

Application of Number Theory in Error Detection Systems (Hamming Codes)

Sakti Bimasena - 13523053^{1,2}

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia

13523053@mahasiswa.itb.ac.id, sbimasena@gmail.com

Abstract—Accurate data transmission becomes more important for communication and information systems in the modern age. Hamming code is one of the most well-known error detection and correction techniques in digital data transmission. In this paper, number theory is discussed in error detection systems, with special emphasis on the concept of parity bits and the basic principles of Hamming Codes. This paper also discusses how number theory, such as modulo arithmetic and congruence properties, function in the implementation of error detection systems. This paper demonstrates the ability of Hamming Code to find and correct single-bit errors and its limitations to handle multi-bit errors through Python program implementation and a number of tests. The results of the analysis show that number theory plays an important role in the development and effectiveness of contemporary error detection systems.

Keywords—Hamming codes, error detection, number theory, parity bits, data transmission

I. INTRODUCTION

In this digital age, reliable data transmission is very important for communication and information systems. However, data transmission processes encounter many obstacles that result in an error in the data. Encountering these obstacles can be very dangerous, especially in high-risk applications such as industrial control systems, satellite communications, and banking systems [1].

To solve this problem, many error detection and error correction methods have been developed. One of the most famous ones are Hamming codes that can detect and correct single-bit errors in data. This method was created by Richard W. Hamming in 1950 and uses a mathematical approach that is linked to number theory such as binary properties and arithmetic modulo operations [1],[2].

Concepts like binary representation, modulo operations, and prime number properties are very important to calculating parity bits that are used to detect and correct error. Number theory plays a big part in the development of hamming code algorithm dan code structure. This relationship shows how basic number theory can be used to solve real problems in information technology.

The purpose of this paper is to see how number theory is used in error detection systems, with focus on hamming codes. The paper will discuss the basic theory of hamming codes, how the program will be implemented, and analysis on experiment data. It is hoped that this discussion can

bring better understanding as to the role of number theory in contemporary technology.

II. THEORETICAL FOUNDATION

A. Hamming Codes

Hamming codes is an error correction technique that is used to find and correct single-bit errors in digital data. Richard W. Hamming developed this method in 1950 to increase the reliability of communication systems. The working principle of hamming codes involve the addition of parity bits into the original data. Parity bits are used to satisfy certain conditions based on the combination of bits in the data, and they help find the position of the error if an error were to happen in transmission.

A.1. Parity Bits

The original data that consisted of m bits, is expanded with r parity bits as so the total length of the data would be $n = m + r$. The number r can be calculated by using the following formula:

$$2^r \geq m + r + 1$$

This equation makes sure that there are enough parity bits to detect and correct errors in data. For example, if the number of data bits is 4 (d_1, d_2, d_3, d_4), by using the above equation, you would need at least 3 parity bits ($2^3 \geq 4 + 3 + 1$) to be able to sufficiently detect and correct single-bit errors.

Before continuing, first know that there are two types of parity bits, even parity bits and odd parity bits. A parity bit will be added to the original data to make sure the total amount of ones in the data is even or odd. For both types, the value of the parity bits is dependent on the number of ones that appear in a certain number of bits. In the case of even parity bits, if that amount is odd, the value of the parity bit is one, as so the total amount of ones that appear in that certain group of bits is even. If the total amount of ones is already even, the value of the parity bit is zero. For odd parity bits, it is the exact opposite, if the number of ones is even, the parity bit is set to one as to make sure the total number of ones is odd. For the rest of this paper, the type of parity bits that will be used is even parity bits.

A.2. Creating Hamming codes

The creation of hamming code starts by determining the

value of each parity bit. To do that, first keep in mind the position of every bit, starting from one, in binary form, like 1, 10, 11, and so on. Then, the positions are classified into two main groups: parity bits and data bits. Bit positions that are a power of two, are classified as parity bits. All the other bits are classified as data bits, these bits are the ones that will carry the actual information that will be transmitted.

