SJÄLVSTÄNDIGA ARBETEN I MATEMATIK

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET

Post-Quantum cryptography and McElliece Cryptosystem

av

Ali Alwan

2024 - No K28

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET, 106 91 STOCKHOLM

Post-Quantum cryptography and McElliece Cryptosystem

Ali Alwan

Självständigt arbete i matematik 15 högskolepoäng, grundnivå

Handledare: Jonas Bergström

2024

Contents

Abstract

The emergence of quantum computers threatens the security of many current public key cryptosystems (PKCs) by enabling quantum algorithms that can break them. Post-quantum cryptography (PQC) seeks to address this challenge with both innovative and time-tested cryptographic systems. Among these is the McEliece Key Encapsulation Mechanism (KEM), which utilizes Goppa codes, a type of error-correcting code with a long history of security. This paper will explore the construction of Goppa codes, the implementation of the McEliece KEM, and an analysis of information set decoding—the most extensively studied attack method against these systems. Additionally, we will discuss the implications of quantum algorithms on the security of McEliece KEM.

Abstract

Framväxten av kvantdatorer hotar säkerheten för många nuvarande publika nyckelsystem (PKC) genom att möjliggöra kvantalgoritmer som kan bryta dem. Postkvantumkryptografi (PQC) syftar till att hantera denna utmaning med både innovativa och beprövade kryptografiska system. Bland dessa finns McEliece Key Encapsulation Mechanism (KEM), som använder Goppa-koder, en typ av felkorrigerande kod med en lång historia av säkerhet. Denna uppsats kommer att utforska konstruktionen av Goppa-koder, implementeringen av McEliece KEM och en analys av informationsmängdsavkodning - den mest omfattande studerade attackmetoden mot dessa system. Dessutom kommer vi att diskutera konsekvenserna av kvantalgoritmer för säkerheten i McEliece KEM.

1 Introduction

Given the advances in quantum computing and its implications on previuosly considered secure cryptosystems, the cryptographic community is currently working to change these out for other crpytosystems that are believed to be secure in the face of these computers, and their algorithms. Many such systems relying on problems such as integer factorization and the discrete logarithm problem are rendered broken by Shor's quantum algorithm. Currently the National Institute of Standards and Technology (NIST) is undergoing a competition to find replacements for the broken systems. One of these cryptosystems is called Classic McEliece, it is a code based cryptosystem using specifically Goppa codes to encrypt and decrypt it's messages. This thesis aims to give the reader a complete understanding of Goppa codes and the Classic McEliece.
The reader is assumed to have a introductory understanding of algebraic structures and linear algebra. Refer to any introductory text on the subjects if so is needed.

2 Cryptography

Cryptography is the study of hiding or encrypting information such that only the ones with a specific key can decrypt the information without too much effort. With this one can communicate over public channels without fear of eavesdroppers. A common type of cryptosystems is called Public Key Cryptography (PKC) where the encryption and decryption key are separate, with the encryption key being public and the decryption key being private. Doing it like this simplifies the setup of the cryptosystem such that one does not need to ensure private communication before sharing any private keys.
A lot of our current cryptographic methods rely on a small set of problems to which there is no traditional algorithm solving them in polynomial time, as far as we know.
One of these problems is the Discrete Logaritm Problem (DLP). And another on is the Integer Factorization Problem.
Though there are quantum algorithms, specifically Shor's algorithm solving these in polynomial time.

2.1 Post-Quantum Cryptography

As it stands many if not all public key cryptosystems currently used are secure, but with the future of quantum computing they are soon broken. This is due to Shor's algorithm which solves the problems they are based on in polynomial time.
McEliece relies on a linear code, most notable is a class of codes called Goppa code. These codes rely on the difficulty of a problem called the syndrome decoding problem.
We can use Grover's algorithm and the quantum walk algorithm to speed up decryptions without the private key but this is easily remedied with larger key sizes.

3 Coding Theory

Originally derived from ensuring messages can be reconstructed from noise corrupting it when sent over a communication channel, coding theory has found a surprising application in Cryptography. By adding noise to a message before sending it we can ensure that only the people with the appropriate decryption key are capable of reading the message.

In this section we will develop the mathematical core needed for the McEliece Cryptosystem.

This section starts with the basics of finite fields and linear codes before describing Goppa codes and how we encode and decode messages using Goppa codes. Throughout the section we create examples meant to illuminate the concepts and show how we work with them.

3.1 Finite Fields and Field Extensions

Though we assume the reader has a fundamental understanding of fields and algebraic structures, we will spend some time discussing the fundamentals of finite fields and how we represent them.
The theoerems and definitions in this section are from [1]

Definition 3.1.1. *A finite field is a field with a finite amount of elements.*

A finite integer ring $\mathbb{Z}/p\mathbb{Z}$ is only a field if it's order p is prime.

Example 3.1.1. *$\mathbb{Z}/p\mathbb{Z} = \mathbf{F}_p$ is a field iff p is a prime.*

It is important to note that if the order is p prime $\mathbb{Z}/p\mathbb{Z}$ is isomorphic to $\mathbf{F}_p$, if the order is p^m for a positive integer $m \geq 1$ then $\mathbf{F}_{p^m}$ is still a field which we will construct but $\mathbb{Z}/p^m\mathbb{Z}$ is not. Hence the isomorphism only holds if the order is prime.

Definition 3.1.2. *The characteristic of a field is the least positive integer p such that $p*1 = 0$ where 1 is the multiplicative identity of the field. If there is no such integer in the field we define it to be of characteristic 0.*

Definition 3.1.3. *An element $x \neq 0$ is a zero divisor if there exists $y \neq 0$ s.t. $xy = 0$.*

A field cannot contain a zero divisor, since every non-zero element in a field has a multiplicative inverse. Because $a * b = 0$ multiplied by a^{-1} gives $b = 0$.

Example 3.1.2. *We see that for $p, p^{m-1} \in \mathbb{Z}/p^m\mathbb{Z}$ we have that $p * p^{m-1} = 0$ making p and p^{m-1} zero divisors. Therefore $\mathbb{Z}/p\mathbb{Z}$ can't be a field.*

Theorem 3.1.1. *A field with positive characteristic must have a prime characteristic.*

Proof. Since F contains nonzero elements, F has characteristic $n \geq 2$. If n were not prime, we could write $n = km$. Then $0 = n * 1 = (km) * 1 = (k * 1) * (m * 1)$ implying that either $k * 1 = 0$ or $m * e = 0$ since F has no zero divisors it follows that either $kr = (k * 1)r = 0$ for all $r \in R$ or $mr = (m * 1)r = 0$ for all $r \in R$ in contradiction to the definition of the characteristic n. □

Following from this it is easy to see that any field with prime subfield F_p has characteristic p. since if they had characteristic p^m then $p^m * 1 = (p * 1)^m = 0$ which would make p a zero divisor, a contradiction.

Definition 3.1.4. *Let K be a subfield of the field F and M any subset of F. Then the field $K(M)$ is the intersection of all subfields of F containing both K and M and is called the extension field of K obtained by adjoining the elements in M. If M only contains one element θ then $K(\theta)$ is a simple extension of K with θ being the defining element.*

Definition 3.1.5. *Let K be a subfield of F and $\theta \in F$. If θ satisfies a nontrivial polynomial with coefficients in K, meaning $a_n\theta^n + \cdots + a_1\theta + a_0 = 0$ with $a_i \in K$ not all being 0, then θ is said to be algebraic over K. An extension L of K is called algebraic over K if every element of L is algebraic of K.*

Definition 3.1.6. *If $\theta \in F$ is algebraic over K, the the uniquely determined monic polynomial $g \in K[X]$ where g is of smallest possible degree then g is called the minimal polynomial of θ over K.*

Theorem 3.1.2. *If $\theta \in F$ is algebraic over K, then its minimal polynomial g over K is irreducible in $K[X]$.*

Let $\theta \in F$ be algebraic of degree n over K and let g be the minimal polynomial of θ over K then $K(\theta)$ is isomorphic to $K[X]/(g)$.
By showing different isomorphisms we are given more tools to understand and work with our finite fields.

Theorem 3.1.3. *Let $f \in K[X]$ be irreducible over K. then there exists a simple algebraic extension of K with a root θ of f as a defining element.*

Theorem 3.1.4. *If F is a finite field and $|F| = p^m$ where p is a prime, then it follows that F has charecteristic p. And F is a simple algebraic extension of F_p*

Following from this and theorem 3.1.1 it is easy to see that any field with prime subfield F_p has characteristic p. Since if they had characteristic p^m then $p^m * 1 = (p * 1)^m = 0$ which would make p a zero divisor, a contradiction.

Theorem 3.1.5. *Let $\theta \in F$ be algebraic over degree n over K and let g be the minimal polynomial of θ over K. Then $K(\theta)$ is isomorphic to $K[X]/(g)$*

Example 3.1.3. *We consider F_9 as a simple algebraic extension of F_3, which is obtained by adjunction of a root α of an irreducible quadratic polynomial over F_3, say $f(x) = x^2 + 1 \in F_3[x]$. Thus $f(\alpha) = \alpha^2 + 1 = 0$ in F_9 and the nine elements of F_9 are given in the form $a_0 + a_1\alpha$ with $a_0, a_1 \in F_3$. $F_9 = \{0, 1, \alpha, 1 + \alpha, 2 + \alpha, 2\alpha, 1 + 2\alpha, 2 + 2\alpha\}$*

This gives us a general idea on how to work with finite fields of order p^m. We can work in the polynomial field $F_3[x]\backslash(f(x))$ as it makes the operations simple with respect to the irreducible polynomial.

Example 3.1.4. *We now look at F_{2^2} as an extension of F_2. First off we look for a irreducible polynomial $f(X) \in F_2[X]$ say $x^2 + x + 1$ with root $\lambda \in F_{2^2}$. Then we have that $F_{2^2} = F_2[\lambda] = F_2[X]\backslash(x^2 + x + 1)$. With this we can easily define the elements of this field,*

$$\{0, 1, \lambda, \lambda^2 = \lambda + 1\}.$$

Which is the four elements we have. A simple show of computation

$$\lambda * (\lambda + 1) = \lambda^2 + \lambda \equiv -1 \equiv 1.$$

With the first congruence being due to our polynomial and the second due to modular arithmetic.

Another example with $m = 3$ instead.

Example 3.1.5. *In this case we have the irreducible polynomial $x^3 + x + 1$ with root λ. So we have the three fields*

$$F_8 = F_2[\lambda] = F_2[X]/(x^3 + x + 1)$$

With elements $\{0, 1, \lambda, \lambda^3 = \lambda + 1, \lambda^2, \lambda^6 = \lambda^2 + 1, \lambda^4 = \lambda^2 + \lambda, \lambda^5 = \lambda^2 + \lambda + 1\}$

We give both forms and note it is easier to use powers for multiplication and easier to use addition when in the form $a\lambda^2 + b\lambda + c$.

Before moving on to linear codes we show that every element in a finite field is a square of another element.

Definition 3.1.7. ***Frobenius Map:*** *over* $\mathbb{F}_{2^m}$

$$f : \mathbb{F}_{2^m} \to \mathbb{F}_{2^m} \quad f(x) = x^2$$

Theorem 3.1.6. *The Frobenius map over* $\mathcal{F}_{2^m}$ *is bijective.*

Proof. We start by showing that f is injective. f is injective if $f(\lambda^k) = \phi(\lambda^r)$ implies that $\lambda^k = \lambda^r$. $f(\lambda^k) = f(\lambda^r) = \lambda^{2k} = \lambda^{2r}$ which gives us the equality $\lambda^{2k} - \lambda^{2r} = (\lambda^k - \lambda^r)^2 = 0$ which is only true if $\lambda^k = \lambda^r$.
Now we show that f is surjective. If f is surjective then for all $y \in \mathbb{F}_{2^m}$ there is a x such that $f(x) = y$. We start with λ^k with $k = 2r$ in this case it is easy to see that $f(\lambda^r) = \lambda^k$. Now for $k = 2r + 1$, we start by noticing that $\lambda^{2^m - 1} = 1$ so $\lambda^{2^m} = \lambda$. Meaning we can represent every odd power as an even power, $\lambda^k = (\lambda^{2^m})^k = f((\lambda^{2^{m-1}})^k)$.

□

Here the surjective part tells us that every element of a finite field is a square of an element in the finite field.

3.2 Linear codes

The core theory of linear codes is mostly that of linear algebra with different names for the concepts. In this section we will briefly define what we need before moving on to Goppa codes.

Let $\mathcal{V}$ be a vector space over $\mathbb{F}$.

Definition 3.2.1. ***Subspace of a vector field:****[2]*
A subset $\mathcal{U}$ *of* $\mathcal{V}$ *is called a subspace of* $\mathcal{V}$ *if* $\mathcal{U}$ *is also a vector space with the same additive identity, addition, and scalar multiplication as on* $\mathcal{V}$.

Theorem 3.2.1. ***Conditions for a subspace:*** *[2]*
A subset $\mathcal{U}$ *of* $\mathcal{V}$ *is a subspace if and only if* $\mathcal{U}$ *satisfies the following three conditions.*

- ***Additive identity:*** $0 \in \mathcal{U}$.
- ***Closed under addition:*** $u, w \in \mathcal{U}$ *implies* $u + w = \mathcal{U}$.
- ***Closed under scalar multiplication:*** $a \in \mathbb{F}$ *and* $u \in \mathcal{U}$ *implies* $au \in \mathcal{U}$

Though the theorem above is provided in it's more general state its important to note that our scalars only constitute 0 and 1. Hence the last conditions is not really needed.

Definition 3.2.2. ***Linear Code****[3]*
A linear code C *of length* n *over* $\mathbb{F}_p$ *is a subspace of* $\mathbb{F}_p^n$ *[3]*

Example 3.2.1. *If we set $p = 2$ and $n = 8$ we have the simple subspace of $\mathbb{F}_p^n$*
$C = \{[00000000], [00111111], [11010101], [11101010]\}$ To see that this is a subspace we need to show that it is closed under vector addition and scalar multiplication and that it contains the additive identity. We see immediately that it contains the additive identity and since the characteristic is 2 the only scalar multiples are 0 and 1. Now we only need to show additive closure.

$$[00111111] + [11010101] = [11101010]$$

$$[11101010] + [00111111] = [11010101]$$

$$[11010101] + [00111111] = [11101010]$$

We see then that it is a linear code. This example as we will see later is a Goppa code.

Definition 3.2.3. ***Dual Code****[3]*
The dual code of C is C^1, the orthogonal complement of the subspace C of $\mathbb{F}_p^n$ Meaning

$$C^{\perp} = \{\mathbf{v} \in \mathbb{F}_p^n : \mathbf{c} \cdot \mathbf{v} = 0 \ \forall \ \mathbf{c} \in C\}.$$

Where $\cdot$ is the dot product of two vectors.

Definition 3.2.4. ***Dimension of linear code****[3]*
The dimension of a linear code is the dimension of C as a vector space over $\mathbb{F}_q$. That is the amount of vectors any basis of C consists of.

Definition 3.2.5. ***Hamming Distance****[3]*
Let $\mathbf{x}$ and $\mathbf{y}$ be vectors of length n. The Hamming distance from $\mathbf{x}$ to $\mathbf{y}$, denoted by $\mathbf{d}(\mathbf{x}, \mathbf{y})$, is defined to be the number of places at which $\mathbf{x}$ and $\mathbf{y}$ differ. If $\mathbf{x} = x_1 \cdots x_n$ and $\mathbf{y} = y_1 \cdots y_n$, then

$$\mathbf{d}(\mathbf{x}, \mathbf{y}) = d(x_1, y_1) + \cdots + d(x_n, y_n)$$

where x_i and y_i are regarded as words of length 1, and

$$d(x_i, y_i) = \begin{cases} 1 \text{ if } x_i \neq y_i \\ 0 \text{ if } x_i = y_i \end{cases}$$

b

Definition 3.2.6. ***Hamming Weight****[3]*
Let $\mathbf{x}$ be a word in $\mathbf{F}_p^n$. The Hamming weight of $\mathbf{x}$, denoted by $wt(\mathbf{x})$, is defined to be the number of nonzero coordinates in $\mathbf{x}$; i.e.,

$$wt(\mathbf{x}) = d(\mathbf{x}, \mathbf{0})$$

Definition 3.2.7. ***Generator Matrix****[3] A generator matrix for a linear code C is a matrix G whose rows form a basis for C.*

Definition 3.2.8. ***Parity-Check Matrix****[3] A parity-check matrix H for a linear code C is a generator matrix for the dual code $C^{\perp}$.*

Since the rows of the parity-check matrix H is the basis for the orthogonal complement to the linear code C we know that for any vector $c \in C$ we have that $Hc^T = 0$ and $cH^T = 0$. We can use this to find vectors belonging to C

3.3 Goppa Codes

Goppa codes are the mathematical core of this thesis, here we explain in detail it's construction and how to effectively decode a cipher using a Goppa code. The majority of this section (3.3) and the next (3.4) relies heavily on Overbeck's review article[4].

Definition 3.3.1. ***Goppa polynomial:*** *Let m and t be positive integers, and p a prime then*

$$g(X) = \sum_{i=0}^{t} g_i X^i \in \mathbb{F}_{p^m}[X] \tag{1}$$

is a monic polynomial of degree t.

We begin by constructing the finite field that we then construct a polynomial ring over.

Example 3.3.1. *We set $p = 2$ and $m = 3$ such that*

$$\mathbb{F}_{2^3} = \{0, 1, \lambda, \lambda^3 = \lambda + 1, \lambda^2, \lambda^6 = \lambda^2 + 1, \lambda^4 = \lambda^2 + \lambda, \lambda^5 = \lambda^2 + \lambda + 1\} = F_2[X]/(x^3 + x + 1)$$

as given in a previous section. These elements are then the coefficients used for the polynomials in $\mathbb{F}_{p^m}[X]$. With this we can now look for a Goppa polynomial preferably irreducible.
If we try with

$$X^2 + X + 1$$

we see that for all our values in $\mathbb{F}_{2^3}$ it never reduces to 0. Hence we have a irreducible Goppa polynomial of degree $t = 2$.

Definition 3.3.2. ***code support*** *we have a vector of n distinct elements*

$$L = (\gamma_0, \ldots, \gamma_{n-1}) \tag{2}$$

such that $\gamma \in \mathbb{F}_{p^m}$ and

$$g(\gamma_i) \neq 0 \ \forall \ 0 \leq i \leq n$$

Note that order of the elements is important for Patterson's algorithm and in case the generator matrix does not have a systematic form.
Since we have opted for an irreducible polynomial we could take L to be all of $\mathbb{F}_{2^3}$.

