

**DIGITAL COMMUNICATIONS**

**(EC-431)**

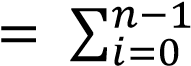
## **Student Name: AMINA QADEER**

**Degree/Syndicate: A**

**EC-431 Digital Communication Lab Grading Rubric**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Traits** | **Exceptional** **(9-10)** | **Acceptable** **(6-8)** | **Amateur**  **(3-5)** | **Unsatisfactory** **(0-2)** |
| **Tasks Completion**  **Weight = 40%** | All the lab tasks have been completed in class and shown in the report. | Most of the lab tasks have been completed and shown in the report. | A few of lab tasks have been completed and shown in the report. | No lab tasks or hardly any of the lab tasks have been completed and shown in the report. |
| **Tasks**  **Specifications**  **Weight = 30%** | The code work properly, shows perfect outputs and meets all the required specifications. | The code works somewhat properly and produces correct results and meets most of the specifications. | The code works somewhat properly but barely produces any correct results and hardly meets any specifications. | The code neither works properly nor produces correct results. And it does  not meet any specifications. |
| **Timeliness**  **Weight = 10%** | The report is submitted in time within the submission deadline. | The report is submitted within an hour of the deadline. | The report is submitted within a day of the deadline. | The report is submitted within a week of the deadline. |
| **Report**  **(Documentation)**  **Weight = 20%** | The report is well written with proper commenting and indentation, and separate functions /header files, and clearly explains what the code is accomplishing and how. | The report is written in a concise manner with appropriate comments and indentation. However, the explanation is somewhat useful for the reader and does not explain every aspect of the code. | The report is written in a simple manner with partial comments and improper indentation. The explanation makes it somewhat challenging for the reader to read and understand the code. | The report is written in an unclear manner, with no indentation and comments, which makes it very difficult for the reader to read and understand the code. |

**\*In all cases copied work will not be graded**

𝑺𝒄𝒐𝒓𝒆  𝑡𝑟𝑎𝑖𝑡\_𝑤𝑒𝑖𝑔ℎ𝑡𝑖 ∗ 𝑡𝑟𝑎𝑖𝑡\_𝑠𝑐𝑜𝑟𝑒𝑖

### Table of Contents

[Lab # 01: MATLAB Basics for Communication System Design 5](file:///C:\Users\Amina%20Qadeer\OneDrive%20-%20National%20University%20of%20Sciences%20&amp;%20Technology\Downloads\Lab_Manual_EC-431_DC%20-%20Formatted.pdf#_Toc44045)

[Lab # 02: Communication Signals: Generation and Interpretation 12](file:///C:\Users\Amina%20Qadeer\OneDrive%20-%20National%20University%20of%20Sciences%20&amp;%20Technology\Downloads\Lab_Manual_EC-431_DC%20-%20Formatted.pdf#_Toc44046)

[Lab # 03: Communication Signals: Signal Operations 22](file:///C:\Users\Amina%20Qadeer\OneDrive%20-%20National%20University%20of%20Sciences%20&amp;%20Technology\Downloads\Lab_Manual_EC-431_DC%20-%20Formatted.pdf#_Toc44047)

[Lab # 04: Communication Signals: Sampling Theorem and Aliasing 28](file:///C:\Users\Amina%20Qadeer\OneDrive%20-%20National%20University%20of%20Sciences%20&amp;%20Technology\Downloads\Lab_Manual_EC-431_DC%20-%20Formatted.pdf#_Toc44048)

[Lab # 05: Simulation of a Simple Communication System in MATLAB 37](file:///C:\Users\Amina%20Qadeer\OneDrive%20-%20National%20University%20of%20Sciences%20&amp;%20Technology\Downloads\Lab_Manual_EC-431_DC%20-%20Formatted.pdf#_Toc44049)

[Lab # 06: Calculation of Bit Error Rate in a Simple Communication System 40](file:///C:\Users\Amina%20Qadeer\OneDrive%20-%20National%20University%20of%20Sciences%20&amp;%20Technology\Downloads\Lab_Manual_EC-431_DC%20-%20Formatted.pdf#_Toc44050)

[Lab # 07: Generation of Bit Streams for Different Line Codes 43](file:///C:\Users\Amina%20Qadeer\OneDrive%20-%20National%20University%20of%20Sciences%20&amp;%20Technology\Downloads\Lab_Manual_EC-431_DC%20-%20Formatted.pdf#_Toc44051)

[Lab # 08: Calculating PSD Estimates of Different Line Codes 45](file:///C:\Users\Amina%20Qadeer\OneDrive%20-%20National%20University%20of%20Sciences%20&amp;%20Technology\Downloads\Lab_Manual_EC-431_DC%20-%20Formatted.pdf#_Toc44052)

[Lab # 09: Plotting the Constellation Diagram for 8-PSK using Natural Mapping 46](file:///C:\Users\Amina%20Qadeer\OneDrive%20-%20National%20University%20of%20Sciences%20&amp;%20Technology\Downloads\Lab_Manual_EC-431_DC%20-%20Formatted.pdf#_Toc44053)

[Lab # 10: Plotting the Constellation Diagram for 8-PSK using Gray Mapping 48](file:///C:\Users\Amina%20Qadeer\OneDrive%20-%20National%20University%20of%20Sciences%20&amp;%20Technology\Downloads\Lab_Manual_EC-431_DC%20-%20Formatted.pdf#_Toc44054)

[Lab # 11: Plotting the Constellation Diagram for 16-PSK using Natural Mapping 50](file:///C:\Users\Amina%20Qadeer\OneDrive%20-%20National%20University%20of%20Sciences%20&amp;%20Technology\Downloads\Lab_Manual_EC-431_DC%20-%20Formatted.pdf#_Toc44055)

[Lab # 12: Plotting the Constellation Diagram for 16-PSK using Gray Mapping 52](file:///C:\Users\Amina%20Qadeer\OneDrive%20-%20National%20University%20of%20Sciences%20&amp;%20Technology\Downloads\Lab_Manual_EC-431_DC%20-%20Formatted.pdf#_Toc44056)

[Lab # 13: Simulation of an Image Transmission through 8-ASK 54](file:///C:\Users\Amina%20Qadeer\OneDrive%20-%20National%20University%20of%20Sciences%20&amp;%20Technology\Downloads\Lab_Manual_EC-431_DC%20-%20Formatted.pdf#_Toc44057)

[Lab # 14: Simulation of an Image Transmission through 4-PSK 56](file:///C:\Users\Amina%20Qadeer\OneDrive%20-%20National%20University%20of%20Sciences%20&amp;%20Technology\Downloads\Lab_Manual_EC-431_DC%20-%20Formatted.pdf#_Toc44058)

# Lab # 01: MATLAB Basics for Communication System Design

**Objective:**

learn the basics of MATLAB and to develop understanding of MATLAB environment, commands, and syntax, so to solve communication engineering problems.

**LAB TASKS**

• Try to implement these two relations and show the result

**25 ( 3 ^1/3 ) + 2 (2+9 ^2) =**

result = 25\*(3^(1/3)) + 2\*(2+9^2);

disp(result)

202.0562

5x^3+ 3x^2+ 5x + 14 for x = 3

x = 3

result = 5\*x^3 + 3\*x^2 + 5\*x + 14

disp(result)

x =  
  
 3  
  
  
result =  
  
 191

* + Try to solve these.
    1. 6x + 12y + 4z = 70

7x – 2y + 3z= 5

2x + 8y -9z = 64

A=[6 12 4 ;7 -2 3;2 8 -9]

A =  
  
 6 12 4  
 7 -2 3  
 2 8 -9

B=[70;5;64]

B =  
  
 70  
 5  
 64

solution-linesolve(A,B)

solution =  
  
 3  
 5  
 -2

* + 1. A = [2 3 4 5; 1 8 9 0; 2 3 1 3; 5 8 9 3] Solve 6A – 2I + A^2 =

1. clear all
2. A = [2, 3, 4, 5;
3. 1, 8, 9, 0;
4. 2, 3, 1, 3;
5. 5, 8, 9, 3];
6. A\_squared = A \* A;
7. I = eye(size(A));
8. equation\_result = 6\*A - 2\*I + A\_squared;
9. A\_solution = equation\_result / 6;
10. disp(A\_solution)
11. 8.3333 16.6667 18.0000 11.1667  
     5.6667 23.3333 23.1667 5.3333  
     6.0000 12.5000 11.1667 6.6667  
     13.5000 29.6667 30.3333 12.8333

* + Try this one:

>> A= [ones(1,3), [2:2:10], zeros(1,3)]

A= [ones(1,3), [2:2:10], zeros(1,3)]

A =  
  
 1 1 1 2 4 6 8 10 0 0 0

What is the length and size of this?

length(A)

ans =  
  
 11

size(A)

ans =  
  
 1 11

* + Try to develop a function that will compute the maximum and minimum of two numbers

function[max,min]=max\_min(num1,num2)

if num1>num2

max=num1;

else

max=num2;

end

if num1<num2

min=num1;

else

min=num2;

end

end

function calling:

[Maxnum,Minnum]=max\_min(100,50);

disp(['Maximum number :',num2str(Maxnum)])

Maximum number :100

disp(['Minimum number :', num2str(Minnum)])

Minimum number :50

# Lab # 02: Communication Signals: Generation and Interpretation

**Objective:**

To learn the use of MATLAB for the generation of different communication signals and develop understanding of basic operations and properties of those signals.

**Generation of Signals:**

Signals are represented mathematically as a function of one or more independent variables. We will generally refer to the independent variable as time. Therefore, we can say a signal is a function of time. Write these instructions in m-file as execute to see the result.

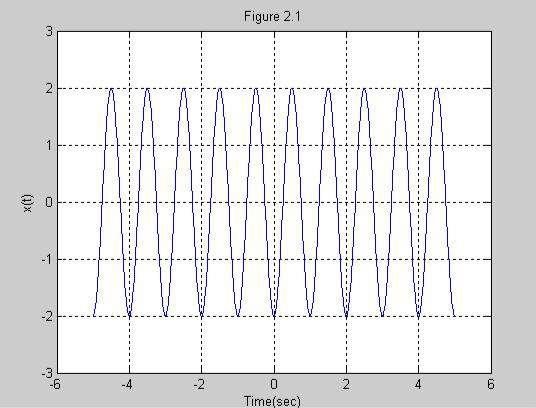
**Sinusoidal Sequence:**

% Example 2.1

% Generation of sinusoidal signals

% 2sin( 2πτ-π/2) t=[-5:0.01:5]; x=2\*sin((2\*pi\*t)-(pi/2));

plot(t,x) grid on; axis([-6 6 -3 3]) ylabel ('x(t)') xlabel ('Time(sec)') title ('Figure 2.1')



#### Figure 2.1

See the output, change the phase shift value, and observe the differences.

t = -5:0.01:5;

x = 2\*sin(2\*pi\*t - (pi/4));

figure;

plot(t, x);

grid on;

axis([-6 6 -3 3]);

ylabel('x(t)');

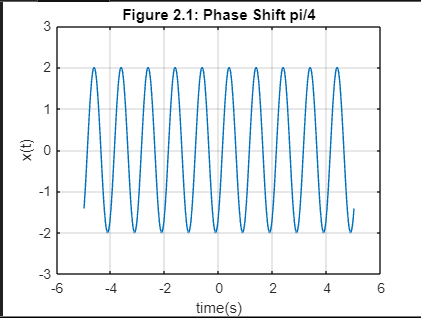
xlabel('time(s)');

title('Figure 2.1: Phase Shift pi/4');

title('Figure 2.1')

t = -5:0.01:5;

x = 2\*sin(2\*pi\*t - (pi/4));



