Application of Polynomial Interpolation in Reed Solomon Error Correction

Muhammad Jibril Ibrahim 13523085[1,2]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13523085@mahasiswa.itb.ac.id, *[2]mjibrahimcollege@gmail.com*

***Abstract*— Error correction is an essential component of modern digital communication and data storage systems, ensuring data reliability in the face of noise, interference, and corruption. This article explores the application of polynomial interpolation in error correction, particularly its use in Reed-Solomon (RS) codes. Polynomial interpolation forms the mathematical foundation for constructing and decoding RS codes, enabling the detection and correction of errors. The discussion includes the principles of interpolation and polynomial interpolation, emphasizing methods like Lagrange and Newton interpolation, and their role in solving systems of linear equations. Furthermore, the article delves into the algebraic structure of Galois fields, which underpin the operations in RS codes. These fields facilitate efficient encoding and decoding processes, allowing RS codes to recover original data even in adverse conditions. Reed-Solomon codes are highlighted for their robustness and versatility, with detailed explanations of their encoding and decoding mechanisms. Practical examples illustrate their applications in correcting burst and random errors, as seen in CDs, DVDs, QR codes, and satellite communications. By examining the interplay between mathematics and technology, this article underscores the critical role of polynomial interpolation and RS codes in preserving data integrity across diverse domains.**

***Keywords*— Error Correction, Polynomial Interpolation, Reed-Solomon Codes, Galois Fields, Encoding and Decoding, Finite Fields,**

I. Introduction

Error correction is a cornerstone of modern digital communication and storage systems, ensuring the reliability and integrity of data in environments prone to noise, interference, and corruption. At the heart of error correction lies polynomial interpolation, a mathematical technique that enables the reconstruction of missing or corrupted information. Among its most impactful applications is in Reed-Solomon codes, a class of error-correcting codes that have become ubiquitous in everyday technology.

Reed-Solomon codes leverage polynomial interpolation to encode data into codewords that are resilient to errors. When data is transmitted or stored, it is often subjected to distortions caused by physical damage, electromagnetic interference, or other forms of corruption. By applying the principles of polynomial interpolation during decoding, Reed-Solomon codes can detect and correct errors, recovering the original data even when parts of it are compromised. This capability is essential for maintaining the integrity of data in a wide range of applications.

The importance of error correction cannot be overstated. It is the reason why scratched compact discs can still play music, damaged QR codes can still be scanned, and satellite communications remain reliable despite atmospheric interference. Error correction through Reed-Solomon codes is also critical in high-density data storage, such as DVDs and Blu-ray discs, as well as in data transmission over fiber-optic cables and wireless networks. The ability to ensure accurate data retrieval in these scenarios is a testament to the robustness and versatility of error-correcting algorithms.

This article delves into the role of polynomial interpolation in error correction, with a focus on its implementation in Reed-Solomon codes. By exploring the mathematical foundations and practical applications of these codes, we aim to highlight their pervasive presence in modern technology and their crucial role in preserving data integrity across diverse domains.

II. Fundamental Theorem

A. Interpolation

Interpolation, in mathematics, is the process of estimating values between known data points by constructing a function that passes through those points. Given a set of distinct data points (x_i, y_i) interpolation seeks to find a function such that $f(x_i) = y_i$ for every points in the dataset.

For example, given these 4 known points $(1,-2), (2,6), (4,28), (5,42)$. We can create a function $f(x) = x^2 + 5x - 8$ that satisfies every known points. Through this function we can estimate the value of an unknown points such as $(3,16)$. But do note that the estimation may not be accurate as there may exist another function that satisfies the known points.

B. Polynomial Interpolation

One of the simplest type of interpolation is the polynomial interpolation. Polynomial interpolation is a method to estimate values between known data points by constructing a polynomial that passes through all the known points. The basis of using polynomial interpolation to estimate values is from Weierstrass' theorem, which states that every continuous function on a closed interval can be approximated arbitrarily well by a polynomial. Polynomial interpolation is also supported by the uniqueness theorem which states that polynomials that fit the interpolation condition exist and is unique.

