SJÄLVSTÄNDIGA ARBETEN I MATEMATIK

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET

Reed-Solomon Codes

av

Elise Reuterskiöld

2025 - No K19

Reed-Solomon Codes

Elise Reuterskiöld

Självständigt arbete i matematik 15 högskolepoäng, grundnivå

Handledare: Samuel Lundqvist

2025

Abstract
Reed-Solomon codes are a class of error-correcting codes widely used in digital communication and storage systems due to their ability to detect and correct multiple errors that occur during transmission. This paper provides a comprehensive overview of the theoretical foundation, encoding and decoding processes of cyclic Reed-Solomon codes over Galois fields. Two decoding methods are analyzed, the Direct method and the Euclidean method. Their theoretical structure are discussed in detail, including syndrome computation, error locator and error magnitude polynomials and error correction limits. To evaluate practical performance, both methods were implemented in Python and their computational efficiency was compared under varying parameters such as Galois field order, block length and number of errors. The results show that the Euclidean method scales more efficiently with increasing possible and actual errors, making it more suitable for real-world applications involving high error rates. Finally, implementation challenges are explored along with an analysis of how algorithmic choices affect performance and reliability.

Sammanfattning
Reed-Solomon-koder är en klass av felrättande koder som används i stor utsträckning inom digital kommunikation och lagringssystem, tack vare deras förmåga att upptäcka och korrigera flera fel som kan uppstå under överföring. Denna uppsats ger en omfattande översikt över den teoretiska grunden samt kodnings och avkodningsprocesserna för cykliska Reed-Solomon-koder över olika ändliga kroppar. Två avkodningsmetoder analyseras, den direkta metoden och den euklidiska metoden. Metodernas teoretiska grund diskuteras i detalj, såsom syndromberäkning, fellokaliserings- och felmagnitudspolynom, samt begränsningar för felkorrigering. För att utvärdera praktisk prestanda implementerades båda metoderna i Python, och deras beräkningsmässiga effektivitet jämfördes under varierande parametrar, såsom antal element i kroppen, blocklängd och antal fel. Resultaten visar att den euklidiska metoden är effektivare vid ökande möjliga och faktiska fel. Detta gör den mer lämplig för verkliga tillämpningar med höga felnivåer. Slutligen utforskas implementeringsutmaningar, tillsammans med en analys av hur algoritmiska val påverkar prestanda och tillförlitlighet.

Contents

1 Introduction

Error-correcting codes play a significant role in ensuring reliable data transmission and storage in modern communication systems. These codes enable the recovery of lost or corrupted data by introducing redundancy in the transmitted message, allowing the detection and correction of errors. Among the various error-correcting codes, *Reed-Solomon codes* have gained widespread recognition due to their robustness and efficiency in correcting errors in noisy environments.

These codes are a class of non-binary cyclic codes that are widely used in applications such as digital communications, data storage, and broadcasting. Defined over finite fields, they are particularly effective in correcting multiple consecutive errors. The error-correction capability of a Reed-Solomon code is determined by its codeword length, n, and the length of the actual information k.

The Reed–Solomon codes were first developed in 1960 by the American mathematicians Irving Stoy Reed and Gustave Solomon. During the early 1960s, further advancements were made, particularly a new approach based on the **B**ose–**C**haudhuri–**H**ocquenghem (BCH) codes, developed by Raj Chandra Bose, D. K. Ray-Chaudhuri, and Alexis Hocquenghem, were introduced. The original decoding method for the BCH version of Reed–Solomon codes is referred to as the *Direct method*, as it solely depends on the structure and definitions of the code. Throughout the latter half of the 20th century, various decoders and new versions of the original scheme were created. Among them, a decoding method called the *Euclidean decoder*, which utilizes the Euclidean algorithm, was developed. The Euclidean method and the Direct method will be the main focus of this paper.

The versatility of Reed-Solomon codes has led to their widespread use across various domains. They have had a profound impact on data storage technologies such as CDs, DVDs, and Blu-ray disks, where they enable recovery of data even in the presence of scratches or damage. They have also been applied to digital television broadcasting, were they protect against errors that occur during signal transmission, ensuring high-quality reception. Reed-Solomon codes also play a critical role in applications like QR codes for data retrieval, satellite communication, and even deep space communication, where data transmission is subject to extreme environmental conditions. Multiple NASA space crafts such as Voyager, the Galileo orbiter and the Mars observer, utilize Reed-Solomon encoding for deep space communication [6]. Although newer techniques for deep space communication are continuously implemented, Reed-Solomon remain in use [2], with the latest publicly available reference dating to 2019 [3].

This paper presents the theoretical background of error-correcting codes, with a primary focus on Reed-Solomon codes. Furthermore, the decoding process of a Reed-Solomon code is explained, accompanied by an example. Two different decoding methods, the Direct method and the Euclidean method, are described,

with examples provided for both. Lastly, the paper discusses an implementation of the encoding and decoding methods in Python. The efficiency of the two methods is compared, and their differences are further analyzed.

2 Preliminaries

This section provides an overview of key concepts and definitions essential for understanding the theoretical foundation of this study, as well as introduce the relevant literature.

2.1 Literature

This study incorporates several fundamental definitions and key concepts from *Error-control coding for data networks* by Irving S. Reed and Xuemin Chen [1]. It uses the clear and well-established definitions presented in the text to provide a consistent theoretical framework. Specifically, the fundamental concepts and principles discussed in Chapters 2, 4, 5, and 6 have been referenced to ensure consistency with established academic discourse in the field of error correction.

These foundational concepts serve as the starting point for the theoretical discussions in this study, providing clarity and a shared understanding of key ideas. Where necessary, additional sources have been used to supplement the initial concepts derived from *Error-control coding for data networks*, ensuring a comprehensive approach to the topic while maintaining consistency with established definitions.

2.2 Groups, Rings and Fields

This paper assumes a basic understanding of Galois fields. A brief introduction to the properties of a group, ring, field and Galois field is therefore provided. Firstly, a group is defined as a set of elements that, together with an associative operation, satisfies three fundamental properties.

Definition 1. Group *A group G is an algebraic structure consisting of a set of elements such that under an associative operation $\star$, the set is closed, contains an identity element and every element in the set has an inverse.*

An *abelian group* is a group that is commutative, meaning that for two elements $a, b \in F$ with operation $\star$, it holds that $a \star b = b \star a$. A *ring* R is a set of elements under addition and multiplication, where addition forms an abelian group and multiplication is associative and distributes over addition.

Definition 2. Ring *A ring R is an algebraic structure consisting of a set R with two binary operations, addition and multiplication. The set R is an abelian group under addition and it is closed under multiplication. There exists a*

multiplicative identity element and multiplication is associative and distributive over addition. That is, for all $a, b, c \in F$, it holds that

$$a \cdot (b + c) = a \cdot b + a \cdot c.$$

By adding further constraints on a ring R, a *field* can be constructed. A field F is a set of elements under two operations, $+$ and $\cdot$, such that F is an abelian group under $+$ and $F \backslash \{0\}$ is an abelian group under $\cdot$. If the number of elements in the field is finite, the field is called a *finite field*, also known as a *Galois field*, named after the French mathematician Évariste Galois.

Definition 3. Field *A field F is a ring in which multiplication is commutative and every nonzero element has a multiplicative inverse.*

Definition 4. Galois Field *A Galois field is a field F that contains a finite number of elements.*

An important property of a Galois field is its order, i.e., the number of elements it contains. The order of a Galois field is always a prime power, p^m.

Theorem 1. Order of a Galois Field *The order of a Galois field is a prime power p^m, for a prime p and a positive integer m.*

Most commonly, Galois fields are finite fields consisting of p elements. Such fields are isomorphic to the field of integers modulo p, denoted $\mathbb{Z}_p$. In this paper, the notation $GF(p)$ will be used to represent a Galois field of order p, where p is a prime.

Theorem 2. Field of Prime Order *If a field has p elements, where p is a prime, then the field is isomorphic to the finite field $\mathbb{Z}_p$.*

A finite field always contains at least one primitive element, called a generator. This primitive element α can be used to generate all other elements of the field. This means that every non-zero element in the field can be expressed as a power of α. In particular, for every element $x \in GF(p) \setminus \{0\}$, there exists a natural number i such that $x = \alpha^i$.

Definition 5. Generator *Let F be a finite field. A generator $\alpha \in F$ is an element such that for every $x \in F \setminus \{0\}$, there exists a natural number i for which $x = \alpha^i$.*

3 Overview Error Correcting Codes

The purpose of any error-correcting code is to facilitate the handling of errors in data transmission over a noisy channel. The fundamental principle of error correction is the introduction of redundant elements into the original information. This additional information is then used by the receiver to reconstruct the original data. Error-correcting codes can be broadly classified into two main

categories: block codes and convolutional codes. The focus of this paper, Reed-Solomon codes, belongs to the class of block codes. Block codes operate on fixed-length blocks of data, encoding each block independently to produce a corresponding codeword. The encoding process, typically performed within a Galois field, transforms a message $\mathbf{m}$ of length k into a codeword of length n, where $k < n$, and can be described by the encoding function

$$E : GF(p)^k \to GF(p)^n,$$

where $GF(p)^k$ is the set of vectors of length k with elements from $GF(p)$. The set of all possible codewords,

$$C = \{E(\mathbf{m}) | \mathbf{m} \in GF(p)^k\} \subseteq GF(p)^n,$$

constitutes the code. Based on the algebraic structure of a code, it can be classified as a linear code and, more specifically, as a cyclic code.

3.1 Linear Codes

When working with error correction, efficiency is an important factor to consider. A specific structure of the code ensures that decoding codewords can be performed more efficiently. This is achieved by constructing a linear code.

Definition 6. Linear Code *A code C of length n over a Galois field $GF(p)$ is called a linear code if it forms a k-dimensional subspace of the vector space $GF(p)^n$. That is, for any two codewords $c_1, c_2 \in C$, any linear combination of c_1 and c_2 is also in C.*

For a linear code, a codeword $\boldsymbol{c}$ of length n is generated from an original message $\boldsymbol{m}$ of length k. A code C over a Galois field $GF(p)$ therefore contains p^k distinct codewords created from messages $\boldsymbol{m}$. The additional $t = n - k$ elements in the codeword are called the *parity*.

