Electronics and Electrical Communications Engineering Department

Faculty of Engineering

Cairo University

Implementation and Comparative Analysis of Huffman and Fano Source Coding Algorithms

ELC4020 “Advanced Communication Systems “

4th Year

1st Semester - Academic Year 2025/2026

Prepared by:

NAME	SECTION	ID
Mohamed Ahmed Abd El Hakam	3	9220647
Yousef Khaled Omar Mahmoud	4	9220984

Submission Date: 2 November 2025

Instructor: Eng. Mohamed Khaled
Dr. Mohammed Nafie & Dr. Mohamed Khairy

1. Contents

2. Table of Figures

3. Role of Each Member

ROLE	NAME
code the Huffman source coding	Youssef Khaled
compute the realization calculation	Ahmed Mohamed
compute the time calculation	Shahd Hamed
Report and Hand Analysis	Mohamed Ahmed
Report and Hand Analysis	Omar Ahmed

4. Project Description

Using software radio technique (SDR) to transmit stream of randomness bits through an ideal channel (which performing a small delay) using Matlab. Performing measures and analysis to see the performance of the system through three main line codes (unipolar, polar nrz and polar rz).

هنا ممكن تشرحله انه تشرحله انه
source code implementqation using matlab

5. Introduction

Software radio is a revolutionary approach that brings the programming code directly to the antenna, minimizing reliance on traditional radio hardware as shown in figure 1.

By doing so, it transforms challenges associated with radio hardware into software- related issues. Unlike conventional radios, where signal processing primarily relies on analog circuitry or a combination of analog and digital chips, software radio operates by having software dictate both the transmitted and received waveforms.

هنا ممكن تشرحله انه تشرحله انه
what’s source coding and why it’s important

This paradigm shift allows for greater flexibility and adaptability in radio systems, as they can be easily reconfigured and optimized through software updates, rather than hardware modifications.

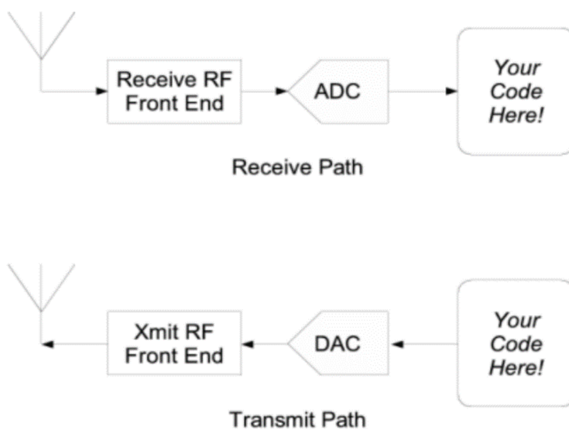

Figure 1 Rx and Tx path

6. Control Flags

مفيش المرة ده ممكن تمسحها لو عاوز

Flag	Value	Description
A	4	Amplitude of line code
N_realizations	500	Number of waveforms (ensemble size)
num_bits	101	Bits per waveform and one extra bit for shifting
bit_duration	70e-3	Duration of each bit
dac_interval	10e-3	DAC update interval

7. Input Data Symbols

```
% Given Symbols probabilities
symbols = {'A','B','C','D','E','F','G'};
P = [0.35 0.30 0.20 0.10 0.04 0.005 0.005];
```

As seen this is the input given

كمل هنا عشان معيش أفكار

8. Huffman Source Coding

حط هنا شوية معلومات عنه زي مثلا انه ممكن يوصل ل
100% لو كان عددهم ما لا نهاية

8.1. Algorithm Overview[2]

Step 1: Sort all the N_s symbols based on their probability in descending order(من الكبير للصغير).

Step 2: Crate a new colume has $(N_s - 1)$ by Sum the two symbols with the smallest probabilities.

Step 3: Repeat step 1 and 2 utill only two symbols remain.

Step 4: Assing codes for: the first as 0 and the second as 1.

Step 5: go step back to Left coalum and copy all non changed proabiltes codes as right colum (exept the lowst two in the left they alrady add in step 3), the chagned ones thier codes will be the same code of their summtion but zero is added in the least significant bit for the first and one for the second .

اشرحه عادي زي ما هو شرح في السيكشن

8.2. Hand Analysis

Symbol	P0	C0	P1	C1	P2	C2	P3	C3	P4	C4	P5	C5
A	0.35	00	0.35	00	0.35	00	0.35	00	0.35	1	0.65	0
B	0.3	01	0.3	01	0.3	01	0.3	01	0.35	00	0.35	1
C	0.2	10	0.2	10	0.2	10	0.2	10	0.3	01		
D	0.1	110	0.1	110	0.1	110	0.15	11				
E	0.04	1110	0.04	1110	0.05	111						
F	0.005	11110	0.01	1111								
G	0.005	11110										

So here's the output that we are aiming for.
With the Kraft's Sum,entropy, average length and efficiency:

$$H\big(P(x)\big) = -\sum_{xi} P(xi)\log_2 P(xi)$$

$$L(C) = -\sum_{xi} P(xi)L(xi)$$

$$\eta = \frac{H(P(x))}{L(C)} * 100\%$$

$$Kraft's\ Sum = \sum_{Li} 2^{-Li}$$

$$Kraft's\ Inequality:\ Kraft's\ Sum <= 1$$

So by calculating them we can find that:

$$H(P(x)) = -(0.35\log_2 0.35 + 0.3\log_2 0.3 + 0.2\log_2 0.2 + 0.1\log_2 0.1 + 0.04\log_2 0.04 + 0.005\log_2 0.005 + 0.005\log_2 0.005) = 2.11\ bits/symbol$$

And

$$L(C) = 0.35 * 2 + 0.3 * 2 + 0.2 * 2 + 0.1 * 3 + 0.04 * 4 + 0.005 * 5 + 0.005 * 5 = 2.21\ bits/symbol$$

So the overall efficiency is $\eta = \frac{L(C)}{H(P(x))} * 100\% = \frac{2.11}{2.21} * 100\% = 95.475\%$

احسب هنا ال
kraft
اقله,
stratify
ارسمله الشجرة,

8.3. MatLab Implementation[3]

8.3.1. Calculations Functions

First we got the calculations Functions:

```
%% ---------------------------------------------------------------------------
%              Entropy Calculation
% ---------------------------------------------------------------------------

function H = entropy_calc(P)
%ENTROPY_CALC Compute the source entropy H(P(x))
%    H = entropy_calc(P)
%    P : vector of symbol probabilities
%    H : entropy in bits

    % Validate input
    if any(P < 0) || abs(sum(P) - 1) > 1e-6
        warning('Probabilities should sum to 1. Normalizing...');
        P = P / sum(P);
    end

    % Remove zeros (to avoid log2(0))
    P = P(P > 0);

    % Compute entropy
    H = -sum(P .* log2(P));
end
```

And

```
%% ---------------------------------------------------------------------------
%              Average Length Calculation
% ---------------------------------------------------------------------------
function L = average_length_calc(dict)
%AVERAGE_LENGTH_CALC Compute average codeword length L(C)
%    L = average_length_calc(dict, P)
%    dict : Huffman dictionary cell array {symbol, code}
%    P : vector of symbol probabilities (same order as dict)
%
%    If P is empty, it tries to extract from dict(:,2) if present
%    L : average code length


    % --- Compute code lengths ---
    code_lengths = cellfun(@length, codes);

    % --- Normalize probabilities ---
    P = P(:) / sum(P);

    % --- Check dimensions ---
    if length(P) ~= length(code_lengths)
        error('Number of probabilities does not match number of codewords.');
    end

    % --- Compute average code length ---
    L = sum(P .* code_lengths);
end
```

And

```
%% -----------------------------------------------------------
%                Efficiency Calculation
% -----------------------------------------------------------

function eta = efficiency_calc(H, L)
%EFFICIENCY_CALC Compute Huffman coding efficiency η
%   eta = efficiency_calc(H, L)
%   H : entropy
%   L : average code length
%   eta : efficiency in percentage (%)

    if L <= 0
        error('Average length L must be positive.');
    end

    eta = (H / L) * 100;
end
```

And

```
%% ------------------------------------------------------------------------
%                Print Kraft Inequality Function
% ------------------------------------------------------------------------
function [kraft_sum, kraft_flag]=kraft_analysis(dict)
% KRAFT_ANALYSIS  Compute Kraft's inequality and visualize Kraft tree
%   kraft_analysis(dict)
%
%   Input:
%       dict : cell array {N x 3}  → {symbol, P, code}
%
%   Example:
%       dict = {'A','0'; 'B','10'; 'C','110'; 'D','111'};
%       kraft_analysis(dict);



    % --- 1. Compute Kraft's inequality ---
    code_lengths = cellfun(@length, codes);
    kraft_sum = sum(2.^(-code_lengths));

    fprintf('\n=== Kraft Inequality Check ===\n');
    fprintf('Sum(2^{-l_i}) = %.4f\n', kraft_sum);

end
```

8.3.2. Getting input data

```
% Combine into dictionary-like cell array
dict_input = [symbols(:), num2cell(P(:))];

% Assume not great until great
err_flag = 1;

% Validate using the check_symbols() function
[ok, msg] = check_symbols(dict_input);

% Display validation result
if ok
    disp('✅ Dictionary is valid!');
    err_flag =0;
else
    disp(['❌ Error: ' msg]);
    err_flag = 1;
end

% ----------------------------------------------------------------------
% Compute entropy (only if valid)
% ----------------------------------------------------------------------
H = entropy_calc(P)
```

So the output is:

--- Input Symbol Dictionary ---

	Symbol	Probability
1	A	0.3500
2	B	0.3000
3	C	0.2000
4	D	0.1000
5	E	0.0400
6	F	0.0050
7	G	0.0050

Entropy: H = 2.1100 bits/symbol

8.3.3. Huffman Function

First I merged the last 2 probabilities to have in last just 2 probabilities

```
% --- Iteratively merge ---
for step = 2:numCols
    % Sort ascending to pick smallest two
    curP = sort(curP, 'ascend');
    if numel(curP) >= 2
        p1 = curP(1);
        p2 = curP(2);
        mergedP = p1 + p2;
        % Remove two smallest and add merged one
        curP = [mergedP; curP(3:end)];
    end
    % Sort descending for display
    curP = sort(curP, 'descend');

    % Fill current column
    for r = 1:maxRows
        if r <= numel(curP)
            history{r,step} = curP(r);
        else
            history{r,step} = NaN;
        end
    end
end
```

Then I assign the codes from last column to first

```
% === Assign child codes ===
% Last two rows in previous P column are merged into this parent
history_table_full{raw_counter, prevCcol}   = [parentCode '0'];
history_table_full{raw_counter+1, prevCcol} = [parentCode '1'];

% For each previous non-merged row (in display order top->bottom)
for ii = 1:(raw_counter-1)
    % Skip the rows that were just merged (raw_counter and raw_counter+1)

    % Get the probability value in the previous column for this row
    valPrev = history_table_full{ii, prevPcol};
    if isnan(valPrev)
        continue; % nothing to copy
    end

    % Find matching value in the current column (exclude merged parent)
    currMatches = find(abs(currPvals - valPrev) < 1e-12);

    % Remove the matchIdx (the merged parent) if it appears
    currMatches(currMatches == matchIdx) = [];

    if isempty(currMatches)
        continue; % no corresponding match found
    end

    % If there are duplicates (two identical probabilities)
    if numel(currMatches) > 1
        % take both, and copy their codes to the two rows
        history_table_full{ii,   prevCcol} = currCvals{currMatches(1)};
        if (ii+1) <= numRows
            history_table_full{ii+1, prevCcol} = currCvals{currMatches(2)};
        end
    else
        % single match — copy code directly
        history_table_full{ii, prevCcol} = currCvals{currMatches(1)};
    end
```

So the output is:

Huffman Encoding: Probability and Code Evolution (P/C Steps)

P0	C0	P1	C1	P2	C2	P3	C3	P4	C4	P5	C5
0.3500	00	0.3500	00	0.3500	00	0.3500	00	0.3500	1	0.6500	0
0.3000	01	0.3000	01	0.3000	01	0.3000	01	0.3500	00	0.3500	1
0.2000	10	0.2000	10	0.2000	10	0.2000	10	0.3000	01		
0.1000	110	0.1000	110	0.1000	110	0.1500	11				
0.0400	1110	0.0400	1110	0.0500	111						
0.0050	11110	0.0100	1111								
0.0050	11111										

8.3.4. Theoretical Vs Practical

The results are:

--- Huffman Coded Dictionary ---

	Symbol	Probability	Code
1	A	0.3500	00
2	B	0.3000	01
3	C	0.2000	10
4	D	0.1000	110
5	E	0.0400	1110
6	F	0.0050	11110
7	G	0.0050	11111

H = 2.1100 | L = 2.2100 | η = 95.47 % | Kraft = 1.0000

As theoretical

9. Generation of Data

```
Data = randi([0, 1], 1, num_bits, 'int8');   % Random bit sequence
```

Using the function: “**Randi**” to generate random binary data of size 500x101[3]
(500 waveforms each with 101 bits). This data represents the binary bits that need to be encoded.