In general, given a set of $n+1$ distinct known points $(x_i, y_i), i = 0,1,2,3,\ldots,n$. The polynomial that is constructed by the interpolation will be of degree n. $P(x) = c_0 + c_1x + c_2x^2 + c_3x^3 + \cdots + c_nx^n$. Though there are many ways to construct such polynomial, one way is to construct a system of linear equation where $P(x_i) = y_i$ for every known points.

$$P(x_0) = c_0 + c_1x_0 + c_2x_0^2 + c_3x_0^3 + \cdots + c_nx_0^n = y_0$$
$$P(x_1) = c_0 + c_1x_1 + c_2x_1^2 + c_3x_1^3 + \cdots + c_nx_1^n = y_1$$
$$P(x_2) = c_0 + c_1x_2 + c_2x_2^2 + c_3x_2^3 + \cdots + c_nx_2^n = y_2$$
$$P(x_3) = c_0 + c_1x_3 + c_2x_3^2 + c_3x_3^3 + \cdots + c_nx_3^n = y_3$$
$$\ldots.$$
$$P(x_n) = c_0 + c_1x_n + c_2x_n^2 + c_3x_n^3 + \cdots + c_nx_n^n = y_n$$

With this linear equation system, we can then find the coefficients of the polynomial and create the final polynomial solution of the interpolation. To prove the uniqueness of the solution we can create a *Vandermonde* matrix from the linear equation system.

$$\boldsymbol{V} = \begin{vmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{vmatrix}$$

In linear algebra, we found that the whole system of equations has a unique solution if and only if the determinant of the *Vandermonde* matrix is 0. when $i \neq j$ and $x_i \neq x_j$, then

$$|\boldsymbol{V}| = \prod_{0 \le i < j \le n} (x_j - x_i) \neq 0$$

Which means as long as the given points are distinct, the polynomial created from the interpolation exist and is unique.

C. System of Linear Equation

A system of linear equations consists of multiple linear equations that share a common set of variables. The purpose is to find values for these variables that satisfy all the equations simultaneously. For example, a system like the following,

$$3x_1 + 2x_2 = 5$$
$$x_1 + 3x_2 = -3$$

Has the solution $x_1 = 3$ and $x_2 = -2$.

While there are many methods to solve linear equation system such as elimination, substitution, and using graphs, the most common method is the matrix method as it makes it easier to code and compute. The matrix method write the linear equation system into $Ax = b$ where A is the coefficient matrix while b is vector of constant. For example, the previous system can be rewritten into

$$\begin{vmatrix} 3 & 2 \\ 1 & 3 \end{vmatrix} x = \begin{vmatrix} 5 \\ -3 \end{vmatrix}$$

From here, there are multiple ways to solve the linear equation system such as *Gaussian Elimination, Gauss-Jordan Elimination, Cramer's Rule,* and others.

D. Reed Solomon Code

Reed-Solomon (RS) codes are a non-binary cyclic codes with symbols made up of m-bit sequence, where m is an integer bigger than 2. Reed-Solomon codes are defined over a finite field, specifically Galois field, and are characterized by tow parameters, n and k, where n is the total number of symbols in the encoded block and k is the number of data symbols being encoded. Reed-Solomon (n, k) codes exist for all n and k where,

$$0 < k < n < 2^m + 2$$

In general, Reed-Solomon (n, k) code is used with the parameter $(n, k) = (2^m - 1,\ 2^m - 1 - 2t)$. Where t is the number of symbol error correction capability of the code and $n - k$ is the number of parity symbols in the code. The code is capable of correcting any combination of t or fewer error with

$$t = \left\lfloor \frac{n-k}{2} \right\rfloor$$

Reed-Solomon codes are constructed by evaluating polynomials at distinct points in a finite field. Given a message polynomial $P(x)$ of degree $k - 1$, the encoded block is formed by evaluating $P(x)$ at n distinct points, resulting in an codeword of length n.

RS codes can correct both burst errors and random errors. A burst error affects consecutive symbols, while random errors occur at arbitrary positions. The use of finite fields ensures that all operations remain within a fixed range, simplifying computations and enabling hardware implementations.

Reed-Solomon codes are highly effective against burst and random errors, making them indispensable in technologies like CDs, DVDs, QR codes, and satellite transmissions. By leveraging polynomial interpolation, these codes ensure robust error correction and data recovery even in challenging conditions.

E. Galois Fields

Galois fields, also known as finite fields, are mathematical structures comprising a finite number of elements in which addition, subtraction, multiplication, and division (excluding division by zero) are well-defined and satisfy the field axioms. These fields are fundamental in various areas of mathematics and have practical applications in coding theory, cryptography, and digital signal processing.

