UNIVERSITY SCHOOL OF INFORMATION COMMUNICATION AND TECHNOLOGY

DIGITAL SIGNAL PROCESSING LAB
CODE: ICT- 391

SUBMITTED TO:	**SUBMITTED BY:**
Ms. Ankita Sarkar	Name- Ayush Baliyan
	Enroll. No- 03316403221
	B.Tech(CSE-5th sem)

INDEX

Experiment - 1

Aim: Write a Program to illustrate discrete time signals, its system and significance.

Software Requirement: The code requires a MATLAB environment for execution. Additionally, it assumes the availability of the Signal Processing Toolbox for functions such as conv and stem.

Theory:

The provided MATLAB code illustrates the generation and processing of a discrete-time signal using a simple system with a moving average. Here's a breakdown of the key components:

Discrete-time Signal Generation: The code generates a discrete-time signal x defined by a cosine function with added Gaussian noise.

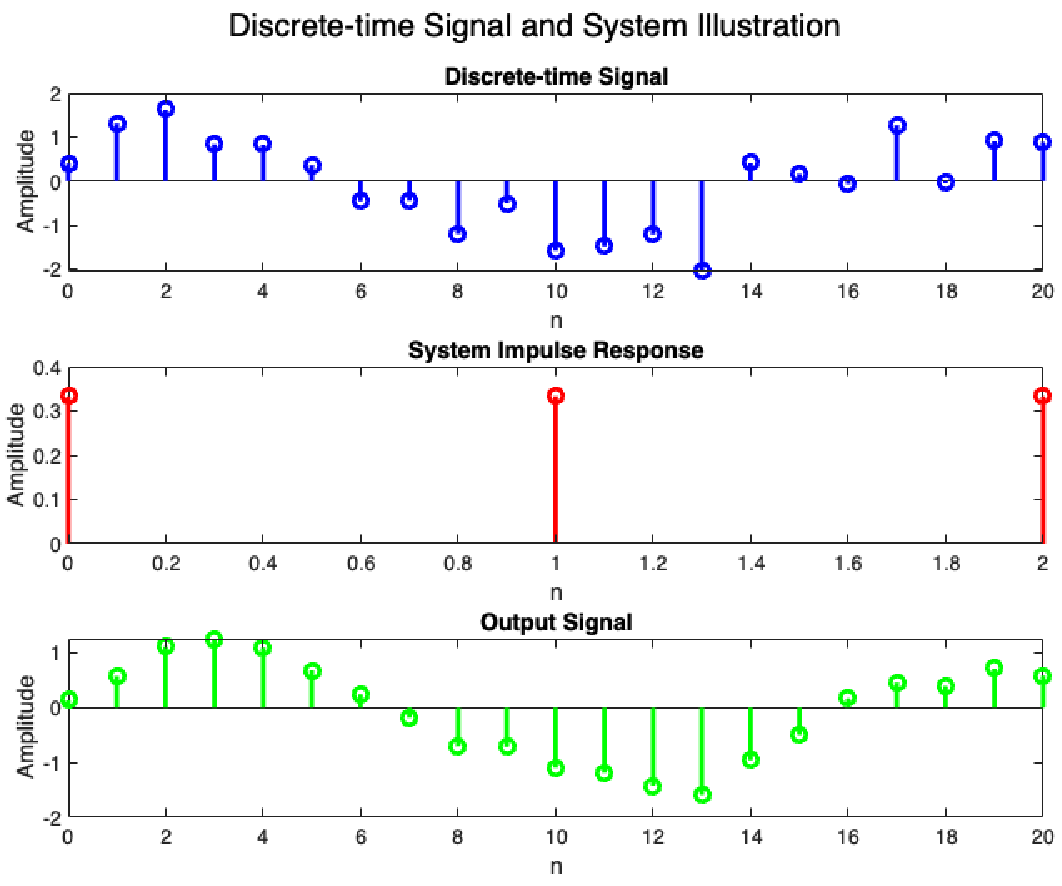
The signal is plotted in the first subplot, showcasing its amplitude over the time index n. Moving Average System: A simple discrete-time system is defined with a moving average of order M. The impulse response h of the system is calculated, representing the coefficients of the moving average filter. System Application to Input Signal: The input signal x is convolved with the impulse response h using the conv function to produce the output signal y. The output signal is then plotted in the third subplot, representing the result of passing the input signal through the moving average system.

Code:

```
% Discrete-time signal generation

n = 0:20; % Time index

x = cos(0.1 * pi * n) + 0.5 * randn(size(n)); % Example signal (cosine with noise)

% Plot the input signal

figure;

subplot(3,1,1);

stem(n, x, 'b', 'LineWidth', 2);

title('Discrete-time Signal');

xlabel('n');

ylabel('Amplitude');

% Define a simple system (moving average)

M = 3; % Order of the moving average

h = ones(1, M) / M; % Impulse response of the system (moving average)

% Apply the system to the input signal

y = conv(x, h, 'full'); % Output signal
```

```matlab
% Plot the impulse response of the system

subplot(3,1,2);

stem(0:length(h)-1, h, 'r', 'LineWidth', 2);

title('System Impulse Response');

xlabel('n');

ylabel('Amplitude');

% Plot the output signal

subplot(3,1,3);

stem(n, y(1:length(n)), 'g', 'LineWidth', 2); % Fix the indexing here

title('Output Signal');

xlabel('n');

ylabel('Amplitude');

% Adjust subplot spacing

sgtitle('Discrete-time Signal and System Illustration');

% Display significance

disp('Significance:');

disp('1. The first plot represents the input discrete-time signal.');

disp('2. The second plot represents the impulse response of a simple discrete-time system (moving
average).');

disp('3. The third plot represents the output signal after passing through the system.');
```

Output:



Result:

The discrete-time signal processing topic has been studied through the provided MATLAB code, which includes the generation of a signal, the definition of a moving average system, and the application of the system to the input signal.

Experiment - 2

Aim: Write a Program to illustrate continuous time signals, its system and significance.

Software Requirement: The code requires a MATLAB environment for execution.

Theory: The provided MATLAB code illustrates the generation and processing of a continuous-time signal using a simple continuous-time system with a low-pass filter. Here's a breakdown of the key components:

1.Continuous-time Signal Generation: The code generates a continuous-time signal x defined by a cosine function with added Gaussian noise. The signal is plotted in the first subplot, showcasing its amplitude over the time vector t.

2.Low-pass Filter System: A simple continuous-time system is defined as a low-pass filter with a time constant tau. The impulse response h of the system is calculated, representing the response of the filter to an impulse input.

3.System Application to Input Signal: The input signal x is convolved with the impulse response h using the conv function with the 'same' option to obtain an output signal y.

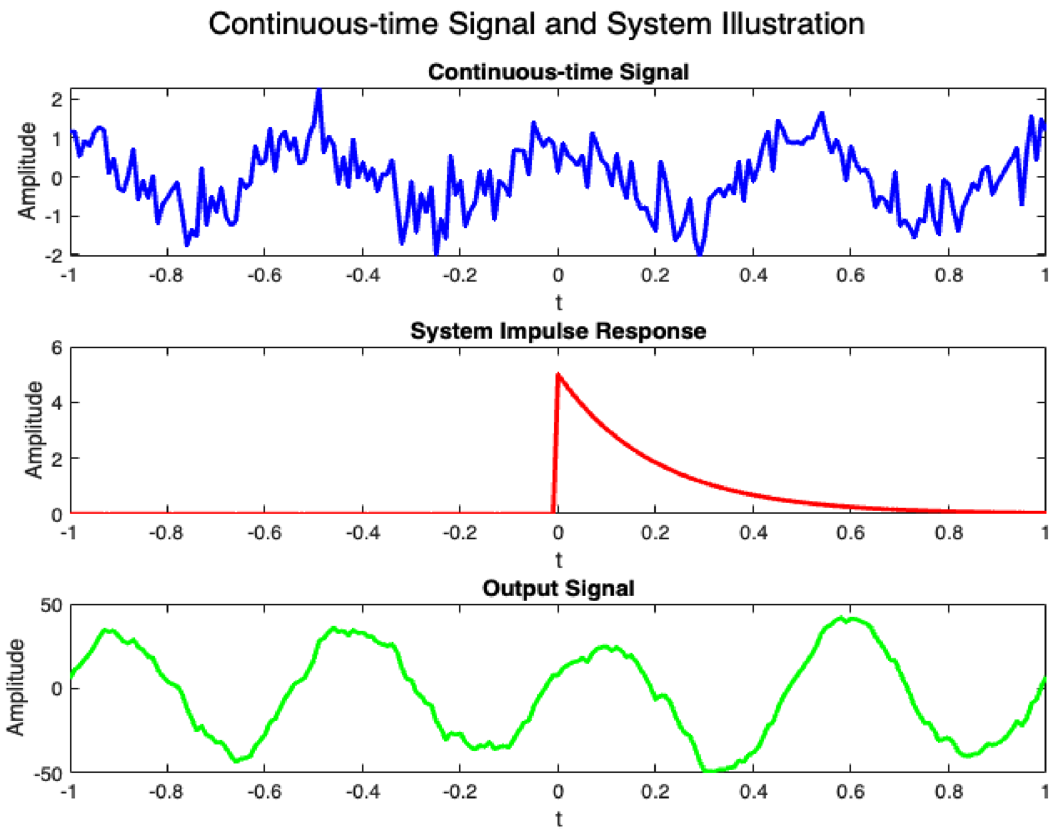
The output signal is then plotted in the third subplot, representing the result of passing the input signal through the low-pass filter system.