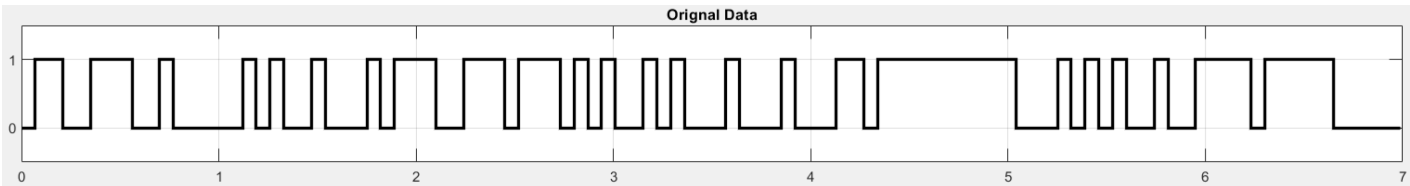

Figure 2 ADC Binary Output

For the line codes we will use this function:

```
function [Unipolar, PolarNRZ, PolarRZ] = generate_linecodes(Data, A, samples_per_bit)...
```

10. polar NRZ ensemble creation

```
% Polar NRZ: 0 → -A, 1 → +A
PolarNRZ = int8((2 * Data - 1) * A);
PolarNRZ = repelem(PolarNRZ, samples_per_bit);
```

- The data consists of 0s and 1s. We converted these values to A and -A respectively.
- Then, we utilized the “**repelem”** function to repeat each element seven times (samples_num).

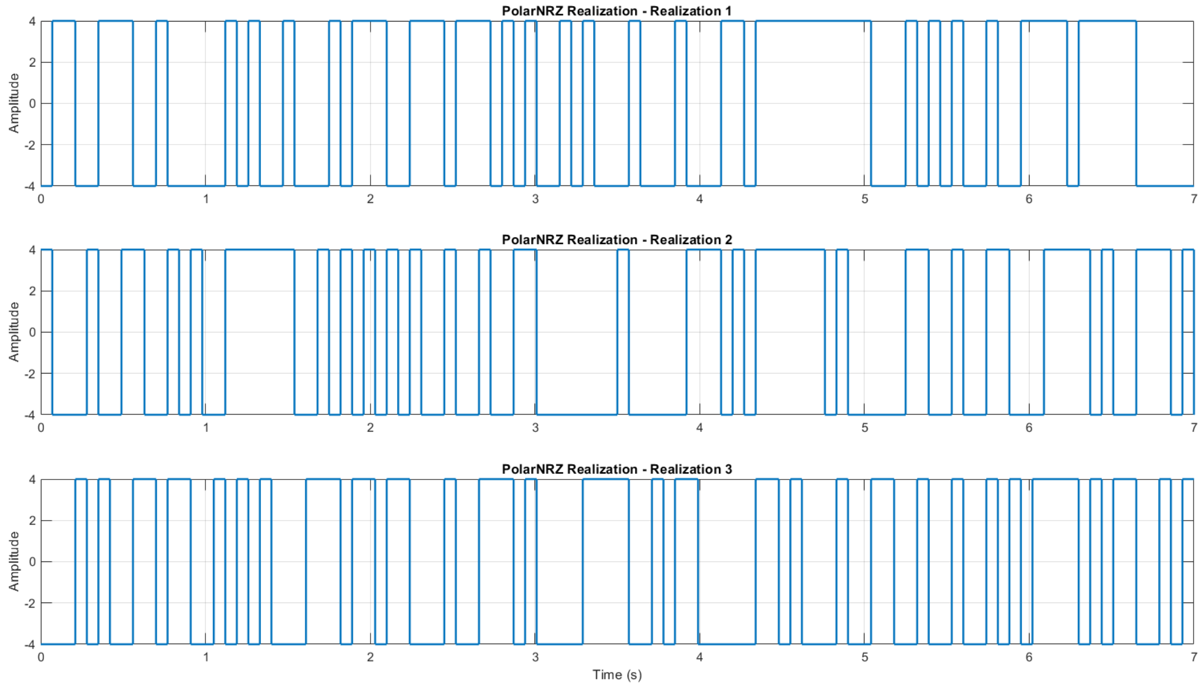

Figure 3 PolarNRZ Realizations

11. Uni polar NRZ ensemble creation

```
% Unipolar NRZ: 0 → 0V, 1 → A
Unipolar = int8(Data * A);
Unipolar = repelem(Unipolar, samples_per_bit); % Repeat each bit for duration
```

- We then generate unipolar NRZ amplitudes along with its realization.
- We convert data (1,0) to 1 →A ,0→ 0 to have uni_polar_NRZ.

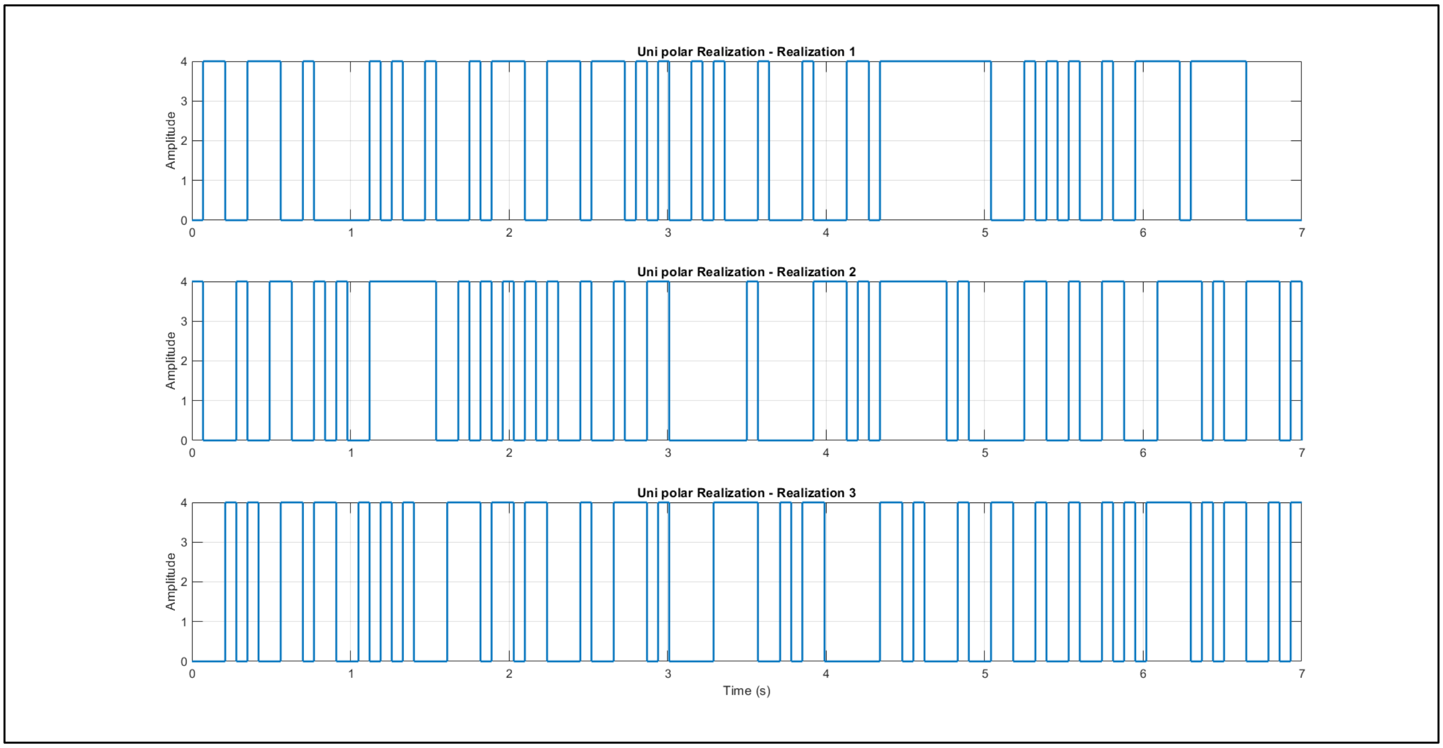

Figure 4 Uni Polar Realizations

12. polarRZ ensemble creation

```
% Polar Return-to-Zero (RZ): Same as Polar NRZ but second half set to 0
PolarRZ = PolarNRZ;

% Apply RZ rule: second half of each bit period should be zero
i = length(Data);  % Start from the last bit
while i > 0
    end_idx = i * samples_per_bitd;  % Last sample of the bit
    start_idx = end_idx - floor(samples_per_bitd / 2) + 1;  % Start of the second half
    PolarRZ(start_idx:end_idx) = 0;  % Set the second half of the bit period to zero
    i = i - 1;  % Move to the previous bit
end
```

- The data consists of 0s and 1s. We first convert these values to amplitudes: `0` → **`-A`**, `1` → **`+A`** (this is the standard **Polar NRZ** encoding).
- Then, we utilized the `repelem` function to repeat each amplitude value `samples_per_bit` times. This creates a constant level for each bit across its time duration.
- To convert **Polar NRZ** to **Polar Return-to-Zero (RZ)**, we start with the Polar NRZ waveform.
- We apply the RZ rule by modifying the **second half of each bit period**:
  For every bit, we calculate the index range that corresponds to the second half of its duration and set those values to zero.

- This creates a waveform where the signal returns to zero in the second half of each bit period, while the first half retains the Polar NRZ value (`+A` or `-A`).
- The result is a **Polar RZ** line code that has a non-zero level only during the first half of each bit, making it more suitable for synchronization at the receiver.

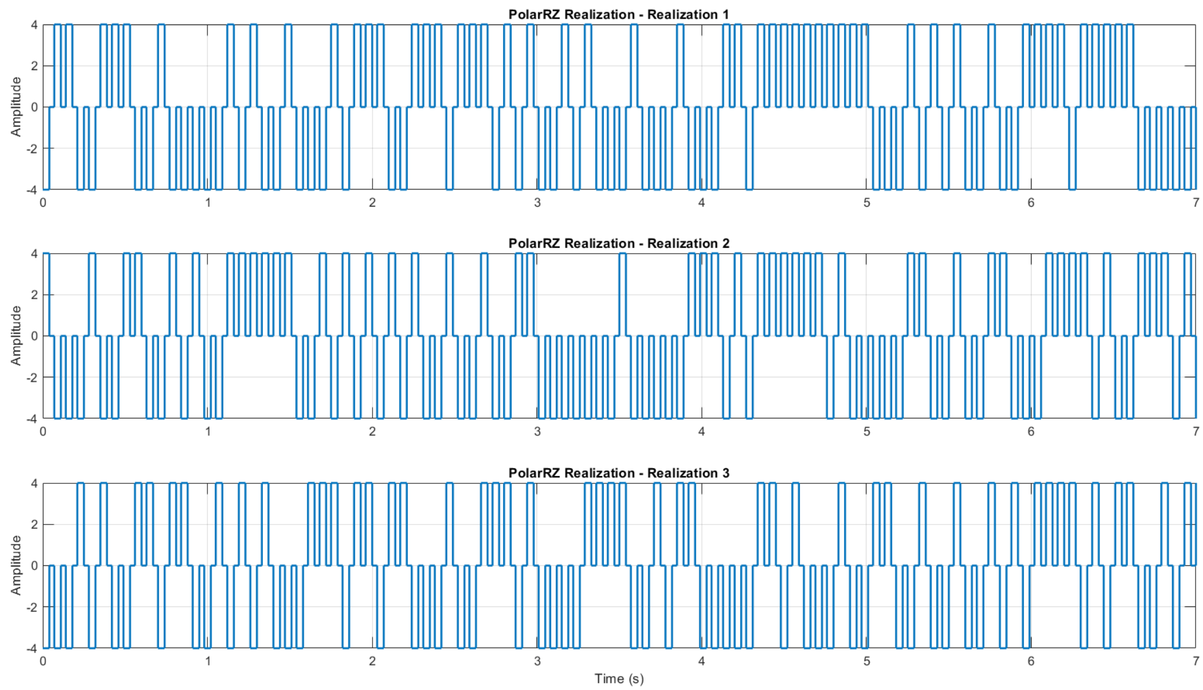

Figure 5 PolarRZ Realization

13. Random initial time shift

For the random shift we made this function:

```
function [Unipolar_Shifted, PolarNRZ_Shifted, PolarRZ_Shifted] =...
    apply_random_shift_fixed_size(Unipolar_All, PolarNRZ_All, PolarRZ_All, samples_per_bit)...
```

```
% Apply random shift to each realization
for i = 1:N_realizations
    % Generate random shift in range [0, samples_per_bit-1] samples
    random_shift_bits = randi([0, samples_per_bit-1]);

    % Extract shifted region
    Unipolar_Shifted(i, :) = Unipolar_All(i, random_shift_bits+1 : random_shift_bits+total_samples);
    PolarNRZ_Shifted(i, :) = PolarNRZ_All(i, random_shift_bits+1 : random_shift_bits+total_samples);
    PolarRZ_Shifted(i, :) = PolarRZ_All(i, random_shift_bits+1 : random_shift_bits+total_samples);
end
```

- Generating a single random initial time delay that can range from ‘0’ to ‘6’ samples for each waveform using the function “randi”.

- Then, we utilized the randi function to generate a random number ranging from 0 to 6, which represents the delay or start time, then we take the elements from this random index (start_indices) to 700+(start_indices).