According to its position in binary representation, every data bit contributes to a certain amount of parity bits. For example, the parity bit in the position of one (p_1) covers all bit positions that has a one in the least significant position in the binary representation, like 1, 3, 5, 7, etc. the parity bit in the position of two (p_2) covers all bit positions that has a one as its second digit (2, 3, 6, 7, etc.). Then the parity bit in the position of 4 (p_4) covers all bit positions that has a one as its third digit (4-7, 12-15, etc.). A parity bit mathematically includes all bits where when a bitwise AND operation is performed between the parity bit position and the associated bit position, the value is non-zero. The parity bits value is then set depending on the type

Position	R8	R4	R2	R1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1

R1 -> 1,3,5,7,9,11
R2 -> 2,3,6,7,10,11
R3 -> 4,5,6,7
R4 -> 8,9,10,11

of parity bits that is used.

Fig. 2.1 Parity bits coverage

Source: <https://www.geeksforgeeks.org/hamming-code-in-computer-network>

To put this into perspective, if the data that wants to be transmitted is 1101010, to make the hamming code, start by placing the parity bits in the specified position. The final position would look like this:

$$(p_1, p_2, 1, p_4, 1, 0, 1, p_8, 0, 1, 0)$$

p_1 covers all positions that have 1 in its least significant position (1, 3, 5, 7, 9, 11). With the first position being the far right, there are 3 total ones, since we use even parity bits, p_1 is set to one. Then move on to the second parity bit

and so on. It gets much harder to count manually each occurrence of ones the larger the data gets. This can be replaced by using an algorithm that uses XOR for each digit. For example, for p_2 , we can calculate its value by taking the bits in the position 3, 6, 7, 10, and 11 and using XOR each bit like this:

$$0 \oplus 0 \oplus 1 \oplus 1 \oplus 1 = 1$$

A.3. Error Detection and Correction

By using the parity bits that was calculated in the encoding process, Hamming codes can detect errors in transmission. Each parity bit value will be recalculated when the data has reached its destination. The calculation will be done the same as when the data was encoded, namely by using binary rules to select bits at certain positions. This parity bit value will then be compared to the parity bit value that was sent with the data.

Hamming code uses something called error syndrome, which is the XOR result between the received parity bit value and the recalculated parity bit value, to determine the position of the incorrect parity bit. Data is considered error-free if all recalculated parity bit values match the received parity bit values.

Each digit in this error syndrome corresponds to a comparison result for a particular parity bit. If the error syndrome is equal to zero (0000), there are no errors found. On the contrary, a non-zero value shows the position of the incorrect bit. For example, if the error syndrome has a value of 0011 (3), this means that the bit in the third position is incorrect. The bit can be corrected by flipping its value (from 1 to 0, or 0 to 1) after finding its position.

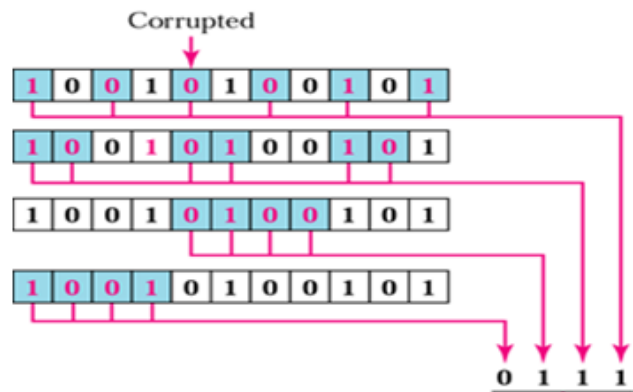


Fig. 2.2 Error detection visualization

Source: <https://www.expertsmind.com/questions/show-the-error-correction-by-hamming-code-30191826.aspx>

B. Number Theory in Hamming Code

Number theory, a branch of pure mathematics that discusses the properties of whole numbers, has an important role in the development and implementation of hamming code. Hamming code takes advantage of principles like modulo arithmetic, congruency, and whole number properties to detect and correct errors in data transmission.

B.1. Modulo Arithmetic

In hamming code, parity bit calculation depends on modulo arithmetic. This operation is used to determine the even odd parity value from a subset of data. For example, if the total number of ones in a certain number of bits is even, the parity value will be set to zero, and if the number of ones is odd, the parity value will be one. In this case, the modulo 2 operation is used to ensure that the result of the calculation always lies in the domain $\{0, 1\}$.

B.2. Congruency

The bits checked by each parity bit are grouped based on the concept of congruence. In binary bit position representation, modulo properties is used to determine which bits are covered by a parity bit. For example, a binary position with a particular value can be used to determine whether that bit is controlled by a particular parity bit, which corresponds to a bitwise AND at that position. This supports mathematical detection of fault positions and enables effective data management.

