PROJECT BASED LEARNING REPORT

on

“**WRITE A PROGRAM FOR HAMMING CODE USING MATLAB**”

Submitted in the partial fulfillment of the requirements

for the Project based learning (PBL) INFORMATION THEORY AND CODING

in

Electronics & Communication Engineering

By

PRN NAME

2214110414 AASTHA GUPTA

2214110415 TRISHA HALI

2214110450 MANIKA JAIN

Under the guidance of Course In-charge

Prof. Dr. TANUJA S. DHOPE



Department of Electronics & Communication Engineering

Bharati Vidyapeeth (Deemed to be University)

College of Engineering,

Pune – 4110043

Academic Year: 2024-25

**CERTIFICATE**

Certified that the Project Based Learning report entitled, “**WRITE A PROGRAM FOR HAMMING CODE USING MATLAB**” is work done by

2214110414 AASTHA GUPTA

2214110415 TRISHA HALI

2214110450 MANIKA JAIN

in partial fulfillment of the requirements for the award of credits for Project Based Learning (PBL) in Information Theory And Coding of Bachelor of Technology Semester V, in Department of Electronics and Communication Engineering.

Date:

Prof. Dr. Tanuja .S Dhope Dr. Arundhati A. Shinde

Course In-charge Professor & Head

**INDEX**

|  |  |  |
| --- | --- | --- |
| SR.NO. | CONTENT | PAGE NO. |
| 1 | Chapter 1   * Why do we need hamming code * Solution of hamming code | 1-3 |
| 2 | Chapter-2   * Introduction | 4-6 |
| 3 | Chapter-3   * Hamming code structure | 8-9 |
| 4 | Chapter-4   * Code | 10-12 |
| 5 | Chapter-5   * Output | 13 |
| 6 | Chapter-6   * Conclusion * Outcome * Reference * Github link | 15 |

**CHAPTER-1**

**Why Do We Need “Hamming Code”**

Hamming code is used for **error detection and correction** in data transmission and storage. When data is transmitted across a medium (like a network or saved to a disk), errors may occur due to interference, noise, or other factors. These errors can corrupt the data, leading to incorrect information being received or read.

Here’s why Hamming code is essential:

**1. Error Detection**

Hamming code helps in detecting single-bit and two-bit errors that may occur during transmission. When errors are detected, corrective actions can be taken.

**2. Error Correction**

The primary purpose of Hamming code is to **correct single-bit errors**. It uses a mathematical method to locate and fix a single-bit error, which makes it different from simpler error-detecting codes like parity bits. The capability to correct errors makes Hamming code useful in critical applications where reliability is key.

**3. Efficient Redundancy**

Hamming code uses **redundant bits** efficiently by placing them in positions that allow both error detection and correction. These bits are calculated based on the data being transmitted, and they add minimal overhead to the data stream.

**4. Cost-Effective in Memory Systems**

In memory systems like RAM, where data integrity is crucial, Hamming code can be implemented with relatively low cost, while still offering a robust method to correct errors. It ensures the data retrieved is the same as what was stored, protecting against single-bit memory failures.

**5. Reliable Data Transmission**

In telecommunications, networking, and digital storage systems, using Hamming code improves the reliability of data transmission. It ensures that even in the presence of noise or interference, small errors in the data can be fixed automatically, thus maintaining data integrity.

**Solution Of “Hamming Code”**

Hamming code is a method used for error detection and correction in digital communication systems and data storage. Its primary purpose is to ensure data integrity during transmission or storage, where errors can occur due to noise or interference. When data is transmitted, the possibility of single-bit errors—where one bit in the data sequence is incorrectly flipped—can corrupt the message. Hamming code addresses this issue by adding redundant parity bits to the original data, enabling not just the detection of errors but also their correction.

To generate a Hamming code, the first step is determining the number of parity bits needed based on the number of data bits. These parity bits are placed in specific positions within the data sequence, corresponding to powers of two (like positions 1, 2, 4, etc.). Each parity bit checks a subset of the data bits, ensuring that the total number of 1s in those positions is even (for even parity systems). The parity bits are carefully chosen so that if a single-bit error occurs, the erroneous bit can be identified by recalculating the parity values and analyzing the pattern of discrepancies.