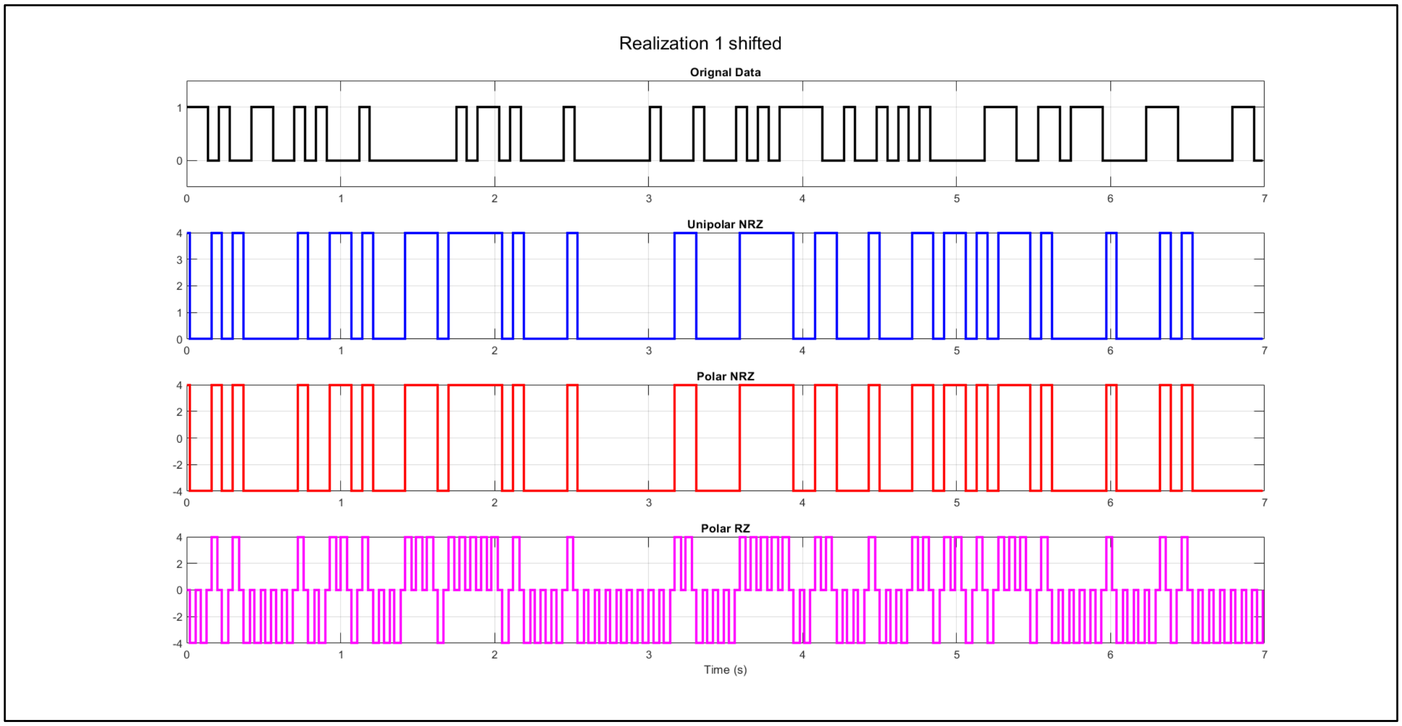

Figure 6 Realization Shifted

14. Getting cell arrays ready to calculate the statistical mean and autocorrelation:

For the mean the cells are ready, as for the autocorrelation we’re going to use this function:

```
function [R_unipolar_nrz_t, R_polar_nrz_t, R_polar_rz_t, taw2] = ...
    compute_time_autocorr(UnipolarNRZ, PolarNRZ, PolarRZ)...
```

In which we’re making the array ready by shifting it with tau.

```
% Get number of realizations and samples
[num_realizations, num_samples] = size(UnipolarNRZ);

% Define range of time lags
max_lag = num_samples - 1; % Maximum lag value
taw2 = -max_lag:max_lag; % Lag vector

% Initialize autocorrelation matrices (each row for a realization)
R_unipolar_nrz_t = zeros(num_realizations, length(taw2));
R_polar_nrz_t = zeros(num_realizations, length(taw2));
R_polar_rz_t = zeros(num_realizations, length(taw2));
```

So the array will have the length of max tau which is 700.

15. Questions

15.1. Statistical Mean

15.1.1. Hand Analysis

For the "Statistical Mean" which represents the average of all the realizations at the same time instant, let us consider the first line code method "Unipolar NRZ"

$\mu X(t) = 0 * 0.5 + 4 * 0.5 = 2\ (Constant\ across\ time)$

And in the same matter, we can calculate the "Statistical Mean" for both "Polar NRZ" and "Polar RZ" as following:

$\mu X\_PNRZ(t) = 4 * 0.5 + (-4) * 0.5 = 0\ (Constant\ across\ time)$.

$\mu X\_PRZ(t) = 4 * 0.5 + (-4) * 0.5 = 0\ (Constant\ across\ time)$.

15.1.2. Code Snippet

```
function mean_waveform = calculate_mean(waveform_matrix)
    % Calculates the mean across all realizations without using the mean function
    % waveform_matrix: Matrix where each row is a realization

    [num_realizations, num_samples] = size(waveform_matrix); % Get matrix dimensions
    mean_waveform = sum(waveform_matrix, 1) / num_realizations; % Sum and divide by count

end
```

- The mean is calculated as μ=ΣX/N (the sum divided by the number of the elements).

15.1.3. Plotting the Statistical Mean:

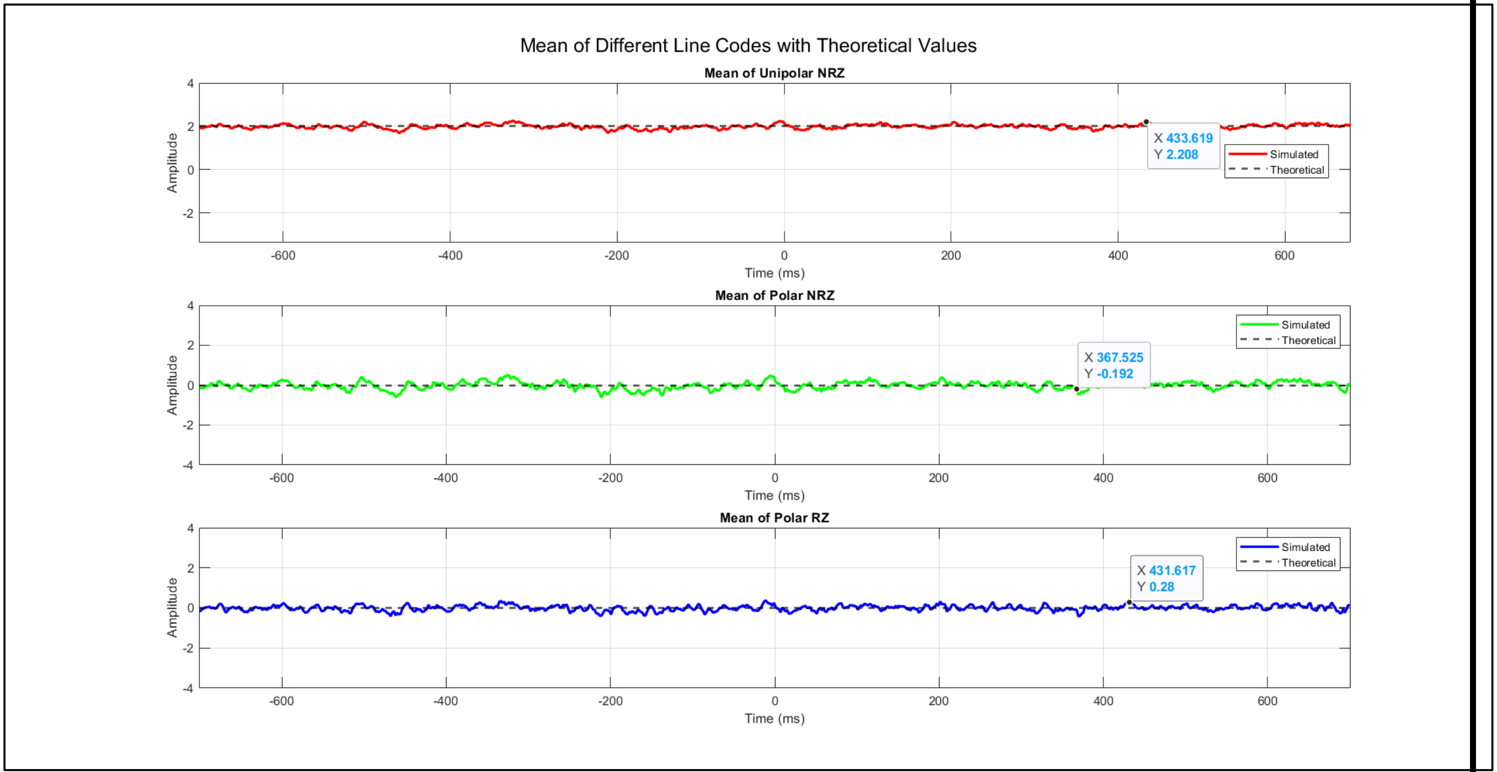

Figure 7 Plot of Statistical Mean

- As expected, polar RZ & NRZ have almost zero mean and the uni polar has mean around 2 Bec its amplitude ranges from 0:4.

15.2. Statistical Autocorrelation

15.2.1. Hand Analysis

$$R_X(\tau) = E[X(\tau)\,X(t+\tau)] = \sum X(\tau)\,X(t+\tau)\,P(X(\tau)\,X(t+\tau))$$

- **For Unipolar NRZ:**
  We have 2 cases **(Considering T to be 70ms or 7 samples),**
  1. $|\tau| < T$

$$R_X(\tau) = E[X(\tau)\,X(t+\tau)]$$
$$= 4^2 * P(4,4) + 0^2 * P(0,0) + 4*0*P(0,4) + 0*4*P(4,0)$$
$$= 4^2 * P(4,4)$$
$$P(4,4) = P(X(t+\tau) = 4 \mid X(t) = 4) * P(X(t) = 4)$$
$$P(X(t+\tau) = 4 \mid X(t) = 4) = P(T) + P(T) * P(X(t+\tau) = 4)$$

$$P(T) = \int_{t_1}^{t_2} \frac{1}{T}\,dt = \frac{\tau}{T} \Rightarrow P(T') = 1 - P(T') = 1 - \frac{\tau}{T}$$

$$R_X(\tau) = \frac{4^2}{2}\cdot\left(1 - \frac{|\tau|}{2T}\right) = 8\left(1 - \frac{|\tau|}{2T}\right)$$

  2. $|\tau| > T$

$$R_X(\tau) = E[X(\tau)\,X(t+\tau)]$$
$$= 4^2 * 0.5 * 0.5 + 0^2 * 0.5 * 0.5 + 4*0*0.5*0.5 + 0*4*0.5*0.5$$
$$= 4^2 * 0.5 * 0.5$$
$$= 4$$

  - And using the same flow, we can find that the ACF for "**Polar NRZ**" is

$$\text{if } |\tau| < T \rightarrow R_X(\tau) = 4^2 \cdot \left(1 - \frac{\tau}{T}\right) = 16\left(1 - \frac{|\tau|}{T}\right)$$
$$\text{if } |\tau| > T \rightarrow R_X(\tau) = \text{Zero}$$

- And similarly, the ACF for "**Polar RZ**" is

$$\text{if } |\tau| < \frac{T}{2} \rightarrow R_X(\tau) = \frac{4^2}{2}\cdot\left(1 - \frac{2|\tau|}{T}\right) = 8\left(1 - \frac{2|\tau|}{T}\right)$$
$$\text{if } |\tau| > \frac{T}{2} \rightarrow R_X(\tau) = \text{Zero}$$

And as we know:

$\mathit{Total\ Power} = RX(0)$ & $\mathit{DC\ Power} = RX(\infty)$.

$\mathit{AC\ Power} = \mathit{Total\ Power} - \mathit{DC\ Power}$.

	Unipolar NRZ	Polar NRZ	Polar RZ
Total Power	**8**	**16**	**9.147**
DC Power	**4**	**0**	**0**
AC Power	**4**	**16**	**9.147**

15.2.2. Code Snippet

```
function [Unipolar_AutoCorr, PolarNRZ_AutoCorr, PolarRZ_AutoCorr] =...
    compute_stat_autocorr(Unipolar_Shifted, PolarNRZ_Shifted, PolarRZ_Shifted, max_lag)
    %{...%}

    % Set x-axis limits dynamically
    x_limit = max_lag / 10;

    % Initialize autocorrelation arrays
    Unipolar_AutoCorr = zeros(1, max_lag + 1);
    PolarNRZ_AutoCorr = zeros(1, max_lag + 1);
    PolarRZ_AutoCorr = zeros(1, max_lag + 1);

    % Compute mean autocorrelation using calculate_mean function
    for i = 0:max_lag
        Unipolar_AutoCorr(i+1) = calculate_mean(Unipolar_Shifted(:, 1) .* Unipolar_Shifted(:, i+1));
        PolarNRZ_AutoCorr(i+1) = calculate_mean(PolarNRZ_Shifted(:, 1) .* PolarNRZ_Shifted(:, i+1));
        PolarRZ_AutoCorr(i+1) = calculate_mean(PolarRZ_Shifted(:, 1) .* PolarRZ_Shifted(:, i+1));
    end
```

Annotations

- The Statistical Autocorrelation is created by taking the element-wise product of each column with the first column of a selected matrix of data points, then averaging the resulting column-wise products.