A Galois field contains a limited set of elements. The number of elements in such a field is always a power of a prime number, denoted as p^n, where p is a prime number and n is a positive integer. For instance, the field $GF(2)$ consists of two elements, 0 and 1, and is called a base field. While the field $GF(2^4)$ consist of sixteen elements, from 0 to 15, and is called an extension field.

Within a Galois field, the operations of addition, subtraction, multiplication, and division (except by zero) are defined and adhere to the field axioms, ensuring properties like associativity, commutativity, distributivity, and the existence of additive and multiplicative identities and inverses.

In extension fields $GF(p^n)$ the elements are polynomials with coefficients in the base field $GF(p)$. These polynomials have degrees less than n, and arithmetic operations on them (addition, subtraction, multiplication, and division) are performed modulo an irreducible polynomial $P(x)$. This irreducible polynomial ensures that the set forms a field, meaning all operations (except division by zero) are valid.

In a Galois field $GF(2)$ (binary field), addition and subtraction operations is followed with modulo 2 which is equivalent to XOR operation. For example, assume $a(x) = x^2 + 1$ and $b(x) = x + 1$. Polynomial $a(x)$ can be represented as 101 and $b(x)$ can be represented as 011. The addition result of the polynomial is $x^2 + x$ which can be represented as 110 and is the same value we get when we XOR $a(x)$ and $b(x)$. The same principle applies for subtraction as well.

III. Reed-Solomon Code System

A. Encoding

The encoding process begins by representing the input data as a polynomial over a finite field, $GF(q)$, where $q = p^m$, p is a prime number, and m determines the size of the field. The message is composed of k symbols, each corresponding to an element of $GF(q)$. These symbols are arranged into a polynomial $m(x)$ with coefficients representing the data symbols:

$$m(x) = m_0 + m_1x + m_2x^2 + \cdots + m_{k-1}x^{k-1}$$

With $m_0, m_1, m_2, \dots, m_{k-1} \in GF(q)$ and the degree of $m(x)$ is $k - 1$. This polynomial is a mathematical abstraction of the data and serves as the basis for constructing the Reed-Solomon codeword.

Before we can use the previously made polynomial, there is another polynomial we must create first which is the generator polynomial $g(x)$, which determines the structure of the codeword. The generator polynomial is of degree $n - k$, where n represents the total number of symbols in the final codeword, and $n - k$ represents the parity-check symbols. The generator polynomial is constructed as:

$$g(x) = (x - \alpha)(x - \alpha^1)(x - \alpha^2) \dots (x - \alpha^{n-k})$$

Where α is a primitive element of $GF(q)$. The choice of $g(x)$ ensures that it has $n - k$ consecutive roots in $GF(q)$. This property is fundamental to the error-correcting capability of RS codes, as it guarantees that any valid codeword polynomial $U(x)$ is divisible by $g(x)$.

After creating both the message polynomial $m(x)$ and generator polynomial $g(x)$, we can now encode the data by first shifting the message polynomial into the rightmost k stages of a codeword register and then appending a parity polynomial $p(x)$ by placing it in the leftmost $n - k$ stages. We can do this by multiplying $m(x)$ by x^{n-k} which would algebraically shift the message polynomial. After this, we divide $x^{n-k}m(x)$ by the generator polynomial $g(x)$, which is written in the following form:

$$x^{n-k}\mathrm{m}(x) = \mathrm{q}(x)\mathrm{g}(x) + \mathrm{p}(x)$$

Where $\mathrm{q}(x)$ is the quotient polynomials and $\mathrm{p}(x)$ is the remainder polynomial which is also the parity. With both the message polynomial and parity polynomial created, we can construct the final codeword polynomial by combining the message polynomial and parity polynomial, which is written in the following form:

$$U(x) = x^{n-k}m(x) + p(x)$$

Where $\mathrm{q}(x)$ is the quotient polynomials and $\mathrm{p}(x)$ is the remainder. This codeword is structured so that the first k terms correspond to the message, while the remaining $n - k$ terms are parity symbols. A valid codeword polynomial is of the following form:

$$\mathrm{U}(x) = \mathrm{m}(x)\mathrm{g}(x)$$

Which means that the root of a generator polynomial must be the same as the root of the codeword polynomial. Therefore, for an arbitrary codeword polynomial, when evaluated at any root of the generator polynomial must result in zero. This fact is used for decoding and error detection a codeword.