For example, consider a 4-bit data sequence, such as 1011. To protect this data using Hamming code, three parity bits are added, resulting in a 7-bit codeword. The final Hamming code, in this case, might look like 0110011, where the parity bits are included at designated positions. If an error occurs during transmission, like a bit flip in one position, the receiver recalculates the parity bits. The results of these recalculations help pinpoint the exact position of the error, which is then corrected by flipping the identified bit.

This process allows Hamming code to not only detect errors but also correct them, making it a powerful tool in ensuring reliable data transmission and storage. Its efficiency in adding minimal redundancy while enabling error correction makes it widely used in memory systems, digital communications, and networking

**CHAPTER-2**

**INTRODUCTION**

Hamming code is a method used for error detection and correction in digital communication systems and data storage. Its primary purpose is to ensure data integrity during transmission or storage, where errors can occur due to noise or interference. When data is transmitted, the possibility of single-bit errors—where one bit in the data sequence is incorrectly flipped—can corrupt the message. Hamming code addresses this issue by adding redundant parity bits to the original data, enabling not just the detection of errors but also their correction.

To generate a Hamming code, the first step is determining the number of parity bits needed based on the number of data bits. These parity bits are placed in specific positions within the data sequence, corresponding to powers of two (like positions 1, 2, 4, etc.). Each parity bit checks a subset of the data bits, ensuring that the total number of 1s in those positions is even (for even parity systems). The parity bits are carefully chosen so that if a single-bit error occurs, the erroneous bit can be identified by recalculating the parity values and analyzing the pattern of discrepancies.

For example, consider a 4-bit data sequence, such as 1011. To protect this data using Hamming code, three parity bits are added, resulting in a 7-bit codeword. The final Hamming code, in this case, might look like 0110011, where the parity bits are included at designated positions. If an error occurs during transmission, like a bit flip in one position, the receiver recalculates the parity bits. The results of these recalculations help pinpoint the exact position of the error, which is then corrected by flipping the identified bit.

This process allows Hamming code to not only detect errors but also correct them, making it a powerful tool in ensuring reliable data transmission and storage. Its efficiency in adding minimal redundancy while enabling error correction makes it widely used in memory systems, digital communications, and networking.

**CHAPTER-3**

**Hamming Code Structure**

The structure for implementing Hamming code involves several key steps, both during the encoding (for error detection and correction) and decoding (for error detection and correction) processes. Below is a breakdown of the steps to structure a Hamming code implementation:

**1. Identify the Number of Redundant Bits (Parity Bits)**

The first step is to determine how many redundant bits (parity bits) are needed for a given set of data bits. The formula for calculating the number of redundant bits is:

2^r ≥ m + r + 1

Where:

* r is the number of redundant (parity) bits.
* m is the number of data bits.
* m+r is the total number of bits in the encoded message.

The number of redundant bits depends on the size of the data being transmitted. This step ensures that enough redundant bits are inserted to cover error detection and correction.

**2. Position the Redundant Bits**

After determining the number of redundant bits, they are inserted into the binary data sequence at positions that are powers of 2 (i.e., positions 1, 2, 4, 8, etc.). These positions are reserved for the parity bits:

* Position 1: For 2^0
* Position 2: For 2^1
* Position 4: For 2^2
* Position 8: For 2^3, and so on.

Data bits are then placed in the remaining positions of the binary sequence.

**3. Calculate the Values of the Parity Bits**

Each redundant (parity) bit checks the values of a specific set of data bits. For instance:

* Parity bit at position 1 checks positions 1, 3, 5, 7, 9, etc.
* Parity bit at position 2 checks positions 2, 3, 6, 7, 10, 11, etc.
* Parity bit at position 4 checks positions 4, 5, 6, 7, 12, 13, etc.

To calculate the parity bit's value:

* If the number of 1s in the positions it checks is even, the parity bit is set to 0.
* If the number of 1s is odd, the parity bit is set to 1.