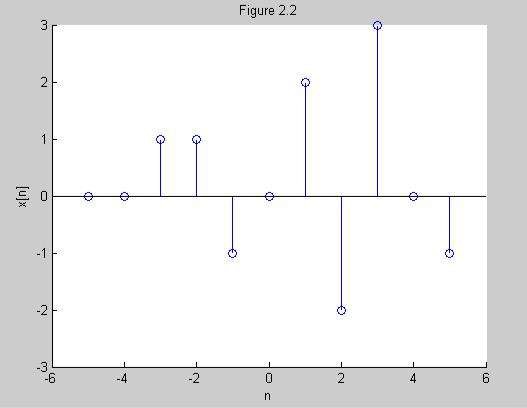
**Discrete Time Sequences:** See the example below:

% Example 2.2

% Generation of discrete time signals

n = [-5:5];

x = [0 0 1 1 -1 0 2 -2 3 0 -1]; stem (n,x); axis ([-6 6 -3 3]); xlabel ('n'); ylabel ('x[n]'); title ('Figure 2.2');



#### Figure 2.2

**Unit Impulse Sequence:**

A unit impulse sequence is defined as

*Delta (n) = 1 n = 0*

*= 0 n* ≠ *0*

We are making a function named **imseq** and we further use this function in next experiments of this lab. The MATLAB code is given below:

function [x,n] = impseq(n0,n1,n2)

% Generates x(n) = delta (n-n0); n1<=n, n0 <= n2

% x[x,n] = imseq(n0,n1,n2)

% n0 = impulse position, n1 = starting index, n2 = ending index if ((n0 < n1) | (n0 > n2) | (n1 > n2))

Error('arguments must satisfy n1 <= n0 <=n2') end n = [n1:n2];

% x = [zeros(1,(n0-n1)),1,zeros(1,(n2-n0))]; x = [(n-n0) == 0];

end

**Unit Step Sequence:**

It is defined as *u(n) = 1 n ≥ 0*

*= 0 n < 0*

The MATLAB code for stem sequence function **stepseq** is to be implemented by the class.

**Real Valued Exponential Sequence:**

It is defined as:

*x (n) = an, for all n; a € Real numbers*

We require an array operator “ .^ ” to implement a real exponential sequence. See the MATLAB code below:

>> n = [0:10];

>> x = (0.9).^n;

Observe the result/output.

**Complex Valued Exponential Sequence:**

It is defined as: *x(n) = e (a + jb) n , for all n*

Where **a** is called the *attenuation* and **b** is the *frequency in radians*. It can be implemented by the following MATLAB script.

>> n = [0:10];

>> x = exp ((2+3j)\*n);

**Random Sequence:**

Many practical sequences cannot be described by the mathematical expressions like above, these are called *random sequences*. In MATLAB two types of random sequences are available. See the code below:

>> rand (1,N)

>> randn (1,N)

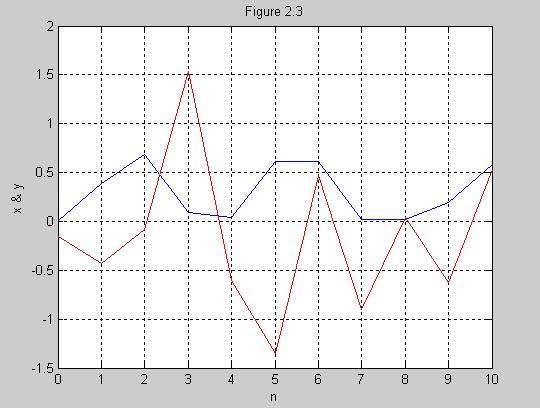
The above instruction **rand** generates *a length N random sequence whose elements are uniformly distributed between [0,1]*. And the last instruction, **randn** generates *a length N Gaussian random sequence with mean 0 and variance 1.* Plot these sequences and observe the difference.

% Example 2.3

%Generation of random sequence

n = [0:10]; x = rand (1, length (n)); y = randn (1, length (n)); plot (n,x) ;

grid on; hold on; plot(n,y,'r'); ylabel ('x & y') xlabel ('n') title ('Figure 2.3')



#### Figure 2.3

**Periodic Sequence:**

clear all

% Generation of periodic sequences

n = [0:4];

x = [1, 1, 2, -1, 0];

% Plotting the original sequence

subplot(2,1,1);

stem(n, x);

grid on;

axis([0 14 -1 2]);

xlabel('n');

ylabel('x(n)');

title('Figure 2.4(a)');

% Generating the periodic sequence

xtilde = [x, x, x];

length\_xtilde = length(xtilde);

n\_new = 0:length\_xtilde-1;

% Plotting the periodic sequence

subplot(2,1,2);

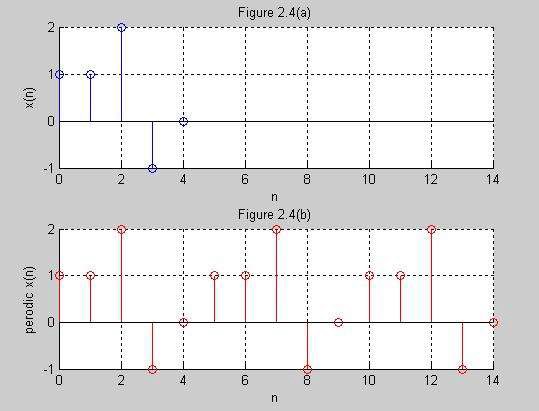
stem(n\_new, xtilde, 'r');

grid on;

xlabel('n');

ylabel('periodic x(n)');

title('Figure 2.4(b)');



#### Figure 2.4

**SIGNALS OPERATIONS:**

**Function:**

function [y, n] = sigadd(x1, n1, x2, n2)

% Signal addition function

% Determine the range of indices for the combined signal

n\_start = min(min(n1), min(n2));

n\_end = max(max(n1), max(n2));

n = n\_start:n\_end;

% Interpolate the signals to align them at the same indices

y1 = interp1(n1, x1, n);

y2 = interp1(n2, x2, n);

% Add the interpolated signals

y = y1 + y2;

end

clear; clc;

n1 = 0:10;

x1 = sin(n1);

n2 = -5:7;

x2 = 4\*sin(n2);

[y, n] = sigadd(x1, n1, x2, n2);

subplot(3,1,1);

stem(n1, x1);

grid on;

axis([-5 10 -5 5]);

xlabel('n1');

ylabel('x1(n)');

title('1st signal');

subplot(3,1,2);

stem(n2, x2);

grid on;

axis([-5 10 -5 5]);

xlabel('n2');

ylabel('x2(n)');

title('2nd signal');

subplot(3,1,3);

stem(n, y, 'r');

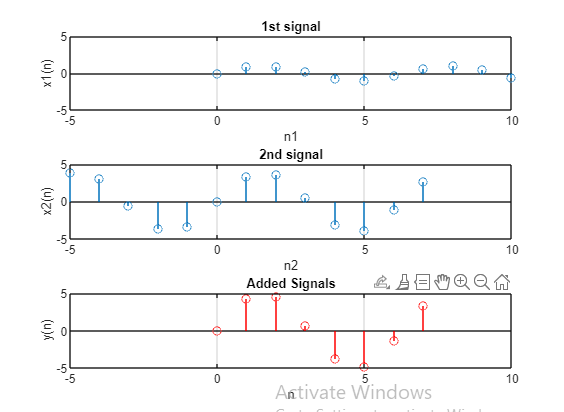
grid on;

axis([-5 10 -5 5]);

xlabel('n');

ylabel('y(n)');

title('Added Signals');



#### Figure 2.5

**Signal Multiplication:**

Function:

function [y, n] = sigmult(x1, n1, x2, n2)

% Signal multiplication function

% Determine the range of indices for the combined signal

n\_start = min(min(n1), min(n2));

n\_end = max(max(n1), max(n2));

n = n\_start:n\_end;

% Interpolate the signals to align them at the same indices

y1 = interp1(n1, x1, n);

y2 = interp1(n2, x2, n);

% Multiply the interpolated signals

y = y1 .\* y2;

end

% Example 2.6

% signal multiplication using sigmult function

clear; clc;

% Define the time indices and signals

n1 = 0:0.1:10;

x1 = sin(n1);

n2 = -5:0.1:7;

x2 = 4\*sin(n2);

% Perform signal multiplication using sigmult function

[y, n] = sigmult(x1, n1, x2, n2);

% Plot the original signals

subplot(3,1,1);

stem(n1, x1);

grid on;

axis([-5 10 -5 5]);

xlabel('n1');

ylabel('x1(n)');

title('1st signal');

subplot(3,1,2);

stem(n2, x2);

grid on;

hold on;

axis([-5 10 -5 5]);

xlabel('n2');

ylabel('x2(n)');

title('2nd signal');

% Plot the multiplied signal

subplot(3,1,3);

stem(n, y, 'r');

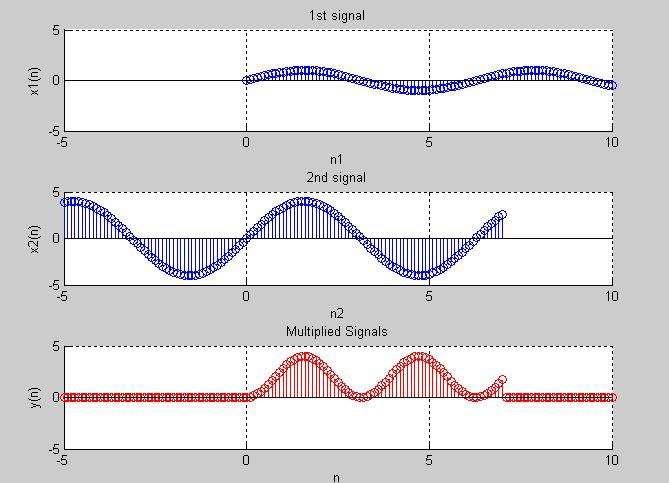
grid on;

axis([-5 10 -5 5]);

xlabel('n');

ylabel('y(n)');

title('Multiplied Signals');



#### Figure 2.6

**LAB TASKS:**

1. Write a generic MATLAB code for the ‘stepseq’ function, that generates a step sequence.

**Function:**

function [x, n] = stepseq(n0, n1, n2)

% Generates a step sequence u(n)

% Check if arguments satisfy the condition

if n0 < n1 || n0 > n2 || n1 > n2

error('Arguments must satisfy n1 <= n0 <= n2');

end

% Generate the time index vector

n = n1:n2;

% Generate the step sequence

x = zeros(size(n));

x(n >= n0) = 1;

end

Calling function:

[n, u] = stepseq(0, -5, 5);

stem(n, u);

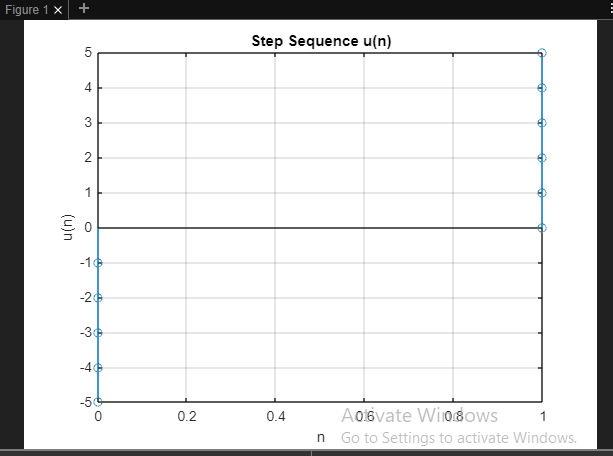
xlabel('n');

ylabel('u(n)');

title('Step Sequence u(n)');

grid on

**Output:**



1. Write a generic MATLAB code for the ‘sigmult’ function, that multiplies two signals/sequences together. (Use example 2.6 for code testing)