B. Decoding & Error-Correction

The decoding process is relatively more complex than its encoding counterpart. Decoding can be divided to four steps, syndrome calculation for detecting error, locating error, determining the error magnitude, and correcting the error. The last two process can be skipped if there is no error detected by the syndrome calculation.

The decoding starts with a received sequence, which may differ from the transmitted codeword due to errors which can be represented as:

$$r(x) = U(x) + e(x)$$

Where $r(x)$ is the received polynomial, $U(x)$ is the original (transmitted) codeword polynomial, and $e(x)$ is the error polynomial, which has nonzero coefficients only at positions corresponding to errors.

The decoder then calculates syndromes by evaluating $r(x)$ at specific points related to the generator polynomial's roots, which can be written to,

$$S_i = r(\alpha^i)\ for\ i = 1, 2, \dots, 2t$$

Where α is a primitive element of the Galois is field and t is the error-correcting capability of the code. Each S_i is essentially the evaluation of the received polynomial at successive powers of the primitive element.

As mention before in the encoding section, for an arbitrary codeword, when evaluated at any root of the generator polynomial, α^i, must result in zero. Therefore, if all syndromes are zero, the received sequence is error-free. Otherwise, the syndromes reveal the presence of errors.

After knowing that the error exist, we must first locate it before we can correct it. An error locator polynomial $\sigma(x)$ is defined as:

$$\sigma(x) = \prod_{j=1}^{v}(1 - \beta_j x) = 1 + \sigma_1 X + \sigma_2 X^2 + \cdots + \sigma_v X^v$$

Where $\beta_j = \alpha^{l_j}$ are the error locations, v is the number of errors, and σ_i are the coefficients of the error locator polynomial. The reciprocal roots of $\sigma(x)$ are the error locations. To compute $\sigma(x)$ we can derive a system of equations using the error locator polynomial and syndromes, which can be written as a matrix like the following:

$$\begin{bmatrix} S_t & S_{t-1} & \dots & S_1 \\ S_{t+1} & S_t & \dots & S_2 \\ \vdots & \vdots & \ddots & \vdots \\ S_{2t-1} & S_{2t-2} & \dots & S_t \end{bmatrix} \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \vdots \\ \sigma_t \end{bmatrix} = \begin{bmatrix} -S_{t+1} \\ -S_{t+2} \\ \vdots \\ -S_{2t} \end{bmatrix}$$

The matrix equation can be solved using many techniques, but the most used is the Berlekamp-Massey which can efficiently finds the coefficient of the error locator polynomial $\sigma(x)$.

After finding the error locations, the magnitudes of the errors must be determined to correctly fix the error in the codeword. To determine them, another set of equation system that uses the syndrome and the error locations as is the following:

$$S_i = \sum_{j=1}^{v} e_j \beta_j^i \, , i = 1,2, \dots ,2t$$

Where e_j are the error values and β_j are the error locations. Solving this equation system provides the magnitudes of errors at the identified locations.

With both the locations and values of errors identified, the errors can now be subtracted from the received codeword to recover the original data. To do this, we construct the error polynomial with the following definition:

$$e(X) = \sum_{j=1}^{v} e_j X^{l_j}$$

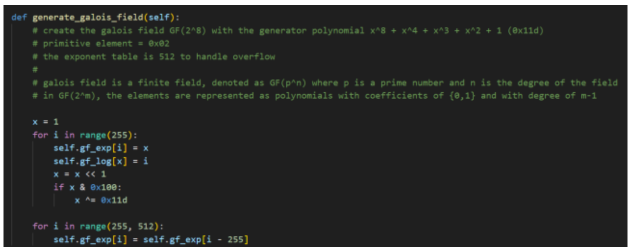
Where j is the error index (the j-th error) and l_j is the location of the j-th error. The corrected codeword is then calculated by the following equation:

$$\hat{\mathbf{U}}(X) = \mathbf{r}(X) + \hat{\mathbf{e}}(X) = \mathbf{U}(X) + \mathbf{e}(X) + \hat{\mathbf{e}}(X)$$

With this the removes the errors and restores the transmitted codeword to its original state with The rightmost k symbols represent the original message.

IV. Implementation

The program is made by python and has a single class of ReedSolomon that encompass all the functions needed to encode and decode Reed-Solomon codes. The created functions is mainly split into two types, function for Galois field arithmetic operation and polynomial arithmetic operation. The function that we first create is the function to create the Galois field.

