# SIGNALS AND SYSTEMS LAB (CSE-301L)

**Spring 2024, 4th Semester**

**Lab Report 01**

Submitted by**: Hassan Zaib Jadoon**

Registration Number**: 22PWCSE2144**

Section: **A**

"On my honor, as a student at the University of Engineering and Technology Peshawar, I have neither given nor received unauthorized assistance on this academic work."

Recoverable Signature

X [illegible]

Hassan Zaib Jadoon
Student
Signed by: 6fc0c9f9-f92a-4c79-aa63-cf063a60e638

Signature:

Date: 10/3/2024

**Submitted To: Dr. Safdar Nawaz Khan Marwat**

**Department of Computer Systems Engineering**
**University of Engineering and Technology Peshawar**

# Lab 04

## TASK 01

**Given the signals:**
**xବ[n] = {2, 5, 8, 4, 3} and xଣ**

**[n] = {4, 3, 2}**
**a) Write a MATLAB program that adds and multiply these two signals. Use vector addition and multiplication respectively.**
**b) Instead of using vector addition and multiplication, use for loop to add and multiply the corresponding elements of the two signals.**

## Problem Analysis:

This task involves adding and multiplying two signals using both vector operations and for loops.

## Algorithm:

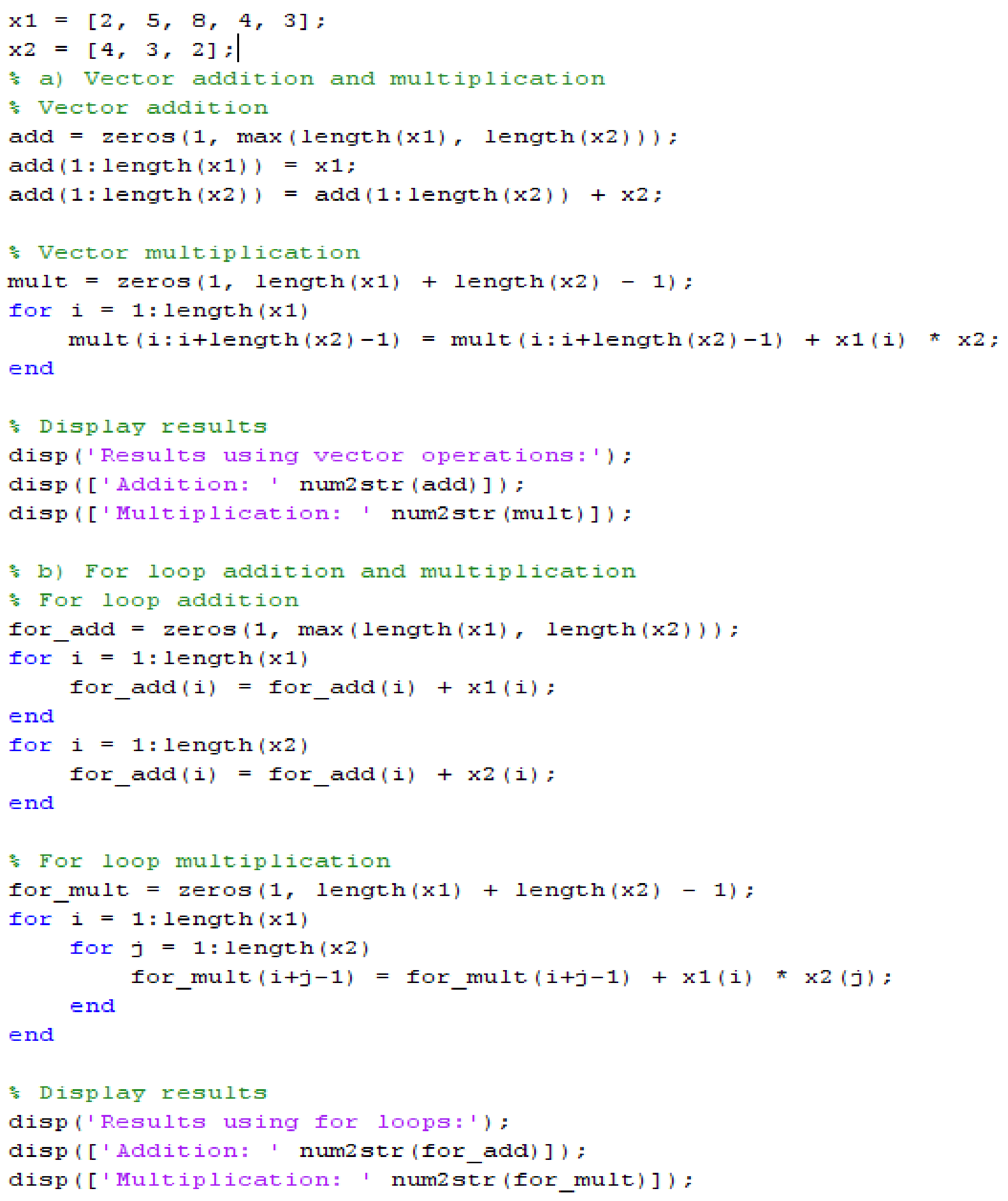
a) Vector Addition and Multiplication:

1. Define two signals.
2. Perform vector addition by adding corresponding elements of the two signals. If one signal is shorter, pad it with zeros.
3. Perform vector multiplication by multiplying corresponding elements of the two signals. If one signal is shorter, pad it with zeros.
4. Store the results.

b) For Loop Addition and Multiplication:

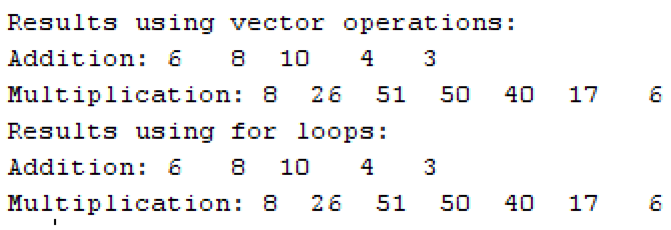
1. Define two signals.
2. Initialize empty arrays for addition and multiplication results.
3. Iterate through the signals using a for loop.
4. At each iteration, add the corresponding elements of the signals and store the result.
5. At each iteration, multiply the corresponding elements of the signals and store the result.
6. Store the results.

