McEliece Cryptosystem

Institute of Mathematics

University of Debrecen

Mathematics, BSc.

Amjad Ali

supervised by

Dr. Szabolcs Tengely

Debrecen, Hungary

April 2023

Abstract

The aim of this thesis is to explore the McEliece cryptosystem, a post-quantum cryptosystem based on linear codes. As modern cryptosystems based on number theoretic problems, such as integer factorization and discrete logarithms, are no longer considered secure with the advent of quantum computers, there has been a shift towards the study and development of cryptosystems based on other difficult mathematical problems. The McEliece cryptosystem, based on the NP-hard general decoding problem of linear codes, is one such candidate. In this thesis, we provide a simplified explanation of the McEliece cryptosystem, using SageMath interactive codes to provide a hands-on experience with its basic working principles. We begin by giving a brief introduction to cryptography, focusing on the widely used public-key cryptosystem, RSA, in Chapter 1. We then proceed to provide preliminary information on error-correcting codes in Chapter 2. Finally, we implement the McEliece cryptosystem using two distinct types of linear codes: Reed-Solomon in Chapter 3 and binary Goppa Codes in Chapter 4. We have included the SageMath codes used to develop the interactions in the appendix section, arranged in sequential order.

Acknowledgement

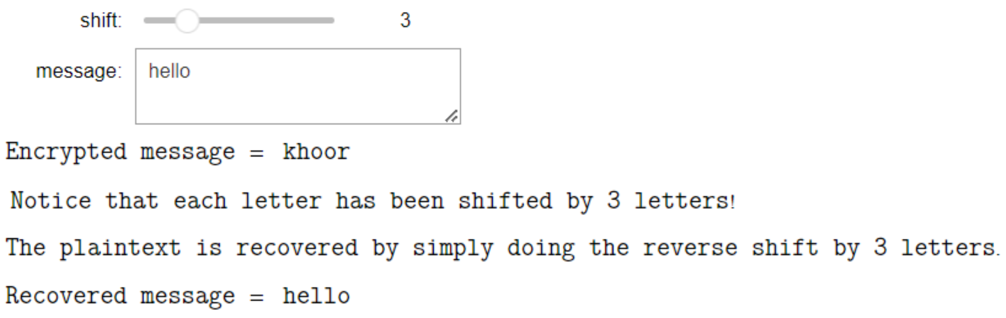
First, I would like to express my gratitude to my supervisor, Dr. Szabolcs Tengely, for admitting me under his supervision and providing constant guidance and assistance. Without his expert knowledge, I could not have completed my thesis. Moreover, I would like to thank my family, my girlfriend, and her family for their encouragement and support, which have been invaluable in pursuing my thesis. Finally, I am grateful to my friends who reviewed and provided feedback on my writing.

Contents

Chapter 1

Introduction

Cryptography is described as the study and practice of secure transmission of information in the presence of entities trying all possible means to disrupt the communication or gain unauthorized access to the information. In cryptography, these entities are commonly referred to as adversaries. The earliest recorded use of cryptography can be traced back to 1900 B.C. Egypt, where it was employed primarily for recreational purposes [1]. Over time, the use of cryptography evolved from mere amusement to essential functions such as safeguarding military and diplomatic communications. One such widely known use was by the Roman general Julius Caesar around 100 B.C. The scheme of encryption he used became famous after his name and is still called as "Caesar Ciphers". A simple description of his scheme with possibility of using different values for the shift is given in the Figure 1.1.



shift: 3

message: hello

```
Encrypted message = khoor
Notice that each letter has been shifted by 3 letters!
The plaintext is recovered by simply doing the reverse shift by 3 letters.
Recovered message = hello
```

Figure 1.1: Caesar Cipher.

Subsequently, a number of classical cryptographic algorithms, including the Vigenère cipher, Hill cipher, and various other schemes, were developed and implemented on a small scale. Most of these methods involved using a single key for both encryption and decryption, which was known only to the sender and intended recipient. This form of cryptography, where a single key is used for both encryption and decryption, is referred to as private-key cryptography.

However, the main challenge with this type of cryptosystem lies in securely transmitting the key to the communicating parties, particularly when it is not possible for the parties to physically meet in a secure location or use a reliable courier, which can be both slow and expensive.

An alternative to the private-key cryptosystem is the public-key cryptosystem, which was first proposed by Diffie and Hellman in their seminal paper [2]. Although they did not develop a practical working model, their idea was revolutionary in the field of cryptography. In 1977, Rivest, Shamir, and Adleman introduced one of the first practically useful public-key cryptosystems in [3], which is based on the ideas of Diffie and Hellman. This system is called RSA, which stands for the initials of the three authors.

The RSA cryptosystem is a number theory-based cryptosystem that utilizes a different set of public and private keys. It is based on a fundamental result from elementary number theory known as Fermat's Little Theorem. We state the theorem below without proof:

Theorem 1.0.1 (Fermat's little theorem)**.** *Let* p *be any prime number,* a *and* n *be any coprime natural numbers, that is,* $\gcd(a, n) = 1$*. Then we have:*

$$a^{p-1} \equiv 1 \mod p.$$

To generate keys for the RSA cryptosystem, it is necessary to first generate sufficiently large prime numbers p and q, and then calculate their product $n = pq$. Next, a natural number e is chosen such that $\gcd(e, \phi(n)) = 1$, where ϕ is the Euler-Totient function. For any positive integer n, $\phi(n)$ is defined as the number of positive integers less than n that are coprime to n. Because e and $\phi(n)$ are coprime, there exists an integer d in the integer ring modulo $\phi(n)$ such that $ed \equiv 1 \mod \phi(n)$. The resulting keys are a public key, (n, e), and a private key, (d, p, q).

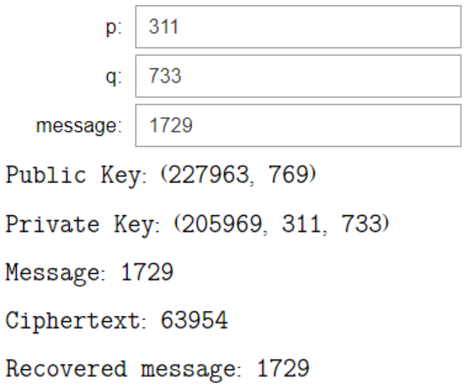
To encrypt a message m using the RSA cryptosystem, we raise it to the power of e and then reduce the result modulo n. That is, we calculate $c \equiv m^e \mod n$, where c is the resulting ciphertext. To decrypt the ciphertext, we raise it to the power of d and reduce it modulo n. That is, we calculate $c^d \equiv m \mod n$, where m is the original message.We briefly outline the proof of why the decryption process works.

Proof. From the relation $ed \equiv 1 \mod \phi(n)$, we obtain $ed = 1 + k\phi(n)$ for some integer k. Therefore, by using this relation, the multiplicative property of ϕ, and the Fermat's little theorem, we obtain the following chain of congruences:

$$c^d = m^{ed} \equiv m^{1+k\phi(n)} \equiv m \cdot m^{k\phi(n)} \equiv m \cdot (m^{p-1})^{k(q-1)} \equiv m \mod p.$$

Now repeating the same with q, we obtain $c^d = m^{ed} \equiv m \mod q$. From these two relations we have for primes p, q that $p | m^{ed} - m$ and $q | m^{ed} - m$. This implies that $n = p \cdot q | m^{ed} - m$. That is, $m^{ed} \equiv m \mod n$. $\square$

To demonstrate how RSA works in practice, we include an interactive SageMath code, taking inspiration from [4], in the appendix that implements the algorithm. We also provide a screenshot of an example that we tested using the code in Figure 1.2.



p: 311
q: 733
message: 1729

```
Public Key: (227963, 769)
Private Key: (205969, 311, 733)
Message: 1729
Ciphertext: 63954
Recovered message: 1729
```

Figure 1.2: RSA.

The security of RSA is based on the computational infeasibility of factoring numbers formed from large primes by modern computers. However, according to Shor's algorithm [5], the development of a practical quantum computer would render factoring these numbers to no longer be computationally infeasible . Therefore, post-quantum secure cryptographic algorithms are being developed that do not rely on factoring of integers. Instead, the trend is towards problems based on Lattice theory, multivariate polynomials, and error-correcting codes. In this thesis, we will focus on code-based cryptography. In the next chapter, we will describe error-correcting codes in detail and show how they are used to build cryptosystems.

Chapter 2

Error-Correcting Codes

Error-correcting codes, as their name suggests, were developed to detect and correct errors caused during the transmission of data over a noisy channel. These errors can manifest in various forms, such as bit flips, insertion, or deletion of bits. The central idea behind the development of these codes is to append the original data with redundant data, which is mathematically connected to the original data. This mathematical relation is known by the receiver, and upon receiving the data, the redundant bits are compared with the bits if no error had occurred. If a mismatch is detected, it means that errors have been detected therefore, they are then corrected using some decoding scheme.

An example of such a code(for error detection) in everyday life is the use of 13-digit barcodes on supermarket products (see [6]). The first 12 digits contain information about the product, while the last digit is a redundant digit that is added to check the integrity of the barcode number. For instance, if the barcode is: $a_1a_2a_3a_4a_5a_6a_7a_8a_9a_{10}a_{11}a_{12}a_{13}$ then the check digit a_{13} is calculated as

$$a_{13} = 10 - \left(\sum_{i=1}^{6} a_{2i-1} + \sum_{i=1}^{6} 3 \cdot a_{2i} \right) \mod 10.$$

2.1 Linear Codes

Linear codes are a well-known and practically useful example of error-correcting codes. Before presenting the formal definition, we will consider the following scenario. Let us imagine that two computers, A and B, are communicating over a network. Computer A sends a message, denoted by a vector m,

$$m = \begin{pmatrix} a & b & c & d \end{pmatrix}.$$

where each entry in m represents a bit. To enable error detection capability, computer A appends three redundant bits or collections of bits to m, producing a new vector v,

$$v = \begin{pmatrix} a & b & c & d & a+b & b+c & a+d \end{pmatrix},$$

which is then transmitted to computer B. This process of adding redundancy is known as encoding. Suppose that computer B receives a message $\bar{v}$, where

$$\bar{v} = \begin{pmatrix} \bar{a} & \bar{b} & \bar{c} & \bar{d} & x & y & z \end{pmatrix}$$

which is related to v but may have been altered during transmission. To verify whether any errors have occurred, computer B checks the following three parity relations,

$$\begin{aligned} \bar{a} + \bar{b} + x &= 0 \\ \bar{b} + \bar{c} + y &= 0 \\ \bar{a} + \bar{d} + z &= 0 \end{aligned}$$

which are deduced from the encoding scheme involving the addition of bits over $\mathbb{F}_2$ (the XOR operation). For example,

$$\begin{aligned} \bar{a} + \bar{b} + x &= \bar{a} + \bar{b} + (\bar{a} + \bar{b}) \\ &= 2\bar{a} + 2\bar{b} \\ &= 0. \end{aligned}$$

These relations are called parity check equations because each of them checks if the sum of bits (according to encoding scheme) is $0 \mod 2$, i.e, the sum is of even parity. These relations can be used to locate errors in the encoded message. For instance, if the equation $\bar{a} + \bar{b} + x = 1$ holds, then it is likely that the error occurred in either $\bar{a}$, $\bar{b}$, or x. However, we can suppose that the likelihood of only one error occurring is higher than the likelihood of two or more errors (see [7]). In that case, it is reasonable to assume that only one error occurred, and if the other two parity relations also hold, it is highly likely that the error occurred in x. In this case, the computer can correct the error simply by flipping the bit x. This process of error detection and correction is known as decoding.

Notice that there is a nice linear structure behind these processes. For example, to encode the message m to v, where v is called a codeword, we simply need to multiply m by a 4×7 matrix G whose first three columns generate a 4×4 identity matrix and the remaining columns produces the redundant bits after multiplication with m. This matrix is called a Generator Matrix of the code and in our example it looks like:

$$G = \left(\begin{array}{cccc|c|cc} 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right)$$

and we can easily observe that the result of the product $m \cdot G = v$. Similarly, in the decoding process the parity relations to be verified can be obtained by multiplying the codeword v by a 3×7 matrix whose last three columns forms an identity matrix and the first four columns are such that after multiplication by v^T, it generates a $\vec{0}$, whose entries are nothing but the right hand side of the parity check equations. This matrix is therefore called the Parity-Check Matrix and in our case looks like:

$$H = \left(\begin{array}{ccccccc} \multicolumn{4}{c}{\boxed{1 \quad 1 \quad 0 \quad 0}} & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{array}\right)$$

and hence the result of the product $H \cdot v^T = \vec{0}$. However, if the computer B receives a message which has been corrupted by an error, i.e $v \neq \bar{v}$, then the result of this product will not be a $\vec{0}$ and hence we can deduce $\bar{v}$ is not a codeword. As in our above example, if the bit x in $\bar{v}$ is flipped then result of the computation will be:

$$H \cdot \bar{v}^T = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}^T .$$

The product $H \cdot \bar{v}^T$ is called the **syndrome** of $\bar{v}$.

From the representation of the generator and parity check matrices, we can observe the nice underlying structure. For instance, if our original message is of size m and we append $n - m$ bits to our original message, where $m \leq n$ then our generator matrix has the block structure:

$$G = \begin{pmatrix} I_{m \times m} & | & P_{m \times (n-m)} \end{pmatrix}$$

and the parity check matrix, as seen form our example, has the following block structure:

$$H = \left(P_{(n-m)\times m} \quad | \quad I_{(n-m)\times(n-m)} \right).$$

This highlights the connection between the two matrices which is that the rows of first $n-m$ columns of the parity check matrix are exactly transposed entries of last $n-m$ columns of the generator matrix. This is not a coincidence but because of how encoding procedure and the parity relations are defined. In the specific example we gave above for the message being of 4 bits and the codeword being 7 bits, this can be easily observed. The 5th bit is calculated as $a+b$ therefore, the 5th column(in red box) of G contains 1 in the first two rows while 0 in the rest so the 5th bit in the codeword will be calculated as:

$$a \cdot 1 + b \cdot 1 + c \cdot 0 + d \cdot 0 = a + b.$$

Accordingly, the parity check relations are utilized to ascertain whether the fifth bit of the received word $\bar{v}$ is intact. If this bit remains uncorrupted by an error, then the fifth bit of the received word is the sum of bits a and b. This, in turn, implies that the entries in the first row of the parity matrix corresponding to columns 1, 2, and 5 must be 1, while all other entries must be 0. By multiplying the first row of the parity matrix H with the received word $\bar{v}$ we obtain:

$$1 \cdot a + 1 \cdot b + 0 \cdot c + 0 \cdot d + 1 \cdot (a+b) + 0 \cdot (b+c) + 0 \cdot (a+d) = a + b + (a+b) = 0.$$

Therefore, we can observe that the first row of the parity check matrix is such that the first 4 entries form a vector which is the transpose of the 5th column of the generator matrix(see red boxes). Similarly, we can obtain the other entries. This method allows us to convert between the parity-check matrix and the generator matrix. These structures are called systematic forms and can be obtained by using Gaussian Elimination.

Now that we have shown a detailed example, we are ready to list down the formal definitions of Linear Codes and relevant concepts. The following definitions have been adapted from [8].