Function:

function [y, n] = sigmult(x1, n1, x2, n2)

% Signal multiplication function

% Determine the range of indices for the combined signal

n\_start = max(min(n1), min(n2));

n\_end = min(max(n1), max(n2));

n = n\_start:n\_end;

% Interpolate the signals to align them at the same indices

y1 = interp1(n1, x1, n);

y2 = interp1(n2, x2, n);

% Multiply the interpolated signals

y = y1 .\* y2;

end

Calling function in command window:

clear all

 % Example 2.6

% signal multiplication using sigmult function

clear; clc;

n1 = 0:0.1:10;

x1 = sin(n1);

n2 = -5:0.1:7;

x2 = 4\*sin(n2);

[y, n] = sigmult(x1, n1, x2, n2);

subplot(3,1,1);

stem(n1, x1);

grid on;

axis([-5 10 -5 5]);

xlabel('n1');

ylabel('x1(n)');

title('1st signal');

subplot(3,1,2);

stem(n2, x2);

grid on;

hold on;

axis([-5 10 -5 5]);

xlabel('n2');

ylabel('x2(n)');

title('2nd signal');

subplot(3,1,3);

stem(n, y, 'r');

grid on;

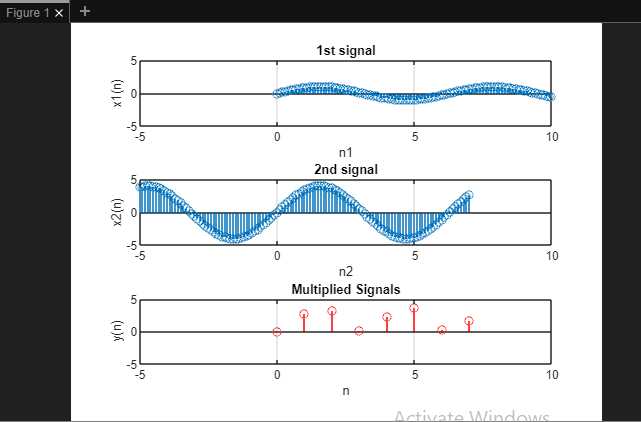
axis([-5 10 -5 5]);

xlabel('n');

ylabel('y(n)');

title('Multiplied Signals');

Output:



1. Generate the following sequences.

x [n] = 2sin (3n) + 2cos (3n) x [n] = u[n] + 4cos (3n)

Note: You are not allowed to multiply impulse sequences with a number. Implement this by using **impseq, stepseq, sigadd,** and **sigmult** functions.

Functions:

function [x, n] = stepseq(n0, n1, n2)

% Generates a step sequence u(n)

% Check if arguments satisfy the condition

if n0 < n1 || n0 > n2 || n1 > n2

error('Arguments must satisfy n1 <= n0 <= n2');

end

% Generate the time index vector

n = n1:n2;

% Generate the step sequence

x = zeros(size(n));

x(n >= n0) = 1;

end

n = -10:10;

function [y, n] = sigmult(x1, n1, x2, n2)

% Signal multiplication function

% Determine the range of indices for the combined signal

n\_start = max(min(n1), min(n2));

n\_end = min(max(n1), max(n2));

n = n\_start:n\_end;

% Interpolate the signals to align them at the same indices

y1 = interp1(n1, x1, n);

y2 = interp1(n2, x2, n);

% Multiply the interpolated signals

y = y1 .\* y2;

end

function [y, n] = sigadd(x1, n1, x2, n2)

% Signal addition function

% Determine the range of indices for the combined signal

n\_start = min(min(n1), min(n2));

n\_end = max(max(n1), max(n2));

n = n\_start:n\_end;

% Interpolate the signals to align them at the same indices

y1 = interp1(n1, x1, n);

y2 = interp1(n2, x2, n);

% Add the interpolated signals

y = y1 + y2;

end

function [x,n] = impseq(n0,n1,n2)

% Generates x(n) = delta(n-n0); n1<=n, n0 <= n2

% [x,n] = impseq(n0,n1,n2)

% n0 = impulse position, n1 = starting index, n2 = ending index

if ((n0 < n1) || (n0 > n2) || (n1 > n2))

error('Arguments must satisfy n1 <= n0 <= n2');

end

n = (n1:n2);

x = (n - n0) == 0;

end

**Code command window:**

clear all

% Generate the first sequence x1[n] = 2sin(3n) + 2cos(3n)

n = -10:10;

x1 = 2\*sin(3\*n) + 2\*cos(3\*n);

% Generate the second sequence x2[n] = u[n] + 4cos(3n)

[x2, ~] = sigadd(stepseq(0, -10, 10), -10:10, 4\*cos(3\*n), -10:10);

% Plot the first sequence

subplot(2, 1, 1);

stem(n, x1);

title('x1[n] = 2sin(3n) + 2cos(3n)');

xlabel('n');

ylabel('Amplitude');

% Plot the second sequence

subplot(2, 1, 2);

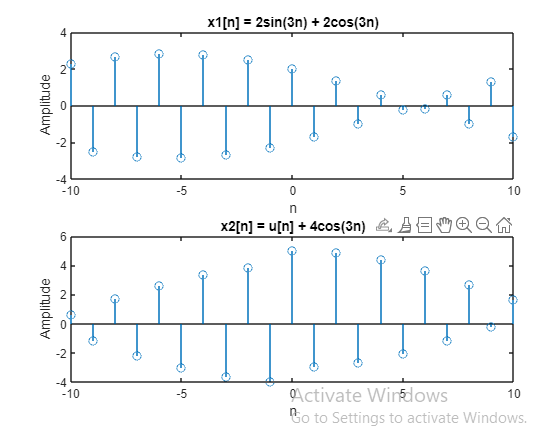
stem(n, x2);

title('x2[n] = u[n] + 4cos(3n)');

xlabel('n');

ylabel('Amplitude');

**Figure:**



# Lab # 03: Communication Signals: Signal Operations

**Objective:**

To learn the use of MATLAB for learn the use of MATLAB for different operations such as Scaling, Shifting, Folding, Sample Summation, Sample product, Energy, Even and Odd sequences.

**Signal Operations:**

**Scaling:**

function [x, n] = stepseq(n0, n1, n2)

% Generates a step sequence u(n)

% Check if arguments satisfy the condition

if n0 < n1 || n0 > n2 || n1 > n2

error('Arguments must satisfy n1 <= n0 <= n2');

end

x=n1:0.01:n2;

n=x-n0>=0;

end

% Define the signal x(n) using the provided function stepseq

[x, n] = stepseq(-1, -5, 5);

% Define the scaling factor

a = 2;

% Perform scaling operation

y = a \* x;

% Plot the original signal

subplot(2, 1, 1);

stem(n, x);

title('Original Signal x(n)');

xlabel('n');

ylabel('Amplitude');

grid on;

% Plot the scaled signal

subplot(2, 1, 2);

stem(n, y, 'r');

title('Scaled Signal y(n) = 2 \* x(n)');

xlabel('n');

ylabel('Amplitude');

grid on;

**Shifting:**

**Function:**

**Example 3.1**

% Define the original signal using the provided function stepseq

[x, n] = stepseq(0, -10, 10);

% Plot the original signal

subplot(3, 2, 1);

stem(n, x);

axis([-12 12 0 3]);

grid on;

xlabel('n');

ylabel('x(n)');

title('Original Signal');

% Shift the signal by 2.5 units

[y1, n1] = sigshift(x, n, 2.5);

subplot(3, 2, 2);

stem(n1, y1);

axis([-12 12 0 3]);

grid on;

xlabel('n');

ylabel('y1(n)');

title('Shifted signal, x(n-2.5)');

% Shift the signal by -2.5 units

[y2, n2] = sigshift(x, n, -2.5);

subplot(3, 2, 4);

stem(n2, y2);

axis([-12 12 0 3]);

grid on;

xlabel('n');

ylabel('y2(n)');

title('Shifted signal, x(n+2.5)');

% Add the shifted signals

[y\_add, n\_add] = sigadd(y1, n1, y2, n2);

subplot(3, 2, 6);

stem(n\_add, y\_add, 'r');

axis([-12 12 0 3]);

grid on;

xlabel('n');

ylabel('y1(n) + y2(n)');

title('Added Signal');

% Multiply the shifted signals

[y\_mul, n\_mul] = sigmult(y1, n1, y2, n2);

subplot(3, 2, 5);

stem(n\_mul, y\_mul, 'k');

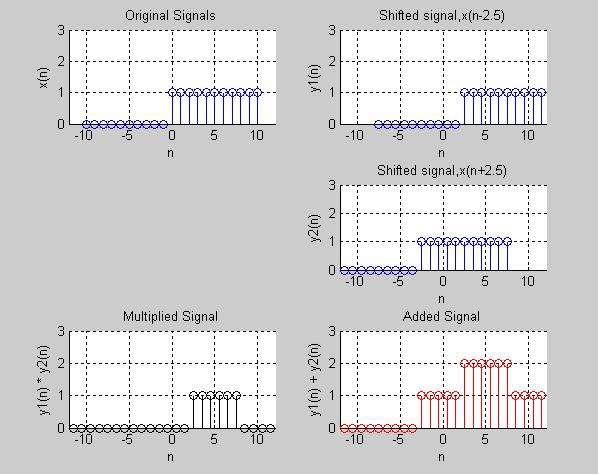
axis([-12 12 0 3]);

grid on;

xlabel('n');

ylabel('y1(n) \* y2(n)');

title('Multiplied Signal');



#### Figure 3.1

**Folding:**

% Define the original signal x(n)

n = -10:10;

x = 2\*sin(3\*n) + 2\*cos(3\*n);

% Fold the signal

y = fliplr(x);

% Plot the original and folded signals

subplot(2, 1, 1);

stem(n, x);

title('Original Signal x(n)');

xlabel('n');

ylabel('Amplitude');

grid on;

subplot(2, 1, 2);

stem(n, y);

title('Folded Signal y(n) = x(-n)');

xlabel('n');

ylabel('Amplitude');

grid on;

**Sample Summation:**

% Define the signal x(n) using the provided function stepseq

[x, n] = stepseq(5, 0, 10);

% Calculate the sum of samples between indices 2 and 7

sum\_result = sum(x(2:7));

% Display the result

disp(['Sum of samples between indices 2 and 7: ' num2str(sum\_result)]);

**Sample Product:**

% Define the signal x

x = [0 1 2 3 4 5];

% Calculate the product of samples between indices 2 and 5

product\_result = prod(x(2:5));

% Display the result

disp(['Product of samples between indices 2 and 5: ' num2str(product\_result)]);

**Energy:**

% Define the signal x

x = [0 1 2 3 4 5];

% Compute the energy using the first method

Ex1 = sum(x .\* conj(x));

% Compute the energy using the second method

Ex2 = sum(abs(x).^2);

% Display the results

disp(['Energy of the signal using the first method: ' num2str(Ex1)]);

disp(['Energy of the signal using the second method: ' num2str(Ex2)]);

**Even and Odd Sequence:**

% Generation of even and odd signals

n1 = 0:0.01:1;

x1 = 2 \* n1;

n2 = 1:0.01:2;

x2 = -2 \* n2 + 4;

n = [n1, n2];

x = [x1, x2];

% Even Signal

[xe, ne] = sigfold(x, n);

% Plotting of original signal

subplot(3, 1, 1);

plot(n, x);

axis([-4 4 0 2.5]);

grid on;

title('Original Signal');

xlabel('n');

ylabel('Amplitude');

% Plotting of original signal + even signal

subplot(3, 1, 2);

plot(n, x/2, ne, xe/2);

axis([-4 4 0 2.5]);

grid on;

title('Original Signal + Even Signal');

xlabel('n');

ylabel('Amplitude');