## Code



```
x1 = [2, 5, 8, 4, 3];
x2 = [4, 3, 2];
% a) Vector addition and multiplication
% Vector addition
add = zeros(1, max(length(x1), length(x2)));
add(1:length(x1)) = x1;
add(1:length(x2)) = add(1:length(x2)) + x2;

% Vector multiplication
mult = zeros(1, length(x1) + length(x2) - 1);
for i = 1:length(x1)
    mult(i:i+length(x2)-1) = mult(i:i+length(x2)-1) + x1(i) * x2;
end

% Display results
disp('Results using vector operations:');
disp(['Addition: ' num2str(add)]);
disp(['Multiplication: ' num2str(mult)]);

% b) For loop addition and multiplication
% For loop addition
for_add = zeros(1, max(length(x1), length(x2)));
for i = 1:length(x1)
    for_add(i) = for_add(i) + x1(i);
end
for i = 1:length(x2)
    for_add(i) = for_add(i) + x2(i);
end

% For loop multiplication
for_mult = zeros(1, length(x1) + length(x2) - 1);
for i = 1:length(x1)
    for j = 1:length(x2)
        for_mult(i+j-1) = for_mult(i+j-1) + x1(i) * x2(j);
    end
end

% Display results
disp('Results using for loops:');
disp(['Addition: ' num2str(for_add)]);
disp(['Multiplication: ' num2str(for_mult)]);
```

## Output



```
Results using vector operations:
Addition: 6    8   10    4    3
Multiplication: 8   26   51   50   40   17    6
Results using for loops:
Addition: 6    8   10    4    3
Multiplication: 8   26   51   50   40   17    6
```

## Conclusion:

We explored two methods of performing signal addition and multiplication in MATLAB. The first method utilized vector operations, which offer a concise and efficient way to perform these operations. The second method employed for loops, providing a more explicit control over the computation process.

## TASK 02

**Amplitude scaling by a factor $\beta$ causes each sample to get multiplied by $\beta$. Write a user-defined function that has two input arguments: (i) a signal to be scaled and (ii) scaling factor $\beta$. The function should return the scaled output to the calling program. In the calling program, get the discrete time signal as well as the scaling factor from user and then call the above-mentioned function.**

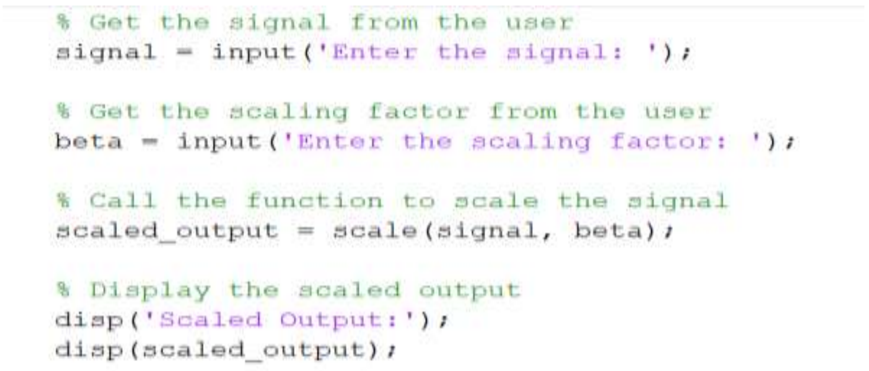
## Problem Analysis:

This task requires creating a user-defined function to scale a signal by a given factor.

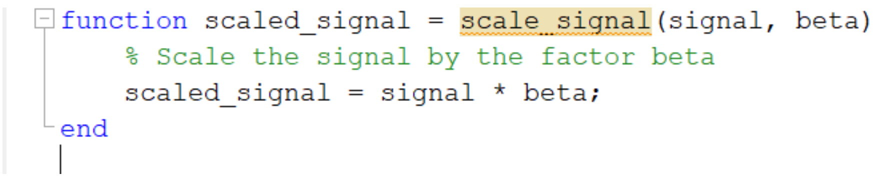
## Algorithm:

1. Define a user-defined function that takes two input arguments: the signal to be scaled and the scaling factor $\beta$.
2. Within the function, multiply each sample of the input signal by the scaling factor $\beta$.
3. Return the scaled output signal.

### Code



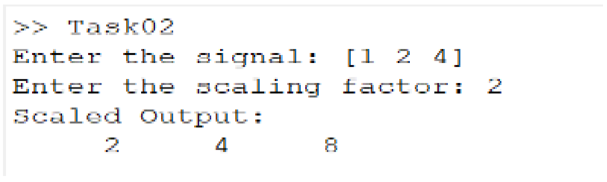
```
% Get the signal from the user
signal = input('Enter the signal: ');

% Get the scaling factor from the user
beta = input('Enter the scaling factor: ');

% Call the function to scale the signal
scaled_output = scale(signal, beta);

% Display the scaled output
disp('Scaled Output:');
disp(scaled_output);
```



```
function scaled_signal = scale_signal(signal, beta)
    % Scale the signal by the factor beta
    scaled_signal = signal * beta;
end
```

*Fuction*

**Output**

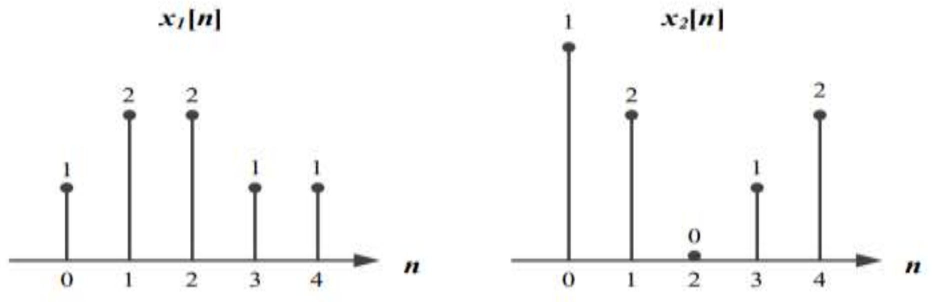


```
>> Task02
Enter the signal: [1 2 4]
Enter the scaling factor: 2
Scaled Output:
     2     4     8
```

## Conclusion:

We implemented a user-defined function to scale a signal by a given factor. This function provides a convenient way to modify the amplitude of a signal according to a scaling factor specified by the user.

