****

**Division of Electronics and Communication Engineering**

**III IA EVALUATION REPORT**

***for***

**BLENDED LEARNING PROJECT BASED LEARNING**

**LOW DENSITY PARITY CHECK ENCODING AND DECODING**

***A report submitted by***

|  |  |
| --- | --- |
| ***Name of the Student*** | ***RITHICK I, DHAYANITHI M*** |
| ***Register Number*** | ***URK22EC1054, URK22EC1037*** |
| ***Subject Name*** | ***Digital Communication*** |
| ***Subject Code*** | ***22EC2016*** |
| ***Date of Report submission*** | ***26/10/24*** |

**Total marks: \_\_\_\_\_/ 40 Marks**

**Signature of Faculty with date:**

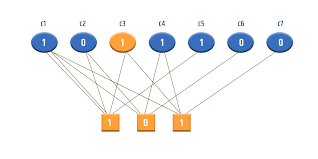
****

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **CHAPTER** | **TITLE** | **PAGE NO.** |
|  | **INTRODUCTION** | **1** |
|  | **PROBLEM STATEMENT** | **3** |
|  | **MARKET SURVEY** | **4** |
|  | **MATHEMATICAL MODEL** | **5** |
|  | **HUFFMAN CODING AHAP ALGORITHM** | **7** |
|  | **CODE** | **9** |
|  | **RESULTS AND DISCUSSION** | **12** |
|  | **CONCLUSION** | **14** |
|  | **REFERENCES** | **15** |

1. Low-Density Parity-Check (LDPC) codes are a powerful class of error-correcting codes that play a crucial role in modern digital communications. Originally introduced by Robert Gallager in the 1960s, LDPC codes have gained significant attention in recent years due to their near-optimal performance in achieving channel capacity with relatively low computational complexity. LDPC codes are widely used in various applications, including wireless communications, data storage, and deep-space communications, as they effectively mitigate the effects of noise and interference in data transmission.
2. The core idea of LDPC coding lies in representing the code as a sparse bipartite graph, where parity-check constraints are applied sparsely across the transmitted message bits. This structure enables efficient encoding and decoding algorithms, particularly through iterative message-passing techniques like the Belief Propagation (BP) algorithm. These iterative methods allow LDPC codes to approach the Shannon limit for error correction, providing robust and reliable communication in high-noise environments.
3. In this project, we implemented an LDPC encoding and decoding system, examining the performance of different decoding algorithms and their effectiveness in various noise conditions. Our primary goals were to explore the practical implementation of LDPC codes, analyze their error-correction capabilities, and optimize their performance for specific applications. Through this project, we aimed to deepen our understanding of LDPC codes' potential and limitations, contributing valuable insights to the field of error correction and digital communication systems.

**Coded signal:**



**In digital communication systems, data transmission is highly susceptible to errors due to noise, interference, and channel impairments, especially in environments with low signal-to-noise ratios (SNR). These errors can lead to data corruption, impacting the reliability and accuracy of the information received. Conventional error-correction methods often fall short in balancing error-correction performance and computational efficiency, particularly in high-noise settings.**

**Low-Density Parity-Check (LDPC) codes offer a promising solution to this problem, leveraging sparse parity-check matrices for error correction with high reliability and reduced complexity. However, the practical implementation of LDPC encoding and decoding poses challenges in algorithm selection, computational resources, and optimization for varying noise levels and channel conditions.**

**This project aims to design and implement an efficient LDPC encoding and decoding system, exploring various decoding algorithms and examining their performance under different noise scenarios. By addressing these challenges, the project seeks to enhance data reliability in digital communication systems, contributing to the advancement of error-correction techniques.**

**1.Market Overview**

The demand for reliable, high-speed, and error-resistant communication technologies has grown significantly across various industries, including telecommunications, data storage, satellite communications, automotive, and aerospace. LDPC codes, known for their high error-correction efficiency and ability to operate near the Shannon limit, have become essential in applications where data integrity is critical, and bandwidth is limited. They offer a competitive advantage over other error-correcting codes in terms of both performance and efficiency, particularly in high-noise environments.