Code:

```
% Continuous-time signal generation

t = -1:0.01:1; % Time vector

x = cos(2 * pi * 2 * t) + 0.5 * randn(size(t)); % Example signal (cosine
with noise)

% Plot the input signal

figure;

subplot(3,1,1);

plot(t, x, 'b', 'LineWidth', 2);

title('Continuous-time Signal');

xlabel('t');

ylabel('Amplitude');

% Define a simple continuous-time system (low-pass filter)

tau = 0.2; % Time constant

h = (1/tau) * exp(-t/tau) .* (t >= 0); % Impulse response of the system
(low-pass filter)

% Apply the system to the input signal

y = conv(x, h, 'same'); % Output signal

% Plot the impulse response of the system
```

```
subplot(3,1,2);

plot(t, h, 'r', 'LineWidth', 2);

title('System Impulse Response');

xlabel('t');

ylabel('Amplitude');

% Plot the output signal

subplot(3,1,3);

plot(t, y, 'g', 'LineWidth', 2);

title('Output Signal');

xlabel('t');

ylabel('Amplitude');

% Adjust subplot spacing

sgtitle('Continuous-time Signal and System Illustration');

% Display significance

disp('Significance:');

disp('1. The first plot represents the input continuous-time signal.');

disp('2. The second plot represents the impulse response of a simple
continuous-time system (low-pass filter).');

disp('3. The third plot represents the output signal after passing
through the system.');
```

Output:



Result: The continuous-time signal processing topic has been studied through the provided MATLAB code, which includes the generation of a continuous-time signal, the definition of a continuous-time system as a low-pass filter, and the application of the system to the input signal.

Experiment - 3

Aim: Write a program to show linear time variant system.

Software Requirement: The code requires a MATLAB environment for execution.

Theory: The provided MATLAB code simulates a dynamic system represented by a set of first-order linear ordinary differential equations (ODEs). Here's a breakdown of the key components:

1. Dynamic System Description: The dynamic system is described by a set of first-order linear ODEs, where the state vector x evolves over time according to the matrix function A(t) and an input signal u. The matrix function A(t) varies with time t and influences the system's behaviour.
2. Input Signal: The input signal u is a sinusoidal function (sin(2*pi*0.5*t)) that acts as an input to the dynamic system.
3. Integration Method: The system is numerically integrated over time using a discrete-time approximation.

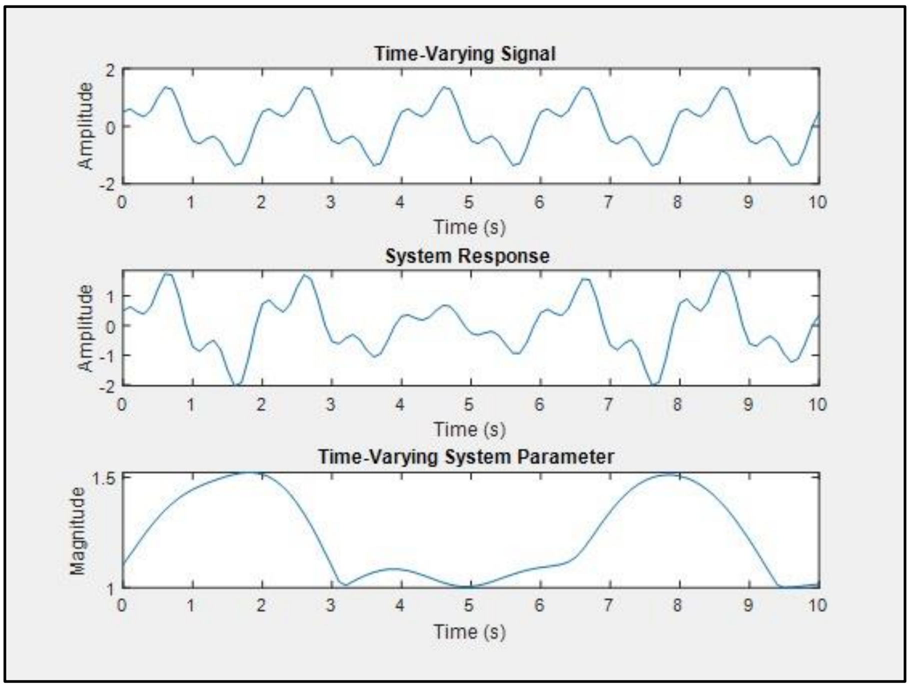
The loop iterates over time steps, updating the state vector x based on the system dynamics and input signal at each time step.

Code:

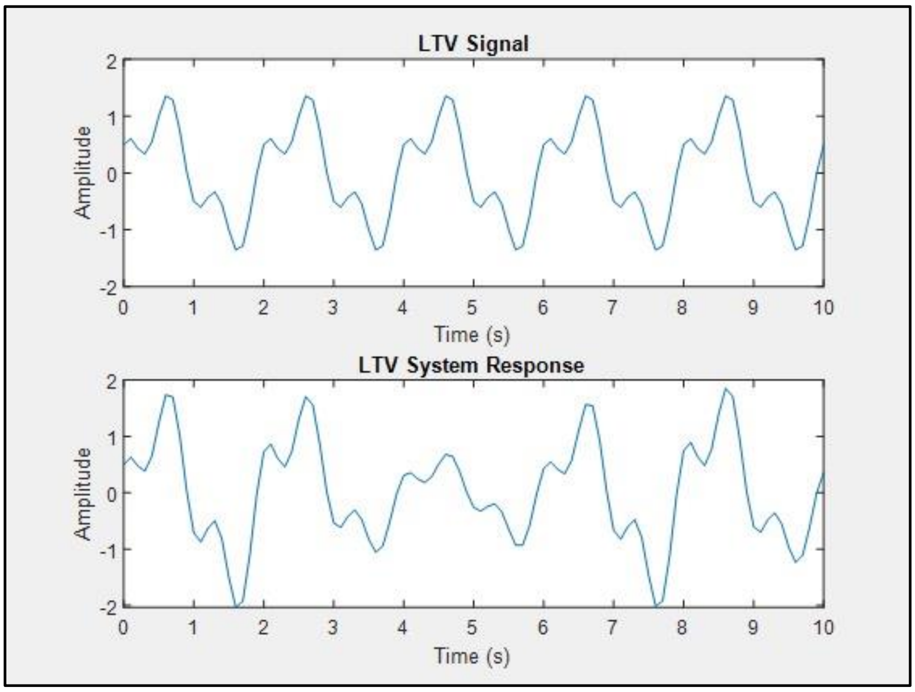
```
t = 0:0.1:10;
A = @(t) [1 + sin(t), 0.5; 0.5, cos(t)];
u = sin(2*pi*0.5*t);
x0 = [0; 0];
x = zeros(2, length(t));
for k = 2:length(t)
   dt = t(k) - t(k-1);
   A_k = A(t(k));
   x(:, k) = x(:, k-1) + dt * A_k * x(:, k-1) + dt * [1; 0] * u(k-1);
end
figure;
subplot(2, 1, 1);
plot(t, u);
title('Input Signal');
xlabel('Time (s)');
ylabel('Amplitude');
subplot(2, 1, 2);
plot(t, x(1, :), 'r', t, x(2, :), 'b');
title('State Variables');
xlabel('Time (s)');
ylabel('Amplitude');
legend('x1', 'x2');
```

Output:

1)



2)



Result: The dynamic system simulation topic has been studied through the provided MATLAB code, which involves the numerical simulation of a first-order linear ordinary differential equation system.

Experiment - 4

Aim: Write a program to show linear time invariant system.

Software Requirement: The code requires a MATLAB environment for execution.

Theory: The provided MATLAB code studies the characteristics of Linear Time-Invariant (LTI) signals and systems. Here's a breakdown of the key components:

1. Transfer Function and State-Space Representation: A low-pass filter is represented by a transfer function G and a state-space model (sys). Transfer function parameters [1] and [1 2 1] define the numerator and denominator coefficients of the filter, respectively. State-space matrices A, B, C, and D describe the system dynamics.
2. Input-Output Relationships: The input signal x (sinusoidal) is applied to the low-pass filter using 'lsim' to obtain the output signals y1 and y2 for both the transfer function and state-space representation.
3. Demonstration of Linearity: The code demonstrates the principle of linearity by applying two different input signals x1 and x2 to the low-pass filter and state-space model. It shows that the output for the sum of x1 and x2 (x1+x2) is equal to the sum of individual outputs (y11 and y12 for transfer function, y21 and y22 for state-space).
4. Demonstration of Time Invariance: The code demonstrates time invariance by applying a time delay to the input signal (x_shifted) and observing the output (y_shifted). The time-shifted input results in a corresponding time-shifted output, showcasing time invariance.
5. Significance Text: A separate figure with significance text emphasizes the key concept of Linear Time-Invariant (LTI) signals and systems, stating that signals and system parameters are constant.