Example 3.3.2. $L = (0, 1, \lambda, \lambda^3 = \lambda + 1, \lambda^2, \lambda^6 = \lambda^2 + 1, \lambda^4 = \lambda^2 + \lambda, \lambda^5 = \lambda^2 + \lambda + 1)$

Definition 3.3.3. ***Syndrome of*** **c**

$$S_c(X) = -\sum_{i=0}^{n-1} \frac{c_i}{g(\gamma_i)} \frac{g(X) - g(\gamma_i)}{X - \gamma_i} \mod g(X) \tag{3}$$

Definition 3.3.4. ***Goppa code*** $\mathcal{G}(\mathbf{L}, g(X))$. *The Goppa code is the set of all $\mathbf{c} = (c_0, \ldots, c_{n-1}) \in \mathbb{F}_p^n$ such that*

$$S_c(X) \equiv \sum_{i=0}^{n-1} \frac{c_i}{X - \gamma_i} \equiv 0 \mod g(X) \tag{4}$$

Where $g(X)$ is the Goppa polynomial defined above.
Note that our codevectors are vectors over $\mathbb{F}_p$ and not $\mathbb{F}_{p^m}$

We can write this as

$$\mathcal{G}(\mathbf{L}, g(X)) = \{\mathbf{c} \in \mathbb{F}_p^n | S_c(X) \equiv 0 \mod g(X)\}$$

If our polynomial is irreducible over $\mathbb{F}_{p^m}$, then we call $\mathcal{G}(\mathbf{L}, g(X))$ an irreducible Goppa code. With all of this information we can now construct a simple Goppa code.

Example 3.3.3. *we see that $n = 8$ and we write the syndrome out in it's entirety.*

$$S_c(X) \equiv \frac{c_0}{X-0} + \frac{c_1}{X-1} + \frac{c_2}{X-\lambda} + \frac{c_3}{X-\lambda^2} + \frac{c_4}{X-\lambda^3} + \frac{c_5}{X-\lambda^4} + \frac{c_6}{X-\lambda^5} + \frac{c_7}{X-\lambda^6} = \sum_{i=0}^{7} \frac{c_i}{X-\gamma_i} =$$

$$c_0(X+1) + c_1(X) + c_2(\lambda^2 X + \lambda^5) + c_3(\lambda^4 X + \lambda^3) + c_4(\lambda^2 X + \lambda^3) + c_5(\lambda X + \lambda^6) + c_6(\lambda X + \lambda^5) + c_7(\lambda^4 X + \lambda^6)$$

calculations for finding the inverses are in appendix A.
Now this leaves us with only $\mathbf{c} = (0,0,1,1,1,1,1,1)$, $\mathbf{c} = (1,1,0,1,0,1,0,1)$ and $\mathbf{c} = (1,1,1,0,1,0,1,0)$ as the only vectors in $\mathcal{G}$ from a total of $2^8 = 256$ possibilities. In our example since n is very small we can easily compute but for larger vectors we would need to set up a system of equations and solve for all solutions. Before we move on we give a quick proof to show that Goppa codes indeed are linear codes.

Proof. Take $k \in \mathbb{F}_p$ and $c, c' \in \mathcal{G}(L, g(X))$ and $\gamma_i \in L$ for $i = 0, ..., n-1$
Closure under scalar multiplication
since c is a code vector its syndrome is congruent to 0 modulo $g(x)$

$$S_c(X) \equiv 0 \mod g(X)$$

it is easy to see that

$$kS_c(X) \equiv \sum_{i=0}^{n-1} \frac{kc_i}{X-\gamma_i} \equiv S_{kc}(X) \equiv 0 \mod g(X)$$

hence $\forall k \in \mathbb{F}_p$, kc is also a Goppa code vector.
Additive identity

$$S_0(X) \equiv \sum_{i=0}^{n-1} \frac{0}{X-\gamma_i} \equiv 0 \mod g(X)$$

Additive closure

$$S_{c+c'}(X) \equiv \sum_{i=0}^{n-1} \frac{c_i + c'_i}{X-\gamma_i} \equiv \sum_{i=0}^{n-1} \frac{c_i}{X-\gamma_i} + \sum_{i=0}^{n-1} \frac{c'_i}{X-\gamma_i} \equiv S_c(X) + S_{c'}(X) \equiv 0 \mod g(X)$$

□

3.3.1 Constructing a Parity-Check Matrix

This section we will show how we can construct a parity-check matrix for our goppa code. We start by looking at the equality

$$\frac{g(X) - g(\gamma)}{X - \gamma_i} = \sum_{j=0}^{t} g_j \frac{X^j - \gamma_i^j}{X - \gamma_i} = \sum_{s=0}^{t-1} X^s \sum_{j=s+1}^{t} g_j \gamma_i^{j-1-s} \tag{5}$$

While the first equality follows from the definition of the Goppa polynomial the second is a bit more involved hence we prove it here. Expanding the middle equation gives

$$\sum_{j=0}^{t} g_j \frac{X^j - \gamma_i^j}{X - \gamma_i} = 0 + g_1 + g_2(X + \gamma_i) + g_3 \frac{X^3 - \gamma_i^3}{X - \gamma_i} + \cdots + g_t \frac{X^t - \gamma_i^t}{X - \gamma_i}$$

With the difference of powers we see that

$$\sum_{j=0}^{t} g_j \frac{X^j - \gamma_i^j}{X - \gamma_i} = \sum_{j=0}^{t} g_j \sum_{r=0}^{j-1} X^{j-1-r} \gamma_i^r$$

Expanding this step by step we get that

$$\sum_{j=0}^{t} g_j \sum_{r=0}^{j-1} X^{j-r-1} \gamma_i^r = \sum_{j=0}^{t} g_j ((\sum_{r=0}^{j-2} X^{j-1-r} \gamma_i^r) + X^0 \gamma_i^{j-1}) =$$

$$X^0 \sum_{j=0}^{t} g_j \gamma_i^{j-1} + \sum_{j=0}^{t} g_j ((\sum_{r=0}^{j-3} X^{j-1-r} \gamma_i^r) + X^1 \gamma_i^{j-2}) =$$

$$\vdots$$

$$\sum_{s=0}^{k} X^s \sum_{j=0}^{t} g_j \gamma_i^{j-s-1} + \sum_{j=0}^{t} g_j ((\sum_{r=0}^{j-k-2} X^{j-1-i} \gamma_i^r) + X^{k+1} \gamma_i^{j-k-2}) =$$

$$\vdots$$

$$\sum_{s=0}^{t-1} X^s \sum_{j=0}^{t} g_j \gamma_i^{j-s-1}$$

Proving that the second equality of (5) is true.
Let c_i be the ith position of a codeword c in our Goppa code. Now using equation 5 and plugging it into the definition of the syndrome we see that for all $s = 0, \ldots, t-1$

$$\sum_{i=0}^{n-1} \left(\frac{1}{g(\gamma_i)} \sum_{j=s+1}^{t} g_j \gamma_i^{j-1-s} \right) c_i = 0$$

To show how this sum can give us a parity-check matrix we start by looking at $s = 0$.

$$\sum_{i=0}^{n-1}\left(\frac{1}{g(\gamma_i)}\sum_{j=1}^{t} g_j\gamma_i^{j-1}\right)c_i = 0$$

Here we can more clearly see how the outer sum gives the i:th position of the last row vector of our matrix H below. This sum is then the dot product of the row vector and our codevector, that it is resulting in 0 confirms that the row vector is an element of our dual code. With this we can use it as a row vector in our parity check matrix.
so we can write a parity-check matrix as

$$H = \begin{pmatrix} g_t g(\gamma_0)^{-1} & \cdots & g_t g(\gamma_{n-1})^{-1} \\ (g_{t-1} + g_t\gamma_0)g(\gamma_0)^{-1} & \cdots & (g_{t-1} + g_t\gamma_{n-1})g(\gamma_{n-1})^{-1} \\ \vdots & \ddots & \vdots \\ (\sum_{j=1}^{t} g_j\gamma_0^{j-1})g(\gamma_0)^{-1} & \cdots & (\sum_{j=1}^{t} g_j\gamma_{n-1}^{j-1})g(\gamma_{n-1})^{-1} \end{pmatrix} = XYZ$$

Such that

$$X = \begin{pmatrix} g_t & 0 & 0 & \cdots & 0 \\ g_{t-1} & g_t & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ g_1 & g_2 & g_3 & \cdots & g_t \end{pmatrix} \quad Y = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \gamma_0 & \gamma_1 & \cdots & \gamma_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_0^{t-1} & \gamma_1^{t-1} & \cdots & \gamma_{n-1}^{t-1} \end{pmatrix} \quad Z = \begin{pmatrix} \frac{1}{g(\gamma_0)} & & & \\ & \frac{1}{g(\gamma_1)} & & \\ & & \ddots & \\ & & & \frac{1}{g(\gamma_1)} \end{pmatrix}$$

Since H is a $t \times n$ matrix and $\mathbb{F}_{2^m} \cong \mathbb{F}_2^m$ we can write this matrix as a $mt \times n$. We will prove this at the end of this section.
Now we only need to show that each sum for different values of s (or each row) are linearly independent. We can show this for $s = t - 1$ and $s = t - 2$

$$\sum_{i=0}^{n-1}\left(\frac{1}{g(\gamma_i)}\sum_{j=t}^{t} g_j\gamma_i^{j-1-t+1}\right), \quad \sum_{i=0}^{n-1}\left(\frac{1}{g(\gamma_i)}\sum_{j=t-1}^{t} g_j\gamma_i^{j-1-t+2}\right)$$

These then result in

$$\sum_{i=0}^{n-1}\left(\frac{1}{g(\gamma_i)}g_t\right), \quad \sum_{i=0}^{n-1}\left(\frac{1}{g(\gamma_i)}(g_{t-1} + g_t\gamma_i)\right)$$

To show more clearly that the sums represent the row vectors above we multiply by a unit vector v_i where only the i:th position is 1 and the rest are 0.

$$\sum_{i=0}^{n-1}\left(\frac{1}{g(\gamma_i)}g_t\right)v_i, \quad \sum_{i=0}^{n-1}\left(\frac{1}{g(\gamma_i)}(g_{t-1} + g_t\gamma_i)\right)v_i$$

To show that these vectors are linearly independent we need to show that

$$a\sum_{i=0}^{n-1}\left(\frac{1}{g(\gamma_i)}g_t\right)v_i + b\sum_{i=0}^{n-1}\left(\frac{1}{g(\gamma_i)}(g_{t-1} + g_t\gamma_i)\right)v_i = 0$$

if and only if $a = b = 0$. To show this we can assume that it is not true for $i = i_1$, we then get that.

$$ag_t + b(g_{t-1} + g_t\gamma_{i_1}) = 0$$

where $a, b \neq 0$. This would then mean that

$$ag_t + b(g_{t-1}) = -bg_t\gamma_{i_1}$$

If we then take another index $i = i_2$ we already know that $\gamma_{i_1} \neq \gamma_{i_2}$ so this would result in

$$ag_t + b(g_{t-1}) = -bg_t\gamma_{i_1} \neq \gamma_{i_2}$$

which tells us that the rows have to be linearly independent. Similar proofs follow for other values of s, telling us that all the row vectors from the parity check matrix are linearly independent.
We now continue from our previous examples and show an example of the parity check matrix.

Example 3.3.4. *We start our example by listing $g(\gamma_i)^{-1}$ for $0 \leq i \leq 7$ in order.*

$$\{1, 1, \lambda^2, \lambda^4, \lambda^2, \lambda, \lambda, \lambda^4\}$$

And g_t are just the coefficients from our polynomial so in our case the parity check matrix is

$$H = \begin{bmatrix} 1 & 1 & \lambda^2 & \lambda^4 & \lambda^2 & \lambda & \lambda & \lambda^4 \\ 1 & 0 & \lambda^5 & \lambda^3 & \lambda^3 & \lambda^6 & \lambda^5 & \lambda^6 \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & \lambda & \lambda^2 & \lambda^3 & \lambda^4 & \lambda^5 & \lambda^6 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \lambda^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \lambda^4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \lambda^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \lambda & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \lambda & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \lambda^4 \end{bmatrix}$$

With the parity check matrix we can now find the generator matrix We start by computing the kernel of

$$H\mathbf{c} = \begin{bmatrix} 1 & 1 & \lambda^2 & \lambda^4 & \lambda^2 & \lambda & \lambda & \lambda^4 \\ 1 & 0 & \lambda^5 & \lambda^3 & \lambda^3 & \lambda^6 & \lambda^5 & \lambda^6 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

To create the generator matrix from the parity check matrix we have to first find the null space to H. We already know this to be our Goppa code and then have the matrix of row vectors

$$G = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Now if we add one of row two and one of row three to the last row we end up with the matrix

$$G = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

with this we see that the needed basis vectors for our Goppa code can be the two middle vectors, making our generator matrix.

$$G = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Before we move on we want to rewrite our matrix to be a 6×8 matrix over $\mathbb{F}_2$ instead. To do this we first need to prove that $\mathbb{F}_2^3 \cong \mathbb{F}_{2^3}$

Theorem 3.3.1. *As vector spaces $\mathbb{F}_2^m$ and $\mathbb{F}_{2^m}$ are isomorphic, $\mathbb{F}_2^m \cong \mathbb{F}_{2^m}$.*

Proof. We know that $\mathbb{F}_{2^m} \cong \mathbb{F}[\mathbb{X}]/f(x)$ where $\deg(f(x)) = m$. So every element in $\mathbb{F}_{2^m}$ can be written as $\phi = \sum_{i=0}^{m-1} a_i x^i$, where $a_i \in \mathbb{F}_2$. This has an obvius bijection to $\mathbb{F}_2^m$, we define our map $K : \mathbb{F}_{2^m} \to \mathbb{F}_2^m$ such that

$$K(\phi) = \begin{pmatrix} a_{m-1} \\ \vdots \\ a_0 \end{pmatrix}$$

And it follows that this is homomorphic as well. □

Applying this to our field we get the vectors

Example 3.3.5.

$$0 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \quad 1 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \quad \lambda = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad \lambda^2 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix},$$

$$\lambda^3 = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}, \quad \lambda^4 = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}, \quad \lambda^5 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \quad \lambda^6 = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix},$$

And now we can rewrite our parity check matrix using these vectors.

Example 3.3.6.

$$H = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

3.4 Binary Goppa Codes

By setting $q = 2$ we have a binary Goppa code, this will be the code we use in the McEliece Cryptopsystem. A binary Goppa code offers some benefits over normal Goppa codes, for example the minimum distance between codewords is increased, meaning there is a higher amount of "noise" we can add without losing information, what this means is that the error vector we can add to mask our codeword can have a higher weight. Also we have a more efficient decoding algorithm we can use to remove said error vector.

3.4.1 Minimum Distance of Irreducible binary Goppa codes

The minimum distance between codewords in our Goppa codes is the smallest hamming distance between any two codewords.

Definition 3.4.1. *[3] For a code C containing at least two words, the minimum distance of C, denoted by $d(C)$, is*

$$d(C) = min\{d(x, y) : x, y \in C, x \neq y\}$$

It is important that we have this distance as it allows us to add a certain amount of errors to our codeword without losing any information.
Let $\mathcal{G}(\mathbf{L}, g(X))$ be an irreducible binary Goppa code with $\mathbf{L} = (\gamma_0, \ldots, \gamma_{n-1})$ and $\mathbf{c} = (c_0, \ldots, c_{n-1}) \in \mathcal{G}(\mathbf{L}, g(X))$ be a codeword
We now define the error locator polynomial which we will also use later to remove any errors from our codeword.

Definition 3.4.2. ***Error Locator Polynomial***
First we define our index set $T_c = \{i : c_i = 1\}$

$$\sigma_c(X) = \prod_{j \in T_c} (X - \gamma_j) \in \mathbb{F}_{2^m}[X].$$

Using the product rule we differentiate and get

$$\sigma_c'(X) = \sum_{i \in T_c} \prod_{j \in T_c \setminus \{i\}} (X - \gamma_j)$$

Which we can write out in the form of our syndrome. Note that $i_k = j_k$ and set $wt(c) = p$ meaning $|T_c| = p$.

$$\frac{X - \gamma_{j_1}}{X - \gamma_{i_1}}\Big[(X - \gamma_{j_2})(X - \gamma_{j_3}) \ldots (X - \gamma_{j_p})\Big] +$$

$$\frac{X - \gamma_{j_2}}{X - \gamma_{i_2}}\Big[(X - \gamma_{j_1})(X - \gamma_{j_3}) \ldots (X - \gamma_{j_p})\Big] +$$

$$\vdots$$

$$\frac{X - \gamma_{j_1}}{X - \gamma_{i_1}}\Big[(X - \gamma_{j_2})(X - \gamma_{j_2}) \ldots (X - \gamma_{j_p})\Big] =$$

$$\Big[(X - \gamma_{j_1})(X - \gamma_{j_2}) \ldots (X - \gamma_{j_p})\Big]\Big[\frac{1}{(X - \gamma_{i_1})} + \cdots + \frac{1}{(X - \gamma_{i_p})}\Big] =$$

$$\left[(X-\gamma_{j_1})(X-\gamma_{j_2})\dots(X-\gamma_{j_p})\right]\left[\frac{c_1}{(X-\gamma_1)}+\dots+\frac{c_{n-1}}{(X-\gamma_{n-1})}\right]=$$

$$\prod_{j\in T_c}(X-\gamma_{j_1})\sum_{i=0}^{n-1}\frac{c_i}{X-\gamma_i}=$$

$$\sigma_c(X)S_c(X)\equiv\sigma_c'(X)\bmod g(X) \tag{6}$$

since $\gcd(g(X),\sigma_c(X))=1$ we have that σ_c is invertible modulo $g(X)$ so we can write.

$$\frac{\sigma_c'(X)}{\sigma_c(X)}\equiv S_c(X)\bmod g(X)$$

Therefore we have

$$\forall\mathbf{c}\in\mathbb{F}_2^n:\qquad \mathbf{c}\in\mathcal{G}(\mathbf{L},g(X))\iff\sigma_c'(X)\equiv 0\bmod g(x)$$

.