Every linear code with codewords of length n and message length k can be described by a *generator matrix* G, which defines how message vectors are mapped to codewords. The generator matrix has dimensions $k \times n$ and is given by

$$G \in GF(p)^{k \times n}.$$

Definition 7. Generator Matrix *Let C be a linear code with codewords of length n and message length k over a Galois field $GF(p)$. The generator matrix G is a $k \times n$ matrix such that the code is the row space of G, meaning each codeword is a linear combination of the rows of G.*

For a given message row vector $\boldsymbol{m}$ of length k, the corresponding codeword $\boldsymbol{c}$ is obtained through the matrix multiplication

$$\boldsymbol{c} = \boldsymbol{m}G.$$

If the $k \times n$ generator matrix G is structured so that the first k symbols correspond to the message itself, it is said to be in *standard form*. For a $k \times k$ identity matrix I_k and some matrix A, the generator matrix in standard form is given by

$$G = [I_k \mid A].$$

Every generator matrix for a linear code can be expressed in standard form due to the linearity of the code. The generator matrix can also be used to construct the *parity-check matrix* H, which provides a way to verify whether a given vector is a valid codeword. Assuming that the generator matrix is given in standard form as described above, the corresponding parity-check matrix is given by

$$H = [-A^T \mid I_t].$$

Every linear code thus has a parity-check matrix H of dimensions $(n-k) \times n$ (alternatively, $t \times n$), which is related to the generator matrix G by the condition

$$HG^T = 0.$$

Definition 8. Parity-Check Matrix *The parity-check matrix H of a linear code C is an $t \times n$ matrix that satisfies the condition*

$$HG^T = 0,$$

where G is the generator matrix of the code and 0 is the $t \times k$ null matrix.

As a direct consequence, every valid codeword satisfies

$$H\boldsymbol{c}^T = 0,$$

where $\mathbf{c}^T$ is the transpose of the codeword vector $\mathbf{c}$. This equation ensures that $\mathbf{c}$ lies within the code defined by the generator matrix G.

3.2 Cyclic and Polynomial Codes

Cyclic codes are a subset of linear codes in which the cyclic shift of any codeword $\mathbf{c} \in C$ is also in C. This property makes cyclic codes particularly useful in error correction, as they allow for efficient encoding and decoding algorithms.

Definition 9. Cyclic Code *Let C be a linear code with codewords of length n over a Galois field $GF(p)$. The code C is a cyclic code if, for every codeword $\boldsymbol{c} = (c_1, c_2, \ldots, c_{n-1}, c_n) \in C$, the codeword obtained by a cyclic shift, $\boldsymbol{c}' = (c_n, c_1, \ldots, c_{n-1})$, is also in C.*

The elements of a codeword $\boldsymbol{c} = (c_1, c_2, \ldots, c_n)$ in a cyclic code can be viewed as the coefficients of a polynomial

$$c(x) = c_1 + c_2 x + \cdots + c_n x^{n-1}.$$

Since every cyclic code can be represented in polynomial form, it is also known as a polynomial code. These codes exist within the polynomial ring $GF(p)[x]$, which is the set of one-variable polynomials with coefficients in $GF(p)$.

Definition 10. Polynomial Ring *A polynomial ring $GF(p)[x]$ consists of polynomials in one variable, where the coefficients are elements from the finite field $GF(p)$. The set is equipped with two operations, polynomial addition $+$ and polynomial multiplication $(\cdot)$.*

While codewords in other linear codes can also be expressed as polynomials, this representation is particularly important for cyclic codes.

Definition 11. Polynomial Code *Let C be a linear code of length n over a Galois field $\mathrm{GF}(p)$, where each codeword $\mathbf{c} = (c_1, c_2, \ldots, c_n) \in C$ is represented by a polynomial*

$$c(x) = c_1 + c_2 x + \cdots + c_n x^{n-1} \in GF(p)[x].$$

The code C is called a polynomial code if there exists a fixed polynomial $g(x) \in GF(p)[x]$ such that each codeword polynomial $c(x) \in C$ can be written as

$$c(x) = q(x)g(x),$$

for some polynomial $q(x) \in GF(p)[x]$.

The polynomial $g(x)$ is called the generator polynomial. It is the lowest-degree polynomial that generates the code C of length n and also divides $x^n - 1$ over a given finite field $GF(p)$. The algebraic structure of these polynomial codes is naturally described within the quotient ring

$$GF(p)[x]/(x^n - 1),$$

where $(x^n - 1)$ is the ideal generated by $x^n - 1$, meaning that any multiple of $x^n - 1$ is equal to zero in the quotient ring.

Definition 12. Ideal *For a commutative ring R, a non-empty subset I, is called an ideal of R if $a - b \in I$ for all $a, b \in I$ and $a \cdot r = r \cdot a \in I$ for $a \in I$ and $r \in R$.*

The quotient ring is formed by considering all polynomials modulo the ideal, meaning that two polynomials are considered equivalent if their difference lies in the ideal. Specifically, in this context, computations proceed as usual with the added rule that x^n can be replaced by 1. The formal definition of a quotient ring, however, is beyond the scope of this paper, as the focus is on the application to error correcting codes rather than the abstract algebraic theory.

Since codewords are polynomials of degree less than n, addition and multiplication in this ring are performed modulo $x^n - 1$, making polynomial codes particularly suitable for cyclic structures. Formally, a polynomial code can be described as

$$C = \langle g(x) \rangle = \{q(x)g(x) \mid q(x) \in GF(p)[x]/(x^n - 1)\}.$$

This underlying structure of polynomial codes allows for efficient encoding and decoding through polynomial division. These properties are particularly useful in constructing error-correcting cyclic codes, such as BCH codes and Reed-Solomon codes.

3.3 BCH Codes

The **B**ose–**C**haudhuri–**H**ocquenghem (BCH) codes form a class of cyclic codes, and more specifically polynomial codes. In 1959 and 1960, BCH codes were independently discovered by Raj Chandra Bose, D. K. Ray-Chaudhuri, and Alexis Hocquenghem. These codes are specifically designed to detect and correct precise and more importantly, multiple errors, making them highly effective in error correction for digital communication systems and storage. As previously mentioned, the foundation of cyclic polynomial codes is the generator polynomial.

BCH codes are defined over a finite Galois field $GF(p^m)$, where p is a prime and m is a positive integer. The field $GF(p^m)$ is a finite field extension of $GF(p)$, with a generator α of the multiplicative group of nonzero elements in $GF(p^m)$. A common choice of p is 2, creating the binary field $GF(2)$ with elements $\{0, 1\}$. Note that this corresponds to choosing to $m = 1$. An extension of this field, for $m = 3$ is the given by $GF(2^3)$ with 8 elements

$$\{0,\ 1,\ x,\ x+1,\ x^2,\ x^2+1,\ x^2+x,\ x^2+x+1\}$$

The key feature of BCH codes is their ability to correct multiple errors by selecting a set of consecutive powers of a primitive element α in the field $GF(p^m)$, and using these powers as roots for the generator polynomial of the code. The generator polynomial of a BCH code is the lowest-degree monic polynomial whose roots are a set of consecutive powers of the generator α.

Definition 13. Minimal polynomial *The minimal polynomial $M_i(x)$ of α^i is the monic polynomial of smallest degree with coefficients in $GF(p)$ for which $M_i(\alpha^i) = 0$.*

Definition 14. BCH Codes *Let C be a cyclic code over a Galois field $GF(p^m)$ with a generator α. The code C is a BCH code if the generator polynomial $g(x)$ is given by*

$$g(x) = \text{lcm}(M_1(x), M_2(x), \ldots, M_{2v}(x)),$$

where $M_i(x)$ is the minimal polynomial of α^i over $GF(p^m)$, v is the maximum number of errors that can be corrected, and lcm denotes the least common multiple of the minimal polynomials.

As with all cyclic polynomial codes, any codeword polynomial $c(x)$ that is a multiple of the generator polynomial $g(x)$ is a valid codeword in C. This is because the generator polynomial always evaluates to zero at certain points, and thus, if the generator polynomial is part of the codeword, the entire codeword will evaluate to zero at those points. This property is an important aspect of decoding BCH code messages.

While BCH codes are effective for correcting multiple errors, they are somewhat limited in their flexibility, especially when dealing with more complex error correction needs. To overcome some of these limitations, Reed-Solomon codes offer enhanced error correction capabilities.

3.4 Reed-Solomon Codes

An important class of BCH codes is Reed-Solomon codes. They were initially invented by Irving S. Reed and Gustave Solomon in 1960 and later developed to incorporate the principles of BCH codes for greater efficiency. Reed-Solomon codes, like BCH codes, are typically defined over the finite field $GF(p^m)$, for a prime p and a positive integer m. The length of the codewords n has to satisfy $n \leq p^m$ and is usually chosen as $n = p^m - 1$ or $n = q^m$ for optimal efficiency. The length of the message, k, must be chosen so that $k < n$. A Reed-Solomon code is commonly denoted as $RS(n, k)$. The key distinction between BCH codes and their Reed-Solomon counterpart is that the number of correctable errors for a Reed-Solomon code is determined by the choice of n and k, not by the construction of the generator polynomial itself.

This paper focuses on cyclic Reed-Solomon codes, specifically the BCH version. These codes use a fixed generator polynomial, as defined below. However, the original Reed-Solomon codes are not necessarily cyclic, depending on the choice of evaluation points. In contrast, linear Reed-Solomon codes use a variable polynomial based on the message to be encoded, while the encoder and decoder share only a fixed set of evaluation points. The original theoretical decoding method involved generating potential polynomials from subsets of k out of n received values and selecting the most frequently occurring polynomial as the correct one. A process that was computationally impractical except for very simple cases.

Definition 15. Reed-Solomon Codes (BCH version) *Let C be a cyclic code over a Galois field $GF(p^m)$ with a generator α. The code C is a Reed-Solomon code if the generator polynomial $g(x)$ is given by*

$$g(x) = \prod_{i=1}^{t} (x - \alpha^i),$$

where t is the parity of the code.

Due to the definition of the generator polynomial $g(x)$, it evaluates to zero for all α^i when $i \in [1, t]$. Consequently, all valid codewords also evaluate to zero at these values due to the cyclic structure. As will be explained in greater detail in the next section, this property enables the correction of a specific number of errors. Specifically, up to $v = \left\lfloor \frac{t}{2} \right\rfloor$ errors can be corrected.