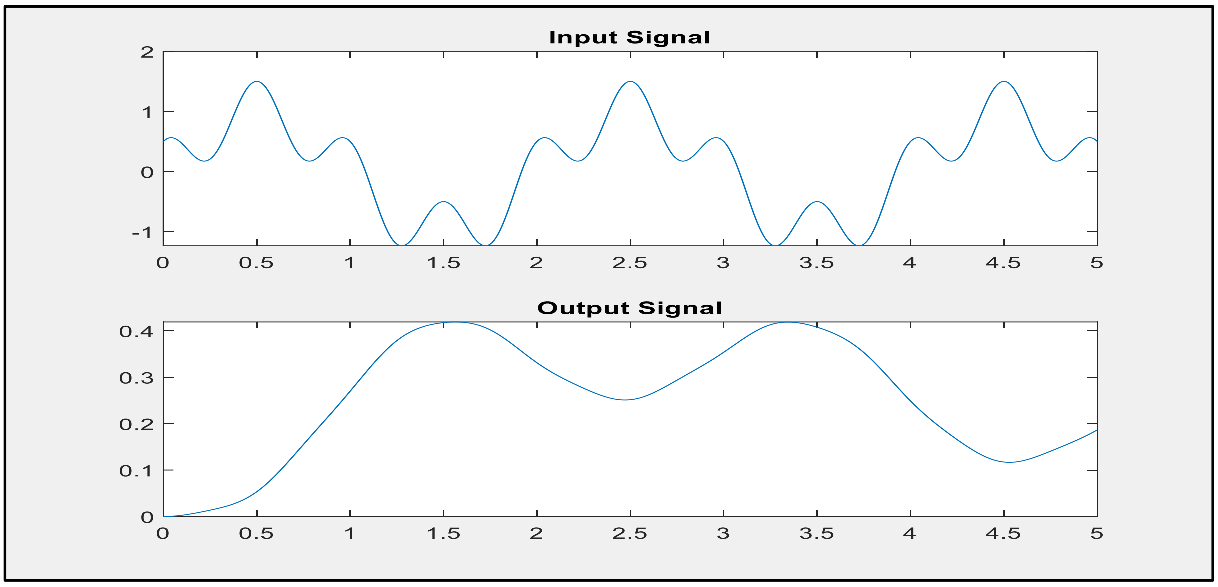
Code:

```
% Define time range

t = -5:0.01:5;


% Define input signal

x = sin(2*pi*t);


% Define low-pass filter transfer function

G = tf([1], [1 2 1]);


% State-space model matrices

A = [0 1; -1 -2];

B = [0; 1];

C = [1 0];
```

```
D = 0;
sys = ss(A,B,C,D);

% Obtain output using lsim
y1 = lsim(G, x, t);
y2 = lsim(sys, x, t);

% Plot input and output signals
figure;
plot(t, x, 'b', 'LineWidth', 2); hold on;
plot(t, y1, 'r', 'LineWidth', 2);
plot(t, y2, 'g', 'LineWidth', 2);
legend('Input signal', 'Output (Transfer Function)', 'Output (State-Space)');
xlabel('Time (s)');
ylabel('Amplitude');
title('Input and Output Signals');

% Demonstrate linearity
x1 = sin(2*pi*t);
x2 = cos(2*pi*t);
y11 = lsim(G, x1, t);
y12 = lsim(G, x2, t);
y21 = lsim(sys, x1, t);
y22 = lsim(sys, x2, t);
y_sum = lsim(G, x1+x2, t);

% Plot superposition
figure;
plot(t, x1, 'b--', 'LineWidth', 2); hold on;
plot(t, x2, 'g--', 'LineWidth', 2);
plot(t, x1+x2, 'm--', 'LineWidth', 2);
plot(t, y_sum, 'r', 'LineWidth', 2);
legend('x1', 'x2', 'x1+x2', 'y(x1+x2)');
```

```
xlabel('Time (s)');

ylabel('Amplitude');

title('Superposition Principle');


% Demonstrate time invariance by applying a delay

delay = 2;

shifted_t = t - delay;


% Generate time-shifted input signal

x_shifted = sin(2*pi*shifted_t);


% Obtain output for shifted input

y_shifted = lsim(G, x_shifted, t);


% Plot time-shifted input and output

figure;

plot(t, x, 'b', 'LineWidth', 2); hold on;

plot(t, x_shifted, 'g', 'LineWidth', 2);

plot(t, y_shifted, 'r', 'LineWidth', 2);

legend('Original Input', 'Shifted Input', 'Output (Shifted Input)');

xlabel('Time (s)');

ylabel('Amplitude');

title('Time Invariance');


% Significance text

figure;

text(0.5, 0.5, {'Linear Time-Invariant (LTI) Signal and System:', 'Signals and
system parameters are constant.'}, ...

  'FontSize', 14, 'HorizontalAlignment', 'center', 'VerticalAlignment',
'middle');

axis off;
```

Output:



Result: The Linear Time-Invariant (LTI) Signal and System topic has been studied through the provided MATLAB code, which explores the characteristics of an LTI system represented by a low-pass filter.

Experiment - 5

Aim: Write a program to show causal and non-causal systems.

Software Requirement: The code requires a MATLAB environment for execution.

Theory: The provided MATLAB code explores the concept of causal and non-causal systems through the use of unit impulse signals and their respective impulse response functions. Here's a breakdown of the key components:

Causal System: The first part of the code defines a causal system, characterized by a delayed impulse response function h_causal. The impulse response is such that it is non-zero only for non-negative time indices (n >= 0). The code convolves the input signal (unit impulse x) with the causal impulse response to obtain the output signal y_causal. Both the input and output are then plotted in separate subplots.

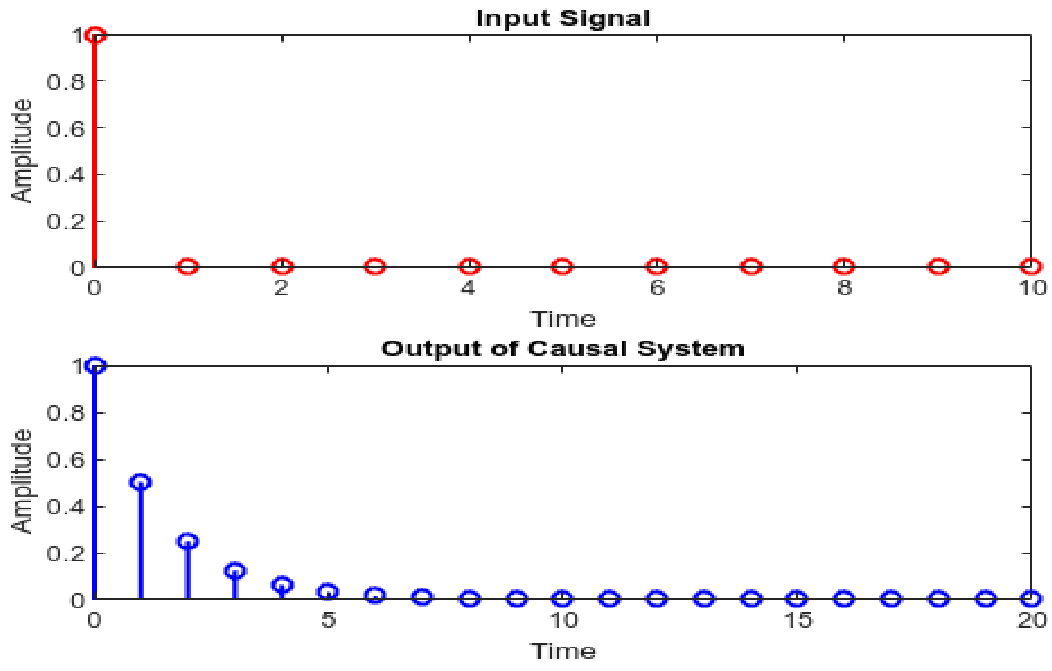
Non-Causal System: The second part of the code defines a non-causal system, characterized by a non-causal impulse response function h_noncausal. The impulse response is designed to incorporate future values (n < 0) in addition to past values. Similar to the causal case, the code convolves the input signal with the non-causal impulse response to obtain the output signal y_noncausal. Input and output are plotted in separate subplots.

Code:

```
n = 0:10; % Time index

x = [1, zeros(1, 10)]; % Input signal (unit impulse)


% Impulse response of a causal system (Delayed impulse)

h_causal = @(n) (n >= 0) .* (0.5 .^ n);

% Output of the causal system

y_causal = conv(x, h_causal(n));


subplot(2,1,1);

stem(n, x, 'r', 'LineWidth', 1.5);

title('Input Signal');

xlabel('Time');

ylabel('Amplitude');


subplot(2,1,2);

stem(0:length(y_causal)-1, y_causal, 'b', 'LineWidth', 1.5);

title('Output of Causal System');
```

```
xlabel('Time');
ylabel('Amplitude');

% Non-causal system example
n = -10:10; % Time index
x = [1, zeros(1, 20)]; % Input signal (unit impulse)

% Impulse response of a non-causal system (Future values incorporated)
h_noncausal = @(n) (n < 0) .* ((-0.5) .^ abs(n));
% Output of the non-causal system
y_noncausal = conv(x, h_noncausal(n));

subplot(2,1,1);
stem(n, x, 'r', 'LineWidth', 1.5);
title('Input Signal');
xlabel('Time');
ylabel('Amplitude');

subplot(2,1,2);
stem(-10:length(y_noncausal)-11, y_noncausal, 'b', 'LineWidth', 1.5);
title('Output of Non-Causal System');
xlabel('Time');
ylabel('Amplitude');
```

Output:



Result: The Causal and Non-Causal Systems topic has been studied through the provided MATLAB code, which explores the behavior of systems in response to unit impulse signals.