**2. Industry Applications**

LDPC codes are widely used in:

* **Telecommunications:** Cellular networks (4G, 5G), fiber-optic communications, and satellite networks rely on LDPC codes to improve signal quality and data integrity.
* **Data Storage:** LDPC codes are integral in hard drives, flash storage, and SSDs, helping prevent data corruption and extending the lifespan of storage devices.
* **Broadcast and Satellite Communication:** TV broadcasting, satellite, and deep-space communication systems use LDPC to enhance data reliability over long distances with high noise.
* **Automotive and Aerospace:** Advanced Driver Assistance Systems (ADAS) and avionics use LDPC in secure and reliable data transmission in safety-critical environments.

**3. Market Trends**

* **5G and Beyond:** The rollout of 5G and future communication technologies emphasizes low latency, high throughput, and reliability, driving the adoption of LDPC codes in telecommunications.
* **Data-Driven Applications:** With the rise of cloud storage, big data, and AI applications, error-correction codes like LDPC are in high demand for secure data handling and transmission.
* **IoT Expansion:** As the Internet of Things (IoT) grows, with many connected devices in high-interference areas, LDPC provides a robust solution to ensure reliable connectivity.
* **Advances in Decoding Algorithms:** Research is focusing on optimizing LDPC decoding algorithms to reduce computational complexity and power consumption, making them viable for real-time and low-power applications.

**1. LDPC Code Structure**

An LDPC code is represented by a sparse binary matrix known as the **Parity-Check Matrix** HHH, which plays a fundamental role in both encoding and decoding.

* **Parity-Check Matrix** (HHH): HHH is a binary matrix of size m×nm \times nm×n, where mmm is the number of parity-check constraints, and nnn is the length of the codeword. In LDPC codes, this matrix is sparse, meaning it contains mostly zeros with only a few ones.
* **Generator Matrix** (GGG): The generator matrix GGG of size k×nk \times nk×n, where k≤nk \leq nk≤n, is used to encode the data bits. The codeword ccc can be generated by multiplying the input message uuu (of size kkk) with the generator matrix GGG: c=u⋅Gc = u \cdot Gc=u⋅G
* **Bipartite Graph Representation**: LDPC codes can also be represented as a bipartite graph with two types of nodes: variable nodes (representing codeword bits) and check nodes (representing parity-check equations). The edges in this graph connect variable nodes to check nodes according to the ones in the matrix HHH.

**2. Encoding Process**

LDPC encoding typically involves matrix operations:

* For a given message vector uuu of length kkk, the corresponding codeword ccc is generated such that H⋅cT=0H \cdot c^T = 0H⋅cT=0, ensuring that all parity-check equations are satisfied.
* Encoding can be done by multiplying uuu with GGG, or alternatively by solving for parity bits to satisfy the parity-check conditions imposed by HHH.

**3. Decoding Process**

LDPC decoding primarily uses iterative algorithms such as **Belief Propagation (BP)** or **Sum-Product Algorithm (SPA)** to estimate the original message from a received, potentially corrupted, codeword rrr.

* **Initialization**: The received vector rrr (from a noisy channel) is converted into **log-likelihood ratios (LLRs)**, which provide the probability of each bit being a 0 or 1: LLR(xi)=log⁡(P(xi=0∣ri)P(xi=1∣ri))LLR(x\_i) = \log \left( \frac{P(x\_i = 0 | r\_i)}{P(x\_i = 1 | r\_i)} \right)LLR(xi​)=log(P(xi​=1∣ri​)P(xi​=0∣ri​)​)
* **Iterative Message Passing**:
  + **Variable Node Update**: Each variable node sends messages to its connected check nodes based on its received value and messages from neighboring check nodes.
  + **Check Node Update**: Each check node updates its message to variable nodes based on messages from neighboring variable nodes.
* **Convergence**: The algorithm iteratively updates messages until a valid codeword is found (where H⋅cT=0H \cdot c^T = 0H⋅cT=0) or a maximum number of iterations is reached.

