STUDIES ON IMPLEMENTATION OF SOME MRD CODES

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF APPLIED MATHEMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

RIDVAN ÖZKERİM

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
CRYPTOGRAPHY

SEPTEMBER 2021

Approval of the thesis:

STUDIES ON IMPLEMENTATION OF SOME MRD CODES

submitted by **RIDVAN ÖZKERİM** in partial fulfillment of the requirements for the degree of **Master of Science in Cryptography Department, Middle East Technical University** by,

Prof. Dr. A. Sevtap Selçuk Kestel
Director, Graduate School of **Applied Mathematics** ________________

Prof. Dr. Ferruh Özbudak
Head of Department, **Cryptography** ________________

Prof. Dr. Ferruh Özbudak
Supervisor, **Cryptography, METU** ________________

Examining Committee Members:

Assoc. Prof. Dr. Murat Cenk
Institute of Applied Mathematics, METU ________________

Prof. Dr. Ferruh Özbudak
Institute of Applied Mathematics, METU ________________

Assist. Prof. Dr. Eda Tekin
Department of Mathematics, Karabük University ________________

Date: ________________

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: RIDVAN ÖZKERİM

Signature :

ABSTRACT

STUDIES ON IMPLEMENTATION OF SOME MRD CODES

Özkerim, Rıdvan
M.S., Department of Cryptography
Supervisor : Prof. Dr. Ferruh Özbudak

September 2021, 56 pages

With the development of quantum computers that can process much faster than classical computers, the classical cryptosystems used today began to be strengthened or replaced with other cryptosystems. Parallel to this aim, studies for faster and more effective use of cryptosystems using coding theory have also increased. In this thesis, coding and decoding of maximum rank distance codes was implemented using programming.

Keywords: cryptography, coding theory, post-quantum, gabidulin, mrd, pari-gp

ÖZ

MAKSİMUM RANK UZAKLIKLI BAZI KODLARIN UYGULANMASI ÜZERİNE ÇALIŞMALAR

Özkerim, Rıdvan
Yüksek Lisans, Kriptografi Bölümü
Tez Yöneticisi : Prof. Dr. Ferruh Özbudak

Eylül 2021, 56 sayfa

Klasik bilgisayarlara göre çok daha hızlı işlem yapabilen kuantum bilgisayarların geliştirilmesi ile birlikte günümüzde kullanılan klasik kriptosistemler güçlendirilmeye veya başka kriptosistemler ile değiştirilmeye başlandı. Bu amaca paralel olarak kodlama teorisini kullanan kriptosistemlerin daha hızlı ve etkili kullanımı için yapılan çalışmalar da arttı. Bu tez kapsamında maksimum rank uzaklıklı kodların kodlanması ve kod çözümlemesi programlama kullanılarak uygulandı.

Anahtar Kelimeler: kriptografi, kodlama teorisi, kuantum sonrası, gabidulin, mrd, pari-gp

In memory of my father

ACKNOWLEDGMENTS

I would like to thank my supervisor, Prof. Dr. Ferruh Özbudak, for his patience and encouraging attitude. I would like to thank my teacher Assoc. Prof. Dr. Murat Cenk, who showed that theoretical mathematics can meet and apply programming. I would like to thank my dear wife, Assoc. Prof. Dr. Merve Güven Özkerim, for her unwavering support throughout the process. I would like to thank all the labourers who worked to reduce the negative effects of the pandemic at the time of writing this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

AGTG Code	Additive Generalized Twisted Gabidulin Code
FSF	Free Software Foundation
GF	Galois Field
GG Code	Generalized Gabidulin Code
GNU	GNU's Not Unix
GPL	General Public License
GPT	Gabidulin-Paramonov-Tretjakov
GTG Code	Generalized Twisted Gabidulin Code
IDE	Integrated Development Environment
MRD	Maximum Rank Distance
PKC	Public Key Cryptography
TG Code	Twisted Gabidulin Code

CHAPTER 1

INTRODUCTION

Cryptography has been used for basic needs such as communication, information security, locking the gates throughout the ages, and today in countless fields for these and similar needs. In order to respond to different needs, cryptography is also divided into different branches. The most basic distinction starts with whether the key to be used in encryption and decryption is the same or different. Systems in which the key used in encryption and decryption are the same are called symmetric cryptosystems, while systems in which they are different are called asymmetric cryptosystems. Since there is only one key in symmetric systems, this key must be securely shared and kept secret among the parties who want to decrypt it. On the other hand, in asymmetric systems, usually two keys are generated and one is kept secret while the other is shared openly. Although both public and private keys are used, these systems are called Public-Key Cryptosystem (briefly PKC). The concept of Public-Key Cryptosystem was first introduced in 1976 by Diffie and Hellman, with starting sentence "We stand today on the brink of a revolution in cryptography" of the article "New Directions in Cryptography". [11]

Coding Theory, which will cross paths with Public-Key Cryptosystem in the future, is based on an older past. Claude Shannon, who works on communication on Noisy channels, examined the error correcting capacities of encoding and decoding processes in different situations in his article titled "A Mathematical Theory of Communication" in 1948.[25] In 1949, Marcel Golay extended Shannon's work on blocks of seven symbols and applied it to blocks of $2^n - 1$ binary symbols.[7] In 1950, Richard Wesley Hamming published "Error Detecting and Error Correcting Codes", the second work in this field after Golay.[9] In 1960, Irving Stoy Reed and Gustave Solomon published their work, which will be named after themselves, as "Polynomial Codes over Certain Finite Fields".[22] In 1970, Valery Denisovich Goppa published his work, later known as "Binary Goppa Codes", in his article titled "A new class of linear correcting codes".[8] In 1978, Robert J. McEliece created a new Public-Key Cryptosystem using Goppa codes.[16] Thus, the first Public-Key Cryptosystem based on algebraic coding theory was formed.

In 1985, Ernst M. Gabidulin developed the rank metric instead of the Hamming metric used in most studies in the field.[3] In 1991, Ernst M. Gabidulin, A.V. Paramonov and O.V. Tretjakov presented rank metric instead of Hamming metric in McEliece Public-Key Cryptosystem in their article titled "Ideals over a Non-Commutative Ring and their Application in

Cryptology".[4] They used Maximum Rank Distance code instead of Goppa code. This application was later called GPT cryptosystem in the literature.

Cryptosystems applied in fields such as telegraph, telephone, radio, and computers faced the danger of collapse when the integer factorization algorithm, which was developed theoretically by Peter Shor in 1994[27], was run on a 7-qubit quantum computer by Isaac Chuang and Neil Gershenfeld in 2001. Thereupon, algorithms and cryptosystems in many fields were tried to be re-evaluated and strengthened by considering the existence of quantum computers and their potential computational power. Studies in this new world can be evaluated under the title of "Post-Quantum Cryptography". Due to technological developments and increasing need, studies in this field have accelerated.

In 2005, Alexander Kshevetskiy and Gabidilun presented the Generalized Gabidilun code in their article titled "The new construction of rank codes".[11] In 2013, Nina Pilipchuk, together with Ernst M. Gabidulin, published the article "GPT Cryptosystem for information network security".[5] In 2015, John Sheekey obtained Twisted Gabidilun Codes in his article titled "A new family of linear maximum rank distance codes".[26] In 2017, Kamil Otal and Ferruh Özbudak presented additive generalized twisted Gabidulin codes (or briefly AGTG codes) in their article "Additive rank metric codes".[19] In 2019, Chunlei Li and Wrya K. Kadir presented a new interpolation decoding algorithm approach to improve the decoding of MRD codes in their study titled "On decoding additive generalized twisted Gabidulin codes".[12] Within the scope of this thesis, the application of the decoding algorithm presented by Chunlei Li and Wrya K. Kadir will be implemented with the Pari/GP programming language and examined.

CHAPTER 2

PRELIMINARY

In this section we will give some definitions and theorems as a background of this thesis work.

2.1 Metrics

Metrics are used for measure some properties of mathematichal elements.

2.1.1 Hamming Distance

Hamming defines a distance $D(x, y)$ between two points as a metric. The definition of the metric is there exists d different cooridinates between two points. This distance function satisfies the usual three conditions for a metric, namely, [9]

$$D(x,y) = 0 \text{ if and only if } x \neq y$$
$$D(x,y) = D(y,x) > 0 \text{ if } x \neq y$$
$$D(x,y) + D(y,z) \geq D(x,z) \text{ triangle inequality}$$

as an example,

$$x = (0,1,1,0)$$
$$y = (0,1,0,0)$$
$$d = D(x,y) = 1 \text{ because there is only one different coordinate}$$

2.1.2 Rank Metric

Gabidulin defines rank as count of independent coordinates in a vector.

Let $\alpha = (\alpha_1, \alpha_2, ..., \alpha_n)$ be a vector with coordinates in the extension field F_q^N .

$Rk(\alpha|F_q)$ denotes the *Rank* norm of α which means the maximal number of α_i, which are linearly independent over the *base* field F_q .

$\alpha - \beta : d(\alpha, \beta) = Rk_{col}(\alpha - \beta|F_q)$ denotes the *Rank distance* between α and β over the *base* field F_q .

Similarly for a matrix $M \in F_q^N$, the column rank is defined as the maximal number of columns, which are linearly independent over the field F_q , and is denoted $Rk_{col}(M|F_q)$

Any linear (n, k, d) code $\mathcal{C} \subset F_{q^N}^n$ fulfils the Singleton-style bound for the rank distance:

$$Nk \leq Nn - (d-1)max\{N, n\}$$

A code $\mathcal{C}$ reaching that bound is called a MRD (Maximal Rank Distance) code.

Gabidulin gives the theory of optimal MRD codes in 1985 [3] .

The notation $g[i] := g^{q^{imodN}}$ means the *i*-th Frobenius power of g. It allows to consider both positive and negative Frobenius powers *i*.

For $n \leq N$, a generator matrix G_k of a (n, k, d) MRD code is defined by a matrix of the following form:

$$G_k = \begin{bmatrix} g_1 & g_2 & \cdots & g_n \\ g_1^{[1]} & g_2^{[1]} & \cdots & g_n^{[1]} \\ g_1^{[2]} & g_2^{[2]} & \cdots & g_n^{[2]} \\ \vdots & \vdots & \ddots & \vdots \\ g_1^{[k-1]} & g_2^{[k-1]} & \cdots & g_n^{[k-1]} \end{bmatrix}$$

where $g_1, g_2, \ldots, g_n$ are a set of elements of the extension field F_q^N which are linearly independent over the base field F_q. A code with the generator matrix G_k is referred to as (n, k, d) code, where n is code length, k is the number of information symbols, d is code distance. For MRD codes, $d = n - k + l$. Let $m = (m_1, m_2, \ldots, m_k)$ be an information vector over the extension field F_q^N of dimension k. The corresponding code vector is the n-vector

$$g(m) = mG_k$$

If $y = g(m) + e$ and $Rk(e) = s \leq t = \frac{d-1}{2}$, then the information vector m can be recovered uniquely from y by some decoding algorithm. There exist fast decoding algorithms for MRD codes (for instance, [[3], [4]]).