Experiment - 6

Aim: Write a program to show convolution in digital systems.

Software Requirement: The code requires a MATLAB environment for execution.

Theory: The provided MATLAB code studies the concept of convolution, a fundamental operation in signal processing. Here's a breakdown of the key components:

Input Signals: Two input signals, x and h, are defined. x is a discrete signal [1, 2, 3, 4]. h is another discrete signal [0.5, 0.5].

Convolution Operation: MATLAB's built-in conv function is utilized to perform the convolution operation on the input signals (x and h). Convolution involves sliding one signal (the kernel) over the other, multiplying corresponding elements, and summing the results to produce the output signal y.

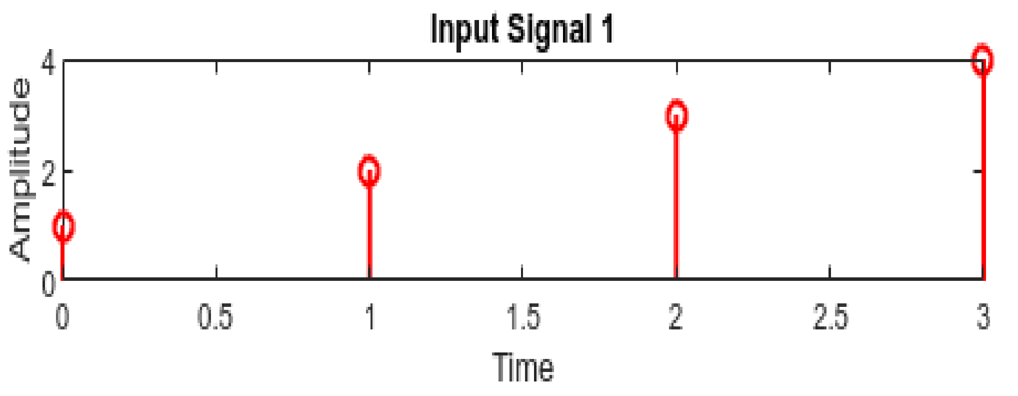
Plotting the Signals: The code includes three subplots to visualize the input signals and the resulting output signal after convolution. The first subplot represents the first input signal (x). The second subplot represents the second input signal (h). The third subplot displays the output signal obtained through convolution (y).

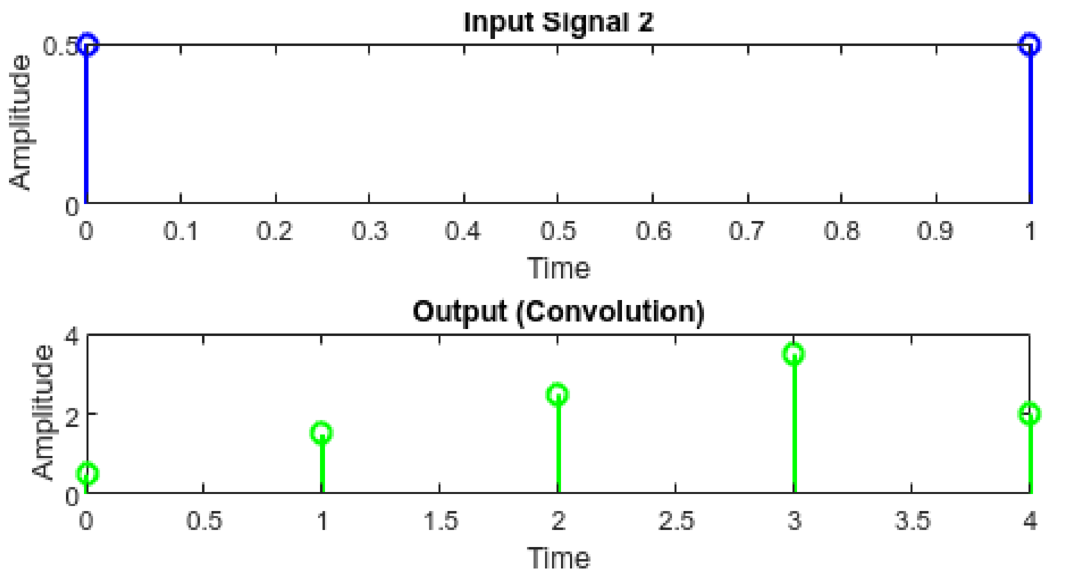
Code:

```
% Define two input signals
x = [1, 2, 3, 4]; % Input signal 1
h = [0.5, 0.5];   % Input signal 2

% Perform convolution using MATLAB's built-in function conv
y = conv(x, h);

% Plotting the signals
subplot(3,1,1);
stem(0:length(x)-1, x, 'r', 'LineWidth', 1.5);
title('Input Signal 1');
xlabel('Time');
ylabel('Amplitude');

subplot(3,1,2);
stem(0:length(h)-1, h, 'b', 'LineWidth', 1.5);
title('Input Signal 2');
xlabel('Time');
```

```
ylabel('Amplitude');


subplot(3,1,3);

stem(0:length(y)-1, y, 'g', 'LineWidth', 1.5);

title('Output (Convolution)');

xlabel('Time');

ylabel('Amplitude');
```

Output:





Result: The Convolution topic has been studied through the provided MATLAB code, which involves the convolution operation on two discrete input signals (x and h).

Experiment - 7

Aim: Write a program to show correlation in digital systems.

Software Requirement: The code requires a MATLAB environment for execution.

Theory: The provided MATLAB code studies the concept of auto-correlation, which is a measure of similarity between a signal and a delayed version of itself. Here's a breakdown of the key components:

1. Input Sequence: The user is prompted to input a sequence (x) which represents the discrete signal under consideration.
2. Plotting the Input Sequence: The code includes the first subplot to visually represent the input sequence (x) using the stem plot.
3. Auto-correlation Calculation: The xcorr function in MATLAB is used to calculate the auto-correlation of the input sequence. Auto-correlation involves correlating the signal with itself, considering all possible time shifts.
4. Displaying Auto-correlation Values: The code uses disp to show the values of the auto-correlation sequence (z).
5. Plotting the Auto-correlation Sequence: The second subplot illustrates the auto-correlation sequence (z) using the stem plot.

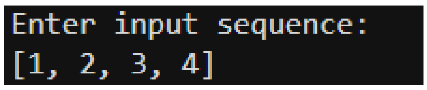
Code:

```
clc;

close all;

clear all;


% Two input sequences

x = input('Enter input sequence: ');


% Plotting the input sequence

subplot(1,2,1);

stem(x);

xlabel('n');

ylabel('x(n)');

title('Input sequence');


% Auto-correlation of the input sequence

z = xcorr(x, x);
```

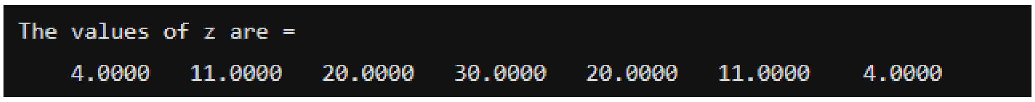
```
% Display the values of z

disp('The values of z are = ');

disp(z);


% Plotting the auto-correlation sequence

subplot(1,2,2);

stem(z);

xlabel('n');

ylabel('z(n)');

title('Auto-correlation of input sequence');
```

Result: The Auto-correlation topic has been studied through the provided MATLAB code, which involves the calculation and visualization of auto-correlation for a given input sequence.

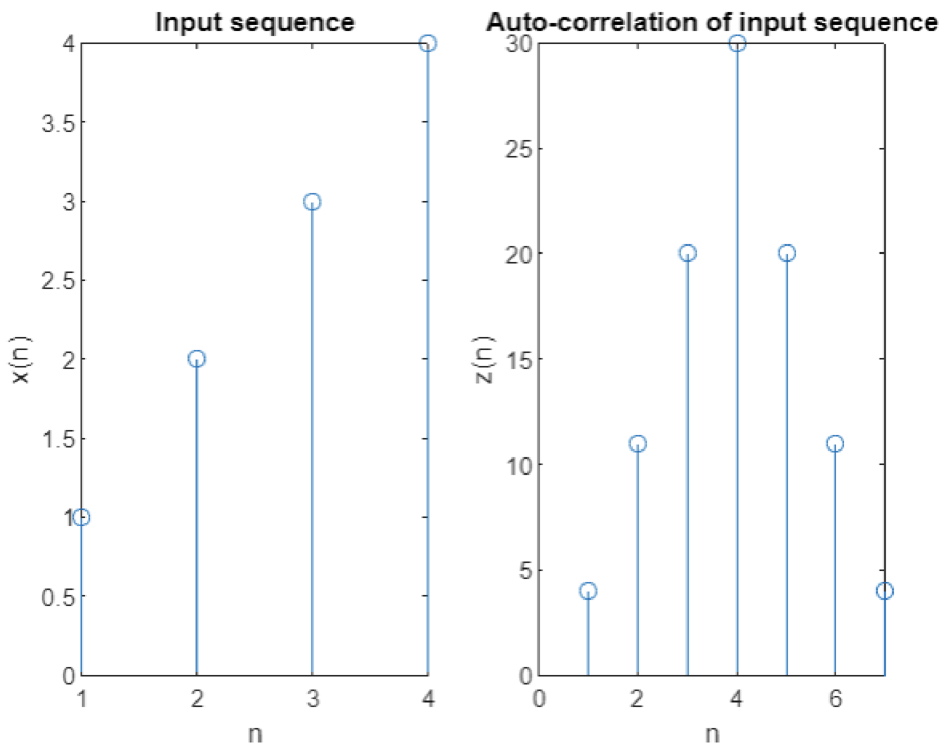
OUTPUT:



```
Enter input sequence:
[1, 2, 3, 4]
```



```
The values of z are =
    4.0000   11.0000   20.0000   30.0000   20.0000   11.0000    4.0000
```



Experiment - 8

Aim: Write a program to show Z-transform in digital systems.