Definition 2.1.1 (Word). *Let S be a set of alphabets and $n \in \mathbb{N}$. Then $s \in S^n$ is called a word of length n.*

Remark 2.1.2. *Although the set of alphabets can be arbitrary, we will restrict it to Finite Fields since most of the results have been developed in this setting.*

Definition 2.1.3 (Code). *A code C is defined over an alphabet S and is just a subset of S^n, for some $n \in \mathbb{N}$.*

Definition 2.1.4 (Codeword)**.** *An element of the code C is called a codeword. If $C \subset S^n$, then each codeword in C has length n.*

Definition 2.1.5 (Linear Code)**.** *If the code $C \subset S^n$ generates a vector subspace over the alphabet S, then this subspace is called a Linear Code, the elements of which are simply the codewords. A Linear Code is characterized by three essential parameters: the length n of its codewords, the dimension k, which is the dimension of C as a vector space over S, and the minimum distance d, and is usually denoted as $[n, k, d]$-linear code over S.*

Proposition 2.1.6. *If $S = \mathbb{F}_q$ is a finite field, where q is a prime or a prime power and C is a $[n, k, d]$-linear code over S, then $k = \log_q(|C|)$, where $|C|$ is the cardinality of C, i.e. the number of codewords in C.*

Proof. Consider a vector space V of dimension k over S and let $\mathcal{B}$ be its basis. Then we have, $|\mathcal{B}| = \dim V$. This means that there exist vectors $v_1, v_2, \ldots, v_k \in \mathcal{B} \subset V$ and constants $a_1, a_2, \ldots, a_k \in S$ such that for any $v \in V$, we have:

$$v = a_1 v_1 + a_2 v_2 + \cdots + a_k v_k.$$

Since $|S| = q$, the number of vectors $v \in V$ is equal to q^k, i.e, $|V| = q^k$. Therefore, by homomorphism between C and V, we have $|C| = q^k$ and this implies the statement. ☐

Definition 2.1.7 (Hamming distance, Hamming weight)**.** *Let $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$, $\boldsymbol{y} = (y_1, y_2, \ldots, y_n) \in S^n$. Then the Hamming distance from $\boldsymbol{x}$ to $\boldsymbol{y}$ is given by*

$$d_{ham}(\boldsymbol{x}, \boldsymbol{y}) := |i \ ; \ x_i \neq y_i|,$$

and the Hamming weight of $\boldsymbol{x}$ is given by

$$d_{ham}(\boldsymbol{x}, \vec{0}) = |i \ ; \ x_i \neq 0|.$$

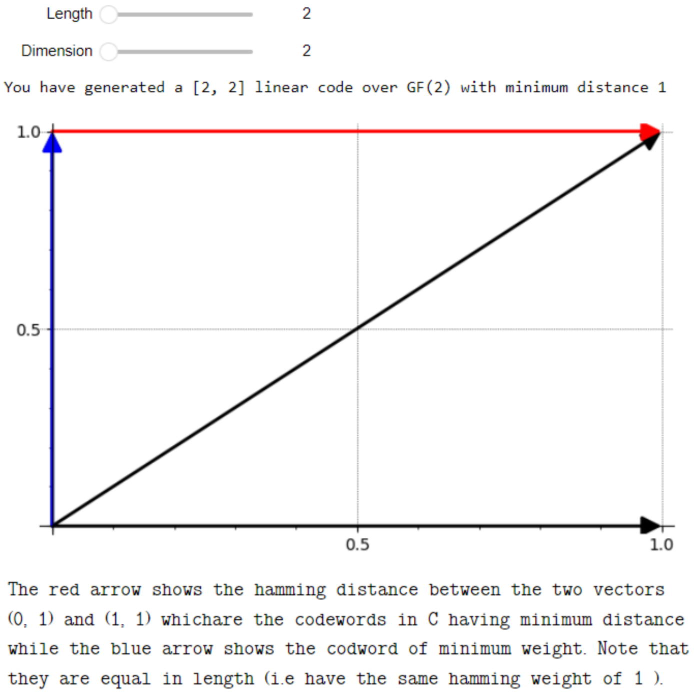
Definition 2.1.8 (minimum distance)**.** *Let C be a $[n, k, d]$-linear code over S. Then the minimum distance of C is defined as*

$$d := \min\{d_{ham}(\boldsymbol{x}, \boldsymbol{y}) | \boldsymbol{x}, \boldsymbol{y} \in C \ and \ \boldsymbol{x} \neq \boldsymbol{y}\}.$$

Remark 2.1.9. *If C is a $[n, k, d]$-linear code over S, then the* ***minimum distance*** *of the C is equal to the Hamming weight of the* ***minimum weight*** *codeword. Thus, we can can calculate the minimum distance as:*

$$d := \min\{d_{ham}(\boldsymbol{x}, \vec{0}) | \boldsymbol{x} \in C \textit{ and } \boldsymbol{x} \neq \vec{0}\}.$$

A visual representation of this is shown in Figure 2.1.



Length 2

Dimension 2

```
You have generated a [2, 2] linear code over GF(2) with minimum distance 1
```

The red arrow shows the hamming distance between the two vectors (0, 1) and (1, 1) whichare the codewords in C having minimum distance while the blue arrow shows the codword of minimum weight. Note that they are equal in length (i.e have the same hamming weight of 1).

Figure 2.1: Minimum Distance and Minimum Weight.

Definition 2.1.10 (Generator Matrix)**.** *Let C be a $[n, k]$-linear code over the alphabet S. A Generator Matrix of C is a $k \times n$ matrix, denoted by G, such that the row space of G forms a basis for C. This enables us to define a linear code C over an alphabet S alternatively as*

$$C = \{m \cdot G \mid m \in S^k\}.$$

This process of generating codeword from the original message is also called encoding.

Definition 2.1.11 (Parity-Check Matrix)**.** *For a $[n, k]$-linear code C over the alphabet S, the parity check matrix is an $(n - k) \times n$ matrix, denoted by H, such that for any codeword $c \in C$, $H \cdot c^T = \vec{0}$. Therefore, we can also define linear codes as*

$$C = \{c \in S^n \mid H \cdot c^T = \vec{0}\}.$$

Definition 2.1.12 (Dual Code)**.** *For a $[n, k]$-linear code C over the alphabet S, the dual code is defined as*

$$C^{\perp} := \{\bar{c} \in S^n | \bar{c} \cdot c = 0, \forall c \in C\}.$$

Remark 2.1.13. *The dual code of C, denoted by $C^{\perp}$, is a $[n, n-k]$-linear code whose generator matrix is the parity-check matrix of C.*

2.2 Error Detection and Error Correction capacity of Linear Codes

At the beginning of this section we showed how error correcting codes can be used to detect one bit errors and thus correct it by flipping the bit. However, in practice, usually more than one bit error occurs so to maintain efficient information transfer, our error-correcting codes must be capable of detecting and correcting more than one bit error. The error detection and correction ability of the linear code is closely related to its minimum distance.

The $[n, k, d]$-linear code C can detect errors up to a certain limit, which is strictly less than the minimum distance d of the code. In other words, $0 < t < d$, where t represents the number of errors. For example, if t denotes the number of errors that occur in our codeword c, resulting in a received word r containing t errors, we can express this as $r = c + e$, where e is a vector that introduces t errors into our codeword c. If $t = d$, then the Hamming distance between the codeword $\mathbf{c}$ and the received word $\mathbf{r}$ is $d_{ham}(\mathbf{c}, \mathbf{r}) = d$, which means that r is again a codeword, i.e., $r \in C$. Therefore, during the decoding stage, we obtain $H \cdot r^T = 0$, where H is the parity check matrix. This implies that no error has occurred, and thus, the errors go undetected. Hence, to ensure effective decoding, we must ensure that the number of errors is less than the minimum distance of the code.

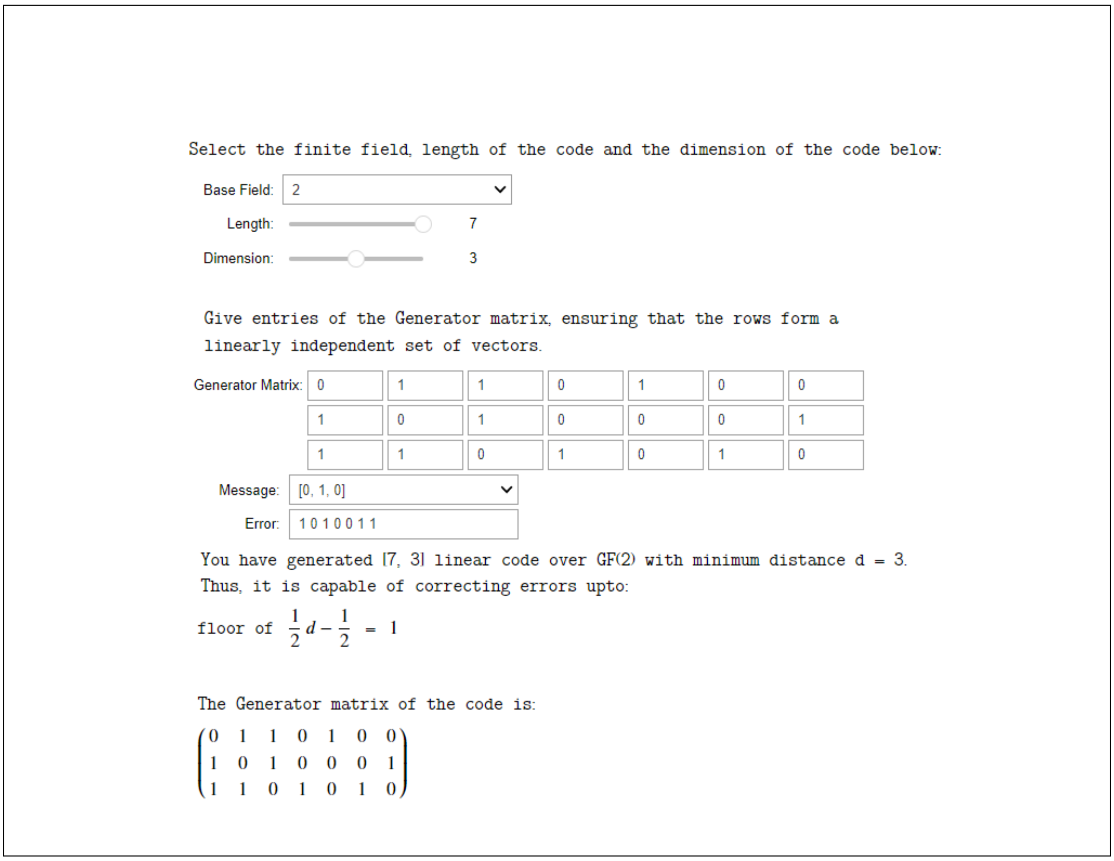
The ability of our code to correct errors depends on the decoding algorithm used. However, the most efficient and commonly used decoding scheme is based on minimum distance decoding, which involves finding the nearest codeword to the received word. Using this scheme, our code can correct up to

$$t = \left\lfloor \frac{d-1}{2} \right\rfloor,$$

where d is the minimum distance of the code. This can be easily visualized by considering C as a metric space with d_{ham} as the metric and drawing a ball of radius $\frac{d-1}{2}$ around each codeword. It is clear that these balls are non-intersecting since any two codewords are at a distance greater than $2 \cdot \frac{d-1}{2}$ apart. Therefore, any error-induced word located within this ball can be mapped to a unique codeword, achieving error correction.

2.3 Example

In this section we will demonstrate some of the above concepts in a SageMath Interact session, implementing the generic Linear Codes using the built in sage LinearCode class(see Figure 2.2).



Select the finite field, length of the code and the dimension of the code below:

Base Field: 2

Length: 7

Dimension: 3

Give entries of the Generator matrix, ensuring that the rows form a linearly independent set of vectors.

Generator Matrix:	0	1	1	0	1	0	0
	1	0	1	0	0	0	1
	1	1	0	1	0	1	0

Message: [0, 1, 0]

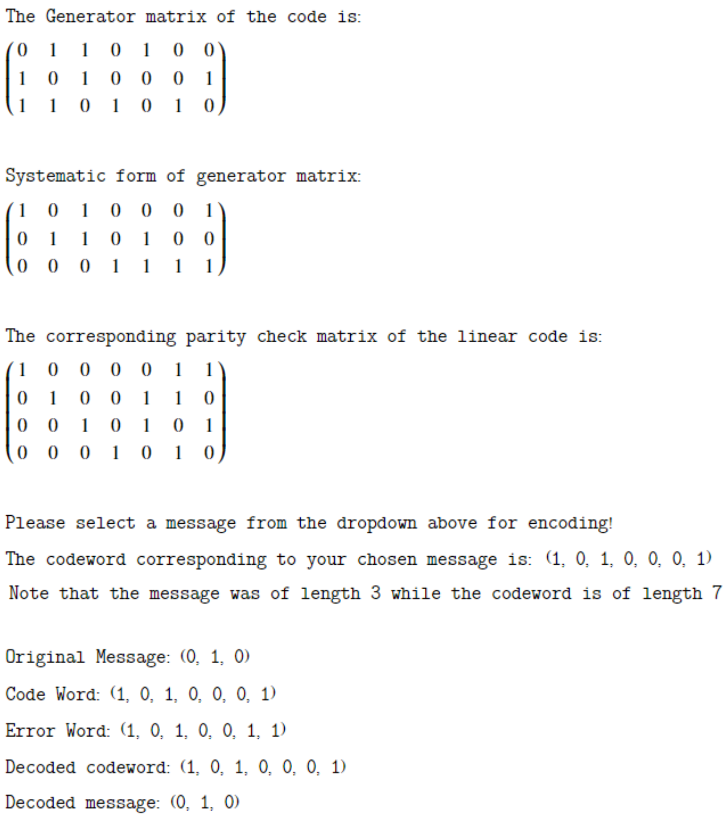
Error: 1 0 1 0 0 1 1

You have generated [7, 3] linear code over GF(2) with minimum distance d = 3. Thus, it is capable of correcting errors upto:

floor of $\frac{1}{2}d - \frac{1}{2}$ = 1

The Generator matrix of the code is:

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$



The Generator matrix of the code is:

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Systematic form of generator matrix:

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

The corresponding parity check matrix of the linear code is:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

```
Please select a message from the dropdown above for encoding!
The codeword corresponding to your chosen message is: (1, 0, 1, 0, 0, 0, 1)
Note that the message was of length 3 while the codeword is of length 7

Original Message: (0, 1, 0)
Code Word: (1, 0, 1, 0, 0, 0, 1)
Error Word: (1, 0, 1, 0, 0, 1, 1)
Decoded codeword: (1, 0, 1, 0, 0, 0, 1)
Decoded message: (0, 1, 0)
```

Figure 2.2: Interactive Linear Code

In summary, this chapter has provided an overview of the concept of Error-Correcting codes, with a particular emphasis on Linear codes. In the following chapter, we will shift our focus to a well-known cryptographic system based on Error-Correcting codes, the McEliece Cryptosystem.

Chapter 3

The McEliece Cryptosystem

In 1978, McEliece [9] introduced the first ever asymmetric cryptosystem based on error-correcting codes. The core concept of this cryptosystem is to intentionally add errors to the original message during encryption and subsequently employ the error-correcting properties of linear codes to recover the original message during decryption. However, the encoding and decoding algorithms for Linear Codes cannot be used verbatim, as the advantageous structure between the Generator matrix of the Code (in reduced echelon form) and the Parity-check matrix can be exploited by an adversary. To avoid this vulnerability, McEliece introduced modifications to the encryption and decryption processes rather than simply deploying the usual encoding and decoding of Linear Codes, as we will see in the subsequent sections.

3.1 Key Generation

As with any Public-key cryptosystem, the initial step in setting up the McEliece cryptosystem involves the generation of a pair of public and private keys. Given that the McEliece cryptosystem is based on linear codes, the first step is to determine the code's length, dimension, and minimum distance as indicated by the parameters: n, k, d. With this information, we can readily produce a $k \times n$ Generator Matrix (G) of the underlying Linear Code over a chosen base field. However, instead of using this generator matrix for encryption, McEliece proposed a scheme for disguising the generator matrix by multiplying it with the Scrambler Matrix (S) on the left and the Permutation Matrix (P) on the right. The Scrambler Matrix is a randomly generated $k \times k$ non-singular matrix. One way to generate the Scrambler Matrix is by randomly generating matrices and using fast algorithms to compute the determinants until a matrix with a non-zero determinant is obtained (see [10]). Alternatively, we can generate a vector space of dimension k over the base field and select a random element from the vector space. We then iterate over the remaining elements, selecting only those that are not in the span of the already chosen vectors, and stop once we obtain k such vectors. This process gives us an invertible $k \times k$ matrix whose rows consist of these basis vectors. Next, we generate an $n \times n$ Permutation Matrix by permuting the columns of an Identity Matrix of size n, which we can easily create in SageMath using the built-in sage class Permutations and its methods. Now that we have obtained these matrices, we compute their product to obtain

$$G' = S \cdot G \cdot P$$

and call G' a disguised matrix because it nicely hides the structure of the generator matrix G. We can describe this transformation as a *trapdoor function* in which the easier direction is to compute G' from S, G and P as this is a mere matrix multiplication. The difficult direction is to decompose G' back into S, G and P without the knowledge of S and P. This makes G' a perfect candidate for encryption therefore, a public key. Note that this does not change the underlying Linear Code so all the properties including error-correcting capacity, given by t, stays the same. Therefore, the **Public Key** is given by (G', t).

