Engineering Faculty

Lab of networks and computation

Software Development: python package for fast error-correction codes

Chani Milshtein

Dolev Shmaryahu

Submitted in partial fulfillment of the requirements for the Bachelor Degree of Science in Computer Engineering, Bar Ilan University

Academic instructor: Prof. Ran Gelles

Ramat Gan, Israel October 2024

Acknowledgment

We would like to express our heartfelt gratitude to our advisor, Prof. Ran Gelles, for his invaluable guidance, support, and encouragement throughout this project. Your insights and expertise greatly shaped our work and helped us navigate challenges effectively.

We also want to thank our families and friends for their unwavering support and understanding during this journey. Your encouragement provided us with the motivation we needed to see this project through.

Lastly, we appreciate the resources and facilities provided by the Engineering Faculty in Bar Ilan University, which were essential in bringing our project to life. This experience has significantly contributed to our growth, and we look forward to applying the knowledge we gained in our future.

Contents:

1 Introduction

This project deals with error-correcting codes.

Error-correcting codes (ECC) enable safe transmission and storage of information despite interference or noise that may occur in memory components or communication channels.
These codes have many applications and uses, from encoding hard drives, CDs, and QR codes, to protecting databases and communication with long-distance spaceships.

The main idea of ECC is that the sender encodes a message, by adding redundancy over the information. The redundancy allows the receiver to detect errors, and even correct a limited number of them.

ECC is based on increasing the difference between valid codewords. If an error occurred, the message received would be significantly different from any other valid codeword, making it easier to identify that an error occurred. The larger the difference between valid codewords is, the more errors can be detected and corrected.

ECCs must be efficient for real-time communication, and capable of detecting and correcting a large fraction of errors, to ensure reliable data transmission.

There are many error-correcting codes in use, a popular one is Reed-Solomon, which can correct the maximum fraction of errors possible, but has a quadratic decoding time complexity.
To improve performances, the data is usually split into smaller blocks. However, this isn't a perfect solution, because when the data is split to blocks, the error correcting capability is negatively affected. For instance, if an entire block is corrupted, it can't be repaired.
Currently, we are not familiar with an implementation of an error-correcting code that has a linear-time encoding and decoding functions, as well as a sufficient error-correcting capability.

There is such code described in the article: *"Linear-Time Encodable/Decodable Codes With Near-Optimal Rate"* [1].
We aim to implement the code in a python package, and publish it on the internet. We will also find the best-suited parameters for the code, and perform simulations – check the time complexity, error correcting capability, and relevancy to everyday use, compared to existing ECC.

The package is available in the Git-Hub repository:
https://github.com/ChaniMil/ECC-linear-time

2 Theoretical Background

2.1 Error-Correcting Codes

Introduction to Error-Correcting Codes

Error-Correcting Codes (ECC) are algorithms designed to detect and correct errors in digital data transmission or storage. The fundamental goal of ECC is to ensure reliable communication over noisy channels by adding redundancy to the transmitted information. This redundancy allows detection and correction of errors without needing retransmission, thereby enhancing the robustness of data communication systems.

Fundamentals of Coding Theory

The theory behind ECC revolves around encoding data in such a way that errors introduced by the channel can be corrected.

Definitions:

Alphabet size (q): the number of distinct symbols used in the code, which corresponds to the size of the finite field $\mathbb{F}_q = GF(q)$ over which the code is defined.

Code dimension (k): denoted by k, the number of information symbols to encode.

Block size (n): total number of symbols in each codeword, including both information symbols and redundancy (or parity) symbols.

Codeword: a sequence of n symbols over a finite field $\mathbb{F}_q$ that represents the original data after applying an encoding function.

Code: a set of codewords, denoted by C, where $C \subseteq \mathbb{F}_q^n$**.**

Code rate (r): the ratio between the number of information symbols (k) to the total number of symbols in the codeword (n). Denoted by $r = \frac{k}{n}$.

Hamming Distance: the number of positions at which two vectors differ.

Code Distance (d): the distance of a code is defined as: $d = min_{c_i, c_j \in C, c_i \neq c_j} d(c_i, c_j)$, where $d(u, v)$ is the hamming distance between two vectors u and v.

Error Detection: the ability to detect whether an error has occurred during transmission. A code with minimum distance d can detect $d - 1$ errors.

Error Correction: the process of identifying and fixing errors. This requires both identifying which bits are incorrect and determining their correct values. A code with minimum distance d can correct $t = \left\lfloor \frac{d-1}{2} \right\rfloor$ errors.

ECC strategies are divided into two major categories:

- **Block codes**: data is divided into fixed-size blocks and encoded individually.
- **Convolutional codes**: the encoding process considers both the current and previous blocks.

Linear Block Codes

A linear $[n, k, d]_q$ code C over $\mathbb{F}_q$ is a subset of $\mathbb{F}_q^n$.

Linear block codes are a key family of error-correcting codes, where the codewords are generated by linear combinations of basis vectors.

These codes have the form: $c = m \cdot G$ where c is the codeword, m is the message, and G is a $k \times n$ matrix - the generator matrix. The algebraic structure of linear codes simplifies both encoding and decoding processes.

The Syndrome-based decoding technique is commonly used for linear codes. In this approach, a vector computed using the received message and the check matrix reveals the location of the error.

Error-Correcting Capacity and Trade-offs

The performance of error-correcting codes is characterized by their ability to detect and correct errors, quantified by parameters such as code rate and error-correcting capability.

There is an inherent trade-off between the error-correcting capability of a code and its efficiency. Codes with a lower rate provide more redundancy and are more capable of correcting errors but require more bandwidth or storage.

Singleton bound:

The singleton bound is a theoretical limit in coding theory that establishes the maximum error-correcting capability of a block code. It states that for a linear [n, k, d] code the following inequality holds: $d \leq n - k + 1$.

MDS code: Codes that achieve the singleton bound are known as maximum distance separable, meaning they can correct the maximum number of errors possible given their length and dimension.

Near-MDS code:

We divide the singleton bound by n:

$\frac{d}{n} \leq 1 - \frac{k}{n} + \frac{1}{n}$, we define the relative distance $\delta = \frac{d}{n}$ and get $\delta \leq 1 - r + \frac{1}{n}$.

MDS code satisfy the equality: $\delta = 1 - r + \frac{1}{n}$. Where n is big we can neglect the $\frac{1}{n}$ and get $\delta = 1 - r$.

Near-MDS code satisfy that $\delta = 1 - r + \varepsilon$, for some arbitrary small $\varepsilon > 0$.

Notice that the smaller ε is, the closer we are to MDS-code.

Channel Models and Error Types

The reliability of any communication system depends on the characteristics of the channel through which data is transmitted. Noise in these channels can lead to various types of errors, categorized broadly into:

- **Random errors**, errors that affect individual bits or symbols in the data stream. They are usually caused by continuous or distributed phenomena, such as thermal noise, crosstalk, or quantization**.**

- **Burst errors**, where sequences of adjacent symbols are affected due to events like fading or interference in wireless communication.

2.2 Reed-Solomon

Reed-Solomon codes are MDS error-correcting codes based on polynomial interpolation over finite fields (Galois fields). They encode data by representing it as coefficients of polynomials, which allows the code to recover the original data from a subset of its evaluations.

The code can correct up to $t = \left\lfloor \frac{n-k}{2} \right\rfloor$ symbol errors.

Time complexity

Encoding: $O(n \log n)$ when using efficient algorithm such as FFT.

Decoding: The Berlekamp-Massey algorithm has a time complexity of $O(n^2)$ for decoding, optimized algorithms can reduce this to $O(n \log^2 n)$.

2.3 Ramanujan Graphs

Expander graphs are a class of sparse but highly connected graphs that play a significant role in computer science, combinatorics, and network theory. These graphs are characterized by their strong connectivity properties despite having relatively few edges compared to the number of vertices.

They have the property that for every small set of nodes S, the set of their neighbors N satisfy that $|N \setminus S|$ is very large.

Expander graphs are suitable for error-correcting codes because their strong connectivity and expansion properties ensure that information spreads widely and evenly across the graph. This makes it difficult for errors to corrupt a large portion of the encoded message without being detected or corrected.

Ramanujan graphs, named after the mathematician Srinivasa Ramanujan, are expander graphs with the special spectral property.

the graphs are d-regular, and satisfy that $\max_{|\lambda_i|<d} |\lambda_i| \leq 2\sqrt{d-1}$ where λ_i are the eigen values of the adjacency matrix.