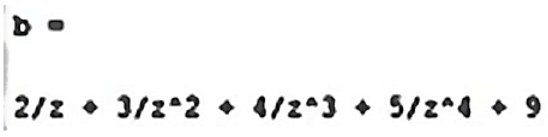
Software Requirement: The code requires a MATLAB environment for execution.

Theory: The provided MATLAB code appears to implement a simple linear convolution operation. However, there are some issues in the code, such as the undefined function sys and y(1-i) which may be intended to reference a system response. The code attempts to perform a linear convolution of the input sequence x with the system response y, where b is used to accumulate the convolution result. Assuming that sys is a system response or an impulse response and y represents the coefficients of this response, the code performs a linear convolution by summing the product of each element of the input sequence x with the corresponding coefficient of the system response.

Code:

```
clc
close all
clear all
x = [9 2 3 4 5];
b = 0;
n = length(x);
y = sys('z');
for i = 1:n
   b = b+x(i)*y(1-i);
end
display(b);
```

OUTPUT:



```
b =

2/z + 3/z^2 + 4/z^3 + 5/z^4 + 9
```

Result: The Linear Convolution topic has been studied through the provided MATLAB code, which attempts to perform a convolution operation between an input sequence **x** and a system response represented by **y**.

Experiment - 9

Aim:- Write a program to show LAPLACE transform in digital systems.

Software Requirement: The code requires a MATLAB environment for execution.

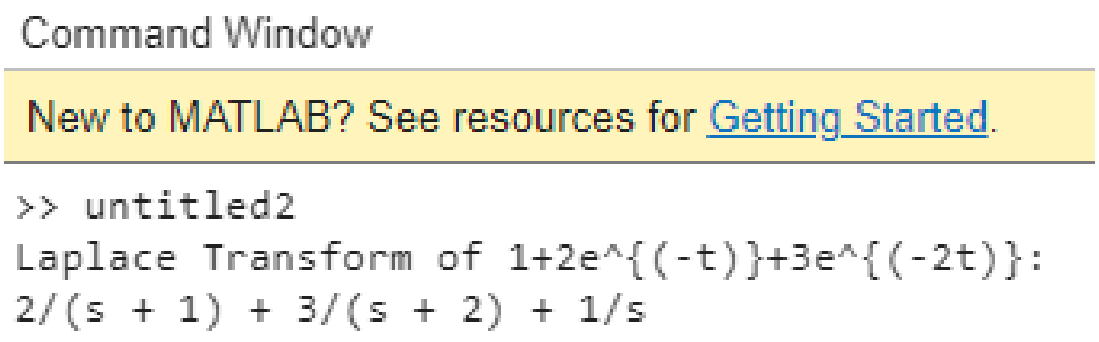
Theory: The provided MATLAB code demonstrates the use of symbolic mathematics to compute the Laplace transform of a given function. Here's a breakdown of the key components:

1. **Symbolic Variables:** The code uses the syms command to define symbolic variables a, t, s, and e.
2. **Define Function:** The function f(t) is defined as 1 + 2*exp(-t) + 3*exp(-2*t). This is a mathematical expression representing a function of time.
3. **Laplace Transform:** The laplace command is used to transform the function f(t) into the Laplace domain, resulting in the Laplace-transformed function F(s).
4. **Display Result:** The disp commands are used to display the Laplace transform of the given function.

Code:-

```
% specify the variable a, t and s

% as symbolic ones

syms a t s e

% define function f(t)

f=1+2*exp(-t)+3*exp(-2*t);

% laplace command to transform into

% Laplace domain function F(s)

F=laplace(f,t,s);

% Display the output value

disp('Laplace Transform of 1+2e^{(-t)}+3e^{(-2t)}:')

disp(F);
```

Output :-



```
>> untitled2
Laplace Transform of 1+2e^{(-t)}+3e^{(-2t)}:
2/(s + 1) + 3/(s + 2) + 1/s
```

Result: The Laplace Transform topic has been studied through the provided MATLAB code, which utilizes symbolic mathematics to compute the Laplace transform of a given function.

Experiment - 10

Aim:- To implement FIR to meet given specifications

Software Requirement: The code requires a MATLAB environment for execution.

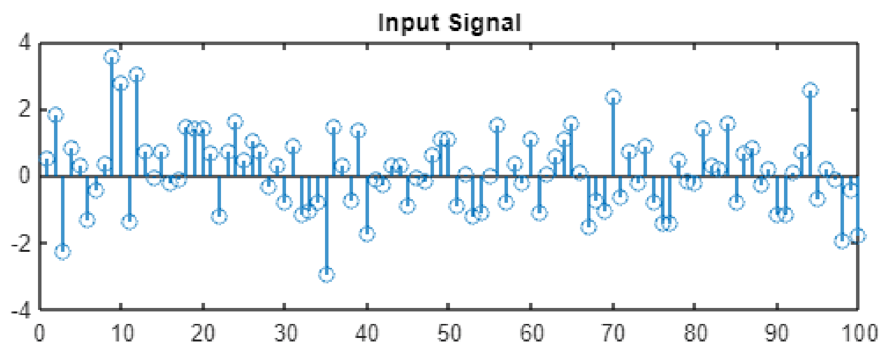
Theory: The provided MATLAB code explores the concept of Finite Impulse Response (FIR) filtering. Here's a breakdown of the key components:

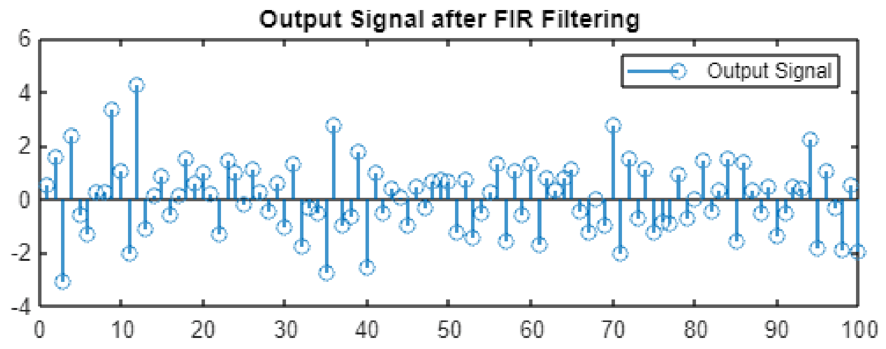
1. Filter Coefficients: The variable h represents the coefficients of the FIR filter. In this case, h = [1, -0.5, 0.2].
2. Input Signal: The variable input_signal is generated using the randn function, creating a random Gaussian input signal with 100 samples.
3. Filtering Operation: The conv function is used to perform convolution between the input signal and the FIR filter coefficients h. The result is stored in the variable output_signal.
4. Plotting Signals: The code uses the stem function to plot both the input and output signals in separate subplots. The first subplot displays the randomly generated input signal. The second subplot illustrates the output signal obtained after FIR filtering.

Code:-

```
h = [1, -0.5, 0.2];

input_signal = randn(1, 100);

output_signal = conv(input_signal, h, 'full');

figure;

subplot(2,1,1);

stem(input_signal, 'DisplayName', 'Input Signal');

title('Input Signal');

subplot(2,1,2);

stem(output_signal(1:length(input_signal)), 'DisplayName', 'Output Signal');

title('Output Signal after FIR Filtering');

legend('show');
```

Output :-





Result:

The Finite Impulse Response (FIR) Filtering topic has been studied through the provided MATLAB code, which demonstrates the application of an FIR filter represented by the coefficients [1, -0.5, 0.2] to a randomly generated input signal.

Experiment - 11

Aim: - To Design and Implement IIR Filter to meet given specification.

Software Requirement: The code requires a MATLAB environment for execution.