4 Encoding Reed-Solomon Codes

The BCH version of Reed-Solomon encoding is based on the generator polynomial $g(x)$. The construction of the generator polynomial and its use in creating codewords is important to ensure that the code is cyclic and can be corrected.

As previously mentioned, Reed-Solomon codes consists of four main parameters. The first is the order p^m of the Galois field, the second is a generator of that field α, the third is the length of the codewords n, also known as the block length, and finally, the length of the message k.

4.1 Message Encoding

The first step in encoding a message as a Reed-Solomon codeword is the construction of the generator polynomial $g(x)$. To create a cyclic code, the generator polynomial has $t = n - k$ consecutive powers of α as roots. This ensures that the polynomial evaluates to zero at these points. Since α is a generator of the Galois field, each value α^i, for $i \in [1, t]$, is unique.

Definition 16. Generator Polynomial *Let n be the length of the codewords in a cyclic code and k the length of the message. The generator polynomial $g(x)$ over a given Galois field $GF(p^m)$ is a polynomial of degree $t = n - k$ that is given by*

$$g(x) = \prod_{i=1}^{t}(x - \alpha^i)$$

for some generator α of the Galois field.

To encode a message of length k, a corresponding polynomial is constructed so that the coefficients of the polynomial represent the message.

Definition 17. Message Polynomial *The message polynomial for a message $\boldsymbol{m} = (m_1, m_2, ..., m_k)$ of length k is a polynomial of degree $k - 1$ defined as*

$$m(x) = \sum_{i=1}^{k} m_i x^{i-1}.$$

The goal when constructing the codeword is to ensure that it can be expressed as the product of a polynomial and the generator polynomial. As noted above, the generator polynomial evaluates to zero at α^i for $i \in [1, t]$. Due to the construction of $g(x)$, any correctly received codeword will also evaluate to zero at these points. Conversely, if any errors occur, only information about the errors will be preserved during evaluation.

Definition 18. Reed-Solomon encoding formula *Let C be a Reed-Solomon code with block length n, message length k and parity $t = n - k$. Let $g(x)$ be the generator polynomial and $m(x)$ the message polynomial. The codeword $s(x)$ is given by:*

$$s(x) = m(x)x^t - (m(x)x^t \pmod{g(x)}).$$

Theorem 3. Reed-Solomon Codeword Structure *Let $s(x)$ be a Reed-Solomon encoded message of length n with parity t. The codeword $s(x)$ can be expressed as a multiple of the generator polynomial $g(x)$ such that*

$$s(x) = q(x)g(x),$$

where $q(x)$ is the quotient when dividing $m(x)x^t$ by $g(x)$.

Proof. First, note that $m(x)x^t \pmod{g(x)}$ is equal to the remainder $r(x)$ when $m(x)x^t$ is divided by the generator polynomial $g(x)$. This gives the equation

$$m(x)x^t = q(x)g(x) + r(x),$$

for some quotient $q(x)$. Thus, the codeword can be expressed as

$$\begin{aligned} s(x) &= m(x)x^t - (m(x)x^t \pmod{g(x)}) \\ &= m(x)x^t - r(x) \\ &= m(x)x^t - (m(x)x^t - q(x)g(x)) \\ &= q(x)g(x). \end{aligned}$$

□

Using the Reed-Solomon encoding formula results in a polynomial of degree $n-1$. The coefficients of this polynomial form the final codeword that is transmitted. Due to the construction of the codeword, the last k elements, which are the coefficients of the highest-degree terms, correspond to the original message. Note that if the polynomial is viewed in reverse, the first k elements correspond to the message, as described in Section 3.1. However, due to the structure of the parity-check matrix for Reed-Solomon codes, it is more practical to reverse the codeword, as will be shown in the decoding section.

4.2 Encoding Example

In this section, a simple example of the encoding procedure described above is presented using a Reed Solomon code with block length $n = 7$, message length $k = 3$ and parity $t = 7-3 = 4$. To avoid large numbers, the Galois field is chosen as $GF(11)$ and the generator $\alpha = 2$. Using previously mentioned definitions, the generator polynomial is calculated in the following way,

$$g(x) = \prod_{i=1}^{4}(x - 2^i) = (x-2)(x-4)(x-8)(x-5) = 1 + 8x + 5x^2 + 3x^3 + x^4.$$

Assuming the message $\boldsymbol{m} = (1, 1, 2)$, the message polynomial is given by

$$m(x) = 1 + x + 2x^2.$$

Polynomial long division is used to calculate the remainder $r(x)$ when dividing $m(x)x^t$ by $g(x)$.

$$
\begin{array}{rl}
 & 2x^2+6x+6 \\
\hline
2x^6+x^5+x^4 & \left|\underline{x^4+3x^3+5x^2+8x+1}\right. \\
\underline{-(2x^6+6x^5+10x^4+5x^3+2x^2)} & \\
6x^5+2x^4+6x^3+9x^2 & \\
\underline{-(6x^5+7x^4+8x^3+4x^2+6x)} & \\
6x^4+9x^3+5x^2+5x & \\
\underline{-(6x^4+7x^3+8x^2+4x+6)} & \\
2x^3+8x^2+x+5 &
\end{array}
$$

The codeword polynomial is then calculated as

$$
\begin{aligned}
s(x) &= m(x)x^t - r(x) \\
&= (2x^6+x^5+x^4) - (2x^3+8x^2+x+5) \\
&= 6+10x+3x^2+9x^3+x^4+x^5+2x^6.
\end{aligned}
$$

Note that the codeword can be factored into the generator polynomial (x) and a quotient $q(x)$, such that

$$
s(x) = (6+6x+2x^2)(1+8x+5x^2+3x^2+x^4) = q(x)g(x),
$$

in agreement with Theorem 3. The final codeword that the sender transmits consists of the coefficients of the codeword polynomial $\boldsymbol{s} = (6, 10, 3, 9, 1, 1, 2)$. The four additional symbols appended to the original message represent the parity of the codeword, corresponding to $t = 4$.

5 Decoding Reed-Solomon Codes

During the transmission of a Reed-Solomon codeword, encoded as shown above, errors may occur. Due to the structure of the codeword, the key to identifying these errors is to evaluate the codeword polynomial at the roots of the generator polynomial, given by α^i for $i \in [1, t]$. The evaluation of the codeword at each root is called a syndrome and serves as the basis for error correction.

5.1 Essential Concepts in Error Correction

To understand the different methods for decoding Reed-Solomon codewords, it is essential to first grasp some key foundational concepts. This includes the calculation of syndromes and their function, as well as an understanding of the structure of received errors. Additionally, the limitations of error correction are discussed.

5.1.1 Syndromes

The received codeword is interpreted as a polynomial in the same way as it was created. However, during transmission, some coefficients change. These changes can be written as a polynomial called the error polynomial.

Definition 19. Error Polynomial *Let $s(x)$ be a Reed-Solomon codeword polynomial of degree $n-1$, corresponding to the codeword vector $\boldsymbol{s}$. Let $r(x)$ be the polynomial representation of the received codeword $\boldsymbol{r}$, with degree at most $n-1$. The error polynomial $e(x)$, with coefficients $\boldsymbol{e} = (e_1, e_2, ..., e_n)$, represents the difference between the original and the received codeword given by*

$$e(x) = r(x) - s(x),$$

where the degree of the polynomial depends on the locations of the errors.

As defined in the previous section, the codeword polynomial is constructed as a product of the generator polynomial $g(x)$ and some quotient $q(x)$. When errors occur during transmission, the received codeword polynomial is therefore given by

$$r(x) = s(x) + e(x) = q(x)g(x) + e(x).$$

Each syndrome S_i is computed by evaluating $r(x)$ at α^i. Since the generator polynomial equals zero when evaluated at α^i for $i \in [1, t]$ the syndromes depend only on the error polynomial at these points. Thus,

$$S_i = r(\alpha^i) = s(\alpha^i) + e(\alpha^i) = q(\alpha^i)g(\alpha^i) + e(\alpha^i) = e(\alpha^i),$$

for $i \in [1, t]$. This results in t syndromes. Note that received codewords without errors result in all syndromes being zero.

Definition 20. Syndrome *Let $\boldsymbol{r}$ be a received Reed-Solomon codeword of length n and parity t. The syndrome S_i for $i \in 1, t$ is given by*

$$S_i = e(\alpha^i).$$

An effective method to calculate the syndromes is by utilizing the Vandermonde matrix. The Vandermonde matrix is commonly used to evaluate a polynomial of degree n at multiple values, given by the vector of evaluation points $\boldsymbol{x} = (x_1, x_2, \ldots, x_m)$. By multiplying the coefficient vector of a polynomial with the Vandermonde matrix constructed from these evaluation points, all evaluations of the polynomial can be computed simultaneously, resulting in a vector of values corresponding to the polynomial evaluated at each point in $\boldsymbol{x}$.

Definition 21. Vandermonde matrix *Given a sequence of distinct numbers $\boldsymbol{x} = (x_1, x_2, ..., x_m)$, the $m \times n$ Vandermonde matrix is given by*

$$V = \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^{n-1} \end{pmatrix}.$$

The t syndromes $\boldsymbol{S} = (S_1, S_2, \ldots, S_t)$, computed from the received codeword $\boldsymbol{r} = (r_1, r_2, \ldots, r_n)$ of length n, are obtained using a $t \times n$ Vandermonde matrix with $\boldsymbol{x} = (\alpha^1, \alpha^2, \ldots, \alpha^t)$. This method calculates all syndromes in a single step, by evaluating the error polynomial at multiple values, according to Definition 18.

$$\boldsymbol{S}^T = \boldsymbol{V}\boldsymbol{r}^T = \boldsymbol{V}\boldsymbol{e}^T = \begin{pmatrix} 1 & \alpha & \alpha^2 & \cdots & \alpha^{n-1} \\ 1 & \alpha^2 & \alpha^{2^2} & \cdots & \alpha^{2^{(n-1)}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha^t & \alpha^{t^2} & \cdots & \alpha^{t^{(n-1)}} \end{pmatrix} \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{pmatrix} = \begin{pmatrix} S_1 \\ S_2 \\ \vdots \\ S_t \end{pmatrix}.$$

5.1.2 Error Correction Limitations

Given the construction of the generator polynomial, only t syndromes can be computed. As a direct consequence, only a limited number of errors can be detected and corrected. Therefore, at least $n-v$ of the error polynomial coefficients $\boldsymbol{e} = (e_1, e_2, ..., e_n)$ must equal zero. Since the locations of the v nonzero error coefficients are unknown, each of these errors e_j for $j \in [1, v]$ can be described with a magnitude E_j and a location l_j, where $l_j \in [1, n]$.