## TASK 03



**Write a MATLAB program to compare the signals 𝑥ଵ[𝑛] and 𝑥ଶ[𝑛]. Determine the index where a sample of 𝑥ଵ[𝑛] has smaller amplitude as compared to the corresponding sample of 𝑥ଶ[𝑛]. Use for loop.**

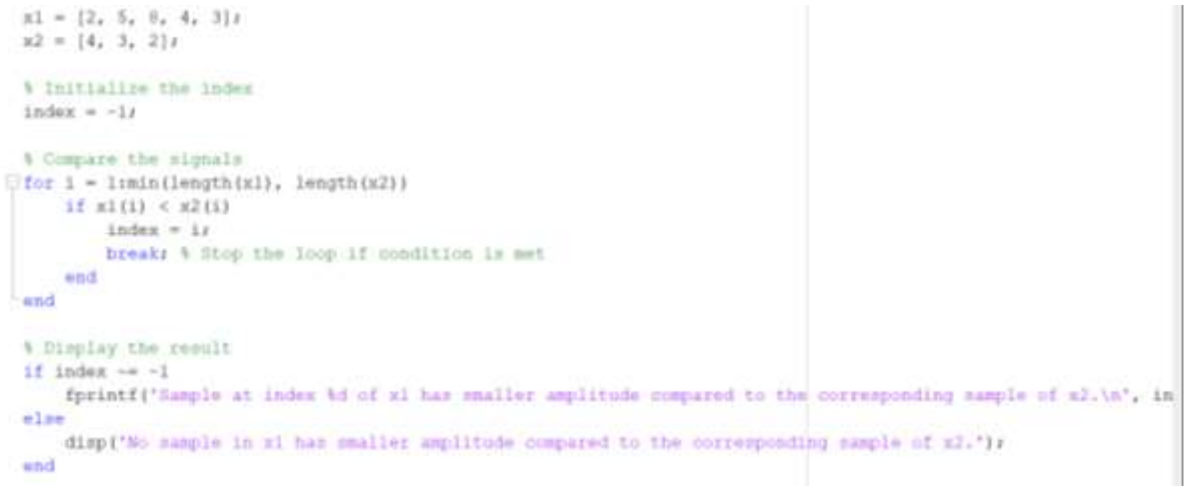
## Problem Analysis:

For this task, a MATLAB program needs to compare two signals and determine where one signal has smaller amplitude than the other.

## Algorithm:

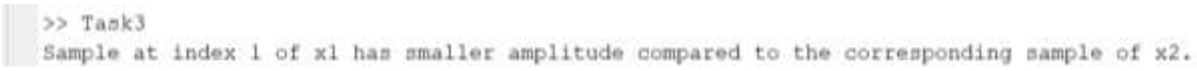
1. Define two signals.
2. Iterate through the signals using a for loop.
3. At each iteration, compare the amplitude of the corresponding samples of the two signals.
4. Determine the index where has a smaller amplitude than.
5. Return the index.

## Code:



```
x1 = [2, 5, 8, 4, 3];
x2 = [4, 3, 2];

% Initialize the index
index = -1;

% Compare the signals
for i = 1:min(length(x1), length(x2))
    if x1(i) < x2(i)
        index = i;
        break; % Stop the loop if condition is met
    end
end

% Display the result
if index ~= -1
    fprintf('Sample at index %d of x1 has smaller amplitude compared to the corresponding sample of x2.\n', in
else
    disp('No sample in x1 has smaller amplitude compared to the corresponding sample of x2.');
end
```

## Output



```
>> Task3
Sample at index 1 of x1 has smaller amplitude compared to the corresponding sample of x2.
```

## Conclusion:

This task demonstrates the practical application of signal comparison techniques, which are essential in various signal processing applications such as pattern recognition and signal classification.

## TASK 04

Plot the two curves $y$ଵ $= 2x + 3$ and $y$ଶ $= 4x + 3$ on the same graph using different plot styles.

## Problem Analysis:

The objective here is to plot two curves on the same graph using different plot styles.

## Algorithm:

1. Define two functions.
2. Define a range of *x* values.
3. Calculate *x* value.
4. Plot both on the same graph using different plot styles.

## CODE:

```
x = -10:0.1:10;

% Define the equations
y1 = 2*x + 3;
y2 = 4*x + 3;

% Plot the curves
plot(x, y1, '-r', 'LineWidth', 2); % Plot y1 with a solid red line
hold on; % Hold the plot
plot(x, y2, '--b', 'LineWidth', 2); % Plot y2 with a dashed blue line

% Add labels and legend
xlabel('x');
ylabel('y');
title('Plot of y1 = 2x + 3 and y2 = 4x + 3');
legend('y1 = 2x + 3', 'y2 = 4x + 3');

% Turn the grid on
grid on;

% Display the plot
hold off; % Release the plot
```

## OUTPUT:

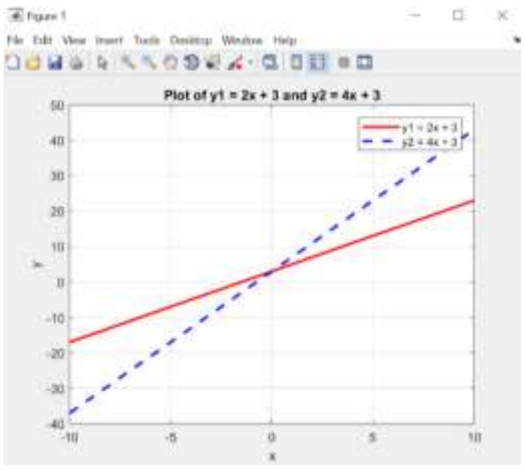


Figure 1
File Edit View Insert Tools Desktop Window Help
Plot of y1 = 2x + 3 and y2 = 4x + 3
y1 = 2x + 3
y2 = 4x + 3
50
40
30
20
10
0
-10
-20
-30
-40
-10
-5
0
5
10
x
y

## Conclusion:

This task highlights the importance of graphical representation in data analysis and visualization tasks.