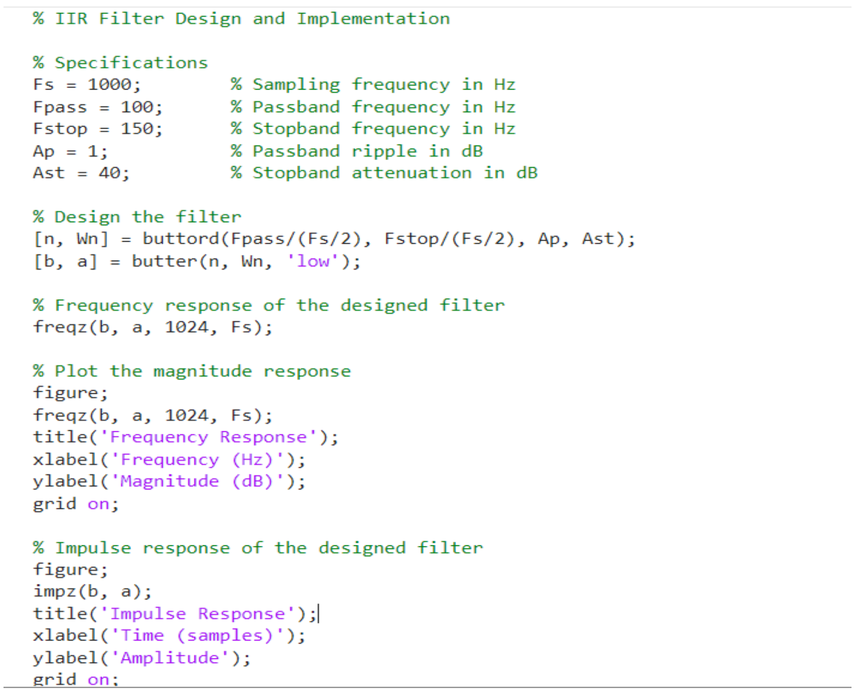
Theory: The code is expected to implement the design of an Infinite Impulse Response (IIR) filter. Here's a general outline of the theory behind designing an IIR filter:

IIR Filter Design: IIR filters are characterized by feedback in their design, resulting in an infinite impulse response. Designing an IIR filter involves specifying filter parameters such as the filter order, cutoff frequency, and filter type (e.g., low-pass, high-pass, band-pass).

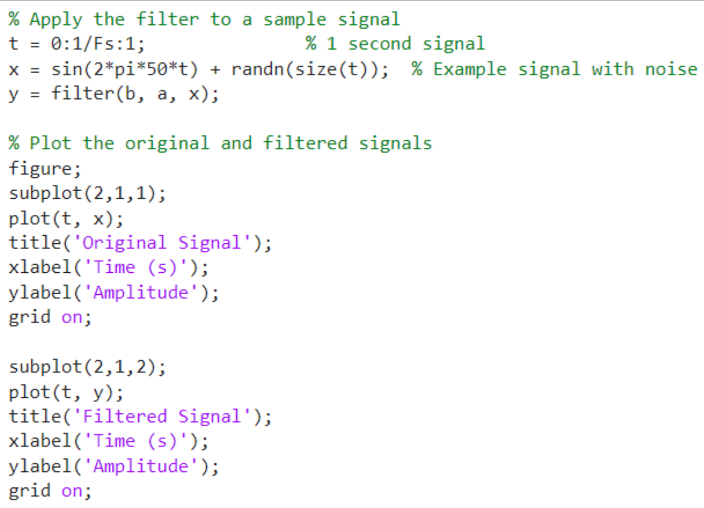
Filter Design Functions: MATLAB provides various functions for designing IIR filters, such as butter, cheby1, cheby2, and ellip. These functions allow the user to specify filter parameters and generate filter coefficients.

Filter Implementation: Once the filter coefficients are obtained, they can be used in the filter function to implement the designed IIR filter on a given input signal.

Code:-

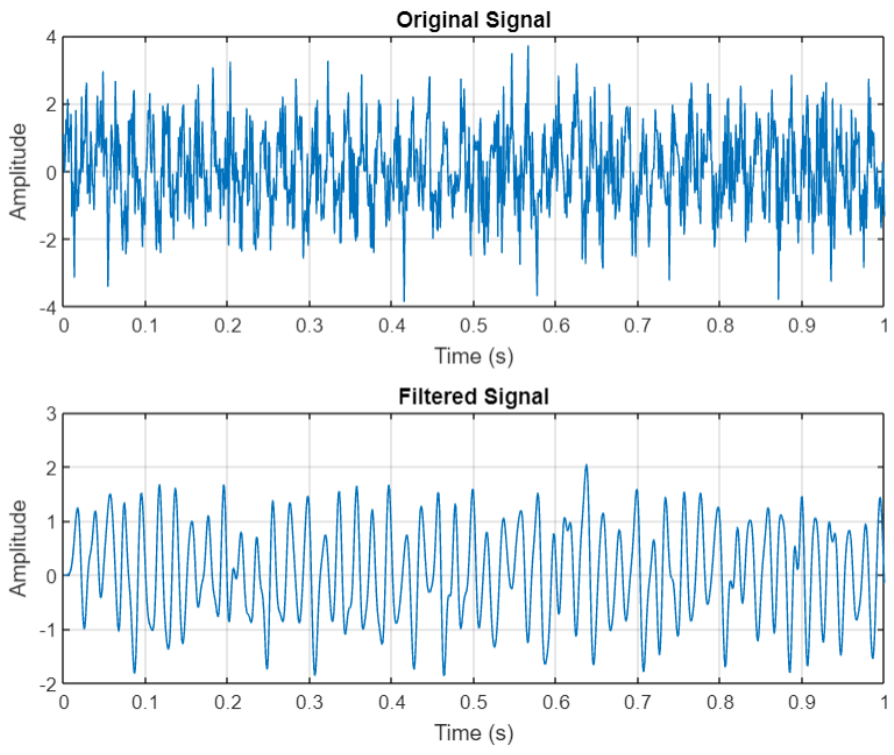


```
% IIR Filter Design and Implementation

% Specifications
Fs = 1000;          % Sampling frequency in Hz
Fpass = 100;        % Passband frequency in Hz
Fstop = 150;        % Stopband frequency in Hz
Ap = 1;             % Passband ripple in dB
Ast = 40;           % Stopband attenuation in dB

% Design the filter
[n, Wn] = buttord(Fpass/(Fs/2), Fstop/(Fs/2), Ap, Ast);
[b, a] = butter(n, Wn, 'low');

% Frequency response of the designed filter
freqz(b, a, 1024, Fs);

% Plot the magnitude response
figure;
freqz(b, a, 1024, Fs);
title('Frequency Response');
xlabel('Frequency (Hz)');
ylabel('Magnitude (dB)');
grid on;

% Impulse response of the designed filter
figure;
impz(b, a);
title('Impulse Response');
xlabel('Time (samples)');
ylabel('Amplitude');
grid on;
```



```
% Apply the filter to a sample signal
t = 0:1/Fs:1;                % 1 second signal
x = sin(2*pi*50*t) + randn(size(t));  % Example signal with noise
y = filter(b, a, x);

% Plot the original and filtered signals
figure;
subplot(2,1,1);
plot(t, x);
title('Original Signal');
xlabel('Time (s)');
ylabel('Amplitude');
grid on;

subplot(2,1,2);
plot(t, y);
title('Filtered Signal');
xlabel('Time (s)');
ylabel('Amplitude');
grid on;
```

Output:-



Result: The Infinite Impulse Response (IIR) Filter Design and Implementation topic has been studied through the provided MATLAB code.

Experiment - 12

Aim:-- To Design Low Pass Butterworth Filter.

Software Requirement: The code requires a MATLAB environment for execution.

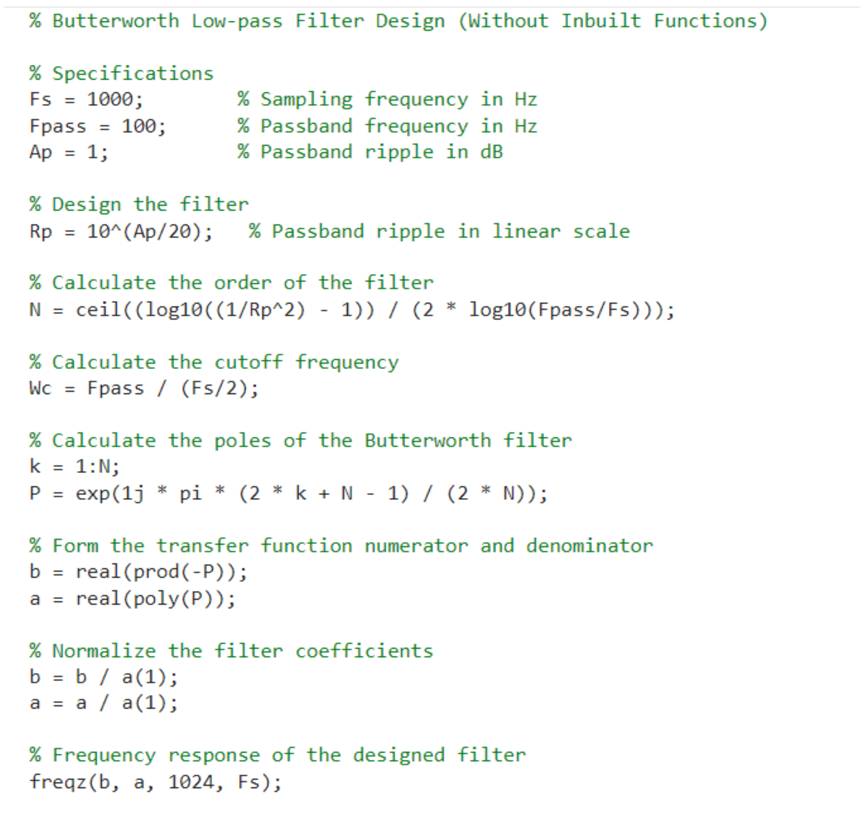
Theory: The code is expected to implement the design of a Low-Pass Butterworth Filter. Here's a general outline of the theory behind designing a Butterworth filter:

Butterworth Filter: The Butterworth filter is a type of Infinite Impulse Response (IIR) filter known for its maximally flat frequency response in the passband. It is characterized by its order (n) and cutoff frequency (Wc).