The rank of a vector $v = (v_0, v_1, \dots, v_{n-1}) over F_{q^n}$ is defined as the dimension of $span_{F_q}\langle v_0, v_1, \dots, v_{n-1}\rangle$ which is the vector space spanned by v_i's over F_q. [10]

[21] [5]

2.2 Error Detecting and Error Correcting Codes

Error detecting codes may be a single code or a sequence of code to check if original codeword changed on the noisy channel.

Error correcting codes are used for correcting errorenous codeword.

Hamming codes, Reed & Solomon codes, Goppa codes are most popular former examples.

Cryptographic hash functions are widely used for error detecting.

Gabidulin codes was introduced by Gabudilin.

2.3 Cryptosystems

2.3.1 McEliece Cyptosystem

McEliece cryptosystem introduced by Robert J. McEliece in 1978. The cryptosystem is based on difficulty of finding the nearest codeword for a linear binary code [30] . It ensures efficient encryption and decryption procedures and a good practical and theoretical security. However it has a public key of large size and its ciphertext is larger than plaintext [24] .

Parameters:

k : length of binary form of plaintext

t : error threshold. maximum count of erroneous bits on codeword

n : parameter, $k \leq n - tlog_2 n$

Γ: a family of binary irreducable t-error correcting Goppa codes of length n and dimension k.

Key Generation:

C : a randomly and uniformly chosen code in the family Γ

G_0 : generator matrix of C

S : a random kxk non-singular binary matrix

P : a random nxn permutation matrix

$G = SG_0P$ is the public key

Encryption:

x : the plaintext, where $x \in F_2^k$

e : randomly chosen error matrix with a Hamming weight t, where $e \in F_2^n$.

$c = xG + e, \in F_2^n$: is the ciphertext

Decryption:

D : decoding algorithm

$c = xG + e$

$c = x(SG_0P) + e$

$cP^{-1} = (xS)G_0 + eP^{-1}$, eP^{-1} has weight t

$c' = D(cP^{-1}) = xS$

$x = c'S^{-1}$

2.3.2 GPT Cyptosystem

GPT cryptosystem is proposed [4] as another version of McEliece's PKC based on *rank* error correcting codes. The GPT cryptosystem has smaller key size and more strength againist decoding attacks [5].

Parameters:

k : length of binary form of plaintext

t : error threshold. maximum count of erroneous bits on codeword

n : parameter, $k \leq n - tlog_2n$

Γ: a family of binary irreducable t-error correcting Goppa codes of length n and dimension k.

Key Generation:

α : a non-zero k-tuple over $GF(q^n)$

e_g : a non-zero k-tuple over $GF(q^n)$

G_0 : a chosen kxn generator matrix of a MRD code for Γ

S : a random kxk non-singular binary matrix

P : a random nxn permutation matrix

$G = SG_0 + \alpha^T e_g$ is the public key

Encryption:

x : the plaintext, where $x \in F_2^k$

e_e : randomly chosen pattern of $t_e = t - t_g$

$c = xG + e_e, \in F_2^n$: is the ciphertext

Decryption:

D : Decoding algorithm

$c = xG + e$

$c' = D(c) = xS$

$x = c'S^{-1}$

2.4 Primitive Polynomial

If a polynomial $g \in F_q^n$ and g is the minimal polynomial over F_q, then g is called a primitive element over F_q^n.

Therefore, a primitive polynomial over F_q^n can be described as a monic polynomial that is irreducable over F_q and has a root $r \in F_q^n$ that generates the multiplicative group of F_q^n. [13]

2.5 Linearized Polynomials

Linearized polynomials were firstly studied by Ore [18] A polynomial of the form

$$L(x) = \Sigma_{i=0}^{k-1} \alpha_i x^{q^i} \text{over} F_{q^n}$$

is known as a q-polynomial. [31]

2.6 Moore Matrix

Moore matrix is a matrix form where Eliakim Hastings Moore used for studies on generalization of Fermat's theorem [17]. Let M be an m x n matrix.

$$M_{i,j} = \alpha_i^{q^{j-1}} \text{ where } 0 < i \leq n \text{ and } 0 < j \leq m$$

Let M be an n x n square matrix.

$$M_{i,j} = \alpha_i^{q^{j-1}} \text{ where } 0 < i{,}j \leq n$$

$$M_{nxn} = \begin{bmatrix} \alpha_1 & \alpha_1^q & \dots & \alpha_1^{q^{n-1}} \\ \alpha_2 & \alpha_2^q & \dots & \alpha_2^{q^{n-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_n & \alpha_n^q & \dots & \alpha_n^{q^{n-1}} \end{bmatrix}$$

2.7 Dickson Matrix

$D_n(F_q^n)$ is an algebra formed by all n x n matrices over F_q^n of the form

$$D_{i,j} = [\alpha_{i\text{-}j(\text{mod} n)}^{q^j}]_{n\text{x}n} \text{ where } 0 < i{,}j \leq n$$

$$D = \begin{bmatrix} \alpha_0 & \alpha_{n-1}^q & \dots & \alpha_1^{q^{n-1}} \\ \alpha_1 & \alpha_0^q & \dots & \alpha_2^{q^{n-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{n-1} & \alpha_{n-2}^q & \dots & \alpha_0^{q^{n-1}} \end{bmatrix}$$

which are called Dickson matrices. [31]

Let $(x) \in F_{q^n}$ be a linearized polynomial, where $Rk(d) = t$ and D be the associated Dickson matrix for (x). Then we have the following properties:

- $Rk(D) = Rk(d) = t$

- Any r successive columns $D_i, \ldots, D_{i+t}$ are linearly independent and the other columns can be generated by using linear combinations of them.
- All txt submatrices $D_{(i \bmod n),(j \bmod n)}$ are invertible.

2.8 Twisted Gabidulin Codes

Sheekey [26] made a breakthrough in the construction of new linear MRD codes using linearized polynomials.

Let $n, k, h \in Z^+$ and $k < n$. Let η be in F_{q^n} such that $N_{q^n/q}(\eta) \neq (-1)^{nk}$, where $N_{q^n/q}(\eta) = \eta^{1+q+\cdots+q^{n-1}}$. Then the set

$$\mathcal{H}_k(\eta, h) = \{a_0 x + a_1 x^q + \cdots + a_{k-1} x^{q^{k-1}} + \eta a_0^{q^h} x^{q^k} : a_0, a_1, \ldots, a_{k-1} \in F_{q^n}\}$$

is an F_q-linear MRD code of size q^{nk}, which is called a *twisted Gabidulin code* [14].

2.9 Generalized Twisted Gabidulin Codes

Let $n, k, s, h \in Z^+$ satisfying $gcd(n, s) = 1$ and $k < n$. Let η be in F_{q^n} such that $N_{q^s n/q^s}(\eta) \neq (-1)^{nk}$. Then the set

$$\mathcal{H}_{k,s}(\eta, h) = \{a_0 x + a_1 x^{q^s} + \cdots + a_{k-1} x^{q^{s(k-1)}} + \eta a_0^{q^h} x^{q^{sk}} : a_0, a_1, \ldots, a_{k-1} \in F_{q^n}\}$$

is an F_q-linear MRD code of size q^{nk}, and we call them *generalized twisted Gabidulin code* [14].

2.10 Additive Generalized Twisted Gabidulin Codes

Let $n, k, s, u, h \in Z^+$ satisfying $gcd(n, s) = 1$, $q = q_0^u$ and $k < n$. Let η be in F_{q^n} such that $N_{q^s n/q_0^s}(\eta) \neq (-1)^{nku}$. Then the set

$$\mathcal{A}_{k,s,q_0}(\eta, h) = \{a_0 x + a_1 x^{q^s} + \cdots + a_{k-1} x^{q^{s(k-1)}} + \eta a_0^{q_0^h} x^{q^{sk}} : a_0, a_1, \ldots, a_{k-1} \in F_{q^n}\}$$

is an F_q-linear (but does not have to be linear) MRD code of size q^{nk} and distance $n - k + 1$.

We call the obtained this family as *additive generalized twisted Gabidulin codes*, or briefly AGTG codes.

The conditions about the parameters can be summarized as follows:

- When u divides h, an AGTG code is a GTG code.
- When u divides h and $s = 1$, an AGTG code is a TG code.
- When u divides h and $\eta = 0$, an AGTG code is a GG code.
- When u divides h, $s = 1$ and $\eta = 0$, an AGTG code is a Gabidulin code.

[19]

CHAPTER 3

ENCODING AND DECODING OF AGTG CODES

In this chapter we will give a brief way of encoding and decoding of AGTG Codes as explained in [10].

3.1 Encoding

To construct AGTG codes and system variables, we will use the definition and the same variable names from 2.10. We will work over $GF(q^n)$, encode message with length k, by using error matrix with rank t. We will use linearly independent evaluation points and choose a message.

Example 3.1. *Let the parameters with values be* $n = 7$*,* $k = 3$*,* $t = 2$*,* $q_0 = 1, s = 1$*,* $h = 1$*,* $u = 1$

$t \leq \lfloor \frac{n-k}{2} \rfloor$ *and* $gcd(n, s) = 1$*,*

then we can say our parameters are valid.

Calculate $q = q_0^u = 3^1 = 3$

Then choose a primitive polynomial over $GF(q^n) = GF(3^7)$ *as a generator*

$w^7 + w^2 + 2w + 1$

and choose a message over $GF(3^7)$ *with* k *elements.*

$[(1012200)_3, (2010201)_3, (1110120)_3] \rightarrow f = [f_0, f_1, f_2] = [w^6 + w^4 + 2w^3 + 2w^2, 2w^6 + w^4 + 2w^2 + 1, w^6 + w^5 + w^4 + w^2 + 2w]$

Let η *be* w*, where* $N_{\frac{q^n}{q_0}}(\eta) \neq (-1)^{nku}$

$(\eta)^{\frac{q^n-1}{q_0-1}} = (w)^{\frac{3^7-1}{3-1}} = 2 \neq (-1)^{nku} = -1$

α_i*'s denoted the linearly independent evaluation points over* F_q*, where* $0 \leq i < n$*.*

Let,

$$
\begin{aligned}
\alpha_0 &= 2w^6 + 2w^4 + w^3 + 2w^2 + 2w + 1 \\
\alpha_1 &= w^6 + w^4 + 2w^3 + w^2 + w + 1 \\
\alpha_2 &= 2w^6 + 2w^5 + 2w^4 + 2w^2 + w + 1 \\
\alpha_3 &= 2w^5 + w^4 + w^3 + w^2 + w + 1 \\
\alpha_4 &= w^4 + 2w^2 + w + 2 \\
\alpha_5 &= w^5 + 2w^3 + 2w^2 + w + 1 \\
\alpha_6 &= 2w^6 + w^2 + w + 1
\end{aligned}
$$

3.1.1 Evaluation of the linearized polynomial

Let message be $f = (f_0, \dots, f_{k-1})$ over $GF(q^n)$.