## TASK 05:

**Make two separate functions for signal addition and multiplication. The functions should take the signals as input arguments and return the resultant signal. In the main program, get the signals from user, call the functions for signal addition and multiplication, and plot the original signals as well as the resultant signals.**

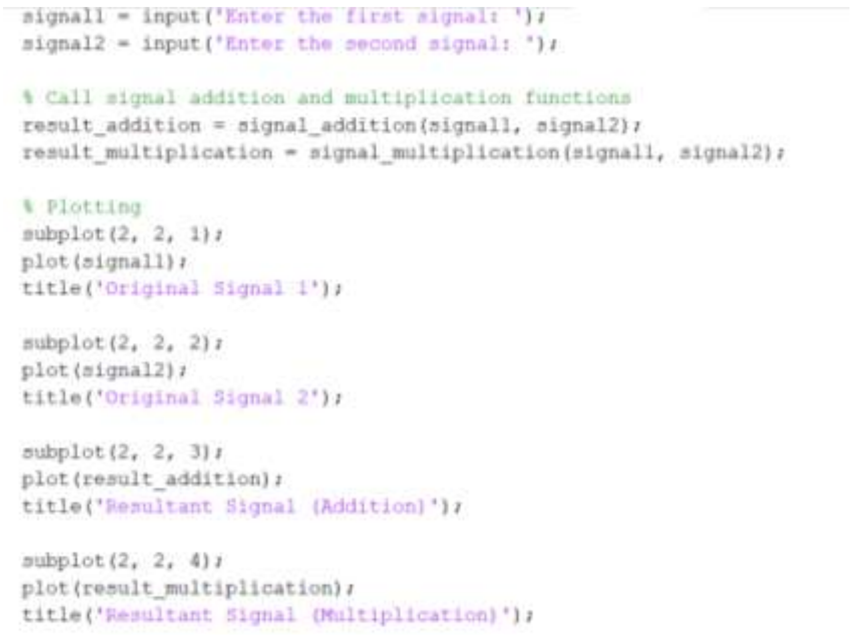
### Problem Analysis:

This task involves creating separate functions for signal addition and multiplication, then plotting the original and resultant signals.

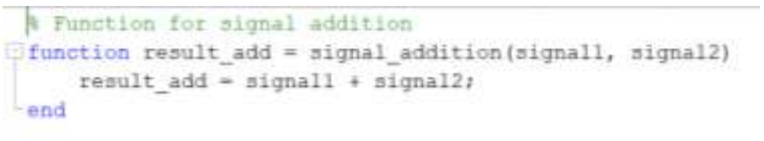
### Algorithm:

1. Define two separate functions for signal addition and multiplication.
2. Each function takes two signals as input arguments.
3. Perform addition or multiplication operation on corresponding samples of the two signals.
4. Return the resultant signal.
5. In the main program, get the signals from the user.
6. Call the functions for signal addition and multiplication.
7. Plot the original signals as well as the resultant signals.

### CODE:

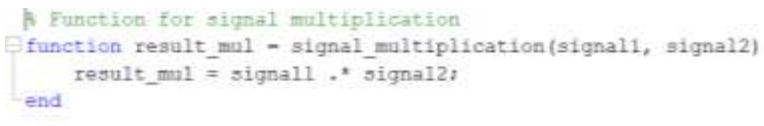


```
signal1 = input('Enter the first signal: ');
signal2 = input('Enter the second signal: ');

% Call signal addition and multiplication functions
result_addition = signal_addition(signal1, signal2);
result_multiplication = signal_multiplication(signal1, signal2);

% Plotting
subplot(2, 2, 1);
plot(signal1);
title('Original Signal 1');

subplot(2, 2, 2);
plot(signal2);
title('Original Signal 2');

subplot(2, 2, 3);
plot(result_addition);
title('Resultant Signal (Addition)');

subplot(2, 2, 4);
plot(result_multiplication);
title('Resultant Signal (Multiplication)');
```



```
% Function for signal addition
function result_add = signal_addition(signal1, signal2)
    result_add = signal1 + signal2;
end
```

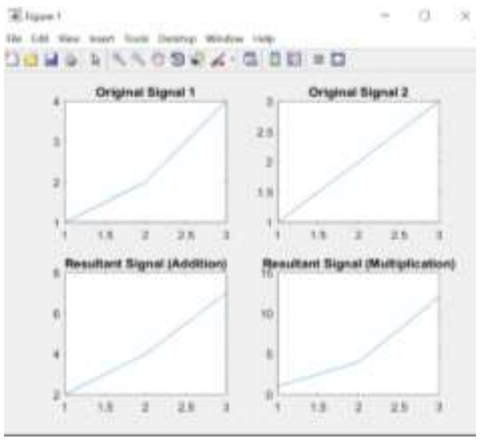
*Figure 1: For Signal Addition*



```
% Function for signal multiplication
function result_mul = signal_multiplication(signal1, signal2)
    result_mul = signal1 .* signal2;
end
```

*Figure 2: For Signal Multiplication*

**OUTPUT:**



## Conclusion:

We modularized signal addition and multiplication operations into separate functions.