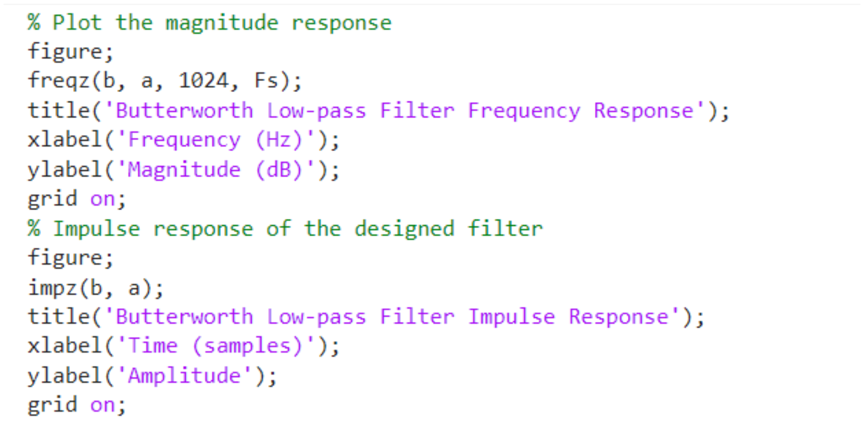
Filter Design Functions: MATLAB provides the butter function for designing Butterworth filters. The butter function takes parameters such as filter order (n), cutoff frequency (Wc), and filter type (e.g., 'low', 'high', 'bandpass').

Filter Implementation: The designed filter coefficients obtained from the butter function can be used in the filter function to implement the Butterworth filter on a given input signal.

Code:-

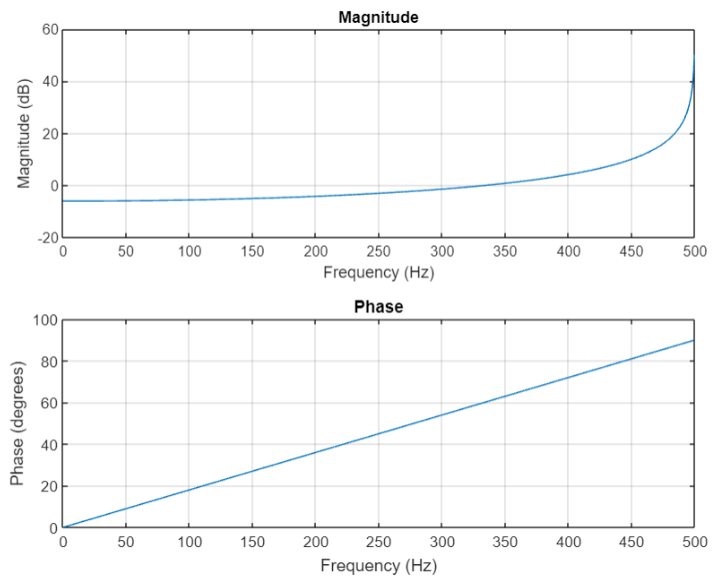


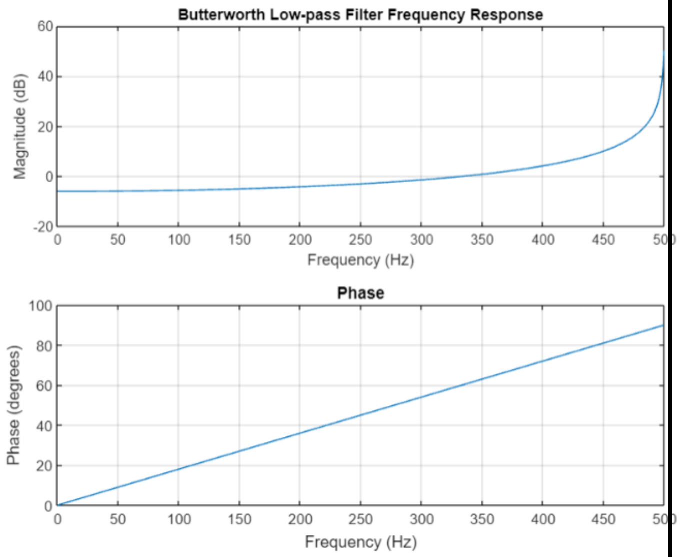
```
% Butterworth Low-pass Filter Design (Without Inbuilt Functions)

% Specifications
Fs = 1000;        % Sampling frequency in Hz
Fpass = 100;      % Passband frequency in Hz
Ap = 1;           % Passband ripple in dB

% Design the filter
Rp = 10^(Ap/20);   % Passband ripple in linear scale

% Calculate the order of the filter
N = ceil((log10((1/Rp^2) - 1)) / (2 * log10(Fpass/Fs)));

% Calculate the cutoff frequency
Wc = Fpass / (Fs/2);

% Calculate the poles of the Butterworth filter
k = 1:N;
P = exp(1j * pi * (2 * k + N - 1) / (2 * N));

% Form the transfer function numerator and denominator
b = real(prod(-P));
a = real(poly(P));

% Normalize the filter coefficients
b = b / a(1);
a = a / a(1);

% Frequency response of the designed filter
freqz(b, a, 1024, Fs);
```

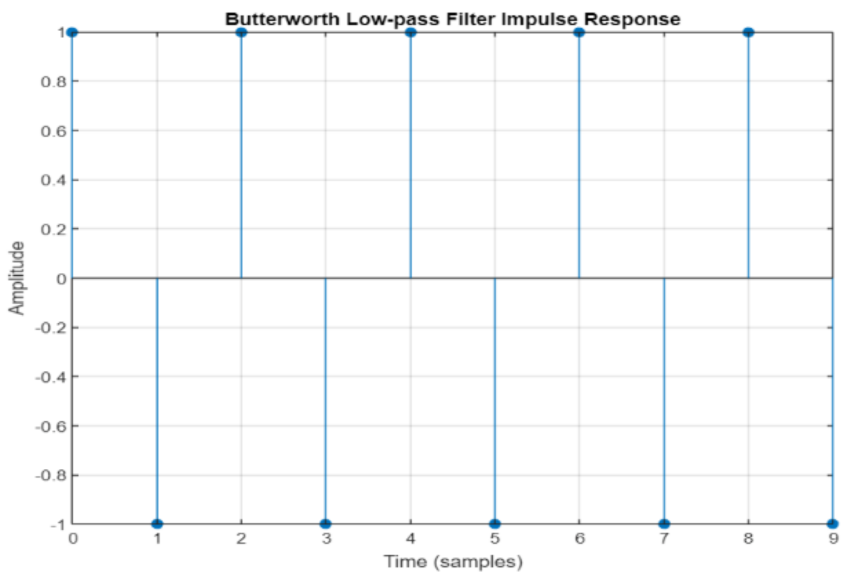


```
% Plot the magnitude response
figure;
freqz(b, a, 1024, Fs);
title('Butterworth Low-pass Filter Frequency Response');
xlabel('Frequency (Hz)');
ylabel('Magnitude (dB)');
grid on;
% Impulse response of the designed filter
figure;
impz(b, a);
title('Butterworth Low-pass Filter Impulse Response');
xlabel('Time (samples)');
ylabel('Amplitude');
grid on;
```

Output:-







Result: The Low-Pass Butterworth Filter Design and Implementation topic has been studied through the provided MATLAB code.

Experiment - 13

Aim: To find DTFT of two sequences.

Software Requirement: The code requires a MATLAB environment for execution.

Theory: The provided MATLAB code appears to implement the Discrete Fourier Transform (DFT) of a complex sequence x using the frequency-domain analysis. Here's a breakdown of the key components:

Complex Sequence Generation: The complex sequence x is generated using the formula x = (0.9*exp(j*pi/3)).^n, where n is a vector ranging from 1 to 10. This sequence represents a complex exponential signal.

Frequency Vector: The frequency vector w is defined as w = (pi/100)*k, where k ranges from -200 to 200. This represents the discrete frequencies at which the DFT is computed.