2.4 Legendre symbol

An element a of $\mathbb{Z}_q$ is called quadratic residue if exists $b \in \mathbb{Z}_q$ such that $a = b^2$.

Let p be a prime number, we say that for $a \not\equiv_p 0$ the legendre symbol $\left(\frac{a}{p}\right)$ is:

1 if a is a quadratic residue modulo p.

-1 if a is a quadratic nonresidue modulo p.

Important property of the Legendre symbol is the multiplicativity.

$$\left(\frac{a}{p}\right) \cdot \left(\frac{b}{p}\right) = \left(\frac{ab}{p}\right)$$

In our project we use primes with Legendre symbol of -1.

2.5 PGL

The group $GL(2,\mathbb{Z}_q)=\left\{\begin{pmatrix}a & b\\ c & d\end{pmatrix}\middle|a,b,c,d\in\mathbb{Z}_q, ad-bc\neq 0\right\}$ are all the invertible matrices with elements of $\mathbb{Z}_q$.

The <u>center</u> of a group is a subgroup of the elements commuting with all the elements of the group.

Denote by $Z(2,\mathbb{Z}_q)$ the center of $GL(2,\mathbb{Z}_q)$.

We will now prove that the center of the group is $\{aI|a\in\mathbb{Z}_q\}$:

$$Z(2,\mathbb{Z}_q)=\{A|\forall B\in GL(2,\mathbb{Z}_q): AB=BA\}$$
$$=\left\{\begin{pmatrix}x & y\\ z & w\end{pmatrix}\middle|\forall\begin{pmatrix}a & b\\ c & d\end{pmatrix}\in GL(2,\mathbb{Z}_q):\begin{pmatrix}x & y\\ z & w\end{pmatrix}\begin{pmatrix}a & b\\ c & d\end{pmatrix}=\begin{pmatrix}a & b\\ c & d\end{pmatrix}\begin{pmatrix}x & y\\ z & w\end{pmatrix}\right\}$$
$$=\left\{\begin{pmatrix}x & y\\ z & w\end{pmatrix}\middle|\forall\begin{pmatrix}a & b\\ c & d\end{pmatrix}\in GL(2,\mathbb{Z}_q):\begin{pmatrix}ax+cy & bx+dy\\ az+cw & bz+dw\end{pmatrix}=\begin{pmatrix}ax+bz & ay+bw\\ cx+dz & cy+dw\end{pmatrix}\right\}$$
$$\subseteq\left\{\begin{pmatrix}x & y\\ z & w\end{pmatrix}\middle|\forall\begin{pmatrix}a & b\\ c & d\end{pmatrix}\in GL(2,\mathbb{Z}_q): cy=bz, bx+dy=ay+bw\right\}$$

For every matrix in the set, Because $\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}$ is in GL, $y=z$.

Also, the matrix $\begin{pmatrix}1 & 1\\ 0 & 1\end{pmatrix}$ is in GL and $y=0$.

from that we get $y=z=0$.

$$Z(2,\mathbb{Z}_q)\subseteq\left\{\begin{pmatrix}x & y\\ z & w\end{pmatrix}\middle|\forall\begin{pmatrix}a & b\\ c & d\end{pmatrix}\in GL(2,\mathbb{Z}_q): cy=bz, bx+dy=ay+bw\right\}$$
$$=\left\{\begin{pmatrix}x & 0\\ 0 & w\end{pmatrix}\middle|\forall\begin{pmatrix}a & b\\ c & d\end{pmatrix}\in GL(2,\mathbb{Z}_q): bx=bw\right\}$$

The matrix $\begin{pmatrix}1 & 1\\ 0 & 1\end{pmatrix}$ is in GL, therefore $x=w$

We got, $Z(2,\mathbb{Z}_q)\subseteq\{aI|a\in\mathbb{Z}_q\}$

Easy to see that $\{aI|a\in\mathbb{Z}_q\}\subseteq Z(2,\mathbb{Z}_q)$, and therefore $Z(2,\mathbb{Z}_q)=\{aI|a\in\mathbb{Z}_q\}$.

PGL is the group defined by

$$GL(2,\mathbb{Z}_q)\Big/Z(2,\mathbb{Z}_q)=\{A\cdot Z(2,\mathbb{Z}_q)|A\in GL(2,\mathbb{Z}_q)\}=$$
$$\left\{\begin{pmatrix}a & b\\ c & d\end{pmatrix}\cdot\left\{e\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}\middle|e\in\mathbb{Z}_q\right\}\middle|ad-bc\neq 0\ (mod\ q)\right\}=$$
$$\left\{\left\{e\begin{pmatrix}a & b\\ c & d\end{pmatrix}\middle|e\in\mathbb{Z}_q\right\}\middle|ad-bc\neq 0\ (mod\ q)\right\}.$$

Notice that one of these happens:

1. All the determinants of the matrices in the element are quadratic residue
2. None of the determinants of the matrices in the element are quadratic residue

Proof:

Assume there is an e such that: $\left|e\begin{pmatrix}a & b\\ c & d\end{pmatrix}\right|$ is a quadratic residue.

That means that there is an x such that,

$$x^2 = \left|e\begin{pmatrix}a & b\\ c & d\end{pmatrix}\right| = e^2(ad-bc)$$

From that we infer:

$$(xe^{-1})^2 = x^2e^{-2} = ad-bc$$

$ad-bc$ is a quadratic residue. Say that $y^2 = ad-bc$

For any other e, $\left|e\begin{pmatrix}a & b\\ c & d\end{pmatrix}\right| = e^2(ad-bc) = e^2y^2 = (ey)^2$

We get that the determinant of the matrix is a quadratic residue.

2.6 PSL

The group $SL(2,\mathbb{Z}_q) = \left\{\begin{pmatrix}a & b\\ c & d\end{pmatrix}\middle|a,b,c,d \in \mathbb{Z}_q, ad-bc=1\right\}$ are all the matrices with determinant 1 and elements of $\mathbb{Z}_q$.

Denote by $SZ(2,\mathbb{Z}_q)$ the center of $SL(2,\mathbb{Z}_q)$.

We will now prove that the center of the group is $\{aI|a \in \mathbb{Z}_q\}$:

$$SZ(2,\mathbb{Z}_q) = \{A|\forall B \in SL(2,\mathbb{Z}_q): AB = BA\}$$
$$= \left\{\begin{pmatrix}x & y\\ z & w\end{pmatrix}\middle|\forall\begin{pmatrix}a & b\\ c & d\end{pmatrix} \in SL(2,\mathbb{Z}_q): \begin{pmatrix}x & y\\ z & w\end{pmatrix}\begin{pmatrix}a & b\\ c & d\end{pmatrix} = \begin{pmatrix}a & b\\ c & d\end{pmatrix}\begin{pmatrix}x & y\\ z & w\end{pmatrix}\right\}$$
$$= \left\{\begin{pmatrix}x & y\\ z & w\end{pmatrix}\middle|\forall\begin{pmatrix}a & b\\ c & d\end{pmatrix} \in SL(2,\mathbb{Z}_q): \begin{pmatrix}ax+cy & bx+dy\\ az+cw & bz+dw\end{pmatrix} = \begin{pmatrix}ax+bz & ay+bw\\ cx+dz & cy+dw\end{pmatrix}\right\}$$
$$\subseteq \left\{\begin{pmatrix}x & y\\ z & w\end{pmatrix}\middle|\forall\begin{pmatrix}a & b\\ c & d\end{pmatrix} \in SL(2,\mathbb{Z}_q): cy = bz, bx+dy = ay+bw\right\}$$

For every matrix in the center, Because $\begin{pmatrix}1 & 0\\ 1 & 1\end{pmatrix}$ is in the group, $0 = z$

Also, the matrix $\begin{pmatrix}1 & 1\\ 0 & 1\end{pmatrix}$ is in the group, $y = 0$

From that, we get $y = z = 0$.

$$Z(2,\mathbb{Z}_q) \subseteq \left\{\begin{pmatrix}x & 0\\ 0 & w\end{pmatrix} \in SL(2,\mathbb{Z}_q)\right\}$$

$$SZ(2,\mathbb{Z}_q) \subseteq \left\{\begin{pmatrix}x & y\\ z & w\end{pmatrix}\middle|\forall\begin{pmatrix}a & b\\ c & d\end{pmatrix} \in SL(2,\mathbb{Z}_q): cy = bz, bx+dy = ay+bw\right\}$$
$$= \left\{\begin{pmatrix}x & 0\\ 0 & w\end{pmatrix}\middle|\forall\begin{pmatrix}a & b\\ c & d\end{pmatrix} \in SL(2,\mathbb{Z}_q): bx = bw\right\}$$

Because the matrix $\begin{pmatrix}1 & 1\\ 0 & 1\end{pmatrix}$ is in SL, $x = w$.