## TASK 06:

**Given the signals:**

**$X_1[n] = 2\delta[n] + 5\delta[n-1] + 8\delta[n-2] + 4\delta[n-3] + 3\delta[n-4]$ $X_2[n] = \delta[n-4] + 4\delta[n-5] + 3\delta[n-6] + 2\delta[n-7]$**

**Write a MATLAB program that adds these two signals. Plot the original signals as well as the final result.**

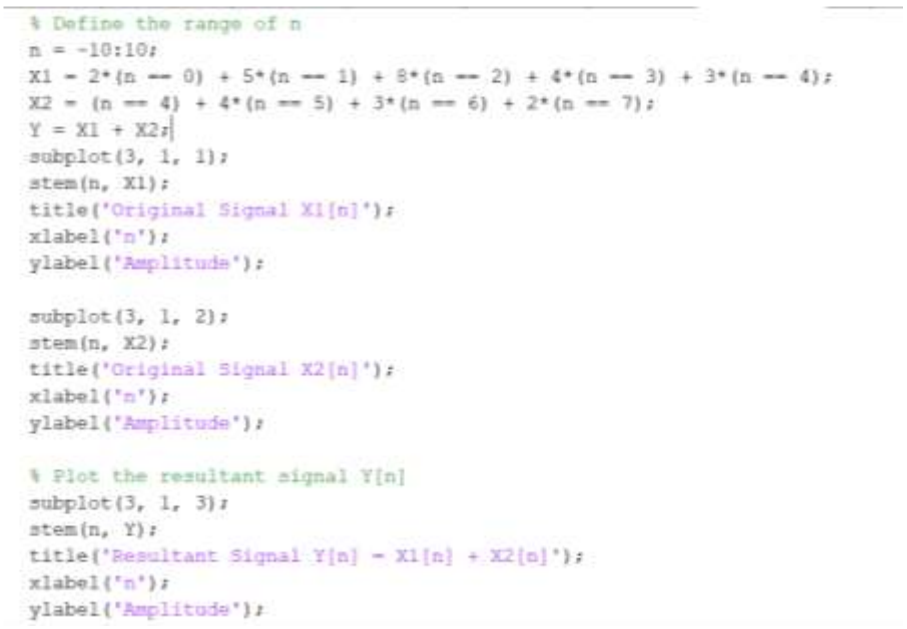
## Problem Analysis:

This task requires counting the number of samples with amplitudes greater than a threshold and less than another threshold in a given signal.

## Algorithm:

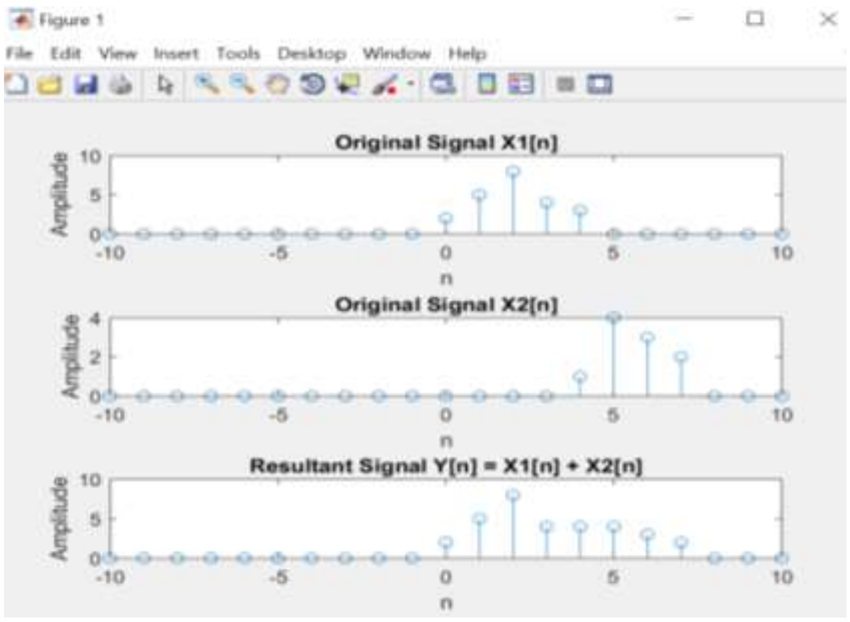
1. Define two signals.
2. Add corresponding samples to get the resultant signal.
3. Plot the original signals as well as the resultant signal.

## Code



```
% Define the range of n
n = -10:10;
X1 = 2*(n == 0) + 5*(n == 1) + 8*(n == 2) + 4*(n == 3) + 3*(n == 4);
X2 = (n == 4) + 4*(n == 5) + 3*(n == 6) + 2*(n == 7);
Y = X1 + X2;
subplot(3, 1, 1);
stem(n, X1);
title('Original Signal X1[n]');
xlabel('n');
ylabel('Amplitude');

subplot(3, 1, 2);
stem(n, X2);
title('Original Signal X2[n]');
xlabel('n');
ylabel('Amplitude');

% Plot the resultant signal Y[n]
subplot(3, 1, 3);
stem(n, Y);
title('Resultant Signal Y[n] = X1[n] + X2[n]');
xlabel('n');
ylabel('Amplitude');
```

## Output



## Conclusion:

This task demonstrates the process of combining signals to create a new composite signal. By visualizing the original and resultant signals, we can analyze how their properties change because of the addition operation.