% Plotting of original signal + odd signal

xo = -xe;

no = ne;

subplot(3, 1, 3);

plot(n, x/2, no, xo/2);

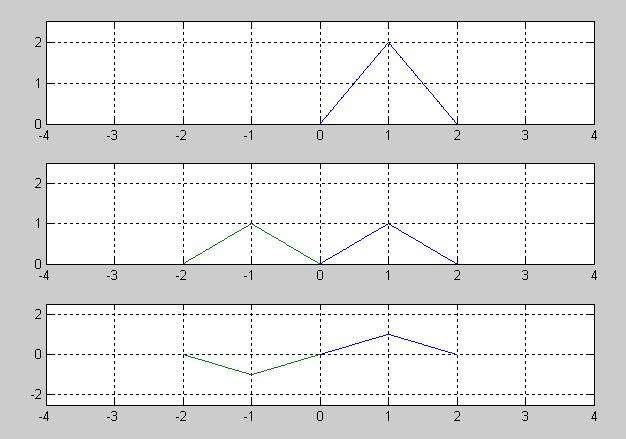
axis([-4 4 -2.5 2.5]);

grid on;

title('Original Signal + Odd Signal');

xlabel('n');

ylabel('Amplitude');



#### Figure 3.2

The above example shows to develop the even and odd signals from a given signal. Now develop a function to compute the even and odd signals for yourself. Here is the sample of how an ‘evenodd’ function would look like:

function [xe, xo, m] = evenodd(x, n)

% Decomposes a real function into its even and odd parts

% [xe, xo, m] = evenodd(x, n)

% xe = even signal

% xo = odd signal

% m = indexes

% x = original signal

% n = indexes for original signal

% Compute even and odd parts

xe = 0.5 \* (x + fliplr(x)); % xe = (x(n) + x(-n))/2

xo = 0.5 \* (x - fliplr(x)); % xo = (x(n) - x(-n))/2

% Generate indexes

m = min(n):max(n);

end

**LAB TASKS LAB 03:**

Run Example 3.1 and show its final output plots as shown in the manual. Use the desired functions i.e. stepseq, sigmult, sigadd, sigshift etc.

**Functions:**

function [y, n] = sigshift(x, m, n0)

% Implements y(n) = x(n - n0)

% x = samples of original signal

% m = index values of the signal

% n0 = shift amount, may be positive or negative

% [y, n] = sigshift(x, m, n0)

% Shift the indexes

n = m + n0;

% Output the shifted signal

y = x;

end

function [x,n] = impseq(n0,n1,n2)

% Generates x(n) = delta(n-n0); n1<=n, n0 <= n2

% [x,n] = impseq(n0,n1,n2)

% n0 = impulse position, n1 = starting index, n2 = ending index

if ((n0 < n1) || (n0 > n2) || (n1 > n2))

error('Arguments must satisfy n1 <= n0 <= n2');

end

n = (n1:n2);

x = (n - n0) == 0;

end

function [y, n] = sigadd(x1, n1, x2, n2)

% Signal addition function

% Determine the range of indices for the combined signal

n\_start = min(min(n1), min(n2));

n\_end = max(max(n1), max(n2));

n = n\_start:n\_end;

% Interpolate the signals to align them at the same indices

y1 = interp1(n1, x1, n);

y2 = interp1(n2, x2, n);

% Add the interpolated signals

y = y1 + y2;

end

function [x, n] = stepseq(n0, n1, n2)

% Generates a step sequence u(n)

% Check if arguments satisfy the condition

if n0 < n1 || n0 > n2 || n1 > n2

error('Arguments must satisfy n1 <= n0 <= n2');

end

% Generate the time index vector

n = n1:n2;

% Generate the step sequence

x = zeros(size(n));

x(n >= n0) = 1;

end

function [y, n] = sigmult(x1, n1, x2, n2)

% Signal multiplication function

% Determine the range of indices for the combined signal

n\_start = max(min(n1), min(n2));

n\_end = min(max(n1), max(n2));

n = n\_start:n\_end;

% Interpolate the signals to align them at the same indices

y1 = interp1(n1, x1, n);

y2 = interp1(n2, x2, n);

% Multiply the interpolated signals

y = y1 .\* y2;

end

**CODE COMMAND WINDOW:**

clear all

% Define the original signal using the provided function stepseq

[x, n] = stepseq(0, -10, 10);

% Plot the original signal

subplot(3, 2, 1);

stem(n, x);

axis([-12 12 0 3]);

grid on;

xlabel('n');

ylabel('x(n)');

title('Original Signal');

% Shift the signal by 2.5 units

[y1, n1] = sigshift(x, n, 2.5);

subplot(3, 2, 2);

stem(n1, y1);

axis([-12 12 0 3]);

grid on;

xlabel('n');

ylabel('y1(n)');

title('Shifted signal, x(n-2.5)');

% Shift the signal by -2.5 units

[y2, n2] = sigshift(x, n, -2.5);

subplot(3, 2, 4);

stem(n2, y2);

axis([-12 12 0 3]);

grid on;

xlabel('n');

ylabel('y2(n)');

title('Shifted signal, x(n+2.5)');

% Add the shifted signals

[y\_add, n\_add] = sigadd(y1, n1, y2, n2);

subplot(3, 2, 6);

stem(n\_add, y\_add, 'r');

axis([-12 12 0 3]);

grid on;

xlabel('n');

ylabel('y1(n) + y2(n)');

title('Added Signal');

% Multiply the shifted signals

[y\_mul, n\_mul] = sigmult(y1, n1, y2, n2);

subplot(3, 2, 5);

stem(n\_mul, y\_mul, 'k');

axis([-12 12 0 3]);

grid on;

xlabel('n');

ylabel('y1(n) \* y2(n)');

title('Multiplied Signal');

A screenshot of a computer

Description automatically generated

2.Write a generic MATLAB code for the ‘sigfold’ function. Use any example to show its working.

**Function:**

function [y, n] = sigfold(x, m)

% Flips the signal x(n) around n = 0

% y(n) = x(-n)

% [y, n] = sigfold(x, m)

% Flip the indexes

n = -fliplr(m);

% Flip the signal

y = fliplr(x);

end

**Code command window:**

 % Example usage

% Define a signal x and its indexes

x = [1, 2, 3, 4];

m = 0:length(x)-1;

% Use sigfold function to fold the signal

[y, n] = sigfold(x, m);

% Plot the original and folded signals

subplot(2, 1, 1);

stem(m, x, 'b', 'LineWidth', 1.5);

xlabel('n');

ylabel('x(n)');

title('Original Signal');

grid on;

subplot(2, 1, 2);

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

xlabel('n');

ylabel('y(n)');

title('Folded Signal');

grid on;

**A screenshot of a graph

Description automatically generated**

3Write a generic MATLAB code for the ‘evenodd’ function, that decomposes a real function into its even and odd parts. Test it on example 3.2, by making desired changes and get the same output (See the sample of ‘evenodd’ function is given in manual).

function [xe, xo, m] = evenodd(x, n)

% Decomposes a real function into its even and odd parts

% [xe, xo, m] = evenodd(x, n)

% xe = even signal

% xo = odd signal

% m = indexes

% x = original signal

% n = indexes for original signal

% Compute even and odd parts

xe = 0.5 \* (x + fliplr(x)); % xe = (x(n) + x(-n))/2

xo = 0.5 \* (x - fliplr(x)); % xo = (x(n) - x(-n))/2

% Generate indexes

m = min(n):max(n);

end

function [y, n] = sigfold(x, m)

% Flips the signal x(n) around n = 0

% y(n) = x(-n)

% [y, n] = sigfold(x, m)

% Flip the indexes

n = -fliplr(m);

% Flip the signal

y = fliplr(x);

end

clear all

% Generation of even and odd signals

n1 = 0:0.01:1;

x1 = 2 \* n1;

n2 = 1:0.01:2;

x2 = -2 \* n2 + 4;

n = [n1, n2];

x = [x1, x2];

% Even Signal

[xe, ne] = sigfold(x, n);

% Plotting of original signal

subplot(3, 1, 1);

plot(n, x);

axis([-4 4 0 2.5]);

grid on;

title('Original Signal');

xlabel('n');

ylabel('Amplitude');

% Plotting of original signal + even signal

subplot(3, 1, 2);

plot(n, x/2, ne, xe/2);

axis([-4 4 0 2.5]);

grid on;

title('Original Signal + Even Signal');

xlabel('n');

ylabel('Amplitude');

% Plotting of original signal + odd signal

xo = -xe;

no = ne;

subplot(3, 1, 3);

plot(n, x/2, no, xo/2);

axis([-4 4 -2.5 2.5]);

grid on;

title('Original Signal + Odd Signal');

xlabel('n');

ylabel('Amplitude');

A screenshot of a graph

Description automatically generated

4.Research assignment: Explain Sampling Theorem, Nyquist Frequency, Proper Sampling, Improper Sampling with an example. Also demonstrate the effects of Aliasing arising from improper sampling.

1. **Sampling Theorem**:
   * States that a continuous-time signal can be perfectly reconstructed from its samples if the sampling frequency is greater than twice the maximum frequency component in the signal.
   * Mathematically: *fs*​>2⋅ *fN*​, where *fs*​ is the sampling frequency and *fN*​ is the Nyquist frequency.
2. **Nyquist Frequency**:
   * Half of the sampling frequency.
   * Represents the maximum frequency that can be accurately represented in a discrete-time signal without introducing distortion or aliasing.
   * Mathematically: 2*fN*​=2*fs*​​.
3. **Proper Sampling**:
   * Occurs when the sampling frequency is greater than twice the maximum frequency present in the signal.
   * Ensures that no information is lost during the sampling process, allowing accurate reconstruction of the original signal from its samples.
4. **Improper Sampling**:
   * Occurs when the sampling frequency is less than twice the maximum frequency present in the signal.
   * Results in aliasing, where high-frequency components are incorrectly represented as lower-frequency components in the sampled signal.
5. **Aliasing**:
   * Phenomenon where high-frequency components in the original signal are incorrectly represented as lower-frequency components in the sampled signal due to improper sampling.
   * Leads to distortion and loss of information in the sampled signal.

# Lab # 04: Communication Signals: Sampling Theorem and Aliasing

**Objective:**

To learn about sampling theorem and demonstrate the effects of aliasing arising from improper sampling

**Explanation:**

The definition of proper sampling is quite simple. Suppose you sample a continuous signal in some manner. If you can exactly reconstruct the analog signal from the samples, you must have done the sampling properly. Even if the sampled data appears confusing or incomplete, the key information has been captured if you can reverse the process.

Figure 3-3 shows several sinusoids before and after digitization. The continuous line represents the analog signal entering the ADC, while the square markers are the digital signal leaving the ADC.

In (a), the analog signal is a constant DC value, a cosine wave of zero frequency. Since the analog signal is a series of straight lines between each of the samples, all the information needed to reconstruct the analog signal is contained in the digital data. According to our definition, this is proper sampling.