The privacy of the scrambler matrix S and permutation matrix P is crucial in hiding the structure of G and therefore, they must be kept confidential. In addition, knowledge of S and P is necessary for message decoding since the decomposition of G' is challenging without this information. Thus, S, P, and G form part of the Private Key. Moreover, the Private Key must also include a decoding algorithm, denoted as D, which corrects errors introduced during encryption. Therefore, the complete **Private Key** is represented as (S, G, P, D). A pseudocode of the key-generation process is given in Algorithm 1.

Algorithm 1 Key Generation

```
Input: Generator Matrix, Linear Code
Output: Public Key, Private Key
function keyGeneration(generatorMatrix $G$, linearCode $C$)
    $k \leftarrow$ numRows($G$), $n \leftarrow$ numCols($G$)
    $P \leftarrow$ permutationMatrix($n$)
    $S \leftarrow$ scramblerMatrix($\mathbb{F}$, $k$)
    $G' \leftarrow S * G * P$                    ▷ Compute modified generator matrix
    $d \leftarrow$ minimumDistance($C$)
    $t \leftarrow \left\lfloor \frac{(d-1)}{2} \right\rfloor$                    ▷ Compute error correction parameter
    $D \leftarrow$ decodingAlgorithm($C$)
    publicKey $\leftarrow (G', t)$
    privateKey $\leftarrow (S, P, G, D)$
return publicKey, privateKey
```

3.2 Encryption

Once the keys have been generated, the encryption process becomes fairly simple. First the original message is split into blocks of length k which we denote by m. Then for each block m, we generate a random error vector of length n and Hamming weight t, that is, $e \in \mathbb{F}^n$. Finally the cipher text c, for each block m, is obtained as

$$c = m \cdot G' + e.$$

To facilitate comprehension, we shall limit our discussion to the encryption and decryption of a single block, noting that the process can be repeated for each block in the sequence. Algorithm 2 summarizes the encryption procedure.

Algorithm 2 Encryption

```
Input: Message Block, Public Key
Output: Ciphertext
function encryption($m, t, G'$)
    $n \leftarrow$ numCols($G'$)
    $L \leftarrow$ list containing $t$ elements from $\mathbb{F}$ and $(n - t) \quad 0's$
    $e \leftarrow$ generatePermutation($L$)
    $c \leftarrow m \cdot G' + e$
return $c$
```

3.3 Decryption

The ciphertext undergoes several transformations before converting back to the original message. First it is multiplied by inverse of the Permutation Matrix to undo the permutation action. Then the decoding algorithm for the underlying Linear Code is used to correct the errors introduced during encryption. Finally the multiplication by inverse matrices of the Scrambler Matrix and the Generator Matrix produces back the original message. These transformations indeed yield the original message as we will show in the following proof. Let c be the ciphertext, S, G, P be the scrambler, generator, and permutation matrices as described above and D be the decoding algorithm. Then

$$\begin{aligned}
c \cdot P^{-1} &= (m \cdot G' + e) \cdot P^{-1} \\
c \cdot P^{-1} &= (m \cdot S \cdot G \cdot P + e) \cdot P^{-1} \\
c \cdot P^{-1} &= m \cdot S \cdot G + e \cdot P^{-1} \\
D(c \cdot P^{-1}) &= D(m \cdot S \cdot G + e \cdot P^{-1}) \\
D(c \cdot P^{-1}) &= m \cdot S \cdot G \\
D(c \cdot P^{-1}) \cdot (S \cdot G)^{-1} &= m.
\end{aligned}$$

In the fourth line above, the decoding algorithm is able to retrieve the code word from the $e \cdot P^{-1}$ vector because the inverse Permutation Matrix, P^{-1}, only changes the permutation of the bits of the error vector and not its weight. The inverse matrices $P^{-1}, (S \cdot G)^{-1}$, can be pre-computed and then directly multiplied for each decryption process. Note that the final step of multiplying by $(S \cdot G)^{-1}$ can be achieved by using some solving algorithm for linear systems. Algorithm 3 summarizes the decryption process.

Algorithm 3 Decryption

```
Input: Ciphertext, Private Key
Output: Message
function DECRYPTION($c, S, G, P, D$)
    $m_0 \leftarrow c \cdot P^{-1}$
    $m_1 \leftarrow D(m_0)$
    $m \leftarrow$ solveEquation($(S \cdot G) \cdot m = m_1$)
return $m$
```

3.4 Implementation Using Classical Reed-Solomon Codes

In this section we will implement the McEliece cryptosystem using Reed-Solomon code as the underlying linear code. Reed-Solomon codes are a class of $[n, k, d]$ Linear Codes (see Definition 2.1.5) over a finite field of size q ($\mathbb{F}_q$) such that $n|q-1$ and $d = n-k+1$. For a detailed account of these codes, please refer to [11].

To set up the cryptosystem, we begin by generating the public and private keys. Since each linear code based cryptosystem differ only in the encoding matrix and the decoding algorithm (D), we will only focus on their description. The other components of keys can be generated in the same way as described in the Section 3.1. To encode a message of length k, a polynomial of degree less than k over $\mathbb{F}_q$ is generated with the coefficients being the coordinates of the message vector. Then n elements from $\mathbb{F}_q$, $(a_1, a_2, \ldots, a_n)$ are chosen, on which this polynomial is evaluated to obtain the codeword for the message. These points are called the evaluation points of the code. For instance, if the message is $m = (m_0, m_1, \ldots, m_{k-1}) \in \mathbb{F}_q^k$, then the corresponding polynomial will be $p_m(x) = \sum_{i=1}^{k-1} m_i x^i$ and if $\alpha \in \mathbb{F}_q$ is a primitive element, then the evaluation points will be $1, \alpha, \alpha^2, \ldots, \alpha^{n-1}$. Therefore, the codeword obtained is given by

$$(p_m(1), p_m(\alpha), p_m(\alpha^2), \ldots, p_m(\alpha^{n-1})).$$

The encoding procedure can be seen by the matrix multiplication of the message and the generator matrix below

$$\begin{pmatrix} m_1 & m_2 & \ldots & m_k \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \alpha & \alpha^2 & \ldots & \alpha^{n-1} \\ 1 & \alpha^2 & (\alpha^2)^2 & \ldots & (\alpha^{n-1})^2 \\ \vdots & \vdots & \vdots & \ddots & \\ 1 & \alpha^{k-1} & (\alpha^2)^{k-1} & \ldots & (\alpha^{n-1})^{k-1} \end{pmatrix}.$$

It can be readily observed that the Generator matrix has similar structure to a Vandermonde matrix generated from a primitive element $\alpha \in \mathbb{F}_q$. This structure is utilized in the decoding procedure as well and thus α forms part of the private key.

For the decoding of Reed-Solomon codes, we use one of the earliest know algorithm deployed for its decoding, namely the Berlekamp-Welch Algorithm (see [12]). Assume that the codeword we obtined from encoding of the message (as shown above) is given by $c = (c_1, c_2, \ldots, c_n)$ and denote the received word, possibly containing errors, by $y = (y_1, y_2, \ldots, y_n)$. This means that $y = c + e$, where e is the error vector of weight

$$t \leq \frac{d-1}{2} = \frac{n-k}{2}.$$

To find this error vector, we define a so called Error-Locator polynomial, denoted by $\epsilon(x)$ such that if there was an error in the i^{th} coordinate y_i of the received word, that is $p_m(a_i) \neq y_i$, then a_i is a root of $\epsilon(x)$. Therefore, we can define this polynomial as $\epsilon(x) = \prod_i (x - a_i)$.

The primary objective of decoding is to recover the original encoding polynomial $p_m(x)$. One possible approach to achieving this objective is to locate the roots of the error locator polynomial $\epsilon(x)$, which in turn would allow us to identify the positions of the errors. Subsequently, we could retrieve the encoding polynomial $p_m(x)$ by applying some interpolation technique to the non-error positions. Specifically, for each non-error position, we have the value $p_m(a_i) = y_i$, which can be used in the interpolation process.

However, in practice these errors are not known at the receiver end therefore, another technique is required to obtain $p_m(x)$. This technique can be derived form a nice property satisfied by $\epsilon(x)$ which is the following

$$p_m(a_i)\epsilon(a_i) = y_i\epsilon(a_i)$$

for all $a_i \in \mathbb{F}_q, y_i \in y$. If we let $Q(x) = p_m(a_i)\epsilon(a_i)$ then we obtain an equation

$$\begin{aligned} &Q(a_i) = y_i\epsilon(a_i) \\ &(q_0 + q_1 a_i + q_2 a_i^2 \cdots + q_q a_i^j) - y_i(e_0 + e_1 a_i + \cdots + e_t a_i^t) = 0, \end{aligned}$$

for $i = 0, 1, \ldots, n-1$. Note that $\deg Q = k - 1 + t$ and this implies $j = k - 1 + t$. The above equations form a linear system with $e_0, e_1, \ldots, e_t, q_0, q_1, \ldots, q_{k-1+t}$ as the unknowns that is $2t + k - 1 \leq n - 1 < n$ unknowns and we have n equations for each i, thus this system can be solved.

The motivation behind designing the algorithm to obtain $p_m(x)$ from the above linear system can be seen in the matrix multiplication,

$$\begin{pmatrix} 1 & a_1 & {a_1}^2 & \cdots & a_1^j & -y_1 & -y_1a_1 & \cdots & -y_1a_1^t \\ 1 & a_2 & {a_2}^2 & \cdots & a_2^j & -y_2 & -y_2a_2 & \cdots & -y_2a_2^t \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & a_n & {a_n}^2 & \cdots & a_n^j & -y_n & -y_na_n & \cdots & -y_na_n^t. \end{pmatrix} \cdot \begin{pmatrix} q_0 \\ q_1 \\ q_2 \\ \vdots \\ q_j \\ e_0 \\ e_1 \\ \vdots \\ e_t \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0. \end{pmatrix}$$

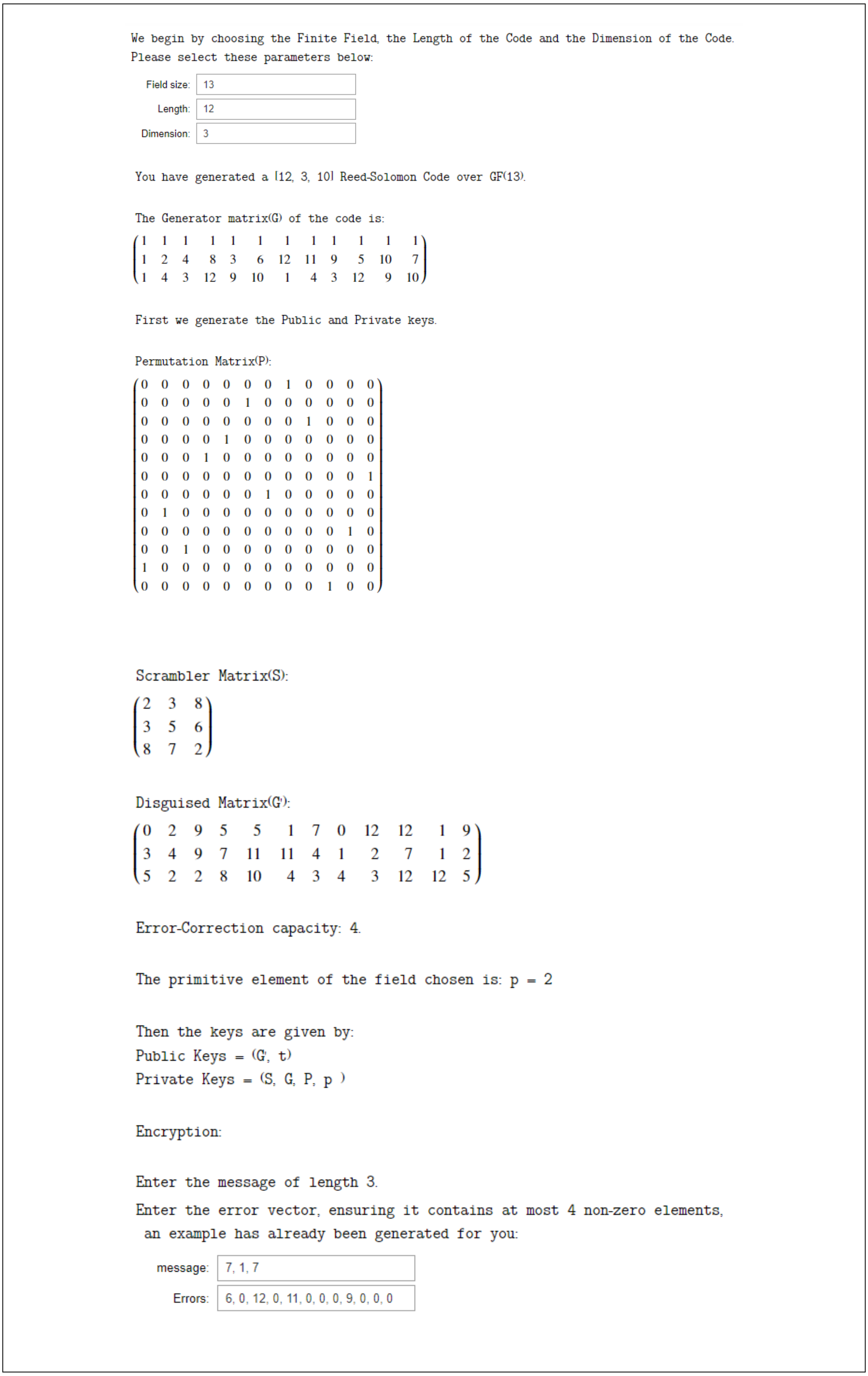
Note that since we generated the points $a_1, a_2, \ldots, a_n$ by a primitive element $\alpha \in \mathbb{F}_q$, we can see that the first q columns have a structure similar to the Vandermonde Matrix (in blue box) and the last t columns(in red box) can be generated by augmenting the blue block with the vectors obtained by taking coordinate wise (i.e. pairwise) multiplication of the received word y and the columns of the blue block. Therefore, this idea is used in the decoding procedure. The remaining procedure is to transform this matrix into a reduced echolon form. Then using the last column obtain the two polynomials, $Q(x)$ with first j entries as its coefficients and $\epsilon(x)$ with last $t+1$ entries as its coefficients.

Recall that according to the definition of $Q(x)$ we have,

$$\frac{Q(x)}{\epsilon(x)} = p_m(x)$$

Therefore, finding this quotient is the only step required to obtain the coefficients of the decoded word. With the decoded word in hand, we can proceed to decrypt the original message that was encrypted using the McEliece scheme. It is important to note that the decoded word must be multiplied by the inverse of the scrambler matrix to complete the decryption process.

In the following figure, we demonstrate the implementation of the McEliece cryptosystem using Reed-Solomon codes with parameters $n = 12, k = 3$ over $\mathbb{F}_{13}$.



We begin by choosing the Finite Field, the Length of the Code and the Dimension of the Code. Please select these parameters below:

Field size: 13

Length: 12

Dimension: 3

You have generated a [12, 3, 10] Reed-Solomon Code over GF(13).

The Generator matrix(G) of the code is:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 3 & 6 & 12 & 11 & 9 & 5 & 10 & 7 \\ 1 & 4 & 3 & 12 & 9 & 10 & 1 & 4 & 3 & 12 & 9 & 10 \end{pmatrix}$$

First we generate the Public and Private keys.

Permutation Matrix(P):

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Scrambler Matrix(S):

$$\begin{pmatrix} 2 & 3 & 8 \\ 3 & 5 & 6 \\ 8 & 7 & 2 \end{pmatrix}$$

Disguised Matrix(G'):

$$\begin{pmatrix} 0 & 2 & 9 & 5 & 5 & 1 & 7 & 0 & 12 & 12 & 1 & 9 \\ 3 & 4 & 9 & 7 & 11 & 11 & 4 & 1 & 2 & 7 & 1 & 2 \\ 5 & 2 & 2 & 8 & 10 & 4 & 3 & 4 & 3 & 12 & 12 & 5 \end{pmatrix}$$

Error-Correction capacity: 4.

