usual tools are applicable. A simple Pade approximation for this delay is $e^{-sT_d} \approx \dfrac{1-sT_d/2}{1+sT_d/2}$. Show that if the target bandwidth is high and the delay is unknown, the nominal design can be unstable. (*Hint: At high frequencies the angle of the plant approaches* -270° *and the controller cannot make it greater than* -180° *to achieve closed loop stability*.)

Problem 5:

Controller discretization is a topic of increased importance whenever the sampling rate is constrained by physical or computational reasons and becomes comparable to closed-loop bandwidth. For example, when a cheap microprocessor with low computing power is used, or when sensing is a time-consuming process. In such cases, the method of controller discretization can have significant impact on the closed-loop response. In this problem, however, we are only concerned with the deterioration of the response as the sampling rate is decreased.

Discretize the controller of Pr.1, using sample times 0.01, 0.1, 1, 10 and compare the resulting closed-loop responses with the continuous-time case.

Hint: In MATLAB, a suitable command is

```
Ts=0.1;step(feedback(H0*C0,1),feedback(c2d(H0,Ts)*c2d(C0,Ts),1))
```

Notice that c2d(H0,Ts)*c2d(C0,Ts) is not the same as c2d(H0*C0,Ts) (why?).

The 1st and 6th order Butterworth filters.

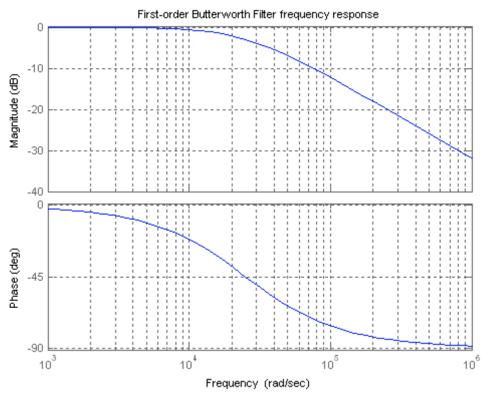


Fig1 bode plot of 1st order filter

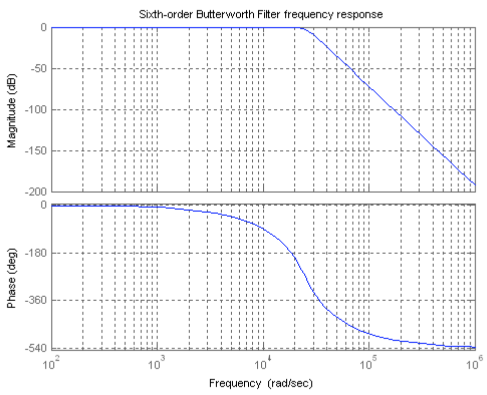


Fig2 bode plot of 6th order filter

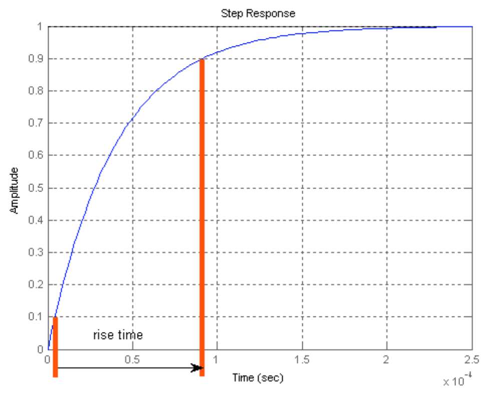


Fig3 step response of 1st order filter

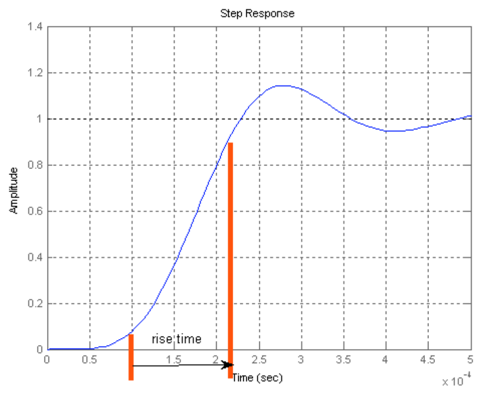


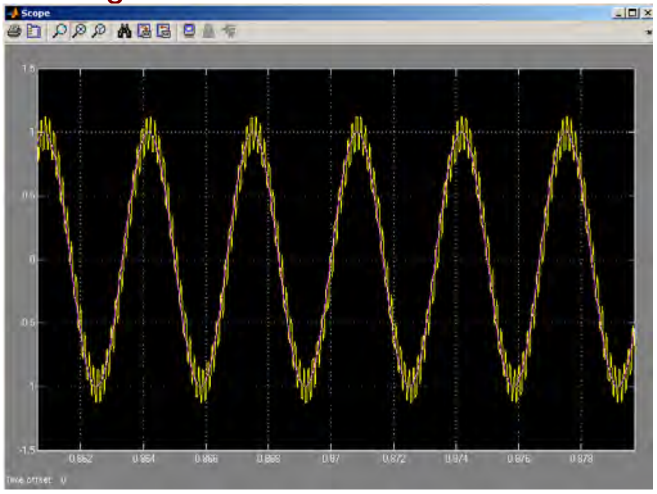
Fig4 step response of 6th order filter

Fig3 and 4 show that rise time of 1st order filter is slightly less (but comparable) than that of 6th order filter.