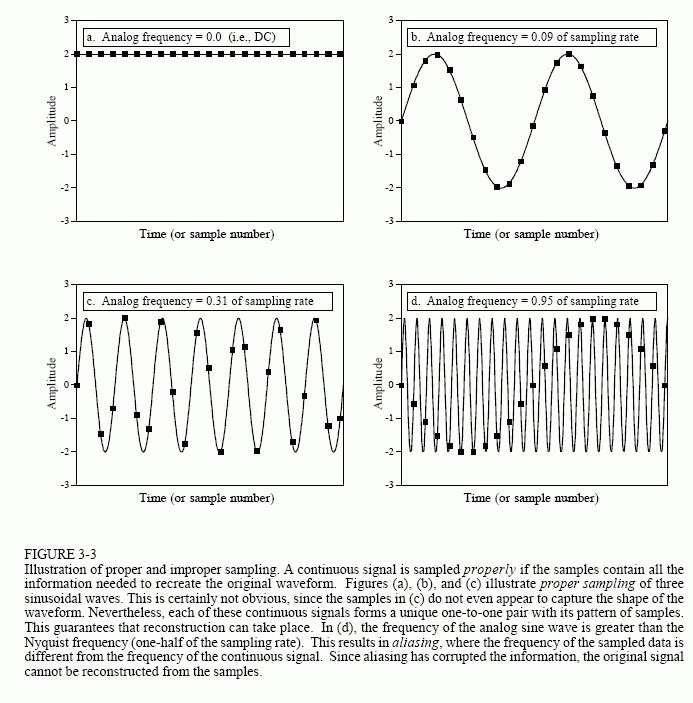
The sine wave shown in (b) has a frequency of 0.09 of the sampling rate. This might represent, for example, a 90 cycle/second sine wave being sampled at 1000 samples/second. Expressed in another way, there are 11.1 samples taken over each complete cycle of the sinusoid. This situation is more complicated than the previous case, because the analog signal cannot be reconstructed by simply drawing straight lines between the data points. Do these samples properly represent the analog signal? The answer is yes, because no other sinusoid, or combination of sinusoids, will produce this pattern of samples (within the reasonable constraints listed below). These samples correspond to only one analog signal, and therefore the analog signal can be exactly reconstructed. Again, an instance of proper sampling.

In (c), the situation is made more difficult by increasing the sine wave's frequency to 0.31 of the sampling rate. This results in only 3.2 samples per sine wave cycle. Here the samples are so sparse that they don't even appear to follow the general trend of the analog signal. Do these samples properly represent the analog waveform? Again, the answer is yes, and for the same reason. The samples are a unique representation of the analog signal. All the information needed to reconstruct the continuous waveform is contained in the digital data. How you go about doing this will be discussed later in this chapter. Obviously, it must be more sophisticated than just drawing straight lines between the data points. As strange as it seems, this is proper sampling according to our definition.

In (d), the analog frequency is pushed even higher to 0.95 of the sampling rate, with a mere 1.05 samples per sine wave cycle. Do these samples properly represent the data? No, they don't! The samples represent a different sine wave from the one contained in the analog signal. In particular, the original sine wave of 0.95 frequency misrepresents itself as a sine wave of 0.05 frequency in the digital signal. This phenomenon of sinusoids changing frequency during sampling is called **aliasing**. Just as a criminal might take on an assumed name or identity (an alias), the sinusoid assumes another frequency that is not its own. Since the digital data is no longer uniquely related to a particular analog signal, an unambiguous reconstruction is impossible. There is nothing in the sampled data to suggest that the original analog signal had a frequency of 0.95 rather than 0.05. The sine wave has hidden its identity completely; the perfect crime has been committed! According to our definition, this is an example of improper sampling.

This line of reasoning leads to a milestone in DSP, the sampling theorem. Frequently this is called the Shannon sampling theorem, or the Nyquist sampling theorem, after the authors of 1940s papers on the topic. The sampling theorem indicates that a continuous signal can be properly sampled, only if it does not contain frequency components above one-half of the sampling rate.

For instance, a sampling rate of 2,000 samples/second requires the analog signal to be composed of frequencies below 1000 cycles/second. If frequencies above this limit are present in the signal, they will be aliased to frequencies between 0 and 1000 cycles/second, combining with whatever information that was legitimately there.



Two terms are widely used when discussing the sampling theorem: the Nyquist frequency and the Nyquist rate. Unfortunately, their meaning is not standardized. To understand this, consider an analog signal composed of frequencies between DC and 3 kHz. To properly digitize this signal, it must be sampled at 6,000 samples/sec (6 kHz) or higher. Suppose we choose to sample at 8,000 samples/sec (8 kHz), allowing frequencies between DC and 4 kHz to be properly represented. In this situation there are four important frequencies: (1) the highest frequency in the signal, 3 kHz; (2) twice this frequency, 6 kHz; (3) the sampling rate, 8 kHz; and (4) one-half the sampling rate, 4 kHz. Which of these four is the Nyquist frequency and which is the Nyquist rate? It depends on who you ask! All the possible combinations are used. Fortunately, most authors are careful to define how they are using the terms. In this book, they are both used to mean one-half the sampling rate. Figure 3-4 shows how frequencies are changed during aliasing. The key point to remember is that a digital signal cannot contain frequencies above one-half the sampling rate (i.e., the Nyquist frequency/rate).

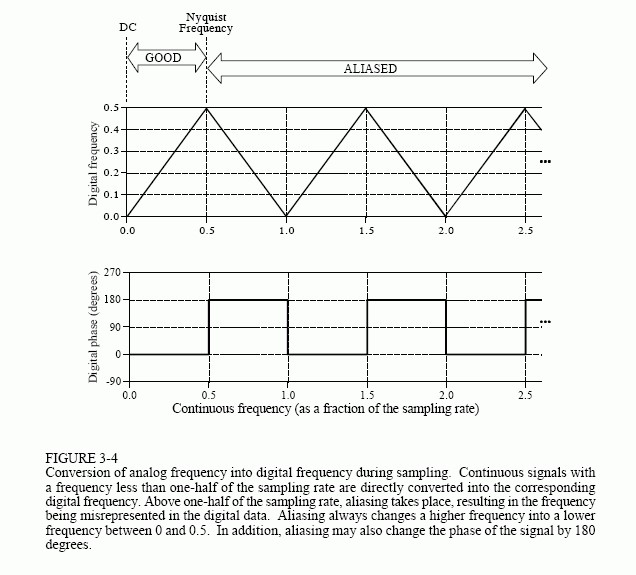
When the frequency of the continuous wave is below the Nyquist rate, the frequency of the sampled data is a match. However, when the continuous signal's frequency is above the Nyquist rate, aliasing changes the frequency into something that can be represented in the sampled data. As shown by the zigzagging line in Fig. 3-4, every continuous frequency above the Nyquist rate has a corresponding digital frequency between zero and one-half the sampling rate. It there happens to be a sinusoid already at this lower frequency, the aliased signal will add to it, resulting in a loss of information. Aliasing is a double curse; information can be lost about the higher and the lower frequency. Suppose you are given a digital signal containing a frequency of 0.2 of the sampling rate. If this signal were obtained by proper sampling, the original analog signal must have had a frequency of 0.2. If aliasing took place during sampling, the digital frequency of 0.2 could have come from any one of an infinite number of frequencies in the analog signal: 0.2, 0.8, 1.2, 1.8, 2.2, … .

Just as aliasing can change the frequency during sampling, it can also change the phase. For example, look back at the aliased signal in Fig. 3-3d. The aliased digital signal is inverted from the original analog signal; one is a sine wave while the other is a negative sine wave. In other words, aliasing has changed the frequency and introduced a 180? phase shift. Only two-phase shifts are possible: 0? (no phase shift) and 180? (inversion). The zero-phase shift occurs for analog frequencies of 0 to 0.5, 1.0 to 1.5, 2.0 to 2.5, etc. An inverted phase occurs for analog frequencies of 0.5 to

1.0, 1.5 to 2.0, 3.5 to 4.0, and so on.

Now we will dive into a more detailed analysis of sampling and how aliasing occurs. Our overall goal is to understand what happens to the information when a signal is converted from a continuous to a discrete form. The problem is, these are very different things; one is a continuous waveform while the other is an array of numbers. This "apples-to-oranges" comparison makes the analysis very difficult. The solution is to introduce a theoretical concept called the impulse train.

Figure 3-5a shows an example analog signal. Figure (c) shows the signal sampled by using an impulse train. The impulse train is a continuous signal consisting of a series of narrow spikes (impulses) that match the original signal at the sampling instants. Each impulse is infinitesimally narrow, a concept that will be discussed in Chapter 13. Between these sampling times the value of the waveform is zero. Keep in mind that the impulse train is a theoretical concept, not a waveform that can exist in an electronic circuit. Since both the original analog signal and the impulse train are continuous waveforms, we can make an "apples-apples" comparison between the two.

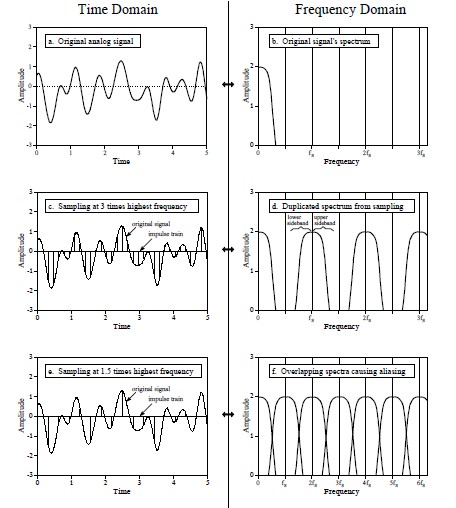


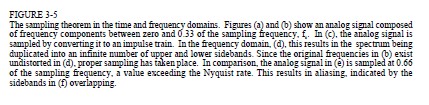
Now we need to examine the relationship between the impulse train and the discrete signal (an array of numbers). This one is easy; in terms of information content, they are identical. If one is known, it is trivial to calculate the other. Think of these as different ends of a bridge crossing between the analog and digital worlds. This means we have achieved our overall goal once we understand the consequences of changing the waveform in Fig. 3-5a into the waveform in Fig. 3.5c.

Three continuous waveforms are shown in the left-hand column in Fig. 3-5. The corresponding frequency spectra of these signals are displayed in the right-hand column.

Figure (a) shows an analog signal we wish to sample. As indicated by its frequency spectrum in (b), it is composed only of frequency components between 0 and about 0.33fs, where fs is the sampling frequency we intend to use. For example, this might be a speech signal that has been filtered to remove all frequencies above 3.3 kHz. Correspondingly, fs would be 10 kHz (10,000 samples/second), our intended sampling rate. Sampling the signal in (a) by using an impulse train produces the signal shown in (c), and its frequency spectrum shown in (d). This spectrum is a duplication of the spectrum of the original signal. Each multiple of the sampling frequency, fs, 2fs, 3fs, 4fs, etc., has received a copy and a left-for-right flipped copy of the original frequency spectrum. The copy is called the upper sideband, while the flipped copy is called the lower sideband. Sampling has generated new frequencies. Is this proper sampling? The answer is yes, because the signal in (c) can be transformed back into the signal in (a) by eliminating all frequencies above? fs. That is, an analog low-pass filter will convert the impulse train, (b), back into the original analog signal, (a). If you are already familiar with the basics of DSP, here is a more technical explanation of why this spectral duplication occurs. (Ignore this paragraph if you are new to DSP). In the time domain, sampling is achieved by multiplying the original signal by an impulse train of unity amplitude spikes. The frequency spectrum of this unity amplitude impulse train is also a unity amplitude impulse train, with the spikes occurring at multiples of the sampling frequency, fs, 2fs, 3fs, 4fs, etc. When two time domain signals are multiplied, their frequency spectra are convolved. This results in the original spectrum being duplicated to the location of each spike in the impulse train's spectrum. Viewing the original signal as composed of both positive and negative frequencies accounts for the upper and lower sidebands, respectively. This is the same as amplitude modulation.

Figure (e) shows an example of improper sampling, resulting from too low of sampling rate. The analog signal still contains frequencies up to 3.3 kHz, but the sampling rate has been lowered to 5 kHz. Notice that along the horizontal axis are spaced closer in (f) than in (d). The frequency spectrum, (f), shows the problem: the duplicated portions of the spectrum have invaded the band between zero and one-half of the sampling frequency. Although (f) shows these overlapping frequencies as retaining their separate identity, in actual practice they add together forming a single confused mess. Since there is no way to separate the overlapping frequencies, information is lost, and the original signal cannot be reconstructed. This overlap occurs when the analog signal contains frequencies greater than one-half the sampling rate, that is, we have proven the sampling theorem.