Definition 22. Error magnitude *Let $\boldsymbol{r}$ be a received Reed-Solomon codeword containing v errors. The vector of nonzero error magnitudes, $\boldsymbol{E} = (E_1, E_2, \ldots, E_v)$, represents the size of each of the v deviations from the original, transmitted codeword.*

Definition 23. Error locations *Let r be a received Reed-Solomon codeword of length n with v errors. The error locations $\boldsymbol{l} = (l_1, l_2, ..., l_v)$ represents the location of the v deviations from the original codeword so that $l_j \in [1, n]$.*

Using this interpretation of the v errors, the t syndrome can be expressed as a function of errors magnitude and its location.

Theorem 4. Syndrome Representation *Let $\boldsymbol{r}$ be a received Reed-Solomon codeword with v errors. Let E_j be the error magnitude and l_j the error location of error j for $j \in [1, v]$. The syndrome $S_i = e(\alpha^i)$ for $i \in [1, t]$, can be expressed as*

$$S_i = \sum_{j=1}^{v} E_j (\alpha^i)^{l_j}.$$

Proof. Since the syndrome $S_i = e(\alpha^i)$ is the evaluation of the polynomial $e(x)$, it can be further expressed as

$$S_i = e_1 + e_2(\alpha^i) + e_3(\alpha^i)^2 + \cdots + e_n(\alpha^i)^{n-1}.$$

As previously mentioned, if v errors can be corrected, then at least $n - v$ coefficients of the error polynomial must be zero. The positions of the remaining

nonzero terms in the syndrome expression are referred to as the error locations, as defined in Definition 21, and correspond to the degrees of the polynomial terms. To further distinguish the nonzero coefficients from the original error polynomial, the magnitudes of the errors, denoted as E_j, are defined according to Definition 20. The syndrome can then be written using only nonzero terms as

$$S_i = E_1(\alpha^i)^{l_1} + E_2(\alpha^i)^{l_2} + \cdots + E_v(\alpha^i)^{l_v} = \sum_{j=1}^{v} E_j(\alpha^i)^{l_j},$$

when v errors has occurred. □

All error correcting codes has limitations regarding the amount of errors in a codeword that can be corrected. For the Reed-Solomon codes, the limits depend of the parity of the codewords and the definition of the syndromes.

Theorem 5. Maximum Error Correction *Let C be a Reed-Solomon code with codewords of length n with a message of length k and parity t. The maximum amount of errors that can be corrected is given by*

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

Proof. From the definitions above, the Reed-Solomon code generates t syndromes that each depend on the magnitude and location of the error. Assume that at most v errors has occurred and can be corrected. This generates a system of t equations

$$S_i = \sum_{j=1}^{v} E_j(\alpha^i)^{l_j} = E_1(\alpha^i)^{l_1} + ... + E_v(\alpha^i)^{l_v}$$

and $2v$ unknown variables E_j and l_j for $j \in [1, v]$. A unique solution to a system of equations is possible, but not guaranteed, if and only if the number of equations is greater than or equal to the number of variables. To solve the system of equations that the syndromes make up, the following must apply

$$t \geq 2v \quad \iff \quad v \leq \frac{t}{2}.$$

Since v has to be a positive integer, the floor function of the expression defines the maximum amount of errors. Hence,

$$v_{max} = \left\lfloor \frac{t}{2} \right\rfloor \quad \text{and} \quad v \leq \left\lfloor \frac{t}{2} \right\rfloor.$$

□

5.1.3 Error localization

The final key aspect in understanding different decoding methods is error localization. To determine the magnitudes of the errors, one must first identify their locations, l_j. This is achieved by defining the error locator polynomial $\Lambda(x)$. The roots of this polynomial correspond to α^{-l_j}, allowing the error locations to be found by solving $\Lambda(x) = 0$.

Definition 24. Error locator polynomial *For a received Reed-Solomon codeword with v errors at locations l_j for $j \in [1, v]$ the error locator polynomial $\Lambda(x)$ is given by*

$$\Lambda(x) = \prod_{j=1}^{v}(1 - x\alpha^{l_j}) = 1 + \sum_{i=1}^{v}\Lambda_i x^i.$$

There are several methods for determining the error locator polynomial and its coefficients. The following section describes how the Direct and Euclidean methods are used to compute both the error locations and their magnitudes.

5.2 Direct Method

The direct method of Reed-Solomon decoding utilizes the definitions of the syndromes and the error locator polynomial to solve the system of equations in two main steps. First, the coefficients of the error locator polynomial are calculated. This is done by solving a system of linear equations, where the equations are derived from the syndromes, under the assumption that v errors have occurred.

Theorem 6. Syndrome-Derived Error Locator Polynomial *Let $\Lambda(x)$ be the error locator polynomial with roots α^{-l_j} for $j \in [1, v]$, and let C be a code that corrects v errors and generates $2v$ syndromes $S_i = \sum_{j=1}^{v} E_j(\alpha^i)^{l_j}$. The coefficients Λ_i of the error locator polynomial $\Lambda(x)$ can be obtained by solving the system of linear equations*

$$S_{v+k} + \Lambda_1 S_{v+k-1} + ... + \Lambda_{v-1} S_{k+1} + \Lambda_v S_k = 0,$$

for $k \in [1, v]$, with the corresponding augmented matrix

$$\begin{pmatrix} S_1 & S_2 & \cdots & S_v \\ S_2 & S_3 & \cdots & S_{v+1} \\ \vdots & \vdots & \ddots & \vdots \\ S_v & S_{v+1} & \cdots & S_{2v-1} \end{pmatrix} \begin{pmatrix} \Lambda_v \\ \Lambda_{v-1} \\ \vdots \\ \Lambda_1 \end{pmatrix} = \begin{pmatrix} -S_{v+1} \\ -S_{v+2} \\ \vdots \\ -S_{2v} \end{pmatrix}.$$

Proof. Given the construction of the error location polynomial $\Lambda(x)$, the roots of the polynomial are known. Thus,

$$\Lambda(\alpha^{-lj}) = 0$$

for $j \in [1, v]$. Furthermore, the syndromes can be expressed as

$$S_i = \sum_{j=1}^{v} E_j(\alpha^i)^{l_j} = \sum_{j=1}^{v} E_j(\alpha^{l_j})^i,$$

where E_j is the error magnitude and l_j the error location.

Both sides of the expression $\Lambda(\alpha^{-l_j}) = 0$ is first multiplied by the sum of the error magnitude $\sum_{j=1}^{v} E_j$.

$$\begin{aligned} 0 &= \Lambda(\alpha^{-l_j}) \\ &= 1 + \Lambda_1(\alpha^{-l_j}) + ... + \Lambda_{v-1}(\alpha^{-l_j})^{v-1} + \Lambda_v(\alpha^{-l_j})^v \\ &= \sum_{j=1}^{v} E_j + \Lambda_1 \sum_{j=1}^{v} E_j(\alpha^{-l_j}) + ... + \Lambda_{v-1} \sum_{j=1}^{v} E_j(\alpha^{-l_j})^{v-1} + \Lambda_v \sum_{j=1}^{v} E_j(\alpha^{-l_j})^v. \end{aligned}$$

To obtain the syndromes from the expression, each term is further multiplied by $\alpha^{l_j(v+1)}$ as shown below.

$$\begin{aligned} 0 &= \sum_{j=1}^{v} E_j + \Lambda_1 \sum_{j=1}^{v} E_j(\alpha^{-l_j}) + ... + \Lambda_{v-1} \sum_{j=1}^{v} E_j(\alpha^{-l_j})^{v-1} + \Lambda_v \sum_{j=1}^{v} E_j(\alpha^{-l_j})^v \\ &= \sum_{j=1}^{v} E_j(\alpha^{l_j})^{v+1} + \Lambda_1 \sum_{j=1}^{v} E_j(\alpha^{l_j})^v + ... + \Lambda_{v-1} \sum_{j=1}^{v} E_j(\alpha^{-l_j})^2 + \Lambda_v \sum_{j=1}^{v} E_j(\alpha^{-l_j}) \\ &= S_{v+1} + \Lambda_1 S_v + ... + \Lambda_{v-1} S_2 + \Lambda_v S_1. \end{aligned}$$

Note that the final equation corresponds to $k = 1$ and by continuing to multiply the expression by α^{l_j}, the expression is shifted. This process results in the remaining equations corresponding to $k = 1, 2, \ldots, v$, ultimately leading to the final system of linear equations represented by the augmented matrix described above.

□

The error locator polynomial, expressed with coefficients Λ_i, is a polynomial of degree v. Normally, there is no general method for solving polynomials of a higher degree. However, because the polynomial is constructed over a Galois field $GF(p^m)$, brute force can be used to find the solutions. Using trial and error, $\Lambda(x)$ is evaluated at α^{-l_j} for $l_j \in [0, n-1]$ until v solutions are found.

The final step in decoding the message is computing the magnitude of each

error. Since the error locations are no longer unknown, the initial definition of the syndromes now forms a linear equation system that can be solved.

$$\begin{pmatrix} \alpha^{l_1} & \alpha^{l_2} & \cdots & \alpha^{l_v} \\ \alpha^{2l_1} & \alpha^{2l_2} & \cdots & \alpha^{2l_v} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha^{vl_1} & \alpha^{vl_2} & \cdots & \alpha^{vl_v} \end{pmatrix} \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_v \end{pmatrix} = \begin{pmatrix} S_1 \\ S_2 \\ \vdots \\ S_v \end{pmatrix}.$$

The calculations above leave two vectors, $\boldsymbol{l} = (l_1, l_2, ..., l_v)$ and $\boldsymbol{E} = (E_1, E_2, ..., E_v)$. By matching each error magnitude with its corresponding location, the error vector $\boldsymbol{e} = (e_1, e_2, \ldots, e_n)$ is obtained. Subtracting the error vector from the received codeword $\boldsymbol{r} = (r_1, r_2, ..., r_n)$, the original codeword $\boldsymbol{s} = (s_1, s_2, ..., s_n)$ is obtained. Note that the actual message is given only by the last k elements of $\boldsymbol{s}$.