- To guarantee that Autocorr is an even fun we concatenate between the result of fliplr fun & the averages vector before flipping (2:700 to ensure no repeated value at zero).

15.2.3. Plotting the statistical autocorrelation

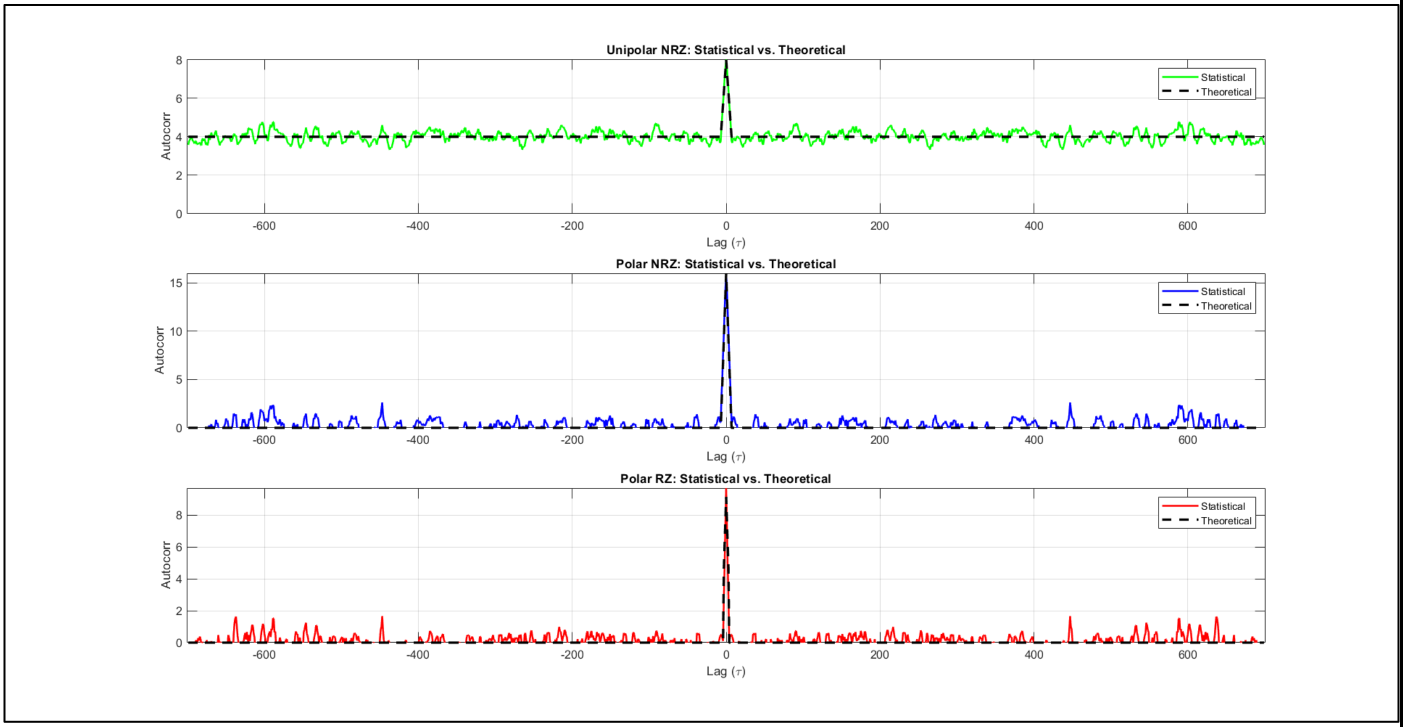

Figure 8 Statistical Auto Correction plot

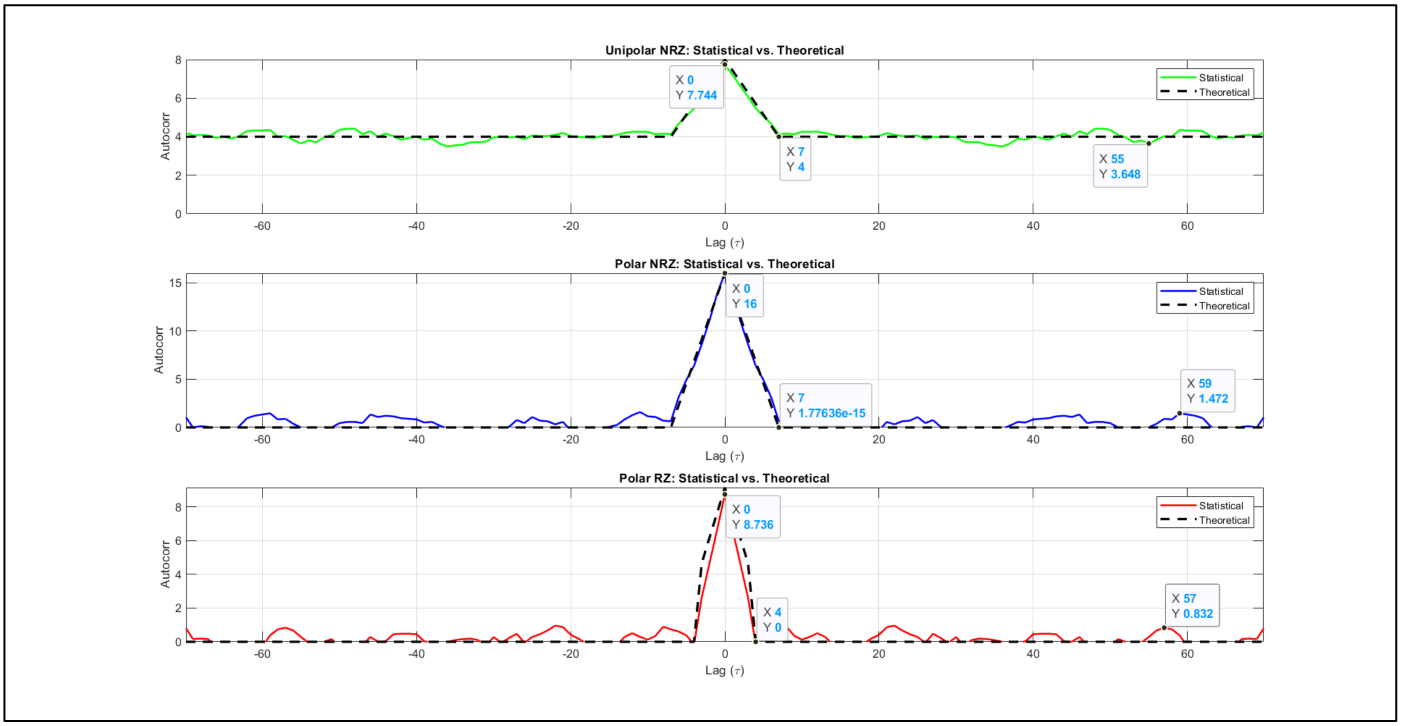

Figure 9 Statistical Auto Correction plot zoomed

The resulting autocorrelation values are plotted against the corresponding time delays (τ). We observe that at $\tau = 0$ the autocorrelation with the point itself is maximum, indicating perfect correlation.

- **Uni polar:** The autocorrelation becomes constant after 7 samples, as we calculated to be the bit duration and it's around 4, The maximum at zero equals $7.744 \approx 8$.

- **Polar NRZ:** The autocorrelation becomes constant after 7 samples, as we calculated to be the bit duration and it's around zero, The maximum at zero equals 16.

- **Polar RZ:** The autocorrelation becomes constant after 4 samples, as we calculated to be the half bit duration and it's around zero, The maximum at zero equals $8.736 \approx 9.147$.

15.3. Is the Process Stationary

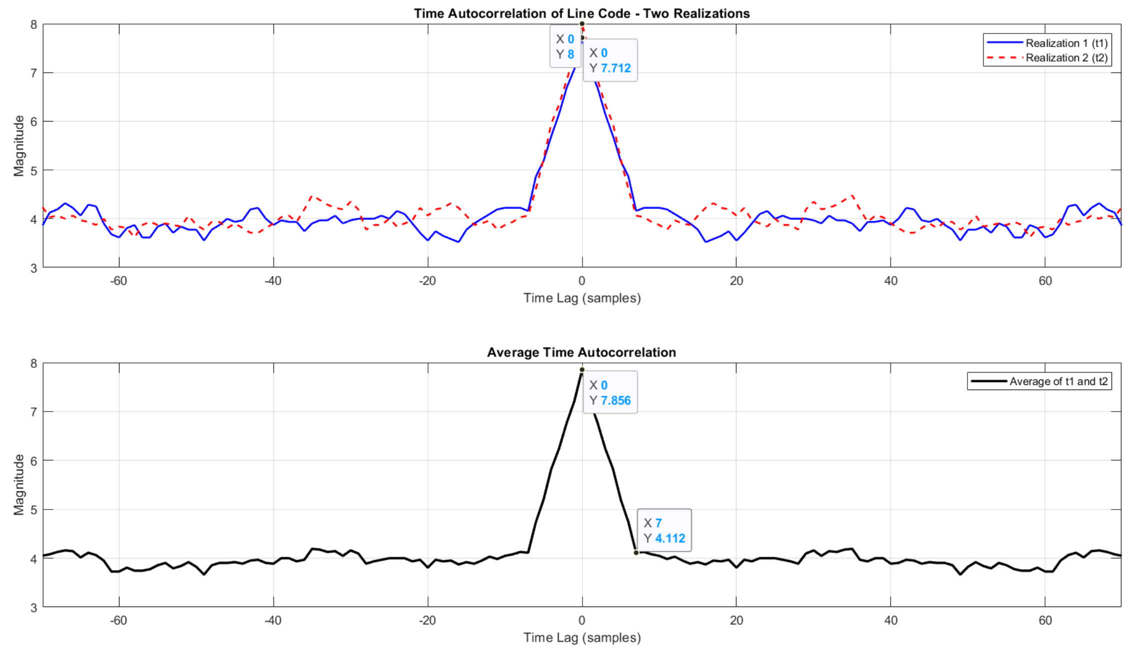

Figure 10 autocorrelation at two different times

- For the **mean**, as shown in section 1 figure 7 the **mean** ≈ **constant with time**.
- For the **autocorrelation**, as shown in figure 9 the **autocorrelation R(t1=1)** ≈ **R(t2=8).**

Yes, the process is stationary (WSSP) because the mean is constant function in time as shown in Figure 7 Plot of Statistical Mean and the autocorrelation depends only on the time difference not the absolute time.

15.4. The time mean and autocorrelation function for one waveform

15.4.1. Time Mean

```
function TimeMean = compute_time_mean(waveform_matrix)
    % Computes the time mean for each realization of a given waveform
    % Inputs:
    %   waveform_matrix - Matrix where each row represents a realization
    % Output:
    %   TimeMean - Column vector containing the time mean for each realization

    % Compute time mean for each realization (mean along rows)
    TimeMean = sum(waveform_matrix, 2) / size(waveform_matrix, 2);
end
```

- We add the values of a realization across time instant then divide by the number of samples (700 sample per realization).