Recall from the end of section 3.1 where we showed that every element in our finite field is a square and also a square root. Now we consider the Frobenius map

$$\mathbb{F}_{2^m}[X]\to\mathbb{F}_{2^m}[X]$$

$$f(X)=\sum_{i=0}^{n}f_iX^i\to f^2(X)=\sum_{i=0}^{n}f_i^2X^{2i}$$

For polynomial rings the Frobenius map is only injective and not surjective. It's image is $\mathbb{F}_{2^m}[X^2]$
Let $i\sigma_i$ be the coeeficients of $\sigma_c'(X)$ when $\sigma_c'(X)$ is written in the form

$$\sigma_c'(X)=\sum_{i=0}^{n}i\sigma_iX^{i-1}$$

Since $\mathbb{F}_{2^m}$ has characteristic 2 every $i=2k$ is 0 so we are left with only even powers of X. Hence σ_c' is a perfect square
Now if $c\in\mathcal{G}$ then $\sigma_c'(X)\equiv 0 \mod g(X)$ so $\sigma_c'(X)=g(X)q(X)$ for some polynomial $q(x)$ and since it is a perfect square $\sigma_c'(X)=(g(X)q_1(X))^2$ for some polynomial $q_1(x)$ meaning $\sigma_c'(X)\equiv 0 \mod g^2(X)$
Thus for any $c\in\mathcal{G}$ we can define a relation between it's hamming weight $(wt(c))$ to the degree of the Goppa polynomial.

$$wt(c)=deg(\sigma_c(X))\geq 1+deg(\sigma_c'(X))\geq 2\cdot deg(g(x))+1$$

Before we move on we give a quick proof that shows how the weight of our codewords also give us a minimun distance.

Theorem 3.4.1. *The distance between any codevectors in $\mathcal{G}$ has a minimum distance of $2t+1$.*

Proof. For all codevector $c\in\mathcal{G}$ we can represent it as $c=c_1-c_2$ such that $c_1,c_2\in\mathcal{G}$ then we have that $wt(c)=wt(c_1-c_2)=d(c_1,c_2)=2t+1$. □

With this we have a minimum distance between codewords and can move on to understanding error correction for binary Goppa codes

3.4.2 Decoding: Patterson's Algorithm

As we will see in the next section the McElliece cryptosystem refers to a decoding algorithm, this algorithm is used to find the added error vector to the codeword. Normally this decoding algorithm is Patterson's algorithm. The process of error-correcting is the removal of any added error vector to our codeword, in cryptography this error vector is explicitly added to obfuscate our codeword. We start by motivating the algorithm then stating it in its entirety and end this section with a simple example.
As we see above we have now shown that our binary Goppa code $\mathcal{G}(\mathbf{L}, g(X))$ has minimum distance $2t+1$, where g is irreducible and has degree t.

Proposition 3.4.1. *If $m \in \mathcal{G}$ then we can correct up to t errors in a ciphertext $c = m + e$ where $m \in \mathcal{G}$ and $c, e \in \mathbb{F}_2^n$ and $wt(e) \leq t$*

Proof. We know that $wt(e) = t$ and $d(m, m_1) \leq 2t+1$ for $m, m_1 \in \mathcal{G}$ so then we get that $t+1 \leq d(m+e, m_1) \leq 3t + 1$ and $d(m, m + e) = t$ so we are still closer to our original codevector than any other vector in our Goppa code. □

Recall from the section on finite fields that a finite field with characteristic 2 has a unique square root for every element, we also showed that every finite field is isomorphic to a polynomial quotient field. Then every element of this quotient field has a unique square root.
With Patterson's algorithm we can now correct up to t errors of any codeword $m \in \mathcal{G}$. Take any error vector $e \in \mathbb{F}_2^n$ such that $w(e) = t$
we are given $c = m + e$ and want to compute m and e we know that $S_m \equiv 0 \mod g(X)$ so $S_c \equiv S_e \mod g(X)$. We use the error locator polynomial for e and split it into squares and non-squares.

$$\sigma_e(X) = \alpha^2(X) + X\beta^2(X)$$

Then its derivative would be

$$\sigma_e'(X) = 2\alpha(X)\alpha'(X) + 2X\beta(X)\beta'(X) + \beta^2(X) = \beta^2(X)$$

Plugging this into equation (6) from the previous section we get that

$$(\alpha^2(X) + X\beta^2(X))S_e(X) \equiv \beta^2(X) \mod g(X)$$

which we can rewrite into

$$\beta^2(1 + XS_e(X)) \equiv \alpha^2(X)S_e(X) \mod g(X)$$

Since e is not a codeword, we have that $S_e(X) \not\equiv 0 \mod g(X)$, this tells us that $gcd(S_e(X), g(X)) = 1$ meaning that we have an inverse $S_e^{-1}(X) = T(X)$, we multiply our equation with this inverse.

$$\beta^2(X)(T(X) + X) \equiv \alpha^2(X) \mod g(X)$$

Each element of $\mathbb{F}_{2^{mt}}$ has a unique square root and since $\mathbb{F}_{2^{mt}} \cong \mathbb{F}_{2^m}[X]/(f(x))$ where $\deg(f(x)) = t$. We have that there is a $\tau(X) \in \mathbb{F}_{2^m}[X]$. that is the unique square root of $T(X) + X$. The square root of our equation is then

$$\beta(X)\tau(X) \equiv \alpha(X) \mod g(X) \tag{7}$$

Now we want to solve for $\alpha(X)$ and $\beta(X)$. We know that $\deg(\sigma_e(X)) \leq t$, it follows that $\deg(\alpha(X)) \leq \lfloor t/2 \rfloor$ and $\deg(\beta(X)) \leq \lfloor (t-1)/2 \rfloor$.

We can find unique polynomials satisfying equation (7) using the Euclidean algorithm. We give initial values $\alpha_0(X) = g(X), \alpha_1 = \tau(X)$, $\beta_0(X) = 0, \beta_1(X) = 1$ and the degree of our Goppa polynomial. The uniqueness is dependent on the relation of the degrees of our polynomials.

$$deg(\beta_k(X)) = deg(g(X)) - deg(\alpha_{k-1}(X))$$

Meaning the degree of β_k increases as α_k decreases.
We iterate through the algorithm until we have that

$$deg(\alpha_k(X)) \leq \lfloor (t+1)/2 \rfloor - 1 = \lfloor t/2 \rfloor$$

and our beta polynomial would be of degree.

$$deg(\beta_k(X)) = deg(g(X)) - deg(\alpha_{k-1}(X) = t - \lfloor (t-1)/2 \rfloor$$

Given our initial values we have unique polynomials with these degrees. We set

$$\alpha(X) = \alpha_k(X), \quad \beta(X) = \beta_k(X)$$

And with this we now have our error locator polynomial.
Lastly we look at each value in our code support and note that this is one of the main reasons why the order in our codesupport is important. The index of element in our codesupport that correspond to a zero in our error locator polynomial $\sigma_e(X)$ is the index where we have a one in our error vector e.

```
##not really eea
def extended_euclidean_algorithm(g, tau, t):
    i = 0
    r = [g, tau]
    alpha = [g, tau]
    beta = [galois.Poly([0], GF), galois.Poly([1], GF)]

    while r[i].degree >= (t + 1) // 2:
        i += 1
        q, r_i = divmod(r[i-1], r[i-2])
        r.append(r_i)
        beta.append(beta[i-2] + q * beta[i-1])
        alpha.append(r[i])

    return alpha[i], beta[i]

def pattersons(notcodeword):
    syndrome = np.matmul(coefffield.transpose(), notcodeword)
    if (syndrome == zeroMat).all():
        return (notcodeword, 0)
    else:
        syndrome = galois.Poly(syndrome)
        T = galois.egcd(syndrome, goppaPol)[1]
        Tpx = T +galois.Poly([1,0], GF)
        tau = 0

        for i in range(fOrd**gPolDeg):
            tau = pow(Tpx, i, goppaPol)
            if pow(tau, 2, goppaPol) == Tpx:
```

```
            print(tau)
            break

    alpha, beta = extended_euclidean_algorithm(goppaPol, tau, gPolDeg)
    alpha_squared = pow(alpha,2)
    beta_squared = pow(beta,2)
    x = galois.Poly([1, 0], GF)
    c = GF(2)

    sigma =  (alpha_squared + x * beta_squared)


    leading_coeff = sigma.coeffs[0]

    e = np.zeros(len(errVec[0]), int)
    for i in range(len(errVec[0])):
        if sigma.__call__(GF(i)) == 0:
            e[i] = 1

    e = GF(e)
    m = notcodeword + e
    return  m, e
```

All code used is show in it's entirety in the appendix. Let us now show an example of this algorithm.

Example 3.4.1. *We are given the ciphertext* $c = (1,1,1,0,1,1,1,1)$ *and want to find the codeword* $m = (1,1,1,0,1,0,1,0)$ *and the error vector* $e = (0,0,0,0,0,1,0,1)$ *and our Goppa polynomial* $g(x) = x^2 + x + 1$. *First we check the syndrome*

$$S_c = x + 1 + x + \lambda^2 x + \lambda^5 + 0 + \lambda^2 x + \lambda^3 + \lambda x + \lambda^6 + \lambda x + \lambda^5 + \lambda^4 x + \lambda^6 = \lambda^4 x + \lambda \neq 0$$

.
Since the syndrome is not zero we know the cipher is not a codeword. We now take the inverse of our syndrome $S_c^{-1} = T = \lambda^2 x + \lambda^4$. *(A method to find the inverse is given in the appendix, another method is the extended euclidean algorithm).*
We follow the algorithm $Tx = \lambda^6 x + \lambda^4$ *and check every power* i *of* Tx *where* $0 < i < 8^2$ *until we find an* i *such that* $(Tx)^i = \sqrt{Tx} = \tau = \lambda^3 x + \lambda^5$ *meaning we look for the square root of* Tx *from the other way. This works since the multiplicative group of a finite field is cyclical, we can see this in the finite fields defined above. With this we can now look for polynomials* $\alpha(x) = \lambda^3 x + \lambda^5$ *and* $\beta(x) = 1$ *we do this using the extended euclidean algorithm but stopping when* $deg(\alpha(x)) < \lfloor t/2 \rfloor$.
It's important to note that the values given for $\alpha(x)$ *and* $\beta(x)$ *are dependent on the degree of the Goppa polynomial, and in this case* $\deg(g) = 2$ *hence our polynomials* $\alpha(x)$ *and* $\beta(x)$. *We can now compute our error locator polynomial. The square of our polynomials are then* $\alpha^2 = (\lambda^6 x + \lambda^4)$ *and* $\beta^2 = 1$. *And we end up with*

$$\sigma_c(X) = \lambda^6 x^2 + x + \lambda^3$$

At this point all we have to do is plug in the values of our finite field and note the positions of those equating the error locator polynomial to 0. Those positions are then the 'ones' of our error vector.

$$\sigma_c(0) = \lambda^3 \quad \sigma_c(1) = \lambda^5 \quad \sigma_c(\lambda) = \lambda^3 \quad \sigma_c(\lambda^3) = \lambda^5 \qquad \sigma_c(\lambda^2) = \lambda^2 \quad \sigma_c(\lambda^6) = 0 \quad \sigma_c(\lambda^4) = \lambda^4 \quad \sigma_c(\lambda^5) = 0$$

Giving us the error vector $e = (0,0,0,0,0,1,0,1)$ *the same as the one above. Adding this to the cipher gives the message/code vector* $c + e = m = (1,1,1,0,1,0,1,0)$.

4 McEliece

The McEliece cryptosystem is a code based cryptosystem from 1978[5].
There have been many variations to this system replacing the Goppa codes with some other code in order to decrease the key size but most of these have been shown to be insecure.
Code base cryptography is in general a practice in the trade off between security and efficiency.
The McEliece cryptosystem recommends a public key size of around 1Mb for optimal security. More specifically we know from a paper called McBits[6] that with a keysize of 1046739 bits we have a security level of 2^{263} compared to AES-256 which has it's key size at 256 bits with a security level of 2^{256}. However such large keys can pose challenges in practical applications. To address this, techniques exist to distribute the cost of each public key across multiple ciphertexts, effectively sharing the burden and improving efficiency. Despite the large public keys, the resulting ciphertexts remain relatively small, ensuring efficient communication without compromising security.
To further optimize key size and performance, we leverage public keys in systematic form. This means the parity-check matrix H is structured as $H = (I|T)$, where I represents the identity matrix and T is a smaller matrix. By using this form, only the matrix T needs to be transmitted, significantly reducing the amount of data involved.
Research indicates that approximately 29% of Goppa codes, which are fundamental to the McEliece system, are naturally in systematic form for relevant parameters. This makes it a practical and effective approach.
If using systematic form with the original McEliece, any resulting ciphertext would leak parts of the plaintext, this is mitigated in the classic McEliece KEM.
Where the Key encapsulation mechanism (KEM) is a sort of wrapper over our PKE bringing it to a higher standard of security. What a KEM is and what this higher standard means will be explained in a later section.
The main benefit of the McEliece crypstosystem is the efficiency of encrypting and decrypting messages.

4.1 Original McEliece PKE

While the original McEliece cryptosystem was set aside in favour of other algorithms with smaller keysizes, the modern implementation the so called classic McEliece Key encapsulation mechanism is currently regaining favour as an alternative to our current methods that do not hold up against Shor's algorithm.
In this section, we will define the original McEliece cryptosystem, a public-key encryption scheme proposed by Robert McEliece in 1978.
The McEliece cryptosystem is based on the hardness of the decoding problem for general linear codes, which is believed to be computationally intractable for certain parameters, meaning there is no polynomial time deterministic algorithm solving it. This property makes the McEliece cryptosystem an viable alternative to traditional public-key encryption schemes, in the context of post-quantum cryptography.
We will begin by writing out the algorithm, then we will show why the decryption is possible and finish with an example written in python.
Before we move on we need to define the notation for our submatrix.

Definition 4.1.1. *[4] We define $\mathbf{G}_{.I}$ as the submatrix of $\mathbf{G}$ where the k columns with it's index in I constitute the columns of $\mathbf{G}_{.I}$. And I is called an information set or index set.*

Algorithm 4.1.1 Original McElliece [4]

- **System Parameters:** $n, t \in \mathbf{N}$, where $t << n$
- **Key Generation:** Given the parameters n,t generate the following matrices:
  $\mathbf{G}$: $k \times n$ generator matrix of a code $\mathcal{G}$ over $\mathbb{F}$ of dimension k and minimum distance $d \geq 2t + 1$.(A binary irreducible Goppa code in the original proposal.)
  $\mathbf{S}$: $k \times k$ random binary non-singular scrambler matrix.
  $\mathbf{P}$: $n \times n$ random permutation matrix.
  Then, compute the $k \times n$ matrix $\mathbf{G}^{pub} = \mathbf{SGP}$.
- **Public Key:** ($\mathbf{G}^{pub}$,t)
- **Private Key:** (S, $D_{\mathcal{G}}$, G,P), where $D_{\mathcal{G}}$ is an efficient decoding algorithm for $\mathcal{G}$.
- **Encryption ($E_{(\mathbf{G}^{pub},t)}$):** To encrypt a plaintext $m \in \mathbf{F}^k$ choose a vector $\mathbf{z} \in \mathbf{F}^k$ of weight t randomly and compute the ciphertext $\mathbf{c}$ as follows :

$$\mathbf{c} = \mathbf{mG}^{pub} + \mathbf{z}$$

- **Decryption ($D_{(S,D_{\mathcal{G}},P)}$):** to decrypt a ciphertext $\mathbf{c}$ calculate

$$\mathbf{cP^{-1}} = (\mathbf{mS})\mathbf{G} + \mathbf{zP^{-1}}$$

first and apply the decoding algorithm (In our case this is Patterson's algorithm.) $D_{G^{pub}}$ for $\mathcal{G}$ to it. Since $\mathbf{cP^{-1}}$ has a hamming distance of t to $\mathcal{G}$ we obtain the codeword

$$\mathbf{mSG} = D_{\mathcal{G}}(\mathbf{cP^{-1}})$$

At this point we take $I \subset \{1, \ldots, n\}$ such that $|I| = k$ and that $\mathbf{G_{.I}}$ is invertible. Then we can compute the plaintext

$$\mathbf{m} = (\mathbf{mSG})_{.I}\mathbf{G}_{.I}^{-1}\mathbf{S}^{-1}$$

Theorem 4.1.1. ***Existence of invertible submatrix of*** G
Let G *be a* $k \times n$ *matrix with rank* k*. Then* G *has a* $k \times k$ *invertible submatrix.*

Proof. Starting with the matrix G and putting it in row reduced echelon form we get a matrix with all rows containing leading ones with only zeroes below.

$$G = \begin{bmatrix} g_{1,1} & \cdots & g_{1,n} \\ \vdots & & \vdots \\ g_{k,1} & \cdots & g_{k,n} \end{bmatrix} \to G_{rref}$$

If we then take the column index where the leading ones are and make a submatrix $G_{.I}$ out of those columns, we would have a matrix with a diagonal of ones in its row reduced echelon form. If we apply Gaussian on $G_{.I}$ by putting it in the form $[G_{.I}|I_{k\times k}]$ and tracking the changes on the identity matrix we get the result $[I_{k\times k}|G_{.I}^{-1}]$

□

Using an invertible submatrix of G to retrieve the plaintext.
One possible issue lies in the decryption of a cipher, here we see that the decryption does not use the whole generator matrix to decrypt but rather a submatrix and yet we expect to receive the whole plaintext.
The reason we use a submatrix is that we can't always expect to have a square generator matrix, therefore we have to take a submatrix that is square and invertible.
Now the question we want answered is why is the following equation correct.

$$\mathbf{m} = (\mathbf{mSG})_{.I}\mathbf{G}_{.I}^{-1}\mathbf{S}^{-1}$$

To see why this works we need to show in general how the computation acts.

$$\mathbf{mSG} = \begin{bmatrix} a_1 & \cdots & a_k \end{bmatrix} \begin{bmatrix} g_{1,1} & \cdots & g_{1,n} \\ \vdots & \ddots & \vdots \\ g_{k,1} & \cdots & g_{k,n} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{k} a_i g_{i,1} & \sum_{i=1}^{k} a_i g_{i,2} & \cdots & \sum_{i=1}^{k} a_i g_{i,n} \end{bmatrix}$$

Lemma 4.1.1. *If* v *is a vector of length* k *and* W *is a* $k \times n$ *matrix with* I *being a subset as defined above, then*

$$(vW)_{.I} = vW_{.I}$$

Proof. If we set vW as $\mathbf{mSG}$ and take $vW_{.I}$ we end up with the array

$$vW_{.I} = \begin{bmatrix} a_1 & \cdots & a_k \end{bmatrix} \begin{bmatrix} w_{1,I_1} & \cdots & w_{1,I_k} \\ \vdots & \ddots & \vdots \\ w_{k,I_1} & \cdots & w_{k,I_k} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{k} a_i w_{i,I_1} & \sum_{i=1}^{k} a_i w_{i,I_2} & \cdots & \sum_{i=1}^{k} a_i w_{i,I_k} \end{bmatrix} = (vW)_{.I}$$

□

Meaning there is no loss of information in taking a submatrix, we can still get out all of our plaintext without issue.