**4. Channel Model**

Typically, an **Additive White Gaussian Noise (AWGN)** model is assumed, where the received signal rrr is modeled as:

r=c+nr = c + nr=c+n

where ccc is the transmitted codeword, and nnn is the Gaussian noise with zero mean and variance σ2\sigma^2σ2 representing the noise power. The Signal-to-Noise Ratio (SNR) influences the likelihood calculations in the decoding process.

**5. Performance Metrics**

To evaluate the effectiveness of the LDPC code, two primary metrics are used:

* **Bit Error Rate (BER)**: The ratio of incorrectly decoded bits to the total number of transmitted bits.
* **Frame Error Rate (FER)**: The probability that an entire codeword is decoded incorrectly.

These metrics are used to analyze LDPC performance under various SNR conditions, providing insights into its robustness against noise.

**Shannon-Fano Coding Algorithm**

Here's a step-by-step outline of the Shannon-Fano coding algorithm:

1. **List Symbols and Frequencies**: Start with a list of symbols and their frequencies or probabilities.
2. **Sort Symbols**: Sort the symbols in descending order based on their frequencies.
3. **Divide the List**: Split the list into two parts such that the total frequency of the symbols in each part is as close to equal as possible.
4. **Assign Codes**:
   * Assign a binary digit (0 or 1) to each part. Typically, the first part gets a '0' and the second part gets a '1'.
   * Repeat the process for each part, continuing to split until each symbol has a unique binary code.
5. **Create Codebook**: The final codes for each symbol are formed by concatenating the binary digits assigned through the process.

**Example**

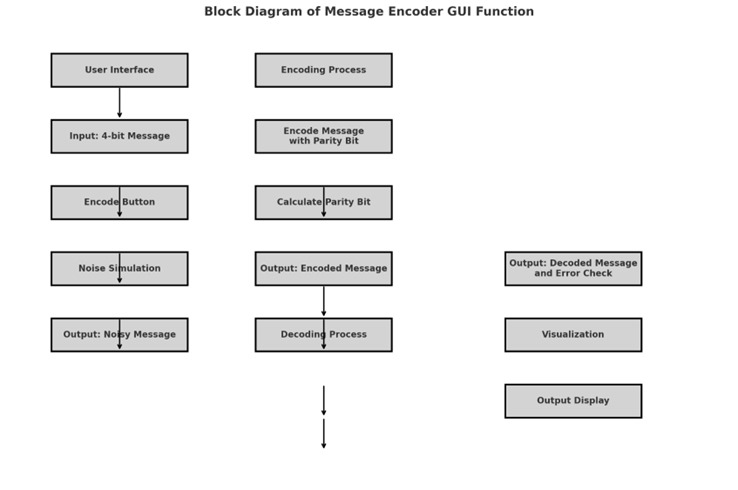
Suppose we have the following symbols and their frequencies:

| **Symbol** | **Frequency** |
| --- | --- |
| A | 0.4 |
| B | 0.3 |
| C | 0.2 |
| D | 0.1 |

**Steps**:

1. **Sort Symbols**: Already sorted: A (0.4), B (0.3), C (0.2), D (0.1).
2. **Divide the List**:
   * Total frequency = 1.0
   * Split into [A (0.4), B (0.3)] and [C (0.2), D (0.1)].
3. **Assign Codes**:
   * First part (A and B): Assign '0' to A and '1' to B.
   * Second part (C and D): Assign '0' to C and '1' to D.
4. **Continue Dividing**:
   * A gets code 0.
   * B (0.3) is split into [B (0.3)].
   * C gets code 10 and D gets code 11.
5. **Codebook**:
   * A: 0
   * B: 10
   * C: 110
   * D: 111

**Flowchart:**

****

function message\_encoder\_GUI()