**Verification of sampling theorem:**

clc clear all close all t=-100:01:100; fm=0.02;

x=cos(2\*pi\*t\*fm); subplot(2,2,1);

plot(t,x); xlabel('time in sec'); ylabel('x(t)'); title('continuous time signal');

fs1=0.02; n=-2:2; x1=cos(2\*pi\*fm\*n/fs1); subplot(2,2,2); stem(n,x1); hold on subplot(2,2,2); plot(n,x1,':');

title('discrete time signal x(n) with fs<2fm');

xlabel('n'); ylabel('x(n)'); fs2=0.04; n1=-4:4; x2=cos(2\*pi\*fm\*n1/fs2); subplot(2,2,3); stem(n1,x2); hold on subplot(2,2,3); plot(n1,x2,':');

title('discrete time signal x(n) with fs>2fm');

xlabel('n');

ylabel('x(n)'); n2=-50:50; fs3=0.5; x3=cos(2\*pi\*fm\*n2/fs3); subplot(2,2,4); stem(n2,x3); hold on subplot(2,2,4); plot(n2,x3,':'); xlabel('n'); ylabel('x(n)');

title('discrete time signal x(n) with fs=2fm'); **LAB TASKS:**

**Demonstrate the effects of aliasing arising from improper sampling:**

1. Consider an analog signal x(t) consisting of three sinusoids of frequencies of 1 kHz, 4 kHz, and 6 kHz:

x(t)= cos(2πt)+ cos(8πt)+ cos(12πt)

where t is in milliseconds. Show that if this signal is sampled at a rate of fs = 5 kHz, it will be aliased with the following signal, in the sense that their sample values will be the same: xa(t)= 3 cos(2πt)

On the same graph, plot the two signals x(t) and xa(t) versus t in the range 0 ≤ t ≤ 2 msec.

To this plot, add the time samples x(tn) and verify that x(t) and xa(t) intersect precisely at these samples. These samples can be evaluated and plotted as follows:

fs = 5; T = 1/fs; tn = 0:T:2; xn = x(tn);

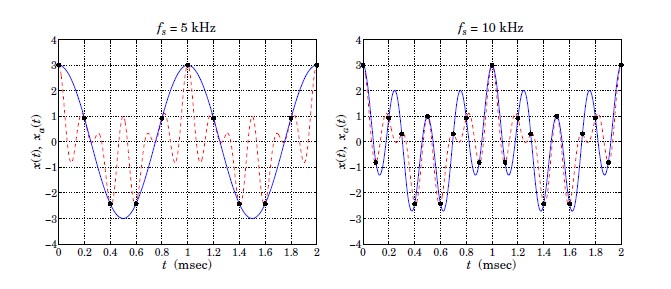
plot(tn, xn, ’.’);

2. Repeat Q1 with fs = 10 kHz. In this case, determine the signal xa(t) with which x(t) is aliased.

Plot both x(t) and xa(t) on the same graph over the same range 0 ≤ t ≤ 2 msec. Verify

again that the two signals intersect at the sampling instants.

(Hint: At least try to produce these output plots as shown below, with both signals intersecting at same intersection points as shown in figures/plots).



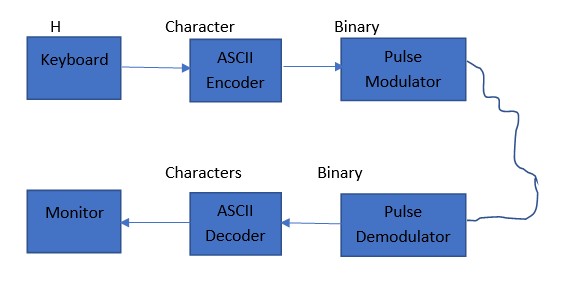
# Lab # 05: Simulation of a Simple Communication System in MATLAB

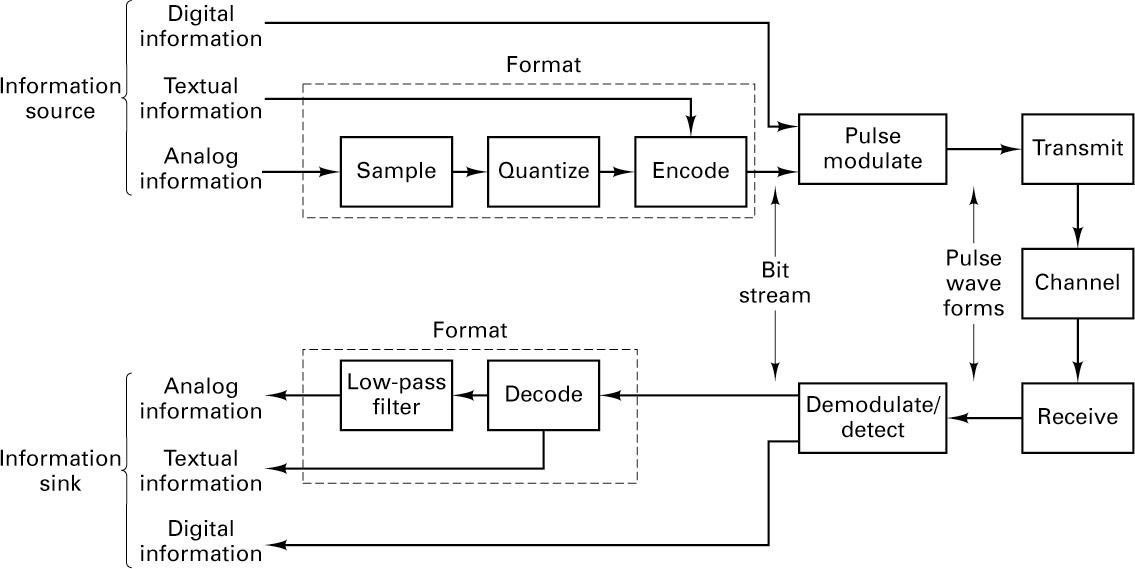
**Objective:**

observe digital communication by digitally transmitting and receiving a character in MATLAB.

**Explanation:**

Consider this simple communication system.

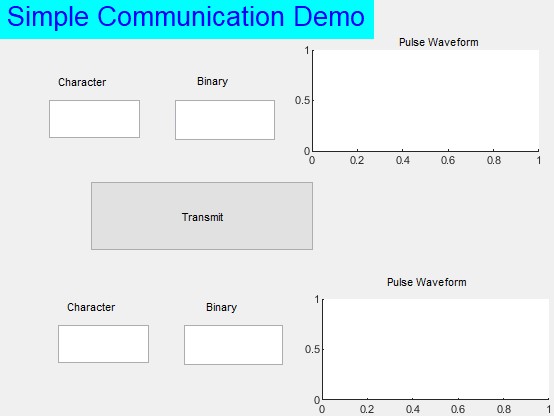




Block diagram representing formatting and transmission of baseband signals.

**LAB TASKS:**

Write a MATLAB Code in GUI to perform the following tasks.



**Task 1:**

Build a GUI in MATLAB as given above. Containing a string box for character, a string box for respective ASCII code, a string box for binary value and a graph showing the modulated pulse signal. You should write a character in the box and once you press the Transmit button the binary of the GUI, ASCII value appears in the respective box, and its respective binary value is displayed in the binary text box. Moreover, the pulse waveform graph plots the pulse modulated signal. You may use a pulse width of 0.1 seconds.

**Task 2:**

After that the signal is received at the pulse demodulator. In the pulse waveform box given below you should plot the received signal. After that the pulse is demodulated (converted back to binary) and displayed in the binary text box. Finally, the received character is displayed in the character box.

**Task 3:**

Next you should add some noise on the channel e.g., by flipping a signal pulse. After that the signal is received at the pulse demodulator. In the pulse waveform box given below you should plot the received signal with noise. After that the pulse is demodulated (converted back to binary) and displayed in the binary text box. Finally, the received character is displayed in the character box.

ASCII Table is given below:

**Letter ASCII Code Binary Letter ASCII Code Binary** a 097 01100001 A 065 01000001 b 098 01100010 B 066 01000010 c 099 01100011 C 067 01000011 d 100 01100100 D 068 01000100 e 101 01100101 E 069 01000101 f 102 01100110 F 070 01000110 g 103 01100111 G 071 01000111 h 104 01101000 H 072 01001000 i 105 01101001 I 073 01001001 j 106 01101010 J 074 01001010 k 107 01101011 K 075 01001011 l 108 01101100 L 076 01001100 m 109 01101101 M 077 01001101 n 110 01101110 N 078 01001110 o 111 01101111 O 079 01001111 p 112 01110000 P 080 01010000 q 113 01110001 Q 081 01010001 r 114 01110010 R 082 01010010 s 115 01110011 S 083 01010011 t 116 01110100 T 084 01010100 u 117 01110101 U 085 01010101 v 118 01110110 V 086 01010110 w 119 01110111 W 087 01010111 x 120 01111000 X 088 01011000 y 121 01111001 Y 089 01011001 z 122 01111010 Z 090 01011010

# Lab # 06: Calculation of Bit Error Rate in a Simple Communication System

**Objective:**

observe digital communication by digitally transmitting and receiving numbers in MATLAB, and then calculating the bit error rate (BER) with/without the influence of noise.

**Explanation:**

In digital transmission, the number of bit errors is the number of received bits of a data stream over a communication channel that have been altered due to noise, interference, distortion, or bit synchronization errors.

The bit error rate (BER) is the number of bit errors per unit time. The bit error ratio (also BER) is the number of bit errors divided by the total number of transferred bits during a studied time interval. Bit error ratio is a unitless performance measure, often expressed as a percentage.

As an example, assume this transmitted bit sequence:

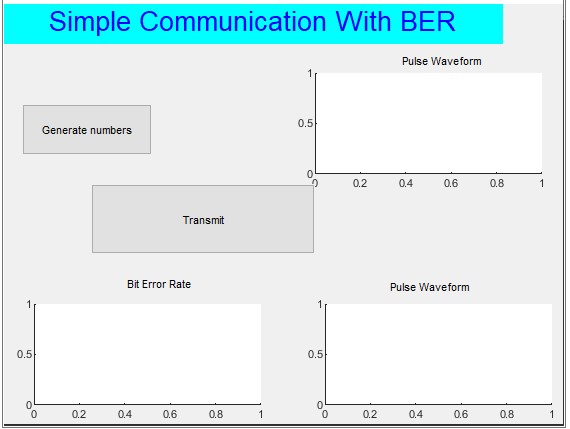
1 1 0 0 0 1 0 1 1 and the following received bit sequence:

0 1 0 1 0 1 0 0 1,

The number of bit errors (the underlined bits) is, in this case, 3. The BER is 3 incorrect bits divided by 9 transferred bits, resulting in a BER of 0.333 or 33.3%.

**LAB TASKS:**

You have transmitted and received a single character in the last lab using MATLAB with additive Gaussian noise. In this lab you would transmit numbers from 0000 to 9999 as characters in the format “zzzz” where z represents a single character for a number. At the end you will calculate the bit error rate for the received bit.



**Task 1:**

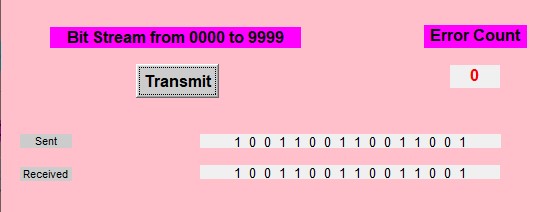
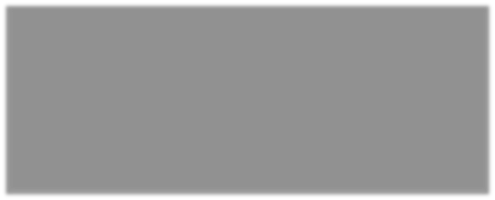
First do the following tasks without considering noise.