These parity bits ensure that the sequence satisfies the even parity condition, which allows error detection later.

**4. Generate the Hamming Code**

Once the data bits and parity bits have been placed in their respective positions, and the values of the parity bits are calculated, the full encoded message (data + parity bits) is ready for transmission. This encoded message is called the Hamming code.

**5. Transmit the Hamming Code**

The encoded message is transmitted through a communication channel to the receiver. During transmission, errors might occur (i.e., one or more bits might flip from 0 to 1 or vice versa).

**6. Error Detection and Correction at the Receiver**

Once the Hamming code is received, the receiver needs to check for errors:

* The receiver recalculates the values of the parity bits based on the received data.
* The parity bits are then compared with the values of the redundant bits in the received message.
* If all parity checks pass (i.e., the parity bits match their expected values), no error has occurred.
* If one or more parity checks fail, the receiver sums the positions of the erroneous parity bits to determine the position of the error.

For example:

* If parity bits at positions 1 and 4 fail, the error is at position 1+4=5. The bit at position 5 is then flipped to correct the error.

**7. Correct the Error**

After identifying the position of the error, the erroneous bit can be flipped to correct the message. This process allows the receiver to recover the correct data even if an error occurred during transmission.

**8. Extract the Original Data**

Once error correction is complete, the receiver can remove the parity bits and extract the original data. The corrected data is now ready to be processed or used by the receiving system.

**Example of Hamming Code Structure:**

Assume we are sending 4 data bits (e.g., 1011). The steps would look like this:

1. Calculate Redundant Bits:  
   We need to find r such that 2^r≥4+r+1.  
   For 4 data bits, 3 redundant bits are required (positions 1, 2, and 4).
2. Position the Parity Bits:  
   Insert the data bits and parity bits:  
   \_ \_1\_0 1 1 (parity bits in positions 1, 2, and 4).
3. Calculate Parity Bits:  
   Calculate parity for each group:
   * Parity bit 1 checks positions 1, 3, 5, 7.
   * Parity bit 2 checks positions 2, 3, 6, 7.
   * Parity bit 4 checks positions 4, 5, 6, 7.

Set parity bits according to the even parity rule.

1. Generate Hamming Code:  
   The final encoded message might look like: 1110011 (with parity bits calculated).
2. Transmit the Code:  
   Transmit 1110011.
3. Error Detection:  
   On receiving, if there is an error (e.g., 1111011 received), parity checks will identify the error.
4. Error Correction:  
   Identify and correct the erroneous bit (e.g., position 5), flipping the bit at position 5 back to restore the original code.
5. Extract Data:  
   After correction, extract the original data bits (e.g., 1011).

This structured approach ensures that Hamming code can efficiently detect and correct errors in transmitted data.

**CHAPTER- 4**

**CODE**

% MATLAB Program for Hamming Code (7,4)

% Clear all variables and close figures

clear all;

close all;

clc;

% Input: 4-bit message

msg = input('Enter a 4-bit binary message (as a row vector): ');

% Check if the input is valid (4-bit binary vector)

if length(msg) ~= 4 || ~all(ismember(msg, [0 1]))

error('Input must be a 4-bit binary vector.');

end

% Initialize Hamming Code (7,4) Parity positions

% Bits: [p1 p2 d1 p3 d2 d3 d4] (p = parity, d = data)

p1 = 0; p2 = 0; p3 = 0; % Initialize parity bits

% Assign data bits

d1 = msg(1);

d2 = msg(2);

d3 = msg(3);

d4 = msg(4);

% Calculate parity bits based on data bits

p1 = xor(xor(d1, d2), d4); % Parity for positions 1, 3, 5, 7

p2 = xor(xor(d1, d3), d4); % Parity for positions 2, 3, 6, 7

p3 = xor(xor(d2, d3), d4); % Parity for positions 4, 5, 6, 7

% Form the Hamming codeword

hamming\_code = [p1 p2 d1 p3 d2 d3 d4];

% Display the generated Hamming code

disp('Generated Hamming Code (7,4): ');

disp(hamming\_code);