Example 4.1.1. *If we continue on the previous examples we would only need to define a scrambler matrix* $\mathbf{S}$ *and a permutation matrix* $\mathbf{P}$*. We set*

$$\mathbf{S} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad \mathbf{P} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

we end up with the public key being

$$G^{pub} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

and the degree t *of our Goppa code. With the private key being our two random matrices and the efficient decoding algorithm for our code.*

Both encryption and decryption are basic computation of binary vectors with the addition of Pattersons algorithm being used as the efficient decryption algorithm. We will later show how to decode a specific cipher text without the private key.

```
def encrypt(g_pub, binary_plaintext):

    c_arr = []
    binary_plaintext = GF(binary_plaintext)
    g_pub = GF(g_pub)
    ## we multiply k bits to our public key at a time and add the encrypted bits to an array
                                                                .
    for i in range(0, len(binary_plaintext),2):
        c_arr.append((binary_plaintext[i:i+k]@ g_pub))

    randint = rand.randint(1,len(err_vec)-1) ## cant use a new one for every k bits
    ##now we add an error vector to the vectors ## maybe take random err vector every time.

    c_arr_err= [(i+err_vec[4]) for i in c_arr]
    return c_arr_err
```

```
def decrypt(G,S,P, c_arr_err):
    ## we multiply with the inverse of the permutation matrix
    c_arr_err = c_arr_err@np.linalg.inv(P)

    c_arr_err = GF(c_arr_err.astype(int))

    ##remove error vector
    decoded_c_arr =[]
    for i in range(len(c_arr_err)):
        plup,_ = pattersons(c_arr_err[i])
        decoded_c_arr.append(plup)

    ## we have to find an index set such that G_.I is invertible
    I  = information_set(G)
```

```
    ## now we revert our ciphertext
    GInv = GF(np.linalg.inv(G[:,I]).astype(np.int32))
    SInv = GF(np.linalg.inv(S).astype(np.int32))

    c_arr_rev =[i[I]@GInv@SInv for i in decoded_c_arr]
    return c_arr_rev
```

Let's now look at an example. We want to encrypt the string

```
    text = "Hello! My name is."
    ## 01001000 01100101 01101100 01101100 01101111 00100001 00100000 01001101 01111001
    ## 00100000 01101110 01100001 01101101 01100101 00100000 01101001 01110011 00101110

    binary_array = str_to_binary(text)
```

First we have to translate the string into binary using the auxiliary function str_to_bin (available in the appendix). Then we pass it into the encryption function with our previously defined public key G^{pub}

```
c_arr_err = encrypt(g_pub, binary_array)
```

A small part of the encrypted binary vectors end up being.

```
##[0 0 0 0 0 0 0 0], [0 0 1 1 1 1 1 1], [1 1 0 1 0 1 0 1], [1 1 1 0 1 0 1 0],
##[1 1 1 0 1 1 1 1], [0 0 0 0 0 1 0 1], [0 0 0 1 1 0 1 0], [0 0 0 0 0 1 0 1]
```

If we then feed into the decrypt function and retranslate the binary into a string

```
    c_arr_rev = decrypt(G,S,P,c_arr_err)
    bin_2_str(c_arr_rev)
```

We see that we end up with the original plaintext "Hello! My name is.".

4.2 From OW-CPA to IND-CCA2

This section relies on [7] and [8] for defining the following concepts.
The original McEliece PKC was designed to be OW-CPA, this mean it is difficult to invert the map from ciphertext back to our plaintext given the public key but this is only when the plaintext is chosen uniformly at random.

This will not work for encrypting an actual message which usually contains structure, this can supply the attacker with some pattern that simplifies the inversion to the plaintext.

To solve this we want our cryptosystem to be IND-CCA2 secure, this type of security comes in two different parts first is Adaptive chosen-ciphertext attack (CCA2).

In the CCA2 attack model the attacker is given a decryption oracle that can decrypt all ciphertext except one specific challenge ciphertext c. A CCA2 secure cryptosystem is then a cryptosystem where such an oracle won't help with decrypting the challenge c.

And in the other part we have Indistinguishable (IND) where in addition to the CCA2 attack model the attacker is given no advantage in determining between two plaintext which encrpyts into the given challenge ciphertext.
To create a IND-CCA2 cryptosystem we can wrap our original McEliece PKE into a Key Encapsulation Mechanism (KEM).
A KEM consists of three parts

1. Key generation: First we generate a public and a private key.
2. Encapsulation: Encrypt plaintext e into ciphertext c and compute session key K, output c and K.
3. Decapsulation: Decode the ciphertext into the plaintext and use it to compute the session key K. Output K.

The KEM construction from a PKE is as follows.

- Start from any correct deterministic PKE.
- The KEM public key is the PKE public key.
- The KEM private key includes the PKE private and public key, and an implicit-rejection key s generated uniformly at random.
- The session key for plaintext m and PKE ciphertext c is $H(1, m, c)$.
- The KEM ciphertext is the same as the PKE ciphertext.
- Decapsulation of c checks if the decryption m reencrypts to c if yes then it outputs $H(1, m, c)$ if not the it outputs $H(0, s, c)$.

Classic McEliece only difference from the above construction is that it does not include a copy of the public key in the private key.

Let's look at some attacks on the original McEliece to show how it is broken in the IND-CCA2 attack model. We can later try the same attack model on the classic McElliece to confirm that that it is indeed IND-CCA2.

1. Attacker chooses two different message a and a' and a challenge ciphertext c such that $c = Ga + e$ or $c = Ga' + e'$, it then checks if $wt(c - Ga) = t$.
2. Attacker chooses $\delta \neq 0$ adds to $Ga + e$ resulting in $Ga + G\delta + e$, is then given $a + \delta$ subtracts δ to receive a.
3. Add two errors to $Ga + e$ there is a chance that one error is in e and the other is not. Turning a 1 into 0 and a 0 into 1. This produces a new ciphertext $c = Ga + e' = Ga + e + e_0$ (e_0 being the added errors.), Giving the attacker a.

We show an example of the first case.

```
plain_1 = goppaCodes[3]
plain_2 = goppaCodes[2]


ciph = GF(encrypt(G, plain_1, err_vec[3]))


#print("test parameters")
print("plain_1", plain_1)
print("plain_2", plain_2)
print("ciph", ciph)
#print("error", err_vec[3])
#print("G",G)
zero_err_vec = GF([0,0,0,0,0,0,0,0])

## testing against the indcca2 attackmodel
def ind_cca2(G,c, m1, m2):
    #print("encrypt m1 with 0 errors",GF(encrypt(G, m2, zero_err_vec)))
    m1 = encrypt(G, m1, zero_err_vec)
    m1 = GF(m1)
    m2 = encrypt(G, m2, zero_err_vec)
    m2 = GF(m2)
    su = [0,0,0,0]
    sa = [0,0,0,0]

    for i in range(len(c- m1)): ##np.sum not working as expected
        for j in (c- m1)[i]:
            if j == 1:
                su[i] = su[i] +  1

    for i in range(len(c- m2)): ##np.sum not working as expected
        for j in (c- m2)[i]:
            if j == 1:
                sa[i] = sa[i] +  1

    def check_something(sasa):
        for i in sasa:
            if i != gPolDeg:
                return False
        return True
    print(check_something(su), check_something(sa))
    print(su,sa)
    if check_something(su) == True:
        return plain_1

    if check_something(sa) == True:
        return plain_2


print(ind_cca2(G, ciph, plain_1, plain_2))
```

4.3 Classic McEliece

Classic Mceliece is built from Niederreiter's dual version (Defined in appendix.) of the McEliece PKE using binary goppa codes. It is a conversion from OW-CPA to IND-CCA2.

Over time, as new research and literature on the topic emerged, various defenses have been incorporated into the McEliece cryptosystem to enhance its security.
Before we define the classic McEliece KEM we will define some things, this gives us a clearer understanding of any potential matrix we are working with and will be needed later in the thesis as well.

Definition 4.3.1. *[8]* ***Reduced row echelon form*** *X is a matrix with n columns,where Gaussian elimination computes the unique matrix R in row reduced echelon form (rref). Then there is a subsequence $\{c_1, c_2, \ldots, c_{r-1}\}$ of $\{1, 2, \ldots, n\}$ such that every row i starts with a 1 in column c_i, and is the only nonzero term in the column. Meaning $r_{i,c_i} = 1$ and $r_{j,c_i} = 0$ for all $j \neq i$. The remaining rows of R are 0.*

Definition 4.3.2. ***Rank of a matrix*** *[2]*
The rank of a matrix R is the dimension of the span of the columns or rows of R.

Definition 4.3.3. ***Systematic form*** *[8]*
A rref matrix R is in systematic form if the following conditions are met.

1. *R has exactly r rows*
2. *$r_{i,i} = 1$ for all $0 \leq i \leq r$*

meaning the matrix has form $R = (I_r|T)$

In case our parity check matrix is not in systematic form we can generalize the concept

Definition 4.3.4. ***Semi-systematic form*** *[8]*
Given $\mu \geq 0$ and $\nu \geq 0$, let R be an rank-r matrix in reduced row echelon form. Assume that $r \geq \mu$, and that there are at least $r - \mu + \nu$ columns.
We say that R is in (μ, ν)-semi-systematic form if R has r rows (i.e., no zero rows); $c_i = i$ for $0 \leq i < r - \mu$; and $c_i \leq i - \mu + \nu$ for $r - \mu \leq i < r$.

As a special case, (μ, ν)-semi-systematic form is equivalent to systematic form if $\mu = \nu$. However, if $\nu > \mu$ then (μ, ν)-semi-systematic form allows more matrices than systematic form.

Example 4.3.1. ***General example of a matrix in (μ, ν)-semi-systematic form*** *Since c_i is the columns where row i has its first 1 we see that the statement $c_i = i$ means that row i has its first 1 in column i, this would yield an identity matrix of size $r - \mu$. With this we can show the matrix in the form*

$$\begin{bmatrix} I_{r-\mu} & Q_1 \\ 0 & Q_2 \end{bmatrix}$$

The first matrix 0 is a $\mu \times r - \mu$ 0 matrix (We know this since R is in Rref form.), while $Q = \begin{pmatrix} Q_1 \\ Q_2 \end{pmatrix}$ is a $r \times \nu^+$ matrix, where $\nu^+ > \nu$. The statement $c_i \leq i - \mu + \nu$ simply states that the remaining leading ones in Q are in the bound of the least amount of columns in R.

In case our we cannot generate a systematic but only a semi systematic parity check matrix we permute the columns and the code support such that it results in a matrix in systematic form. This permutation would then be a part of the private key.
We can now go on to define the classic McEliece KEM. This algorithm is written to be concise and cohesive with the rest of the thesis. The information itself is taken from the most current specification sheet[8] at the time of writing.

Algorithm 4.3.1 Classic McEliece KEM

- **System Parameters:** seed value $\delta \in \mathbb{F}_2^l$
- **Key Generation:** A vector $E \in \mathcal{F}_2^{n+\sigma_2 q+\sigma_1 t+l}$ is generated from δ.
  From E We define a random string s as the first n posittions.
  The code support L is from the next σq positions.
  And the Goppa polynomial is from the following σt positions.
  The last l bits is used as a new random seed in case any operation fails.
  Public Key: T where $H = (I|T)$ is our parity-check matrix.
  Private Key: (δ, g, L, s)
- **Encryption**
  Compute ciphertext $c = He$ where e is the error vector representing our plaintext.
  Compute session key $K = \mathbf{H}(1, e, c)$ where $\mathbf{H}$ is a cryptographic hash function called SHAKE256.
  Output ciphertext c and session key K.
- **Decryption** Decode c using Patterson's algorithm. if the decryption returns valid plaintext e, compute $K = \mathbf{H}(1, e, c)$ otherwise compute $K = \mathbf{H}(0, s, c)$. Output session key K.

We are shown in both [7] and [9] that the classic McElice KEM is IND-CCA2 secure.

5 Security and attacks on the McEliece PKC

The McEliece has maintained it's security since it was first proposed in 1978. Other than an attack coupled with its defence in 2008 [10] there has been no major vulnerabilities.
No other proposal for post-quantum cryptography has such a long history of security.

Though there are several different ways one can attack the McEliece KEM, such as structural attacks or statistical decoding, currently the most efficient set of attacks are called information set decoding (ISD). This is also where quantum algorithms have been applied to improve on their efficiency.
With this in mind we have decided to limit this paper to ISD algorithms and how we can apply quantum algorithms to them. We assume attacker has no information on the algebraic structure of the underlying error correction code. Meaning the attacker would have to correct errors using only the public key. Any reader who wishes to read more on the current research can go to " https://classic.mceliece.org/papers.html".

5.1 Information set decoding

Information set decoding is by far the most researched method of attack, it effects not only the McEliece cryptosystem or Goppa codes but all of code based cryptography.
In this section we will describe the fundamentals of information set decoding as well as go through an example.
ISD algorithms aim is to solve the syndrome decoding problem.

Definition 5.1.1. ***Syndrome Decoding Problem (SDP)****[11]*
Let $n, k, w \in \mathbb{N}$ such that $k \leq n$ and $w \leq n$. Given a parity check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ and $s \in \mathbb{F}_2^{n-k}$, find a vector $e \in \mathbb{F}_2^n$ of $wt(e) = w$ such that $He = s$.

There are many variations on this problem but for our purposes we simply have our target size w be equal the degree of our Goppa polynomial t.
The idea behind information set decoding relies on a simple computation. Given a cipher text $c = mG + e$ and define c_I, G_I and e_I as before we have that $c_I = mG_I + e_I$. We want G_I to be invertible and e_I to be the zero vector, this would turn our equality into $c_I = mG_I$. In case the error vector is not zero we can proceed by guessing the remaining "ones" in it. The final check to validate is to confirm the weight

$$wt((c_I + e_I)G_I^{-1}G + c) = t$$

This works because

$$(c_I + e_I)G_I^{-1}G + c = mG + c = mG + mG + e = e$$

Algorithm 5.1.1 Generalized information set decoding [4]

Input: A $k \times n$ generator matrix G, a ciphertext $c = mG + e$, where m is the plaintext and e is the error vector of weight t, a positive integer $j \leq t$.
Output: The plaintext m

while true **do**
  Choose randomly $\mathcal{I} \subset \{0, \ldots, n-1\}$, with $|\mathcal{I}| = k$
  $Q_1 = G_{\mathcal{I}}^{-1}$; $Q_2 = Q_1 G$
  $z = c + c_{\mathcal{I}} Q_2$
  for $i = 0$ to j **do**
    for all $e_{\mathcal{I}}$ with $\mathrm{wt}(e_{\mathcal{I}}) = i$ **do**
      if $\mathrm{wt}(z + e_{\mathcal{I}} Q_2) = t$ **then**
        return $(c_{\mathcal{I}} + e_{\mathcal{I}})Q_1$
      end if
    end for
  end for
end while=0

Example 5.1.1. *We set*

$$G = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

$$c = (1,1,1,0,1,1,1,1) = mG + e = (1,1)G + (0,0,0,0,0,0,1,0,1)$$

with error correction capability $t = 2$ and a positive integer $j = 2$. We set I such that G_I is invertible and $|I| = k$

$$I = \{2,3\} \text{ and } G_I = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

$$Q_1 = G_I^{-1} = G = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad Q_2 = Q_1 G = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

$$z = c + c_I Q_2 = (1,1,1,0,1,1,1,1) + (1,0)\begin{bmatrix}1 & 1 & 1 & 0 & 1 & 0 & 1 & 0\\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1\end{bmatrix} = \begin{pmatrix}0 & 0 & 0 & 0 & 0 & 1 & 0 & 1\end{pmatrix} = e$$

At this point we see that we have our error vector but an attacker wouldn't know that so we continue with the algorithm.
Now we check the weight of e_J for all information sets J s.t. $wt(e_J) = i$ from $i = 0$ to $i = j$. For $i = 0$ we have the set e_I with weight 0 and see that $wt(z + e_I Q_2) = t$ so we return $(c_I + e_I)Q_1 = c_I Q_1 = (1,1) = m$

As there are very many different variants and optimization we will only go through two of the most notable ones.

5.2 Ball collision decoding algorithm

While Stern's variant of the ISD algorithm (Collision Decoding) held its position as the asymptotically best algorithm since 1989, The paper "Smaller decoding exponents: ball-collision decoding" published in 2010 [11] that this section heavily depends on, generalized this concept into the Ball collision decoding, showing improved efficiency. Any reader interested in Stern's algorithm and the early development of ISD algorithms can look at the 2007 review article [4].

Ball collision decoding is one of several bigger development in researching more efficient ways to solve the syndrome decoding problem.

Theorem 5.2.1. *If G is a generator matrix in systematic form $G = (I_k | -A^T)$, then it's parity check matrix is in form $H = (A|I_{n-k})$.*

Proof. Since the rows of G belong to the null space of H and vice versa, we have that $GH^T = 0$. □

We can consider the parity check matrix in the form shown above, meaning that it is the dual of a generator matrix in systematic form.

We can also think of the information set as $\mathcal{I} = \{1, 2, \ldots, k\}$. This would make $U = I_{n-k}$ the identity matrix in the algorithm. The algorithm divides H and syndrome s into corresponding blocks

$$H = \begin{pmatrix}A_1 & I_1 & 0\\ A_2 & 0 & I_2\end{pmatrix} \qquad s = \begin{pmatrix}s_1\\ s_2\end{pmatrix}$$

$$A_1 \in \mathbb{F}_2^{(l_1+l_2)\times k}, A_2 \in \mathbb{F}_2^{(n-k-l_1-l_2)\times k} \quad s_1 \in \mathbb{F}_2^{l_1+l_2}, s_2 \in \mathbb{F}_2^{n-k-l_1-l_2}$$

We now give one iteration of Ball-Collision Decoding as given in [11] Constants: $n, k, w \in \mathcal{I}$ with $0 \le w, k \le n$. Parameters: p_1, p_2, q_1, q_2, k_1, k_2, l_1, $l_2 \in \mathcal{I}$

$$0 \le p_1 \le k_1 \qquad 0 \le p_2 \le k_2 \qquad k_1 + k_2 = k$$

$$0 \le q_1 \le \ell_1 \qquad 0 \le q_2 \le \ell_2$$

$$0 \le w - p_1 - p_2 - q_1 - q_2 \le n - k - \ell_1 - \ell_2$$

Input: $H \in \mathbb{F}_2^{(n-k)\times n}$ and $s \in \mathbb{F}_2^{n-k}$.
Output: Zero or more vectors $e \in \mathbb{F}_2^n$ s.t. $He = s$ and $wt(e) = w$.