Transmit the binary stream for the number 0000. Here you would represent each 0 with its respective binary representation. Receive the transmitted bits and recover the 0000 being transmitted. Is here 0000 being correctly received if not find the number of

wrong bits? Save the number of erroneous bits in error\_count.

Now write a loop to from 0000 to 9999 and repeat the procedure from step 1 to 3.

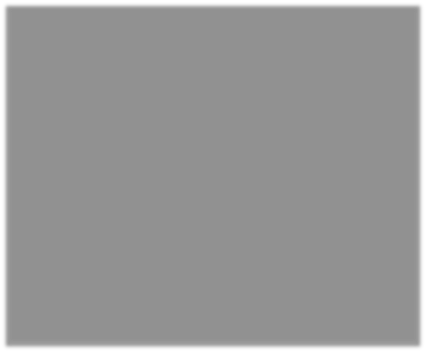
Update the error\_count in each iteration of code.Moreover, the pulse waveform graph plots the pulse modulated signal. You may use a pulse width of 0.1 seconds.



**Task 2:**

Now perform the tasks mentioned in step 1 to 4 for noise with variances 0, 0.1, 0.2,

0.3, 0.4, …, 3. Calculate the Bit error rate (BER) for each variance value by averaging the error\_count over all the numbers from 0000 to 9999. Finally plot the BER with variance on the x-axis and the BER on the y-axis as shown in the GUI below.



# Lab # 07: Generation of Bit Streams for Different Line Codes

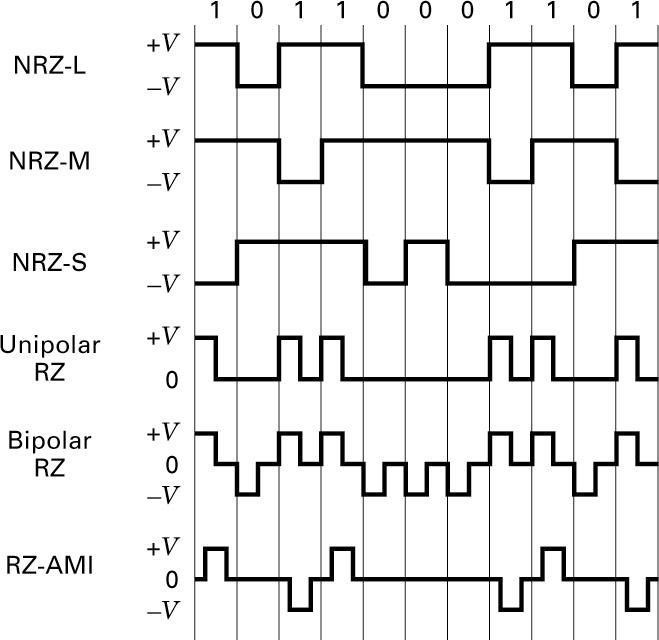
**Objective:**

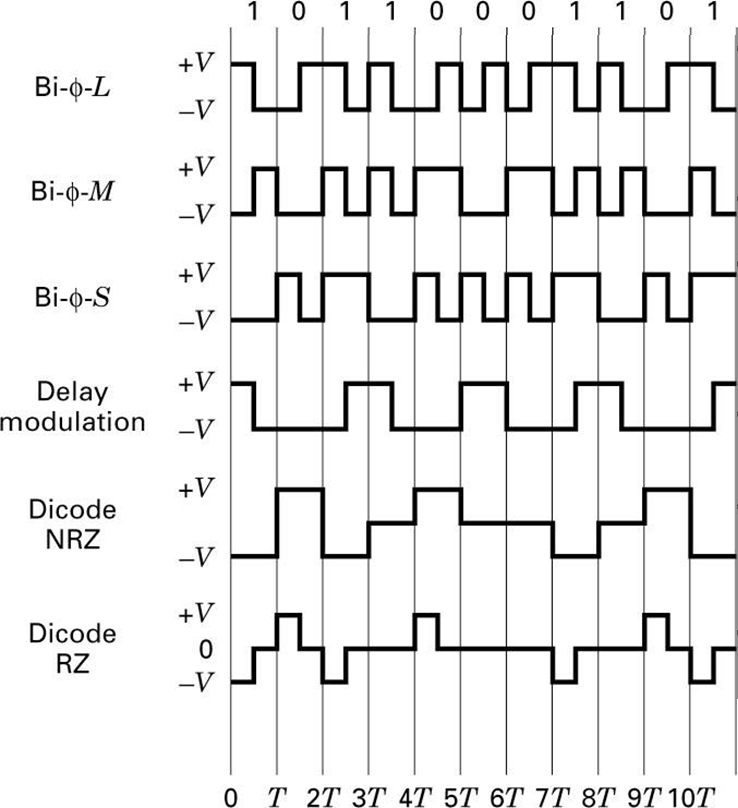
generate bit streams of different line codes in MATLAB and observe the pattern of resulting pulse formed.

**Explanation:**

In In telecommunication, a line code is a pattern of voltage, current, or photons used to represent digital data transmitted down a communication channel or written to a storage medium. This collection of signals is usually called a constrained code in data storage systems.

Some signals are more prone to error than others as the physics of the communication channel or storage medium constrains the repertoire of signals that can be used reliably. Common line encodings are unipolar, polar, bipolar, and Manchester code.





A line code will typically reflect technical requirements of the transmission medium, such as optical fiber or shielded twisted pair. These requirements are unique for each medium, because each one has different behavior related to interference, distortion, capacitance, and attenuation.

**LAB TASKS:**

Generate random (uniform distributed) binary numbers with length 100000. Do the pulse modulation by representing each bit (pulse) by 10 numerical values using the following line codes:

1. NRZ-L
2. RZ-L (Bipolar)
3. Bi-Phase-L
4. RZ-AMI

The maximum magnitude of the pulse should be 1 and minimum should be -1. Plot the waveform *time[sec] vs amplitude[volts]* to show the set pattern of respective pulses of above-mentioned encoding schemes.

# Lab # 08: Calculating PSD Estimates of Different Line Codes

**Objective:**

calculate PSD Estimates of different line codes in MATLAB and observe the pattern of resulting pulse formed.

**Explanation:**

We know that power spectral density provides the strength of different frequency components in a signal. You would like to design a baseband communications system with a bit rate of 15 bits/second. Moreover, we have a limited bandwidth channel with only 30 Hz spectrum available. You need to do an analysis that which signal suits better to your needs by estimating the PSD for NR-Z L, RZ-L, Bi-Phase L pulse (line) and AMI encoding schemes.

**LAB TASKS:**

Using the following line codes *NRZ-L, RZ-L (Bipolar), Bi-Phase-L, RZ-AMI,* from previous lab, perform the following tasks as mentioned below:

1. Estimate the PSD of these signals using psd or periodogram command of MATLAB.
2. Plot the PSD [vs Freq] (dB/Hz vs Hz) plot of these signals with different colors and legends.
3. Analyze the PSDs and find out which line code you can use for your communication system.
4. What would happen to your spectrum if the bit rate requirement were changed to 60 bits/second. How will this affect the spectrum? Will the given channel support this data rate? To show the set pattern of respective pulses of abovementioned encoding schemes.

# Lab # 09: Plotting the Constellation Diagram for 8-PSK using Natural Mapping

**Objective:**

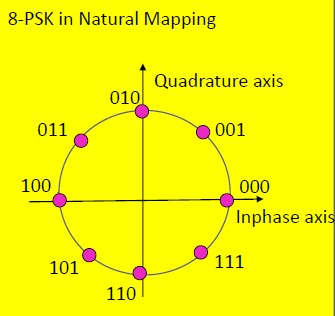
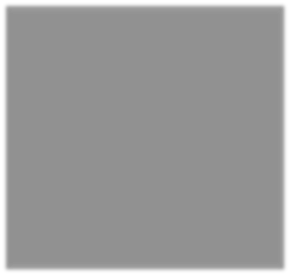
To draw the constellation diagram for 8-PSK using natural mapping.

**Explanation:**

Phase-shift keying (PSK) is a digital modulation process which conveys data by changing (modulating) the phase of a constant frequency reference signal (the carrier wave). The modulation is accomplished by varying the sine and cosine inputs at a precise time. It is widely used for wireless LANs, RFID and Bluetooth communication.

In M-ary PSK, **M** represents no. of symbols (or possible combinations for a given number of binary variables), while **m** represents the minimum no. of bits required per symbol such that ***m = log2 M*** or ***M = 2m***.

Also, the M points (say 8 points, in our case) are equidistant from each other along the circumference of the circle, i.e., with a constant phase (angle) between each other. The natural mapping is the sequential mapping as shown in the sample constellation diagram given below.



**LAB TASKS:**

For this lab, read the given instructions and perform the following tasks as mentioned below:

1. Generate an array A of 100000 random bits using rand function.
2. Convert this array to respective in phase and quadrature components for 8-PSK assuming an energy of Es=1 using natural mapping.
3. Now generate the respective in phase and quadrature pulses for the generated 8-PSK in phase and quadrature values in the previous step. Assume here that in MATLAB each pulse consists of 10 numerical values.
4. Plot the constellation diagram (for natural mapping) by using the in phase and quadrature components/pulses computed in the previous steps.
5. Draw the 8-PSK constellation diagram (natural mapping) via MATLAB’s built-in commands and then compare your results.

# Lab # 10: Plotting the Constellation Diagram for 8-PSK using Gray Mapping

**Objective:**

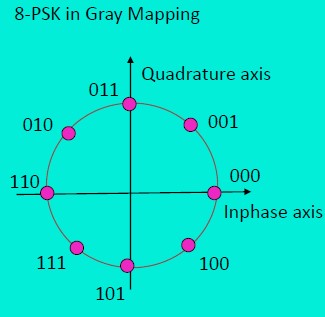
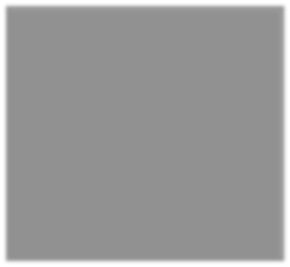
To draw the constellation diagram for 8-PSK using gray mapping.

**Explanation:**

Phase-shift keying (PSK) is a digital modulation process which conveys data by changing (modulating) the phase of a constant frequency reference signal (the carrier wave). The modulation is accomplished by varying the sine and cosine inputs at a precise time. It is widely used for wireless LANs, RFID and Bluetooth communication.

In M-ary PSK, **M** represents no. of symbols (or possible combinations for a given number of binary variables), while **m** represents the minimum no. of bits required per symbol such that ***m = log2 M*** or ***M = 2m***.

Also, the M points (say 8 points, in our case) are equidistant from each other along the circumference of the circle, i.e., with a constant phase (angle) between each other. Gray mapping is an encoding technique that ensures that the difference between two neighboring/adjacent symbols is just 1 bit i.e., a max error of 1 bit per sample. However, in natural mapping, max 3 bits (all) can occur as error. So, gray mapping reduces error. Gray mapping can be observed in the sample constellation diagram given below.



**LAB TASKS:**

For this lab, read the given instructions and perform the following tasks as mentioned below:

1. Generate an array A of 100000 random bits using rand function.
2. Convert this array to respective in phase and quadrature components for 8-PSK assuming an energy of Es=1 using natural mapping.
3. Now generate the respective in phase and quadrature pulses for the generated 8-PSK in phase and quadrature values in the previous step. Assume here that in MATLAB each pulse consists of 10 numerical values.
4. Plot the constellation diagram (for gray mapping) by using the in phase and quadrature components/pulses computed in the previous steps.
5. Draw the 8-PSK constellation diagram (gray mapping) via MATLAB’s built-in commands and then compare your results.