5.2.1 Direct Decoding Example

In this section, the previously encoded message, as shown in section 4.2, will be decoded using the Direct method over the Galois field $GF(11)$. Assuming that two errors have occurred, a possible received codeword is $\boldsymbol{r} = (6, 1, 3, 9, 7, 1, 2)$. First, the four syndromes are computed using the 4×7 Vandermonde matrix as follows

$$\boldsymbol{S}^T = \boldsymbol{V}\boldsymbol{r}^T = \begin{pmatrix} 1 & 2 & 4 & 8 & 5 & 10 & 9 \\ 1 & 4 & 5 & 9 & 3 & 1 & 4 \\ 1 & 8 & 9 & 6 & 4 & 10 & 3 \\ 1 & 5 & 3 & 4 & 9 & 1 & 5 \end{pmatrix} \begin{pmatrix} 6 \\ 1 \\ 3 \\ 9 \\ 7 \\ 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 \\ 4 \\ 7 \\ 9 \end{pmatrix}.$$

The lambda values Λ_i are then calculated by solving the linear equation system

$$\begin{pmatrix} S_1 & S_2 \\ S_2 & S_3 \end{pmatrix} \begin{pmatrix} \Lambda_2 \\ \Lambda_1 \end{pmatrix} = \begin{pmatrix} -S_3 \\ -S_4 \end{pmatrix} \quad \longrightarrow \quad \begin{pmatrix} 1 & 4 \\ 4 & 7 \end{pmatrix} \begin{pmatrix} \Lambda_2 \\ \Lambda_1 \end{pmatrix} = \begin{pmatrix} 4 \\ 2 \end{pmatrix}.$$

The error location polynomial coefficients, $\Lambda_1 = 4$ and $\Lambda_2 = 10$ are obtained and by adding the constant term 1, the error location polynomial is given by

$$\Lambda(x) = 10x^2 + 4x + 1.$$

By trial and error, the roots 2^{-l_j} of the polynomial $\Lambda(x)$ are found such that

$$2^{-l_1} = 6 \quad \text{and} \quad 2^{-l_2} = 9.$$

Solving for the locations, $\boldsymbol{l} = (l_1, l_2) = (1, 4)$. The final step is computing the magnitude of the errors at location 1 and 4. This is done by solving the linear system of equations

$$\begin{pmatrix} \alpha^{l_1} & \alpha^{l_2} \\ \alpha^{2l_1} & \alpha^{2l_2} \end{pmatrix} \begin{pmatrix} E_1 \\ E_2 \end{pmatrix} = \begin{pmatrix} S_1 \\ S_2 \end{pmatrix} \quad \longrightarrow \quad \begin{pmatrix} 2 & 5 \\ 4 & 3 \end{pmatrix} \begin{pmatrix} E_1 \\ E_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 4 \end{pmatrix}.$$

The resulting error magnitudes is given by $\boldsymbol{E} = (E_1, E_2) = (2, 6)$. Matching the error magnitudes with their locations, results in the error vector $\boldsymbol{e} = (0, 2, 0, 0, 6, 0, 0)$. Subtracting the error from the received codeword $\boldsymbol{r} = (6, 1, 3, 9, 7, 1, 2)$, the correct and final codeword is given by

$$\boldsymbol{s} = \boldsymbol{r} - \boldsymbol{e} = (6, 1, 3, 9, 7, 1, 2) - (0, 2, 0, 0, 6, 0, 0) = (6, 10, 3, 9, 1, 1, 2),$$

where the actual message is given by the last k elements, $\boldsymbol{m} = (1, 1, 2)$.

5.3 Euclidean Decoder

In 1975, Yasuo Sugiyama discovered a method to find the locations and magnitudes of the errors, called the Euclidean method [4]. The Euclidean method utilizes the Extended Euclidean algorithm to solve the Key Equation and find the error location polynomial $\Lambda(x)$ and the error magnitude polynomial $\Omega(x)$.

The Key Equation is constructed using the error location polynomial $\Lambda(x)$, the error magnitude $\Omega(x)$ polynomial and the syndrome polynomial $S(x)$. The syndrome polynomial contains $2v$ syndromes. If the parity t of the codeword is an odd number, the last syndrome S_t is excluded.

Definition 25. Syndrome polynomial *Let C be a Reed-Solomon code that corrects at most v errors and generates $2v$ syndromes. The syndrome polynomial $S(x)$ is a polynomial of degree $2v - 1$ given by*

$$S(x) = \sum_{i=1}^{2v} S_i x^{i-1}.$$

Note, that by substituting the expression for the syndromes from Theorem 4, the syndrome polynomial can be written as

$$S(x) = \sum_{i=1}^{2v} \sum_{j=1}^{v} E_j (\alpha^i)^{l_j} x^{i-1}.$$

The Key Equation describes the relationship between the syndrome polynomial $S(x)$, the error locator polynomial $\Lambda(x)$ and the magnitude polynomial $\Omega(x)$, such that $\Omega(x)$ is the greatest common devisor of $S(x)$ and x^{2v}.

Definition 26. Error magnitude polynomial *Let $S(x)$ be the syndrome polynomial and $\Lambda(x)$ the error locator polynomial for a code where at most v errors can be corrected. The error magnitude polynomial $\Omega(x)$ is given by*

$$\Omega(x) = S(x)\Lambda(x) \bmod x^{2v}.$$

The purpose of the Key Equation is to find both the error locator polynomial and the error magnitude polynomial, as they contain all the information needed to compute the error vector $\boldsymbol{e}$. This equation can be solved, meaning that $Q(x)$ and $\Lambda(x)$ can be determined, using the Extended Euclidean Algorithm.

5.3.1 Extended Euclidean Algorithm

The Euclidean algorithm provides a method to compute the greatest common divisor (GCD) of two integers. Bézout's identity states that for two integers a and b with a greatest common divisor c, there exists two integers x and y so that

$$ax + by = c.$$

The Extended Euclidean algorithm not only solves for the greatest common devisor c but also the Bézout coefficients x and y. The algorithm can also be applied to polynomials.

Regardless of whether integers or polynomials are used, the algorithm follows the same steps, except for the stopping condition. There are four parameters that are updated at each step i of the algorithm, with starting point $i = 1$. The quotient q_{i-1}, the remainder r_i and the Bézout coefficients s_i and t_i. The initial setup is as follows

$$\begin{array}{lll} r_0 = a & s_0 = 1 & t_0 = 0 \\ r_1 = b & s_1 = 0 & t_1 = 1. \end{array}$$

At each step, r_1 is updated to r_0 and the remainder when dividing r_0 with r_1 becomes the new r_1, following the standard Euclidean algorithm. The remaining values are updated in the same way, according to

$$\begin{aligned} r_{i+1} &= r_{i-1} - q_{i+1}r_i \\ s_{i+1} &= s_{i-1} - q_{i+1}s_i \\ t_{i+1} &= t_{i-1} - q_{i+1}t_i, \end{aligned}$$

until $r_{i+1} = 0$.

Example 1. Extended Euclidean Algorithm
To find the greatest common divisor (GCD) as well as the Bézout coefficients for two integers $a = 122$ and $b = 52$, the Extended Euclidean Algorithm is applied to solve the equation

$$122x + 52y = GCD(122, 52).$$

At the stopping point i, $s_i = x$, $t_i = y$ and $r_i = GCD(122, 52)$. Using the iterative pattern above, for every step i, the values q_i, r_i, s_i and t_i are updated in the following way

i	q_i	r_i	s_i	t_i
0		122	1	0
1		52	0	1
2	2	$122 - 2 \cdot 52 = 18$	$1 - 2 \cdot 0 = 1$	$0 - 2 \cdot 1 = -2$
3	2	$52 - 2 \cdot 18 = 16$	$0 - 2 \cdot 1 = -2$	$1 - 2 \cdot -2 = 5$
4	1	$18 - 1 \cdot 16 = 2$	$1 - 1 \cdot -2 = 3$	$-2 - 1 \cdot 5 = -7$

Since r_5 would equal zero, the stopping point is $i = 4$. At this point $r_r = 2$, $s_4 = 3$ and $t_4 = -7$. The greatest common divisor is thereby given as $GCD(122, 52) = 2$ and the Bézout coefficients coefficients equal $x = 3$ and $y = -7$, showing that

$$122 \cdot 3 - 52 \cdot 7 = 2.$$

When the algorithm is applied to finding $\Lambda(x)$ and $\Omega(x)$, only the added parameter t is needed. The initial setup is then

$$\begin{array}{rclrcl} r_0(x) & = & x^{2v} & t_0(x) & = & 0 \\ r_1(x) & = & S(x) & t_1(x) & = & 1. \end{array}$$

When working with polynomials, the stopping criteria is instead determined by the degree of the polynomial $r_i(x)$, which equals v. At this stopping point, $t_i(x) = k\Lambda(x)$ for some integer k. To ensure that the constant coefficient of $\Lambda(x)$ is 1, as per the definition, $k\Lambda(x)$ is divided by the constant coefficient of $t_i(x)$. The same procedure is applied to the error magnitude polynomial, resulting in

$$\Lambda(x) = \frac{t_i(x)}{t_i(0)} \quad \text{and} \quad \Omega(x) = \frac{r_i(x)}{t_i(0)}.$$

The roots of $\Lambda(x)$, and consequently the error locations, are determined using the brute-force approach, as described in Section 5.2.

5.3.2 Forney's Algorithm

Once the error magnitude and location polynomials have been computed, Forney's algorithm, named after George David Forney Jr in 1965, is applied to determine the magnitude of each error [5]. For each error location l_j, the error

magnitude polynomial $\Omega(x)$ and the derivative of the error location polynomial, $\Lambda'(x)$, are evaluated at α^{-l_j}. The magnitude is then given by the negative quotient of $\Omega(x)$ and $\Lambda'(x)$.