We got $SZ(2,\mathbb{Z}_q) \subseteq \{aI|a \in \mathbb{Z}_q\}$

Because SZ is a subgroup, the matrices must be with determinant 1.

$$SZ(2,\mathbb{Z}_q) \subseteq \{aI|a^2 = 1\}$$

Easy to see that $\{aI|a^2 = 1\} \subseteq SZ(2,\mathbb{Z}_q)$, and therefore $SZ(2,\mathbb{Z}_q) = \{aI|a^2 = 1\}$.

PSL is the group defined by $SL(2,\mathbb{Z}_q)\Big/_{SZ(2,\mathbb{Z}_q)} =$

$\{A \cdot SZ(2,\mathbb{Z}_q) | A \in GL(2,\mathbb{Z}_q)\} =$

$\left\{\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \left\{e\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \middle| e^2 = 1\right\} \middle| ad - bc = 1\ (mod\ q)\right\} =$

$\left\{\left\{e\begin{pmatrix} a & b \\ c & d \end{pmatrix} \middle| e^2 = 1\right\} \middle| ad - bc = 1\ (mod\ q)\right\}.$

Notice that all determinants are quadratic residue:

$$\left|e\begin{pmatrix} a & b \\ c & d \end{pmatrix}\right| = e^2(ad - bc) = 1 \Longrightarrow quadratic\ residue$$

3 Problem Formulation

In this project we aim to implement a fast error correcting code – that has linear-time encoding and decoding algorithms.
In addition, it is a near-MDS code, meaning it has near optimal trade-off between the amount of redundancy added and the error detection and correction capability.
Near-MDS codes can correct a very high fraction of errors: $\frac{1-r-\varepsilon}{2}$, for arbitrarily small $\varepsilon > 0$.

An explicit construction of such code is brought in the paper:

Linear-Time Encodable/Decodable Codes With Near-Optimal Rate
by Venkatesan Guruswami and Piotr Indyk [1]

We did not find implementations of this construction online, and so we decided to implement it by ourselves.

The goal of the project is to develop a package in python of the encoding and decoding functions, and run simulations to investigate the code – what are its limitations, and whether it can replace existing error correction codes, while achieving better performances.

For example, the running time might be linear, but with very large constants, meaning other algorithms could perform faster for most values of n. This is something we want to verify. Additionally, we want to examine how small the ε values can be, what constraints affect them, and the conditions necessary to achieve such small values.

In addition, the article does not mention explicitly how to choose the parameters in the code. We want to explore it, find the relations between the parameters and their typical sizes.

Finally, we want to publish the package, so everyone can explore it, and use it for their own purposes.

4 Code Overview

The code consists of three main parts: Left code, MDS on blocks, Expander Shuffle.

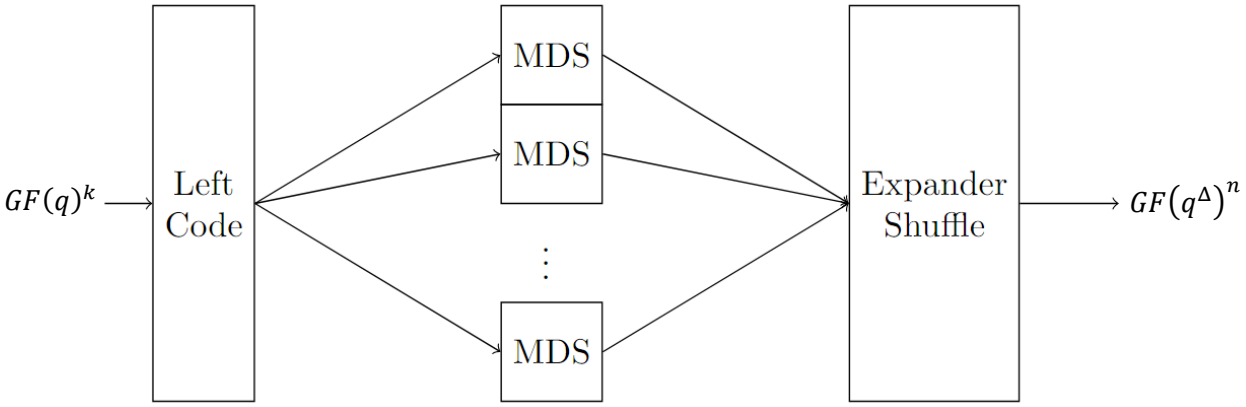


Figure 1: Code scheme

Our code is constructed by combining three objects, a left code C, a constant-sized MDS (Reed–Solomon) code $\widetilde{C}$, and a suitable bipartite expander graph G.
The message will be first encoded by the left code C. The resulting codeword of C will then be broken into blocks, each of constant size, and each of these blocks will be encoded by the Reed–Solomon code $\widetilde{C}$. The symbols of the resulting string will then be redistributed using the edges of the expander G.

We now elaborate a bit on each part of the construction.
First, we look at the Left code:

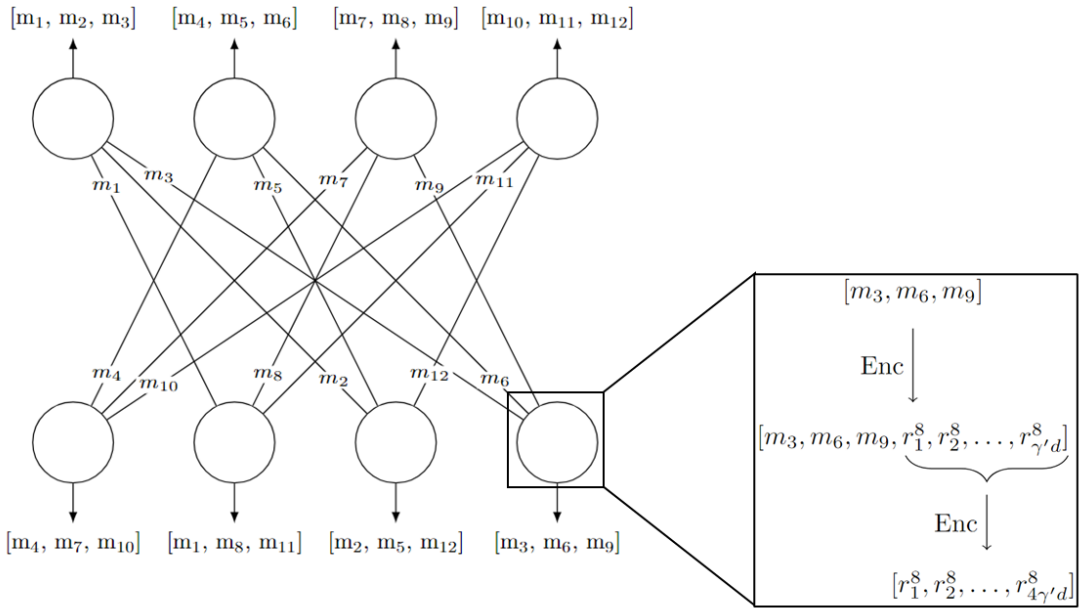


Figure 2: Left code scheme

The algorithm is as follows:

We create a bipartite d-regular Ramanujan Graph. It has k edges and $2N$ nodes.
We assign each edge with a symbol of the word we want to encode.
Now we look at the nodes, every node has d edges connected to it, meaning it holds d symbols of the word.
For every node, we take this small word, of length d, and encode it with systematic Reed-Solomon with rate $\frac{1}{1+\gamma'}$.
Next, we take each encoded node, and encode it's redundant symbols, again with Reed-Solomon, but with rate $\frac{1}{4}$. Note that in the article Reed-Solomon isn't used in this part, but another linear-time code that exists due to Spielman, which can correct a fraction b of errors for some absolute constant $b > 0$, independent of γ.
Eventually we take the encoded redundancy from every node and concatenate it to the original word. This is the output of the left code C. We denote its length by n'.

The next component is the MDS on the blocks.
Here, we take the output of the Left Code – the n' symbols, and break them into n blocks of constant size b.
Every block will be encoded with Reed-Solomon with rate r'.
Eventually we get n blocks of Δ symbols. This is the output of the MDS on the blocks.