1. Choose a uniform random information set $\mathcal{I}$. Subsequent steps of the algorithm write $\mathbb{F}_2^{\mathcal{I}}$ to refer to the subspace of $\mathbb{F}_2^n$ supported on $\mathcal{I}$.
2. Choose a uniform random partition of $\mathcal{I}$ into parts of sizes k_1 and k_2. Subsequent steps of the algorithm write $\mathbb{F}_2^{k_1}$ and $\mathbb{F}_2^{k_2}$ to refer to the corresponding subspaces of $\mathbb{F}_2^{\mathcal{I}}$.
3. Choose a uniform random partition of $\{1, 2, \ldots, n\} \setminus \mathcal{I}$ into parts of sizes ℓ_1, ℓ_2, and $n - k - \ell_1 - \ell_2$. Subsequent steps of the algorithm write $\mathbb{F}_2^{\ell_1}$, $\mathbb{F}_2^{\ell_2}$, and $\mathbb{F}_2^{n-k-\ell_1-\ell_2}$ to refer to the corresponding subspaces of $\mathbb{F}_2^{\{1,2,\ldots,n\}\setminus\mathcal{I}}$.
4. Find an invertible $U \in \mathbb{F}_2^{(n-k)\times(n-k)}$ such that the columns of UH indexed by $\{1, 2, \ldots, n\} \setminus \mathcal{I}$ are an $(n-k)\times(n-k)$ identity matrix. Write the columns of UH indexed by $\mathcal{I}$ as $\begin{pmatrix} A_1 \\ A_2 \end{pmatrix}$ with $A_1 \in \mathbb{F}_2^{(\ell_1+\ell_2)\times k}$, $A_2 \in \mathbb{F}_2^{(n-k-\ell_1-\ell_2)\times k}$.
5. Write Us as $\begin{pmatrix} s_1 \\ s_2 \end{pmatrix}$ with $s_1 \in \mathbb{F}_2^{\ell_1+\ell_2}$, $s_2 \in \mathbb{F}_2^{n-k-\ell_1-\ell_2}$.
6. Compute the set S consisting of all triples $(A_1x_0 + x_1, x_0, x_1)$ where $x_0 \in \mathbb{F}_2^{k_1}$, $\mathrm{wt}(x_0) = p_1$, $x_1 \in \mathbb{F}_2^{\ell_1}$, $\mathrm{wt}(x_1) = q_1$.
7. Compute the set T consisting of all triples $(A_1y_0+y_1+s_1, y_0, y_1)$ where $y_0 \in \mathbb{F}_2^{k_2}$, $\mathrm{wt}(y_0) = p_2$, $y_1 \in \mathbb{F}_2^{\ell_2}$, $\mathrm{wt}(y_1) = q_2$.
8. For each $(v, x_0, x_1) \in S$:
   For each y_0, y_1 such that $(v, y_0, y_1) \in T$:
   If $\mathrm{wt}(A_2(x_0 + y_0) + s_2) = w - p_1 - p_2 - q_1 - q_2$:
   Output $x_0 + y_0 + x_1 + y_1 + A_2(x_0 + y_0) + s_2$.

The first three steps deals with dividing out the subspaces of our vector space $\mathbb{F}_2^n$.
So $\mathbb{F}_2^{\mathcal{I}}$ is then the set where all possible nonzero positions are the the elements of $\mathcal{I}$.
Say $n = 6$, and $k = 2$. Then we set $\mathcal{I} = \{1, 3\}$.
Meaning $\mathbb{F}_2^{\mathcal{I}} = \{(0,0,0,0,0,0), (1,0,0,0,0,0), (0,0,1,0,0,0), (1,0,1,0,0,0)\}$.

Similarly for k_1 and k_2 such that $\mathcal{I}_{k_1}$ and $\mathcal{I}_{k_2}$ allow only their respective positions to be non-zero.
The same follows for $\mathbb{F}_2^{\{1,2,\ldots,n\}\setminus\mathcal{I}}$ and it's subsets. It is important to note that this is true for the matrix spaces as well $\mathbb{F}_2^{(\ell_1+\ell_2)\times k}$ and $\mathbb{F}_2^{(n-k-\ell_1-\ell_2)\times k}$ as well as for the vector subspaces containing s_1 and s_2. This is made clear with the matrix-vector multiplication in the algorithm.

Lets look at the triples we collect at the end of the algorithm.
if we start with A_1x_0 we see that we only sum the columns indexed by $\mathcal{I}_{k_1}$ similarly with A_1y_0 and $\mathcal{I}_{k_2}$
We then have that A_1x_0 and A_1y_0 both belong to $\mathbb{F}_2^{\ell_1+\ell_2}$. At this point x_1 and y_1 both add errors to their respective part of the vectors. We see that the term "Ball-Collision decoding" originates from here, if we set $q_1 = q_2 = 0, p_1 = p_2$ and $k_1 \approx k_2$ the algorithm becomes what is essentially Stern's algorithm, or "Collision decoding" as the paper calls it. The addition of x_1 and y_1 allows us some leeway in looking for matching triples, meaning instead of finding $A_1x_0 = A_1y_0$ we can find vectors within a given distance (q_1 and q_2) of A_1x_0 and A_1y_0. Or equivalently look for collisions (equality of vectors) in the sets $\{A_1x_0 + x_1 : x_1 \in \mathbb{F}_2^{\ell_1}, wt(x_1) = q_1\}$ and $\{A_1y_0 + y_1 : y_1 \in \mathbb{F}_2^{\ell_1}, wt(y_1) = q_2\}$

After this the algorithm becomes clearer and we simply look for two triples satisfying our weight check. Lets look at a general example of what this process could look like, we will not seek to decode any cipher with this but only seek to illustrate the process itself.

Example 5.2.1. *To make it simple we consider the perfect conditions given above.*

$$H = \begin{pmatrix} A_1 & I_1 & 0 \\ A_2 & 0 & I_2 \end{pmatrix} \quad s = \begin{pmatrix} s_1 \\ s_2 \end{pmatrix} \quad U = I_{n-k}$$

$$\textit{Our information set } \mathcal{I} = \{1, 2, \ldots, k\}$$

$\mathbb{F}_2^{\mathcal{I}}$ *is then the subspace such that the first k positions can be non-zero. For this example every position that can be non-zero will be a* $''1''$.

$$\{\underbrace{1, 1, \ldots, 1}_{k}, \underbrace{0, \ldots, 0}_{n-k}\} \in F_2^{\mathcal{I}}$$

similarly for k_1 *and* k_2 *we have*

$$\{\underbrace{1, 1, \ldots, 1}_{k_1}, \underbrace{0, \ldots, 0}_{k_2}, \underbrace{0, \ldots, 0}_{n-k}\} \in F_2^{k_1} \qquad \{\underbrace{0, \ldots, 0}_{k_1}, \underbrace{1, 1, \ldots, 1}_{k_2}, \underbrace{0, \ldots, 0}_{n-k}\} \in F_2^{k_2}$$

It is important to note that since k_1 *and* k_2 *is divided uniformly they can take any shape, such as.*

$$\{0, 1, 0, 1, \ldots, \underbrace{0, \ldots, 0}_{n-k}\} \in F_2^{k_1}$$

we will not consider this for our example.
We also have that $\{1, 2, \ldots, n\} \backslash \mathcal{I} = \{n-k+1, \ldots, n\} = \mathcal{N}$ *Similarly we partition* ℓ_1, ℓ_2*, such that*

$$\{\underbrace{0, \ldots, 0}_{k}, \underbrace{1, 1, \ldots, 1}_{\ell_1}, \underbrace{0, \ldots, 0}_{\ell_2}, \underbrace{0, \ldots, 0}_{n-k-\ell_1-\ell_2}\}$$

as we have done above. Since $U = I_{n-k}$ *we have that* $UH = H$ *and* $H_{\mathcal{N}} = I_{n-k}$.

$$H_{\mathcal{I}} = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \quad A_1 \in \mathbb{F}_2^{(\ell_1+\ell_2)\times k} \quad A_2 \in \mathbb{F}_2^{(n-k-\ell_1-\ell_2)\times k}$$

It is important to note that A_1 *and* A_2 *really are* $(n-k) \times n$ *matrices and we follow the same convention as above*

$$A_1 = \begin{pmatrix} A_{11} & 0 \\ 0 & 0 \end{pmatrix} \textit{ such that } A_{11} \textit{ is a } (\ell_1 + \ell_2) \times k$$

$$A_2 = \begin{pmatrix} 0 & 0 \\ A_{22} & 0 \end{pmatrix} \textit{ such that } A_{22} \textit{ is a } (n - k - \ell_1 - \ell_2) \times k$$

Now if we construct the triples mentioned above we first end up with vectors where we sum the columns of A_1 *indexed by* x_0 *and* y_0 *respectively*

$$A_1 x_0 = A_1 y_0 = \{\underbrace{1, \ldots, 1}_{\ell_1}, \underbrace{1, \ldots, 1}_{\ell_1}, \underbrace{0, \ldots, 0}_{n-k-\ell_1-\ell_2}\}$$

Of course this is only an illustrative example they will not necessarily be equal, and if they were not equal we would correct that with the added vectors x_1 and y_1 such that.

$$A_1x_0 + x_1 = \{\underbrace{1 + x_{11}, \ldots, 1 + x_{1\ell_1}}_{\ell_1}, \underbrace{1, \ldots, 1}_{\ell_1}, \underbrace{0, \ldots, 0}_{n-k-\ell_1-\ell_2}\}$$

And similarly for $A_1y_0 + y_1$. The sets of triples S and T has $\binom{k_1}{p_1}\binom{\ell_1}{q_1}$ and $\binom{k_2}{p_2}\binom{\ell_2}{q_2}$ elements respectively. After we construct the sets of triples S and T we look for collisions and continue with the weight check.

Theorem 5.2.2. ***Correctness of ball-collision decoding****[11] The set of output vectors e of the ball-collision decoding algorithm is the set of vectors e that satisfy $He = s$ and have weights $p_1, p_2, q_1, q_2, w - p_1 - p_2 - q_1 - q_2$ in $\mathbb{F}_2^{k_1}, \mathbb{F}_2^{k_2}, \mathbb{F}_2^{l_1}, \mathbb{F}_2^{l_2}, \mathbb{F}_2^{n-k-l_1-l_2}$ respectively. These weight are so we can manipulate the distribution of "ones" in the error vector that we want to look for.*

The weight check in step 8 checks if the algorithm succeeds. the equality we are looking for is.

$$UHe = UH\begin{pmatrix} x_0 + y_0 \\ x_1 + y_1 \\ A_2(x_0 + y_0) + s_2 \end{pmatrix} = \begin{pmatrix} A_1(x_0 + y_0) + x_1 + y_1 \\ A_2(x_0 + y_0) + A_2(x_0 + y_0) + s_2 \end{pmatrix} = \begin{pmatrix} s_1 \\ s_2 \end{pmatrix}$$

This algorithm gives more fine tuning of the weight distribution of the error vector e with the parameters k_1, k_2, ℓ_1, ℓ_2 dividing e into 5 different parts. With this we can look at more natural patterns of error vectors for Goppa codes.

The success probability

The algorithm requires the error vector e to have the weight distribution p_1, p_2, q_1 and q_2 in positions ranges given by k_1, k_2, ℓ_1 and ℓ_2 respectively. The probability of e having this weight distribution is then

$$b(p_1, p_2, q_1, q_2, k_1, k_2, \ell_1, \ell_2) = \binom{n}{w}^{-1}\binom{n - k - \ell_1 - \ell_2}{w - p_1 - p_2 - q_1 - q_2}\binom{k_1}{p_1}\binom{k_2}{p_2}\binom{\ell_1}{q_1}\binom{\ell_2}{q_2}$$

5.3 MMT Algorithm

The May Meurer Thomae algorithm was published published the same year, showing even more improvements. Taking the time complexity from $O(2^{0.05558n})$ down to $O(2^{0.05363n})$. It is based on Finiaz-Sendrier ISD as it is simpler than the ball collision decoding algorithm described above.

The submatrix matching problem is the problem of finding in a projected $\ell \times (k + \ell)$ submatrix a weight-p sum of columns that sums to a target syndrome $s = Hc^T = Hm^T + He^T = He^T$.

We define Q^I and Q_I as the matrices of Q such that it only contains the rows or columns of Q respectively, corresponding to I. Note that the difference here is that Q_I and Q^I are not defined as in the previous section but as we defined Q_I in section 4.1, Meaning we do not mask the unwanted columns but remove them completely.

we also define the projection of the sum of Q's columns onto the rows of L

$$\pi_L(Q) = \sum_{i=1}^{k} Q^L_{\{i\}} \in \mathbb{F}_2^{\ell}$$

Definition 5.3.1. *[12] The submatrix matching problem with parameters ℓ, k and $p \leq k + \ell$ is defined as follows. Given a random matrix $\mathbf{Q} = [q_1, \ldots, q_{k+\ell}] \in \mathbb{F}_2^{\ell \times (k+\ell)}$ (where q_i are the columns of $\mathbf{Q}$) and a target vector $\mathbf{s} \in \mathbb{F}_2^{\ell}$, find and index set I of size at most p such that the corresponding columns of $\mathbf{Q}$ sum to $\mathbf{s}$, i.e., find $I \subset \{1, 2, 3, ..., k + \ell\} = [k + \ell], |I| \leq p$ with*

$$\pi(\mathbf{Q}_I) = \sum_{i \in I} \mathbf{q}_i = \mathbf{s} \in \mathbb{F}_2^{\ell}$$

This is a vectorial variant of the subset sum problem, meaning each individual row is its own subset sum problem, we solve this problem using the column match algorithm.

We want to find two index sets I_1 and I_2 of size $p/2$ each from the set $\{1, \ldots, k+\ell\}$ so they form a partition of I, giving $\binom{p}{p/2} \approx 2^p$ different partitions. each partition of I gives the equality

$$\sum_{i \in I_1} \mathbf{q}_i = \sum_{i \in I_2} \mathbf{q}_i + \mathbf{s}$$

To guarantee that we find such an equality we add some extra parameters ℓ_1 and ℓ_2 such that $\ell_1 + \ell_2 = \ell$, that corresponds to disjoint subsets $L_1, L_2 \subset \{1, \ldots, \ell\}$ with sizes
With this we can now define two lists

$$\mathcal{L}_1 = \left\{ (I_1, \pi_{L_1}(\mathbf{Q}_{I_1})) : I_1 \subset [k + \ell], |I_1| = \frac{p}{2} \text{ and } \pi_{L_2}(\mathbf{Q}_{I_1}) = 0 \in \mathbb{F}_2^{\ell_2} \right\}$$

$$\mathcal{L}_2 = \left\{ (I_2, \pi_{L_1}(\mathbf{Q}_{I_2}) + \mathbf{s}_{L_1}) : I_2 \subset [k + \ell], |I_2| = \frac{p}{2} \text{ and } \pi_{L_2}(\mathbf{Q}_{I_2}) = \mathbf{s}_{L_2} \in \mathbb{F}_2^{\ell_2} \right\}$$

Let's look at these sets, the important parts in $\mathcal{L}_1$ is $\pi_{L_1}(Q_{I_1})$ and $\pi_{L_2}(Q_{I_1}) = 0$. We know that $\pi_{L_1}(Q_{I_1}) = \sum_{i \in I_1} q_i^{L_1}$ sums the columns but only cares about the rows whose index lie in L_1.
This ends up being a vector in $\mathbb{F}_2^{\ell_1}$

$$\begin{pmatrix} q_{i_{1,1}} + q_{i_{1,2}} + q_{i_{1,3}} \\ q_{i_{2,1}} + q_{i_{2,2}} + q_{i_{2,3}} \end{pmatrix}$$

Similarly for $\pi_{L_2}(Q_{I_1}) = 0$ which is the main restriction defining $\mathcal{L}_1$. This states that given the set of columns if there is a subset of the rows that sum up to the 0-vector we sum the remaining rows. It is easy to consider Q_{I_1} as a standalone matrix and sum it's rows, if the rows with index in L_2 is a 0-vector we want the vector that results from the remaining rows, that being $\pi_{L_1}(Q_{I_1})$.
$\mathcal{L}_2$ is defined similarly but with a different restrictions.
Doing this leaves us with $2^{p-\ell_2}$.
A simple example where L_1 is the first half and L_2 the second and similarly for I_1 and I_2.

$$A = \begin{bmatrix} Q_{I_1} & Q_{I_2} \end{bmatrix} = \begin{bmatrix} Q_{I_1}^{L_1} & Q_{I_2}^{L_1} \\ Q_{I_1}^{L_2} & Q_{I_2}^{L_2} \end{bmatrix}$$

If the sum of the rows of $Q_{I_1}^{L_2}$ is the 0-vector in $\mathbb{F}_2^{\ell_2}$ we save the sum of the rows of $Q_{I_1}^{L_1}$. Similarly for $Q_{I_2}^{L_2}$.
Now we have $\mathcal{L}_1$ and $\mathcal{L}_2$ and search through them for matching vectors $\pi_{L_1}(Q_{I_1}) = \pi_{L_1}(Q_{I_2}) + s_{L_1}$
Since I_1 and I_2 are not necessarily disjoint we can take

$$I' = (I_1 \cup I_2) \backslash (I_1 \cap I_2)$$

Elements in the intersection will cancel out so we end up with I', such that $|I'| = p - 2|(I_1 \cap I_2)|$.
Before moving describing the algorithm we show how to construct $\mathcal{L}_{1,1}$ and$\mathcal{L}_{1,2}$ with $\mathcal{L}_{2,1}$ and $\mathcal{L}_{2,2}$ being constructed similarly. We partition $I_1 = I_{1,1} \cup I_{1,2}$ such that $|I_{1,1}| = |I_{1,2}| = \frac{p}{4}$ with $I_{1,1} \subset [1, \frac{k+\ell}{2}]$ and $I_{1,2} \subset [\frac{k+\ell}{2}, k+\ell]$
we then compute the lists

$$\mathcal{L}_{1,1} = \{(I_{1,1}, \pi_{L_2}(Q_{I_{1,1}}))\}$$
$$\mathcal{L}_{1,2} = \{(I_{1,2}, \pi_{L_2}(Q_{I_{1,2}}))\}$$