% Simulate error by flipping one bit (optional)

error\_sim = input('Do you want to simulate a single-bit error? (1 for yes, 0 for no): ');

if error\_sim

bit\_to\_flip = input('Enter the bit position (1-7) to flip: ');

if bit\_to\_flip >= 1 && bit\_to\_flip <= 7

hamming\_code(bit\_to\_flip) = ~hamming\_code(bit\_to\_flip); % Flip the bit

disp('Hamming Code with simulated error: ');

disp(hamming\_code);

else

error('Invalid bit position. Enter a number between 1 and 7.');

end

end

% Error detection and correction

% Recalculate parity bits for received code

r1 = xor(xor(hamming\_code(1), hamming\_code(3)),

xor(hamming\_code(5), hamming\_code(7)));

r2 = xor(xor(hamming\_code(2), hamming\_code(3)),

xor(hamming\_code(6), hamming\_code(7)));

r3 = xor(xor(hamming\_code(4), hamming\_code(5)),

xor(hamming\_code(6), hamming\_code(7)));

% Determine the position of the error (if any)

error position = r1 + 2\*r2 + 4\*r3;

if error position == 0

disp('No errors detected.');

else

disp(['Error detected at bit position: ' num2str(error\_position)]);

% Correct the error

hamming\_code(error\_position) = ~hamming\_code(error\_position);

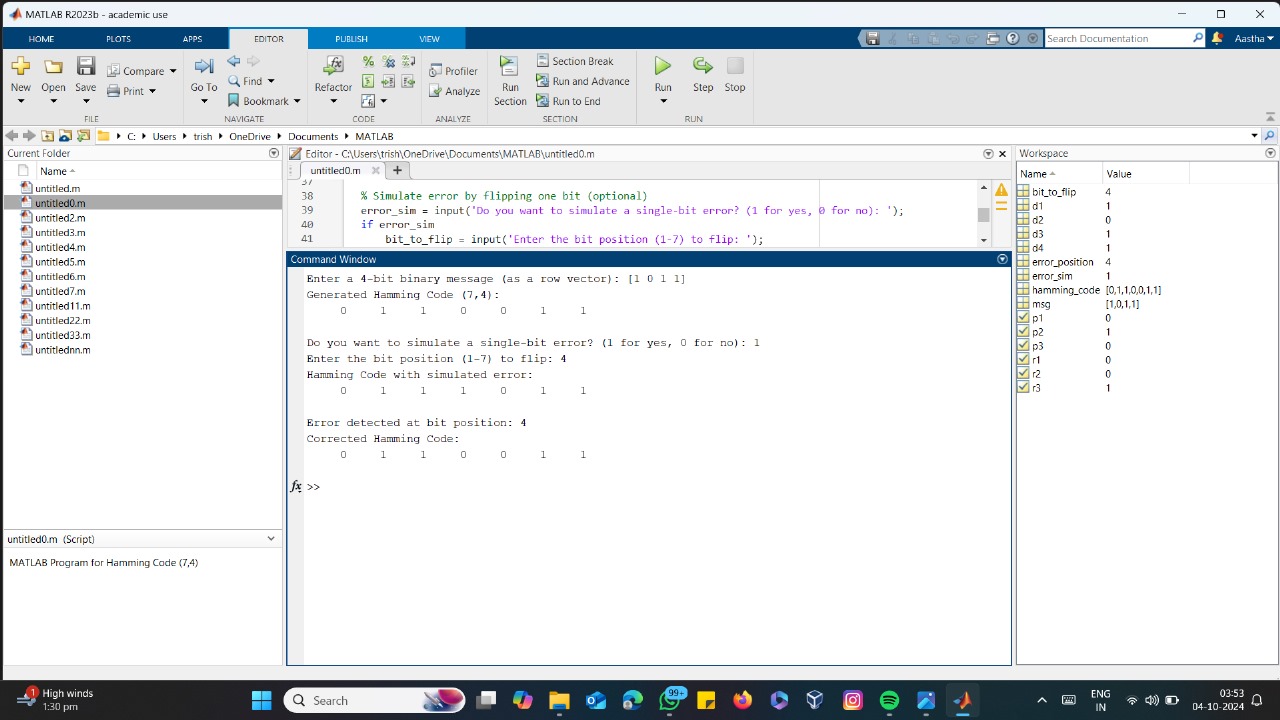
disp('Corrected Hamming Code: ');