## TASK 07

**Take a discrete-time signal from user. Count the number of samples with amplitude greater than a threshold of 3 and less than a threshold of -3 (use for loop).**

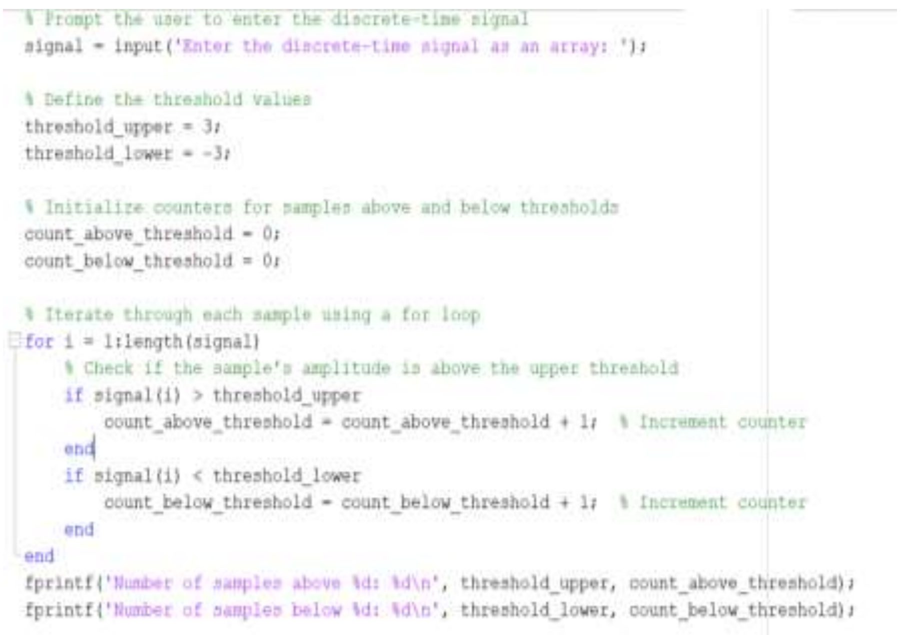
### Problem Analysis:

For this task, a function needs to be implemented to down sample a signal and then verified using MATLAB's built-in function.

## Algorithm:

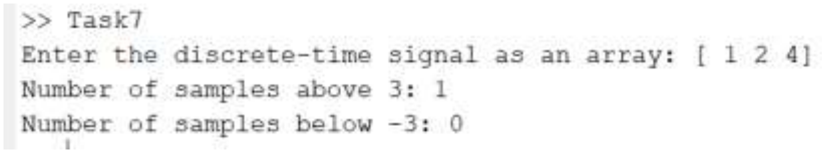
1. Take a discrete-time signal from the user.
2. Initialize counters for samples with amplitude greater than a threshold and less than another threshold.
3. Iterate through the samples of the signal.
4. For each sample, check if it exceeds the thresholds.
5. Increment the corresponding counter based on the condition.
6. Return the counts.

### Code:



```
% Prompt the user to enter the discrete-time signal
signal = input('Enter the discrete-time signal as an array: ');

% Define the threshold values
threshold_upper = 3;
threshold_lower = -3;

% Initialize counters for samples above and below thresholds
count_above_threshold = 0;
count_below_threshold = 0;

% Iterate through each sample using a for loop
for i = 1:length(signal)
    % Check if the sample's amplitude is above the upper threshold
    if signal(i) > threshold_upper
        count_above_threshold = count_above_threshold + 1;  % Increment counter
    end
    if signal(i) < threshold_lower
        count_below_threshold = count_below_threshold + 1;  % Increment counter
    end
end
fprintf('Number of samples above %d: %d\n', threshold_upper, count_above_threshold);
fprintf('Number of samples below %d: %d\n', threshold_lower, count_below_threshold);
```

### Output:



```
>> Task7
Enter the discrete-time signal as an array: [ 1 2 4]
Number of samples above 3: 1
Number of samples below -3: 0
```

### Conclusion:

This task illustrates the importance of signal analysis in identifying relevant information based on predefined criteria. By counting samples exceeding certain thresholds, we can extract useful insights about the characteristics of the signal.

## TASK 08

**Write your own function to downsample a signal i.e. retain odd numbered samples of the original signal and discard the even-numbered (downsampling by 2). The function must take a signal as input and return the downsampled version of that signal. See Figure 4.9 for reference. Call this function from a MATLAB file. Verify your result by using the command downsample. Plot the original signal, downsampled signal determined by your program, and downsampled signal obtained by the command downsample.**

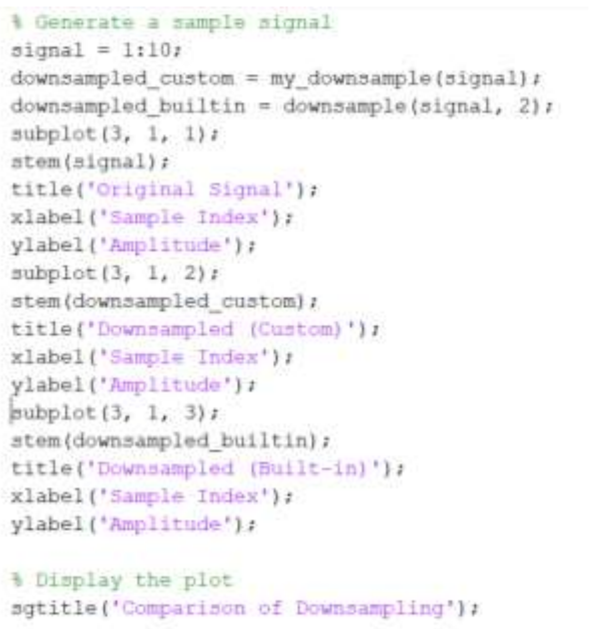
### Problem Analysis:

For this task, a function needs to be implemented to downsample a signal and then verified using MATLAB's built-in function.

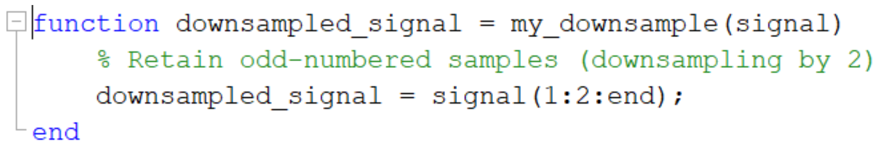
### Algorithm:

1. Define a function to downsample a signal by a factor of 2.
2. Copy odd-numbered samples from the original signal to the downsampled signal.
3. Discard even-numbered samples.
4. Return the downsampled signal.