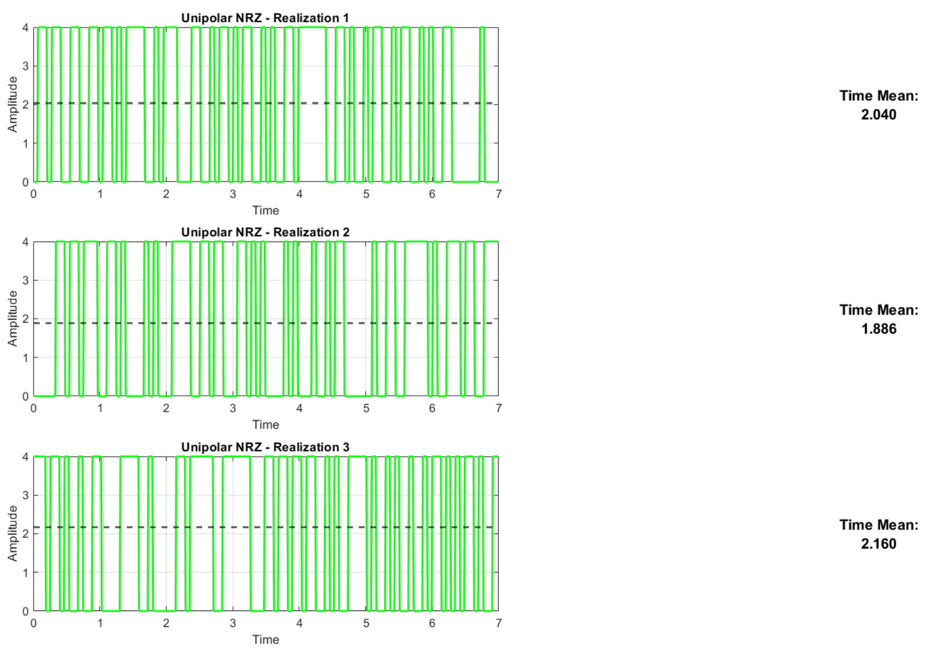

Figure 11 Time Mean for Uni Polar

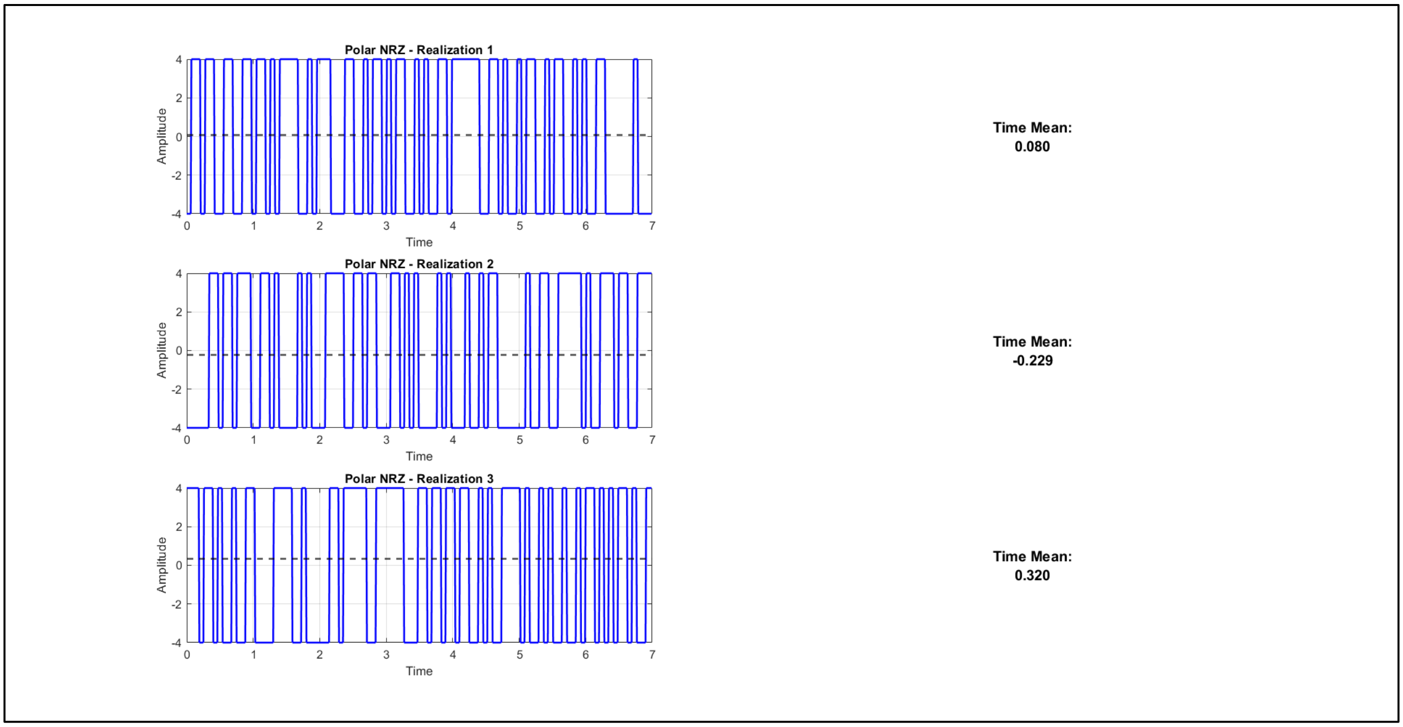

Figure 12 Time Mean for Polar NRZ

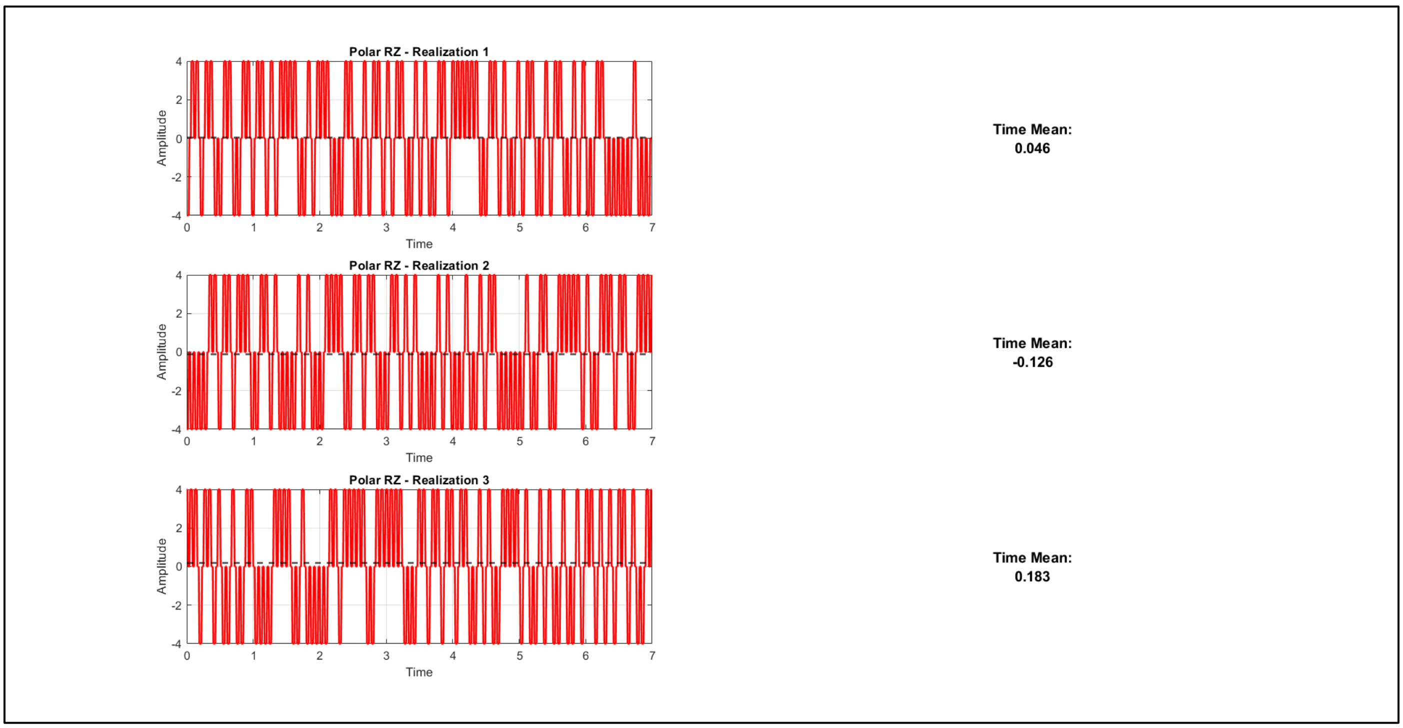

Figure 13 Time Mean for Polar RZ

- As expected, polar RZ & NRZ have almost zero mean and the uni polar has mean around 2 Because its amplitude ranges from 0 : 4

15.4.2. Time Mean Vs Realization

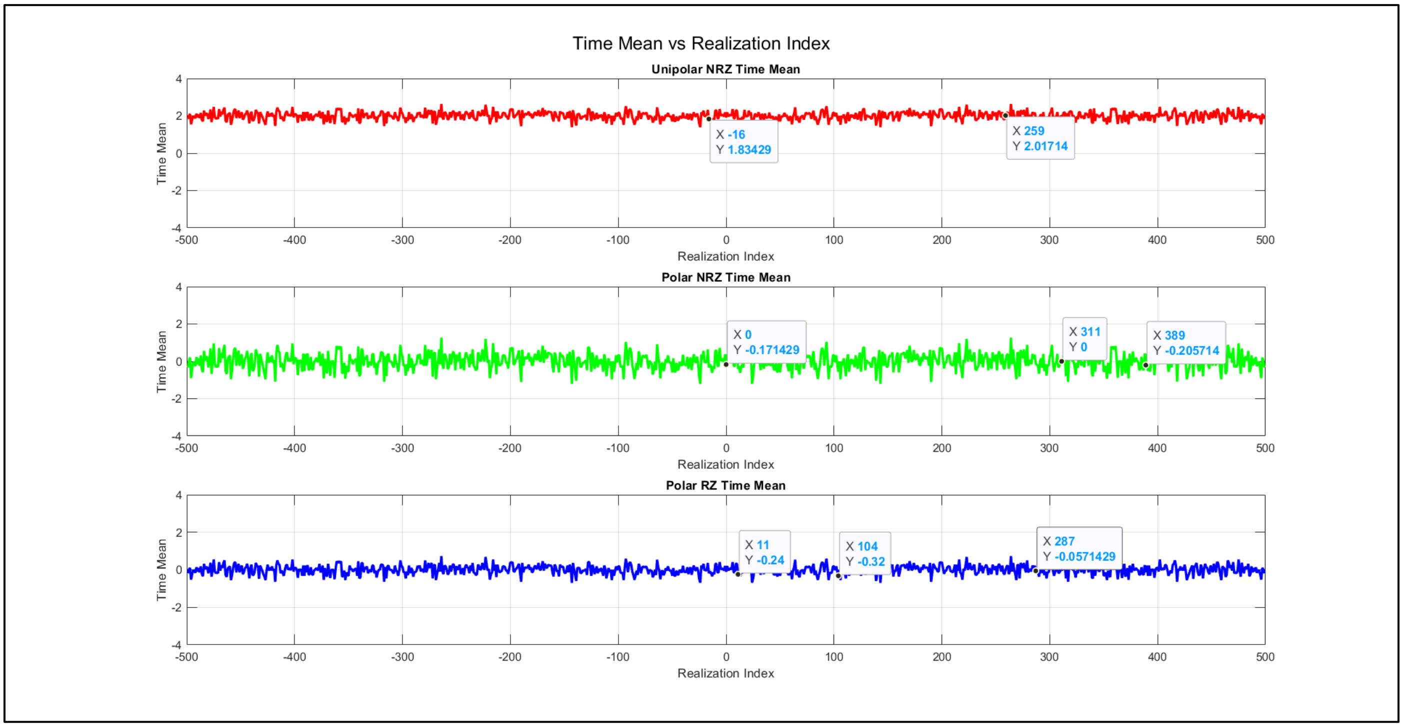

Figure 14 Time Mean Vs Realization

- As expected the time mean is almost equal to the statistical mean.
- Polar RZ & NRZ have almost zero mean and the uni polar has mean around 2.

15.4.3. Time Auto Correlation

For the time Auto Correlation we're going to use this function

```
function [R_unipolar_nrz_t1, R_polar_nrz_t1, R_polar_rz_t1, tau_vec] = ...
    compute_time_autocorr(UnipolarNRZ, PolarNRZ, PolarRZ, t1)...
```

```
% Preallocate
R_unipolar_nrz_t1 = zeros(1, length(tau_vec));
R_polar_nrz_t1 = zeros(1, length(tau_vec));
R_polar_rz_t1 = zeros(1, length(tau_vec));

for idx = 1:length(tau_vec)
    tau = tau_vec(idx);
    t2 = t1 + tau;


    % Compute element-wise products for all realizations at t1 and t1+tau
    prod_unipolar = UnipolarNRZ(:, t1) .* UnipolarNRZ(:, t2);
    prod_polar    = PolarNRZ(:, t1)     .* PolarNRZ(:, t2);
    prod_rz       = PolarRZ(:, t1)      .* PolarRZ(:, t2);

    % Use custom function to compute mean across realizations
    R_unipolar_nrz_t1(idx) = sum(prod_unipolar) / num_realizations;
    R_polar_nrz_t1(idx)    = sum(prod_polar)    / num_realizations;
    R_polar_rz_t1(idx)     = sum(prod_rz)       / num_realizations;

end
```

- The time autocorrelation is calculated by $r_{xx} = \frac{1}{N}\sum_{n=0}^{N-1} x(n)x(n-k)$ of the first waveform.