The primitive element of the field chosen is: p = 2

Then the keys are given by:
Public Keys = (G', t)
Private Keys = (S, G, P, p)

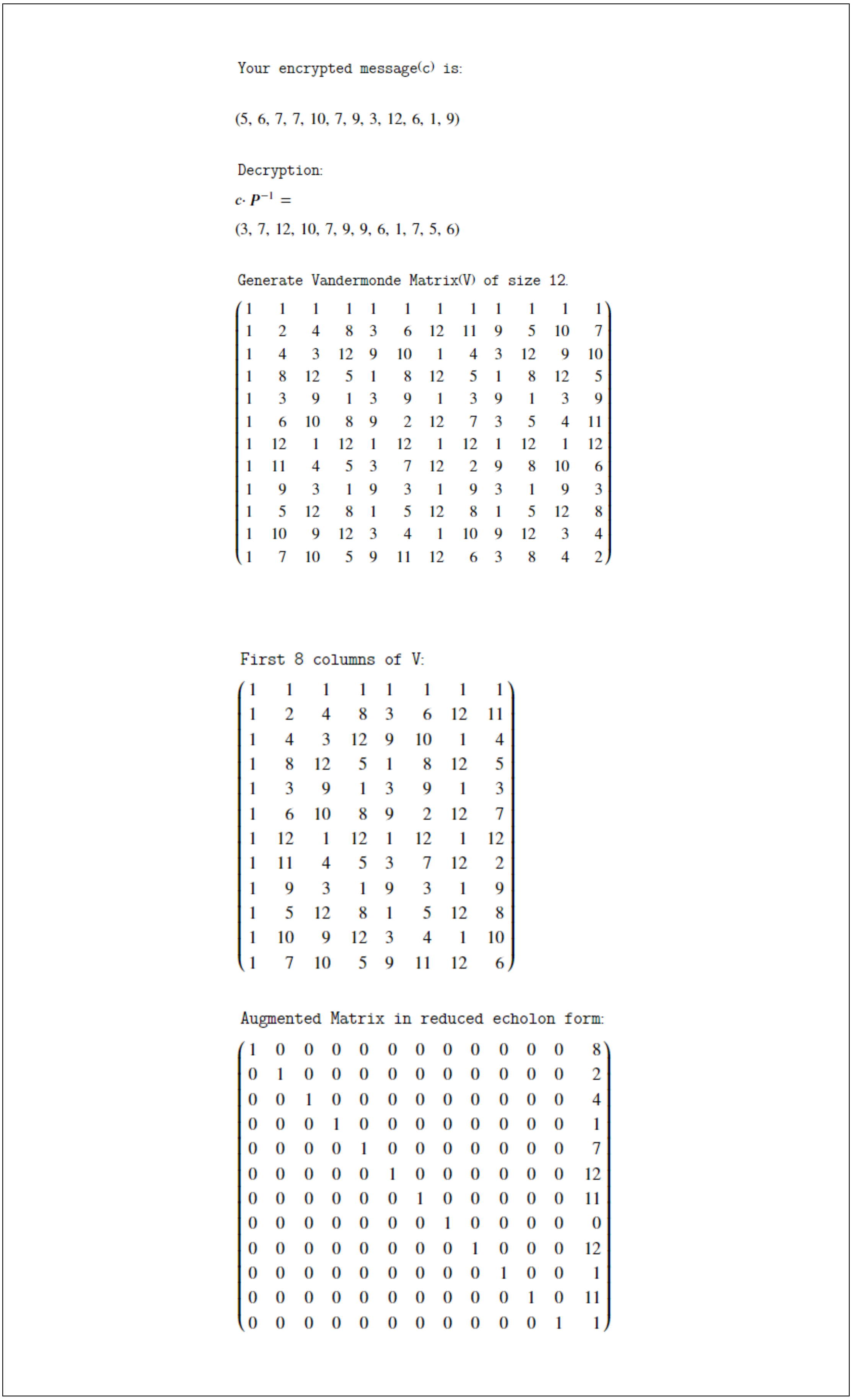
Encryption:

Enter the message of length 3.

Enter the error vector, ensuring it contains at most 4 non-zero elements, an example has already been generated for you:

message: 7, 1, 7

Errors: 6, 0, 12, 0, 11, 0, 0, 0, 9, 0, 0, 0



Your encrypted message(c) is:

(5, 6, 7, 7, 10, 7, 9, 3, 12, 6, 1, 9)

Decryption:

$c \cdot P^{-1} =$

(3, 7, 12, 10, 7, 9, 9, 6, 1, 7, 5, 6)

Generate Vandermonde Matrix(V) of size 12.

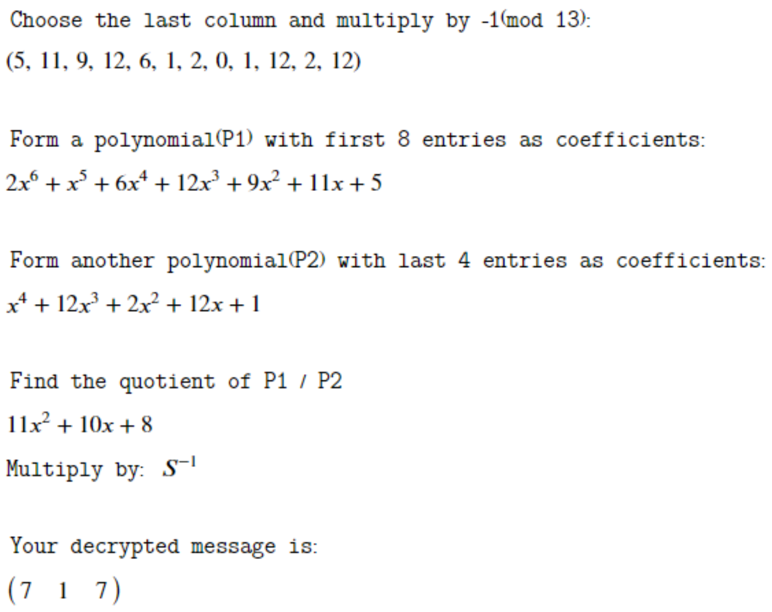
$$\begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 2 & 4 & 8 & 3 & 6 & 12 & 11 & 9 & 5 & 10 & 7 \\
1 & 4 & 3 & 12 & 9 & 10 & 1 & 4 & 3 & 12 & 9 & 10 \\
1 & 8 & 12 & 5 & 1 & 8 & 12 & 5 & 1 & 8 & 12 & 5 \\
1 & 3 & 9 & 1 & 3 & 9 & 1 & 3 & 9 & 1 & 3 & 9 \\
1 & 6 & 10 & 8 & 9 & 2 & 12 & 7 & 3 & 5 & 4 & 11 \\
1 & 12 & 1 & 12 & 1 & 12 & 1 & 12 & 1 & 12 & 1 & 12 \\
1 & 11 & 4 & 5 & 3 & 7 & 12 & 2 & 9 & 8 & 10 & 6 \\
1 & 9 & 3 & 1 & 9 & 3 & 1 & 9 & 3 & 1 & 9 & 3 \\
1 & 5 & 12 & 8 & 1 & 5 & 12 & 8 & 1 & 5 & 12 & 8 \\
1 & 10 & 9 & 12 & 3 & 4 & 1 & 10 & 9 & 12 & 3 & 4 \\
1 & 7 & 10 & 5 & 9 & 11 & 12 & 6 & 3 & 8 & 4 & 2
\end{pmatrix}$$

First 8 columns of V:

$$\begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 2 & 4 & 8 & 3 & 6 & 12 & 11 \\
1 & 4 & 3 & 12 & 9 & 10 & 1 & 4 \\
1 & 8 & 12 & 5 & 1 & 8 & 12 & 5 \\
1 & 3 & 9 & 1 & 3 & 9 & 1 & 3 \\
1 & 6 & 10 & 8 & 9 & 2 & 12 & 7 \\
1 & 12 & 1 & 12 & 1 & 12 & 1 & 12 \\
1 & 11 & 4 & 5 & 3 & 7 & 12 & 2 \\
1 & 9 & 3 & 1 & 9 & 3 & 1 & 9 \\
1 & 5 & 12 & 8 & 1 & 5 & 12 & 8 \\
1 & 10 & 9 & 12 & 3 & 4 & 1 & 10 \\
1 & 7 & 10 & 5 & 9 & 11 & 12 & 6
\end{pmatrix}$$

Augmented Matrix in reduced echolon form:

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 7 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 12 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 11 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 12 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 11 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{pmatrix}$$



Choose the last column and multiply by -1(mod 13):

(5, 11, 9, 12, 6, 1, 2, 0, 1, 12, 2, 12)

Form a polynomial(P1) with first 8 entries as coefficients:

$2x^6 + x^5 + 6x^4 + 12x^3 + 9x^2 + 11x + 5$

Form another polynomial(P2) with last 4 entries as coefficients:

$x^4 + 12x^3 + 2x^2 + 12x + 1$

Find the quotient of P1 / P2

$11x^2 + 10x + 8$

Multiply by: S^{-1}

Your decrypted message is:

$\begin{pmatrix} 7 & 1 & 7 \end{pmatrix}$

Figure 3.1: McEliece PKCS using Reed-Solomon Code

Despite the possibility of implementing the McEliece cryptosystem using Reed-Solomon codes as presented in this section, this approach is not widely adopted due to various known attacks that can compromise its security. One example is the work of Sidelnikov and Shestakov [13], where they demonstrated the feasibility of acquiring evaluation points and the encoding polynomial from the Generator matrix's structural properties. The most secure linear codes which have been found suitable for McEliece cryptosystem are the so-called Binary Goppa Codes. In the next chapter, we will investigate these codes in more detail and present an implementation of McEliece cryptosystem based on these codes.

Chapter 4

Binary Goppa Codes

In 1970, Goppa [14] introduced a new class of linear codes which are now named after him. Here we will present the definition of Goppa codes as presented by Bernstein in [15].

Definition 4.0.1 (Binary Goppa Codes). *Let* $\alpha_1, \alpha_2, \ldots, \alpha_n \in \mathbb{F}_{2^m}$ *be distinct and* $g \in \mathbb{F}_{2^m}[x]$ *be a polynomial such that* $g(\alpha_i) \neq 0$ *for all* $i = 1, 2, \ldots, n$*. Then the Goppa Codes are defined as*

$$C = \left\{ c \in \mathbb{F}_2^n \mid \sum_{i=1}^{n} \frac{c_i}{(x - \alpha_i)} \mod g = 0 \right\}.$$

Remark 4.0.2. *In practice,* $g \in \mathbb{F}_{2^m}[x]$ *is chosen to be irreducible over* $\mathbb{F}_{2^m}$*. This automatically satisfies the condition* $g(\alpha_i) \neq 0$ *for all* $i = 1, 2, \ldots, n$*.*

Proposition 4.0.3. *Let* $h \in \mathbb{F}_{2^m}[x]$ *such that* $h(x) = \prod_{i=1}^{n}(x - \alpha_i)$*. Then the set*

$$\Gamma = \left\{ c \in \mathbb{F}_2^n \mid \sum_{i=1}^{n} c_i \frac{h}{(x - \alpha_i)} \mod g = 0 \right\}$$

gives an equivalent definition of the binary Goppa code and is referred to as the ***polynomial*** *view of the Goppa code.*

Proof. Consider a codeword $c \in \Gamma$. Then $c = (c_1, c_2, \ldots, c_n)$ satisfies that $\sum_{i=1}^{n} c_i \frac{h}{(x-\alpha_i)} \mod g = 0$ and this implies $h \cdot \sum_{i=1}^{n} \frac{c_i}{(x-\alpha_i)} \mod g = 0$, that is, either $h \equiv 0 \mod g$ or $\sum_{i=1}^{n} \frac{c_i}{(x-\alpha_i)} \equiv 0 \mod g$. Since by the choice of g none of the roots of h are the roots of g, we obtain $gcd(h, g) = 1$ and therefore $h \mod (g) \neq 0$. This implies that $c \in C$. The converse is trivially satisfied. $\square$

4.1 Encoding

As explained in Chapter 2, the encoding procedure of a word requires its multiplication by the Generator Matrix of the Linear Code. To obtain this matrix, we begin by selecting an integer $m \geq 3$ which determines the degree of the field extension. We then generate a random irreducible polynomial over $\mathbb{F}_{2^m}$ of degree t(error correction capacity), called the Goppa polynomial, and subsequently choose n code locators $\alpha_1, \alpha_2, \ldots, \alpha_n \in F_{2^m}$.

It is crucial that the values of m, t and n are chosen such that they satisfy the following inequalities:

$$2 \leq t \leq \frac{2^m - 1}{m} \qquad \text{and} \qquad mt < n \leq 2^m.$$

These constraints are derived from the code's properties. For example, we observe that $n > mt$ since the dimension of the Generator Matrix (as we will subsequently demonstrate) is $n - mt > 0$, and by choice $n > 0, mt > 0$. The upper bound on n is trivial because we need to choose n distinct code locators from $\mathbb{F}_{2^m}$ and $|\mathbb{F}_{2^m}| = 2^m$. Similarly, the upper bound on t follows since the opposite inequality leads to contradiction on choice of n. Typically boundary values of t do not produce useful results therefore, it must be chosen somewhere in between. (see [15])

Unlike basic linear codes, in Goppa Codes, the step to obtain the Parity Check Matrix precedes that to obtain the Generator Matrix, owing to the definition of the Code by the parity relation. Thus, upon obtaining the code locators and the Goppa polynomial, we commence constructing the Parity Check Matrix over $\mathbb{F}_{2^m}$ and then expand it to a Parity Check Matrix over $\mathbb{F}_2$. We employ the defining property of the code, as indicated in Proposition 4.0.3. Notably, this property describes the parity relations, for if we consider the column representation of a matrix

$$H = \left(\tfrac{h}{(x-\alpha_1)} \mod g \quad \tfrac{h}{(x-\alpha_2)} \mod g \quad \cdots \quad \tfrac{h}{(x-\alpha_n)} \mod g \right),$$

where each $\frac{h}{(x-\alpha_i)} \mod g$ corresponds to a column, and a codeword $c = (c_1, c_2, \ldots, c_n)$, then the product $H \cdot c^T$ yields the relation. Therefore, the matrix H indeed is a parity check matrix over the extended field.

We will now examine the structure of the entries in the matrix H. As previously noted, for all $i = 1, 2, \ldots, n$, we have that $h, g, (x - \alpha_i) \in \mathbb{F}_{2^m}[x]$. Our objective is to determine the quotient $\frac{h}{(x-\alpha_i)}$ reduced modulo g and its corresponding algebraic parent. Since we selected g to be an irreducible polynomial over $\mathbb{F}_{2^m}$, the ideal generated by g, denoted by $\langle g(x)\rangle$, is a maximal ideal in the polynomial ring $\mathbb{F}_{2^m}[x]$. Consequently, the quotient group $L = \frac{\mathbb{F}_{2^m}[x]}{\langle g(x)\rangle}$ creates an extended field which is isomorphic to $\mathbb{F}_{2^m}(\beta)$, where β represents a root of $g(x)$ in this extension of $\mathbb{F}_{2^m}$. Therefore, any element $l \in L$ can be expressed as $l = a^{[0]} + a^{[1]}\beta + \cdots + a^{[t-1]}\beta^{t-1}$, where $a^{[0]}, a^{[1]}, \ldots, a^{[t-1]} \in \mathbb{F}_{2^m}$. As a result, $\frac{h}{(x-\alpha_i)} \mod g \in L$ for $i = 1, 2, \ldots, n$, forms a polynomial in β of degree no more than $t-1$. Thus, the coefficients form a vector of length t, which correspond to the entries of each column of H. Therefore, the parity check matrix is a $t \times n$ matrix over the extended field $\mathbb{F}_{2^m}$ and has the following form

$$H = \begin{pmatrix} a_1^{[0]} & a_2^{[0]} & \cdots & a_n^{[0]} \\ a_1^{[1]} & a_2^{[1]} & \cdots & a_n^{[1]} \\ \vdots & \vdots & \ddots & \vdots \\ a_1^{[t-1]} & a_2^{[t-1]} & \cdots & a_n^{[t-1]} \end{pmatrix}.$$

However, since computers use binary language, we need to transform the parity check matrix over a binary field to make it easier for them to understand the Goppa error-correction procedure. This means that we need to expand the matrix H to a matrix over the base field, which is $\mathbb{F}_2$.