disp(hamming\_code);

end

**CHAPTER-5**

**OUTPUT**

****

**CHAPTER-6**

**CONCLUSION**

Writing a Python program for Hamming Code offers a detailed exploration into how error-detection and correction mechanisms work in digital communication systems. The process begins with understanding the basic principle behind Hamming Code, which is to detect and correct single-bit errors in a transmitted message using a minimal number of parity bits. In the program, parity bits are placed at positions that are powers of two (e.g., positions 1, 2, 4, 8, etc.), and their values are determined by checking specific groups of bits in the binary message. The goal is to set these parity bits in such a way that if a single bit error occurs during transmission, the error can be detected and located precisely.

The encoding part of the program demonstrates how the input message is first broken into data bits, and then these data bits are interspersed with the parity bits. The parity bits are calculated by using XOR operations on certain positions in the message, ensuring that each parity bit checks for a specific subset of data bits. This setup creates a codeword, which is the transmitted data that includes both the original message and the error-checking bits.

The decoding process is equally important and is handled by recalculating the parity bits for the received message. If the recalculated parity bits match the transmitted ones, the message is considered error-free. If they don’t match, an error has occurred, and the error syndrome is calculated. The error syndrome is a binary number that directly indicates the position of the erroneous bit. For example, if the error syndrome is 5, this means that the 5th bit in the received message is incorrect. The program then flips this erroneous bit, effectively correcting the single-bit error.

Through this program, the efficiency of Hamming Code is demonstrated in terms of both error detection and correction. It shows how Hamming Code can correct single-bit errors and detect two-bit errors, which makes it particularly suitable for applications where minor errors are common but multiple errors are rare. By using a minimal number of parity bits, Hamming Code ensures that redundancy is kept low while still providing reliable error correction. This balance between efficiency and error protection is one of the main reasons Hamming Code is still used in various real-world applications, such as memory systems (e.g., ECC RAM), satellite communication, and digital storage devices.

The Python implementation also highlights the algorithm’s scalability. As the length of the data increases, the number of parity bits grows logarithmically, meaning that even for relatively large messages, the overhead introduced by Hamming Code remains small. This makes it a practical solution in scenarios where bandwidth or storage space is limited but data integrity is critical. Moreover, the simplicity of the XOR operations and binary manipulations ensures that the algorithm is computationally lightweight, making it suitable for real-time applications such as wireless communication systems, where quick error correction is necessary to maintain data flow without retransmission.

In conclusion, writing a Python program for Hamming Code not only reinforces key concepts of error correction but also demonstrates the algorithm's real-world applicability and efficiency. It showcases how a well-designed code can provide robust protection against single-bit errors with minimal redundancy, making it a vital tool in ensuring data integrity in various digital systems. Through this exercise, one gains a deeper understanding of both the theoretical foundations and the practical implementation of error-correcting codes, emphasizing their importance in maintaining the reliability of digital communication and storage.

**OUTCOME**

The outcome of writing a Hamming code program in Python allows us to encode data with additional parity bits for error detection and correction. When the program is executed, it first determines the number of parity bits required based on the length of the data. These parity bits are then placed at specific positions (those that are powers of 2) in the Hamming code, while the original data bits are placed in the remaining positions. The program calculates the values of the parity bits by examining specific subsets of the data bits, ensuring that the number of 1s in each subset is even (in the case of even parity).

**COURSE OUTCOME(CO)**

CO4 Is used in this.

To make student aware of various error control coding algorithm

**REFERENCE**

**www.wikipedia.in**

www.geeksforgeeks.gfg.in

BY P.S.V Rao and R.Chithra

BY John G. Proakis and Masoud Salehi

**GITHUB LINK**