15.4.4. Time Auto Correlation for one wave form:

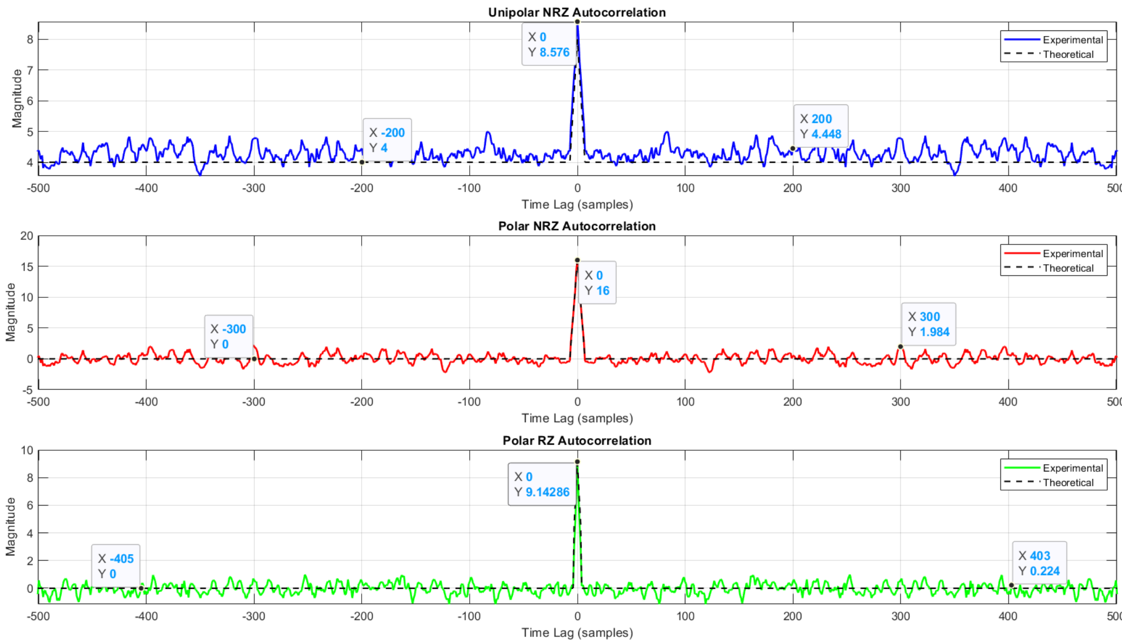

Figure 16 Time Auto Correction plot

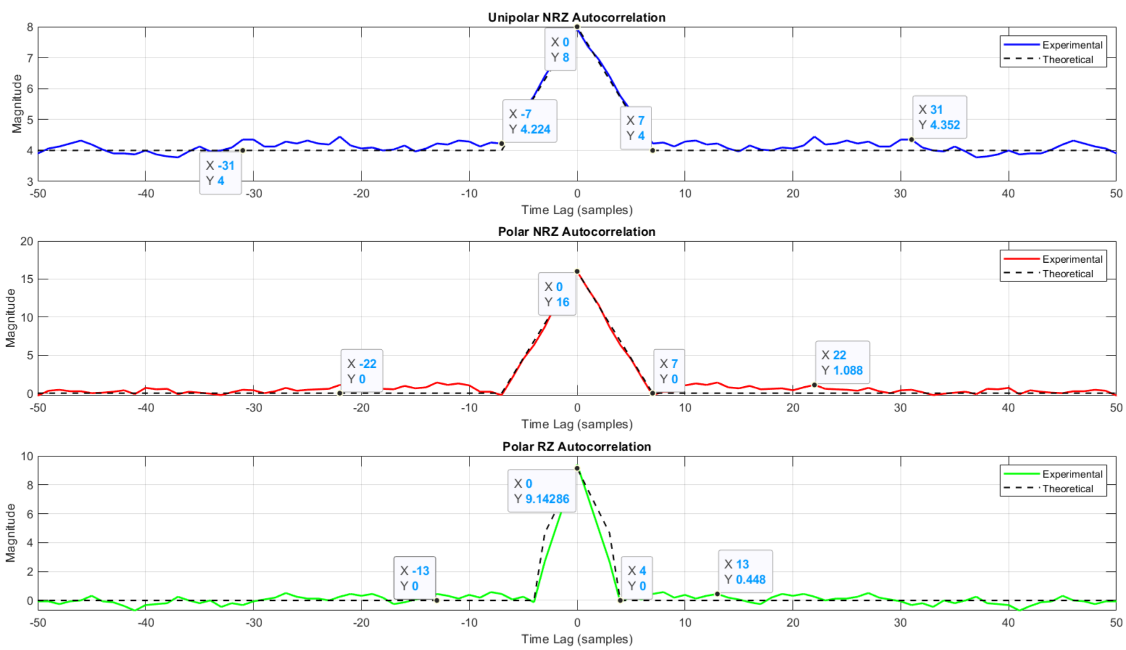

Figure 15 Time Auto Correction plot zoomed

As shown in the graphs:

- The time autocorrelation is closely same as the ensemble autocorrelation.
- The autocorrelation function has maximum at $\tau = 0$ and it is an even function.

15.5. Is The Random Process Ergodic?

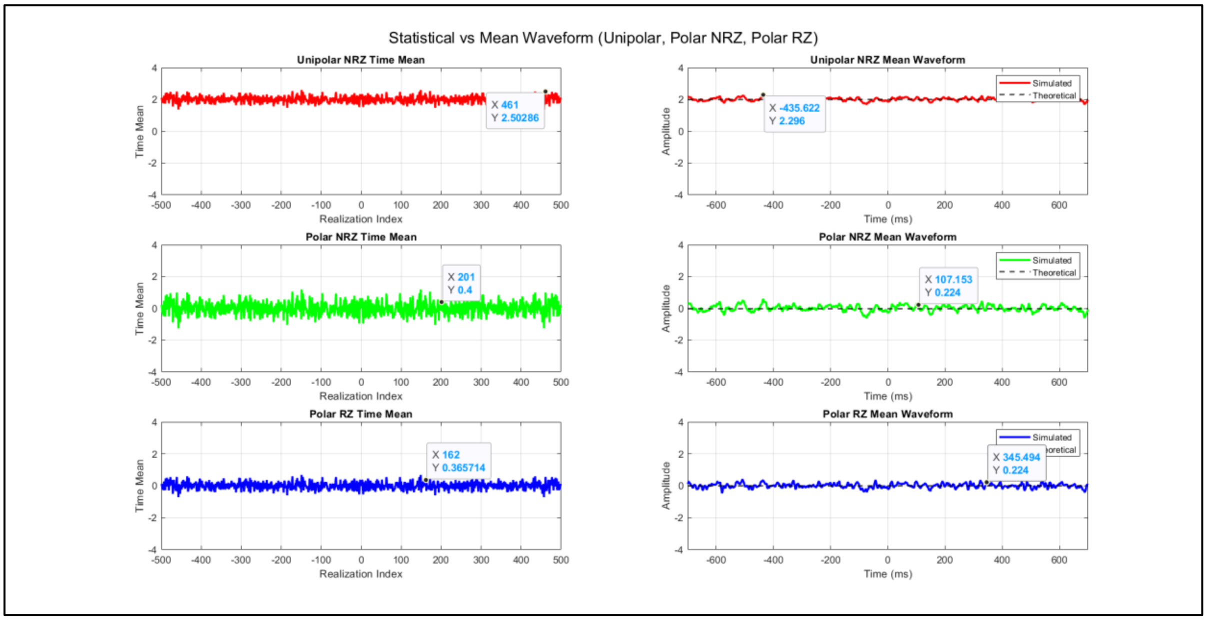

Figure 17 Time Mean vs Statistical

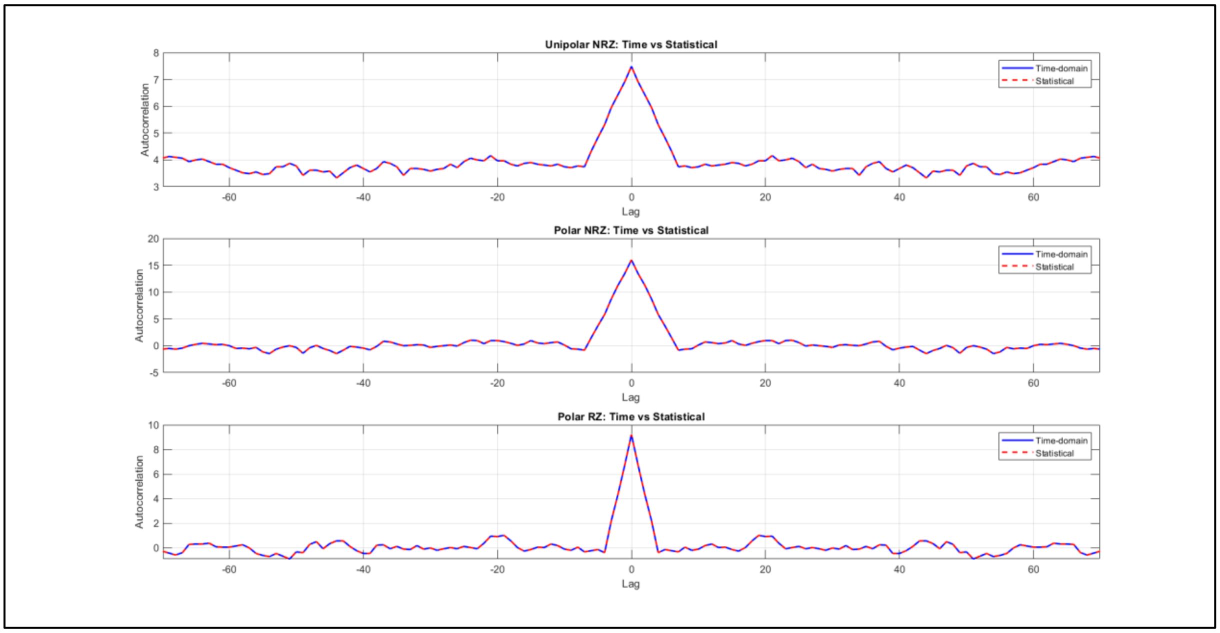

Figure 18 Time Auto Correlation Vs Statistical

- For the **mean**, the Time mean is almost equal to the statistical mean.

- For the **Auto Correlation,** the Time looks almost identical to the statistical.
  But, There not fully identical as we ran this code snippet

  ```
  disp(R_unipolar_full - Unipolar_AutoCorr);
  ```

  And the result was **0.5760, so they are almost Identical.**

- Yes, because the time mean ≈ the Statistical mean and the time autocorrelation is ≈ the ensemble autocorrelation.
  Then this process is ergodic

15.6. the PSD & Bandwidth of the Ensemble

15.6.1. PSD using fft:

For the **PSD,** we are going to use this function:

```
function [PSD_unipolar ,PSD_polarNRZ ,PSD_polarRZ] =...
    plot_linecode_psd(R_Unipolar, R_PolarNRZ, R_PolarRZ, fs, A, Tb)
```

```
% Compute FFTs of autocorrelations
fft_unipolar = fft(R_Unipolar) / n;
fft_polarNRZ = fft(R_PolarNRZ) / n;
fft_polarRZ  = fft(R_PolarRZ) / n;

% Compute PSD magnitudes
PSD_unipolar = abs(fft_unipolar);
PSD_polarNRZ = abs(fft_polarNRZ);
PSD_polarRZ  = abs(fft_polarRZ);

% Frequency axis centered around 0
freq_axis = (-n/2 : n/2 - 1) * (fs / n);

% Center the FFTs for proper plotting
PSD_unipolar = A*fftshift(PSD_unipolar);
PSD_polarNRZ = A*fftshift(PSD_polarNRZ);
PSD_polarRZ  = A*fftshift(PSD_polarRZ);
```

- We take the Fourier transform of the avg time autocorrelation **= 0.5*(R(t1)+ R(t2))** then centralize the graph around zero.
- since $T_s = \frac{\text{Bit time}}{\text{no of samples per bit}} = \frac{70\text{ ms}}{7} = 10\text{ ms} \rightarrow F_s = 100$

- **For the BW**
  - the BW is the frequency of the first zero of sinc^2 function (intersection with frequency-axis)

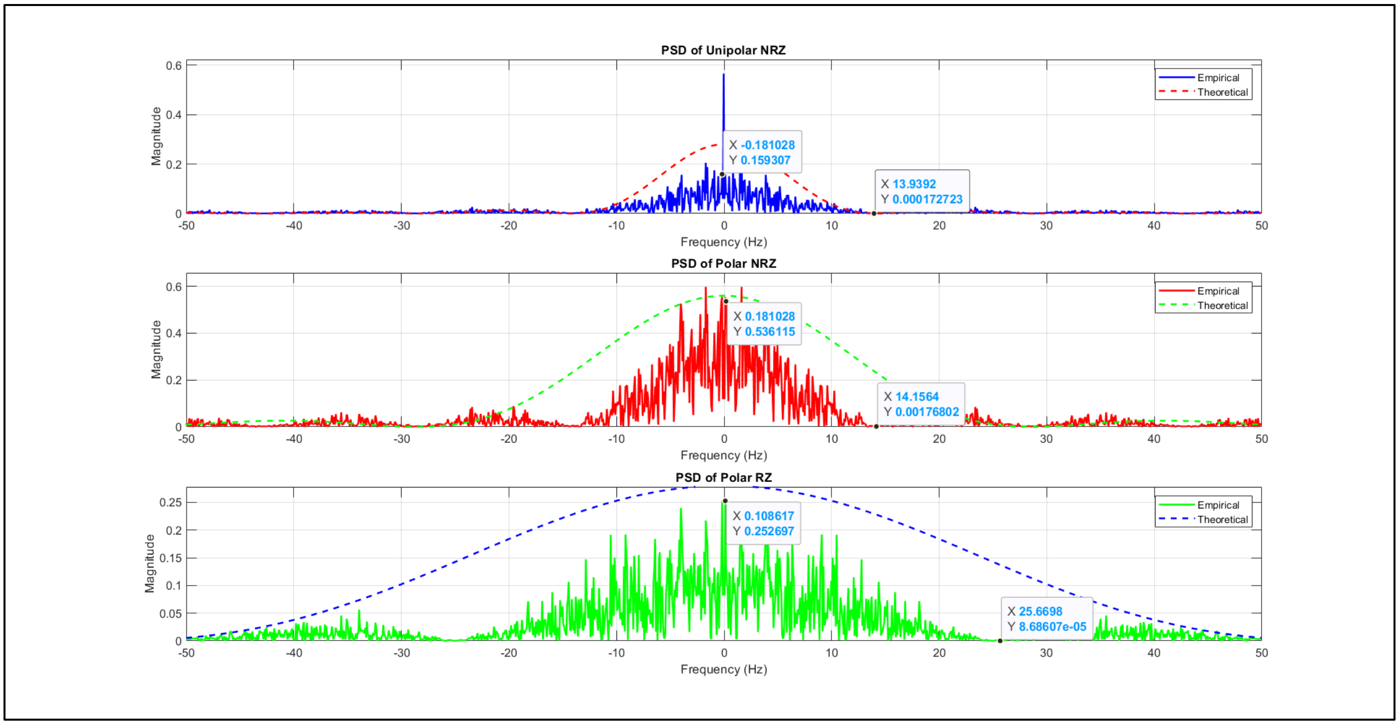

Figure 19 PSD plot of the Ensemble

Annotations

- in polar RZ & NRZ : we have sinc^2 function without delta at zero frequency (NO DC)
- in uni polar NRZ : we have sinc^2 function with delta at zero frequency (there is DC)
- BW of the unipolar NRZ & polar NRZ is the bitrate which approximately equal 14 hz
- BW of the polar RZ is the double of bitrate which approximately equal 25.66 hz