B.3 Whole Number Properties

Hamming codes also rely on the properties of integer division, such as the ability to determine the remainder of a division. This operation is important for calculating the error syndrome, which is the difference between the received parity value and the recalculated parity value. The error syndrome is represented in binary form and indicates the position of the error in the data. This binary representation allows the system to directly correct the erroneous bits.

III. IMPLEMENTATION PROGRAM

The implementation of Hamming code using Python is described in detail in this section. This program is intended to handle the data encoding process, which includes adding parity bits, as well as to find errors and repair corrupted data. The main functions and their working mechanisms are discussed in this explanation.

The number of parity bits (r) required to protect the data is the first step in encoding. Parity bits are additional bits used to detect and correct errors. The length of the original data (m) is used to determine the number of parity bits using the following formula:

$$2^r \geq m + r + 1$$


This formula makes sure that the parity bits are sufficient in covering all data bits. In this python implementation, `calculate_parity_positions` function calculates r iteratively. For example, if the original data contains 7 bits, then r is calculated to be 4 because $2^4 \geq 7 + 4 + 1$. So the total length of the data transmitted is $7+4 = 11$ bits.



```
1 def calculate_parity_positions(data_length):
2     r = 0
3     while (2**r) < (data_length + r + 1):
4         r += 1
5     return r
```

Fig. 3.1 `calculate_parity_positions` function
Source: Author

After calculating the amount of parity bits, the next step is inserting the parity bits into their position (powers of 2). In the beginning, the parity bits hold the value 0 as a placeholder. On the other hand, the data bits are inserted according to their original sequence. `insert_parity_bits` function does this task like this:



```
1 def insert_parity_bits(data):
2     r = calculate_parity_positions(len(data))
3     total_length = len(data) + r
4     encoded = []
5     j = 0
6     for i in range(1, total_length + 1):
7         if (i & (i - 1)) == 0: # Check if i is a power of 2
8             encoded.append(0) # Placeholder for parity bit
9         else:
10            encoded.append(int(data[j]))
11            j += 1
12    return encoded
```

Fig. 3.2 `insert_parity_bits` function
Source: Author

For example, if the original data is 1101010, the function returns `[0,0,1,0,1,0,1,0,1,0,1]` with the parity bits being at position 1, 2, 4, and 8.

The parity bit is calculated by XORing all the bits it covers. A parity bit covers the bits at positions where a particular binary digit is 1. The `calculate_parity_values` function performs this calculation:



```
1 def calculate_parity_values(encoded):
2     n = len(encoded)
3
4     # For each parity bit position (1, 2, 4, 8, ...)
5     for i in range(n):
6         if (i + 1) & i == 0: # Check if it's a parity position
7             parity_pos = i + 1
8             parity_sum = 0
9
10            # Check each position in the encoded message
11            for j in range(n):
12                position = j + 1 # 1-based position
13
14                # Use congruency to check if this bit is covered by current parity bit
15                # A position is covered if: position mod (2 * parity_pos) <= parity_pos
16                if position % (2 * parity_pos) <= parity_pos:
17                    parity_sum += encoded[j]
18
19            encoded[i] = parity_sum % 2
20
21    return encoded
```

Fig. 3.3 *calculate_parity_value function*

Source: Author

The congruence property is used to calculate parity bit values to determine which data bits are covered by each parity bit. Parity bits cover only a certain subset of the overall data. For example, the parity bit at position $p_1=1$ covers the bits with a particular binary position, while the parity bit at position $p_2=2$ covers a different subset. To determine whether a bit is within the range of a particular parity bit, this implementation uses the modular property. If, a bit position k is within the range of the parity bit at position p , it follows this equation:

$$k \bmod (2 \times p) \geq p$$

This means, if the result of modulus operation between bit k with $2 \times p$ is bigger or equal with p , then the bit is covered by that parity bit. This ensures that the coverage of each parity bit is consistent with the Hamming structure code.

For example, for the parity bit in position 1 (p_1), the data bits it covers are located in position 1, 3, 5, 7, 9, etc. because $k \bmod 2 \geq 1$. On the other hand, for the parity bit in position 2 (p_2), the data bits it covers are located in position 2, 3, 6, 7, 10, etc. because $k \bmod 4 \geq 2$. This implementation is done through a loop that uses this condition to check each data position. The `parity_sum` variable is added by every bit that fulfills this condition. This variable stores the total amount of bits that is covered. The parity bit value can be obtained by calculating the modulus 2 result of the `parity_sum` after the calculation is complete. This method is effective because it can identify the bit coverage directly using modular operations without the need to manually map the positions. In this way, the process of calculating the parity bit value can be automated for all types of data.