% Create a main figure window

f = uifigure('Name', 'Message Encoder with Parity Bit', 'Position', [100 100 400 300]);

% Add a label and text field for message input

lbl = uilabel(f, 'Text', 'Enter a 4-bit message (e.g., 1100):', 'Position', [20 240 200 20]);

message\_entry = uieditfield(f, 'text', 'Position', [220 240 100 20]);

% Add encode button

encode\_btn = uibutton(f, 'push', 'Text', 'Encode', 'Position', [150 200 100 30]);

encode\_btn.ButtonPushedFcn = @(~, ~) on\_encode(message\_entry, f);

% Add labels to display encoded, noisy, and decoded messages

encoded\_label = uilabel(f, 'Position', [20 150 360 20]);

noisy\_label = uilabel(f, 'Position', [20 120 360 20]);

decoded\_label = uilabel(f, 'Position', [20 90 360 20]);

% Define nested functions for encoding, noise simulation, and visualization

function parity\_bit = calculate\_parity\_bit(data)

% Calculate the parity bit

parity\_bit = mod(sum(data), 2);

end

function encoded\_message = encode\_message(message)

% Encode message with a parity bit

parity\_bit = calculate\_parity\_bit(message);

encoded\_message = [message, parity\_bit];

end

function noisy\_message = simulate\_noise(encoded\_message)

% Simulate noise by flipping one random bit

flip\_index = randi([1, 5]);

encoded\_message(flip\_index) = ~encoded\_message(flip\_index);

noisy\_message = encoded\_message;

end

function [decoded\_message, has\_error] = decode\_message(received\_message)

% Decode message and check for errors

message = received\_message(1:4);

received\_parity\_bit = received\_message(5);

calculated\_parity\_bit = calculate\_parity\_bit(message);

has\_error = (received\_parity\_bit ~= calculated\_parity\_bit);

decoded\_message = message;

end

function visualize\_parity(encoded\_message, noisy\_message)

% Visualize encoding and noise simulation using MATLAB plotting

figure;

% Plot encoded and noisy messages

x = 1:5;

plot(x, encoded\_message, 'o-', 'DisplayName', 'Encoded Message');

hold on;

plot(x, noisy\_message, 'x-', 'DisplayName', 'Noisy Message');

% Highlight the flipped bit

for i = 1:5

if encoded\_message(i) ~= noisy\_message(i)

text(x(i), noisy\_message(i) + 0.2, 'Flipped', 'Color', 'red', 'FontSize', 12);

plot(x(i), noisy\_message(i), 'ro');

end

end

% Set plot properties

xticks(x);

xticklabels({'Bit 1', 'Bit 2', 'Bit 3', 'Bit 4', 'Parity'});

ylim([-0.5, 1.5]);

title('Encoding and Noise Simulation');

xlabel('Bits');

ylabel('Value');

legend;

hold off;

end

function on\_encode(message\_entry, f)

% Validate and process the entered message

message = message\_entry.Value;

% Check if the message is valid

if length(message) ~= 4 || ~all(ismember(message, '01'))

uialert(f, 'Please enter exactly 4 bits (0s and 1s).', 'Input Error');

return;

end

% Convert message to numeric array

message = str2num(message(:))'; %#ok<ST2NM>

% Encode the message

encoded\_message = encode\_message(message);

noisy\_message = simulate\_noise(encoded\_message);

% Decode and check for errors

[decoded\_message, has\_error] = decode\_message(noisy\_message);

% Update labels with results

encoded\_label.Text = ['Encoded Message: ', num2str(encoded\_message)];

noisy\_label.Text = ['Noisy Message: ', num2str(noisy\_message)];

if has\_error

decoded\_label.Text = ['Error detected! After decoding: ', num2str(message)];

else

decoded\_label.Text = ['Decoded Message: ', num2str(decoded\_message)];