Theorem 7. Forney's algorithm *Let $S(x)$ be the syndrome polynomial, $\Lambda(x)$ the error locator polynomial with roots α^{-l_j} and $\Omega(x)$ the magnitude polynomial so that*

$$\Omega(x) = S(x)\Lambda(x) \bmod x^{2v},$$

where at most v errors can be corrected. The error magnitude E_j at error location $l_j \in [0, n-1]$ is given by

$$E_j = \frac{-\Omega(\alpha^{-l_j})}{\Lambda'(\alpha^{-l_j})}.$$

Proof. Following the definitions above, the syndrome polynomial is given by

$$S(x) = \sum_{i=1}^{2v} S_i x^{i-1} = \sum_{i=1}^{2v} \sum_{j=1}^{v} E_j (\alpha^i)^{l_j} x^{i-1},$$

and the error locator polynomial is defined as

$$\Lambda(x) = \prod_{k=1}^{v} (1 - \alpha^{l_k} x).$$

The coefficients of the magnitude polynomial $\Omega(x)$ is obtained by substituting the expressions of the known polynomials into the key equation

$$\Omega(x) = \sum_{i=1}^{2v} \sum_{j=1}^{v} E_j (\alpha^i)^{l_j} x^{i-1} \prod_{k=1}^{v} (1 - \alpha^{l_k} x) \mod x^{2v}.$$

Since the error locator polynomial $\Lambda(x)$ has degree v, the key equation can be rewritten as follows:

$$\begin{aligned}
\Omega(x) &= \sum_{i=1}^{2v} \sum_{j=1}^{v} E_j (\alpha^i)^{l_j} x^{i-1} \prod_{k=1}^{v} (1 - \alpha^{l_k} x) && \bmod x^{2v} \\
&= \sum_{j=1}^{v} E_j \alpha^{l_j} \sum_{i=1}^{2v-1} (\alpha^{l_j} x)^i \prod_{k=1}^{v} (1 - \alpha^{l_k} x) && \bmod x^{2v} \\
&= \sum_{j=1}^{v} E_j \alpha^{l_j} (1 - \alpha^{l_j} x) \sum_{i=1}^{2v-1} (\alpha^{l_j} x)^i \prod_{\substack{k=1 \\ k \neq j}}^{v} (1 - \alpha^{l_k} x) && \bmod x^{2v} \\
&= \sum_{j=1}^{v} E_j \alpha^{l_j} (1 - \alpha^{l_j} x) \frac{1-(\alpha^{l_j} x)^{2v}}{1-(\alpha^{l_j} x)} \prod_{\substack{k=1 \\ k \neq j}}^{v} (1 - \alpha^{l_k} x) && \bmod x^{2v} \\
&= \sum_{j=1}^{v} E_j \alpha^{l_j} (1 - (\alpha^{l_j} x)^{2v}) \prod_{\substack{k=1 \\ k \neq j}}^{v} (1 - \alpha^{l_k} x) && \bmod x^{2v} \\
&= (\sum_{j=1}^{v} E_j \alpha^{l_j} - x^{2v} \sum_{j=1}^{v} E_j \alpha^{2l_j}) \prod_{\substack{k=1 \\ k \neq j}}^{v} (1 - \alpha^{l_k} x) && \bmod x^{2v} \\
&= \sum_{j=1}^{v} E_j \alpha^{l_j} \prod_{\substack{k=1 \\ k \neq j}}^{v} (1 - \alpha^{l_k} x).
\end{aligned}$$

Evaluation of the magnitude polynomial at $x = \alpha^{-l_j}$ gives

$$\Omega(\alpha^{-l_j}) = \sum_{j=1}^{v} E_j \alpha^{l_j} \prod_{\substack{k=1 \\ k \neq j}}^{v} (1 - \alpha^{l_k} \alpha^{-l_j}) = E_j \alpha^{l_j} \prod_{\substack{k=1 \\ k \neq j}}^{v} (1 - \alpha^{l_k - l_j}).$$

Furthermore, the derivative of the error locator polynomial is calculated using the generalized product rule:

$$\Lambda'(x) = \sum_{j=1}^{v} -\alpha^{l_j} \prod_{\substack{k=1 \\ k \neq j}}^{v} (1 - \alpha^{l_k} x).$$

The evaluation of the derivative at $x = \alpha^{-l_j}$ gives

$$\Lambda'(\alpha^{-l_j}) = \sum_{j=1}^{v} -\alpha^{l_j} \prod_{\substack{k=1 \\ k \neq j}}^{v} (1 - \alpha^{l_k} \alpha^{-l_j}) = -\alpha^{l_j} \prod_{\substack{k=1 \\ k \neq j}}^{v} (1 - \alpha^{l_k - l_j}).$$

The evaluated polynomials yield the final expression for computing the error magnitude E_j at error location $l_j \in [0, n-1]$

$$\frac{-\Omega(\alpha^{-l_j})}{\Lambda'(\alpha^{-l_j})} = \frac{-E_j \alpha^{l_j} \prod_{\substack{k=1 \\ k \neq j}}^{v} (1 - \alpha^{l_k - l_j})}{-\alpha^{l_j} \prod_{\substack{k=1 \\ k \neq j}}^{v} (1 - \alpha^{l_k - l_j})} = E_j.$$

□

As previously described in Section 5.2, the final codeword is computed by matching ut the magnitude of each error with its location and subtracting that error vector from the received codeword.

5.3.3 Euclidean Decoding Example

In this section, the previously demonstrated decoding example will be performed using the Euclidean decoder instead. Once again, the received vector contains two errors, so that $\boldsymbol{r} = (6, 1, 3, 9, 7, 1, 2)$ and the order of the Galois field equal $p = 11$. From the calculations in the previous section, using the direct method, the syndromes are given by $\boldsymbol{S} = (1, 4, 7, 9)$. The syndrome polynomial is therefore given by

$$S(x) = 9x^3 + 7x^2 + 4x + 1.$$

Note that for a Reed-Solomon code with $n = 7$ and $k = 3$, at most $v = 2$ errors can be corrected. Thus, the key equation is given by

$$\Omega(x) = S(x)\Lambda(x) \mod x^4.$$

To solve for $\Lambda(x)$ and $\Omega(x)$, the Extended Euclidean algorithm is applied in two steps by iteratively updating the parameter polynomials as follows

i	$q_i(x)$	$r_i(x)$	$t_i(x)$
0		x^4	0
1		$9x^3 + 7x^2 + 4x + 1$	1
2	$5x + 1$	$6x^2 + 2x + 10$	$6x + 10$
3	$7x + 8$	$6x + 9$	$2x^2 + 3x + 9$

The two polynomials $r_3(x)$ and $t_3(x)$ are further normalized by dividing them by the constant term in $t_3(x)$, yielding the final error locator polynomial and error magnitude polynomial

$$\begin{aligned} \Lambda(x) &= \tfrac{t_3(x)}{t_3(0)} = \tfrac{2x^2+3x+9}{9} = 10x^2 + 4x + 1 \\ \Omega(x) &= \tfrac{r_3(x)}{t_3(0)} = \tfrac{6x+9}{9} = 8x + 1. \end{aligned}$$

The roots to the error locator polynomial $\Lambda(x)$ and the corresponding error locations are once again computed with brute force, resulting in $\boldsymbol{l} = (l_1, l_2) = (1, 4)$. Additionally, the derivative of the error locator polynomial is given by

$$\Lambda'(x) = 9x + 4.$$

Lastly, Forney's algorithm is applied for the two error locations $(l_1, l_2) = (1, 4)$, yielding the two error magnitudes

$$\begin{aligned} E_1 &= \tfrac{-\Omega(\alpha^{-l_1})}{\Lambda'(\alpha^{-l_1})} = \tfrac{-\Omega(2^{-1})}{\Lambda'(2^{-1})} = \tfrac{6}{3} = 2 \\ E_2 &= \tfrac{-\Omega(\alpha^{-l_2})}{\Lambda'(\alpha^{-l_2})} = \tfrac{-\Omega(2^{-4})}{\Lambda'(2^{-4})} = \tfrac{4}{8} = 6. \end{aligned}$$

Subtracting the error magnitudes $\boldsymbol{E} = (E_1, E_2) = (2, 6)$ from the received codeword at the error locations $\boldsymbol{l} = (l_1, l_2) = (1, 4)$ leaves the correct codeword $\boldsymbol{s} = (6, 10, 3, 9, 1, 1, 2)$.

6 Python Implementation

To compare the two decoding methods described in the previous section, both the encoding and decoding processes have been implemented in Python. Since

the operations are performed within a specific Galois field, the SymPy library has been used to support symbolic computations. Specifically, all polynomials are constructed using the SymPy.Poly function, which can create a polynomial within a certain domain, such as a Galois field. The complete program can be found in the appendix.

6.1 Method

The program consists of three main functions. One for encoding a given message and two for the different decoding methods. The arguments used in these functions, which define the environment, include the block length n, the message length k, the order of the Galois field P, where $P = p^m$, with a corresponding generator α and finally, the message or the received codeword, represented as a vector of coefficients.

The encoding function first constructs the generator and message polynomials within the Galois field $GF(P)$. These polynomials are then used to compute the coefficients of the codeword, which is returned as the output. To simulate transmission errors, an error vector is subsequently added to the codeword. Both methods for decoding the received codeword uses the same two functions to find the correct Vandermonde matrix as well as computing the syndromes. The remaining decoding is however different for the two methods.

When decoding the codeword using the Direct method, the first step is computing the coefficients Λ_i for the error locator polynomial. This is done by constructing the syndrome matrix and multiplying its inverse with the syndrome vector. However, the inverse only exists for the matrix created if the correct number of errors or fewer is assumed. Since the number of errors is unknown when a codeword is received, the function assumes that one error has occurred and increases this assumption by one until the syndrome matrix is no longer invertible. The last invertible solution contains the correct number of errors.

Once the coefficients have been computed, the roots of the error location polynomial are calculated, and the corresponding error locations are found using brute force. The final step before computing the original codeword is to determine the error magnitudes. Since there exists a unique solution, the matrix containing the error locations is invertible. Its inverse is multiplied by the first v syndromes, yielding the error magnitudes E_j.

The Euclidean method consists of two main functions: one using the Extended Euclidean algorithm and one using Forney's algorithm. To solve the Key Equation, the Extended Euclidean algorithm is applied exactly as described in Section 5.3.1. The two polynomials $r_i(x)$ and $t_i(x)$ are updated according to their definitions until the degree of $r_i(x)$ equals the maximum number of errors. Note that this method does not depend on the actual number of errors that have occurred. The two polynomials $\Lambda(x)$ and $\Omega(x)$ are then computed by dividing

$r_i(x)$ and $t_i(x)$ by $t_i(0)$. The roots of the error location polynomial are computed using the same function as in the Direct method. For each error location, Forney's algorithm is applied to find the corresponding magnitude.