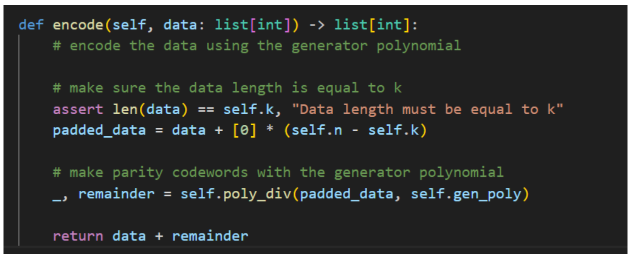


```python
def generate_galois_field(self):
    # create the galois field GF(2^8) with the generator polynomial x^8 + x^4 + x^3 + x^2 + 1 (0x11d)
    # primitive element = 0x02
    # the exponent table is 512 to handle overflow
    #
    # galois field is a finite field, denoted as GF(p^n) where p is a prime number and n is the degree of the field
    # in GF(2^m), the elements are represented as polynomials with coefficients of {0,1} and with degree of m-1

    x = 1
    for i in range(255):
        self.gf_exp[i] = x
        self.gf_log[x] = i
        x = x << 1
        if x & 0x100:
            x ^= 0x11d

    for i in range(255, 512):
        self.gf_exp[i] = self.gf_exp[i - 255]
```

Fig 4.1. Generate Galois field
Source : writer’s archive

We create the Galois field with the primitive polynomial $x^8 + x^4 + x^3 + x^2 + 1$ which can be represented by the hex 0x11d. The Galois field is done by using two list, *gf_exp* and *gf_log* which is a lookup table that is made for efficient multiplication and division in the Galois field.

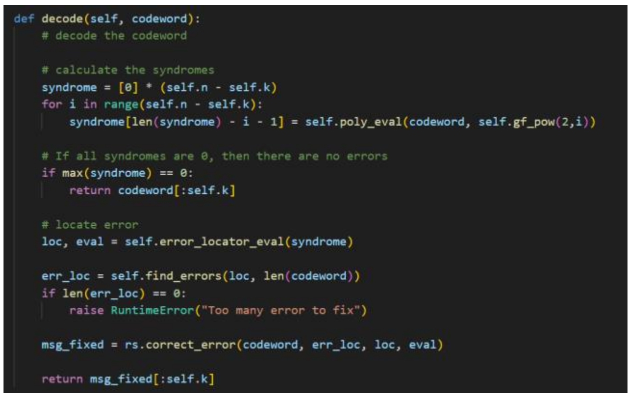
The encoding is computed by making a padded data with the length of n and dividing them with the generator polynomial that we have created. We append the remainder of the division into the original data and we have our encoded data with $(n - k)/2$ error correction capabilities.



```python
def encode(self, data: list[int]) -> list[int]:
    # encode the data using the generator polynomial

    # make sure the data length is equal to k
    assert len(data) == self.k, "Data length must be equal to k"
    padded_data = data + [0] * (self.n - self.k)

    # make parity codewords with the generator polynomial
    _, remainder = self.poly_div(padded_data, self.gen_poly)

    return data + remainder
```

Fig 4.2. Encode the data
Source : writer’s archive

The decoding is done by first calculate the syndrome of the codeword, where if all the syndrome is zero then the codeword has no error. Otherwise, the calculated syndrome is used in the error locator function.

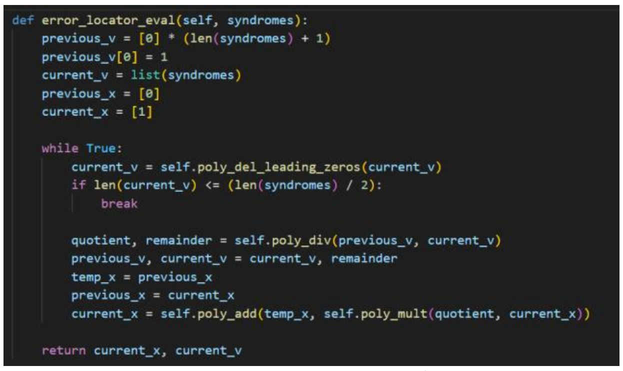


```python
def decode(self, codeword):
    # decode the codeword

    # calculate the syndromes
    syndrome = [0] * (self.n - self.k)
    for i in range(self.n - self.k):
        syndrome[len(syndrome) - i - 1] = self.poly_eval(codeword, self.gf_pow(2,i))

    # If all syndromes are 0, then there are no errors
    if max(syndrome) == 0:
        return codeword[:self.k]

    # locate error
    loc, eval = self.error_locator_eval(syndrome)

    err_loc = self.find_errors(loc, len(codeword))
    if len(err_loc) == 0:
        raise RuntimeError("Too many error to fix")

    msg_fixed = rs.correct_error(codeword, err_loc, loc, eval)

    return msg_fixed[:self.k]
```