In this section, the implementation concentrates on detecting and correcting errors in data that has been encoded using Hamming codes. The function `detect_and_correct_error(encoded)` accepts an encoded parameter, which is a list of encoded data that may contain errors.

The first step is calculating the error syndrome. This is used to know if there are errors and where those errors are located. The program calculates the parity values of each parity bit position based on the encoded data. Specifically, the program iterates through the encoded data to determine whether a particular bit should contribute to the parity sum based on its position compared to the current parity bit.

The syndrome array holds every error syndrome bit. A non-zero error syndrome bit signifies an error in the encoded data. A bitwise XOR operation on the parity position where the syndrome bit is 1 is used to calculate the location of the error.

The program corrects the error by flipping the bit in the error location by using a XOR operation (`encoded[error_pos - 1] ^= 1`). Lastly, the function prints the error syndrome, the location of the error, and before

and after correction data. If there are no errors, the program only tells the user that there are no errors.



```
1 def detect_and_correct_error(encoded):
2     n = len(encoded)
3     error_pos = 0
4     syndrome = []
5
6     # Calculate syndrome bits using congruency
7     for i in range(n):
8         if (i + 1) & i == 0: # Parity position
9             parity_pos = i + 1
10            parity_sum = 0
11
12            for j in range(n):
13                position = j + 1
14                if position % (2 * parity_pos) >= parity_pos:
15                    parity_sum += encoded[j]
16
17            syndrome_bit = parity_sum % 2
18            syndrome.append(syndrome_bit)
19            if syndrome_bit:
20                error_pos ^= parity_pos
21
22    print("Syndrome bits:", syndrome[::-1])
23
24    if error_pos > 0 and error_pos <= n:
25        print(f"Error detected at position: {error_pos}")
26        print("Data before correction:", encoded)
27        encoded[error_pos - 1] ^= 1 # Correct the error
28        print("Data after correction:", encoded)
29    else:
30        print("No errors detected.")
31
32    return encoded
```

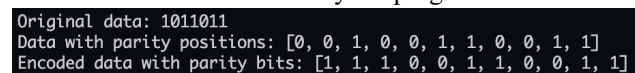
Fig. 3.4 *detect_and_correct_error function*

Source: Author

IV. ANALYSIS

The performance of the error detection algorithm was assessed across a variety of scenarios. Every scenario was created to resemble actual situations and evaluate how accurate the mathematical model is.

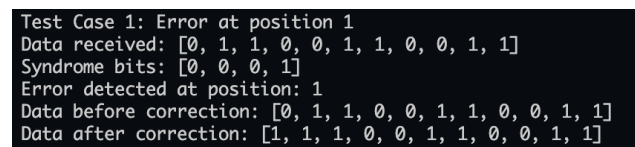
For each testcase, unless stated otherwise, the data that will be transmitted is 1011011. The encoded data that has been calculated and inserted by the program is below.



```
Original data: 1011011
Data with parity positions: [0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1]
Encoded data with parity bits: [1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1]
```

Fig. 4.1 *Original data and after encoded*

Source: Author



```
Test Case 1: Error at position 1
Data received: [0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1]
Syndrome bits: [0, 0, 0, 1]
Error detected at position: 1
Data before correction: [0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1]
Data after correction: [1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1]
```

Fig. 4.2 *First testcase result*

Source: Author

For the first testcase, an error occurs in the first position, or first bit, of the received data. The program calculates the syndrome bit and finds that there is an error in position 1 produced by the syndrome bit. Then, using the detected error position, the program corrects the erroneous bit and returns the correct data. This shows that the system has the ability to detect and correct errors in position.


```

Test Case 2: Error at position 3
Data received: [1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1]
Syndrome bits: [0, 0, 1, 1]
Error detected at position: 3
Data before correction: [1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1]
Data after correction: [1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1]

```

Fig. 4.3 Second testcase result

Source: Author

The error occurs at the third position of the received data in the second test case. After the data is received, the `detect_and_correct_error()` function again detects the error position and corrects the wrong bit. After the correction, the received data becomes the correct data again. This shows that the Hamming system can correct errors at non-parity positions such as 1, 2, 4, 8 and so on. This shows the ability of the system to find and correct errors in bits that are not parity bits.