15.6.2. Theoritical PSD:

From **references**[1], [2], we found out that the PSDs are:

Line Code	PSD
Uni Polar	$S(f)=A^2/4*Tb*(\sin(\pi fTb)/\pi fTb)^2 + A^2/4*\delta(f)$
Polar NRZ	$S(f)=A^2*Tb*(\sin(\pi fTb/2)/\pi fTb/2)^2$
Polar RZ	$S(f)=A^2*Tb*(\sin(\pi fTb/4)/\pi fTb/4)^2$

Note that:

- Uni polar has a DC pulse which is noticeable in figure 19
- Polar don't have the DC pulse
- Polar RZ has double the frequency of Polar NRZ
- A=4, Tb = 70 ms

So by comparing the practical vs theoretical:

Line Code	Theoritcal PSD at f=0	Paractical PSD at f=0
Uni Polar	$A^2/4*Tb$ =0.28	0.159
Polar NRZ	$A^2/2*Tb$ =0.56	0.536
Polar RZ	$A^2/4*Tb$ =0.28	0.252

For **BW:**

Line Code	Theoritcal BW	Paractical BW
Uni Polar	1/Tb = 14.285 Hz	13.972 Hz
Polar NRZ	1/Tb = 14.285 Hz	14.15 Hz
Polar RZ	2/Tb = 28.57 Hz	25.66 Hz

16. References:

[1] **Dr. Mohammed Nafie,** "Lecture Slides in Advanced Communications," *ELC4020 Advanced Communication Systems*, Cairo University, 2025.

[2] **Eng. Mohamed Khaled,** "Section Slides in Advanced Communications," *ELC4020 Advanced Communication Systems*, Cairo University, 2025.