To create the expansion algorithm for H over the base field $\mathbb{F}_2$, we first recall that $\mathbb{F}_{2^m}$ is a field extension of $\mathbb{F}_2$ by the root α of an irreducible polynomial of degree m over $\mathbb{F}_2$. Thus, any element $f \in \mathbb{F}_{2^m}$ can be expressed as $f = b^{[0]} + b^{[1]}\alpha + \cdots + b^{[m-1]}\alpha^{m-1}$, where $b^{[0]}, b^{[1]}, \ldots, b^{[m-1]} \in \mathbb{F}_2$. Since each entry of H is of the form $a_i^{[j]} \in \mathbb{F}_{2^m}$, it can be represented as a polynomial in α of degree no more than $m-1$, and the coefficients of this polynomial form a (column) binary vector of length m. Consequently, the resulting parity check matrix has dimensions $m \cdot t \times n$ over $\mathbb{F}_2$ and takes the following form:

$$\hat{H} = \begin{pmatrix} a_{1,0,0} & a_{2,0,0} & \cdots & a_{n,0,0} \\ a_{1,0,1} & a_{2,0,1} & \cdots & a_{n,0,1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1,0,m-1} & a_{2,0,m-1} & \cdots & a_{n,0,m-1} \\ a_{1,1,0} & a_{2,1,0} & \cdots & a_{n,1,0} \\ a_{1,1,1} & a_{2,1,1} & \cdots & a_{n,1,1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1,1,m-1} & a_{2,1,m-1} & \cdots & a_{n,1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ a_{1,t-1,0} & a_{2,t-1,0} & \cdots & a_{n,t-1,0} \\ a_{1,t-1,1} & a_{2,t-1,1} & \cdots & a_{n,t-1,1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1,t-1,m-1} & a_{2,t-1,m-1} & \cdots & a_{n,t-1,m-1} \end{pmatrix}.$$

The first index of the entry of $\hat{H}$ corresponds to the representation of the expression $\frac{h}{(x-\alpha_i)}$ mod g over L therefore it runs from 1 to n, the second index corresponds to the representation over $\mathbb{F}_{2^m}$ therefore it runs from 0 to $t-1$ and the third index corresponds to the representation over $\mathbb{F}_2$ therefore it runs from 0 to $m-1$. Note that the matrix $\hat{H}$ will be a binary matrix, that is, its entries will be either 0 or 1.

Finally to obtain the Generator matrix, we can solve the kernel equation $\hat{H} \cdot x = 0$. The basis of the kernel space will then form the rows of the generator matrix. Thus, the generator matrix will also be a binary matrix and therefore, the encoding procedure can now be easily done over $\mathbb{F}_2$. We summarize the encoding procedure in Algorithm 4.

Algorithm 4 Encoding

```
Input: Code Length(n), Goppa Polynomial(g), Word(w)
Output: Generator Matrix(G), Parity Check Matrix($\hat{H}$)
function ENCODING($n, g, w$)
    $(\alpha_1, \alpha_2, \ldots, \alpha_n) \leftarrow$ codeLocators($\mathbb{F}_{2^m}$)
    $H \leftarrow$ parityMatrixExtended($\alpha_1, \alpha_2, \ldots, \alpha_n, g$)
    $\hat{H} \leftarrow$ parityMatrixBase($H$)
    $G \leftarrow$ rightKernelBasis($\hat{H}$)
return $G, \hat{H}$
```

4.2 Decoding

In 1975, Patterson presented a highly efficient algebraic decoding algorithm for Goppa Codes [16]. This algorithm can correct errors up to half of the minimum distance of the code. Since its inception, numerous decoding algorithms have been developed that can correct more errors, such as the list decoding algorithm introduced by Bernstein [15]. Despite these advancements, Patterson's algorithm remains the simplest to implement and possesses a good error-correction capacity. Thus, we have chosen to implement Patterson's decoding algorithm for our demonstration.

We will start by discussing the algorithm's motivation, which is outlined by Patterson in [16]. This algorithm relies on syndrome decoding, which involves determining the error vector based on the syndrome of a received word. The objective is to acquire a polynomial that has roots in the code locators set, which match the bit position of the received word in which an error would have occurred. We can formulate this polynomial using the same approach as we did in Reed-Solomon decoding, given that a maximum of $t-1$ errors can occur (to ensure accurate decoding). The polynomial is defined as follows:

$$\epsilon(x) = \prod_{i=1}^{t-1}(x-\alpha_i).$$

Recall that the syndrome polynomial S for Goppa codes aligns with the defining property of the Goppa codes, as presented in definition 4.0.1. It is given by:

$$S(x) = \sum_{i=1}^{n} \frac{c_i}{(x-\alpha_i)} \mod g.$$

It is worth noting that the two polynomials described above have an interesting connection, which can be expressed using the following equation:

$$\begin{aligned} \epsilon'(x) &= \prod_{i=1}^{t-1}(x-\alpha_i) \sum_{i=1}^{t-1} \frac{1}{(x-\alpha_i)} \\ &= \epsilon(x)S(x) \mod g. \end{aligned} \tag{4.1}$$

The sum in 4.1 is indeed equal to the syndrome S since the codeword $c = (c_1, c_2, \ldots, c_n)$ is a binary vector and after reducing the sum in the definition of syndrome S by modulo g, we obtain the the sum in the above equation.

The connection between the error polynomial and syndrome polynomial described in equation 4.1 is crucial to Patterson's algorithm. Additionally, Patterson utilizes the fact that a polynomial can be split into two parts, one consisting of the terms with even powers of the variable x and the other consisting of the terms with odd powers. This splitting can also be applied to the error-locator polynomial ϵ, yielding the representation given by

$$\epsilon(x) = A^2(x) + xB^2(x),$$

where A is the collection of even powers after taking the common square root and xB^2 corresponds to collection of odd powers. Therefore, B is again a collection of even powers left after factoring out x and taking the common square root.

From this representation of ϵ we obtain that $\epsilon'(x) = B^2(x)$ The other terms in the derivative of ϵ are zero because $\mathbb{F}_{2^m}$ has characteristic two. Thus, using equation 4.1, we obtain the following relation

$$\begin{aligned} B^2(x) &\equiv \epsilon(x)S(x) \mod g \\ &\equiv (A^2(x) + xB^2(x))S(x) \mod g. \end{aligned}$$

This equation implies that

$$A(x) \equiv B(x)v(x) \mod g, \tag{4.2}$$

where $v^2(x) \equiv \left(\frac{1}{S(x)} + x\right) \mod g$.

The equation 4.2 is called the *Key Equation* and its solution gives us the error-locator polynomial ϵ.

Since some of the concepts in the solution above are highly non-trivial, we will give their brief explanations in the following subsections.

4.2.1 Syndrome Calculation

The first step in decoding is to compute the syndrome $S \in \mathbb{F}_{2^m}[x]/g(x)$ of the received word r using the formula given in the Proposition 4.0.3. If the output of this calculation is zero, then it implies no error has corrupted the codeword, and the decoding stops. Otherwise we proceed with the next steps.

4.2.2 Splitting of Polynomials

Risse, in [17], demonstrates a method for splitting a polynomial into even and odd parts. This technique showcases the beauty of finite fields $\mathbb{F}_{2^m}$, where the splitting process is greatly simplified. We can collect all the even terms, reduce their powers by half, and then express them as a whole square, as the mixed terms in the multinomial vanish due to the field's characteristic of 2. The same approach can be taken for the odd terms after factoring out x. For example, if $f \in \mathbb{F}_{2^m}[x]$ such that $f(x) = a_1x + a_2x^2 + a_x^3 + a_4x^4$ then we have:

$$\begin{aligned} f(x) &= a_2x^2 + a_4x^4 + a_1x + a_x^3 \\ &= (a_2x^2 + 2a_2a_4xx^2 + a_4x^4) + x(a_1 + 2a_1a_3x + a_3x^2) \\ &= (\sqrt{a}_2x + \sqrt{a}_4x^2)^2 + x(\sqrt{a}_1 + \sqrt{a}_3x)^2 \\ &= f_e^2 + xf_o^2. \end{aligned}$$

Note that since $2a_2a_4xx^2 \equiv 0$ and $2a_1a_3x \equiv 0$ in $\mathbb{F}_{2^m}$, it had no effect on adding to f and therefore, we easily obtained the desired split form of f. This property help us in finding the error locator polynomial as we saw in the motivation behind Patterson's algorithm.

4.2.3 Inverse in $\mathbb{F}_{2^m}[x]/g(x)$

An important step in solving the decoding problem is inverting the syndrome polynomial, which involves computing $S^{-1} \mod g$. This can be accomplished easily by applying the Extended Euclidean Algorithm (EEA) to the $\gcd(g, S)$. Once we have S^{-1}, we can proceed with finding the polynomial v. However, we still need to determine a method for taking the square roots in $\mathbb{F}_{2^m}[x]/g(x)$ before proceeding further.

4.2.4 Square Root in $\mathbb{F}_{2^m}[x]/g(x)$

The technique of splitting polynomials simplifies the process of finding square roots in $\mathbb{F}2^m[x]/g(x)$. We begin by finding the expression for $\sqrt{x} \mod g$. The trick is to split the factoring polynomial g as $g(x) = g_e^2 + xg_o^2$, and reduce it modulo g.

Therefore, we obtain:

$$\begin{aligned} g_e^2 &\equiv x g_o^2 \mod g \\ \implies x &\equiv g_e^2(x)[g_o^2]^{-1}(x) \mod g \\ \implies \sqrt{x} &\equiv g_e(x) g_o^{-1}(x) \mod g. \end{aligned}$$

Using the expression for $\sqrt{x} \mod g$, we can find square root of any polynomial in $\mathbb{F}_{2^m}[x]/g(x)$ by first splitting it in odd and even parts. Therefore, to obtain the polynomial v in the decoding procedure, where $v^2(x) \equiv \left(\frac{1}{S(x)} + x\right) \mod g$, we use the expression for $\sqrt{x} \mod g$ and obtain:

$$\begin{aligned} v^2(x) &= v_e^2(x) + x v_o^2(x) \\ &\equiv v_e^2(x) + (\sqrt{x} v_o(x))^2 \mod g \\ &\equiv v_e^2(x) + (g_e(x) g_o^{-1}(x) v_o(x))^2 \mod g \\ &\equiv (v_e(x) + g_e(x) g_o^{-1}(x) v_o(x))^2 \mod g. \end{aligned}$$

This implies that $v(x) \equiv v_e(x) + g_e(x) g_o^{-1}(x) v_o(x) \mod g$.

4.2.5 Solving the Key Equation

The next major step in solving the decoding problem is to find the error-locator polynomial by solving the following key equation:

$$A(x) \equiv B(x) v(x) \mod g.$$

In other words, we need to find $A(x)$ and $B(x)$ such that $\deg A(x) \leq \frac{t}{2}$ and $\deg B(x) \leq \frac{t-1}{2}$, where t is the degree of g. There are various methods to solve this equation, such as using lattice basis reduction as suggested by Bernstein in [15], or the Berlekamp-Massey algorithm as described by Patterson in [16], or a modified extended Euclidean algorithm (EEA) as used in [18]. The equivalence of these methods for any alternate codes, such as Goppa codes, is discussed in [19].

To find polynomials A and B that satisfy the aforementioned degree bounds and equation, we adopt an approach similar to that described in [18], using the modified extended Euclidean algorithm (EEA). We begin by setting $r_{-1} = g(x)$ and $r_0 = v(x)$, and then compute the subsequent remainders and polynomials C and B until we obtain a remainder polynomial of degree less than $\frac{t}{2}$. Once we have a polynomial that meets this criterion, we output it as $A(x)$ and the other iteratively calculated polynomials as C and B. Since we do not need to find C, we only output A and B.

With the knowledge of the error-locator polynomial ϵ, the decoding of the received message reduces down to simply finding the roots of this polynomial, determining the indices of these roots in our code locator set and flipping the bits located at those indices of the received message.

To get a better overview of the decoding procedure, we summarize it in Algorithm 5.

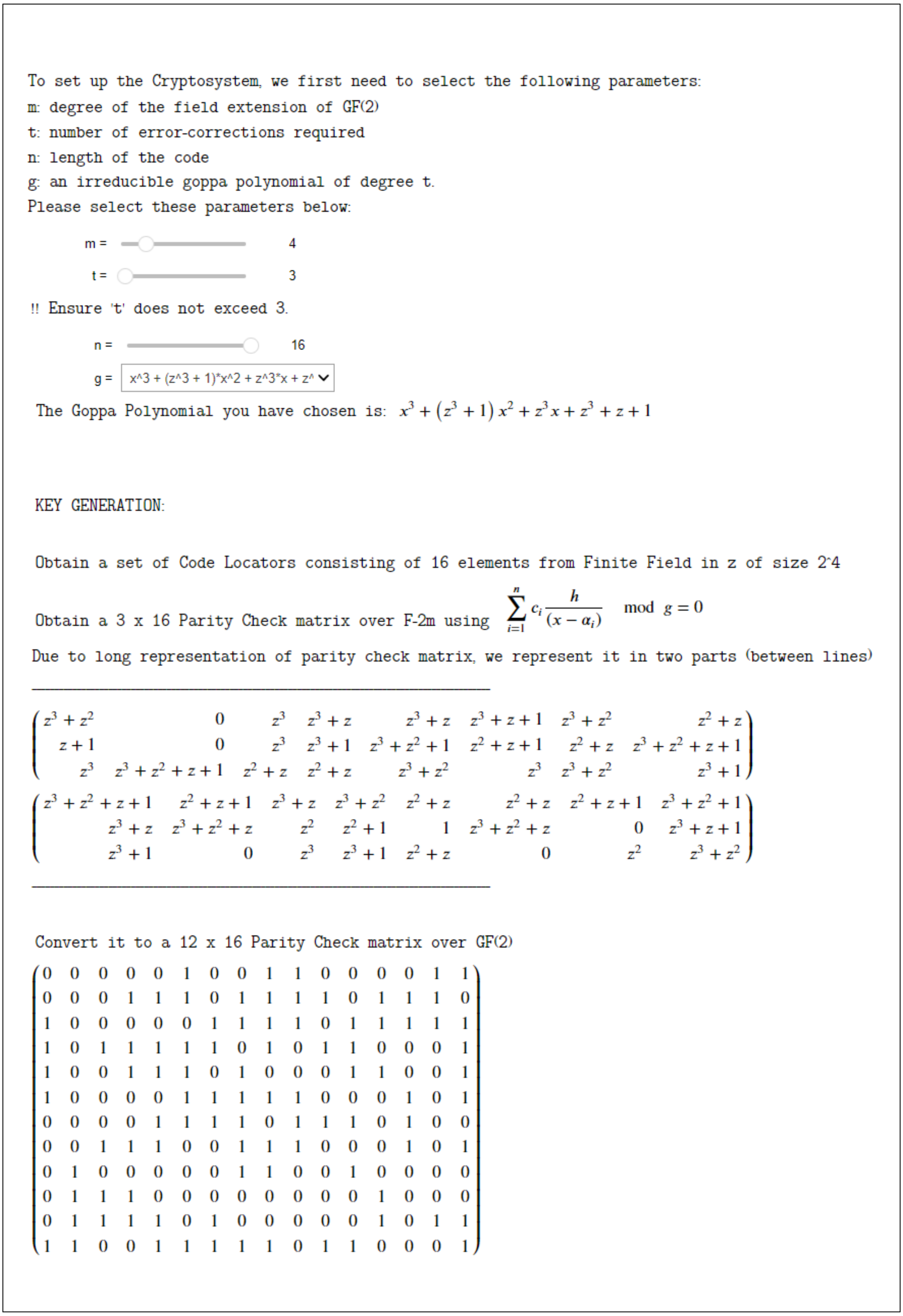
Algorithm 5 Decoding(Patterson)

```
Input: Received Word($r$), Goppa Polynomial($g$), Code Locators($C$)
Output: Decoded word ($d$)
function DECODING($n, g, w$)
    $S \leftarrow$ syndrome($r, g, C$)                ▷ compute the syndrome
    if S == 0 then
        return r.
    $S^{-1} \leftarrow$ XGCD($S, g$)
    if $S^{-1} == x$ then
        $\epsilon \leftarrow x$                        ▷ error polynomial is x
        $roots \leftarrow$ errorPosition($\epsilon, C$)  ▷ index of roots of $\epsilon$ in $C$
        $d \leftarrow$ bitFlip($r, roots$)
        return $d$.
    $v \leftarrow$ sqrt($S^{-1} + x \mod g$)
    $A, B \leftarrow$ keyEquation($g, v$)           ▷ solve key equation
    $\epsilon \leftarrow A^2 + xB^2$
    $roots \leftarrow$ errorPosition($\epsilon, C$)
    $d \leftarrow$ bitFlip($r, roots$)
return d.
```

4.3 Implementing McEliece Cryptosystem based on Goppa Codes

In this section, we will implement a basic McEliece cryptosystem using Goppa Codes, which is the usual choice for this approach due to the reasons discussed previously. As mentioned in Chapter 3, the encryption and decryption procedures remain the same as in any other implementation of McEliece. The only difference is in the encoding and decoding algorithms, which are explained in the previous sections of this chapter. The implementation is done interactively using SageMath. We have included screenshots of one example, but for further testing, the codes included in the appendix can be used.