$\alpha_1, \dots, \alpha_n$ in $GF(q^n)$, are linearly independent evaluation points over $GF(q)$.

Encoding of AGTG codes $\{f \to c\}$ can be expressed by directly evaluation of the associated linearized polynomial to f

$$f(\alpha_i) = \eta f_0^{q_0^h} \alpha_k^{q^{ks}} + \Sigma_{j=0}^{k-1}(f_j \alpha_i^{q^{js}}) = \eta f_0^{q_0^h} \alpha_k^{[k]} + \Sigma_{j=0}^{k-1}(f_j \alpha_i^{[j]})$$

$$c = (f(\alpha_1), f(\alpha_2), \dots, f(\alpha_n))$$

It can be reprsented as production of vector f and associated matrix of α .

$$c = (f_0, f_1, \dots, f_{k-1}, \eta f_0^{q_0^h}) \begin{bmatrix} \alpha_0^{[0]} & \alpha_1^{[0]} & \dots & \alpha_{n-1}^{[0]} \\ \alpha_0^{[1]} & \alpha_1^{[1]} & \dots & \alpha_{n-1}^{[1]} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_0^{[k]} & \alpha_1^{[k]} & \dots & \alpha_{n-1}^{[k]} \end{bmatrix}_{(k+1)xn}$$

If we add $n-k-1$ times zero at the end of the message f, and if we add unused powers of alpha values at the end of the matrix, then we get an nxn square matrix, which is full Moore matrix transposition of vector α .

$$c = (f_0, f_1, \ldots, f_{k-1}, \eta f_0^{q_0^h}, \underbrace{\mathbf{0}, \ldots, \mathbf{0}}_{n-k \text{ times}}) \begin{bmatrix} \alpha_0^{[0]} & \alpha_1^{[0]} & \ldots & \alpha_{n-1}^{[0]} \\ \alpha_0^{[1]} & \alpha_1^{[1]} & \ldots & \alpha_{n-1}^{[1]} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_0^{[k]} & \alpha_1^{[k]} & \ldots & \alpha_{n-1}^{[k]} \\ \boldsymbol{\alpha_0^{[k+1]}} & \boldsymbol{\alpha_1^{[k+1]}} & \boldsymbol{\ldots} & \boldsymbol{\alpha_{n-1}^{[k+1]}} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{\alpha_0^{[n-1]}} & \boldsymbol{\alpha_1^{[n-1]}} & \boldsymbol{\ldots} & \boldsymbol{\alpha_{n-1}^{[n-1]}} \end{bmatrix}_{\boldsymbol{(nxn)}}$$

If we define $\tilde{f}$ as concatation of coefficients of $f(x)$ and $n-k-1$ times zeros, then we can write:

$$\mathcal{M} = [\alpha_{i+1}^{js}]_{nxn} \text{is the Moore matrix.}$$
$$c = \tilde{f}\mathcal{M}^T$$

Example 3.2. *Let us continue with values from [Example 3.1].*

$f_k = \eta f_0^{q_0^h} = w(w^6 + w^4 + 2w^3 + 2w^2)^3 = 2w^5 + w^4 + 2w^3 + 2w^2$

Let Moore matrix be $M = [M_0 M_1 M_2 M_3 \ldots]$ where M_i's are column matrices.

$$M_0 = \begin{bmatrix} 2w^6 + 2w^4 + w^3 + 2w^2 + 2w + 1 \\ w^6 + w^4 + 2w^3 + w^2 + w + 1 \\ 2w^6 + 2w^5 + 2w^4 + 2w^2 + w + 1 \\ 2w^5 + w^4 + w^3 + w^2 + w + 1 \\ \vdots \end{bmatrix} \qquad M_1 = \begin{bmatrix} w^6 + w^4 + w^3 + w^2 + w + 1 \\ 2w^6 + 2w^4 + 2w^3 + 2w^2 + 2w + 1 \\ w^6 + 2w^5 + w^4 + 2w^3 + w^2 + 1 \\ 2w^6 + w^5 + w^4 + 2w^3 + 2w^2 + w + 2 \\ \vdots \end{bmatrix}$$

$$M_2 = \begin{bmatrix} 2w^6 + w^3 + 2w + 1 \\ w^6 + 2w^3 + w + 1 \\ 2w^6 + 2w^5 + w^4 + w^3 + w^2 + w + 1 \\ 2w^5 + w^4 + w^3 + 1 \\ \vdots \end{bmatrix} \qquad M_3 = \begin{bmatrix} 2w^5 + w^4 + w^3 + 2w^2 + 2 \\ w^5 + 2w^4 + 2w^3 + w^2 \\ 2w^6 + 2w^2 + w \\ w^6 + w^5 + w^4 + w^3 + 2w^2 + w + 2 \\ \vdots \end{bmatrix}$$

Its transpositon matrix is $M^T = [M_0^T M_1^T M_2^T M_3^T \ldots]$ where M_i^T's are column matrices.

$$M_0^T = \begin{bmatrix} 2w^6 + 2w^4 + w^3 + 2w^2 + 2w + 1 \\ w^6 + w^4 + w^3 + w^2 + w + 1 \\ 2w^6 + w^3 + 2w + 1 \\ 2w^5 + w^4 + w^3 + 2w^2 + 2 \\ \vdots \end{bmatrix} \qquad M_1^T = \begin{bmatrix} w^6 + w^4 + 2w^3 + w^2 + w + 1 \\ 2w^6 + 2w^4 + 2w^3 + 2w^2 + 2w + 1 \\ w^6 + 2w^3 + w + 1 \\ w^5 + 2w^4 + 2w^3 + w^2 \\ \vdots \end{bmatrix}$$

$$M_2^T = \begin{bmatrix} 2w^6 + 2w^5 + 2w^4 + 2w^2 + w + 1 \\ w^6 + 2w^5 + w^4 + 2w^3 + w^2 + 1 \\ 2w^6 + 2w^5 + w^4 + w^3 + w^2 + w + 1 \\ 2w^6 + 2w^2 + w \\ \vdots \end{bmatrix} \qquad M_3^T = \begin{bmatrix} 2w^5 + w^4 + w^3 + w^2 + w + 1 \\ 2w^6 + w^5 + w^4 + 2w^3 + 2w^2 + w + 2 \\ 2w^5 + w^4 + w^3 + 1 \\ w^6 + w^5 + w^4 + w^3 + 2w^2 + w + 2 \\ \vdots \end{bmatrix}$$

and c is a row matrix with length n. It is shown as transposed because of printing issues.

$$c^T = (\tilde{f}M^T)^T = \begin{bmatrix} w^6 + 2w^5 + 2w^3 + 2w^2 + 2w + 2 \\ 2w^6 + 2w^5 + w^4 + 2w^2 + 2w \\ 2w^5 + w^4 + 2w^3 + 2w^2 + 1 \\ w^5 + w^3 + 2w + 1 \\ w^6 + 2w^5 + 2w^3 + 2w^2 + w \\ 2w^6 + w^5 + 2w^4 + 2w^3 + w^2 + 2w + 1 \\ w^5 + 2w^3 + w^2 + 2 \end{bmatrix}$$

3.2 Transmission

In the real world, original data sent by sender can be corrupted during transmission. The receiver should can be recover the original codeword from the received word with acceptable amount of error. That is one of the main motivations of coding theory.

While defining sytem parameters we set a t value. We are able to recover corrupted codewords up to rank t, which is the threshold value.

This stage is the stage where the c codeword we obtained during the encoding stage is corrupted.

3.2.1 Construction of the error vector

The error vector e with rank t is constructed randomly. That means the rank distance between the vector c and the new vector $c + e$ is less than or equal to t.

Example 3.3. *Let us continue with values from [Example 3.2]. Let* $e = (0, \alpha_1, \alpha_2, 0, 0, 0, 0)$ *is the error vector with rank* $t = 2$.

$$e = (0, w^6 + w^4 + 2w^3 + w^2 + w + 1, 2w^6 + 2w^5 + 2w^4 + 2w^2 + w + 1, 0, 0, 0, 0)$$

3.2.2 Adding Error Vector to Encoded Message

We have found encoding of AGTG codes c in 3.1.1 and generated a random error vector e in **??**. We can complete the encoding operation by adding error to encoded message.

$$r = c + e$$

Example 3.4. *From [Example 3.1] and [Example 3.3].*

$$r^T = (c+e)^T = \begin{bmatrix} w^6 + 2w^5 + 2w^3 + 2w^2 + 2w + 2 \\ 2w^5 + 2w^4 + 2w^3 + 1 \\ 2w^6 + w^5 + 2w^3 + w^2 + w + 2 \\ w^5 + w^3 + 2w + 1 \\ w^6 + 2w^5 + 2w^3 + 2w^2 + w \\ 2w^6 + w^5 + 2w^4 + 2w^3 + w^2 + 2w + 1 \\ w^5 + 2w^3 + w^2 + 2 \end{bmatrix}$$

is the result of encoding operation. This value will be used for decoding as "received word".

3.3 Decoding

In 1968 Berlekamp introduced an efficient technique to decode Reed-Solomon codes [2]. In 1969 Massey interpreted this algorithm as a problem of synthesising the shortest linear feedback shift-register capable of generating a prescribed finite sequence of digits [15]. In 2004 Richter and Plass applied Berlekamp-Massey algorithm to rank codes [23].

In 2017 Randrianarisoa modified the Richter-Plass algorithm and used it to decode Gabidulin codes [20]. In 2019, Li and Kadir re-modified Randrianarisoa's modified Berlekamp-Massey algorithm and used to decode Additive Generalized Twisted Gabidulin codes [12]. Li and Kadir propose an interpolation based decoding algorithm to decode AGTG codes. In this section we will describe how it works.

3.3.1 Reducing the decoding problem

We know from transmission section [3.2.2],

$$r = c + e$$

and from encoding [3.1.1]

$$c = \tilde{f}\mathcal{M}^T$$

Then we can write,

$$r = \tilde{f}\mathcal{M}^T + e$$

Because $\mathcal{M}^T$ is invertible, so we can assume that there is a vector $g = (g_0, g_1, \ldots, g_{n-1}) \in F_{q^n}^n$ which generates the vector e by product with $\mathcal{M}^T$.

$$\begin{aligned} e &= g\mathcal{M}^T \\ r &= \tilde{f}\mathcal{M}^T + g\mathcal{M}^T \end{aligned}$$

Due to the linearity we can say that,

$$r = (\tilde{f} + g)\mathcal{M}^T$$

where values of $\tilde{f}$ and g are unknown.

Suppose there is a vector $\gamma = (\gamma_0, \gamma_1, \ldots, \gamma_{n-1})$ that generates vector r. We can calculate,

$$\gamma = r(\mathcal{M}^T)^{-1}$$

Then we can say that,

$$\gamma = \tilde{f} + g$$

where value of γ is known but values of $\tilde{f}$ and g are unknown. So we can reduce decoding problem to finding vector g. If we find g, then we can calculate $\tilde{f}$ and hence the original message vector f.