The algorithm given in [12] is as follows

Algorithm 5.3.1 COLUMNMATCH

Input: $Q \in \mathbb{F}_2^{\ell \times (k+\ell)}$, $s \in \mathbb{F}_2^{\ell}$, $p \leq k + \ell$
Output: I with $\pi(Q_I) = s$ or $\perp$ if no solution is found
Parameters: L_1, L_2 with $[\ell] = L_1 \dot\cup L_2$ and $|L_i| = \ell_i$ for $i = 1, 2$

Construct $\mathcal{L}_{1,1}, \mathcal{L}_{1,2}, \mathcal{L}_{2,1}, \mathcal{L}_{2,2}$
Sort $\mathcal{L}_{1,2}, \mathcal{L}_{2,2}$ according to their labels $\pi_{L_2}(Q_{I_{1,2}}), \pi_{L_2}(Q_{I_{2,2}}) + s_{L_2}$
Join $\mathcal{L}_{1,1}$ and $\mathcal{L}_{1,2}$ to $\mathcal{L}_1$, i.e., for all $(I_{1,1}, \pi_{L_2}(Q_{I_{1,1}})) \in \mathcal{L}_{1,1}$ do
    for all $(I_{1,2}, \pi_{L_2}(Q_{I_{1,2}})) \in \mathcal{L}_{1,2}$ with $\pi_{L_2}(Q_{I_{1,1}}) = \pi_{L_2}(Q_{I_{1,2}})$ do
        $I_1 = I_{1,1} \cup I_{1,2}$. Insert $(I_1, \pi_{L_1}(Q_{I_1}))$ into $\mathcal{L}_1$
Join $\mathcal{L}_{2,1}$ and $\mathcal{L}_{2,2}$ to $\mathcal{L}_2$, i.e., for all $(I_{2,1}, \pi_{L_2}(Q_{I_{2,1}})) \in \mathcal{L}_{2,1}$ do
    for all $(I_{2,2}, \pi_{L_2}(Q_{I_{2,2}}) + s_{L_2}) \in \mathcal{L}_{2,2}$ with $\pi_{L_2}(Q_{I_{2,1}}) = \pi_{L_2}(Q_{I_{2,2}}) + s_{L_2}$ do
        $I_2 = I_{2,1} \cup I_{2,2}$. Insert $(I_2, \pi_{L_1}(Q_{I_2}) + s_{L_1})$ into $\mathcal{L}_2$
Sort $\mathcal{L}_2$ according to the label $\pi_{L_1}(Q_{I_2}) + s_{L_1}$
Join $\mathcal{L}_1$ and $\mathcal{L}_2$ to $\mathcal{L}$, i.e., for all $(I_1, \pi_{L_1}(Q_{I_1})) \in \mathcal{L}_1$ do
    for all $(I_2, \pi_{L_1}(Q_{I_2}) + s_{L_1}) \in \mathcal{L}_2$ with $\pi_{L_1}(Q_{I_1}) = \pi_{L_1}(Q_{I_2}) + s_{L_1}$ do
        Output $I_1 \Delta I_2 = (I_1 \cup I_2) \setminus (I_1 \cap I_2)$
Output $\perp$ =0

We show a small example before moving on to the MMT algorithm.

Example 5.3.1. *Set $k = 2$, $\ell = 4$ and $p = 4 \leq 6$ this makes Q an 4×6 matrix and s a vector of length 4. We set Q and s to be the following.*

$$Q = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}, \quad s = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

We set our parameters to be $L_1 = \{1, 3\}$ and $L_2 = \{2, 4\}$.
The first step is to construct the lists $\mathcal{L}_{i,j}$ where $i, j = 1, 2$ We do this by taking creating random subsets $I_{i,j}$ of [6] following the constraint given above for the lists. This gives us

$$I_{1,1} = \{1\}, I_{1,2} = \{4\}, I_{2,1} = \{2\}, I_{2,2} = \{5\}$$

Which in turn gives us the lists.

$$\mathcal{L}_{1,1} = \{I_{1,1}, \begin{pmatrix}0\\1\end{pmatrix}\}, \quad \mathcal{L}_{1,2} = \{I_{1,2}, \begin{pmatrix}0\\1\end{pmatrix}\}, \quad \mathcal{L}_{2,1} = \{I_{2,1}, \begin{pmatrix}0\\0\end{pmatrix}\}, \quad \mathcal{L}_{2,2} = \{I_{2,2}, \begin{pmatrix}0\\0\end{pmatrix}\}$$

At this point we take every match between $\mathcal{L}_{1,1}$ and $\mathcal{L}_{1,2}$ and put into $\mathcal{L}_1$ but replace π_{L_2} with π_{L_1} similarly with $\mathcal{L}_{2,1}$ and $\mathcal{L}_{2,2}$. We end up with

$$\mathcal{L}_1 = \{\{1,5\}, \begin{pmatrix}0\\0\end{pmatrix}\}, \quad \mathcal{L}_2 = \{\{3,4\}, \begin{pmatrix}1\\1\end{pmatrix}\}$$

Here we again do the same matching of elements in $\mathcal{L}_1$ and $\mathcal{L}_2$ but since there is no matching elements we find no solution. We could tweak the values in $Q, I_{i,j}$ and L_i to give us such a match but since this illustrates the concept well enough we will leave it at that. In a successful run of the algorithm we are given an information set we can use in the MMT algorithm below.

MMT Algorithm
As usual $H \in \mathbb{F}_2^{(n-k)\times n}$ is our parity check matrix for an $[n,k,d]$-code C. We want to decode $c = m + e$ such that $wt(e) = \lfloor \frac{d-1}{2} \rfloor = t$. To achieve this we want t columns of H that sum to $s(c) = Hc^T$.
We start by putting H in semi systematic form.

$$\tilde{H} = U_G H U_P = \left(Q \quad \middle| \quad \begin{matrix}0\\ I_{n-k-\ell}\end{matrix} \right)$$

With U_P being a permutation matrix on the columns of H and U_G being a Gaussian elimination matrix. This also permutes e so we have $\tilde{e} = U_P e$. We set $p \leq t$ as an optimization parameter. We need that the t positions containing a 1 have a distribution of $\frac{p}{2}, \frac{p}{2}$ and $w - p$ in $[1, \frac{k+\ell}{2}], [\frac{k+\ell}{2} + 1, k + \ell]$ and $[k + \ell + 1, n]$ respectively. We look at $Q^{[1,\ell]}$ for a weight-p sum of the columns that exactly matches the first ℓ rows of $s(x)$.

$$\pi_{[1,\ell]}(Q) = s^{[1,\ell]}(x) \qquad wt(\pi_{[1,\ell]}(Q)) = p$$

we apply these to the column match algorithm, for every iteration that we get at least one I s.t. $\pi_{[1,\ell]}(Q_I) = s^{[1,\ell]}(x)$ we check the weight of their difference $wt(\pi(Q_I - s(x))) = t - |I|$. We can correct these by choosing unit vectors from $I_{n-k-\ell}$

The support function supp($\mathbf{v}$) returns the indices of non-zero elements in the vector $\mathbf{v}$.

Algorithm 5.3.2 MMT

Input: Parity check matrix $\mathbf{H} \in \mathbb{F}_2^{(n-k)\times n}$, syndrome $\mathbf{s}(x) = \mathbf{H}e^t$ with $\mathrm{wt}(e) = \omega$
Output: Error $e \in \mathbb{F}_2^n$
Parameters: p, ℓ, ℓ_1, ℓ_2 with $\ell = \ell_1 + \ell_2$

Repeat
  Compute $\widehat{\mathbf{H}} \leftarrow \mathrm{Init}(\mathbf{H})$ where $\widehat{\mathbf{H}} = U_G \mathbf{H} U_P$
  For all (solutions I found by $\mathrm{ColumnMatch}(\mathbf{Q}[\ell], (U_G s^t(x))[\ell], p, \ell_1, \ell_2)$) **do**
    If $\mathrm{wt}(\pi(\mathbf{Q}_I) + U_G s^t(x)) = \omega - |I|$ **then**
      Compute $\tilde{e} \in \mathbb{F}_2^n$ by setting
      $\tilde{e}_i = 1 \; \forall \; i \in I$
      $\tilde{e}_{k+\ell+j} = 1 \; \forall \; j \in \mathrm{supp}(\pi_{[n-k]\setminus[\ell]}(\mathbf{Q}_I + U_G s^t(x)))$
      Output $e = \tilde{e} U_P^t$

6 Quantum ISD

6.1 Introduction

Quantum computers and quantum computing is a revolution within the field of computer science. Instead of classical bits as we have now, they use qubits, that can exist in multiple states simultaneously.

Several problems whose difficulty we rely on for our current cryptography such as integer factorization problem and the discrete logarithm problem are solved exponentially faster with quantum computers using Shor's algorithm. This renders those cryptosystems useless.

In our use case the syndrome decoding problem has only quadratic improvement with the use of Grovers algorithm and the quantum walk search algorithm. Hence the solution is simply to use larger keys.

The advent of quantum computing changes the P vs NP landscape and introduces BQP as the set of problems that has an efficient solution on a quantum computer.

Now due to the exponential increase that Shor's algorithm brings the cryptographic community has to work to replace these broken cryptographic systems with systems that can resist attackers with quantum computers. This is why the US national institute of standards and technology (NIST) has started a competition to test and find replacements for these broken systems. Mceliece has been a candidate and is still in the running which is now at the fourth round since July 5, 2022.
In this chapter we aim to give a short overview of quantum computing and explain the best we can how Grover's and the quantum walk algorithm works and how they are integrated into our previously discussed MMT algorithm.

6.2 Basics of Quantum Computing

Quantum computing is heavily dependent on linear algebra as the reader will see.
This section and the next is heavily reliant on [13].
The basics of quantum computing starts with qubits. the base vectors of a qubit are usually

$$|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}, \quad |1\rangle = \begin{pmatrix}0\\1\end{pmatrix}$$

But also common are

$$|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}, \quad |-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

these are then called our basis states.
A quantum state of a qubit is then defined as the superposition of the basis states.

$$|\Psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,, \quad |\alpha|^2 + |\beta|^2 = 1$$

Where the qubit exist in both states simultanously and can fall into one or the other with probability α or β respectively.
This superposition of states is valid for any set of vectors, it is then as above just a linear combination of vectors.

$$|\Psi\rangle = \sum_i \alpha_i\,|\psi_i\rangle$$

We denote the complex conjugate and the transpose of a matrix A with A^* and A^T respectively.

Definition 6.2.1. ***Inner product*** *An inner product on a complex vector space V is a mapping $(\cdot,\cdot) : V \times V \to \mathbb{C}$ s.t. $\forall x, y, z \in V$ and all $\lambda \in \mathbb{C}$*

- $(x, y) = \overline{(y, x)}$
- $(\lambda x, y) = \lambda(x, y)$
- $(x + y, z) = (x, z) + (y, z)$
- $(x, x) > 0$ *when* $x \neq 0$
- $(x, x) = 0$ *when* $x = 0$

We will denote the inner product between vectors $|\phi\rangle$ and $|\psi\rangle$ as $\langle\psi|\phi\rangle$, or between $|\phi\rangle$ and $A\,|\psi\rangle$ (or equivalently $A^+\,|\phi\rangle$ and $|\psi\rangle$) as $\langle\psi|\,A\,|\phi\rangle$

Definition 6.2.2. ***Hilbert space*** *A Hilbert space is an inner product space which is a complete metric space with respect to the metric induced by it's inner product.*

Definition 6.2.3. ***Hermitian conjugate (Adjoint)*** *The adjoint of a matrix A is simply the complex conjugate of the transpose of the matrix, meaning $A^+ = (A^T)^*$.*

For a vector $|\phi\rangle$ it's dual vector $\langle\phi|$ as it's normally called is simply the adjoint of the vector.

Example 6.2.1. *given* $|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}$ *we have that* $\langle 0| = \begin{pmatrix}1 & 0\end{pmatrix}$

Definition 6.2.4. ***Linear operator*** *A linear operator is any function* $A : V \to W$ *such that*

$$A\left(\sum_i a_i |v_i\rangle\right) = \sum_i a_i A(|v_i\rangle)$$

We will consider any operator from here on to simply be a matrix

Definition 6.2.5. ***Unitary operator*** *A matrix is said to be unitary if* $A^+A = I$

The tensor product is a way of creating a larger vector space from two given ones.

Definition 6.2.6. *The tensor product* $\otimes$ *of two vectors* $|\psi\rangle \in \mathbb{C}^n$ *and* $|\phi\rangle \in \mathbb{C}^m$ *such that* $|v\rangle = |\psi\rangle \otimes |\phi\rangle \in \mathbb{C}^{nm}$. *Such that* $v_{(i,j)} = \phi_i * \psi_j$ *where the subscript gives the component position in lexicographic order, meaning the first component is checked first and the second after.*
We sometimes denote the tensor product of two vectors as $|\psi\rangle |\phi\rangle$ *or* $|\psi\phi\rangle$ *for brevity.*
We also write out $|\phi\rangle^{\otimes n}$ *when applying the tensor product of the same vector* n *times.*
The tensor product satisfies the following properties.

1. *For an arbitrary scalar* $z \in \mathbb{C}$ *and elements*$|\psi\rangle \in \mathbb{C}^n$ *and* $|\phi\rangle \in \mathbb{C}^m$,

$$z(|\psi\rangle \otimes |\phi\rangle) = z|\psi\rangle \otimes |\phi\rangle = |\psi\rangle \otimes z|\phi\rangle .$$

2. *For arbitrary* $|\psi_1\rangle, |\psi_2\rangle \in \mathbb{C}^n$ *and* $|\phi\rangle \in \mathbb{C}^m$,

$$(|\psi_1\rangle + |\psi_2\rangle) \otimes |\phi\rangle = |\psi_1\rangle \otimes |\phi\rangle + |\psi_2\rangle \otimes |\phi\rangle .$$

3. *For arbitrary* $|\psi\rangle \in \mathbb{C}^n$ *and* $|\phi_1\rangle, |\phi_2\rangle \in \mathbb{C}^m$,

$$|\psi_1\rangle \otimes (|\phi\rangle + |\phi\rangle) = |\psi\rangle \otimes |\phi_1\rangle + |\psi\rangle \otimes |\phi_2\rangle .$$

We also note that the adjoint operation distribute over the tensor product

$$(A^{\otimes n})^+ = (A^+)^{\otimes n}$$

Example 6.2.2.

$$\begin{pmatrix}1\\2\end{pmatrix} \otimes \begin{pmatrix}3\\4\end{pmatrix} = \begin{pmatrix}1*3\\1*4\\2*3\\2*4\end{pmatrix} = \begin{pmatrix}3\\4\\6\\8\end{pmatrix}$$

For matrices we would have the Kronecker product

Example 6.2.3.

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} A_{11}\mathbf{B} & A_{12}\mathbf{B} & \cdots & A_{1n}\mathbf{B} \\ A_{21}\mathbf{B} & A_{22}\mathbf{B} & \cdots & A_{2n}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1}\mathbf{B} & A_{m2}\mathbf{B} & \cdots & A_{mn}\mathbf{B} \end{bmatrix}$$

Nielsen and Chuang in their book[13] define the hadamard operator on one qubit as

Definition 6.2.7.

$$H = \frac{1}{\sqrt{2}}\Big[(|0\rangle + |1\rangle)\langle 0| + (|0\rangle - |1\rangle)\langle 1|\Big]$$

and we take this oppurtunity to write this out as a matrix in order to illustrate how we can work with this notation. We start simply by rewriting the notation in normal linear algebra.

$$H = \frac{1}{\sqrt{2}}\left[(\begin{pmatrix}1\\0\end{pmatrix} + \begin{pmatrix}0\\1\end{pmatrix})\begin{pmatrix}1 & 0\end{pmatrix} + (\begin{pmatrix}1\\0\end{pmatrix} - \begin{pmatrix}0\\1\end{pmatrix})\begin{pmatrix}0 & 1\end{pmatrix}\right]$$

Now it seems obvious how to continue.

$$H = \frac{1}{\sqrt{2}}\left[\begin{pmatrix}1\\1\end{pmatrix}\begin{pmatrix}1 & 0\end{pmatrix} + \begin{pmatrix}1\\-1\end{pmatrix}\begin{pmatrix}0 & 1\end{pmatrix}\right] = \frac{1}{\sqrt{2}}\left[\begin{bmatrix}1 & 0\\1 & 0\end{bmatrix} + \begin{bmatrix}0 & 1\\0 & -1\end{bmatrix}\right] = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}$$

This does not mean the notation used in quantum computation is meaningless it has its value in the density of information, for example

Example 6.2.4. *Say we want to represent n-qubits then we can write* $\underbrace{|00\ldots0\rangle}_{n}$ *which represents a vector in* $\mathbb{C}^{2^n}$ *with only n values. Let's look at some specific values of n. For* $n = 1$ *we have the vectors* $|0\rangle$ *and* $|1\rangle$ *that we have been using so far. For* $n = 2$ *we would normally have a vector of length* $2^2 = 4$ *but can simply write*

$$|00\rangle = \begin{pmatrix}1\\0\\0\\0\end{pmatrix}, |01\rangle = \begin{pmatrix}0\\1\\0\\0\end{pmatrix}, |10\rangle = \begin{pmatrix}0\\0\\1\\0\end{pmatrix}, |11\rangle = \begin{pmatrix}0\\0\\0\\1\end{pmatrix}.$$

Now we would also like to look at $H^{\otimes n}$*. From [13] we get the definition,*

Definition 6.2.8.

$$H^{\otimes n} = \frac{1}{\sqrt{2^n}}\sum_{x,y}(-1)^{xy}|x\rangle\langle y|$$

where $|x\rangle$ *and* $|y\rangle$ *are the basis vectors of our vector space. indexed by* x *and* y*, meaning if* $x = 2$ *then the third (offset by one) element in the vector is* 1*.*

An example for $n = 2$ *would then be* 4^2 *vector operations, which is very tedious and uninspiring work hence we choose to write it out using the tensor product for matrices that we defined above.*

Example 6.2.5.

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix} \otimes \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix} = \frac{1}{2}\begin{bmatrix}1 & 1 & 1 & 1\\1 & -1 & 1 & -1\\1 & 1 & -1 & -1\\1 & -1 & -1 & 1\end{bmatrix}$$

Now we wanna look at how these matrices possibly affect our qubits, we look at some example.