To compare the efficiency of the two methods, the time required for each decoding function to compute the correct message is measured and analyzed. The values of the four main variables, n, k, and P, have been varied to assess their impact on the algorithm's efficiency. By systematically adjusting these parameters, an evaluation of the computational performance of both methods is conducted. The results provide insight into the trade-offs between limitations and speed, helping to determine which method is more suitable for different decoding scenarios.

6.2 Results

The first variables that was examined was the order P of the Galois field and the choice of the generator α. Since the block length n and message length k need to be the same for a fair evaluation, they have been set to $n = 7$ and $k = 3$, and the received message includes two errors. Five different Galois fields have been examined, and each one has been tested with the smallest and largest possible generators, as well as the median of all possible generators. The results are shown in Table 1.

Table 1: Efficiency comparison of the Direct and Euclidean method for $n = 7$ and $k = 3$, varying P and α.

$\boldsymbol{\alpha}$	$\boldsymbol{P = 11}$	$\boldsymbol{P = 47}$	$\boldsymbol{P = 211}$	$\boldsymbol{P = 853}$	$\boldsymbol{P = 3413}$
Minimum					
Direct	0.0027	0.0031	0.0068	0.022	0.075
Euclidean	0.0028	0.0039	0.0071	0.021	0.075
Median					
Direct	0.0022	0.0039	0.0069	0.022	0.076
Euclidean	0.0029	0.0039	0.0071	0.020	0.075
Maximum					
Direct	0.0026	0.0038	0.0071	0.022	0.074
Euclidean	0.0028	0.0041	00.0075	0.021	0.074

From the results shown in Table 1, it is evident that the choice of α does not have any significant impact on the efficiency of either method. Nor is the efficiency between the two methods significantly different. It is further noted that the order of the Galois field has some impact on the efficiency, meaning that larger numbers yield longer computation times.

Secondly, the effect of the maximum number of errors $v_{\max}$, as well as the actual number of errors v, was examined. The block length was chosen as $n = 255$, since it is a very common block length when using Reed-Solomon codes. For example, $RS(255, 223)$ was the version used on the Voyager and Galileo missions by NASA [1]. The message length k was varied to change the maximum number of errors $v_{\max}$. Note from the results shown in Table 2 that the actual number of errors cannot exceed the maximum number of errors.

Table 2: Efficiency comparison of the Direct and Euclidean method for $n = 255$, $P = 257$ and $\alpha = 3$, varying v_{max} and v.

	$v = 2$	$v = 8$	$v = 16$	$v = 32$	$v = 64$	$v = 126$
$\underline{v_{max} = 2}$						
Direct	0.026					
Euclidean	0.025					
$\underline{v_{max} = 8}$						
Direct	0.066	0.086				
Euclidean	0.069	0.086				
$\underline{v_{max} = 16}$						
Direct	0.16	0.24	0.38			
Euclidean	0.15	0.29	0.19			
$\underline{v_{max} = 32}$						
Direct	0.23	0.26	0.40	1.75		
Euclidean	0.24	0.27	0.33	0.41		
$\underline{v_{max} = 64}$						
Direct	0.44	0.47	0.61	2.07	19.5	
Euclidean	0.46	0.51	0.62	0.77	1.02	
$\underline{v_{max} = 126}$						
Direct	0.86	0.88	1.02	2.47	20.8	261.7
Euclidean	0.91	1.03	1.19	1.48	2.04	2.93

The results shown in Table 2 and Figure 1 reveal a significant difference between the two methods as the number of errors increases. It is clear that increasing both the maximum number of errors and the actual number of errors leads to a decrease in efficiency. However, the effect varies somewhat between the two methods. For the Direct method, the number of actual errors appears to be the primary factor that increases computation time, with the increase following an

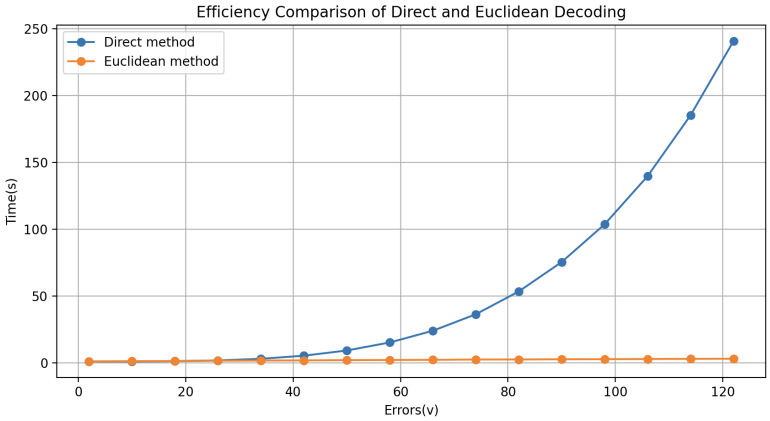

Figure 1: Efficiency comparison of Direct and Euclidean decoding for $n = 255$, $k = 3$, $P = 257$ and $\alpha = 3$. Computing time for v errors measured in second s.

exponential rather than a linear pattern as shown in Figure 1. When the number of actual errors is small, both methods exhibit similar behavior in terms of how efficiency decreases as the maximum number of errors increases. Overall, the Euclidean method proves to be more reliable when dealing with larger numbers of errors.

6.3 Discussion

As seen in the previous section, there are some differences in efficiency between the two methods. During the development of the program, multiple versions were created, and numerous improvements and modifications have influenced efficiency. Of course, it is possible that efficiency partially depends on the program itself and how it is structured. However, the most significant difference between the two methods is that the computation of the error location polynomial depends on the number of errors that have occurred, which is initially unknown. This uncertainty leads to a trial-and-error process to determine the correct values, resulting in increased computation time.

The most noticeable difference between the two methods concerns the Direct method's handling of a large number of errors. Upon closer inspection of the program, it became clear that a specific function significantly increases the computation time, the function responsible for computing the λ values. This occurs because the size of the matrix depends on the number of errors that have occurred. In the initial iterations of the program, the matrix was constructed under the assumption that the maximum possible number of errors had occurred. This resulted in very large matrices that gradually decreased in size until an invertible one was found, yielding the correct number of errors. As expected, computing the inverses of multiple large matrices is extremely time-

consuming. To improve efficiency, the initial matrix was instead constructed under the assumption that only one error had occurred, incrementally increasing until the correct solution was found. This approach significantly improved computational efficiency. However, due to the Direct method's dependence on an unknown number of errors, some unnecessary computation remains unavoidable.

Regarding the decrease in efficiency as the maximum number of errors increases, this can be explained by the fact that a significant portion of the computation in both methods depends on this value. A larger v consequently results in more syndromes and higher-degree polynomials, leading to increased computational complexity.

Another conclusion worth noting with respect to the results shown in Table 1 is that the optimal choice for the order of the Galois field should be as small as possible. Commonly, P and n are chosen such that $n = P - 1$ or $n = P$, which represents the maximum value for n. As confirmed by Table 1, this is the most reasonable choice from an efficiency perspective. In real-world applications—where the underlying technology utilizing Reed-Solomon codes is based on binary systems—the most common choice of field is $GF(2^8)$. This corresponds directly to an 8-bit byte, making it effective for integration with various digital systems [1]. A widely used configuration within this field, particularly at NASA, is $RS(255, 223)$. Although other choices for m are also common, the base of the field is almost always binary, to ensure compatibility with existing systems.

A somewhat surprising result is the striking similarity between the two methods, particularly in Table 1, despite their fundamentally different computational approaches. This presents a potential area for further investigation to better understand the underlying algorithms. Given that the Euclidean method for decoding Reed-Solomon codes is a more recently developed approach, its overall reliability, as demonstrated in the results, appears reasonable.

6.3.1 Challenges and Limitations

The biggest challenge in implementing and timing the two methods, was determining how much of the computation time depends on the implementation itself versus the actual decoding method. Therefore, a significant part of the development process involved optimizing the implementation to ensure maximum efficiency given the circumstances. In particular, working with symbolic mathematics is generally much more time-consuming than working with numerical computations. However, this is a crucial aspect of error-correcting codes that must be included in the program as well.

An additional challenge when interpreting the results has been the difficulty in finding research that shows similar outcomes. After an extensive search for

corroborative information, it appears to be virtually nonexistent. This may be a matter of time, as the original development of these methods occurred half a century ago. Since then, numerous new versions of the Euclidean decoder have been created to enhance efficiency. However, the Direct method may no longer be discussed because it is not applicable in real-world contexts due to its lack of scalability.

6.3.2 Use of Generative AI

When writing the program, the generative AI application ChatGPT has been used in two main areas. Primarily, AI has been utilized to quickly find and understand appropriate Python functions, especially within the SymPy library, such as the `Poly` or `zip` functions.

Furthermore, an attempt was made to identify the most inefficient computations within the program. A few useful suggestions provided by ChatGPT were implemented to improve the efficiency of some loops. However, most of the optimizations were achieved through manual and systematic analysis of computation time, followed by adjustments to the algorithm's implementation.

Lastly, generative AI was used to find the correct formatting of various mathematical expressions, structures, and symbols in LaTeX.

6.4 Conclusion

The comparison of the Direct and Euclidean decoding methods for Reed-Solomon codes reveals clear differences in computational efficiency, particularly as the number of errors increases. While both methods successfully decode messages, the Euclidean method proves to be more reliable and scalable when handling larger error counts due to its structured approach to solving the key equation. The Direct method, on the other hand, suffers from an inherent inefficiency caused by its dependence on an unknown number of errors, leading to a costly trial-and-error process.

The key difference that distinguishes the efficiency of the two methods is that the Euclidean method computes the basis for both the location and magnitude of the error in a single calculation, with lower computational complexity. By using the Extended Euclidean Algorithm, most of the computations are performed independently of the number of errors, which avoids uncertainty and repetition. This approach to determining the unknown error values bypasses the time-consuming process of solving linear equation systems, resulting in a more consistent computation time. In this method, the number of errors only affects the final computation step of Forney's algorithm. Regardless of how a linear system is solved, it is generally more time-consuming than the simple repeated division used in the Euclidean method. Specifically, repeatedly solving linear systems due to uncertainty regarding the number of errors is highly

inefficient.