It should be noted that Goppa Codes are the preferred choice for McEliece cryptosystems due to their higher security compared to other linear codes, as discussed previously.



To set up the Cryptosystem, we first need to select the following parameters:
m: degree of the field extension of GF(2)
t: number of error-corrections required
n: length of the code
g: an irreducible goppa polynomial of degree t.
Please select these parameters below:

m = 4

t = 3

!! Ensure 't' does not exceed 3.

n = 16

g = x^3 + (z^3 + 1)*x^2 + z^3*x + z^

The Goppa Polynomial you have chosen is: $x^3 + (z^3 + 1)x^2 + z^3 x + z^3 + z + 1$

KEY GENERATION:

Obtain a set of Code Locators consisting of 16 elements from Finite Field in z of size 2^4

Obtain a 3 x 16 Parity Check matrix over F-2m using $\sum_{i=1}^{n} c_i \frac{h}{(x-\alpha_i)} \mod g = 0$

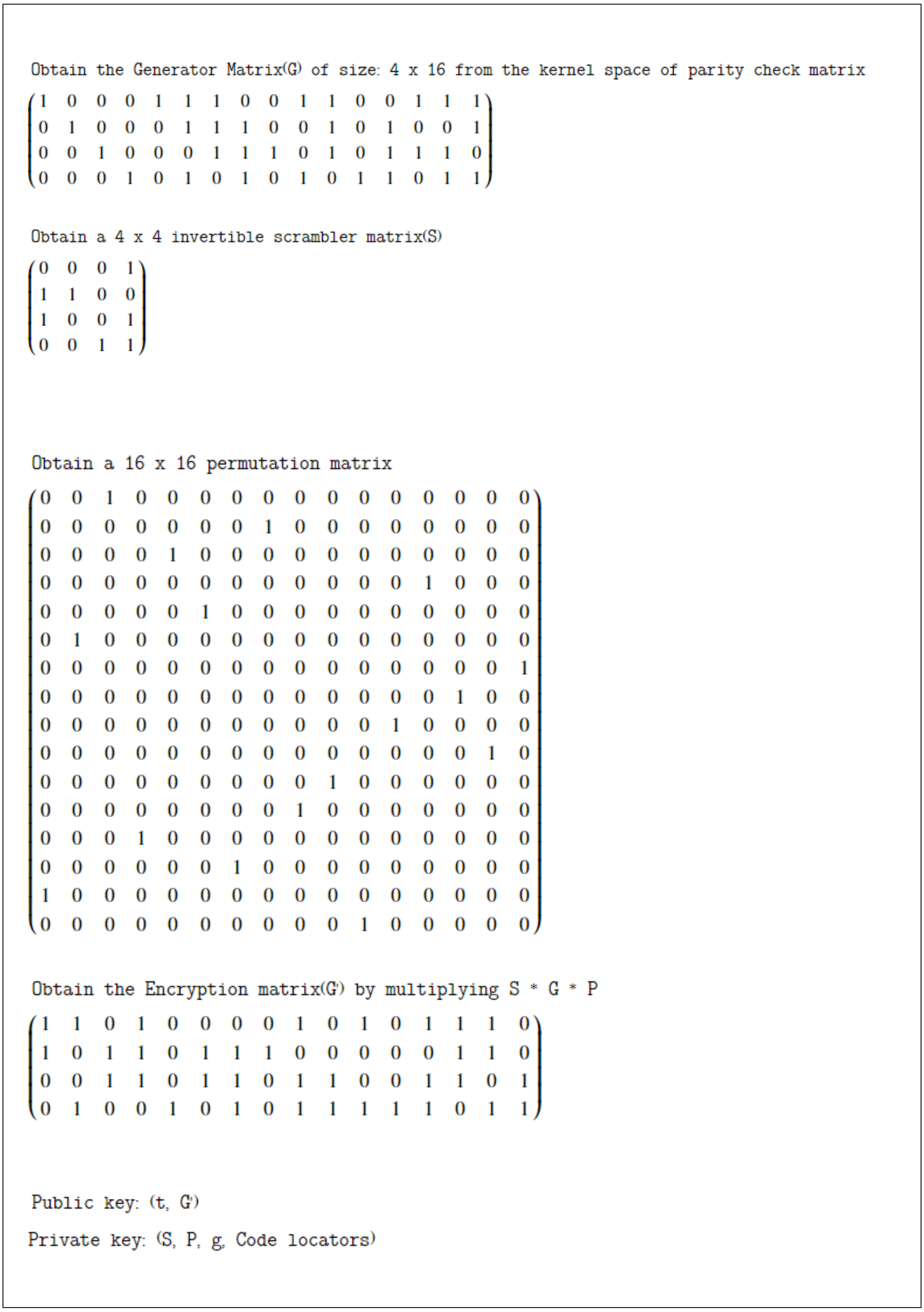
Due to long representation of parity check matrix, we represent it in two parts (between lines)

$$\begin{pmatrix} z^3+z^2 & 0 & z^3 & z^3+z & z^3+z & z^3+z+1 & z^3+z^2 & z^2+z \\ z+1 & 0 & z^3 & z^3+1 & z^3+z^2+1 & z^2+z+1 & z^2+z & z^3+z^2+z+1 \\ z^3 & z^3+z^2+z+1 & z^2+z & z^2+z & z^3+z^2 & z^3 & z^3+z^2 & z^3+1 \end{pmatrix}$$

$$\begin{pmatrix} z^3+z^2+z+1 & z^2+z+1 & z^3+z & z^3+z^2 & z^2+z & z^2+z & z^2+z+1 & z^3+z^2+1 \\ z^3+z & z^3+z^2+z & z^2 & z^2+1 & 1 & z^3+z^2+z & 0 & z^3+z+1 \\ z^3+1 & 0 & z^3 & z^3+1 & z^2+z & 0 & z^2 & z^3+z^2 \end{pmatrix}$$

Convert it to a 12 x 16 Parity Check matrix over GF(2)

$$\begin{pmatrix} 0&0&0&0&0&1&0&0&1&1&0&0&0&0&1&1 \\ 0&0&0&1&1&1&0&1&1&1&1&0&1&1&1&0 \\ 1&0&0&0&0&0&1&1&1&1&0&1&1&1&1&1 \\ 1&0&1&1&1&1&1&0&1&0&1&1&0&0&0&1 \\ 1&0&0&1&1&1&0&1&0&0&0&1&1&0&0&1 \\ 1&0&0&0&0&1&1&1&1&1&0&0&0&1&0&1 \\ 0&0&0&0&1&1&1&1&0&1&1&1&0&1&0&0 \\ 0&0&1&1&1&0&0&1&1&1&0&0&0&1&0&1 \\ 0&1&0&0&0&0&0&1&1&0&0&1&0&0&0&0 \\ 0&1&1&1&0&0&0&0&0&0&0&0&1&0&0&0 \\ 0&1&1&1&1&0&1&0&0&0&0&0&1&0&1&1 \\ 1&1&0&0&1&1&1&1&1&0&1&1&0&0&0&1 \end{pmatrix}$$



Obtain the Generator Matrix(G) of size: 4 x 16 from the kernel space of parity check matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{pmatrix}$$

Obtain a 4 x 4 invertible scrambler matrix(S)

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

Obtain a 16 x 16 permutation matrix

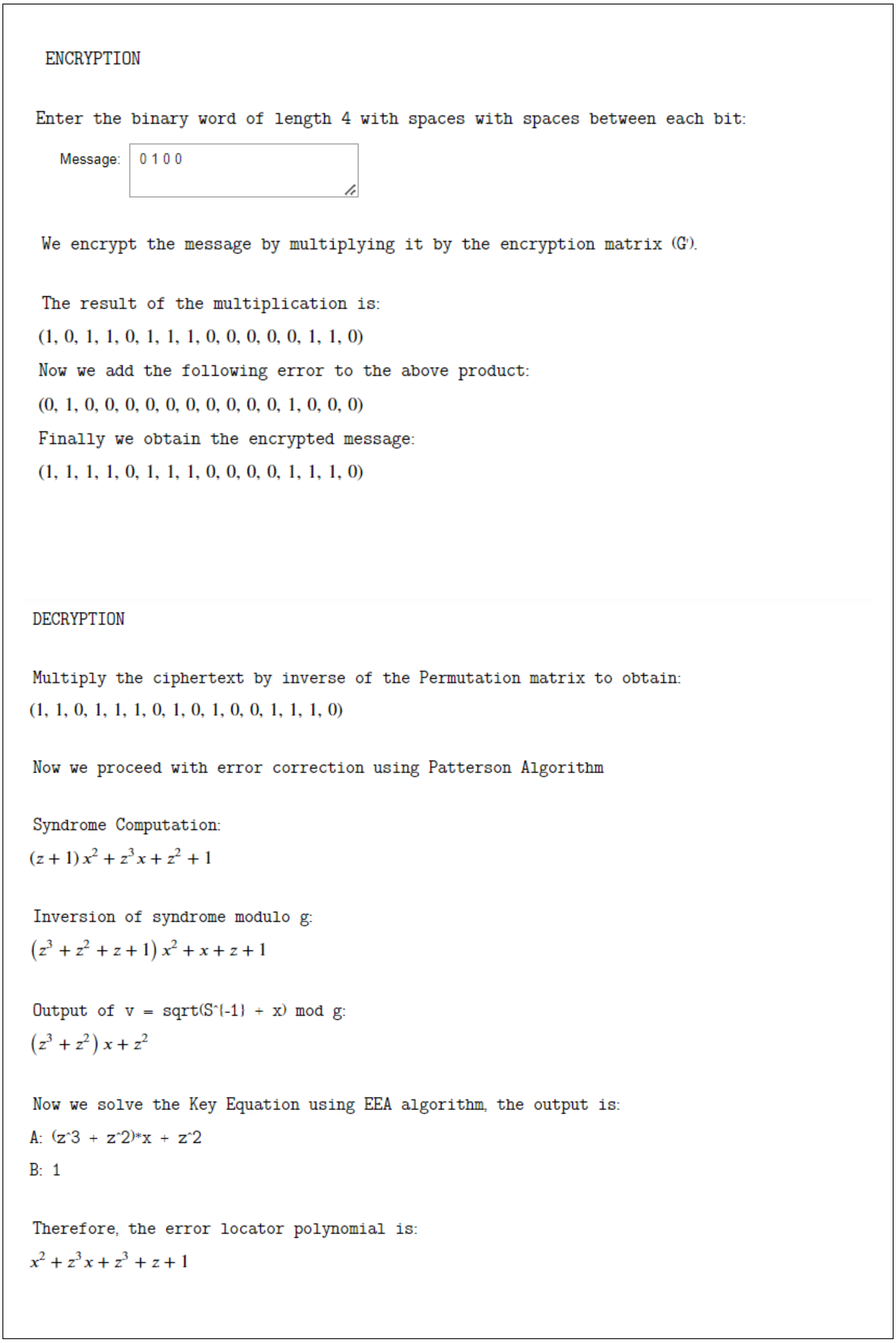
$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Obtain the Encryption matrix(G') by multiplying S * G * P

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \end{pmatrix}$$

Public key: (t, G')

Private key: (S, P, g, Code locators)



ENCRYPTION

Enter the binary word of length 4 with spaces with spaces between each bit:

Message: 0 1 0 0

We encrypt the message by multiplying it by the encryption matrix (G').

The result of the multiplication is:

(1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0)

Now we add the following error to the above product:

(0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0)

Finally we obtain the encrypted message:

(1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0)

DECRYPTION

Multiply the ciphertext by inverse of the Permutation matrix to obtain:

(1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0)

Now we proceed with error correction using Patterson Algorithm

Syndrome Computation:

$(z+1)x^2 + z^3x + z^2 + 1$

Inversion of syndrome modulo g:

$\left(z^3 + z^2 + z + 1\right)x^2 + x + z + 1$

Output of v = sqrt(S^{-1} + x) mod g:

$\left(z^3 + z^2\right)x + z^2$

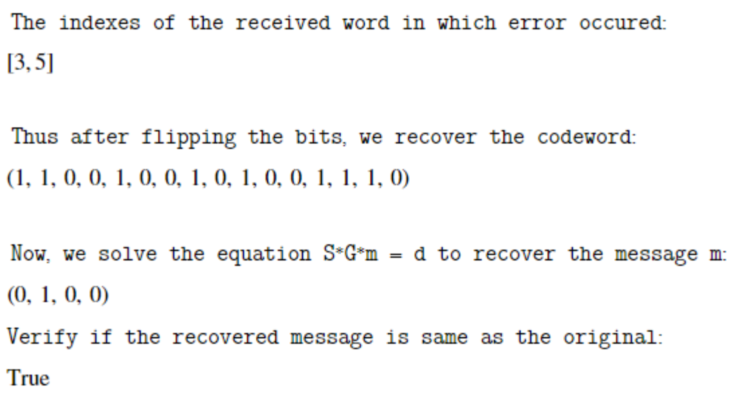
Now we solve the Key Equation using EEA algorithm, the output is:

A: (z^3 + z^2)*x + z^2

B: 1

Therefore, the error locator polynomial is:

$x^2 + z^3x + z^3 + z + 1$



```
The indexes of the received word in which error occured:
[3,5]

Thus after flipping the bits, we recover the codeword:
(1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0)

Now, we solve the equation S*G*m = d to recover the message m:
(0, 1, 0, 0)
Verify if the recovered message is same as the original:
True
```

Figure 4.1: McEliece PKCS using Goppa Codes

4.4 Security of McEliece Cryptosystem

In this section, we will briefly discuss the security aspects of the McEliece cryptosystem. The main security of the cryptosystem lies in the NP-hardness of the general linear code decoding problem [20]. Since the inception of McEliece cryptosystem, several attacks against it have been proposed in the literature. The most famous attack is based on Stern's information set decoding method [21]. This method was used by Bernstein, Lange and Christiane in [22] to practically attack the cryptosystem consisting of small weight codewords. However, a great limitation of this attack is that its complexity is exponential and therefore does not pose a great threat to the cryptosystem if it has been implemented as per correct standards. Some other kinds of attacks have also been proposed such as partial key exposure attack mentioned in [23] in which the secret keys can be recovered if a part of it is leaked. This can be mitigated by carefully designing the cryptosystem to resist side-channel attacks. Moreover, if the underlying linear code is different from Goppa Codes, such as Reed Solomon Codes, the cryptosystem is under threat of efficient (polynomial time) structural attacks as described in [13]. The use of Goppa codes, as opposed to other linear codes like Reed Solomon codes, is also important for avoiding efficient structural attacks. Despite existence of these and other attacks, McEliece cryptosystem (with some modifications) largely remains secure against practically feasible attacks. This makes it a promising candidate for post-quantum cryptographic procedures. In fact, a variant of McEliece Cryptosystem was selected for the third round of NIST's Post-Quantum Cryptography Standardization Process [24] and has been combined with the classic McEliece [25]. This merged project has qualified for the fourth round as well [26].