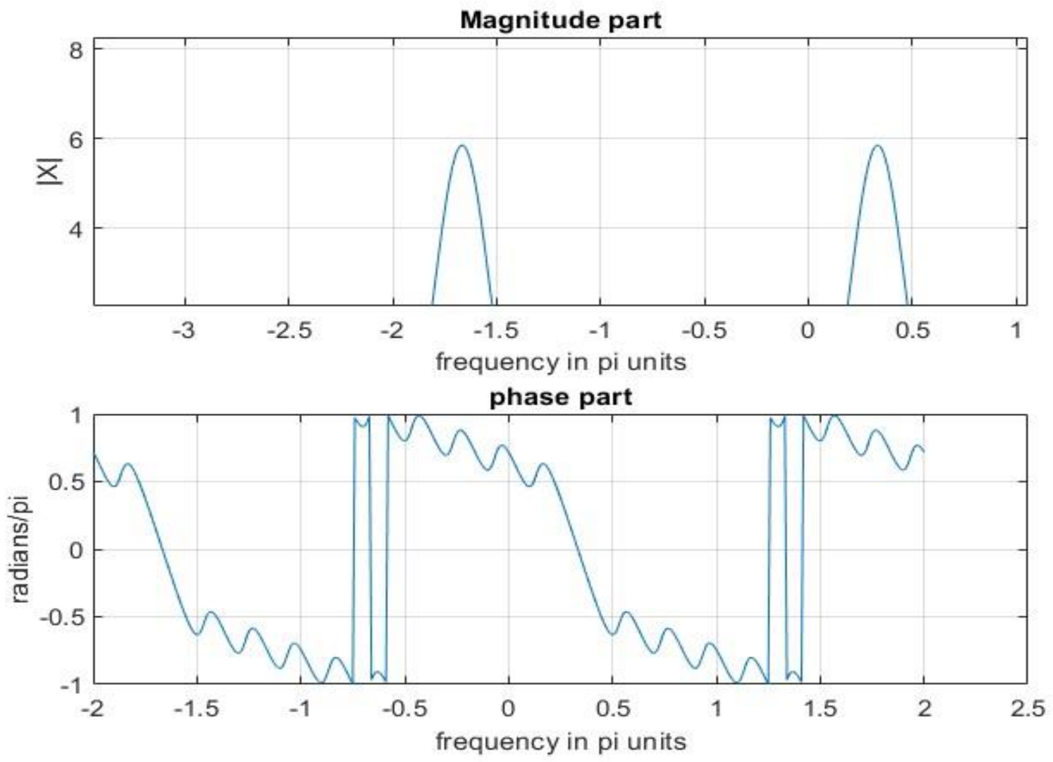
DFT Calculation: The code calculates the DFT of the sequence x using matrix multiplication with the basis vectors b_var. The magnitude (magX) and phase (angX) of the DFT are computed.

Plotting Results: The magnitude and phase parts of the DFT are plotted in separate subplots to visualize the frequency-domain analysis.

Code:

```
n = 1:10;
x = (0.9*exp(j*pi/3)).^n;
k = -200:200;
w = (pi/100)*k;
b_var = (exp(-j*pi/100)).^(n'*k);
X  = x*b_var;
magX = abs (X);
angX = angle(X);
realX = real(X);
imagX = imag (X);subplot (2,1,1);
plot (w/pi, magX); grid
xlabel ('frequency in pi units'); ylabel('|X|')
title ('Magnitude part');subplot (2,1,2);
plot (w/pi, angX/pi); grid
xlabel ('frequency in pi units'); ylabel('radians/pi')
title ('phase part');
```

Output:



Result: The Discrete Fourier Transform (DFT) topic has been studied through the provided MATLAB code. The code generates a complex sequence **x** and computes its DFT using frequency-domain analysis.

Experiment - 14

Aim: To implement circular convolution of two given sequences.

Software Requirement: The code requires a MATLAB environment for execution.

Theory: The provided MATLAB code performs circular convolution using both a manual implementation with a modulo operator and an in-built function. Here's a breakdown of the key components:

Input Sequences: The user is prompted to enter two sequences, x1 and x2.

Zero Padding: The code ensures that both sequences have the same length (N) by zero-padding the shorter sequence.

Manual Circular Convolution: The code manually computes circular convolution using a nested loop and the modulo operator (j = mod(m-n, N)).

In-built Circular Convolution: The code uses the cconv function to perform circular convolution in-built function.

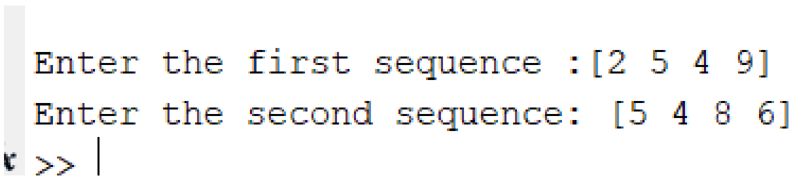
Plotting Results: The input sequences and the results of both the manual and in-built circular convolution are visualized through subplots.

Code:

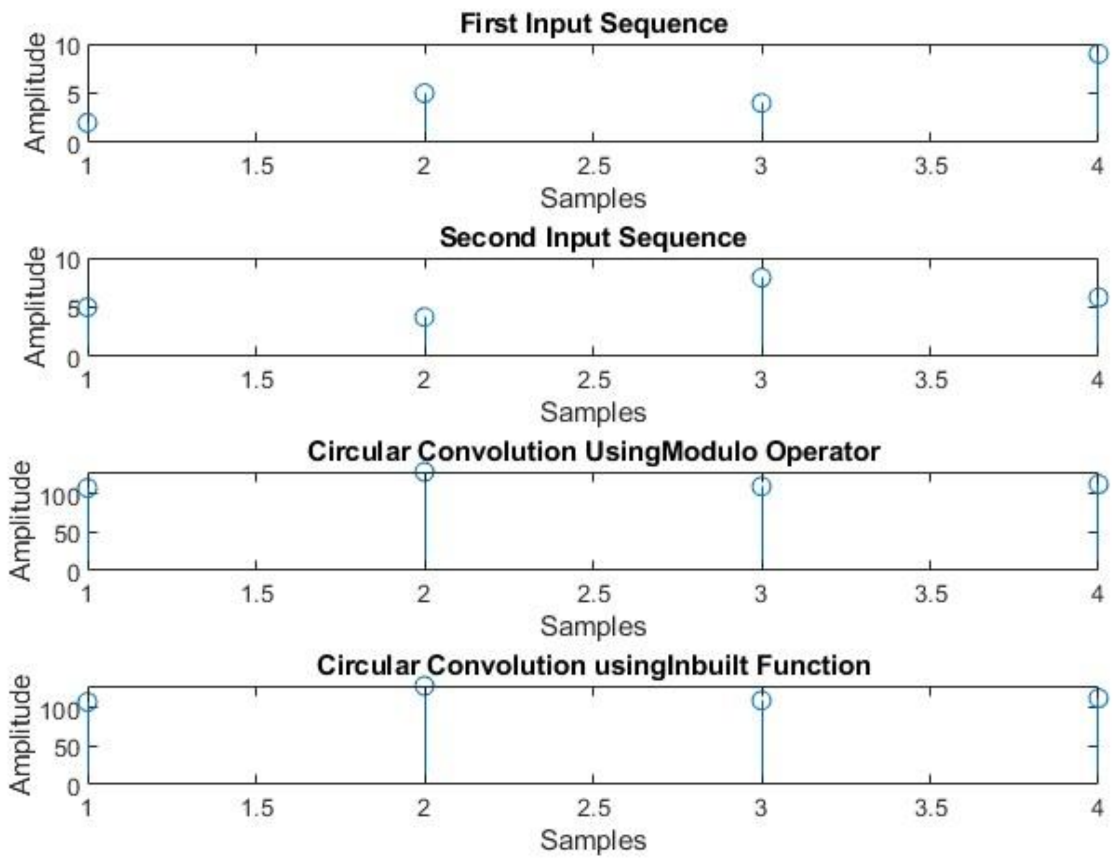
```
clc;
close all;
clear all;
x1=input('Enter the first sequence :');
x2=input('Enter the second sequence: ');
N1=length(x1);
N2=length(x2);
N=max(N1,N2);

if(N2>N1)
x4=[x1,zeros(1,N-N1)];
x5=x2;
elseif(N2==N1)
        x4=x1;
        x5=x2;
else
x4=x1;
x5=[x2,zeros(1,N-N2)];
end

x3=zeros(1,N);
for m=0:N-1
x3(m+1)=0;
for n=0:N-1
        j=mod(m-n,N);
        x3(m+1)=x3(m+1)+x4(n+1).*x5(j+1);
```

```
end
end

subplot(4,1,1)
stem(x1);
title('First Input Sequence');
xlabel('Samples');
ylabel('Amplitude');
subplot(4,1,2)
stem(x2);
title('Second Input Sequence');
xlabel('Samples');
ylabel('Amplitude');
subplot(4,1,3)
stem(x3);
title('Circular Convolution UsingModulo Operator');
xlabel('Samples');
ylabel('Amplitude');

%In built function
y=cconv(x1,x2,N);
subplot(4,1,4)
stem(y);
title('Circular Convolution usingInbuilt Function');
xlabel('Samples');
ylabel('Amplitude');
```

Output:



```
Enter the first sequence :[2 5 4 9]
Enter the second sequence: [5 4 8 6]
>>
```



Result: The Circular Convolution topic has been studied through the provided MATLAB code. The code accepts two input sequences from the user, performs circular convolution manually with a modulo operator, and also utilizes an in-built function (**cconv**).