Finally, we reach to the Expander Shuffle.

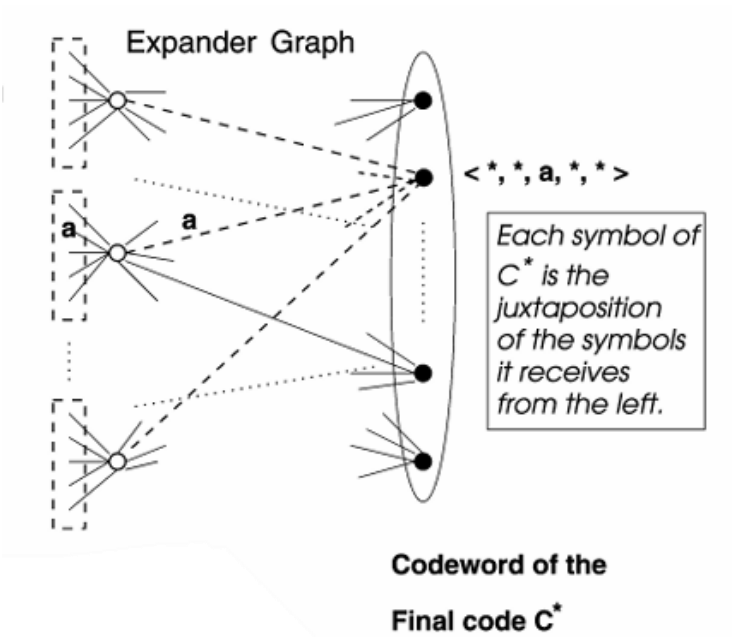


Figure 3: Expander shuffle scheme

In this part we create a bipartite expander, we use Ramanujan Graph. It is Δ-regular and it has $2n$ nodes. We assign every block to a node on the left side of the Graph.
Next we assign each symbol from the block to an edge in its matching node.
For example, in the figure above, the second symbol of the encoded block ('a') is sent

to the second neighbor of the corresponding node of the expander.
On the right side of the graph, the symbol at each position is the juxtaposition of the symbols received from the neighbors on the left.
For example, in the figure the second block receives 'a' from its third neighbor on the left, and therefore has at the third position of the 5-tuple of symbols that it receives.

5 Implementation

5.1 Packages in use

We used a few external packages in our implementation, we'd like to give credit and explain them briefly.

reedsolo [2]

The reedsolo package on PyPI is a Python implementation of Reed-Solomon error-correcting codes.
We chose to use this package rather than other RS packages since it enables the user to determine the block size by themselves, and supports any Galois field higher than 2^3.
In addition, it provides systematic encoding – which is needed for our construction.
The reedsolo library's flexibility and robust implementation played a key role in ensuring the integrity and reliability of our code.
The functions we used are 'encode' to encode a message, and 'decode' to decode a message.

galois [3]
The galois package on PyPI is a Python library for performing computations in Galois fields.
The library provides tools for defining Galois fields, performing arithmetic operations, and working with polynomials, matrices, and other algebraic structures within these fields. It's optimized for high performance and is easy to use.
We performed the division and multiplication of elements using this library in the generation of the Ramanujan graph.

We also used numpy, matplotlib, collections, time and random, but these are popular and standard libraries, so we will not elaborate on them here.

5.2 Pseudo code

Before implementing properly in python, we wrote a pseudo code, which gives a brief understanding of the implementation.

ENCODING

input: word in $GF(q)^k$

output: codeword in $GF(q^{\Delta})^n$ // $\Delta = \frac{b}{r\left(1+\frac{\varepsilon}{4}\right)}$, $n = \frac{\left(1+\frac{\varepsilon}{4}\right)k}{b}$

g ← generate_bipartite_ramanujan(nodes=2k/d, degree=d)

assign each edge with a symbol (letter) from the word

define for each v: $E_v \leftarrow \{(x,v) \in E | x \in V\}$

for each v in nodes:

v_word ← concatenate the symbols of E_v // size d

compute the check symbols of v_word with systematic RS // size γ'd

encode the check symbols with linear time code with rate 1/4 // size 4γ'd

concatenate all words of all the nodes with the information word // size (1+γ)k=n'

split the word to n blocks of constant size b // $nb = n'$

encode each block with RS // n blocks of $\frac{b}{r\left(1+\frac{\varepsilon}{4}\right)} = \Delta$

g ← generate_bipartite_expander(nodes=2n, degree=Δ)

for each node v, N_v ← g.neighbors_of(v)

for each node on the left side:

for $1 \leq i \leq \Delta$ send to the ith neighbor its ith symbol of the corresponding block

for node $1 \leq j \leq n$ on the right side:

codeword[j] ← concatenate the Δ symbols the node got

return codeword

DECODING

input: codeword in $GF(q^{\Delta})^n$ // $\Delta = \frac{b}{r\left(1+\frac{\varepsilon}{4}\right)}$

output: word in $GF(q)^k$

g ← generate_bipartite_expander(nodes=2n, degree=Δ)

transform each of the symbols into a block of Δ smaller symbols over GF(q)

for each node v, N_v ← g.neighbors_of(v)

for each node on the right side:

for $1 \leq i \leq \Delta$ send to the ith neighbor its ith symbol of corresponding block

for each node on the left side:

concatenate the Δ symbols the node got

Decode each block with RS

concatenate all words of all the nodes

g <- generate_bipartite_ramanujan(nodes=2k/d, degree=d)

assign each edge with a symbol (letter) from the corrupted word

define for each v: $E_v \leftarrow \{(x,v) \in E | x \in V\}$

for each v in nodes:

y_{Ev} ← decoding of the check symbols // linear time code with rate 1/4

$x_{Ev} \leftarrow [e.symbol() | \exists u: e = (v,u)]$

A ←left nodes
B ← right nodes

while x contains errors:

for v in A:

if exists z s.t $|z - x_{Ev}|$ < γ'd/2 and its check symbols are y_{Ev}:

$x_{Ev} \leftarrow z$

for v in B:

if exists z s.t $|z - x_{Ev}|$ < γ'd/2 and its check symbols are y_{Ev}:

$x_{Ev} \leftarrow z$

return the symbols on all the edges

5.3 Ramanujan graph

In the generation of the Ramanujan graph there are two options for the construction: deterministic algorithm or probabilistic algorithm, both of them have advantages and disadvantages.

If we construct the graph using the probabilistic construction we might have different graphs in the encoding and in the decoding, causing the decoding to be wrong, and to decode to an incorrect codeword.
Therefore, if we choose to use the probabilistic construction, we must generate it prior to both encoding and decoding, and find a method to transfer this information, which makes the process less user friendly and simple, and less independent.

For the deterministic algorithm, there is a paper describing a deterministic and linear time construction of Ramanujan graphs.
The problem with this construction is that it is very limited to its degree and size.

After thinking what will work the best for our project we came to conclusion that we will construct the graph using the deterministic algorithm. The user also has the option to manually input graphs, which could be a randomly generated graph, a graph created by another implementation, or a pre-generated graph to save time when encoding multiple messages.

Construction of deterministic algorithm [4]
Let p, q be unequal primes congruent to 1 mod 4.
Let i be an integer satisfying $i^2 = -1\ (mod\ q)$.
Denote S as the set of matrices

$$\begin{pmatrix} a_0 + ia_1 & a_2 + ia_3 \\ -a_2 + ia_3 & a_0 - ia_1 \end{pmatrix}$$

such that $a_0^2 + a_1^2 + a_2^2 + a_3^2 = p$, a_0 is odd and positive, and a_1, a_2, a_3 are even.
There are $p + 1$ such matrices.

We will recall the group, $PGL(2, \mathbb{Z}_q) =$
$$\left\{ \left\{ e \begin{pmatrix} a & b \\ c & d \end{pmatrix} \middle| e \in \mathbb{Z}_q \right\} \middle| ad - bc \neq 0\ (mod\ q) \right\}.$$

Notice that if $a \neq 0$ then by choosing $e = a^{-1}$ there is a matrix in each element of the PGL such that the first element is 1. If $a = 0$ then $b \neq 0$ and by choosing $e = b^{-1}$ there is an element in the matrix such that the first value is 0 and the second is 1.

We now build a Cayley graph using the PGL and S, the nodes will be the elements of PGL.
For each x, y nodes of the graph, there is an edge between them if $x = sy$ for some $s \in S$.