[**3**] https://github.com/youefkh05/Advanced_Communication_Coding

17. Appendix

```
%%------------------------------------------------------------
% Problem 1: Binary Huffman Coding
%------------------------------------------------------------

clc; clear; close all force;

% Given Symbols probabilities
symbols = {'A','B','C','D','E','F','G'};
P = [0.35 0.30 0.20 0.10 0.04 0.005 0.005];

% Create Input Dictionary
[dict_input,err_flag, H] = create_symbols_dictionary(symbols, P);

% Check Input
if err_flag ==1
    disp('? Stopping execution due to invalid dictionary.');
    return; % exits the current script or function
end

% Print the dictionary neatly
print_symbols_dic(dict_input, H);


% -----------------------------------------------------------------------
% Manual Huffman Coding (with custom output)
% -----------------------------------------------------------------------
dict_huffman = huffman_encoding_visual(dict_input);

disp('--- Manual Huffman Encoding ---');
disp(dict_huffman);

% Print the codded dictionary neatly
print_coded_dict(dict_huffman, H);


%%------------------------------------------------------------
```

```
% Problem 2: Binary Fano  Coding
%-------------------------------------------------------------

%{
????? ??????? ?? ?????
???????? ?? ??
input
???????
dict_input

??
funtions
H
L
eta
??? ?????? ?????

???? ?????? ??????  ??
dict_Fano  = Fano_encoding_visual(dict_input);
???? ?? ?? function
???? ??? ????
%}

% ---------------------------------------------------------------------------
% Manual Fano Coding (with custom output)
% ---------------------------------------------------------------------------
dict_Fano  = Fano_encoding_visual(dict_input);

disp('--- Manual Fano Encoding ---');
disp(dict_Fano);

% Print the codded dictionary neatly
print_coded_dict(dict_Fano, H);




%%
% ---------------------------------------------------------------------------
%              Function Definition
% ---------------------------------------------------------------------------

%% ---------------------------------------------------------------------------
%              Entropy Calculation
% ---------------------------------------------------------------------------

function H = entropy_calc(P)
%ENTROPY_CALC Compute the source entropy H(P(x))
%   H = entropy_calc(P)
%   P : vector of symbol probabilities
%   H : entropy in bits

    % Validate input
    if any(P < 0) || abs(sum(P) - 1) > 1e-6
        warning('Probabilities should sum to 1. Normalizing...');
```

```
        P = P / sum(P);
    end

    % Remove zeros (to avoid log2(0))
    P = P(P > 0);

    % Compute entropy
    H = -sum(P .* log2(P));
end

%% ---------------------------------------------------------------------------
%               Average Length Calculation
% ---------------------------------------------------------------------------
function L = average_length_calc(dict)
%AVERAGE_LENGTH_CALC Compute average codeword length L(C)
%   L = average_length_calc(dict, P)
%   dict : Huffman dictionary cell array {symbol, code}
%   P : vector of symbol probabilities (same order as dict)
%
%   If P is empty, it tries to extract from dict(:,2) if present
%   L : average code length

    P = dict(:,2);
    % --- Handle inputs ---
    if nargin < 2 || isempty(P)
        % Check if dict has a probability column (3 columns)
        if size(dict, 2) >= 3 && isnumeric(dict{1,2})
            P = cell2mat(dict(:,2));
            codes = dict(:,3);
        else
            error('Probability vector P is required or must be in dict(:,2)');
        end
    else
        % Extract codes (assumed in 2nd column)
        codes = dict(:,3);
    end

    % --- Compute code lengths ---
    code_lengths = cellfun(@length, codes);

    % --- Normalize probabilities ---
    P = P(:) / sum(P);

    % --- Check dimensions ---
    if length(P) ~= length(code_lengths)
        error('Number of probabilities does not match number of codewords.');
    end

    % --- Compute average code length ---
    L = sum(P .* code_lengths);
end

%% ---------------------------------------------------------------------------
%               Efficiency Calculation
% ---------------------------------------------------------------------------
```

```
function eta = efficiency_calc(H, L)
%EFFICIENCY_CALC Compute Huffman coding efficiency ?
%   eta = efficiency_calc(H, L)
%   H : entropy
%   L : average code length
%   eta : efficiency in percentage (%)

    if L <= 0
        error('Average length L must be positive.');
    end

    eta = (H / L) * 100;
end


%% -----------------------------------------------------------------------
%                Print Kraft Inequality Function
% -----------------------------------------------------------------------

function [kraft_sum, kraft_flag]=kraft_analysis(dict)
% KRAFT_ANALYSIS  Compute Kraft's inequality and visualize Kraft tree
%   kraft_analysis(dict)
%
%   Input:
%       dict : cell array {N x 3}  ? {symbol, P, code}
%
%   Example:
%       dict = {'A','0'; 'B','10'; 'C','110'; 'D','111'};
%       kraft_analysis(dict);

    if ~iscell(dict) || size(dict,2) < 2
        error('Input must be a cell array {symbol, code}');
    end

    % Extract codes
    codes = dict(:,3);
    N = length(codes);

    % --- 1. Compute Kraft's inequality ---
    code_lengths = cellfun(@length, codes);
    kraft_sum = sum(2.^(-code_lengths));

    fprintf('\n=== Kraft Inequality Check ===\n');
    fprintf('Sum(2^{-l_i}) = %.4f\n', kraft_sum);



    if abs(kraft_sum - 1) < 1e-6
        fprintf('? Code satisfies equality ? Complete Prefix Code.\n');
        kraft_flag=2;
    elseif kraft_sum < 1
        fprintf('? Code satisfies inequality (valid but not complete).\n');
        kraft_flag=1;
```

```
    else
        fprintf('? Invalid prefix code (violates Kraft''s inequality).\n');
        kraft_flag=0;
    end

end



%% ---------------------------------------------------------------------------
%               Create Dictionary Input Definition
% ---------------------------------------------------------------------------
function [dict_input,err_flag, H] = create_symbols_dictionary(symbols, P)
%CREATE_DICTIONARY  Combines symbols and probabilities into a validated dictionary.
%
%   dict_input = create_dictionary(symbols, P)
%
%   Inputs:
%       symbols - cell array of symbols, e.g. {'A','B','C'}
%       P       - corresponding probabilities (row or column vector)
%
%   Output:
%       dict_input - cell array {symbol, probability}
%
%   Example:
%       symbols = {'A','B','C'};
%       P = [0.5 0.3 0.2];
%       dict_input = create_dictionary(symbols, P);

    % Combine into dictionary-like cell array
    dict_input = [symbols(:), num2cell(P(:))];

    % Assume not great until great
    err_flag = 1;

    % Validate using the check_symbols() function
    [ok, msg] = check_symbols(dict_input);

    % Display validation result
    if ok
        disp('? Dictionary is valid!');
        err_flag =0;
    else
        disp(['? Error: ' msg]);
        err_flag = 1;
    end

    % ---------------------------------------------------------------------
    % Compute entropy (only if valid)
    % ---------------------------------------------------------------------
    H = entropy_calc(P)

end


%% ---------------------------------------------------------------------------
```

```
%                Check Input Validation Function
% --------------------------------------------------------------------------
function [isValid, errMsg] = check_symbols(dict_input)
% CHECK_SYMBOLS  Validates a symbol-probability dictionary
%
%   [isValid, errMsg] = check_symbols(dict_input)
%
%   Input:
%       dict_input : Cell array {N×2}, where first column = symbols,
%                    second column = probabilities
%
%   Output:
%       isValid : Logical true if valid, false otherwise
%       errMsg  : String describing validation error (if any)

    % Default output
    isValid = false;
    errMsg  = '';

    try
        % Extract symbols and probabilities
        symbols = dict_input(:, 1);
        P = cell2mat(dict_input(:, 2));

        % Check same length
        if numel(symbols) ~= numel(P)
            errMsg = 'Symbols and probabilities must have the same length.';
            return;
        end

        % Check probabilities sum to 1 (within tolerance)
        if abs(sum(P) - 1) > 1e-6
            errMsg = sprintf('Probabilities do not sum to 1 (sum = %.6f).',
      sum(P));
            return;
        end

        % Check all probabilities are positive
        if any(P <= 0)
            errMsg = 'All probabilities must be positive.';
            return;
        end

        % If all checks passed
        isValid = true;

    catch ME
        errMsg = ['Invalid dictionary input: ' ME.message];
    end
end


%% --------------------------------------------------------------------------
%                Print Dictionary Function
% --------------------------------------------------------------------------
```

```
function print_symbols_dic(dict_input, H)
% PRINT_SYMBOLS_DIC  Displays a formatted version of the symbol dictionary in a
    figure,
%                    and shows the calculated source entropy.
%
%   print_symbols_dic(dict_input, H)
%
%   Inputs:
%       dict_input - cell array {symbol, probability}
%       H          - source entropy (bits/symbol)

    % Validate input
    if nargin < 1 || isempty(dict_input)
        error('Input dictionary is empty or missing.');
        return;
    end

    % Convert symbols to char (uitable can't handle string objects)
    symbols = cellfun(@char, dict_input(:,1), 'UniformOutput', false);
    probs = cell2mat(dict_input(:,2));

    % Display result in Command Window
    fprintf('\nInformation Source Entropy: H = %.4f bits/symbol\n', H);
    fprintf('-----------------------------------------------------\n');

    % Create a responsive UI figure
    f = uifigure('Name', 'Symbol Dictionary', ...
                 'NumberTitle', 'off', ...
                 'Color', 'w', ...
                 'Position', [500 400 350 320]);

    % Format probabilities as strings
    probStr = arrayfun(@(p) sprintf('%.4f', p), probs, 'UniformOutput', false);

    % Combine into table data
    data = [symbols probStr];

    % Create a grid layout (auto-resizes)
    gl = uigridlayout(f, [3,1]);
    gl.RowHeight = {'fit', '1x', 'fit'};  % title, table, entropy
    gl.ColumnWidth = {'1x'};
    gl.Padding = [10 10 10 10];

    % --- Title ---
    uilabel(gl, ...
        'Text', '--- Input Symbol Dictionary ---', ...
        'FontSize', 14, ...
        'FontWeight', 'bold', ...
        'HorizontalAlignment', 'center');

    % --- Table ---
    uitable(gl, ...
        'Data', data, ...
        'ColumnName', {'Symbol', 'Probability'}, ...
        'FontSize', 12, ...
```

```
        'ColumnWidth', {'1x', '1x'}, ...
        'RowStriping', 'on');

    % --- Entropy Display ---
    uilabel(gl, ...
        'Text', sprintf('Entropy: H = %.4f bits/symbol', H), ...
        'FontSize', 12, ...
        'FontWeight', 'bold', ...
        'FontColor', [0 0.3 0.7], ...
        'HorizontalAlignment', 'center');
end

%% --------------------------------------------------------------------------
%                Print Coded Dictionary Function (with Kraft Tree)
% --------------------------------------------------------------------------
function print_coded_dict(dict, H)
% PRINT_CODED_DICT  Display Huffman dictionary with entropy, avg length,
      efficiency, and Kraft tree.
%
%   print_coded_dict(dict, H)
%
%   Inputs:
%       dict - cell array {symbol, probability, code}
%       H    - entropy (bits/symbol)
%
%   This function:
%       • Calculates average length L(C)
%       • Calculates efficiency ? = (H / L) * 100%
%       • Checks Kraft’s inequality and plots the Kraft tree
%       • Displays all results in MATLAB UI + console

    % === Validate input ===
    if nargin < 1 || isempty(dict)
        disp('Input Huffman dictionary is missing or empty.');
        return;
    end

    if size(dict,2) < 3
        disp('Dictionary must have 3 columns: {symbol, probability, code}.');
        return;
    end

    % === Extract data ===
    symbols = cellfun(@char, dict(:,1), 'UniformOutput', false);
    P       = cell2mat(dict(:,2));
    codes   = dict(:,3);

    % === Compute metrics ===
    L   = average_length_calc(dict);
    eta = efficiency_calc(H, L);
    [kraft_sum, kraft_flag] = kraft_analysis(dict);

    % === Print to Command Window ===
    fprintf('\n--- Final Huffman Coding Results ---\n');
    fprintf('Symbol\tProb.\t\tCode\n');
```

```
    fprintf('------------------------------------------\n');
    for i = 1:length(symbols)
        fprintf('%s\t%.4f\t\t%s\n', symbols{i}, P(i), codes{i});
    end
    fprintf('------------------------------------------\n');
    fprintf('Entropy (H):           %.4f bits/symbol\n', H);
    fprintf('Average length (L):    %.4f bits/symbol\n', L);
    fprintf('Efficiency (?):        %.2f %%\n', eta);
    fprintf('Kraft Sum:             %.4f\n', kraft_sum);
    if kraft_flag == 2
        fprintf('Kraft Result: ? Complete Prefix Code\n');
    elseif kraft_flag == 1
        fprintf('Kraft Result: ? Valid but Not Complete\n');
    else
        fprintf('Kraft Result: ? Invalid Code\n');
    end

    % === UI Figure ===
    f = uifigure('Name','Huffman Dictionary Summary', ...
                 'NumberTitle','off', ...
                 'Color','w', ...
                 'Position',[500 200 480 450]);

    gl = uigridlayout(f,[3 1]);
    gl.RowHeight = {'fit', '1x', 'fit'};
    gl.Padding = [10 10 10 10];

    % --- Title ---
    uilabel(gl, ...
        'Text','--- Huffman Coded Dictionary ---', ...
        'FontSize',14, ...
        'FontWeight','bold', ...
        'HorizontalAlignment','center');

    % --- Table ---
    data = [symbols, arrayfun(@(p) sprintf('%.4f',p), P,'UniformOutput',false),
      codes];
    uitable(gl, ...
        'Data',data, ...
        'ColumnName',{'Symbol','Probability','Code'}, ...
        'FontSize',12, ...
        'RowStriping','on', ...
        'ColumnWidth',{'1x','1x','1x'});

    % --- Summary Labels ---
    uilabel(gl, ...
        'Text', sprintf('H = %.4f | L = %.4f | ? = %.2f %% | Kraft = %.4f', H, L,
      eta, kraft_sum), ...
        'FontSize',12, ...
        'FontWeight','bold', ...
        'FontColor',[0 0.3 0.7], ...
        'HorizontalAlignment','center');

end
```

```matlab
%% ---------------------------------------------------------------------------
%               Huffman Encoding with Visualization Function
% ---------------------------------------------------------------------------

function dict = huffman_encoding_visual(dict_input)
%HUFFMAN_ENCODING_VISUAL Visual Huffman encoding with full table output (UI-based)
%
%   dict = huffman_encoding_visual(symbols, P)
%   - symbols: cell array of symbol names (e.g. {'A','B','C','D','E','F','G'})
%   - P: vector of probabilities (same length as symbols)
%
%   Creates a UI figure showing the probability & code propagation table,
%   and prints the final Huffman dictionary.

    % get the info from dictionary
    symbols = dict_input(:,1);
    P = cell2mat(dict_input(:,2));
    % === Input Validation ===
    if numel(symbols) ~= numel(P)
        error('Symbols and probabilities must have same length.');
    end

    % === Normalize probabilities ===
    P = P(:);
    P = P / sum(P);

    % === Step 1: Generate merging history ===
    history_table = merge_probabilities(P);

    % === Step 2: Assign Huffman codes ===
    history_table_full = assign_coding(history_table);

    % === Step 3: Prepare data for visualization ===
    % Convert numeric NaNs to empty strings for table display
    final_visual_data = cell(size(history_table_full));
    for r = 1:size(history_table_full,1)
        for c = 1:size(history_table_full,2)
            val = history_table_full{r,c};
            if isnumeric(val)
                if isnan(val)
                    final_visual_data{r,c} = '';
                else
                    final_visual_data{r,c} = num2str(val, '%.4f');
                end
            else
                final_visual_data{r,c} = val;
            end
        end
    end

    % Generate column headers (P1, C1, P2, C2, ...)
    numCols = size(history_table_full,2);
    final_visual_headers = cell(1,numCols);
    for c = 1:numCols
```

```
    if mod(c,2)==1
        final_visual_headers{c} = sprintf('P%d', ceil(c/2)-1);
    else
        final_visual_headers{c} = sprintf('C%d', ceil(c/2)-1);
    end
end

% === Step 4: Build UI Visualization ===
close all;
f = uifigure('Name','Huffman Encoding Visualization', ...
             'Position',[100 100 1000 500]);
gl = uigridlayout(f,[2 1]);
gl.RowHeight = {'fit','1x'};

uilabel(gl, ...
    'Text','Huffman Encoding: Probability and Code Evolution (P/C Steps)', ...
    'FontSize',16, ...
    'FontWeight','bold', ...
    'HorizontalAlignment','center');

% Column widths (narrow for numeric columns, wider for code columns)
col_widths = repmat({70}, 1, numCols);
col_widths(2:2:end) = {100}; % widen code columns

uitable(gl, ...
    'Data',final_visual_data, ...
    'ColumnName',final_visual_headers, ...
    'RowName',{}, ...
    'FontSize',12, ...
    'ColumnWidth',col_widths, ...
    'RowStriping','on', ...
    'BackgroundColor',[1 1 1; 0.95 0.95 1]);

% === Step 5: Extract Final Huffman Dictionary ===

% Make a copy
dict = dict_input;

% Ensure dict has at least 3 columns
if size(dict,2) < 3
    dict(:,end+1:3) = {[]};
end
dict(:,3)=history_table_full(:,2);

% === Step 6: Console Output ===
firstPcol = 1;
firstCcol = 2;

probs = cell2mat(history_table_full(:, firstPcol));
codes = history_table_full(:, firstCcol);
validIdx = ~isnan(probs);

symbols = symbols(validIdx);
codes = codes(validIdx);
probs = probs(validIdx);
```

```
    fprintf('\n--- Final Huffman Codes ---\n');
    for i = 1:length(symbols)
        fprintf('Symbol %s (%.4f): %s\n', symbols{i}, probs(i), codes{i});
    end
    fprintf('================================================================\n\n');
end

% === Probability Merge helper function  ===
function history_table = merge_probabilities(P)
%MERGE_PROBABILITIES Builds Huffman probability merging history (descending)
%
%   history_table = merge_probabilities(P)
%
%   Input:
%       P - vector of symbol probabilities
%
%   Output:
%       history_table - table of probabilities after each merge
%                       Columns: P0, P1, P2, ... (N-1 total)
%
%   Note: Probabilities are shown in descending order.

    % --- Input check ---
    if numel(P) < 2
        error('At least two probabilities are required.');
    end

    % --- Initialization ---
    P = P(:);
    P = sort(P, 'descend'); % sort descending
    N = numel(P);

    % Number of P columns = N - 1
    numCols = N - 1;
    maxRows = N;

    % Initialize history as cell
    history = cell(maxRows, numCols);

    % --- Step 0: Fill P0 (descending order) ---
    for i = 1:maxRows
        history{i,1} = P(i);
    end

    curP = P;

    % --- Iteratively merge ---
    for step = 2:numCols
        % Sort ascending to pick smallest two
        curP = sort(curP, 'ascend');
        if numel(curP) >= 2
            p1 = curP(1);
            p2 = curP(2);
            mergedP = p1 + p2;
```

```
            % Remove two smallest and add merged one
            curP = [mergedP; curP(3:end)];
        end
        % Sort descending for display
        curP = sort(curP, 'descend');

        % Fill current column
        for r = 1:maxRows
            if r <= numel(curP)
                history{r,step} = curP(r);
            else
                history{r,step} = NaN;
            end
        end
    end

    % --- Column names ---
    colNames = cell(1,numCols);
    for i = 1:numCols
        colNames{i} = sprintf('P%d', i-1);
    end

    % --- Convert to table ---
    history_table = cell2table(history, 'VariableNames', colNames);
end

% === Assign code helper function  ===
function history_table_full = assign_coding(history_table)
    % assign_coding - expands the history table and assigns Huffman codes
    %
    % Input:
    %   history_table : numeric matrix or table (probability merging history)
    %
    % Output:
    %   history_table_full : cell array with 2N columns
    %       Odd columns: probability values
    %       Even columns: assigned codes

    % If table, convert to numeric array
    if istable(history_table)
        history_table = table2array(history_table);
    end

    % Determine size
    [numRows, numCols] = size(history_table);
    newCols = 2 * numCols;

    % Initialize
    history_table_full = cell(numRows, newCols);

    % === Fill odd columns with probabilities ===
    for col = 1:numCols
        history_table_full(:, 2 * col - 1) = num2cell(history_table(:, col));
    end
```

```
% === Initialize last code column (start with last merge) ===
lastPcol = 2 * numCols - 1;
lastCcol = lastPcol + 1;
history_table_full{1, lastCcol} = '0';
history_table_full{2, lastCcol} = '1';
raw_counter=1;  %for parent assignment

% === Backward propagation of codes ===
for col = numCols:-1:2  % start from last column going backward
    currPcol = 2 * col - 1;
    currCcol = currPcol + 1;
    prevPcol = 2 * (col - 1) - 1;
    prevCcol = prevPcol + 1;
    raw_counter = raw_counter+1;

    % Get non-NaN values from P prev column
    prevPvals = cell2mat(history_table_full(:, prevPcol));
    prevPvals = prevPvals(~isnan(prevPvals));

    % Get non-NaN values from C curr column
    currCvals  = history_table_full(:, currCcol);


    % Identify merged value
    if length(prevPvals) >= 2
        mergedVal = prevPvals(end) + prevPvals(end-1);
    else
        continue;
    end

    % Find which row in current P col matches the mergedVal
    currPvals = cell2mat(history_table_full(:, currPcol));
    matchIdx = find(abs(currPvals - mergedVal) < 1e-12);
    if numel(matchIdx) > 1
        matchIdx = matchIdx(1); % take top one if duplicate
    end

    % Get parent code
    parentCode = history_table_full{matchIdx, currCcol};
    if isempty(parentCode)
        parentCode = '';
    end

    % === Assign child codes ===
    % Last two rows in previous P column are merged into this parent
    history_table_full{raw_counter, prevCcol}   = [parentCode '0'];
    history_table_full{raw_counter+1, prevCcol} = [parentCode '1'];

    % For each previous non-merged row (in display order top->bottom)
    for ii = 1:(raw_counter-1)
        % Skip the rows that were just merged (raw_counter and raw_counter+1)

        % Get the probability value in the previous column for this row
        valPrev = history_table_full{ii, prevPcol};
        if isnan(valPrev)
```

```
            continue; % nothing to copy
        end

        % Find matching value in the current column (exclude merged parent)
        currMatches = find(abs(currPvals - valPrev) < 1e-12);

        % Remove the matchIdx (the merged parent) if it appears
        currMatches(currMatches == matchIdx) = [];

        if isempty(currMatches)
            continue; % no corresponding match found
        end

        % If there are duplicates (two identical probabilities)
        if numel(currMatches) > 1
            % take both, and copy their codes to the two rows
            history_table_full{ii,   prevCcol} = currCvals{currMatches(1)};
            if (ii+1) <= numRows
                history_table_full{ii+1, prevCcol} = currCvals{currMatches(2)};
            end
        else
            % single match — copy code directly
            history_table_full{ii, prevCcol} = currCvals{currMatches(1)};
        end
    end

  end
end


%% --------------------------------------------------------------------------
%                Fano Encoding with Visualization Function
% --------------------------------------------------------------------------
function dict = Fano_encoding_visual(dict_input)
%{??? ??????
%}
    dict = [];
end
```