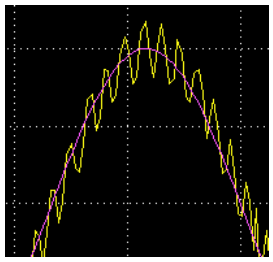
SIMULINK

x(t) = sin 300Hz + sin 21.413kHz, for 1s time interval.

Analog filter



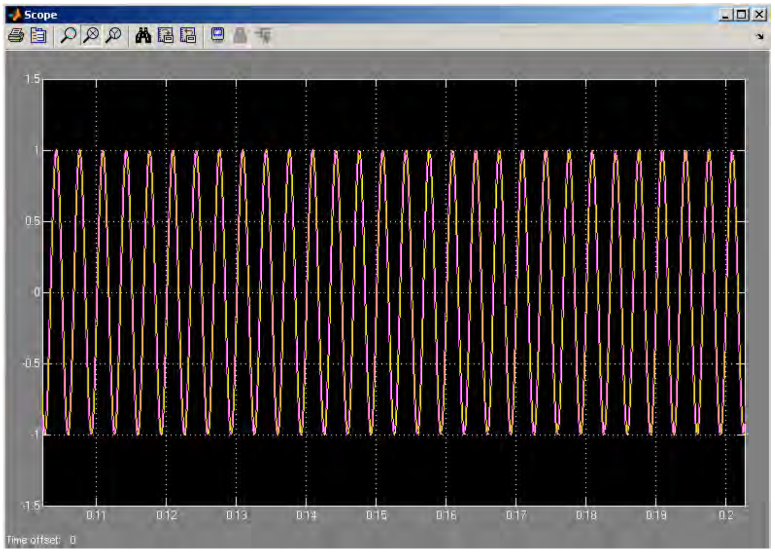
(a)



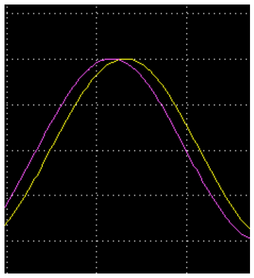
(b)

Fig 5 (a) the output of 1st order Butterworth filter

(b) The local details of the output



(a)



(b)

Fig 6(a) the output of 6th order Butterworth filter

(b) The local details of the output

Fig5 shows that there is high frequency left after filtering with 1st order filter.Fig6 shows that the 6th order filter removes high frequency . The reasons are that frequency magnitude of the 6th order filter attenuate more quickly than that of the 1st order filter after cutoff frequency..

Digital filter

Sampling rate: 10 KHz

With sampling rate 10KHz, the spectrum of sin21.413KHz will be shifted by 10KHz periodically. So the resulted spectrum contain frequencies at 1.413KHz, 11.413KHz, 21.413KHz and 31.413Hz and so on. With the cutoff frequency 4KHz, only 1.413KHz will pass while others will be removed. The same happens to sin 300Hz. So we can see there is high frequency left in Fig 7.

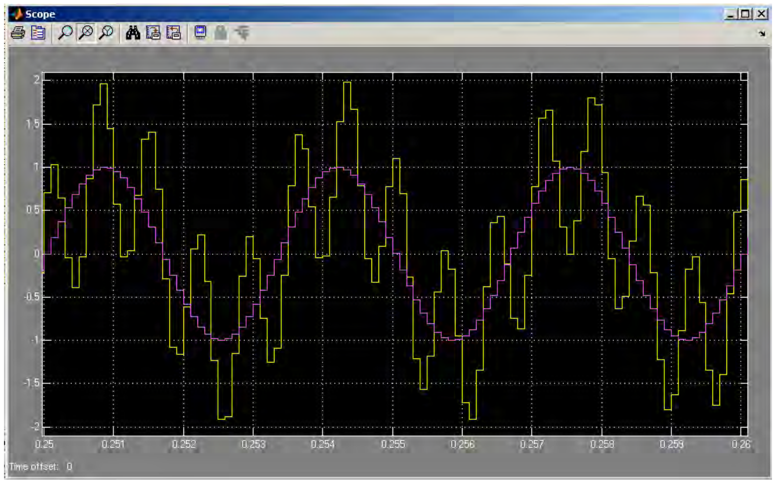


Fig 7 the output of digital filtering with 10 KHz sampling rate

Sampling rate: 100 KHz

100 KHz is enough for sampling the sin21.413KHz and sin300Hz, therefore only sin300Hz will pass while others are removed.

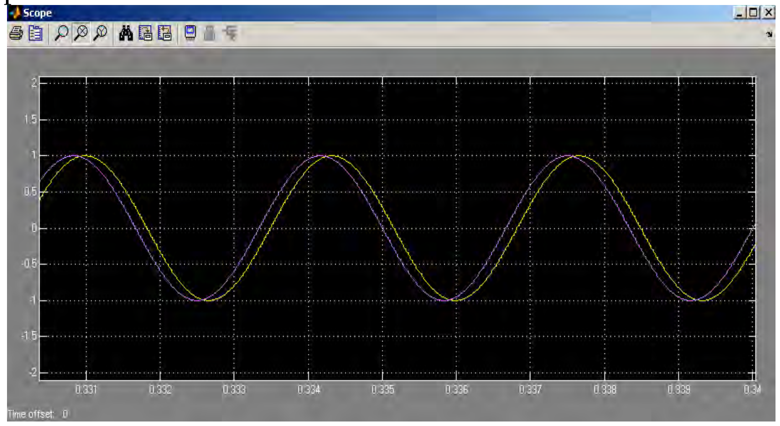


Fig 8 the output of digital filtering with 100 KHz sampling rate