# Lab # 11: Plotting the Constellation Diagram for 16-PSK using Natural Mapping

**Objective:**

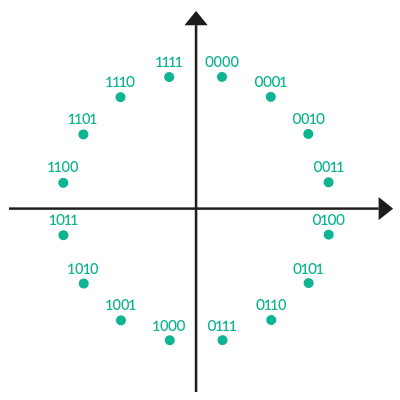
To draw the constellation diagram for 16-PSK using natural mapping.

**Explanation:**

Phase-shift keying (PSK) is a digital modulation process which conveys data by changing (modulating) the phase of a constant frequency reference signal (the carrier wave). The modulation is accomplished by varying the sine and cosine inputs at a precise time. It is widely used for wireless LANs, RFID and Bluetooth communication.

In M-ary PSK, **M** represents no. of symbols (or possible combinations for a given number of binary variables), while **m** represents the minimum no. of bits required per symbol such that ***m = log2 M*** or ***M = 2m***.

Also, the M points (say 16 points, in our case) are equidistant from each other along the circumference of the circle, i.e., with a constant phase (angle) between each other. The natural mapping is the sequential mapping as shown in the sample constellation diagram given below.



**LAB TASKS:**

For this lab, read the given instructions and perform the following tasks as mentioned below:

1. Generate an array A of 100000 random bits using rand function.
2. Convert this array to respective in phase and quadrature components for 16PSK assuming an energy of Es=1 using natural mapping.
3. Now generate the respective in phase and quadrature pulses for the generated 16-PSK in phase and quadrature values in the previous step. Assume here that in MATLAB each pulse consists of 10 numerical values.
4. Plot the constellation diagram (for natural mapping) by using the in phase and quadrature components/pulses computed in the previous steps.
5. Draw the 16-PSK constellation diagram (natural mapping) via MATLAB’s built-in commands and then compare your results.

# Lab # 12: Plotting the Constellation Diagram for 16-PSK using Gray Mapping

**Objective:**

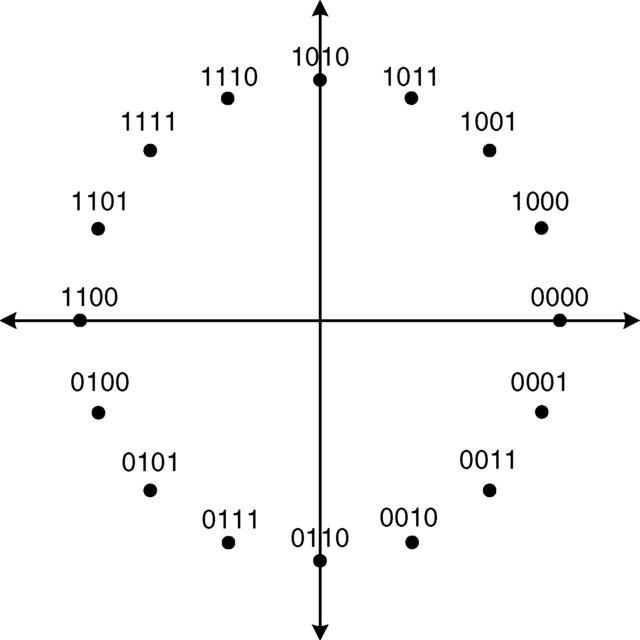
To draw the constellation diagram for 16-PSK using gray mapping.

**Explanation:**

Phase-shift keying (PSK) is a digital modulation process which conveys data by changing (modulating) the phase of a constant frequency reference signal (the carrier wave). The modulation is accomplished by varying the sine and cosine inputs at a precise time. It is widely used for wireless LANs, RFID and Bluetooth communication.

In M-ary PSK, **M** represents no. of symbols (or possible combinations for a given number of binary variables), while **m** represents the minimum no. of bits required per symbol such that ***m = log2 M*** or ***M = 2m***.

Also, the M points (say 16 points, in our case) are equidistant from each other along the circumference of the circle, i.e., with a constant phase (angle) between each other. Gray mapping is an encoding technique that ensures that the difference between two neighboring/adjacent symbols is just 1 bit i.e., a max error of 1 bit per sample. However, in natural mapping, max 3 bits (all) can occur as error. So, gray mapping reduces error. Gray mapping can be observed in the sample constellation diagram given below.



**LAB TASKS:**

For this lab, read the given instructions and perform the following tasks as mentioned below:

1. Generate an array A of 100000 random bits using rand function.
2. Convert this array to respective in phase and quadrature components for 16PSK assuming an energy of Es=1 using natural mapping.
3. Now generate the respective in phase and quadrature pulses for the generated 16-PSK in phase and quadrature values in the previous step. Assume here that in MATLAB each pulse consists of 10 numerical values.
4. Plot the constellation diagram (for gray mapping) by using the in phase and quadrature components/pulses computed in the previous steps.
5. Draw the 16-PSK constellation diagram (gray mapping) via MATLAB’s builtin commands and then compare your results.

# Lab # 13: Simulation of an Image Transmission through 8-ASK

**Objective:**

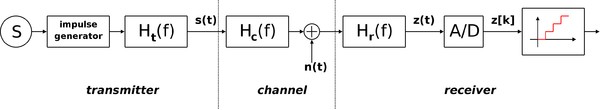
To transmit an image using 8-ASK (via MATLAB simulation).

**Explanation:**

**Amplitude-shift keying (ASK)** is a form of amplitude modulation that represents digital data as variations in the amplitude of a carrier wave. In an ASK system, a symbol, representing one or more bits, is sent by transmitting a fixed-amplitude carrier wave at a fixed frequency for a specific time duration. For example, if each symbol represents a single bit, then the carrier signal could be transmitted at nominal amplitude when the input value is 1 but transmitted at reduced amplitude or not at all when the input value is 0.

Any digital modulation scheme uses a finite number of distinct signals to represent digital data. ASK uses a finite number of amplitudes, each assigned a unique pattern of binary digits. Usually, each amplitude encodes an equal number of bits. Each pattern of bits forms the symbol that is represented by a particular amplitude. The demodulator, which is designed specifically for the symbol-set used by the modulator, determines the amplitude of the received signal and maps it back to the symbol it represents, thus recovering the original data. Frequency and phase of the carrier are kept constant.

ASK system can be divided into three blocks. The first one represents the transmitter, the second one is a linear model of the effects of the channel, the third one shows the structure of the receiver. The following notation is used:

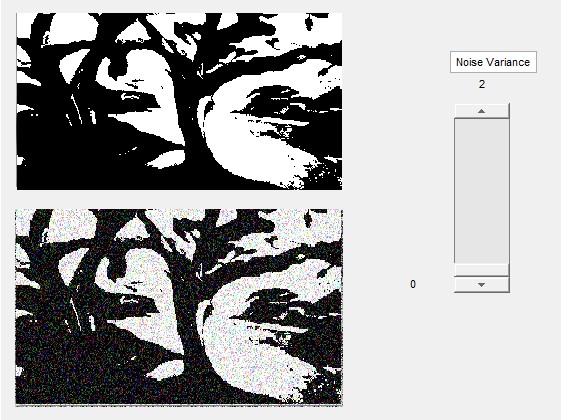


**LAB TASKS:**

For this lab, read the given instructions and perform the following tasks (as mentioned below) in MATLAB by generating a GUI as given at the end of the tasks.:

In this lab we will use 8-ASK to transmit a picture. The picture is uploaded with in LMS with the name river.jpeg. Moreover, please consider a single pulse duration Ts=2ms.

1. Load this image in Matlab by using the imread command, the image will be loaded as RGB.
2. Use im2bw(I,level) command to convert this image to a single channel binary image of size 350x240. Plot this image using imshow command.
3. Convert the binary image matrix to a single new array of size 84000 binary values.
4. Now group the binary stream to bits of group 3 and map these bits to the respective baseband pulses for 8 ASK. Here each pulse consists of 10 numerical values. Please calculate here the values for the amplitudes for pulses i.e V1, V2, V3,…. V8. Here we assume that the average symbol energy is Es=1.
5. Bandpass the signal using Matlab Modulate function with meth am.
6. Bandpass demodulate the signal using Matlab demod function.
7. Add noise with variance var=0 to the baseband signal received.
8. At the receiver side again, pulse demodulate and detect the transmitted bits. From the bits calculate the binary values, and then reshape them to 350x240 matrix and from these values plot the image using imshow command.
9. Keep changing the variance in the 0 to 2 range, observe how the image changes.



# Lab # 14: Simulation of an Image Transmission through 4-PSK

**Objective:**

To transmit an image using 4-PSK (via MATLAB simulation).

**Explanation:**

**Phase-shift keying (PSK)** is a digital modulation process which conveys data by changing (modulating) the phase of a constant frequency reference signal (the carrier wave). The modulation is accomplished by varying the sine and cosine inputs at a precise time. It is widely used for wireless LANs, RFID and Bluetooth communication.

In M-ary PSK, **M** represents no. of symbols (or possible combinations for a given number of binary variables), while **m** represents the minimum no. of bits required per symbol such that ***m = log2 M*** or ***M = 2m***.

Since, we have discussed different bandpass modulation schemes such as ASK And PSK. In this lab we will use 4-PSK to transmit a picture. The picture is uploaded with in LMS with the name monalisa.jpeg. Perform the tasks in Matlab by generating a GUI as given at the end of the tasks. Consider pulse duration Ts=2ms.

**LAB TASKS:**

For this lab, read the given instructions and perform the following tasks as mentioned below in MATLAB by generating a GUI as given at the end of the tasks. The picture is uploaded with in LMS with the name monalisa.jpeg. Consider pulse duration Ts=2ms.

1. Load this image in Matlab by using the imread command, the image will be loaded as RGB.
2. Use [im2bw(I,level)](https://www.mathworks.com/help/images/ref/im2bw.html#d122e87895) command to convert this image to a single channel binary image of size 350x240. Plot this image using imshow command.
3. Convert the binary image matrix to a single new array of size 84000 binary values.
4. Now group the binary stream to bits of group 2 and map these bits to the respective baseband inphase and quadrature pulses for 4 PSK (vIN and vQu). Here each pulse consists of 10 numerical values for each pulse. Please calculate here the values for the amplitudes for pulses i.e (Vx1, Vq1), (Vx2, Vq2), (Vx3, Vq3) and (Vx4, Vq4). Here we assume that the average symbol energy is Es=1.
5. Bandpass the signal using Matlab y= [modulate(vIN,fc,fs,method,vQu)](https://www.mathworks.com/help/signal/ref/modulate.html#d122e99962) function with method qam. Here vIN is the inphase while vQu is the quadrature baseband signals.
6. Bandpass demodulate the signal using Matlab [rIN,rQu] = demod(y,fc,fs,'qam') function. Here rIN and rQu are the received inphase and quadrature pulses respectively.
7. Add noise with variance var=0 to the baseband signal received.
8. At the receiver side again pulse demodulate the inphase and quadrature pulses and detect the transmitted bits. The detection method will be explained on the board. From the bits calculate the binary values, and then reshape them to 350x240 matrix and from these values plot the image using imshow command.
9. Keep changing the variance in the 0 to 2 range, observe how the image changes.