The condition where the received data does not contain errors is tested in the third test case. The `detect_and_correct_error()` function does not find any errors and produces the same received data. This shows the ability of the Hamming system to distinguish correct data from incorrect data. Therefore, the system will not make corrections if there are no errors, showing the system's ability to find normal situations that do not require correction.

```

Test Case 3: No error
Data received: [1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1]
Syndrome bits: [0, 0, 0, 0]
No errors detected.

```

Fig. 4.4 Third testcase result

Source: Author

```

Test Case 4: Multiple errors
Data received: [1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1]
Syndrome bits: [0, 0, 1, 1]
Error detected at position: 3
Data before correction: [1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1]
Data after correction: [1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1]

```

Fig. 4.5 Fourth testcase result

Source: Author

For the fourth testcase, errors in multiple locations are simulated by flipping the 5th and 6th bits simultaneously. When the `detect_and_correct_error` function ran, the system uses the error syndrome to detect and calculate the position of errors. The error syndrome result for this test case is [0,0,1,1] which signifies that there is an error at the third position. Then the program flips the third bit.

But in reality, the errors at bits 5 and 6 was not correctly detected. In this implementation, the program can only detect and correct single-bit errors. Because more than one error occurred, the error syndrome produced is not sufficient to identify the two error locations accurately. As a result, the system only detects and corrects the third bit, even though the errors happen at the 5th and 6th bit. This shows a limitation in the hamming code system that can only detect single-bit errors, and if there are more than one error, the data that is corrected is more than likely not accurate.

Hamming codes work well for detecting and correcting single bit errors, but they cannot handle more than one bit error at a time. Therefore, if a system needs to handle multiple or more errors, more complex methods, such as more robust error detection and correction codes, must be considered.

V. CONCLUSION

Based on the results of the implementation and analysis that have been carried out, it can be concluded that number theory plays an important role in the development of error detection systems, especially Hamming Codes. The implementation of Hamming Code using number theory concepts such as modulo arithmetic and congruence properties has proven effective in detecting and correcting single bit errors in data transmission. The developed program successfully implements the Hamming Code algorithm using XOR operations and parity bit calculations, demonstrating the practical application of mathematical concepts in programming.

Testing shows that the program can detect and correct single bit errors accurately in many positions, but it cannot handle multi bit errors, this shows that more complex methods are needed for applications that need higher reliability. This study shows that basic mathematical ideas, like number theory, can be used to solve real informational technology problems.

It is recommended to study more advanced error detection and correction techniques that can handle multi-bit errors and optimize the implementation to improve computational efficiency for further development.

VI. APPENDIX

The github repository for the program used in this paper can be accessed here:
<https://github.com/sbimasena/Hamming-code-application>

VII. ACKNOWLEDGMENT

As the author of this paper, I would like to express my sincere gratitude to all parties who have provided support and inspiration during the writing process so that I can complete this paper entitled "Application of Outer Product for Vehicle Collision Detection in Automatic Navigation System" well. I would like to thank:

1. Dr. Ir. Rinaldi, M.T. and Ir. Rila Mandala, M.Eng., Ph.D. as the lecturers of IF2123 Aljabar Linier dan Geometri for the teaching of materials that have been shared in the Informatics Engineering class.
2. Both my parents for always supporting me. Their presence and positive affirmations always gives me the strength to finish this paper well.
3. My friends at Informatics Engineering class, who always cheer me up during the stressful times of creating this paper.

REFERENCES

- [1] Hamming, Richard Wesley (1950). "Error detecting and error correcting codes" (PDF). *Bell System Technical Journal*. **29** (2): 147–160.
- [2] A Pandey, H. (2024, July 26). Hamming code in Computer Network. GeeksforGeeks. https://www.geeksforgeeks.org/hamming-code-in-computer-network/?ref=gese_outind
- [3] Munir, Rinaldi. (2023). "Teori Bilangan (Bagian 1)". <https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/15-Teori-Bilangan-Bagian1-2024.pdf>
- [4] Munir, Rinaldi. (2023). "Teori Bilangan (Bagian 2)". <https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/16-Teori-Bilangan-Bagian2-2024.pdf>

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 4 Januari 2024



Sakti Bimasena - 13523053