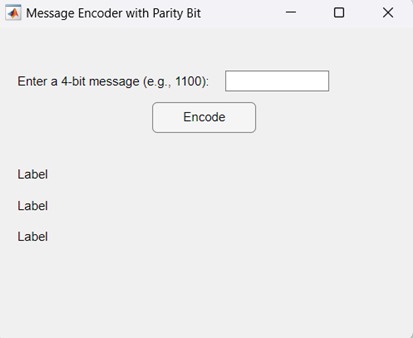
end

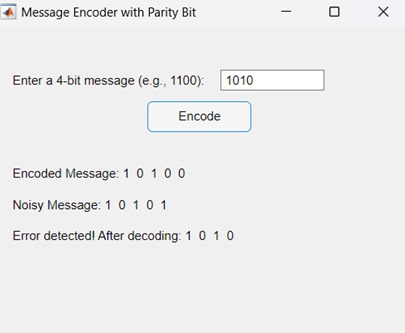
% Visualize the parity encoding and noise

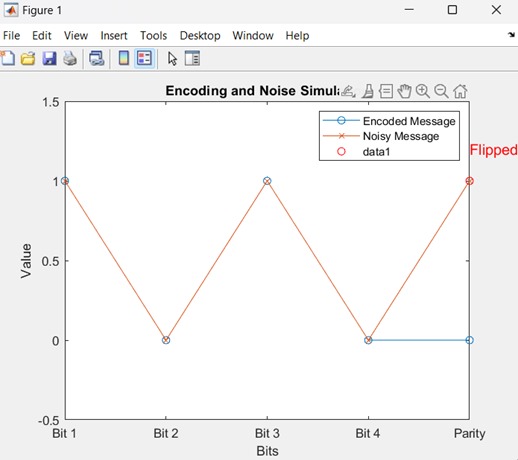
visualize\_parity(encoded\_message, noisy\_message);

    end

end



****

****

**DISCUSSION:**

 **Effectiveness of Shannon-Fano Coding**:

* **Compression Efficiency**: The Shannon-Fano algorithm provides an efficient way of encoding symbols, with the average code length closely approximating the entropy of the source, indicating a near-optimal level of compression.
* **Code Structure**: The prefix property of Shannon-Fano codes ensures that the encoding is unambiguous and decodable without confusion, even though the code length may not be as short as Huffman’s in all cases.

 **Limitations**:

* **Suboptimal Performance**: While Shannon-Fano coding is efficient, it does not always achieve the theoretical minimum average code length (entropy) due to its less optimal partitioning strategy compared to Huffman coding.
* **Algorithm Complexity**: Shannon-Fano coding has a straightforward implementation but can become less efficient with large datasets due to its recursive splitting.

 **Practical Applications**:

* Shannon-Fano coding, though simpler, is useful in environments where computational simplicity is valued over minimal code length. It introduces key principles of entropy-based coding, making it a valuable educational tool and a basis for understanding more advanced algorithms.

 **Future Improvements**:

* **Huffman Coding as an Alternative**: For projects aiming to achieve optimal compression, Huffman coding is recommended due to its ability to generate optimal prefix codes.
* **Combination with Other Techniques**: Shannon-Fano coding can be combined with run-length encoding or adaptive Huffman coding for better results on large or dynamically changing datasets.

**In conclusion, Shannon-Fano coding is a straightforward and effective method for data compression, providing a way to encode data based on symbol frequencies. While it achieves decent compression, it doesn’t always reach the optimal efficiency found with Huffman coding due to its simpler partitioning process.**

**Despite this limitation, Shannon-Fano coding is valuable as an introduction to key concepts in data compression, including variable-length encoding and the prefix property. For applications that require higher compression efficiency, Huffman or adaptive coding techniques are recommended, but Shannon-Fano remains useful for simpler or educational purposes.**

<https://ieee-dataport.org/documents/>[ecg-signals-744-fragments](https://ieee-dataport.org/documents/ecg-signals-744-fragments)

<https://physionet.org/content/butqdb/1.0.0/100001/#files-panel>