These findings emphasize the importance of algorithm selection in error correction applications. While the Direct method may still be useful in scenarios with minimal errors, the Euclidean method is the preferable choice for larger and more complex cases. Ultimately, the results demonstrates that while both methods have their merits, the Euclidean method is better suited for practical applications requiring efficient and scalable error correction.

7 Generative AI disclosure

This paper was written with the assistance of generative AI, specifically the free version of OpenAI's ChatGPT-4. AI was used for formatting in LaTeX, correcting grammar, sentence structure and improving the Python implementation. A detailed description of the specific usage of generative AI can be found in Section 6.3.2. All AI-generated content has been reviewed and edited as needed, and I take full responsibility for the complete content of this paper.

References

[1] Irving S. Reed and Xuemin Chen, *Error-Control Coding for Data Networks*, Springer Science+Business Media, 1999.

[2] Jet Propulsion Laboratory, *208 Telemetry Data Decoding* (DSN Document 810-005, Rev. C), NASA, 2025. `https://deepspace.jpl.nasa.gov/dsndocs/810-005/208/208C.pdf` (Accessed 2025-04-28).

[3] NASA, *ISS Daily Summary Report – 2/06/2019*, NASA, 2019. `https://www.nasa.gov/blogs/stationreport/2019/02/06/iss-daily-summary-report-2-06-2019/` (Accessed 2025-04-18).

[4] R. E. Blahut, *Algebraic Codes for Data Transmission*, Cambridge University Press, 2003.

[5] Wikipedia contributors, "Forney algorithm," *Wikipedia, The Free Encyclopedia*, March 15, 2025. `https://en.wikipedia.org/w/index.php?title=Forney_algorithm&oldid=1280684112` (Accessed 2025-05-10).

[6] William A. Geisel, *Tutorial on Reed-Solomon Error Correction Coding* (NASA Technical Memorandum 102162), NASA, 1990. `https://ntrs.nasa.gov/api/citations/19900019023/downloads/19900019023.pdf` (Accessed 2025-04-18).

A Appendix

```
import sympy as sp
import time

def generator(n):

    """Function finds every generator of a finite field of
       order n."""

    generators = []
    for gen in range(2,n):
        x = set()
        for j in range(1, n):
            x.add(pow(gen, j) % n)
        if len(x) == n-1:
            generators.append(gen)
    return generators

def finding_error_locations(roots, a, mod):

    """Function returns the error locations l_i from list of
        polynomial roots a^(-l_i)."""

    positions = []
    for i in roots:
        for l in range(mod):
            if pow(pow(a, l, mod), -1, mod) == i:
                positions.append(l)
                break
    return positions

def inv_mod(A, mod):

    """Function returns the inverse of a matrix A if it
       exists."""

    matrix = sp.Matrix(A)
    try:
        return matrix.inv_mod(mod)
    except ValueError:
        return None

def vandermonde(n, t, a, mod):

    """Function returns the txn vandermonde matrix for a
       given alpha a."""

    vandermonde = sp.zeros(t, n)
```

```
    for i in range(t):
        for j in range(n):
            vandermonde[i, j] = pow(a, (i+1) * j, mod)
    return vandermonde

def syndrome(vander, r, mod):

    """Functions returns t syndromes by calculating V*r for
        a received vector r."""

    rec = sp.Matrix(r)
    return vander*rec % mod

def lam(s, mod):

    """Function constructs the syndrome matrix and vector.
        The lambda values are returned by multiplying the
        inverse (if it exists) of the matrix with the vector.
         If the inverse does not exist, the last successful
        result is returned."""

    v = 1
    if s[0] == 0:
        v += 1
    lam = None

    while v <= len(s) // 2:
        s_matrix = sp.Matrix([[s[i + j] for j in range(v)]
            for i in range(v)])
        s_vector = sp.Matrix([-s[v + i] for i in range(v)])
        try:
            inv_S = s_matrix.inv_mod(mod)
            lam = (inv_S * s_vector) % mod
        except ValueError:
            return lam
        v += 1
    return lam

def error_location(lam, mod, a):

    """Function creates the error location polynomial (lam_p
        ) from the lambda values (lam) and returns the error
        location, corresponding to the roots of the
        polynomial."""

    x = sp.symbols('x')
    lam_p = sp.Poly((sum(lam.tolist(), []) + [1]), x, domain
        =sp.GF(mod))
    roots = [x for x in range(mod) if lam_p(x) % mod == 0]
    positions = finding_error_locations(roots, a, mod)
```

```
    return positions

def error_magnitude(S, error_loc, mod, a):

    """Function creates error location matrix given the
       error locations and alpha a. Function returns the
       error magnitudes by multiplying inverse of error
       location matrix with the syndrome vector (length
       equals number of errors)."""

    l = len(error_loc)
    l_matrix = sp.zeros(l, l)
    for j in range(l):
        base = pow(a, error_loc[j])
        l_matrix[0, j] = base
        for i in range(1, l):
            l_matrix[i, j] = l_matrix[i - 1, j] * base
    mag = inv_mod(l_matrix, mod) * sp.Matrix(S[0:l]) % mod
    return mag

def error_correction(received, locations, magnitudes):

    """Function matches the error locations with their
       magnitude and creates error vector that is subtracted
        from the received vector to return the correct
       message."""

    error_vector = sp.zeros(len(received), 1)
    for i, mag in zip(locations, magnitudes):
        error_vector[i] = mag
    correct_message = [a - b for a, b in zip(received,
       error_vector)]
    return correct_message

def euklides(S, mod):
    """Function applies the extended euklides algorithm to
       solve the key equation and returns the error location
        polynomial (lamba) and the error magnitude
       polynomial (omega)."""

    x = sp.symbols('x')
    if len(S) % 2 != 0:
        S = S[:-1]
    t = len(S)
    r0 = sp.Poly(x ** t, x, domain=sp.GF(mod))
    r1 = sp.Poly(S[::-1], x, domain=sp.GF(mod))
    t0 = sp.Poly(0, x, domain=sp.GF(mod))
    t1 = sp.Poly(1, x, domain=sp.GF(mod))
    while sp.degree(r1) >= t/2:
        q, r = r0.div(r1)
```

```
        r0, r1 = r1, r
        t0, t1 = t1, t0 - q*t1
    omega = r1.div(t1.eval(0))[0]
    lamba = t1.div(t1.eval(0))[0]
    return omega, lamba

def error_forney(omega, lamba, mod, a):

    """Function computes the roots of the error location
       polynomial and finds the corresponding error
       locations and their magnitude using forney s formula
       ."""

    x = sp.symbols('x')
    roots_lamba = [x for x in range(mod) if lamba(x) % mod
       == 0]
    lamba_diff = lamba.diff(x)
    pos = finding_error_locations(roots_lamba, a, mod)
    mag = []
    for i in range(len(roots_lamba)):
        x = roots_lamba[i]
        mag.append(-omega(x)*pow(lamba_diff(x), -1, mod) %
            mod)
    return pos, mag

def encode(m, n, k, mod, a):

    """Function returns the codeword (length n) of a message
        (length k) within a galois filed of order m with
       generator a."""

    t = n-k
    x = sp.symbols('x')
    roots = [pow(a, i, mod) for i in range(1, t + 1)]
    gen = sp.expand(sp.prod(x - r for r in roots))
    gen = sp.Poly(gen, x, domain=sp.GF(mod))
    x_t = sp.Poly(x ** t, x, domain=sp.GF(mod))
    px_t = x_t * sp.Poly(m[::-1], x, domain=sp.GF(mod))
    _, rem = px_t.div(gen)
    message_sent = [(c % mod) for c in (px_t - rem).
       all_coeffs()[::-1]]
    return message_sent

def decode_direct(r, n, k, mod, a):

    """Function decodes a received codeword using the direct
        method and returns the correct message"""

    start_time1 = time.time()
    V = vandermonde(n, n-k, a, mod)
```

```
    S = syndrome(V, r, mod)
    if all(s == 0 for row in S.tolist() for s in row):
        print("Received message", r[(n - k):], "is correct!"
            )
        return None
    L = lam(S, mod)
    error_loc = error_location(L, mod, a)
    if not error_loc:
        print("Received message has more than", (n-k)//2, "
            errors and cannot be corrected.")
        return None
    error_mag = error_magnitude(S, error_loc, mod, a)
    correct_message = error_correction(r, error_loc,
        error_mag)[(n - k):]
    print("The corrected message is", correct_message, "!")
    timeer = time.time() - start_time1
    print(f"Time algorithm: {timeer:.3}")
    print("The corrected message is:")
    return correct_message

def decode_euklides(r, n, k, mod, a):

    """Function decodes a received codeword using the
        Euclidean method and returns the correct message."""

    start_time = time.time()
    V = vandermonde(n, n-k, a, mod)
    S = syndrome(V, r, mod)
    if all(s == 0 for row in S.tolist() for s in row):
        print("Received message", r[(n - k):], "is correct!"
            )
        return None
    E = euklides(S, mod)
    F = error_forney(E[0], E[1], mod, a)
    if not F[0]:
        print("Received message has more than", (n - k) //
            2, "errors and cannot be corrected.")
        return None
    correct_message = error_correction(r, F[0], F[1])[(n - k
        ):]
    print("The corrected message is", correct_message, "!")
    timeer = time.time() - start_time
    print(f"Time algorithm: {timeer:.3}")
    print("The corrected message is:")
    return  correct_message
```

```python
# Example usage:

n = 7
k = 3
P = 11
a = 2
m = [1, 1, 2]
error = [0, 2, 0, 0, 6, 0, 0]

print("Codeword:", encode(m, n, k, P, a))
r = [(a + b) % P for a, b in zip(encode(m, n, k, P, a),
    error)]
print("Received:", r)
print("\n--- Direct method ---")
print(decode_direct(r, n, k, P, a))
print("\n--- Euclidean method ---")
print(decode_euklides(r, n, k, P, a))


# Output:
```

```
Codeword: [6, 10, 3, 9, 1, 1, 2]
Received: [6, 1, 3, 9, 7, 1, 2]

--- Direct method ---
The corrected message is [1, 1, 2] !
Time algorithm: 0.00196
The corrected message is:
[1, 1, 2]

--- Euclidean method ---
The corrected message is [1, 1, 2] !
Time algorithm: 0.00193
The corrected message is:
[1, 1, 2]
```