3.3.2 Advantage of the Dickson Matrix Property

We know the last $(n-k-1)$ elements of $\tilde{f}$ are equal to zero by definition. So we can say the last $(n-k-1)$ elements of g are equal to the last $(n-k-1)$ elements of γ

$$\begin{array}{ccccccccc} & & f_0, & \ldots, & f_{k-1}, & \eta f_0^{q_0^h}, & \mathbf{0}, & \ldots, & \mathbf{0} \\ & + & g_0, & \ldots, & g_{k-1}, & g_k, & g_{k+1}, & \ldots, & g_{n-1} \\ \hline \gamma & = & (f_0 + g_0), & \ldots, & (f_{k-1} + g_{k-1}), & (\eta f_0^{q_0^h} + g_k), & (\mathbf{g_{k+1}}), & \ldots, & (\mathbf{g_{n-1}}) \end{array}$$

We will use the Dickson matrix and its features to decoding the received codeword r. When we generate dickson matrix of the vector g, we get a matrix G

$$G_{i,j} = [g_{i\text{-}j(\mathrm{mod} n)}^{q^j}]_{nxn} = \begin{bmatrix} g_0^{[0]} & g_{n-1}^{[1]} & \cdots & g_1^{[n-1]} \\ g_1^{[0]} & g_0^{[1]} & \cdots & g_2^{[n-1]} \\ \vdots & \vdots & \ddots & \vdots \\ g_{n-1}^{[0]} & g_{n-2}^{[1]} & \cdots & g_0^{[n-1]} \end{bmatrix}$$

We know that the maximum rank of the error vector e and the associated vector g is t. We know or can easily calculate all the elements of the $(n-k-2)x(t+1)$ submatrix in the lower left corner of the matrix G.

$$G = \left[\begin{array}{cccc|cc} g_0^{[0]} & g_{n-1}^{[1]} & \cdots & g_{n-t}^{[t]} & \cdots & g_1^{[n-1]} \\ g_1^{[0]} & g_0^{[1]} & \cdots & g_{n-t+1}^{[t]} & \cdots & g_2^{[n-1]} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ g_{k+t}^{[0]} & g_{k-1+t}^{[1]} & \cdots & g_k^{[t]} & \cdots & g_{k+1+t}^{[n-1]} \\ \mathbf{g_{k+1+t}^{[0]}} & \mathbf{g_{k+t}^{[1]}} & \cdots & \mathbf{g_{k+1}^{[t]}} & \cdots & g_{k+2+t}^{[n-1]} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \mathbf{g_{n-1}^{[0]}} & \mathbf{g_{n-2}^{[1]}} & \cdots & \mathbf{g_{n-1-t}^{[t]}} & \cdots & g_0^{[n-1]} \end{array}\right]$$

If we take $t+1$ consequtive columns, then we can write the first column as a linear combination of the other columns because of Dickson matrix property from [Section 2.7].

$$G = \left[\begin{array}{cccc|cc} G_0 & G_1 & \ldots & G_t & \ldots & G_{n-1} \end{array}\right]$$

$$\Lambda_0 G_0 = \Lambda_1 G_1 + \cdots + \Lambda_t G_t, \text{ where } \Lambda_0 \triangleq 1$$

We take, $\Lambda_0 \triangleq 1$ by definition. So we need to find other Λ_i values. We have t unknown Λ_i values and $(n-k-1)$ equations.

$$\left.\begin{aligned} g_{k+1+t} &= \Lambda_1 g_{k+t}^{[1]} + \cdots + \Lambda_t g_{k+1}^{[t]} \\ g_{k+2+t} &= \Lambda_1 g_{k+1+t}^{[1]} + \cdots + \Lambda_t g_{k+2}^{[t]} \\ &\vdots \\ g_{n-1} &= \underbrace{\Lambda_1 g_{n-2}^{[1]} + \cdots + \Lambda_t g_{n-1-t}^{[t]}}_{t \text{ unkown } \Lambda \text{ values}} \end{aligned}\right\} \textit{n-k-t-1 equations}$$

$t \leq \lfloor \frac{n-k}{2} \rfloor$ by definition. So $(2t+k) \leq n$. There are two cases in this situation.

Case 1: $(2t+k) < n$ The first case is inequality.

$$n - k - t - 1 \geq t$$

In this case we have more equations than unknown variables count. It means that we have a unique solution for Λ_i values and we can found Λ_i values by using modified Berlekamp Massey algorithm as usual.

Case 2: $(2t+k) = n$

$$n - k - t - 1 = t - 1 < t$$

In the second case, more equations are required to find a unique solution for Λ_i values. We can take two more equations (above and below rows of the previous submatrix), but the values of g_0 and g_k are unknown yet. However we know a relation between them from the last element of $f(x)$. We will use it later.

$$G = \left[\begin{array}{cccc|cc}
g_0^{[0]} & \mathbf{g_{n-1}^{[1]}} & \cdots & \mathbf{g_{n-t}^{[t]}} & \cdots & g_1^{[n-1]} \\
g_1^{[0]} & g_0^{[1]} & \cdots & g_{n-t+1}^{[t]} & \cdots & g_2^{[n-1]} \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
 & & & & & \\
\mathbf{g_{k+t}^{[0]}} & \mathbf{g_{k-1+t}^{[1]}} & \cdots & g_k^{[t]} & \cdots & g_{k+1+t}^{[n-1]} \\
\mathbf{g_{k+1+t}^{[0]}} & \mathbf{g_{k+t}^{[1]}} & \cdots & \mathbf{g_{k+1}^{[t]}} & \cdots & g_{k+2+t}^{[n-1]} \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
\mathbf{g_{n-1}^{[0]}} & \mathbf{g_{n-2}^{[1]}} & \cdots & \mathbf{g_{n-1-t}^{[t]}} & \cdots & g_0^{[n-1]}
\end{array}\right]$$

3.3.3 Applying the modified Berlekamp-Massey Algorithm

For the second case, we replace,

$$\Lambda_i = (\lambda_i + y\lambda_i')$$

Then we have,

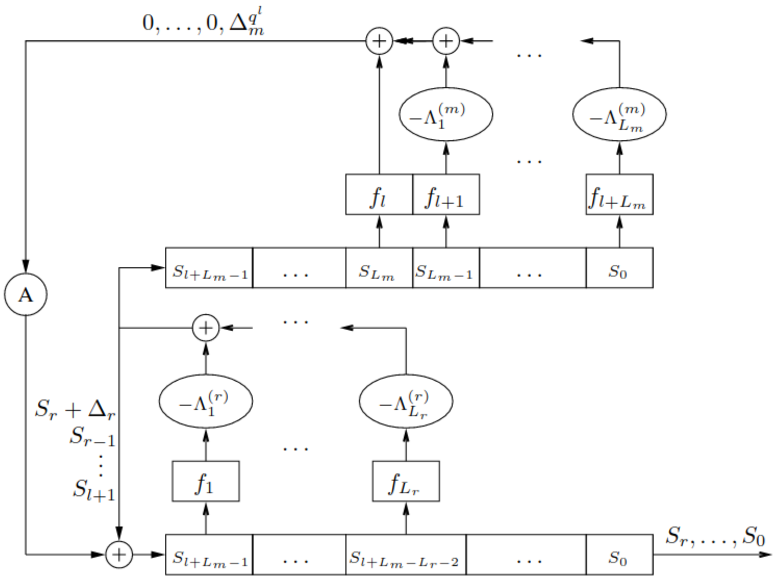

Figure 3.1: Visual representation of BM algorithm

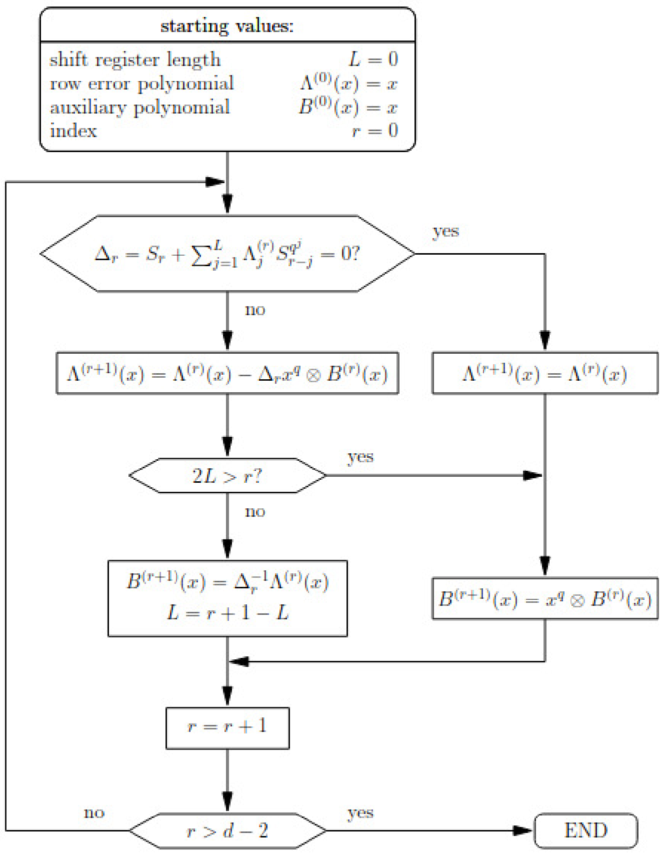

Figure 3.2: Flowchart for BM algorithm

$$G_0 = (\lambda_1 + y\lambda_1')G_1 + \cdots + (\lambda_t + y\lambda_t')G_t$$

To find the λ and λ' vectors, we need to consider one more iteration in the modified Berlekamp-Massey algorithm.

We can derive λ_i and λ_i''s from the output of the modified BM algorithm, but still we dont know value of y. Now, we will use the other two equations in the Dickson matrix of vector g and the relationship between g_0 and g_k.

$$\begin{aligned} g_0 &= (\lambda_1 + y\lambda_1')g_{n-1}^{[1]} + \cdots + (\lambda_t + y\lambda_t')g_{n-t}^{[t]} \\ g_{k+t} &= (\lambda_1 + y\lambda_1')g_{k+t-1}^{[1]} + \cdots + (\lambda_t + y\lambda_t')g_k^{[t]} \end{aligned}$$

And we have from the previous steps:

$$\begin{aligned} \gamma_k &= g_k + f_k \\ f_k &= \eta f_0^{q_0^h} \\ f_0 &= \gamma_0 - g_0 \\ \gamma_k &= g_k + \eta(\gamma_0 - g_0)^{q_0^h} \\ \gamma_k &= g_k + \eta(\gamma_0)^{q_0^h} - \eta(g_0)^{q_0^h} \\ g_k &= \gamma_k - \eta(\gamma_0)^{q_0^h} + \eta(g_0)^{q_0^h} \end{aligned}$$

The values of γ_k and γ_0 are already known. Using these three equations we get a polynomial to solve. By solving this polynomial we may find value of y and hence the Λ_i values.