Fig 4.3. Decode codeword main function
Source : writer’s archive

The error_locator_eval function is a function for identifying the locations of errors in the received codeword. It uses the Berlekamp-Massey algorithm to iteratively compute the error locator polynomial and its corresponding evaluator polynomial from the syndromes.

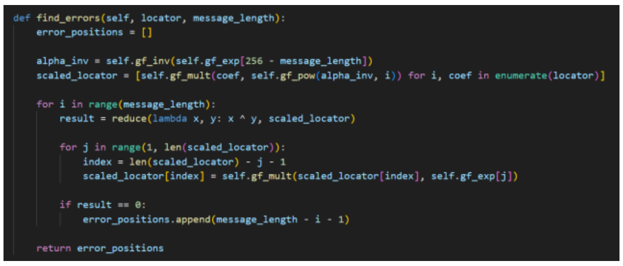
The syndromes are values that indicate discrepancies between the received codeword and the expected codeword. The output of this function is the final error locator polynomial (current_x) and the evaluator polynomial (current_v), which will later be used for finding the exact error positions and magnitudes.



```python
def error_locator_eval(self, syndromes):
    previous_v = [0] * (len(syndromes) + 1)
    previous_v[0] = 1
    current_v = list(syndromes)
    previous_x = [0]
    current_x = [1]

    while True:
        current_v = self.poly_del_leading_zeros(current_v)
        if len(current_v) <= (len(syndromes) / 2):
            break

        quotient, remainder = self.poly_div(previous_v, current_v)
        previous_v, current_v = current_v, remainder
        temp_x = previous_x
        previous_x = current_x
        current_x = self.poly_add(temp_x, self.poly_mult(quotient, current_x))

    return current_x, current_v
```

Fig 4.4. Error locator and evaluator function
Source : writer’s archive

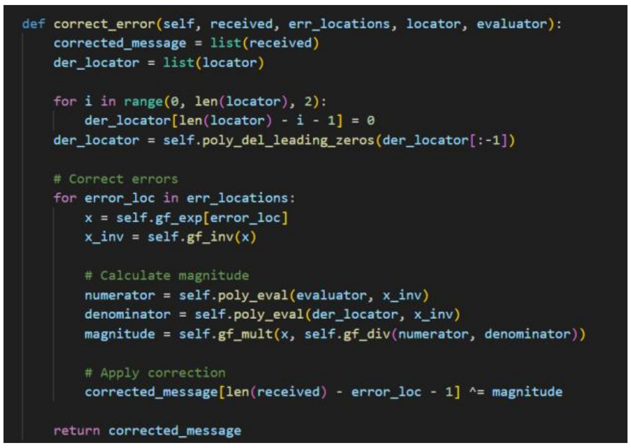
The find_errors function is responsible for locating the positions of the errors in the codeword. It leverages the error locator polynomial generated by error_locator_eval. The error locator polynomial has roots that correspond to the locations of the errors in the received message. This function evaluates the locator polynomial at different powers of a primitive element α, essentially checking where the polynomial evaluates to zero. The positions where the polynomial evaluates to zero indicate the error locations.



```python
def find_errors(self, locator, message_length):
    error_positions = []

    alpha_inv = self.gf_inv(self.gf_exp[256 - message_length])
    scaled_locator = [self.gf_mult(coef, self.gf_pow(alpha_inv, i)) for i, coef in enumerate(locator)]

    for i in range(message_length):
        result = reduce(lambda x, y: x ^ y, scaled_locator)

        for j in range(1, len(scaled_locator)):
            index = len(scaled_locator) - j - 1
            scaled_locator[index] = self.gf_mult(scaled_locator[index], self.gf_exp[j])

        if result == 0:
            error_positions.append(message_length - i - 1)

    return error_positions
```