### Code



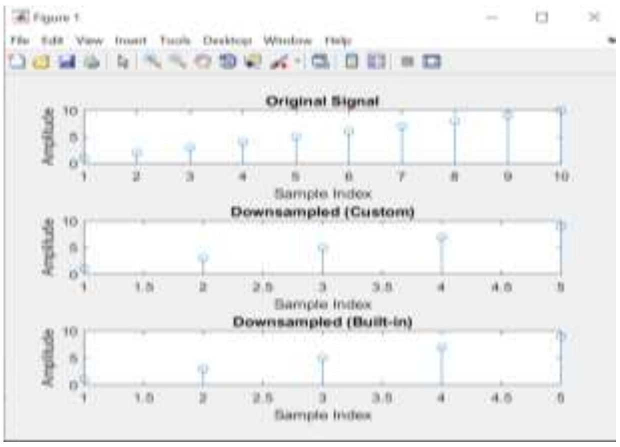
```matlab
% Generate a sample signal
signal = 1:10;
downsampled_custom = my_downsample(signal);
downsampled_builtin = downsample(signal, 2);
subplot(3, 1, 1);
stem(signal);
title('Original Signal');
xlabel('Sample Index');
ylabel('Amplitude');
subplot(3, 1, 2);
stem(downsampled_custom);
title('Downsampled (Custom)');
xlabel('Sample Index');
ylabel('Amplitude');
subplot(3, 1, 3);
stem(downsampled_builtin);
title('Downsampled (Built-in)');
xlabel('Sample Index');
ylabel('Amplitude');

% Display the plot
sgtitle('Comparison of Downsampling');
```



```matlab
function downsampled_signal = my_downsample(signal)
    % Retain odd-numbered samples (downsampling by 2)
    downsampled_signal = signal(1:2:end);
end
```

*Figure 3: Function.*

## Output:



## Conclusion:

By implementing a custom downsampling function and verifying the results using MATLAB's built-in function, we ensure the correctness of the downsampling process.

## TASK 09:

**Write your own function to upsample a signal i.e. copy the 1st sample of original signal in the new signal and then place an extra sample of 0, copy the 2nd sample of original signal and then place a 0, and so on. See Figure 4.10 for example. Call this function from a MATLAB file. Verify your result by using the command upsample. Plot the original signal, upsampled signal determined by your program, and upsampled signal obtained by the command upsample.**

**a) Call this function from a MATLAB file. Verify your result by using the built-in command “upsample”. Plot the original signal, upsampled signal determined by your function upsamp, and upsampled signal obtained by the command upsample.**
**b) Modify your function to work as generic up sampling function that takes both the input signal and the sampling factor from the user. c) Your function must also perform other up sampling methods such as instead of 0, the new sample is the copy of preceding or succeeding sample of the original signal or the new sample is the average of both. Check for possibility new up sampling methods by comparing the samples in the input signal.**

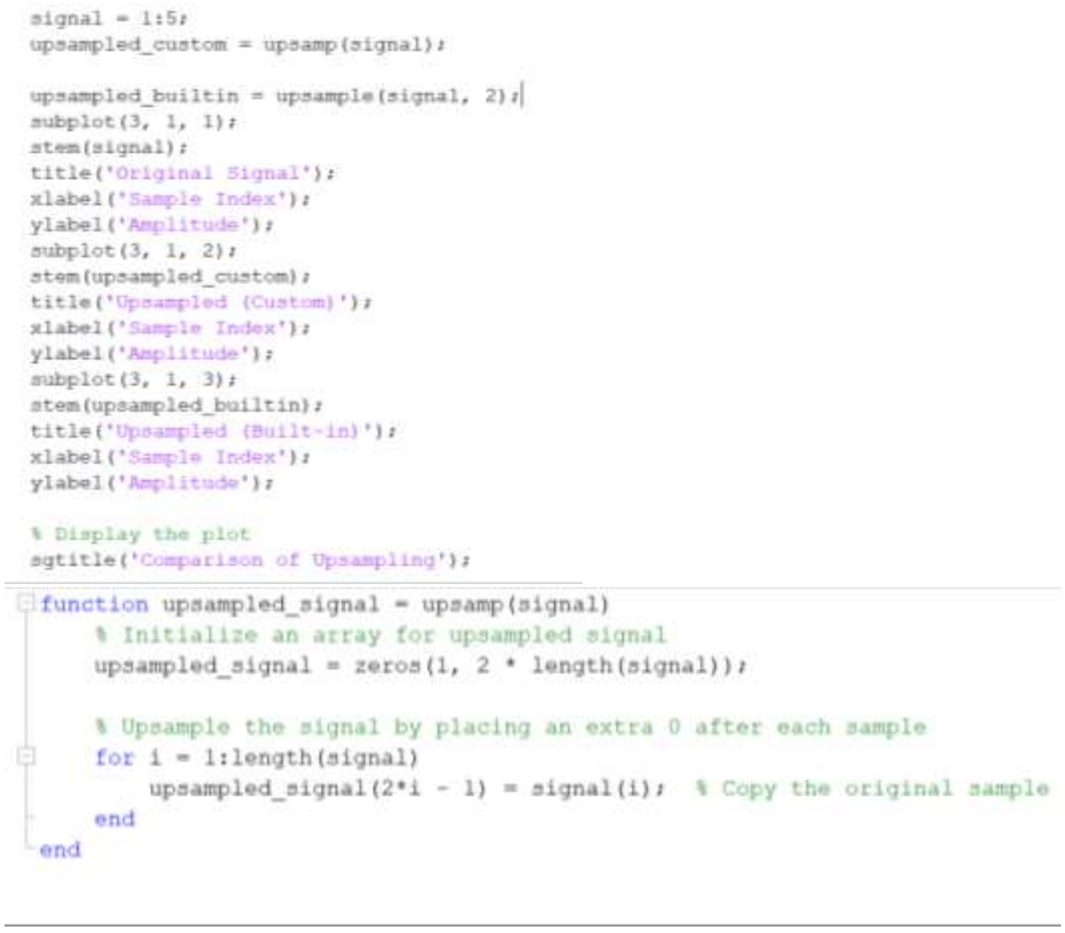
## Problem Analysis:

This task involves creating a function to upsample a signal and providing options for different upsampling methods.