The algorithm may found zero, one or more than one candidate value for y. If any candidate y value and the calculated Λ_i values by using the value of y can derive all the elements of vector g with the period n, then the original message can be recovered successfully. If all the elements of vector g could not bet derived with any y value, with period the algorithm outputs decoding failed.

3.3.4 Recovering the original message

If we found a unique solution with all elements of vector g, then we can easily compute $\tilde{f}$ then hence f.

$$g_{k+1}, \ldots, g_{n-1} \text{ are already known.}$$

$$g_0, \ldots, g_k \text{ can be calculated.}$$

$$\begin{aligned}
&\tilde{f} = \gamma - g \\
&f = \{f_i = \bar{f}_i, \forall i \in [0, k)\}
\end{aligned}$$

CHAPTER 4

THE IMPLEMENTATION

As is mentioned before, a decoding algorithm for AGTG codes offered by Li & Kadir. We implemented this algorithm for both encoding and decoding by using GP programming language and Git version control system. GP is a part of PARI/GP computer algebra system designed for fast computations. [1]

4.1 PARI/GP Computer Algebra System

Pari/GP was developed by Prof. Dr. Henri Cohen at the University Bordeaux. Now, it is free and open source. Many volunteer contributors help to development of PARI/GP. [1]

GP is a scripting programming language. It has own shell with same name. GP commands can run from script files or directly from shell or mixed.

Pari is a huge mathematical library for C and C++ programming languages. It can be directly used from C files. GP scripts can also be converted to C language by using gp2c compiler. Pari Group claims that gp2c-compiled scrpits 3 or 4 times faster than GP scripts.[1]

In this implementation, we used source distribution of PARI-2.13.0 stable version. We focused readability and maintainabilty rather than performance. So, optimizing the source code may be a work for after this work. Default configuration was used while running the implementation. GMP kernel is used instead of PARI's native kernel with advice of Pari Group.

4.2 License

The GP implementation of both encoding and decoding algorithm is licensed under GNU GPL v3.0. GPL is published by FSF (Free Software Foundation) and it allows to modifiying or redistributing this code under GNU GPL v3.0 or later version.

4.3 Directory Structure

While developing this implementation we used git version control system [29]. So we have a *.gitignore* [6] file on the root folder to exclude some IDE specific files and generated output files.

EditorConfig is a tool which helps maintain consistent coding styles for multiple developers working on the same project across various editors and IDEs [32]. The root folder contains an *.editorconfig* configuration file.

The root folder also includes *src* folder for source code, a license file and a readme file.

```
root
└─ src
    ├─ constants
    │   ├─ constants.gp
    │   ├─ decoding_sample_1_constants.gp
    │   └─ encoding_sample_1_constants.gp
    ├─ decoding
    │   ├─ functions
    │   │   ├─ berlekamp_massey.gp
    │   │   └─ find_omega_candidates.gp
    │   └─ decoding.gp
    ├─ encoding
    │   ├─ functions
    │   │   ├─ calculate_eta.gp
    │   │   ├─ calculate_f_bar.gp
    │   │   ├─ evaluate_codeword.gp
    │   │   └─ generate_error_vector.gp
    │   └─ encoding.gp
    ├─ utils
    │   ├─ check_parameters.gp
    │   ├─ composite_function.gp
    │   ├─ dickson.gp
    │   ├─ die.gp
    │   ├─ fliep.gp
    │   ├─ generate_finite_field.gp
    │   ├─ list_to_vector.gp
    │   ├─ modular_index_for_vector.gp
    │   ├─ moore.gp
    │   ├─ polynomial_utils.gp
    │   ├─ print_vector.gp
    │   └─ transpose.gp
    └─ decode.sh
```

```
root
├── src
│   └── encode.sh
├── .editorconfig
├── .gitignore
└── LICENSE.txt
```

The *src* folder contains two shell script files which are includes related gp commands for encoding and decoding.

This is the gp command in encode.sh file.

```
gp --fast --quiet encoding/encoding.gp
```

The gp command takes two option and a parameter in this command. The option *fast* means *fast start: do not read .gprc file*[28]. Pari/GP allows to specify general preferences in a configuration file named *.gprc*, but we do not need to use it for current work. The option *quiet* means *quiet mode: do not print banner and history numbers*[28]. By default, gp prints this information at the beginning of the session. And the parameter is the filepath of main gp file. gp shell can include source files and it can be halted from any of these files instead of waiting to type quit command manually on shell.

There are four main folders in src folder. *constants* folder includes parameter files with predefined values. *constants.gp* file specifies which file will be used and *constants_n_k_t.gp* files have various preferenced values inside.

```
/**
* This file reads global system parameters and inputs
* Both encoding and decoding constants files should include
*   global system parameters:
*     @param n degree of finite field
*     @param k length of codeword
*     @param t maximum rank of the error vector
*     @param q_0 base prime number
*     @param h twisting power
*     @param s generalization power
*     @param u additive power
* Encoding constants file should include:
*   input:
*     @param ff_generator a primitive polynomial over F_q^n
*                         to generate a finite field.
* Decoding constants file should include:
*   input:
*     @param ff_generator a primitive polynomial over F_q^n
* For more information
```

```
*     @see https://ieeexplore.ieee.org/document/7723881
*/

\r constants/decoding_sample_1_constants.gp

allocatemem(1000*2^20);
```

\r stands for *read* and the command is used for importing another gp file.

utils folder includes common mathematical functions, such as transpose a matix, can be used for both encoding and decoding.

encoding and *decoding* folders include a main file for related operation with same name *encoding.gp* and *decoding.gp*. Both has *functions* folder which are include gp files for operation specific mathematical functions, such as calculate $\tilde{f}$.

4.4 Constants

constants.gp file imports a file with prefered parameter as mentioned above. A parameter file should define following parameters: $n, k, t, s, h, u, q_0, q, ff\_generator, f$, where: n, k, s, h, u, q_0, q described in the section (2.10), $ff\_generator$ is the primitive function which generates the finite field, f is the hardcoded input message as a list of finite field elements will be encoded. To decode any encoded message, same parameter file should be used.

4.5 Encoding

encoding.gp file contains main method of encoding and transmission processes. To keep the code simple encoding and transmission stages are kept together. This method reads *constants.gp* file to take global parameters. Reads the input message from same file. Defined global variables are checked. The main method generates codeword from the input message, and adds an error vector to codeword. Outputs linearly independent evaluation points vector α, received word r and the value of η.

4.6 Decoding

decoding.gp file contains main method of decoding process. This method reads *constants.gp* file to take global parameters.

Reads the linearly independent evaluation points vector α, received word r and the value of η from same file. Defined global variables are checked.

The main method finds known elements of the vector g, applies the modified Berlekamp-Massey algorithm to the known elements. If the solution could not be found, the algorithm continues to one more iterate with known relation between g_0, g_k and g_{k+t}. If the algorithm find possible solutions, then iterates them to find exact solution, outputs the founded original message f, or prints *"Decoding failure!"*.

CHAPTER 5

CONCLUSION

We implemented the interpolation based decoding algorithm by proposing Li and Kadir for Additive Generalized Twisted Gabidulin Codes. In general we have a decoding algorithm with complexity $\mathcal{O}(n^2)$

We did not do implementation performance-oriented. So we did not put the execution time results for the examples into this thesis. If AGTG codes will be used in practise to encode some messages, then optimized version of this algorithm can be used.

Converting the GP scripts to C language by using gp2c and making it work with the GMP/MPIR library can be considered for optimization.

REFERENCES

[1] B. Allombert and K. Belabas, Pari/gp official homepage, `https://pari.math.u-bordeaux.fr`, Apr 2021, accessed: 2021-08-11.

[2] E. Berlekamp, Algebraic coding theory, in *McGraw-Hill series in systems science*, 1968.

[3] E. Gabidulin, Theory of codes with maximum rank distance (translation), Problems of Information Transmission, 21, pp. 1–12, 01 1985.

[4] E. Gabidulin, A. Paramonov, and O. Tretjakov, Ideals over a non-commutative ring and their application in cryptology, pp. 482–489, 04 1991, ISBN 3-540-54620-0.

[5] E. Gabidulin and N. Pilipchuk, Gpt cryptosystem for information network security, pp. 17–21, 01 2013.

[6] Git Community, *Gitignore Documentation*, 2021, available from `https://git-scm.com/docs/gitignore/`.

[7] M. Golay, Notes on digital coding, Proceedings of The IEEE - PIEEE, 37, 01 1949.

[8] V. Goppa, A new class of linear error correcting codes, Problemy Peredachi Informatsii, 6, pp. 24–30, 01 1970.

[9] R. Hamming, *Error Detecting and Error Correcting Codes (1950)*, pp. 135–146, 02 2021, ISBN 9780262363174.

[10] W. Kadir and C. Li, On decoding additive generalized twisted gabidulin codes, Cryptography and Communications, 12, 09 2020.

[11] A. Kshevetskiy and E. Gabidulin, The new construction of rank codes, pp. 2105 – 2108, 10 2005.

[12] C. Li and W. Kadir, On decoding additive generalised twisted gabidulin codes, 03 2019.

[13] R. Lidl and H. Niederreiter, *Introduction to Finite Fields and their Applications*, Cambridge University Press, 2 edition, 1994.

[14] G. Lunardon, R. Trombetti, and Y. Zhou, Generalized twisted gabidulin codes, Journal of Combinatorial Theory, Series A, 159, p. 79–106, Oct 2018, ISSN 0097-3165.

[15] J. Massey, Shift-register synthesis and bch decoding, IEEE Transactions on Information Theory, 15(1), pp. 122–127, 1969.

[16] R. Mceliece, A public-key cryptosystem based on algebraic coding theory, JPL DSN Progress Report, 44, 05 1978.

[17] E. H. Moore, A two-fold generalization of Fermat's theorem, Bulletin of the American Mathematical Society, 2(7), pp. 189 – 199, 1900.

[18] O. Ore, On a special class of polynomials, Transactions of the American Mathematical Society, 35, pp. 559–584, 1933.

[19] K. Otal and F. Özbudak, Additive rank metric codes, IEEE Transactions on Information Theory, 63(1), pp. 164–168, 2017.

[20] T. Randrianarisoa, A decoding algorithm for rank metric codes, 12 2017.

[21] H. Rashwan, E. M. Gabidulin, and B. Honary, Security of the gpt cryptosystem and its applications to cryptography, Security and Communication Networks, 4(8), pp. 937–946, 08 2011.

[22] I. Reed and G. Solomon, Polynomial codes over certain finite fields, Journal of the Society for Industrial and Applied Mathematics, 8, pp. 300–304, 06 1960.

[23] G. Richter and S. Plass, Error and erasure decoding of rank-codes with a modified berlekamp-massey algorithm, 01 2004.

[24] N. Sendrier, *McEliece Public Key Cryptosystem*, pp. 767–768, Springer US, Boston, MA, 2011, ISBN 978-1-4419-5906-5.