Chapter 5

Conclusion

In this thesis, we have provided a simplified explanation of the McEliece cryptosystem and its implementation using interactive SageMath. We began by introducing the basics of cryptography and public key cryptography, followed by a discussion on error correcting codes and their implementation in SageMath.

Next, we introduced the McEliece cryptosystem and presented its key generation, encryption, and decryption algorithms, along with a screenshot of its implementation using Reed Solomon codes. We then explored binary Goppa codes, the usual approach for implementing the McEliece cryptosystem. We explained their encoding and decoding algorithms and provided a screenshot of the implementation of the McEliece cryptosystem using Goppa codes.

Our main goal in this thesis was to provide a simplified explanation of the McEliece cryptosystem and its implementation using interactive SageMath. We hope that this thesis has been successful in achieving this goal, and that it has helped the readers to gain a deeper understanding of the McEliece cryptosystem.

Bibliography

[1] H. Sidhpurwala. A brief history of cryptography. `https://www.redhat.com/en/blog/brief-history-cryptography`, 2021.

[2] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[3] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[4] Sz. Tengely. The RSA algorithm. Lecture Notes on Cryptography.

[5] P. W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[6] T. Bell. Check digits on product barcodes. Computer Science Field Guide.

[7] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*, volume 16. Elsevier, 1977.

[8] J. L. Walker. *Codes and Curves*, volume 7. American Mathematical Soc., 2000.

[9] R. J. McEliece. A Public-Key Cryptosystem Based On Algebraic Coding Theory. *Deep Space Network Progress Report*, 44:114–116, January 1978.

[10] Y. X. Li, D. X. Li, and C. K. Wu. How to generate a random nonsingular matrix in McEliece's public-key cryptosystem. In *[Proceedings] Singapore ICCS/ISITA92*, pages 268–269. IEEE, 1992.

[11] I. S. Reed and G. Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[12] L. R. Welch and E. R. Berlekamp. *Error Correction for Algebraic Block Codes*. Springer, 1986.

[13] V. M. Sidelnikov and S.O. Shestakov. On insecurity of cryptosystems based on generalized Reed-Solomon codes. *Discrete Mathematics and Applications*, 2(4):439–444, 1992.

[14] V. D. Goppa. A new class of linear error-correcting codes. *Probl. Inf. Transm.*, 6:300–304, 1970.

[15] D. J. Bernstein. List decoding for binary Goppa codes. In *Coding and Cryptology: Third International Workshop, IWCC 2011, Qingdao, China, May 30-June 3, 2011. Proceedings 3*, pages 62–80. Springer, 2011.

[16] N. J. Patterson. The algebraic decoding of Goppa codes. *IEEE Transactions on Information Theory*, 21(2):203–207, 1975.

[17] T. Risse. How SAGE helps to implement Goppa codes and McEliece PKCSs. *Trans. Inform. Theory. Vol. IT*, 15(1):122–127, 1969.

[18] Y. Sugiyama, M. Kasahara, S. Hirasawa, and T. Namekawa. A method for solving key equation for decoding Goppa codes. *Information and Control*, 27(1):87–99, 1975.

[19] K. Abdelmoumen, H. Ben-Azza, and A. Otmani. Pade approximants and key lattices for decoding alternant codes. *Gulf Journal of Mathematics*, 4(4), 2016.

[20] E. R. Berlekamp, R. J. McEliece, and H. Van Tilborg. On the inherent intractability of certain coding problems (corresp.). *IEEE Transactions on Information Theory*, 24(3):384–386, 1978.

[21] J. Stern. A method for finding codewords of small weight. *Coding theory and applications*, 388:106–113, 1989.

[22] D. J. Bernstein, T. Lange, and C. Peters. Attacking and defending the McEliece cryptosystem. In *Post-Quantum Cryptography: Second International Workshop, PQCrypto 2008 Cincinnati, OH, USA, October 17-19, 2008 Proceedings 2*, pages 31–46. Springer, 2008.

[23] E. Kirshanova and A. May. Breaking Goppa-Based McEliece with Hints. *IACR Cryptology ePrint Archive*, 2022:525, 2022.

[24] M. Albrecht, C. Cid, K.G. Paterson, C.J. Tjhai, and M. Tomlinson. Nts-kem, 2019. *NIST Post-Quantum Cryptography Project: Second Round Candidate Algorithms. Available online: https://csrc. nist. gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions (accessed on 27 September 2019)*, 2020.

[25] T. Chou, C. Cid, S. UiB, J. Gilcher, T. Lange, V. Maram, R. Misoczki, R. Niederhagen, K.G. Paterson, and E. Persichetti. Classic McEliece: conservative code-based cryptography, 10 October 2020, 2020.

[26] G. Alagic, D. Apon, D. Cooper, Q. Dang, T. Dang, J. Kelsey, J. Lichtinger, C. Miller, D. Moody, R. Peralta, et al. Status report on the third round of the NIST post-quantum cryptography standardization process. *US Department of Commerce, NIST*, 2022.

Appendix A

Introductory Codes

A.1 Caesar Cipher

```
#Caesar cipher
def caesar_cipher_encryption(message, k):
    m = message.split()
    c = []
    for word in m:
        encryptedWord = ""
        for j in range(len(word)):
            encryptedWord += chr(97 + ((ord(word[j]) - 97) + k) % 26)
        c.append(encryptedWord)
    return "␣".join(c)
def caesar_cipher_decryption(ciphertext, k):
    c = ciphertext.split()
    m = []
    for word in c:
        decryptedWord = ""
        for j in range(len(word)):
            decryptedWord += chr(97 + ((ord(word[j]) - 97) - k) % 26)
        m.append(decryptedWord)
    return "␣".join(m)
@interact
def caesar_cipher(k=slider(vmin=1, vmax=10, default=3, label="shift:␣"),
                  m=input_box(default="hello", height=5, type=str, label="message:␣")):
    cipherText = caesar_cipher_encryption(m, k)
    pretty_print(f"Encrypted␣message␣=␣\t", cipherText)
    pretty_print(f"Notice␣that␣each␣letter␣has␣been␣shifted␣by␣{k}␣letters!␣\n")
    plainText = caesar_cipher_decryption(cipherText, k)
    pretty_print(f"The␣plaintext␣is␣recovered␣by␣simply␣doing␣the␣reverse␣shift␣by␣{k}␣letters.")
    pretty_print(f"Recovered␣message␣=␣\t", plainText)
```

A.2 RSA

```
@interact
def RSA(p=input_box(label="p:␣", default=311), q=input_box(label="q:␣", default=733),
        m=input_box(label="message:␣", default=1729)):
    #key generation
    n = p * q
    phi = (p-1) * (q-1)
    e = max(p, q) + 2
    while not e.is_prime():
        e = ZZ(randint(max(p, q) + 2, max(p, q) + 100))
    d = inverse_mod(e, phi)
    public_key = (n, e)
    private_key = (d, p, q)
    pretty_print(f"Public␣Key:␣{public_key}")
```

```
    pretty_print(f"Private␣Key:␣{private_key}")
    #encryption
    c = power_mod(m, e, n)
    #decryption
    r = power_mod(c, d, n)
    pretty_print(f"Message:␣{m}")
    pretty_print(f"Ciphertext:␣{c}")
    pretty_print(f"Recovered␣message:␣{r}")
```

A.3 Basic Linear Code

```
pretty_print("\n\nSelect␣the␣finite␣field,␣length␣of␣the␣code␣and␣the␣dimension␣of␣the␣code␣below:␣")
@interact
def linear_code1(f=selector([2, 3, 4, 5, 7], label="Base␣Field:␣"),
                n=slider(vmin=2, vmax=7, default=5, label="Length:␣"),
                k=slider(vmin=1, vmax=5, default=3, label="Dimension:␣")):


    #genearte a list of messages and defualt matrix
    F = GF(f)
    messages = [[choice(F.list()) for j in range(k)] for i in range(5)]
    G0 = scramblerMatrix(F, k, n)
    pretty_print("\n␣Give␣entries␣of␣the␣Generator␣matrix,␣ensuring␣that␣the␣rows␣form␣a␣\n␣linearly␣independent␣set␣of␣vectors.\n")
    @interact
    def linear_code2(w=input_grid(k, n, default=list(map(list, G0.rows())), label="Generator␣Matrix:␣"),
        m=selector(messages, label="Message:␣"), e=input_box(type=str, label="Error:␣")):

        #generate Code and G, H matrices
        G = matrix(F, [list(map(F, w[i])) for i in range(k)])
        C = codes.LinearCode(G)
        H = C.parity_check_matrix()

        #print Code and G, H matrices
        pretty_print(f"You␣have␣generated␣{C}␣with␣minimum␣distance␣d␣=␣{C.minimum_distance()}.␣\nThus,␣it␣is␣capable␣of␣correcting␣errors␣upto:")
        var('d')
        pretty_print("floor␣of␣\t", (d - 1) / 2, "\t␣=␣\t", floor((C.minimum_distance() - 1) / 2))

        print()

        pretty_print("The␣Generator␣matrix␣of␣the␣code␣is:␣")
        pretty_print(G)
        print()
        pretty_print("Systematic␣form␣of␣generator␣matrix:")
        pretty_print(C.systematic_generator_matrix())
        print()

        pretty_print(f"The␣corresponding␣parity␣check␣matrix␣of␣the␣linear␣code␣is:␣")
        pretty_print(H)

        print()

        #Encoding
        pretty_print("Please␣select␣a␣message␣from␣the␣dropdown␣above␣for␣encoding!")
        codeword = vector(F, m) * G
        pretty_print(f"The␣codeword␣corresponding␣to␣your␣chosen␣message␣is:␣\t{codeword}")
        pretty_print(f"Note␣that␣the␣message␣was␣of␣length␣{k}␣while␣the␣codeword␣is␣of␣length␣{n}␣\n\n")
        #Error introduction
        #print statement to copy codeword and enter error
        no_errors = (C.minimum_distance() - 1) // 2
        try:
            errorword = vector((map(F, e.split())))
            pretty_print(f"Original␣Message:␣{vector(F,␣m)}")
            pretty_print(f"Code␣Word:␣{codeword}")
            pretty_print(f"Error␣Word:␣{errorword}")

            #Decoding
            try:
                decoded_codeword = C.decode_to_code(errorword)
                pretty_print(f"Decoded␣codeword:␣{decoded_codeword}")
            except:
```

```
                pass
            decoded_message = C.decode_to_message(errorword)
            pretty_print(f"Decoded␣message:␣{decoded_message}")
        except:
            pretty_print(f"Please␣copy␣the␣code␣word␣and␣change␣upto␣{no_errors}␣digits.")
```

A.4 Reed-Solomon Code

```
def scramblerMatrix(base, rows, cols):
    """to generate scrambler matrix(S), set rows=cols"""
    V = base**cols
    vectors = []
    for i in range(rows):
        v = V.random_element()
        while v in V.span(vectors):
            v = V.random_element()
        vectors.append(v)
    S = matrix(vectors)
    return S

def permutationMatrix(size, base):
    """generates permutation matrix(P) of a given size over the 'base' field"""
    P = (Permutations(size).random_element().to_matrix()).change_ring(base)
    return P


text1 = """\nWe begin by choosing the Finite Field, the Length of the Code and the Dimension of the Code.
Please select these parameters below: """
pretty_print(text1)
@interact
def reed_solomon1(f=input_box(default=13, type=Integer, label="Field␣size:␣"), #only to be used with prime fields
                  n=input_box(default=12, type=Integer, label="Length:␣"),
                  m=input_box(default=3, type=Integer, label="Dimension:␣")):

    #generate code
    F.<z> = GF(f)
    PR.<x> = PolynomialRing(F)
    C = codes.ReedSolomonCode(F, n, m)
    pretty_print(f"\nYou␣have␣generated␣a␣{C}.")
    pretty_print(f"\nThe␣Generator␣matrix(G)␣of␣the␣code␣is:␣")
    G = C.generator_matrix()
    pretty_print(G)

    #Key generation
    text2 = """\nFirst we generate the Public and Private keys."""
    pretty_print(text2)
    P = permutationMatrix(n, F)
    S = scramblerMatrix(F, m, m)
    disguisedMatrix = S * G * P
    pretty_print("\nPermutation␣Matrix(P):␣")
    pretty_print(P)
    pretty_print("\nScrambler␣Matrix(S):␣")
    pretty_print(S)
    pretty_print("\nDisguised␣Matrix(G'):␣")
    pretty_print(disguisedMatrix)
    d = C.minimum_distance()
    t = (d-1) // 2
    pretty_print(f"\nError-Correction␣capacity:␣{t}.")
    pretty_print(f"\nThe␣primitive␣element␣of␣the␣field␣chosen␣is:␣p␣=␣{F.primitive_element()}")
    text3 = """\nThen the keys are given by: \nPublic Keys = (G', t) \nPrivate Keys = (S, G, P, p )\n\n"""
    pretty_print(text3)

    #Encryption
    pretty_print("Encryption:␣\n")
    message = ",␣".join(map(str, [F.random_element() for i in range(m)]))
    errorList = [choice(F.list()) for i in range(t)] + [0 for j in range(n-t)]
    shuffle(errorList)
    error = ",␣".join(map(str, errorList))
    pretty_print(f"\nEnter␣the␣message␣of␣length␣{m}.")
    pretty_print(f"Enter␣the␣error␣vector,␣ensuring␣it␣contains␣at␣most␣{t}␣non-zero␣elements,\n␣an␣example␣has␣already␣been␣generated␣for␣you:␣")

```

```
@interact
def reed_solomon2(w=input_box(default=message, type=str, label="message:␣"),
                  e=input_box(default=error, type=str, label="Errors:␣")):

    message = vector(F, map(F, w.split(",␣")))
    error = vector(F, map(F, e.split(",␣")))
    cipherText = message*disguisedMatrix + error
    pretty_print(f"\nYour␣encrypted␣message(c)␣is:␣\n\n")
    pretty_print(cipherText)

    #Decryption
    pretty_print("\nDecryption:␣\n")

    y = cipherText*P.inverse()
    pretty_print(LatexExpr(r"c␣\cdotp␣P^{-1}␣=␣"))
    pretty_print(y)

    pretty_print(f"\nGenerate␣Vandermonde␣Matrix(V)␣of␣size␣{n}.")
    m0 = matrix.vandermonde([(F.primitive_element())^k for k in range(n)], ring=F)
    pretty_print(m0)

    m1Columns = n - t
    m1 = m0[:,0:m1Columns]
    pretty_print(f"\nFirst␣{m1Columns}␣columns␣of␣V:␣")
    pretty_print(m1)
    for k in range(f - m1Columns):
        v1 = m1.column(k)
        m1 = m1.augment(matrix(F, v1.pairwise_product(y)).transpose())
        #m1=m1.augment(v1.pairwise_product(y))

    pretty_print(f"\nAugmented␣Matrix␣in␣reduced␣echolon␣form:␣")
    pretty_print(m1.rref())

    poly = - m1.rref().column(-1)
    pretty_print(f"\nChoose␣the␣last␣column␣and␣multiply␣by␣-1(mod␣{f}):␣")
    pretty_print(poly)

    Q0 = PR(list(poly)[:m1Columns])
    pretty_print(f'\nForm␣a␣polynomial(P1)␣with␣first␣{m1Columns}␣entries␣as␣
        coefficients:␣')
    pretty_print(Q0)

    Q1 = PR(list(poly)[m1Columns:] + [1])
    pretty_print(f'\nForm␣another␣polynomial(P2)␣with␣last␣{n-m1Columns}␣entries␣as␣
        coefficients:␣')
    pretty_print(Q1)

    Q3 = - Q0.quo_rem(Q1)[0]
    pretty_print('\nFind␣the␣quotient␣of␣P1␣/␣P2␣\n')
    pretty_print(Q3)

    pretty_print(f'Multiply␣by:␣\t', LatexExpr(r"S^{-1}"))
    recoveredText = matrix(F, list(Q3))*S.inverse()
    pretty_print("\nYour␣decrypted␣message␣is:␣")
    pretty_print(recoveredText)
```