## Algorithm:

1. Define a function to upsample a signal.
2. Copy each sample of the original signal to the new signal.
3. Insert a zero after each sample.
4. Return the upsampled signal.

**Code**

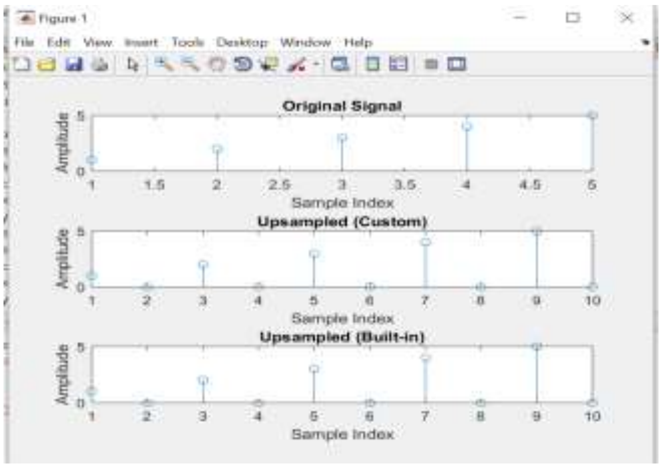


```
signal = 1:5;
upsampled_custom = upsamp(signal);

upsampled_builtin = upsample(signal, 2);
subplot(3, 1, 1);
stem(signal);
title('Original Signal');
xlabel('Sample Index');
ylabel('Amplitude');
subplot(3, 1, 2);
stem(upsampled_custom);
title('Upsampled (Custom)');
xlabel('Sample Index');
ylabel('Amplitude');
subplot(3, 1, 3);
stem(upsampled_builtin);
title('Upsampled (Built-in)');
xlabel('Sample Index');
ylabel('Amplitude');

% Display the plot
sgtitle('Comparison of Upsampling');
```

```
function upsampled_signal = upsamp(signal)
    % Initialize an array for upsampled signal
    upsampled_signal = zeros(1, 2 * length(signal));

    % Upsample the signal by placing an extra 0 after each sample
    for i = 1:length(signal)
        upsampled_signal(2*i - 1) = signal(i);  % Copy the original sample
    end
end
```

*Figure 4: Function Unsampled Signal*

**Output:**



## Conclusion:

By implementing a custom upsampling function and exploring different upsampling methods, we demonstrate the flexibility of signal processing techniques.

## TASK 10

**Plotting 3-D graphics with MATLAB is highlighted in this problem. This is a complementary task for practicing 3D graphs in MATLAB. Surf command is used in MATLAB for plotting 3D graphs, the meshgrid command is used for setting up 2D plane. clear all**
**close all**
**% set up 2-D plane by creating a -2:.2:2 sequence and copying it to all rows of x size (-2:.2:2) times and vice versa [x,y] = meshgrid([-2:.2:2]);**
**% plot 3D on plane Z = x.*exp(-x.^2-y.^2); figure**
**% surface plot, with gradient(Z) determining color distribution**
**surf(x,y,Z,gradient(Z))**
**% display color scale, can adjust location similarly to legend color bar.**

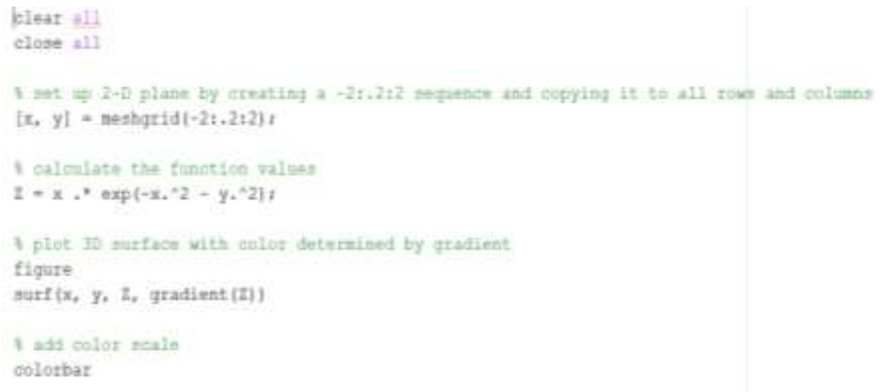
## Problem Analysis:

This task involves plotting a 3D graph using MATLAB's **surf** command.

### Algorithm:

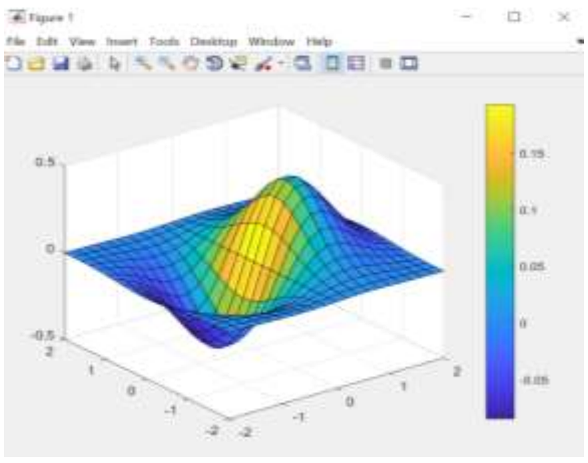
1. Define a range of *x* and *y* values using meshgrid.
2. Calculate *z* values based on the given function.
3. Plot the 3D graph using the surf command.

### Code



```
clear all
close all

% set up 2-D plane by creating a -2:.2:2 sequence and copying it to all rows and columns
[x, y] = meshgrid(-2:.2:2);

% calculate the function values
Z = x .* exp(-x.^2 - y.^2);

% plot 3D surface with color determined by gradient
figure
surf(x, y, Z, gradient(Z))

% add color scale
colorbar
```

### Output



## Conclusion:

By visualizing mathematical functions in three dimensions, we gain deeper insights into their behavior and relationships.