If $\left(\frac{p}{q}\right) = -1$, the nodes are splitted between the 2 sides according to the group PSL,

$$PSL(2, \mathbb{Z}_q) = \left\{ \left\{ e \begin{pmatrix} a & b \\ c & d \end{pmatrix} \middle| e^2 = 1 \right\} \middle| ad - bc = 1 \ (mod\ q) \right\}$$

As we said, the group matrices are all quadratic residue over the field of size q.

We will prove now that PSL splits the nodes correctly into 2 sides.
Let s be an element of S,

$$s = \begin{pmatrix} a_0 + ia_1 & a_2 + ia_3 \\ -a_2 + ia_3 & a_0 - ia_1 \end{pmatrix}$$

then $|s| = a_0^2 + a_1^2 + a_2^2 + a_3^2 = p$, therefore, $|s|$ is not a quadratic residue.
Let's take x, y matrices from PGL, such that there is an edge between them.

$$x = sy \Longrightarrow |x| = |s| \cdot |y|$$

If $|x|$ is a quadratic residue, then $1 = \left(\frac{|x|}{q}\right) = \left(\frac{|s| \cdot |y|}{q}\right) = \left(\frac{|s|}{q}\right)\left(\frac{|y|}{q}\right) = -1 \cdot \left(\frac{|y|}{q}\right)$
therefore, $|y|$ is non-quadratic residue.
If $|x|$ is non-quadratic residue, then $-1 = \left(\frac{|x|}{q}\right) = \left(\frac{|s| \cdot |y|}{q}\right) = \left(\frac{|s|}{q}\right)\left(\frac{|y|}{q}\right) = -1 \cdot \left(\frac{|y|}{q}\right)$
therefore, $|y|$ is a quadratic residue.
We got that the only edges in the graphs, are between quadratic residue, and non-quadratic residue.

Note that this construction is also imperfect, there's a finite number of primes for which the construction provides a graph that is not Ramanujan.

Implementation

To begin with, we will discuss about the class 'Graph':

Its properties are: degree, number of edges, number of nodes, two sets to tell us which nodes are on which side, and a list of lists for the neighbors of each node.

We initialized the list of the neighbors with d-length list in each cell, and created a list of counters *num_of_neighbors*, in order to know where to set the next neighbor of the node and save some time.
We created a function *add_edge* that on input of 2 nodes, it adds the edge between them to the graph.
After that, we expect the function *split_to_sides* to be called, this function purpose is to create 2 lists for the graph, one of the nodes on the left, and the other for the nodes on the right.

Now for the Ramanujan generation:
First we found an i and the p+1 elements, then we computed the set S.

For the elements of PGL we chose matrices to represent them, if the element contains a matrix with 1 in the first location it will be the matrix to represent it,

otherwise there exists a matrix with 1 in the second location that will represent it.
We created a function *PGL* to yield all representatives.

And for the main part of the construction, to save time later we stored all multipications and all inverses in lists. In addtion we created a boolean list to know which of the field elements are quadratic residue.
Then for our main loop, we ran over all PGL elements using the function *PGL*, ran over all S elements, computed $x = s \cdot y$, computed the representative of x, transformed both of them into integers and added the edge to the graph.

During the work, many things have changed and improved, we would like to recall the challenges we overcame, and the decisions we made along the way.

History of optimizations

Our first construction was probabilistic, we started by creating a random bipartite d regular graph using the python library networkX, which is a library to handle graphs.
We generated a random mapping between the nodes on the left side and the nodes on the right side, if all the items in the mappings aren't already edges in the graph, we add all these edges to the graph. This is repeated d times.
Finally we check if the graph is Ramanujan.
The problem with this code is that most of the time it didn't finish, the probability of finding d distinct permutation was very low, we tried different approaches, but decided to let it go, since, as we said earlier we prefer deterministic algorithm.

To solve these problems, we decided to go with the deterministic algorithm, the first version of the construction still used the networkX library, and now also used the library galois, which handles field operations like multiplication and division.
This library helped us to find the representative of an element.

After this change we have a functioning code to construct Ramanujan graphs, the problem now is that it is extremely slow.

We searched for the bottlenecks, and found two major issues:
1. In the galois library, every multiplication and division takes a lot of time.
2. The implementations of creating graphs in networkX wasn't done in linear time – before adding a new node or an edge, the functions verify they are not in the graph already.

In order to handle the first problem, we created two lists: a list that saves all the multiplications of elements in the field, and another list to save all the inverses of the elements in the field.
Now, instead of multiplying the elements of the field many times, the multiplication

happens once, and we just get the results of the multiplications and the inverses by accessing the elements in the lists.

In order to handle the second problem, we created our own class for graphs '*Graph'*, most of the properties in it are standard and similar to the networkX library.
But, in the new version the nodes are represented by integers, since we wanted to approach every node using an index, rather than a matrix, to save complications.
The code was programmed this way already, since networkX has a function that transforms all the nodes to integers.
So, we created the function *matrix_to_int* which converts a representative into an integer in the range $\{0, \dots, (q-1)(q+1)^2\}$.
There are only $q \cdot (q-1)^2$ nodes, so we needed to get rid of empty slots in the integer neighbors list.
We created a function that handles it, this was much faster than the previous version, but not enough.

In our class we couldn't use the function from networkX to split the nodes to two sides, we needed to create our own, the straight forward approach was slow, so we searched for another solution.

We found out that the nodes are splitted by PSL, so we checked for every representative if their determinant is a quadratic residue to determine the side of the node.
Because we didn't want to check each time if the element was quadratic reside, we created a Boolean list to check for each element of the field if it's a quadratic residue, and in the main loop we just needed to access this list.

Finally, in our last version, after understanding that the *remove holes* function takes too much time, we decided to remove the holes one step earlier.
Instead of sending the values and squeeze it in the graph class, we created a list of length $(q+1)^2(q-1)+1$ and for the $(q^2-1)q$ values we can get from the matrix_to_int function, we gave them the values $0, \dots, q(q^2-1)$.
And then to add the edges we just need to convert the matrix into an integer index and access the list in that index location.

All these optimizations had an enormous effect on the running time, to get an idea, just the last optimization shortened the time by half (i.e. 700s to 350s). Accessing the multiplications list instead of multiplying elements from the finite field improved the running time by two orders of magnitude, (i.e. 50s to 0.2s).

5.4 Left code

The encode and decode functions for the left code are located in the file left_code.py.

Encoding

According to the article, in this part there are few stages.
First, we create a Ramanujan graph and match its edges with the symbols of the word.
After that, every node will take the symbols on its edges, concatenate them, and encode them using a Reed-Solomon code with redundancy of $\gamma' d$.
The last part of the encode procedure is to encode each of the redundancies with Reed-Solomon of rate 1/4 and concatenate the original word with all the encodings.

Implementation:
In the beginning of the function we create a list for each node, we ran over all edges (a, b) of the Ramanujan graph and all word symbols, and we append the symbol to the lists of a and b.

Then we use the library 'reedsolo' to encode the lists adding the redundancy.
Then, use this library again to encode the redundancy (the encoding from the d symbol) with rate of 1/4.

Decoding

The inputs to this function are: k symbols of the edges, and 2N codewords of the nodes.
The decoding procedure starts by decoding the redundancies of each of the codewords of the nodes, with RS code of rate 1/4 decoder.

Now each of the nodes has the check symbols of the encoding of its word.
We create a linked list of the nodes to run over, initially it contains all the left side nodes, and in each iteration it's updated.

Because we changed the word on those edges, the neighbors of this node need to check if they are able to be decoded, we first check if the neighbors are already in the linked list using a Boolean list we initialized earlier, and if not we append them to the list.

For every node, we check if concatenating the symbols incident on its edges with its check symbols, can be decoded such that the check symbols are correct.
If so:
1. update the symbols incident on its edges according to the word we decode.

2. append its neighbors to the linked list of the next iteration – the edges connected to those nodes have been updated, so the other side needs to check if it can decode.
3. mark the node as decoded – in order for the decoding to be in linear time, we need to check only for the nodes we didn't already correct.

Notice that every time the linked list contains nodes from the other side.

After there are no changes in the word the algorithm ends and returns the word - the k symbols on the edges.

History of optimizations

In the paper, the algorithm of the decoding that was presented to us has time complexity of $O(n \log n)$, the authors refer to another paper which contains linear time decoding.
In the other paper, we are instructed to decode in the same way as the original paper mentioned, but to change each node only once, meaning if the decoding is successful, we will not try to decode again later.