[25] C. E. Shannon, A mathematical theory of communication, The Bell System Technical Journal, 27(3), pp. 379–423, 1948.

[26] J. Sheekey, A new family of linear maximum rank distance codes, Advances in Mathematics of Communications, 10, 04 2015.

[27] P. Shor, Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, SIAM Journal on Computing, 26, pp. 1484–1509, 10 1997.

[28] The PARI Group, Univ. Bordeaux, *PARI/GP manual*, Sep 2013, available from `https://pari.math.u-bordeaux.fr/dochtml/gpman.html`.

[29] L. Torvalds, Git official homepage, `http://git-scm.com`, Aug 2021, accessed: 2021-08-11.

[30] W. Trappe and L. Washington, Introduction to cryptography with coding theory (2nd edition), 2005.

[31] B. Wu and Z. Liu, Linearized polynomials over finite fields revisited, Finite Fields and Their Applications, 22, pp. 79–100, 2013, ISSN 1071-5797.

[32] H. Xu, Editorconfig homepage, `http://editorconfig.org`, Aug 2021, accessed: 2021-08-11.

APPENDIX A

DIRECTORY TREE OF THE CODES

```
root
└─ src
   ├─ constants
   │  ├─ constants.gp
   │  ├─ decoding_sample_1_constants.gp
   │  └─ encoding_sample_1_constants.gp
   ├─ decoding
   │  ├─ functions
   │  │  ├─ berlekamp_massey.gp
   │  │  └─ find_omega_candidates.gp
   │  └─ decoding.gp
   ├─ encoding
   │  ├─ functions
   │  │  ├─ calculate_eta.gp
   │  │  ├─ calculate_f_bar.gp
   │  │  ├─ evaluate_codeword.gp
   │  │  └─ generate_error_vector.gp
   │  └─ encoding.gp
   ├─ utils
   │  ├─ check_parameters.gp
   │  ├─ composite_function.gp
   │  ├─ dickson.gp
   │  ├─ die.gp
   │  ├─ fliep.gp
   │  ├─ generate_finite_field.gp
   │  ├─ list_to_vector.gp
   │  ├─ modular_index_for_vector.gp
   │  ├─ moore.gp
   │  ├─ polynomial_utils.gp
   │  ├─ print_vector.gp
   │  └─ transpose.gp
   └─ decode.sh
```

```
root
├── src
│   └── encode.sh
├── .editorconfig
├── .gitignore
└── LICENSE.txt
```

APPENDIX B

SOURCE CODE

```
# top-most EditorConfig file
root = true

# Unix-style newlines with a newline ending every file
[*]
charset = utf-8
trim_trailing_whitespace = true
end_of_line = lf
insert_final_newline = true

# Tab indentation (no size specified)
[Makefile]
indent_style = tab

[*.gp]
indent_style = space
indent_size = 2
```

Listing B.1: .editorconfig

```
.idea/
```

Listing B.2: .gitignore

```
Copyright (C) 2021  Rıdvan Özkerim

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
```

```
along with this program.
If not, see <https://www.gnu.org/licenses/>.
```

Listing B.3: LICENSE.txt

```
#!/usr/bin/env bash

gp --fast --quiet encoding/encoding.gp
```

Listing B.4: src/encode.sh

```
#!/usr/bin/env bash

gp --fast --quiet decoding/decoding.gp
```

Listing B.5: src/decode.sh

```
/**
* This file reads global system parameters and inputs
* Both encoding and decoding constants files should include
*   global system parameters:
*     @param n degree of finite field
*     @param k length of codeword
*     @param t maximum rank of the error vector
*     @param q_0 base prime number
*     @param h twisting power
*     @param s generalization power
*     @param u additive power
* Encoding constants file should include:
*   input:
*     @param ff_generator a primitive polynomial over F_q^n
*                         to generate a finite field.
* Decoding constants file should include:
*   input:
*     @param ff_generator a primitive polynomial over F_q^n
* For more information
*     @see https://ieeexplore.ieee.org/document/7723881
*/

\r constants/decoding_sample_1_constants.gp

allocatemem(1000*2^20);
```

Listing B.6: src/constants/constants.gp [A]

```
{
  s=2;
  h=3;
  u=2;
```

```

  n=7;
  k=3;
  t=2;

  q_0 = 2;
  q = q_0^u;

  ff_generator = w^7 + w^1 + Mod(1,2);

  f = [ w^6 + w^4 + w^3 + w^2 + 1,
        w^4 + w^2 + 1,
        w^6 + w^5 + w^4 + w^3 + w^2 + w + 1
      ];
}
```

Listing B.7: src/constants/encoding_sample_1_constants.gp

```
{
  s=2;
  h=2;
  u=2;

  n=7;
  k=3;
  t=2;

  q_0 =2;
  q = q_0^u;

  ff_generator = w^7 + w^1 + Mod(1,2);

  \\ ita is a chosen value from encoding process.
  \\ "eta" is a predefined function in Pari-GP
  \\ for Dedekind's eta function.
  \\ so we will use "ita" as a parameter name to define "eta".
  ita = 0;

  \\ alpha is the linear independent evaluation points
  \\ over the finite field GF(q^n)
  alpha = [ w^5 + w^3 + w^2 + 1,
            w^6 + w^5 + w^2,
            w^5 + w^4 + w^3 + w + 1,
            w^5 + w,
            w^6 + w^5 + w^3 + w^2,
            w^6 + w^4 + w + 1,
            w^6 + w + 1
          ];

  \\ r is the received word from encoding process
```

```
  r = [ w^6 + w^5 + w^4 + w^3 + w + 1,
        w^5 + w^4 + w + 1,
        w^5 + w^2 + w + 1,
        w^6 + w^5 + w^3 + w^2 + w + 1,
        w^6 + w^5 + w^4 + 1,
        w^6 + w^5 + w^3 + w^2 + 1,
        w^6 + w^4 + w^3
      ];
}
```

Listing B.8: src/constants/decoding_sample_1_constants.gp

```
\r constants/constants.gp
\r utils/check_parameters.gp
\r utils/dickson.gp
\r utils/die.gp
\r utils/fliep.gp
\r utils/generate_finite_field.gp
\r utils/modular_index_for_vector.gp
\r utils/moore.gp
\r utils/polynomial_utils.gp
\r utils/print_vector.gp
\r utils/transpose.gp
\r decoding/functions/berlekamp_massey.gp
\r decoding/functions/randrianarisoa.gp
\r decoding/functions/sidorenko.gp
\r decoding/functions/find_omega_candidates.gp

/**
* Main function of whole decoding operation
* for additive generalized twisted gabidulin codes.
* This method reads constants.gp file to take global parameters,
* linearly independent evaluation points alpha,
* eta value and the received word r,
* checks are parameters valid,
* tries to decode given word r to get original message f
*/
decoding() = {
  check_parameters(n, k, t, q_0, h, s, u);

  if(length(r) != n,
    die("The length of the received word r must be n!"));

  if(length(alpha) != n,
    die("The length of the evaluation vector alpha must be n!"));

  w = generate_finite_field(ff_generator);

  if(polisirreducible(w.mod) == 0,
    die("The generator polynomial must be irreducible!"));
```

```
for(i=1,n,alpha[i]=eval(alpha[i]); r[i]=eval(r[i]));

\\ "eta" is a predefined function in Pari-GP
\\ for Dedekind's eta function.
\\ so we will use "ita" as a parameter name to define "eta".
ita = eval(ita) + w - w;
if(type(ita) != "t_FFELT",
  die("eta must be defined!"));

\\ transpose of the moore matrix
mt = transpose(moore(alpha, q, s));

\\ calculate gama, which is equal to (f_bar + g)
gama = eval(r * (mt^(-1)));
print_vector(gama, "gama");

bm = berlekamp_massey(gama, n, k, t, q, s);

L = bm[1];

print("Result of BM Algorithm:");
print(bm);

/* case 2 */
if(L == (n-k) / 2,
  my(delta_r = 'y
               + sum(i = 1, L, bm[2][i+1]
                     * gama[n-i+1]^(q^(s*i))));
  print("delta_r");
  print(delta_r);

  my(delta_r_f = (x) -> delta_r * x^(q^s));
  my(lamda_r(x)=get_linearized_polynomial_of_coefficients(bm[2],
  q^s)(x) + composite(delta_r_f,
  get_linearized_polynomial_of_coefficients(bm[3], q^s))(x));
  cfs = get_coefficients_of_linearized_polynomial(lamda_r, q^s);

  print("cfs");
  print(cfs);

  lamdas = vector(L, i,
                  bm[2][i+1] - (delta_r * (bm[3][i]^(q^s))));
  lamda_overs = vector(L, i,
                       -1 * bm[3][i]^(q^s));

  print("lamdas");
  print(lamdas);
  print("lamda_overs");
  print(lamda_overs);
```

```
    omegas = find_omega_candidates(gama, lamdas, lamda_overs, cfs,
                                   ita, q_0, q, n, k, t, s, h, u)
                                   ;

    is_f_found = 0;
    for(i=1, length(omegas),

      \\ inital g
      \\ we only know last n-k elements, which are same with gama
      g = vector(n, i, if(i<k+2, 0, gama[i]));
      g_init = g; \\ clone for testing
      print_vector(g, "g_init");

      y = omegas[i];
      put_found_y_for_g(y, gama, lamdas, lamda_overs, cfs,
                        ita, q_0, q, n, k, t, s, h, u);
      forstep(i=1, n, 1,
        g[modix(i, n)] = sum(j=1, t,
          (g[modix(i-j, n)]^(q^(s*j)))*(eval(lamdas[j]) )));
          \\ + y*lamda_overs[j])));

      print_vector(gama, "gama");
      print_vector(g, "g in iteration");

      g_counter = 0;
      for(j=k+1, n, g_counter = g_counter + (g[j] == g_init[j]));
      if(g_counter == (n-k-1),
        f_bar = gama - g;
        print_vector(f_bar, "f_bar found");
        is_f_found = 1;
        break();
      );

    );

    if(is_f_found == 0,
       print("Decoding Failure!"));

  );

}

decoding();
quit();
```