Fig 4.5. Find the location of the error
Source : writer’s archive

The correct_error function uses the error locator polynomial and evaluator polynomial to correct the errors found in the received codeword. Once the error locations are determined, the function iterates through each error position and calculates the magnitude of the error at that position. The error magnitude is determined by evaluating the error evaluator polynomial and the derivative of the error locator polynomial at the corresponding position. These evaluations give the values needed to correct the error. Specifically, the function evaluates the locator polynomial at the inverse of the error position (to find the location in the Galois Field), and uses this to compute the error magnitude. It then XORs (corrects) the received codeword by the magnitude at the corresponding positions, fixing the errors. The result is a corrected version of the received codeword, now matching the original transmitted message.



```python
def correct_error(self, received, err_locations, locator, evaluator):
    corrected_message = list(received)
    der_locator = list(locator)

    for i in range(0, len(locator), 2):
        der_locator[len(locator) - i - 1] = 0
    der_locator = self.poly_del_leading_zeros(der_locator[:-1])

    # Correct errors
    for error_loc in err_locations:
        x = self.gf_exp[error_loc]
        x_inv = self.gf_inv(x)

        # Calculate magnitude
        numerator = self.poly_eval(evaluator, x_inv)
        denominator = self.poly_eval(der_locator, x_inv)
        magnitude = self.gf_mult(x, self.gf_div(numerator, denominator))

        # Apply correction
        corrected_message[len(received) - error_loc - 1] ^= magnitude

    return corrected_message
```

Fig 4.6. Error correction function
Source : writer’s archive

V. Conclusion

The application of polynomial interpolation in error correction, particularly through Reed-Solomon codes, highlights the synergy between mathematical theory and practical technology. Reed-Solomon codes, built on the principles of polynomial interpolation and Galois fields, demonstrate exceptional capabilities in detecting and correcting both burst and random errors in diverse data transmission and storage scenarios.

The encoding and decoding mechanisms of these codes, along with their reliance on efficient mathematical algorithms such as the Berlekamp-Massey algorithm, underline their robustness and efficiency in real-world applications. From CDs and DVDs to QR codes and satellite communications, Reed-Solomon codes ensure data integrity even in adverse conditions, making them indispensable in modern digital systems.

This study underscores the enduring importance of mathematical tools in advancing technology, serving as a testament to the practical impact of concepts like polynomial interpolation and finite fields. Future research and development may continue to enhance the performance and versatility of these codes, paving the way for even more resilient data systems.

VI. Acknowledgment

The author expresses heartfelt gratitude to God Almighty, Allah Subhanahu wa Ta’ala, for His blessings and guidance, enabling the completion of this paper titled “Application of Polynomial Interpolation in Reed Solomon Error Correction” in a timely manner. The author extends sincere appreciation to their parents and friends for their unwavering support and encouragement, particularly in providing mental motivation throughout the writing process. Special thanks go to Dr. Rila Mandala, the lecturer for Linear Algebra and Geometry K1 in the 2024/2025 academic year, for sharing valuable knowledge and providing guidance during the learning process. The author is also deeply grateful to Dr. Rinaldi Munir, one of the lecturers for the same course, for delivering extensive materials and references that were instrumental both during the lectures and in the preparation of this paper. Lastly, the author would like to thank all other individuals and parties who contributed to the completion of this paper.

REFERENCES

[1] Reed, I. S. and Solomon, G., “Polynomial Codes Over Certain Finite Fields,” SIAM Journal of Applied Math., vol. 8, 1960, pp. 300-304.

[2] Qunying, Liao. (2010). On Reed-Solomon codes. Chinese Annals of Mathematics. Series B. 32. 89-98. 10.1007/s11401-010-0622-3.

[3] Berlekamp, E. R., Peile, R. E., and Pope, S. P., “The Application of Error Control to Communications,” IEEE Communications Magazine, vol. 25, no. 4

[4] Xu, Y., & Xu, R. (2022). *Research on interpolation and data fitting: Basis and applications*. arXiv preprint arXiv:2208.11825.

[5] Wong, Jeffrey. 2020. "Polynomial Interpolation: The Fundamentals." Math 563 Lecture Notes, Duke University. Available at https://services.math.duke.edu/~jtwong/math563-2020/lectures/Lec1-polyinterp.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, 27 Desember 2024

[illegible]

Muhammad Jibril Ibrahim
13523085