For our first version of the code, we created a boolean list for the nodes to tell us which nodes have already been decoded. Then we ran over all nodes and skipped the ones we already decoded.
This code worked but was quite slow, so we improved it:
We decided to use a linked list to run over all elements and remove them with $O(1)$, that way we can continue to the next undecoded node instead of checking each node if it's decoded.
With that improvement we brought the complexity down to $O(n)$.

After that, in our next version, we changed the decoding so that it is more similar to the decoding in the article.
We did as written in the paper, running over all the left nodes and then over all the right nodes.
We initiated the linked list with the left nodes, and if the decoding of the node is successful, its neighbors are appended to a new linked list.
The next iteration (of the right side) will run over the new linked list, the neighbors of the successfully decoded nodes of the left side, and so on.
This improvement prevents checking a node if its edges didn't change, and saves a lot of time.
The problem now is that a node can be called more than once for the next iteration (if there's a node with more than one neighbor that was decoded correctly), which can take a lot of time. To prevent this, we initiated a boolean list to tell us which nodes are already in the neighbors linked list. If the node is already in the neighbors linked list, we skip to the next node.

We were unsure how to approach decoding with erasures, since there were several options.
The first debate was when decoding the redundancy. In case it fails to decode and there are erasures – should we consider the entire node as invalid, or pick the information part from the systematic encoding as is and move on.
The second debate was when we check if the redundancy is correct – should we check only the part of the redundancy which is not erased, or skip completely the nodes with erased symbols.
Eventually, after trying a lot of combinations we read the paper again and found out that we get rid of the erasures earlier and we struggled for nothing, we learnt a valuable lesson.

To sum up, we had a lot of changes in the decoding of this part, each improved the running time, after all these changes we believe our code is close to its optimal speed.

5.5 MDS on blocks

This part is implemented on the main encode and decode functions.
In this part we divide the word to n blocks of constant size b, and encode each block with Reed-Solomon with rate $\frac{b}{\Delta}$.
The decoding is similar – decode each block with Reed-Solomon.

Encoding

The parameters needed for the encode (the number of blocks – n, the size of each block – b, and the size of each block after the encoding procedure – Δ) are determined outside the encoding procedure, and are given as parameters to the function.

We split the word into n blocks of size b.
Then we check if padding is necessary, by going through all the blocks, and making sure they're of size b. If there's a block that has a different length, it is padded with zeros, and empty blocks are added until we have n blocks with size b.

Eventually, we encode each block with Reed-Solomon with rate $r' = \frac{b}{\Delta}$.

Decoding

In the decoding we simply go through all the blocks, and decode them with the Reed-Solomon decoder. Note that in case there are erasures, we transfer their location to the decoder of Reed-Solomon, after it has been updated by the expander graph decoder.

In case the decoding is false, we just take the first b symbols, since the encoding of each block is systematic, the left-code decoder will try to correct them.

5.6 Expander shuffle

The encode and decode functions for the expander are located in the file expander_code.py.

In this part we create bipartite Δ-regular Ramanujan that has $2n$ nodes. We assign the n blocks to the n left nodes, each symbol on the block is transferred to an edge connected to its matching node.

Encoding

The function gets the bipartite Ramanujan graph, and the blocks – list of lists that holds the symbols.

The output is the final codeword, as a list with length n that contains sub-lists with length Δ.

The shuffle process is made of a loop that goes through all the left nodes.
In each iteration we handle one block.
First we sort the block, so that the encoding and decoding processes will match (the size of each block is constant, so the time complexity is still linear).
Then we go through the Δ symbols in the block.
We send each symbol to its connected node in the right side, and then update the number of symbols in the right side node, so that next time we will update the next symbol – making sure that the ith edge is the ith symbol (since we sorted before).

Decoding

The decoding is exactly the same, instead of transferring from left to right, we transfer from right to left.

Another thing we handle is the erasures, along with updating the symbols, we are updating the locations of the erasures, that are now different.

We return the word, and the new location of erasures.

6 Parameters

The process of choosing the fitting parameters for our code was very challenging. It was difficult to understand the relation between all the parameters, and the constraints regarding to each parameter.

Explanation

We first go through every parameter and explain its meaning, then we will go over the relation between them.

- k – the code dimension, message length
- q – the alphabet size of the code
- n – the block length of the code
- r – the code rate
- ε – the distance from optimal MDS code
- d – the degree of Ramanujan graph
- N – the number of nodes in the Ramanujan graph
- γ' – determines the rate of $C1$
- γ – determines the rate of Left code.
- n' - the message length after Left code.
- b – block length of the MDS stage
- r' - the rate of the MDS code
- Δ – the length of encoded block, degree of expander graph
- β – decoding capability of the left code

The flow of parameters in the code (scheme follows):

1. Start with a message over $GF(q)$, of length k.
2. Create a bipartite Ramanujan graph, with k edges, $2N$ nodes, and degree d. Choose p, q that are primes equivalent to 1 mod 4, with legendre symbol -1. The graph has degree $d = p + 1$, and number of nodes $2N = q(q^2 - 1)$.
3. Encode d symbols to $d(1 + \gamma')$ symbols.
4. Encode $d\gamma'$ symbols to $4d\gamma'$ symbols.
5. Encoded message of length: $k(1 + \gamma) = n'$
6. Divide to n blocks of constant size b.
7. Encode each block with RS with rate $r' = b/\Delta$.
8. Create a bipartite expander (Ramanujan), with $2n$ nodes and degree Δ. Choose p, q that are primes equivalent to 1 mod 4, with legendre symbol -1. The graph has degree $\Delta = p + 1$, and number of nodes $2n = q(q^2 - 1)$.

This can also be demonstrated in the following scheme:

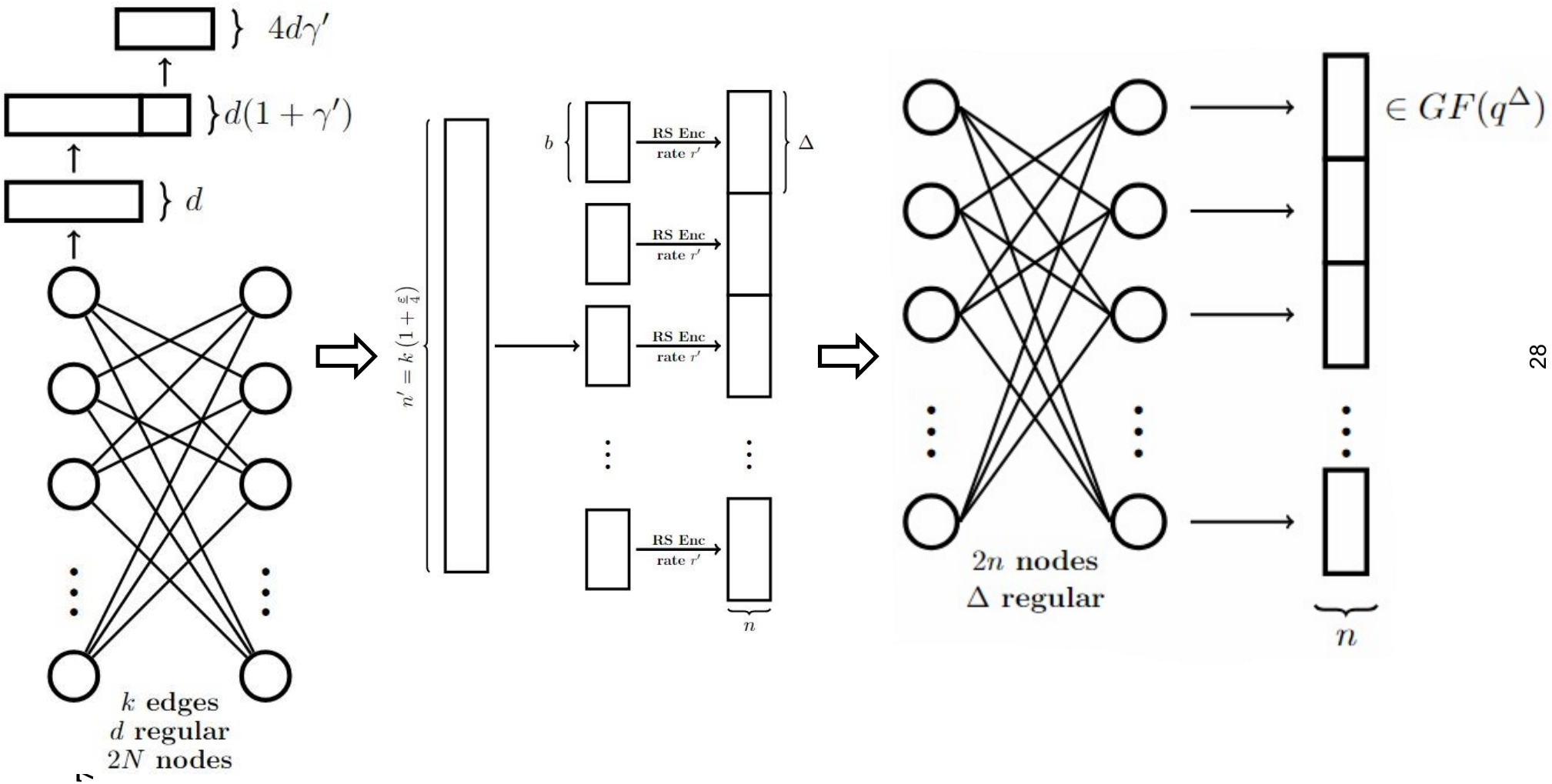


Figure 4: code scheme with parameters

We made a table to try to make sense of all this information, and understand the relations:

Code stage	Length	Can correct	Parameters	Constraints
Ramanujan graph	Degree d $\frac{2k}{d}$ nodes k edges		k, d	$k > d$ $d \mid k$ p, q primes eq. 1 mod 4, legendre symbol -1. $d = p + 1,$ $\frac{2k}{d} = q(q^2 - 1)$
C1 – to all nodes	Start with d and get $d(\gamma' + 1)$	Can correct less than $\frac{\gamma' d}{2}$ errors	$\gamma' = \frac{\gamma}{8} = \frac{\varepsilon}{32}$	To correct more than one error: $\frac{2}{d} < \gamma'$
C2 – to all nodes	Starting from $\gamma' d$ and get $4\gamma' d$	Can correct up to βk errors	d, γ'	$\beta < \frac{3d\gamma'}{2k},$ $\beta \leq \frac{\gamma'^2}{2}$
Divide the word	A word of length n', divide to n blocks of size b			b constant, $n = \frac{n'}{b}$
Reed-Solomon on the blocks	Starting with b, and get $\Delta = \frac{b}{r\left(1+\frac{\varepsilon}{4}\right)}$ meaning $r' = r\left(1 + \frac{\varepsilon}{4}\right)$	Can correct up to $\frac{\frac{b}{r\left(1+\frac{\varepsilon}{4}\right)} - b}{2}$	b, ε, r	$\frac{b}{r\left(1+\frac{\epsilon}{4}\right)} - b > 2$ $0 < r < 1$ $\epsilon > 0, \quad small$
Expander Graph	Degree Δ $2n$ nodes nd edges		Δ, n, d	$\Delta = \frac{b}{r'}$

Table 1: code parameters

We still didn't feel like we fully understand it, so we decided to get more hands-on. We selected some parameters and tested whether they met all the constraints. However, this approach was too complex and didn't work out. So, we shifted our strategy, turned to the Python code, and wrote a function to search for suitable parameters.

All the functions are in the file 'parameters.py'.

The first function we wrote is *choose_params_by_code_dimension*.

This input is the code dimension, and it returns the sizes of the graphs (note that both graphs have the same number of nodes, we will explain why later on), and b, r, epsilon, and the matching m. If we didn't find matching parameters, we increase the code dimension.

The way of operation:
We first find the closest number to k that has a Ramanujan graph with at least k edges.
Next we define d and N by the properties of the graph found.
The main thing in the function is the way we defined epsilon – we need $\frac{\varepsilon}{32} \cdot d = \gamma' \cdot d$ to be whole, since we added $\gamma' d = \frac{\varepsilon}{32} \cdot d$ check symbols in $C1$. So we made sure this is satisfied, and set it to be around 0.25.
After that, n', n, b are determined by the relation with the found parameters.
Then we choose fitting parameters for the expander graph, note that q is the same, so we only need to find p.
We use the relation between the parameters, and demand the constrains:
$\Delta > b, r + \varepsilon < 1$.
If we didn't find fitting parameters, we try again with bigger code dimension.
Eventually we return the parameters found.

Why q is the same?
After the left code there are $n' = k(1 + \gamma) = n \cdot b$ symbols. So a graph with n nodes would fit.
After the MDS on the blocks, there are $n \cdot \Delta$ symols.
A graph with n nodes would fit here as well (with a different degree).
Of course, there are more possibilities, but this was easiest to find.

This function helped us check the correctness of the encoding and decoding functions, since we finally have a set of parameters to insert, but it wasn't user convenience (can't control the rate or epsilon), and wasn't helpful for simulations of running time, since each code dimension has different rate.

We tried writing more functions, that got different inputs, some of them were helpful, and some not.

We will present two more functions: *choose_params_by_exact_rate_epsilon*, and *choose_params*.

These three functions were the most used and relevant in our project.

The second function we present is *choose_params_by_exact_rate_epsilon*.

This function gets epsilon, rate r, and the option to allow or not padding, its default is false.

This function must get r and epsilon that were already found by choose_params, but then it gives a list of more parameters for n, k and b that can be encoded with the r and epsilon given.

The goal of this function was to help us with the simulations of the running time, but it usually outputted no more than 8 sets, not enough to test the time.

For more details about the way it's done, you can check out the function in the file 'paemeters.py'.

The last function we present, and the one used the most both for the experiments and for the user is *choose_params*.

This functions inputs are r, epsilon, the distance from r and the distance from epsilon.

This time r and epsilon don't have to be exact, it generates parameters that are relatively close to the r and epsilon requested by the user, according to the distance specified.

Eventually it returns a list with all the parameters generated.

For more details about the way it's done, you can check out the function in the file 'parameters.py'.

7 Experiments and Results

To evaluate both the running time and the error correcting capability performance of the code, we conducted simulations using different parameters, and injected varying fractions of errors.

7.1 Running Time

To test the running time, we encoded messages of different sizes and measured both the encoding and decoding times.

We aimed to keep r and ε constant so that the essential code parameters unchanged.

This was a major challenge, the sizes of the graphs are very limited, and the parameters relations are quite complicated, it's not trivial to increase or decrease the size while maintaining the same parameters.

We attempted to write additional functions to determine suitable parameters, but encountered a problem: there were not enough parameters found – data points for the graph. With only about five measurements, we couldn't prove linear running time. Even when we managed to obtain more points, the graph and word size became so large that memory limitations in the program prevented further testing. After multiple attempts to resolve this issue, we decided to compromise by using slightly different r values within the same 0.1 range, and similarly close but not identical ε values.

Once we had a sufficient number of parameter sets, we measured the encoding and decoding times and plotted the results:

Encoding:

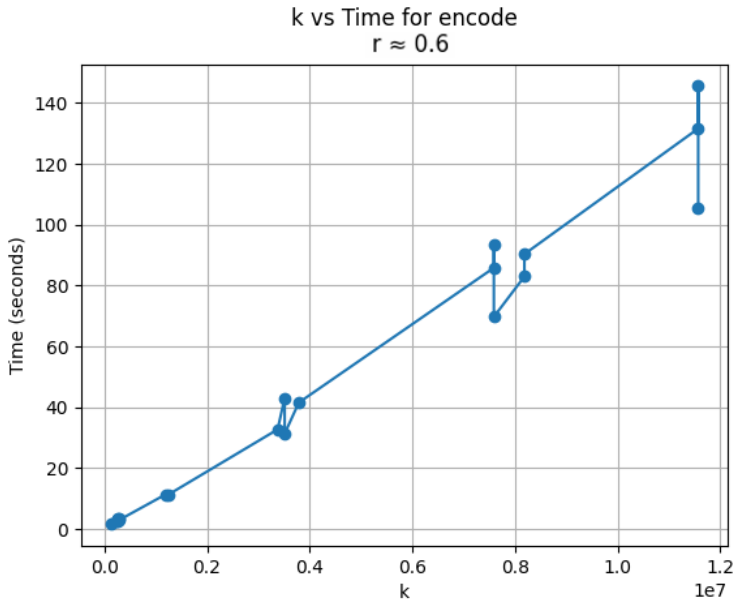


Figure 5: Plot of k vs time for encode, r≈0.6

In the plot above, we can see that in some ks there are multiple measures of times, the reason for this is that the parameters found include identical ks, but different rate (different n).