Listing B.9: src/decoding/decoding.gp

```
/**
* Identity function
```

```
*
* @param x any value
* @return x
*/
identity_function (x) = {
  x
};

/**
* A modified Berlekamp-Massey algorithm by Li & Kadir
*
* The general working logic of the algorithm is as follows.
* It produces an LFSR from scratch.
* It iteratively processes known elements of the vector.
* At each iteration, it calculates the difference between the value
* produced by our LFSR and the value it should actually be.
* According to this difference,
* it updates the conversion functions in LFSR.
* If necessary, it extends the length of the LFSR
* and shifts the lambda values.
* It gets closer to the true lambda values with each iteration.
* It produces a second LFSR in its modified version.
* B denotes the second LFSR.
*
* @param g vector which is partially known
* @param n degree of finite field
* @param k length of codeword
* @param t maximum rank of the error vector
* @param q power of base prime number
* @param s generalization power
*
* @return list of L, Lambda vector and B vector
*/
berlekamp_massey (g: t_VEC, \
                  n: t_INT, \
                  k: t_INT, \
                  t: t_INT, \
                  q: t_INT, \
                  s: t_INT) = {
  result = List();

  \\ the linearized polynomials
  lamda = List();

  \\ the auxiliary linearized polynomial which is used to store the
  \\ value of BS(i)(x) with the largest degree Li such that Li < L.
  B = List();

  \\ L is the linear complexity of
  L = 0;
```

```
listinsert(~lamda, identity_function, 1);
listinsert(~B, identity_function, 1);

for(r = 1, n-k-1,
  printf("r = %d\n", r-1);
  my(lamdas = get_coefficients_of_linearized_polynomial(lamda[r],
                                                         q^s));

  my(delta_r = g[k+1+r]
              + sum(i = 1, L, lamdas[i+1]
                    * g[k+1+r-i]^(q^(s*i))));
  printf("dr = %s\n", delta_r);

  if(delta_r == 0,
    print("if");
    listinsert(~lamda, (x) -> lamda[r](x), r+1);
    listinsert(~B, composite((x) -> x^(q^s), B[r]), r+1);
    printf("  lamda[%d]: %s\n", r,
    get_coefficients_of_linearized_polynomial(lamda[r+1], q^s));
    printf("  B[%d]: %s\n", r,
    get_coefficients_of_linearized_polynomial(B[r+1], q^s));
  ,
    print("else");
    my(dr = (x) -> delta_r * x^(q^s));

    listinsert(~lamda,
               (x) -> lamda[r](x) - composite(dr, B[r])(x),
               r+1);
    printf("  lamda[%d]: %s\n", r,
    get_coefficients_of_linearized_polynomial(lamda[r+1], q^s));

    if(2*L > r-1,
      print("  if");
      listinsert(~B, composite((x) -> x^(q^s), B[r]), r+1);
      printf("    B[%d]: %s\n", r,
      get_coefficients_of_linearized_polynomial(B[r+1], q^s));
    ,
      print("  else");
      listinsert(~B, (x) -> delta_r^(-1) * lamda[r](x), r+1);
      printf("    B[%d]: %s\n", r,
      get_coefficients_of_linearized_polynomial(B[r+1], q^s));
      L = r - L;
      printf("    L = %d\n", L);
    )
  )
);

listput(result, L, 1);

listput(result,
```

```
            get_coefficients_of_linearized_polynomial(lamda[n-k], q^s),
            2);

    listput(result,
            get_coefficients_of_linearized_polynomial(B[n-k], q^s),
            3);

    result
};
```

Listing B.10: src/decoding/functions/berlekamp_massey.gp

```
find_omega_candidates (gama: t_VEC, \
                       lamdas: t_VEC, \
                       lamda_overs: t_VEC, \
                       cfs: t_VEC, \
                       ita: t_FFELT, \
                       q_0: t_INT, \
                       q: t_INT, \
                       n: t_INT, \
                       k: t_INT, \
                       t: t_INT, \
                       s: t_INT, \
                       h: t_INT, \
                       u: t_INT \
                       ) = {
  my(g_0 = sum(i=1, t,
    (cfs[i+1]*gama[n+1-i]^(q^(s*i)))));
\\  (lamdas[i] + 'y * lamda_overs[i]) * gama[n-i]^(q_0^(u*s*i))));

  my(g_k = gama[k+1]
        - (ita * (gama[1]^(q_0^h)))
        + (ita * (g_0^(q_0^h))));

  my(g_k_plus_t = sum(i=1, t-1,
    (cfs[i+1]*gama[k+t+i-1]^(q^(s*i)))));
\\    (lamdas[i] + 'y * lamda_overs[i]) * gama[k+t+1-i]^(q_0^(u*s*i))))
\\    ;
  g_k_plus_t = g_k_plus_t +
\\    (lamdas[t] + 'y * lamda_overs[t]) * g_k^(q_0^(u*s*t));
    cfs[t+1]*g_k^(q^(s*t));

  my(polynomial_to_solve = eval(g_k_plus_t - gama[k+t+1]));
  print("polynomial to solve");
  print(polynomial_to_solve);

  my(omegas = polynomial_solve(polynomial_to_solve));
  print_vector(omegas, "omegas");
  omegas
};
```

```

put_found_y_for_g(y: t_FFELT, \
                  gama: t_VEC, \
                  lamdas: t_VEC, \
                  lamda_overs: t_VEC, \
                  cfs: t_VEC, \
                  ita: t_FFELT, \
                  q_0: t_INT, \
                  q: t_INT, \
                  n: t_INT, \
                  k: t_INT, \
                  t: t_INT, \
                  s: t_INT, \
                  h: t_INT, \
                  u: t_INT \
                  ) = {
  my(g_0 = sum(i=1, t,
      (cfs[i+1]*gama[n+1-i]^(q^(s*i)))));
  g_0 = eval(g_0);
  printf("g_0: %s\n", g_0);

  my(g_k = gama[k+1]
        - (ita * (gama[1]^(q_0^h)))
        + (ita * (g_0^(q_0^h))));
  g_k = eval(g_k);
  printf("g_k: %s\n", g_k);

  my(g_k_plus_t = sum(i=1, t-1,
      (cfs[i+1]*gama[k+t+i-1]^(q^(s*i))))
      + cfs[t+1]*g_k^(q^(s*t)));
  g_k_plus_t = eval(g_k_plus_t);
  printf("g_k_plus_t: %s\n", g_k_plus_t);
}
```

Listing B.11: src/decoding/functions/find_omega_candidates.gp

```
\r constants/constants.gp
\r utils/check_parameters.gp
\r utils/dickson.gp
\r utils/die.gp
\r utils/fliep.gp
\r utils/generate_finite_field.gp
\r utils/list_to_vector.gp
\r utils/modular_index_for_vector.gp
\r utils/moore.gp
\r utils/print_vector.gp
\r utils/transpose.gp
\r encoding/functions/calculate_eta.gp
\r encoding/functions/calculate_f_bar.gp
\r encoding/functions/generate_error_vector.gp
```

```
\r encoding/functions/evaluate_codeword.gp

/**
* Main function of whole encoding and transmission operation
* for additive generalized twisted gabidulin codes.
* This method reads constants.gp file to take global parameters
* and the message f,
* checks are parameters valid,
* encodes given message f,
* adds an error to f to simulate transmission stage,
* prints corrupted codeword r,
* prints linear independent evaluation points vector "alpha"
* prints finite field element eta used
*/
encoding() = {
  check_parameters(n, k, t, q_0, h, s, u);

  if(length(f) != k,
    die("The length of the message f must be k!"));

  w = generate_finite_field(ff_generator);

  if(polisirreducible(w.mod) == 0,
    die("The generator polynomial must be irreducible!"));

  for(i=1,k,f[i]=eval(f[i]));

  \\ "eta" is a predefined function in Pari-GP
  \\ for Dedekind's eta function.
  \\ so we will use "ita" as a parameter name to define "eta".
  ita = calculate_eta(q_0, n, k, u, w);
  printf("eta = %s\n", ita);

  \\ alpha is the linear independent evaluation points vector
  \\ over the finite field GF(q^n)
  alpha = fliep(n, q, w);
  print_vector(alpha, "alpha");

  \\ REGION WAY 1: to calculate the codeword c

  \\ calculate the vector f_bar of length n as defined
  f_bar = calculate_f_bar(f, k, n, q_0, h, ita, w);

  \\ evaluate f(x) over linear independent evaluation points alpha
  \\ then we get the codeword
  c = evaluate_codeword(alpha, n, f, q_0, q, s, h, k, ita);

  \\ END REGION WAY 1

  \\ REGION WAY 2: to calculate the codeword c
```

```

  \\ transpose of the moore matrix of alpha
  m = moore(alpha, q, s);
  mt = transpose(m);

  \\ find the codeword c with an alternative way
  c2 = eval(f_bar * mt);

  \\ END REGION WAY 2

  /*
  \\ see both codeword is identically same
  printf("c and c2 are identically %s\n",
    if(c == c2, "same", "different"));
  */

  \\ generate a random error vector
  e = generate_error_vector(q, t, w);

  \\ add error vector to codeword
  \\ "received word" by decoder or "sent word" by encoder
  r = eval(c + e);
  print_vector(r, "r");
};

encoding();
\\ quit();
```

Listing B.12: src/encoding/encoding.gp

```
/**
* Calculates an eta value
* to generate f_k value
*
* @param q_0 base prime number
* @param n degree of finite field
* @param k length of codeword
* @param u additive power
* @param w base element of finite field
*
* @return calculated eta value
*/
calculate_eta(q_0: t_INT, \
              n: t_INT, \
              k: t_INT, \
              u: t_INT, \
              w: t_FFELT) = {
  q = q_0 ^ u;

  a = (w^0)*((-1)^(n*k*u));
```

```
  b = (((q^n)-1)/(q-1));
  c = (w^b);
  d = ((w^2)^b);

  \\ "eta" is a predefined function in Pari-GP
  \\ for Dedekind's eta function.
  \\ so we will use "ita" as a parameter name to define "eta".
  if(a != c, ita = w,
     a != d, ita = (w^2),
     ita = random(w);
     until(a != ita^b, ita = random(w));
     );

  ita
};
```

Listing B.13: src/encoding/functions/calculate_eta.gp

```
/**
* Calculates an eta value
* to generate f_k value
*
* @param f vector of the chosen message
* @param k length of codeword
* @param n degree of finite field
* @param q_0 base prime number
* @param h twisting power
* @param ita calculated eta value
* @param w base element of finite field
*
* @return correlated f_bar vector of f
*/
calculate_f_bar(f: t_VEC, \
                k: t_INT, \
                n: t_INT, \
                q_0: t_INT, \
                h: t_INT, \
                ita: t_FFELT, \
                w: t_FFELT) = {
  f_bar = f;
  f_bar = concat(f_bar, [ita * f[1]^(q_0^h)]);
  f_bar = concat(f_bar, vector(n-k-1, i, w-w));
  f_bar
};
```

Listing B.14: src/encoding/functions/calculate_f_bar.gp

```
/**
* Correlated f(x) function of the linearized polynomial f
*
```

```
* @param x variable for f(x) function
* @param f array of the chosen message
* @param q_0 base prime number
* @param q power of base prime number
* @param s generalization power
* @param h twisting power
* @param k length of codeword
* @param ita calculated eta value
*
* @return value of f(x)
*/
fx(x: t_FFELT, \
   f: t_VEC, \
   q_0: t_INT, \
   q: t_INT, \
   s: t_INT, \
   h: t_INT, \
   k: t_INT, \
   ita: t_FFELT) = {
    su = sum(i=1, k, f[i]*(x^(q^(s*(i-1)))));
    f_bar_k = ita * f[1]^(q_0 ^ h) * x^(q^(s*k));
    su = su + f_bar_k;
    su
};

/**
* Evaluates codeword f
* on linearly independent points alpha
*
* @param alpha vector of linearly independent points on GF(q^n)
* @param n degree of finite field
* @param f array of the chosen message
* @param q_0 base prime number
* @param q power of base prime number
* @param s generalization power
* @param h twisting power
* @param k length of codeword
* @param ita calculated eta value
*
* @return vector of the codeword c
*/
evaluate_codeword(alpha, n, f, q_0, q, s, h, k, ita) = {
    vector(n, i, fx(alpha[i], f, q_0, q, s, h, k, ita))
};
```