Example 6.2.6. *for $n = 1$ we have $H, |0\rangle$ and $|1\rangle$*

$$H|0\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{pmatrix}1\\0\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix} = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = |+\rangle$$

$$H|1\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{pmatrix}0\\1\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\-1\end{pmatrix} = \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |-\rangle$$

It is common to see $|+\rangle$ and $|-\rangle$ in the litterature and for good reason, but we will not use this here. We move on to $n = 2$

Example 6.2.7.

$$H^{\otimes 2}|00\rangle = \frac{1}{2}\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -1 & 1 & -1\\ 1 & 1 & -1 & -1\\ 1 & -1 & -1 & 1\end{bmatrix}\begin{pmatrix}1\\0\\0\\0\end{pmatrix} = \frac{1}{2}\begin{pmatrix}1\\1\\1\\1\end{pmatrix} = \frac{|00\rangle + |01\rangle + |10\rangle + |11\rangle}{2}$$

Now the similarity in these is obviously the scalar component, if we look at the first one we see that $4|\frac{1}{\sqrt{2}}|^2 = 1$. Similarly for the last one with two qubits we have $4|\frac{1}{2}|^2 = 1$.
What this tells us is that we can use the Hadamard matrix to go from our initial state $|0\rangle^{\otimes n}$ to a superposition of equal probability for all states.
We can write this in it's general form,

$$H^{\otimes n}|0\rangle^{\otimes n} = \frac{1}{\sqrt{2^n}}\sum_{j=0}^{2^n-1}|j\rangle$$

Notice that the Hadamard transform is its own adjoint, meaning $H^+H = I = HH^+$. Followed with the distributivity of the adjoint over tensor products we have that $H^{\otimes n}H^{\otimes n} = I$. This will be used in Grover's algorithm.
The Pauli matrix X will also used later in Grover's algorithm so we will quickly define it, if interested in the other Pauli matrices refer to [13].

Definition 6.2.9. ***Pauli matrix*** *X*

$$X = \sigma_x = \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$$

6.3 Grover's Algorithm

Grover's algorithm is a quantum algorithm that searches through an unstructured set of elements for a solution with a quadratic speedup compared to classical algorithms.
Given a search space of $N = 2^n$ elements and $M \subset N$ possible solutions Grover's algorithm uses a oracle to recognize a solution to our search problem. The oracle is a unitary operator O. If we consider $|x\rangle$ to be our index register, that is $x \in [0, \ldots, N-1]$ and $|q\rangle$ to be our oracle qubit. The action of our oracle on these qubits would then result in

$$O|x\rangle|q\rangle = |x\rangle|q \oplus f(x)\rangle$$

where $\oplus$ is addition mod 2, and

$$f(x) = \begin{cases} 1 & x \in M \\ 0 & x \notin M \end{cases}$$

This essentially means we flip the oracle qubit if we find a solution. If we then use $(|0\rangle - |1\rangle)/\sqrt{2}$ as our oracle qubit we get that our oracle qubit only changes in sign.

$$O|x\rangle\left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) = (-1)^{f(x)}|x\rangle\left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right).$$

with this we can omit the oracle qubit and only write

$$O|x\rangle = (-1)^{f(x)}|x\rangle$$

The procedure itself starts by putting the computer in the state $|0\rangle^{\otimes n}$, then we apply the Hadamard transform putting the computer in an equal superposition state.

$$|\psi\rangle = \frac{1}{\sqrt{N}}\sum_{x=0}^{N-1}|x\rangle, \quad \left|\frac{1}{\sqrt{N}}\right|N = 1$$

Now we repeatedly apply Grover's operator G, which we describe in the four steps below.

1. *Apply oracle O*
2. *Apply Hadamard transform $H^{\otimes n}$*
3. *Conditional phase shift on all basis states but $|0\rangle$*

$$|x\rangle \to -(-1)^{\sigma_{x_0}}|x\rangle$$

4. *Apply Hadamard transform $H^{\otimes n}$*

Note that step 2,3 and 4 gives

$$H^{\otimes n}(2|0\rangle^{\otimes n}\langle 0|^{\otimes n} - I)H^{\otimes n} = 2|\psi\rangle\langle\psi| - I$$

And Grover's operator is defined as

$$G = (2|\psi\rangle\langle\psi| - I)O$$

We formally write out the algorithm as
The algorithm starts with the initial state and applies $H^{\otimes n}$ to the first n qubits, and HX to the last, this is step 1 and 2. It then applies the Grover operator $R \approx \lceil \pi\sqrt{2^n}/4 \rceil$ times. At this point it measures the qubits and returns the resulting vector.
Let's look a little bit at the result from steps $2, 3$ and 4.

$$H^{\otimes n}(2|0\rangle^{\otimes n}\langle 0|^{\otimes n} - I)H^{\otimes n} = 2H^{\otimes n}|0\rangle\langle 0|H^{\otimes n} - H^{\otimes n}IH^{\otimes n} = 2|\psi\rangle\langle\psi| - I$$

Though it is simple, we write it out for clarity.

Algorithm 6.3.1 Grover's Algorithm

Input:A black box oracle as described above and $n+1$ qubits in the state $|0\rangle$
Output: x_0

1. $|0\rangle^{\otimes n}|0\rangle$
2. $\rightarrow \frac{1}{\sqrt{N}}\sum_{x=0}^{2^n-1}|x\rangle\left[\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right]$
3. $\rightarrow [(2|\psi\rangle\langle\psi|-I)O]^R\frac{1}{\sqrt{N}}\sum_{x=0}^{2^n-1}|x\rangle\left[\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right]\approx|x_0\rangle\left[\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right]$
4. return x_0

6.4 Quantum Walk Algorithm

The quantum walk algorithm is another algorithm we will use to further improve on our ISD algorithm, the problem it aims to solve is finding a vertex belonging to a specific subset of vertices in a graph. More specifically for us we will be using a Johnson Graph.

Definition 6.4.1. ***Johnson Graph:****[14] A Johnson Graph $J(x,r)$ is a graph, in which every vertex is labeled by an r-element subset V of the set $\{1,2,\ldots,x\}$, and in which two vertices, U and V, are adjacent to each other if and only if $|U\cap V|=r-1$.*

In our case we have a graph $G=J(x,r)=(V,E)$ where V and E denote our vertices and edges respectively. We define A_G as our adjacency matrix and $P_G=\frac{A_G}{r(x-r)}$ as our stochastic transition matrix. The stochastic transition matrix tells us the probability of transition to each vertex. We define the quantum state of a vertex i with $|i\rangle$ and $|ij\rangle$ the quantum state of of the edge (i,j)
We define H^E to be the hilbert space associated with our edge set E

Definition 6.4.2. *[14] A hilbert space associated with an edge set E, denoted H^E is a hilbert space whose basis vectors are given by the elements in E*

We also define U_o and U_d

$$U_o(|i\rangle|j\rangle)=\begin{cases}-|i\rangle|j\rangle & i\in M\\ |i\rangle|j\rangle & i\notin M\end{cases},\quad U_d(|i\rangle|j\rangle)=U_{dL}(U_{dR}(|i\rangle|j\rangle))$$

Where

$$U_{dR}=2\sum_{x\in V}|\Phi_x\rangle\langle\Phi_x|-I_{|V|^2}\quad U_{dL}=2\sum_{y\in V}|\Psi_y\rangle\langle\Psi_y|-I_{|V|^2}$$

and

$$|\Phi_x\rangle=|x\rangle\Bigg(\sum_{y\in V,(x,y)\in E}\sqrt{P_G[x][y]}\,|y\rangle\Bigg),\quad |\Psi_y\rangle=\Bigg(\sum_{x\in V,(y,x)\in E}\sqrt{P_G[y][x]}\,|x\rangle\Bigg)|y\rangle$$

Where $P_G[x][y]$ is the (x,y) component of the matrix.

Algorithm 6.4.1 QW search algorithm[14]

Input: $G = J(x, r) = (V, E \subset V \times V), P_G, M \subset V$
Output: $x \in M$

1. $|\psi\rangle \leftarrow |0^n\rangle$
2. $|\psi\rangle \leftarrow H^{\otimes n} |\psi\rangle$
3. for $i = 1$ to $\lfloor \frac{1}{\sqrt{\epsilon\delta}} \rfloor$
4. $\quad |\psi\rangle \leftarrow U_o |\psi\rangle$
5. $\quad |\psi\rangle \leftarrow U_d |\psi\rangle$
6. return $|\psi\rangle$

Let's take some time and look at what our operators do. The first operator is simple enough, just as in with the oracle operator in Grover's algorithm we flip the sign if we find a vertex i that belongs to our set of solutions M.
Now the more difficult operator is U_d, we will start by explaining from the bottom up and try to give a concise explanation of what the operator does. Starting with $|\Phi_x\rangle = |x\rangle \left(\sum_{y \in V,(x,y)\in E} \sqrt{P_G[x][y]} \, |y\rangle \right)$, the sum being similar to the superposition of equal states this is the superposition of the transition probabilities from the vertex x to the adjacent vertices y. This sum ends up as a vector as we know and we simply take the tensor product of the quantum state of the vertex x with this superposition. We define $|\Psi_y\rangle$ similarly. One step up we have the so called reflection operators U_dL and U_dR, they are called so since they reflect a vector around the subspace spanned by $|\Phi_x\rangle$ and $|\Psi_y\rangle$. What this means is that given a vector $|\phi\rangle = |\phi_\perp\rangle + \big|\phi_{||}\big\rangle$ where $\big|\phi_{||}\big\rangle$ is the component in the subspace and $|\phi_\perp\rangle$ is orthogonal to the subspace. Orthogonal to a subspace means the vector is a linear combination other basis vectors than those spanning the subspace and cant be defined using those. One can consider the subspace being two dimension and the orthogonal vector being in three dimensions and the vector being reflected around the 2d-plane to give a simple geometric intuition. The orthogonal vector isn't dependent on the set of basis vectors that span the subspace and is reflected around those basis vectors. This ends up being

$$U_{dL} |\phi\rangle = U_{dL}(\big|\phi_{||}\big\rangle + |\phi_\perp\rangle) = \big|\phi_{||}\big\rangle - |\phi_\perp\rangle \, .$$

And of course $U_{dR} |\phi\rangle$ works similarly. After we have done this a set number of times we arrive at our solution. This set number of times,meaning the values given in the loop are $\epsilon = \frac{|M|}{|V|}$ and the spectral gap is $\delta = \frac{x}{r(x-r)}$.

6.5 Quantum MMT Algorithm

We first use Grover's algorithm to find an appropriate permutation matrix. Let V be the entire set of $n \times n$ permutation matrices, and a function $f : V \to \mathbb{F}_2$ equals 1 if there exists a solution for the columnmatch algorithm and 0 otherwise. It is important here to note that the function simply recognizes a solution not that it knows a solution. This permutation coupled with Gaussian elimination puts our parity-check matrix in semi systematic form.

Now the task of finding the error vector(or more specifically the information set) is done using the quantum walk algorithm. Recall from the section on the MMT algorithm that we divide our information set in four section, here we describe each section using a Johnson graph. To do this we need to define the products of graphs.

Definition 6.5.1. ***The product of graphs*** *For finite graphs $G_1 = (v_1, E_1)$ and $G_2 = (v_2, E_2)$, the product being $G = G_1 \times G_2 = (V, E)$ where $V = V_1 \times V_2$ and $E = \{(u_1u_2, v_1v_2) | (u_1 = v_1 \wedge (u_2, v_2) \in E_2) \vee (u_2 = v_2 \wedge (u_1, v_1) \in E_1)\}$*

Essentially what this definition says is that we can only move in one graph at a time. This can then easily be extended to the product of four graphs $J(N, R) = J_{11}(N, R) \times J_{12}(N, R) \times J_{21}(N, R) \times J_{22}(N, R)$. Where $N = \binom{k+\ell}{\frac{p}{4}}$ and r is the number of solution satisfying the column match algorithm. M is then the set of all solution satisfying the column match algorithm.

Algorithm 6.5.1 Quantum MMT Algorithm

Input: $n, k, w, H, s, p, \ell, \ell_1, \ell_2, \epsilon$
Output: e

1. $e \leftarrow 0^n$
2. while $e == 0^n$
3. $\quad P \leftarrow \text{Grovers}(\ell, H)$
4. $\quad Q, U \leftarrow \text{Gaussian}(HP)$
5. $\quad s^- \leftarrow Us$
6. $\quad e^- \leftarrow QW(Q, p, \ell_1, \ell_2, s^-)$
7. $\quad$ **if** $wt(e^-) == w - p - 4\epsilon$ **then**
8. $\quad\quad e \leftarrow Pe^-$
9. return e

The main difference from the normal MMT algorithm is that we use Grover's algorithm for finding an appropriate permutation matrix and the quantum walk algorithm for finding an error vector.

6.6 Post quantum cryptography as a new field

Post quantum cryptography is still a very new field, reviewing the current litterature specifically for code based cryptography has not lead to much understanding of how these quantum algorithms work.

This chapter sought to give an introduction to the topic and the relevant math but there is much more to learn in quantum computing before the algorithms can be properly understood.

Quantum computing will revolutionize the field of cryptography and will bring many dangers and insecurities with it. Cryptography now relies on a long history of research and attempts at breaking a cryptosystem for it to be considered secure. The McEleice PKE for example has 40 years of research on it with no practical improvement in decoding a message quickly without the private key. Quantum computing threatens this balance by bringing in a completely new paradigm of attacks. so far there is no known attacks with exponential speedup. But as quantum computing becomes more and more an integral part of computer science, more and more people research it and the exploration of quantum computing will snowball into possible disasters for the end users of these crytographic systems.
This is why it is so important to quickly find and research any possible cryptosystems before quantum computers are widely available so we don't have to deal with any potential catastrophes.
It might also be of value to base our cryptosystems on several different problems in the union of NP-complete and QMA-complete problems so as to minimize any future damage.

7 Conclusion

In this paper we sought to give a current review article of McEliece used with Goppa codes and to explore it's inner workings akin to Overbecks article from 2007[4], we also wanted a more up to date on current attack algorithms and how quantum algorithms can be applied to improve on these algorithms.
We start by constructing finite fields from the foundation of algebraic structures and show how they can be used to define our Goppa codes. We proved the minimum distance of vectors in a Goppa code to be $2t + 1$ *where* $t \in \mathbb{N}$ *is the degree of our Goppa polynomial, and then showed how we can correct up to* t *errors in a codevector using pattersons algorithm. From there we defined the McEliece PKE using Goppa codes and showed how to encrypt and decrypt and proved some results showing specifically why the decryption works. From there we showed that the PKE is only OW-CPA secure and why this matters for practical purposes, we then solved this problem by defining KEM's and the McEliece KEM which are IND-CCA2 secure.*
Then we look at two notable ISD algorithms one of which (MMT) is currently one of the most efficient classical methods for solving the syndrome decoding problem.
Lastly we give a brief introduction to quantum computation and try to explain Grover's and the quantum walk algorithm and show how these can be used to optimize the MMT-algorithm.

Many parts in this thesis can be lacking in details most notably the section on ISD algorithm and the quantum walk algorithm, also the section on finite fields could be more brief.

8 Appendix

8.A How to find the multiplicative inverse of a polynomial modulo a irreducible polynomial belonging to a polynomial ring over a finite field.

This section will show how we found the inverses to the polynomials defining the syndrome of our Goppa code given the values chosen for our example.
Remember that since we have a characteristic of 2, we have that $1 = -1$. We list the elements of our finite field in both forms to help with any verification.

$$\{0, 1, \lambda, \lambda^2, \lambda^3 = \lambda + 1, \lambda^4 = \lambda^2 + \lambda, \lambda^5 = \lambda^2 + \lambda + 1, \lambda^6 = \lambda^2 + 1\}$$

The Goppa polynomial is $x^2 + x + 1$

- x

$$1 = x(ax + b) = ax^2 + bx = ax + a + bx$$
$$a + b = 0 \quad a = 1 \rightarrow b = 1$$
$$x(x + 1) = 1$$

- $x - 1$ *see above.*

- $x - \lambda$

$$1 = (x + \lambda)(ax + b) = ax + a + bx + a\lambda x + b\lambda$$
$$a + b + a\lambda = 0 \quad a + b\lambda = 1$$
$$b = a\lambda^3 \quad a + b\lambda = 1 = a + a\lambda^4 = a\lambda^5 \rightarrow a = \lambda^2 \text{ and } b = \lambda^5$$
$$(x - \lambda)(\lambda^2 x + \lambda^5) = 1$$

- $x - \lambda^2$

$$1 = (x + \lambda^2)(ax + b) = ax + a + bx + a\lambda^2 x + \lambda^2 b$$
$$a + b + a\lambda^2 = 0 \rightarrow b = a\lambda^6$$
$$a + b\lambda^2 = 1 = a(\lambda^3) \rightarrow a = \lambda^4 \quad b = \lambda^3$$
$$(x + \lambda^2)(\lambda^4 x + \lambda^3) = 1$$

- $x - \lambda^3$

$$1 = (x + \lambda^3)(ax + b) = ax + a + bx + a\lambda^3 x + b\lambda^3$$
$$a + b + a\lambda^3 = 0 \rightarrow b = a\lambda$$
$$a + b\lambda^3 = 1 = a\lambda^5 \rightarrow a = \lambda^2 \quad b = \lambda^3$$
$$(x + \lambda^3)(\lambda^2 x + \lambda^3) = 1$$

- $x - \lambda^4$

$$1 = (x + \lambda^4)(ax + b) = ax + a + bx + a\lambda^4 x + b\lambda^4$$
$$a + b + a\lambda^4 = 0 \to b = a\lambda^5$$
$$a + b\lambda^4 = 1 = a + a\lambda^2 = a\lambda^6 \to a = \lambda \quad b = \lambda^6$$
$$(x + \lambda^4)(\lambda x + \lambda^6) = 1$$

- $x - \lambda^5$

$$1 = (x + \lambda^5)(ax + b) = ax + a + bx + a\lambda^5 x + b\lambda^5$$
$$a + b + a\lambda^5 = 0 \to b = a\lambda^4$$
$$a + b\lambda^5 = 1 = a\lambda^6 \to a = \lambda \quad b = \lambda^5$$
$$(x + \lambda^5)(\lambda x + \lambda^5) = 1$$

- $x - \lambda^6$

$$1 = (x + \lambda^6)(ax + b) = ax + a + bx + a\lambda^6 x + b\lambda^6$$
$$a + b + a\lambda^6 = 0 \to b = a\lambda^2$$
$$a + b\lambda^6 = 1 = a\lambda^3 \to a = \lambda^4 \quad b = \lambda^6$$
$$(x + \lambda^6)(\lambda^4 x + \lambda^6) = 1$$

8.B Niederreiter

Algorithm 8.B.1 Niederreiter

- **System Parameters:** $n, t \in \mathrm{N}$, where $t << n$.
- **Key Generation:** Given the parameters n,t generate the following matrices:
  H:$(n-k) \times n$ check matrix of a binary code $\mathcal{G}$ which can correct up to t errors
  M: $(n-k) \times (n-k)$ random binary non-singular matrix
  P: $n \times n$ random permutation matrix
  Then, compute the systematic $n \times (n-k)$ matrix $H^{pub} = MHP$.
- **Public Key:**(H^{pub},t)
- **Private Key:** (P,$\mathcal{D}_{\mathcal{G}}$, M), where $\mathcal{D}_{\mathcal{G}}$ is an efficient syndrome decoding algorithm for $\mathcal{G}$.
- **Encryption:** A message $\mathbf{m}$ is represented as a vector $\mathbf{e} \in \{0,1\}^n$ of weight t, called plaintext. To encrypt it, we compute the syndrome

$$s = H^{pub}\mathbf{e}^T$$

  .
- **Decryption** To decrypt a ciphertext $\mathbf{s}$ calculate

$$M^{-1}\mathbf{s} = HP\mathbf{e}^T$$

  first, and apply the syndrome decoding algorithm $\mathcal{D}_{\mathcal{G}}$ for $\mathcal{G}$ to it in order to recover $P\mathbf{e}^T$. Now we can obtain the plaintext $\mathbf{e}^T = P^{-1}P\mathbf{e}^T$.