Appendix B

Binary Goppa Codes

B.1 Algorithms

```python
def goppaPolynomial(F, t):
    """generate a set of 5 irreducible goppa polynomials"""
    P.<x> = PolynomialRing(F)
    S = Set([])
    k = 0
    while S.cardinality()<5 and k<10:
        S = S.union(Set([P.irreducible_element(t)]))
        k += 1
    return S

def parityMatrixGCExt(F, n, g):
    """generate Parity check matrix H over extended field
       F: extended field in z
       n: length of code
       g: irreducible goppa polynomial"""
    #initialization
    P.<x> = PolynomialRing(F)
    t = g.degree()

    #generate code locators of n field elements
    code_locators = Permutations(F.list()[1:n-1] + [F(1), F(0)], n).random_element()

    #generate polynomial h with roots in code_locators
    h = F(1)
    for i in code_locators:
        h *= (x - i)

    #generate column vectors
    v = []
    for i in code_locators:
        v.append((h * inverse_mod((x - i), g)).mod(g))

    #generate H
    H = matrix(F, t, n)
    for j in range(n):
        element = list(v[j]) #convert poly to list
        for i in range(t):
            if i < len(element):
                H[i, j] = element[i]
            else:
                H[i, j] = 0
    return H, code_locators

def polyToVector(F, p):
    """Converts polynomial over F to a vector over base of F
       F: extended field
       p: polynomial
       """
    P.<z> = PolynomialRing(F.base())
```

```
    f = P(p)
    coeff = f.list()
    m = F.degree()
    v = []
    for i in range(m):
        if i < len(coeff):
            v.append(coeff[i])
        else:
            v.append(0)
    return vector(F.base(), v)

def parityMatrixGCBase(F, H):
    """Expand parity matrix H over extended field to a matrix over base field
    F: extended field
    H: matrix over F"""
    m = F.degree()
    n = H.ncols()
    k = H.nrows()
    H_base = matrix(F.base(), m*k, n)
    for j in range(n):
        r = 0
        for i in range(k):
            H_base[r:r+m, j] = polyToVector(F, H[i, j])
            r += m       #update row index
    return H_base

def generatorMatrixGCBase(m, t, n, H):
    """generate Generator Matrix(G) over GF(2)
        k: nrows = m*t (dimension of code)
        n: ncols (length of code)
        H: parity check matrix over GF(2)
        """
    H_kernel_basis = H.right_kernel().basis() #define right kernel basis
    G = matrix(GF(2), H_kernel_basis[:n-m*t]) #choose n-mt vectors from basis
    return G

#Decryption
def hammingDistance(v1, v2):
    """Computes Hamming Distance between two vectors v1 and v2"""
    while len(v1) < len(v2):
        v1.append(0)
    while len(v2) < len(v1):
        v2.append(0)
    count = 0
    for i in range(len(v1)):
        if v1[i] != v2[i]:
            count += 1
    return count

def syndrome(m, w, g, support):
    """Compute the syndrome of the word as per definition of Goppa Code"""
    F.<z> = GF(2^m)
    P.<x> = PolynomialRing(F)
    s = [w[i] * inverse_mod((x - support[i]), g) for i in range(len(w))]
    return sum(s)

def sqrtX(m, g):
    """compute square root of x modulo g in extended field"""
    F.<z> = GF(2^m)
    P.<x> = PolynomialRing(F)
    g0_list = g.list()[:len(g.list()):2]
    g1_list = g.list()[1:len(g.list()):2]
    g0 = sum([sqrt(g0_list[i])*x^(i) for i in range(len(g0_list))])
    g1 = sum([sqrt(g1_list[i])*x^(i) for i in range(len(g1_list))])
    g1_inv = xgcd(g1, g)[1]
    root_x = (-g0*g1_inv).mod(g)
    #root_x = (g0*g1_inv).mod(g)
    return root_x

def sqrtP(m, S, g, root_x):
    """compute square root of any polynomial p mod g in extended field"""
    F.<z> = GF(2^m)
    P.<x> = PolynomialRing(F)
    h = inverse_mod(S, g) + x
    h0_list = h.list()[:len(h.list()):2]
    h1_list = h.list()[1:len(h.list()):2]
    h0 = sum([sqrt(h0_list[i])*x^(i) for i in range(len(h0_list))])
    h1 = sum([sqrt(h1_list[i])*x^(i) for i in range(len(h1_list))])
    v = (h0 + root_x * h1).mod(g)
    return v
```

```

def keyEquation(m, v, g):
    """Extended Euclidean Algorithm to solve key equation v*B = a mod g"""
    F.<z> = GF(2^m)
    P.<x> = PolynomialRing(F)
    t = g.degree()

    # Initialize r_0 = g, r_1 = v, u_0 = 0, u_1 = 1
    r_0 = g
    r_1 = v
    u_0 = P(0)
    u_1 = P(1)

    # Repeat until deg(r_1) >= n/2
    while r_1.degree() >= t/2:
        q, r = r_0.quo_rem(r_1)

        # Update r_0 = r_1, r_1 = R
        r_0 = r_1
        r_1 = r

        # Update u_0 = u_0 - Q*u_1 mod g
        u_0 = u_0 - q * u_1.mod(g)

         # Swap u_0 and u_1
        u_0, u_1 = u_1, u_0
    return r_1, u_1

def errorPosition(m, error_polynomial, code_locators):
    """find error positions"""
    F.<z> = GF(2^m)
    P.<x> = PolynomialRing(F)
    errors = [code_locators[i] for i in range(len(code_locators)) if
        error_polynomial(code_locators[i]) == 0]
    error_positions = [code_locators.index(errors[i]) for i in range(len(errors))]
    return error_positions

def bitFlip(error_positions, w):
    """correct errors in the received word w"""
    for i in range(len(error_positions)):
        w[error_positions[i]] = (w[error_positions[i]] + 1) % 2
    return w

#Patterson's decoding algorithm
def pattersonDecoding(m, received_word, goppa_polynomial, code_locators):
    """Patterson's Goppa Code decoding algorithm"""
    F.<z> = GF(2^m)
    P.<x> = PolynomialRing(F)

    #compute the syndrome
    S = syndrome(m, received_word, goppa_polynomial, code_locators)
    if S == 0:
        return "Codeword"

    #checking if error polynomial is linear
    h = inverse_mod(S, goppa_polynomial)
    if (h / h.list()[-1]) == x:
        error_positions = code_locators.index(F(0))
        received_word[error_positions] = (received_word[error_positions] + 1) % 2
        #pretty_print(f"The inverse polynomial is: {h}\n")
        return received_word

    #compute sqrt(x)
    root_x = sqrtX(m, goppa_polynomial)
    #compute sqrt(1/S + x)
    v = sqrtP(m, S, goppa_polynomial, root_x)

    #solve key equation dB = A mod g
    a0, b0 = keyEquation(m, v, goppa_polynomial)
    if (a0, b0) == ('e', 'e'):
        return "BadGoppaPolynomial"

    #define monic error locator polynomial
    error_polynomial = a0^2 + x* b0^2
    error_polynomial = error_polynomial / error_polynomial.list()[-1]

    #find error position and decode
    error_positions = errorPosition(m, error_polynomial, code_locators)
    received_word_copy = received_word[:]
    decodeword = bitFlip(error_positions, received_word_copy)
```

```
    return decodeword, S, h, v, a0, b0, error_polynomial, error_positions

# McEliece
def scramblerMatrix(base, rows, cols):
    """to generate scrambler matrix(S), set rows=cols"""
    V = base**cols
    vectors = []
    for i in range(rows):
        v = V.random_element()
        while v in V.span(vectors):
            v = V.random_element()
        vectors.append(v)
    S = matrix(vectors)
    return S

def permutationMatrix(size, base):
    """generates permutation matrix(P) of a given size over the 'base' field"""
    P = (Permutations(size).random_element().to_matrix()).change_ring(base)
    return P

def keyGeneration(m, t, n, g):
    """key generation algorithm for McEleice Cryptosystem"""
    F.<z> = GF(2^m)
    P.<x> = PolynomialRing(F)

    #generate parity and generator matrices
    H, support = parityMatrixGCExt(F, n, g)
    H_base = parityMatrixGCBase(F, H)
    G = generatorMatrixGCBase(m, t, n, H_base)

    #generate scrambler and permutation matrices
    S = scramblerMatrix(GF(2), n-m*t, n-m*t)
    P = permutationMatrix(n, GF(2))

    #compute the disguised encryption matrix
    disguised_matrix = S * G * P
    return disguised_matrix, S, P, support, G, H, H_base

def encryption(word, disguised_matrix, t, n):
    """Encryption algorithm for McEliece Cryptosystem"""

    #generate error vector
    errors = [1 for i in range(t-1)] + [0 for j in range(n-t+1)]
    p = Permutations(errors)
    error_vector = vector(GF(2), p.random_element())

    #encrypt message
    pre_ciphertext = word * disguised_matrix
    ciphertext = pre_ciphertext + error_vector
    return ciphertext, error_vector, pre_ciphertext

def decryption(m, ciphertext, polynomial, code_locators, P, S, generator_matrix):
    """Decryption algorithm for McEliece Cryptosytem"""
    F.<z> = GF(2^m)
    PR.<x> = PolynomialRing(F)

    #reverse the permutation
    inverse_permutation = ciphertext * P.inverse()

    #decode the errors
    patterson_outcome, syndrome, inv_syndrome, v, a0, b0, epsilon, error_positions =
         pattersonDecoding(m, list(inverse_permutation), polynomial, code_locators)
    if patterson_outcome == "Codeword":
        return "There␣is␣no␣error␣in␣the␣received␣word."
    decodedword = vector(GF(2), patterson_outcome)

    #decrypt the received word
    plaintext = (S * generator_matrix).solve_left(decodedword)
    return plaintext, inverse_permutation, syndrome, inv_syndrome, v, a0, b0, epsilon,
         error_positions, decodedword
```

B.2 Interactive McEliece using Goppa Codes

```
#run Goppa Code Algorithms

text1 = ”””
To set up the Cryptosystem, we first need to select the following parameters:
m: degree of the field extension of GF(2)
t: number of error−corrections required
n: length of the code
g: an irreducible goppa polynomial of degree t.
Please select these parameters below: ”””
pretty_print(text1, figsize=[60, 10])

@interact
def mcEliece1(m=slider(vmin=3, vmax=8, step_size=1, default=4, label=’m␣=’),
              t=slider(vmin=2, vmax=32, step_size=1, default=3, label=’t␣=’)):

    pretty_print(f”!!␣Ensure␣’t’␣does␣not␣exceed␣{(2^m␣−1)␣//␣m}.”)

    F.<z> = GF(2^m)
    P.<x> = PolynomialRing(F)

    @interact
    def mcEliece3(n=slider(vmin=m*t, vmax=2^m, step_size=1, default=2^m, label=’n␣=’),
                  g=selector(list(goppaPolynomial(F, t)), label=’g␣=’)):
        pretty_print(”The␣Goppa␣Polynomial␣you␣have␣chosen␣is:␣\t”, g)

        # Generate keys
        pretty_print(”\n\nKEY␣GENERATION:”)
        disguised_matrix, scrambler_matrix, permutation_matrix, code_locators,
             generator_matrix, parity_ext,
 parity_base = keyGeneration(m, t, n, g)

        pretty_print(f”\nObtain␣a␣set␣of␣Code␣Locators␣consisting␣of␣{n}␣elements␣from␣{F}”)

        pretty_print(f”\nObtain␣a␣{t}␣x␣{n}␣Parity␣Check␣matrix␣over␣F_2m␣using\t”,
 LatexExpr(r”\sum_{i=1}^{n}␣c_i\frac{h}{(x␣−␣\alpha_i)}\mod␣g␣=␣0␣”))
        pretty_print(”Due␣to␣long␣representation␣of␣parity␣check␣matrix,␣we␣represent␣it␣in␣
             two␣parts
(between␣lines)”)

        pretty_print(”——————————————————————————————————————————————————————————————
————————————————”)

          pretty_print(parity_ext[:,:8])
        pretty_print(parity_ext[:,8:])

          pretty_print(”——————————————————————————————————————————————————————————————————
——————————————”)

          #pretty_print(parity_ext)
        pretty_print(f”\nConvert␣it␣to␣a␣{m*t}␣x␣{n}␣Parity␣Check␣matrix␣over␣GF(2)”)
        pretty_print(parity_base)

        pretty_print(f”\nObtain␣the␣Generator␣Matrix(G)␣of␣size:␣{n−m*t}␣x␣{n}␣from
the␣kernel␣space␣of␣parity␣check␣matrix”)

          pretty_print(generator_matrix)

        pretty_print(f”\nObtain␣a␣{n−m*t}␣x␣{n−m*t}␣invertible␣scrambler␣matrix(S)”)
        pretty_print(scrambler_matrix)

        pretty_print(f”\nObtain␣a␣{n}␣x␣{n}␣permutation␣matrix”)
        pretty_print(permutation_matrix)

        pretty_print(”\nObtain␣the␣Encryption␣matrix(G’)␣by␣multiplying␣S␣*␣G␣*␣P”)
        pretty_print(disguised_matrix)

        pretty_print(”\n\nPublic␣key:␣(t,␣G’)”)
        pretty_print(”Private␣key:␣(S,␣P,␣g,␣Code␣locators)”)

        #encryption
        pretty_print(”\n\n␣ENCRYPTION”)
        pretty_print(f”\nEnter␣the␣binary␣word␣of␣length␣{n␣−␣m*t}␣with␣spaces␣with␣spaces␣
             between␣each␣bit:␣”)
        value_plaintext = ”␣”.join(map(str, [randint(0,1) for i in range(n − m*t)]))

        @interact
        def mcEliece4(plaintext=input_box(default=value_plaintext, type=str,  label=’Message:␣
```

```
                  ',
 width=100, height=5)):

                  pretty_print("\nWe encrypt the message by multiplying it by the encryption 
                      matrix (G').")
                 word = vector(GF(2), map(int, plaintext.split()))
                 ciphertext, error, pre_ciphertext = encryption(word, disguised_matrix, t, n)
                 pretty_print(f"\nThe result of the multiplication is: ")
                 pretty_print(pre_ciphertext)
                 pretty_print("Now we add the following error to the above product: ")
                 pretty_print(error)
                 pretty_print(f'Finally we obtain the encrypted message: ')
                 pretty_print(ciphertext)

                 #decryption
                 pretty_print("\n\nDECRYPTION")

                 decryption_output = [0 for i in range(10)]

                   decryption_output =  decryption(m, ciphertext, g, code_locators,
                        permutation_matrix,
scrambler_matrix, generator_matrix)

                 recovered_plaintext = decryption_output[0]
                 inverse_permutation = decryption_output[1]
                 syndrome = decryption_output[2]
                 inv_syndrome = decryption_output[3]
                 v = decryption_output[4]
                 a0 = decryption_output[5]
                 b0 = decryption_output[6]
                 epsilon = decryption_output[7]
                 error_positions = decryption_output[8]
                 decodedword = = decryption_output[9]

                 pretty_print(f"\nMultiply the ciphertext by inverse of the Permutation matrix to 
                      obtain: ")
                 pretty_print(inverse_permutation)

                 pretty_print("\nNow we proceed with error correction using Patterson Algorithm")

                 pretty_print(f"\nSyndrome Computation: ")
                 pretty_print(syndrome)

                 pretty_print(f"\nInversion of syndrome modulo g: ")
                 pretty_print(inv_syndrome)

                 pretty_print("\nOutput of v = sqrt(S^{-1} + x) mod g: ")
                 pretty_print(v)

                 pretty_print("\nNow we solve the Key Equation using EEA algorithm, the output is: 
                      ")
                 pretty_print(f"A: {a0}")
                 pretty_print(f"B: {b0}")
                 pretty_print(f"\nTherefore, the error locator polynomial is: ")
                 pretty_print(epsilon)

                 pretty_print("\nThe indexes of the received word in which error occured: ")
                 pretty_print(error_positions)

                 pretty_print(f"\nThus after flipping the bits, we recover the codeword: ")
                 pretty_print(decodedword)

                 pretty_print(f"\nNow, we solve the equation S*G*m = d to recover the message m: ")
                 pretty_print(recovered_plaintext)

                 pretty_print("Verify if the recovered message is same as the original: ")
                 pretty_print(word == recovered_plaintext)##
```