Listing B.15: src/encoding/functions/evaluate_codeword.gp

```
/**
* Generates an error vector
* to simulate corruption during transmission
```

```
*
* @param q power of base prime number
* @param t maximum rank of the error vector
* @param w base element of finite field
*
* @return a random error vector
*/
generate_error_vector(q, t, w) = {
  e_raw = fliep(t, q, w);
  e_raw = concat(e_raw, vector(n-t, i, w-w));
  print_vector(e_raw, "e_raw_raw");
  u = matrix(n, n, i, j, random(w));
  print_vector(u);
  e_raw * u
};
```

Listing B.16: src/encoding/functions/generate_error_vector.gp

```
/**
* Checks the given parameters are suitable for the system.
* If parameters are not suitable,
* prints error message then halt the program.
*
* @param n degree of finite field
* @param k length of codeword
* @param t maximum rank of the error vector
* @param q_0 base prime number
* @param h twisting power
* @param s generalization power
* @param u additive power
*/
check_parameters(n: t_INT, \
                 k: t_INT, \
                 t: t_INT, \
                 q_0: t_INT, \
                 h: t_INT, \
                 s: t_INT, \
                 u: t_INT) = {
  if(n<1 || k<1 || s<1 || u<1 || h<1,
    die("AGTG parameters should be positive integers!"));

  if(k >= n,
    die("AGTG parameter k should be less than n!"));

  if(isprime(q_0) == 0,
    die("AGTG parameter q_0 must be a prime number!"));

  if(gcd(s, n) != 1,
    die("AGTG parameters s and n should not have common divisor!"));

```

```
  my(max_t = floor((n-k)/2));
  if(t > max_t,
    die("t is bigger than the maximum value!"));
};
```

Listing B.17: src/utils/check_parameters.gp

```
/**
* Symbolic product of two polynomials
* Parameter order is important.
*
* @param f the first polynomial
* @param g the second polynomial
*
* @return composite function of two input functions
*/
composite(f: t_FUNC, g: t_FUNC) = {
   (x) -> f(g(x))
};
```

Listing B.18: src/utils/composite_function.gp

```
/**
* Converts a vector to its correlated Dickson matrix
*
* @param v base vector
* @param q power of base prime number
*
* @return Dickson matrix of the vector
*/
dickson(v: t_VEC, q: t_INT) = {
    len = length(v);
    matrix(len, len, i, j, v[modix(i-j+1, len)]^(q^(j-1)))
};
```

Listing B.19: src/utils/dickson.gp

```
/**
* Prints the error message
* then halts the execution
*
* @param message the error message
*/
die(message: t_STR) = {
  printf("ERROR: %s\n", message);
  quit();
};
```

Listing B.20: src/utils/die.gp

```
fliep_get_span(pl, pq, pa, pksmall) = {
  my(i = listcreate(pl));
  for(j=1,pl,listput(i,1,j));
  my(span = Set());
  while(i[1] < pq+1,
    my(t=0);
    for(j=1, pl, t= t+pa[j]* pksmall[i[j]]);
    span = setunion(span, Set(t));
    listput(i, i[pl]+1, pl);
    forstep(j=pl, 1, -1,
      if(i[j]>pq,
        if(j==1, break,
          listput(i, 1, j);
          listput(i, i[j-1]+1, j-1);
        )
      )
    )
  );
  span;
};

/*
* Calculates the rank of the correlated
* Dickson matrix of the given vector v
*
* @param v base vector
* @param q power of base prime number
*
* @return rank of the vector
*/
rank_of_vector(v: t_VEC, q: t_INT) = {
  matrank(dickson(v, q))
};

/*
* Checks independency of linearized evaluation points
*
* @param v base vector
* @param n degree of finite field
* @param q power of base prime number
*
* @return boolean value of independency
*/
check_liep(v: t_VEC, n: t_INT, q: t_INT) = {
  rank_of_vector(v, q) == n
};

/*
* Finds linearly independent evaluation points
```

```
* by choosing random elements of finite field
* and checking independency.
*
* @param n degree of finite field
* @param q power of base prime number
* @param w base element of finite field
*
* @return n linearly independent points in the finite field.
*/
fliep(n: t_INT, q: t_INT, w: t_FFELT) = {
  my(liep = vector(n, i, random(w)));
  if(check_liep(liep, n, q), liep, fliep(n, q, w))
};

/**
* Finds linearly independent evaluation points
* @deprecated
*
* @param n degree of finite field
* @param q power of base prime number
* @param w base element of finite field
*
* @return n linearly independent points in the finite field.
*/
fliep_by_spanning(n: t_INT, q: t_INT, w: t_FFELT) = {
    my(karray = [w-w]);
    my(karray = concat(karray, vector(q^n-2, i, w^i)));
    my(kset = Set(karray));

    wsmall = w^floor((q^n-1)/(q-1));
    ksmall = [wsmall-wsmall];
    ksmall = concat(ksmall, vector(q-1, i, wsmall^i));

    my(a = listcreate(n));
    listput(a, kset[random(length(kset)) + 1], 1);

    for(i = 1, n-1,
        my(latest_span = fliep_get_span(i, q, a, ksmall));
            my(latest_set = setminus(kset, latest_span));
            listput(a,
                    latest_set[random(length(latest_set)) + 1],
                    i+1);
    );

    liep = list_to_vector(a);

    if(check_liep(liep, n, q), liep, fliep(n, q, w))
};
```

Listing B.21: src/utils/fliep.gp

```
/**
* Generates the finite field GF(q^n) with given primitive function
*
* @param ff_generator a primitive polynomial over GF(q^n)
*
* @return the finite field generated by the input polynomial
*/
generate_finite_field(ff_generator: t_POL) = {
 ffgen(ff_generator)
};
```

Listing B.22: src/utils/generate_finite_field.gp

```
/**
* Converts given list to a vector with same elements and same order.
*
* @param list any list
*
* @return the generated vector with elements of the list.
*/
list_to_vector(list: t_LIST) = {
    vector(length(list), i, list[i])
};
```

Listing B.23: src/utils/list_to_vector.gp

```
/**
* Finds correct index value in modulo m
*
* For example, if we have a collection with m elements:
*   modix(0, 5) will return 5
*   modix(1, 5) will return 1
*   modix(5, 5) will return 5
*   modix(6, 5) will return 1
*
* It is not same with modulo operation in math,
* because in GP, arrays start with index 1 instead of 0.
*
* @param index given index value not has to be in modulo m
* @param m modulo value
*
* @return index value in modulo m
*/
modix(index: t_INT, m: t_INT) = {
  my(i = lift(Mod(index, m)));
  if(i == 0, i+m, i)
};
```

Listing B.24: src/utils/modular_index_for_vector.gp

```
/**
* Converts a vector to its correlated Moore matrix
*
* @param alpha vector of linearly independent points on GF(q^n)
* @param q power of base prime number
* @param s generalization power
*
* @return Moore matrix of the vector alpha
*/
moore(a, q, s) = {
    my(len = length(a));
    matrix(len, len, i, j, a[i]^(q^(s*(j-1))))
};
```

Listing B.25: src/utils/moore.gp

```
get_coefficients_of_polynomial (f: t_CLOSURE) = {
  vector(poldegree(f(x)) + 1, i, eval(polcoef(f(x), i - 1)))
};

get_coefficients_of_linearized_polynomial (f: t_CLOSURE, \
                                            base: t_INT) = \
{
  cs = get_coefficients_of_polynomial(f);
  degree = logint(length(cs)-1, base);
  vecextract(cs, sum(i = 0, degree, 2^(base^i)))
};

get_polynomial_of_coefficients (v: t_VEC) = {
  x -> sum(i=1, length(v), v[i] * x^(i-1));
};

get_linearized_polynomial_of_coefficients (v: t_VEC, \
                                            base: t_INT) = \
{
  x -> sum(i=1, length(v), v[i] * x^(base^(i-1)));
};

composite(f: t_CLOSURE, g: t_CLOSURE) = {
  fog = (x) -> f(g(x))
};

get_coefficients_of_polynomial_addition \
  (f: t_CLOSURE, g: t_CLOSURE) = {
  degree_of_f = poldegree(f(x));
  degree_of_g = poldegree(g(x));
  max_degree = if(degree_of_f >= degree_of_g, degree_of_f, \
                  degree_of_g);
  coefficients = vector(max_degree + 1, i, \
```

```
                     eval(polcoef(f(x), i - 1) + polcoef(g(x), i - 1)));
  coefficients
};

get_coefficients_of_polynomial_subtraction \
  (f: t_CLOSURE, g: t_CLOSURE) = {
  degree_of_f = poldegree(f(x));
  degree_of_g = poldegree(g(x));
  max_degree = if(degree_of_f >= degree_of_g, degree_of_f, \
                  degree_of_g);
  coefficients = vector(max_degree + 1, i, \
                     eval(polcoef(f(x), i - 1) - polcoef(g(x), i - 1)));
  coefficients
};

polynomial_solve(p: t_POL) = {
  liftall(polrootsmod(p))
};
```

Listing B.26: src/utils/polynomial_utils.gp

```
/**
* Prints the vector name and the vector.
* Prints each element of the vector on a new line.
*
* @param v any vector
* @param name name of the vector. its default value is "vector".
*/
print_vector(v: t_VEC, \
             name = "vector": t_STR) = {
  printf("%s = [\n", name);
  for(i=1, length(v), printf(" %d: %s\n", i, v[i]));
  print("]");
};

/**
* Prints the matrix name and the matrix.
* Prints each element of the matrix on a new line
* with row and column numbers.
*
* @param m any matrix
* @param row_count row count of the matrix m
* @param column_count column count of the matrix m
* @param name name of the matrix. its default value is "matrix".
*/
print_matrix(m: t_MAT, \
             row_count: t_INT, \
             column_count: t_INT, \
             name = "matrix": t_STR \
            ) = {
```

```
  printf("%s = [\n", name);
  for(i=1, row_count,
    printf("   r%d = [\n", i);
    for(j=1, column_count,
      printf(" c%d: %s\n", j, m[i, j]);
    );
    print("   ]");
  );
  print("]");
};
```

Listing B.27: src/utils/print_vector.gp

```
/**
* Encapsulation of matrix transposition
*
* @param m matrix to transpose
* @return transpose of m
*/
transpose(m: t_MAT) = {
    mattranspose(m)
};
```

Listing B.28: src/utils/transpose.gp