8.C Code

```
    import galois
import numpy as np
import random as rand

rand.seed()



prima = 2 ## prime
orda = 3 ## prime power
fOrd = prima**orda   #field order
## o = 3 D = 2 gives three codewords
## o = 4 D = 2 gives an insane amount
## o = 4 D = 3 gives 15 codewords

gPolDeg = 2 ## degree of goppa polynomial
## Since the degree of the goppa polynomial increases the weight of codeword it makes far
                                            less vectors viable
## how does this impact security and other aspects of the cryptosystem?
## the smaller the degree is the less values of the message vector we can hide


GF = galois.GF(prima, orda, repr="poly") ##defining our finite field

###increase code support size minize goppa degree for maximum amount of codewords.



print(GF.properties)
print(GF.elements)



goppaPol = galois.irreducible_poly(fOrd, gPolDeg ) ## find irreducible polynomial with
                                            degree gPolDeg ##goppa polynomial

print("goppa polynomial: ",goppaPol)

codeSupportPol = []     ## list of polynomials used in defining the syndrome
codeSupportPolInv = []  ## their inverses
coefffield = []         ##list of coefficient for each polynomial of the syndrome used for
                                            finding valid codevectors in our goppa code
for i in range(fOrd):
    codeSupportPol.append(galois.Poly([1, i], field=GF))
    # print(codeSupportPol[i])
    codeSupportPolInv.append(galois.egcd(codeSupportPol[i], goppaPol)[1])
    # print(codeSupportPolInv[i])
    coefffield.append(codeSupportPolInv[i].coefficients())




coefffield = GF(coefffield)
# print("coefficients for inverse polynomials.")
# print(coefffield)
```

```
bb = 0b00
##help function used in generating every vector in our vectorspace
def bin2GF(bb):
    arr = np.zeros(fOrd, int)
    for i in range(len(arr)):
        if bb % 2 == 1:
            arr[fOrd-1 - i] = 1
        bb >>= 1
    return arr

def hammingWeight(arr):
    w = 0
    for i in arr:
        if i == 1:
            w += 1
    return w

goppaCodes = []  ## valid goppa codevectors
err_vec = [] ## valid error vectors

## this for loop fills our above vectors
zeroMat = np.zeros(gPolDeg, int)
# print("goppa codewords")
for i in range(0, 2**fOrd):##list all c vectors such that goppa condition satisfied
    arr = GF(bin2GF(bb))
    kk = np.matmul(coefffield.transpose(), arr)
    if (kk==zeroMat).all():
        # print(kk)
        print(arr)
        goppaCodes.append(arr)
    elif (hammingWeight(arr) == gPolDeg):
        err_vec.append(arr)

    bb += 1

# print(goppaCodes)
# print(err_vec)

##not really eea
def extended_euclidean_algorithm(g, tau, t):
    i = 0
    r = [g, tau]
    alpha = [g, tau]
    beta = [galois.Poly([0], GF), galois.Poly([1], GF)]

    while r[i].degree >= (t + 1) // 2:
        i += 1
        q, r_i = divmod(r[i-1], r[i-2])
        r.append(r_i)
        beta.append(beta[i-2] + q * beta[i-1])
        alpha.append(r[i])

    return alpha[i], beta[i]
```

```
def pattersons(notcodeword):
    syndrome = np.matmul(coefffield.transpose(), notcodeword)
    if (syndrome == zeroMat).all():
        return (notcodeword, 0)
    else:
        syndrome = galois.Poly(syndrome)
        T = galois.egcd(syndrome, goppaPol)[1]
        Tpx = T +galois.Poly([1,0], GF)
        tau = 0
        # tauS2 = 0
        # i = 0
        # while Tpx != tauS2: #fix this, the order thing online is wrong i = 32 not prima**(
                                                            orda-1) probably ford**gpoldeg instead
        #     tau = pow(Tpx, i, goppaPol)
        #     tauS2 = pow(tau, 2, goppaPol)#
        #     print( tauS2, i)
        #     i += 1

        for i in range(f0rd**gPolDeg):
            tau = pow(Tpx, i, goppaPol)
            if pow(tau, 2, goppaPol) == Tpx:
                #print(tau)
                break

        alpha, beta = extended_euclidean_algorithm(goppaPol, tau, gPolDeg)
        ##print("ap,bve", alpha,beta)
        alpha_squared = pow(alpha,2)
        beta_squared = pow(beta,2)
        x = galois.Poly([1, 0], GF)

        sigma =  (alpha_squared + x * beta_squared)


        leading_coeff = sigma.coeffs[0]
        c = 0
        for i in GF.elements:
            if leading_coeff* i == 1:
                c = i
                break
        #print(c)
        sigma =c*sigma

        # sigma, _ = divmod(sigma, galois.Poly([leading_coeff], GF))
        # print(sigma)
        # sigma = sigma  % goppaPol
        # print(sigma.roots())

        e = np.zeros(len(err_vec[0]), int)
        for i in range(len(err_vec[0])):
            if sigma.__call__(GF(i)) == 0:
                e[i] = 1

        e = GF(e)
        #print(e)
        m = notcodeword + e
        #print(m)
        return  m, e
```

```
# print(goppaCodes[3])
# print(err_vec[4])
# nc = goppaCodes[3]+err_vec[4]
# print(nc)


# pattersons(nc)

S=[[1,1],[0,1]]
P = [[0,0,0,0,1,0,0,0],[0,0,0,0,0,0,1,0],[0,0,0,0,0,0,0,1],[0,1,0,0,0,0,0,0], [0,0,0,1,0,0,0
                                                    ,0], [1,0,0,0,0,0,0,0],[0,0,0,0,0,1,0,0],[0,0,
                                                    1,0,0,0,0,0]]
G = [[0,0,0,1,1,1,1,1,],[1,1,0,1,0,1,0,1]]
g_pub= [[0,0,0,1,1,1,1,1,],[1,1,1,0,1,0,1,0]]
k = len(G)
G = GF(G)
S = GF(S)


## Converts a string into an binary array
def str_to_binary(text):
    # Convert the text to binary
    binary_text = ''.join(format(ord(char), '08b') for char in text)

    # Convert the binary string to an array of integers
    binary_array = [int(bit) for bit in binary_text]

    return np.array(binary_array)


def encrypt(g_pub, binary_plaintext, add_error):

    c_arr = []
    binary_plaintext = GF(binary_plaintext)
    g_pub = GF(g_pub)
    ## we multiply k bits to our public key at a time and add the encrypted bits to an array
                                                        .
    for i in range(0, len(binary_plaintext),k):
        c_arr.append((binary_plaintext[i:i+k]@ g_pub))

    randint = rand.randint(1,len(err_vec)-1) ## cant use a new one for every k bits
    ##now we add an error vector to the vectors ## maybe take random err vector every time.

    c_arr_err= [(i+add_error) for i in c_arr]
    return c_arr_err



def information_set(G):
    while True:
        k = len(G) ## verify this ##verified
        I = []
        ## Tries a random I until G_.I is invertible.  Meaning the determinant is nonzero.
        while True:
            I = rand.sample(population=range(1,f0rd),k=k)
```

```
            gSub =  G[:,I]
            if np.linalg.det(gSub) != 0:
                return I


def decrypt(G,S,P, c_arr_err):
    ## we multiply with the inverse of the permutation matrix
    c_arr_err = c_arr_err@np.linalg.inv(P)

    c_arr_err = GF(c_arr_err.astype(int))

    ##remove error vector
    decoded_c_arr =[]
    for i in range(len(c_arr_err)):
        plup,_ = pattersons(c_arr_err[i])
        decoded_c_arr.append(plup)

    ## we have to find an index set such that G_.I is invertible
    I  = information_set(G)

    ## now we revert our ciphertext
    GInv = GF(np.linalg.inv(G[:,I]).astype(np.int32))
    SInv = GF(np.linalg.inv(S).astype(np.int32))

    c_arr_rev =[i[I]@GInv@SInv for i in decoded_c_arr]
    return c_arr_rev



text = "Hello! My name is."
#### "Hello! My name is."
#### 01001000 01100101 01101100 01101100 01101111 00100001 00100000 01001101 01111001
                                                  00100000 01101110 01100001 01101101 01100101
                                                  00100000 01101001 01110011 00101110

binary_array = str_to_binary(text)

c_arr_err = encrypt(g_pub, binary_array, err_vec[3])


cArrRev = decrypt(G,S,P,c_arr_err)

## we compare to the original plaintext for verification
plainConcat = np.reshape(cArrRev,binary_array.shape)
#print("plain binary", plainConcat)



#print("asda",chr(72))



def bin_2_str(int_array):
    #convert int array to sets of binary values
    bin_int_array = []
    for i in range(0,len(plainConcat),8):
```

```
        bla =0
        for j in range(8):
            bla = bla +(2**(7-j))*plainConcat[i+j]
        bin_int_array.append(bla)

    chr_arr = []
    for i in bin_int_array:
        chr_arr.append(chr(i))

    plain_text = "".join(i for i in chr_arr)
    return plain_text

#print(bin_2_str(plainConcat))



## with this we have taken a plaintext written it in binary using the ascii/utf8 encoding
                                            into an array, taken k bits of the array at a
                                            time and encoded with the public key and later
                                             reverted it.
## TODO: add error vector and remove using the pattersons algorithm.
## Bug: will only work for congruent length of array modulo k. this is not an issue for our
                                            purposes, to keep code in line stick with
                                            ascii and have k be 2,4 or 8. can be fixed
                                            with padding.



#################GISD
## Generalized Information set decoding
## G is a random binary kxn matrix
## c =mG+e is a cipher hiding our plaintext
## integer j, j <= t ## t is in the public key

def GISD(G, c, j, t): ### this code is not completing, make sure it works.
    while True:
        k = len(G)
        c = GF(c)
        I = information_set(G)

        Q_1 =GF(np.linalg.inv(G[:,I]))
        Q_2 = GF(Q_1@G)
        z = c + (c[:,I]@Q_2)

        for i in range(j+1):
            for e in err_vec[:5]:
                if np.sum(np.take(e,I)) == i and np.sum(z + np.take(e,I)@Q_2) == t:
                    return (c[:,I]+np.take(e,I))@Q_1

#print("errorvector",err_vec[4])
#text = "Hello! My name is."
#### "Hello! My name is."
#### 01001000 01100101 01101100 01101100 01101111 00100001 00100000 01001101 01111001
                                            00100000 01101110 01100001 01101101 01100101
                                            00100000 01101001 01110011 00101110

##binary_array = str_to_binary(text)
```

```
##c_arr_err = encrypt(g_pub, binary_array)
##cArrRev = GISD(G,c_arr_err,1,2)

## we compare to the original plaintext for verification
#plainConcat = np.reshape(cArrRev,binary_array.shape)
#print(plainConcat == binary_array)
##TODO: random matrix generator and generalize reused code in functions,


#print(len(G[0]))
def stern(G, t, p, l):
    I = information_set(G)
    N = np.arange(len(G[0]))
    NsI = np.delete(N,I)
    K = np.arange(len(G))

    ## TODO: define P and G_r,c
    P = []
    KsP =  np.delete(K,P)
    for i in range(1,k):
        K = np.arange(len(G))



######## ind-cca2 for original mceliece

plain_1 = goppaCodes[3]
plain_2 = goppaCodes[2]


ciph = GF(encrypt(G, plain_1, err_vec[3]))


print("test parameters")
print("plain_1", plain_1)
print("plain_2", plain_2)
print("ciph", ciph)
print("error", err_vec[3])
print("G",G)
zero_err_vec = GF([0,0,0,0,0,0,0,0])

## testing against the indcca2 attackmodel
def ind_cca2(G,c, m1, m2):
    #print("encrypt m1 with 0 errors",GF(encrypt(G, m2, zero_err_vec)))
    m1 = encrypt(G, m1, zero_err_vec)
    m1 = GF(m1)
    m2 = encrypt(G, m2, zero_err_vec)
    m2 = GF(m2)
    su = [0,0,0,0]
    sa = [0,0,0,0]

    for i in range(len(c- m1)): ##np.sum not working as expected
        for j in (c- m1)[i]:
            if j == 1:
                su[i] = su[i] +  1
```

```
    for i in range(len(c- m2)): ##np.sum not working as expected
        for j in (c- m2)[i]:
            if j == 1:
                sa[i] = sa[i] +  1

    def check_something(sasa):
        for i in sasa:
            if i != gPolDeg:
                return False
        return True
    print(check_something(su), check_something(sa))
    print(su,sa)
    if check_something(su) == True:
        return plain_1

    if check_something(sa) == True:
        return plain_2


print(ind_cca2(G, ciph, plain_1, plain_2))
```

References

1. Lidl, R. & Niederreiter, H. *Finite Fields* 2nd ed. (Cambridge University Press, 1996).
2. Axler, S. in *Linear Algebra Done Right* 1–26 (Springer International Publishing, Cham, 2024). ISBN: 978-3-031-41026-0. https://doi.org/10.1007/978-3-031-41026-0_1.
3. Ling, S. & Xing, C. *Coding Theory: A First Course* https://doi.org/10.1017/CBO9780511755279 (Cambridge University Press, 2004).
4. Engelbert, D., Overbeck, R. & Schmidt, A. A Summary of McEliece-Type Cryptosystems and their Security. *Journal of Mathematical Cryptology* **1,** 151–199. https://doi.org/10.1515/JMC.2007.009 (2007).
5. McEliece, R. J. A public-key cryptosystem based on algebraic. *Coding Thv* **4244,** 114–116 (1978).
6. Bernstein, D. J., Chou, T. & Schwabe, P. *McBits: fast constant-time code-based cryptography* Cryptology ePrint Archive, Paper 2015/610. https://eprint.iacr.org/2015/610. 2015. https://eprint.iacr.org/2015/610.
7. https://classic.mceliece.org/mceliece-security-20221023.pdf. Accessed: 2024-4-29.
8. https://classic.mceliece.org/mceliece-spec-20221023.pdf. Accessed: 2024-4-29.
9. Dent, A. W. *A Designer's Guide to KEMs* in *Cryptography and Coding* (ed Paterson, K. G.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 2003), 133–151. ISBN: 978-3-540-40974-8.
10. Bernstein, D. J., Lange, T. & Peters, C. *Attacking and defending the McEliece cryptosystem* Cryptology ePrint Archive, Paper 2008/318. https://eprint.iacr.org/2008/318. 2008. https://eprint.iacr.org/2008/318.
11. Bernstein, D. J., Lange, T. & Peters, C. *Smaller decoding exponents: ball-collision decoding* Cryptology ePrint Archive, Paper 2010/585. https://eprint.iacr.org/2010/585. 2010. https://eprint.iacr.org/2010/585.
12. May, A., Meurer, A. & Thomae, E. *Decoding Random Linear Codes in $\tilde{\mathcal{O}}(2^{0.054n})$* in *Advances in Cryptology – ASIACRYPT 2011* (eds Lee, D. H. & Wang, X.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 2011), 107–124. ISBN: 978-3-642-25385-0.
13. Nielsen, M. A. & Chuang, I. L. *Quantum Computation and Quantum Information: 10th Anniversary Edition* (Cambridge University Press, 2010).
14. Wakasugi, A. & Tada, M. *Security analysis for BIKE, Classic McEliece and HQC against the quantum ISD algorithms* Cryptology ePrint Archive, Paper 2022/1771. https://eprint.iacr.org/2022/1771. 2022. https://eprint.iacr.org/2022/1771.
15. Bernstein, D. J. in *Post-Quantum Cryptography* 1–14 (Springer Berlin Heidelberg). ISBN: 9783540887027. http://dx.doi.org/10.1007/978-3-540-88702-7_1.
16. Overbeck, R. & Sendrier, N. in *Post-Quantum Cryptography* (eds Bernstein, D. J., Buchmann, J. & Dahmen, E.) 95–145 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2009). ISBN: 978-3-540-88702-7. https://doi.org/10.1007/978-3-540-88702-7_4.
17. https://classic.mceliece.org/mceliece-impl-20221023.pdf. Accessed: 2024-4-29.
18. https://classic.mceliece.org/mceliece-rationale-20221023.pdf. Accessed: 2024-4-29.

19. Becker, A., Joux, A., May, A. & Meurer, A. *Decoding Random Binary Linear Codes in 2n/20: How 1 + 1 = 0 Improves Information Set Decoding* in *Advances in Cryptology – EUROCRYPT 2012* (eds Pointcheval, D. & Johansson, T.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 2012), 520–536. ISBN: 978-3-642-29011-4.

20. Young, N. in *An Introduction to Hilbert Space* 4–12 (Cambridge University Press, 1988).

21. Kirshanova, E. *Improved Quantum Information Set Decoding* in *Post-Quantum Cryptography* (eds Lange, T. & Steinwandt, R.) (Springer International Publishing, Cham, 2018), 507–527. ISBN: 978-3-319-79063-3.

22. Kachigar, G. & Tillich, J.-P. *Quantum Information Set Decoding Algorithms* in *Post-Quantum Cryptography* (eds Lange, T. & Takagi, T.) (Springer International Publishing, Cham, 2017), 69–89. ISBN: 978-3-319-59879-6.