We removed the redundant ks, and plotted the graph again.

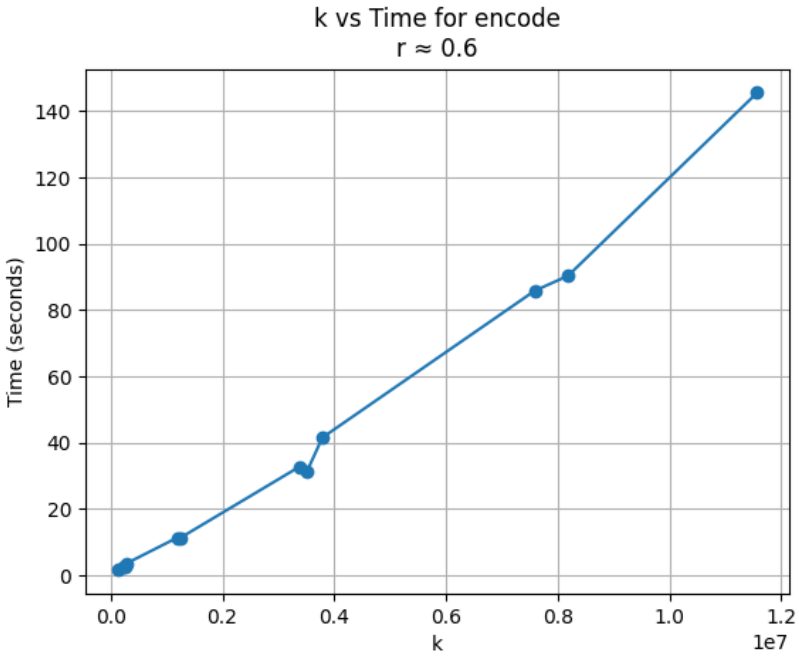


Figure 6: Plot of k vs Time for encode, remove redundant ks, r≈0.6

Then, we moved to the decoding, plotted the graph:

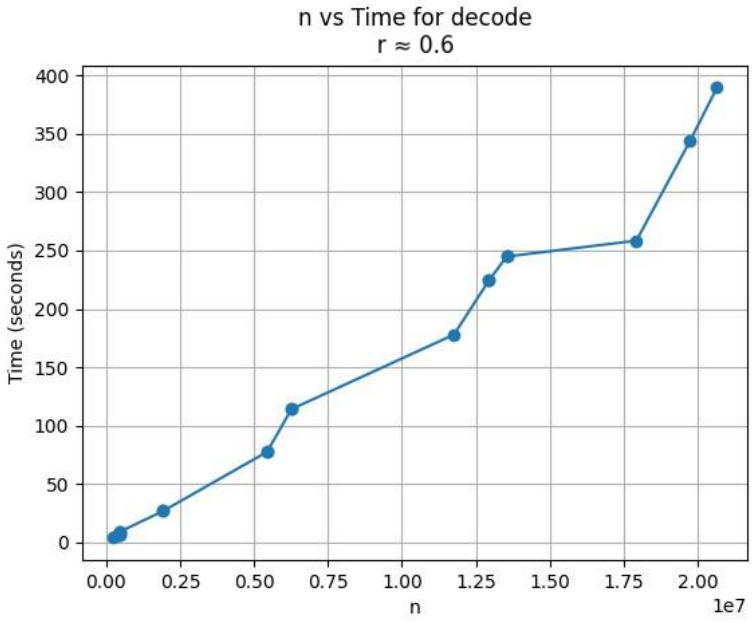


Figure 7: Plot of n vs Time for decode, r≈0.6

Note that n is in scale of $10{,}000{,}000$ bytes. Additionally, the n is actually $n \cdot \Delta$, so that the difference between different Δs is expressed.

In addition, we measured the decoding times after injecting $\frac{1-r-\varepsilon}{2}$, to verify the linearity.

These are the results:

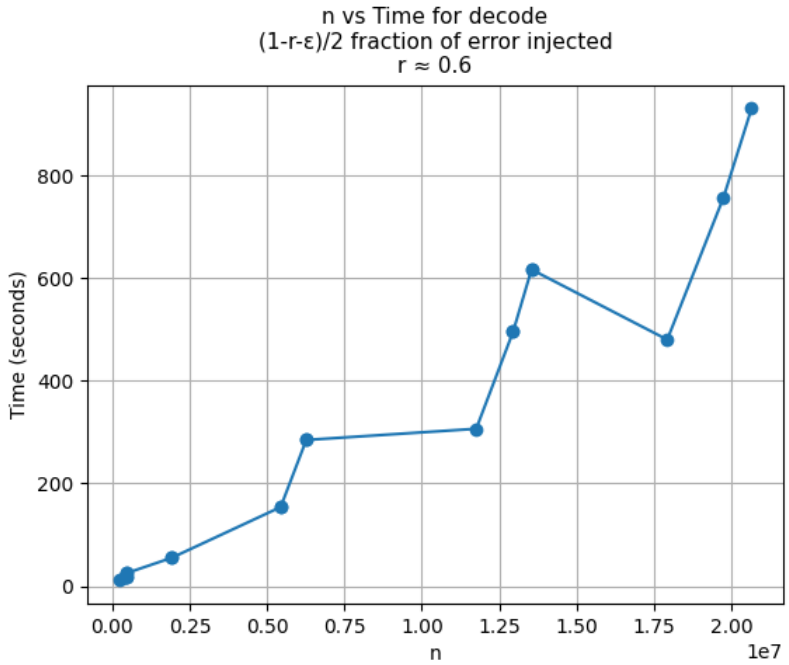


Figure 8: Plot of n vs Time for decode with errors, r≈0.6

This does not seem linear, but it seems there are three different groups that do form a linear result. We checked what is the rate of each such group, and found out that they have very close rs, as demonstrated:

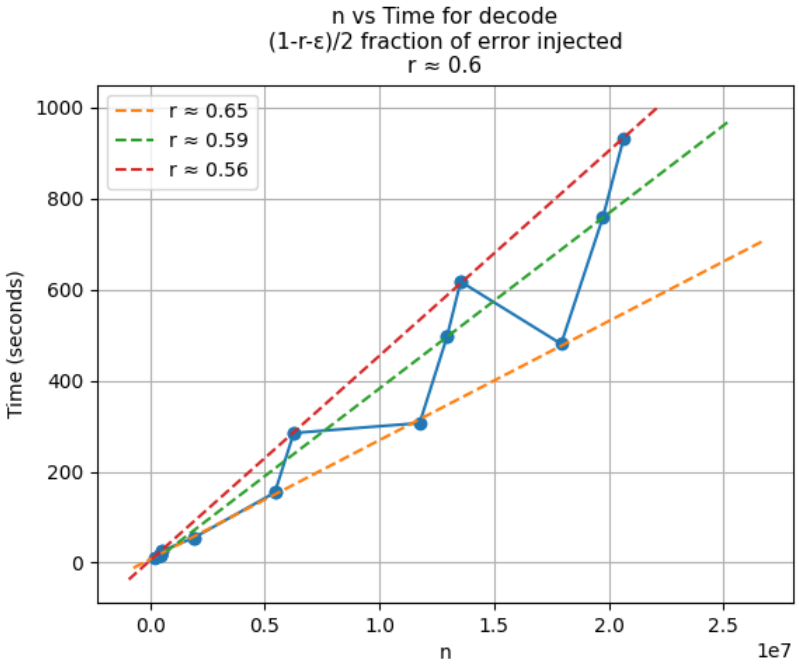


Figure 9: Plot of n vs Time for decode with errors injected, r≈0.6, grouped results

When we look back at the results now, it is clear there have always been three groups.

To conclude:

1. The time complexity experiments shouldn't be conducted on parameters with highly different rate, when the rate differs by ≈ 0.01, the time complexity is linear.
2. The graph does not change much by injecting errors – this gives more certainty to the results.
3. There are not enough parameter sets that allow certainty in the results.

It's important to note that running time experiments are generally imprecise. While the behavior may appear linear for smaller values, it could potentially become exponential for larger ones.
The formal method for determining time complexity is through algorithm analysis rather than empirical testing.
In the article there is such analysis, we believe that it applies to our implementation as well, but can't say for certainty, because there are some conditions that were not satisfied in our package, for example: $d, q \in O(1/\gamma^2), \Delta \in O(1/\varepsilon^4)$ (these values are very large, and demand a lot of time and memory resources).

7.2 Error Correcting Capability

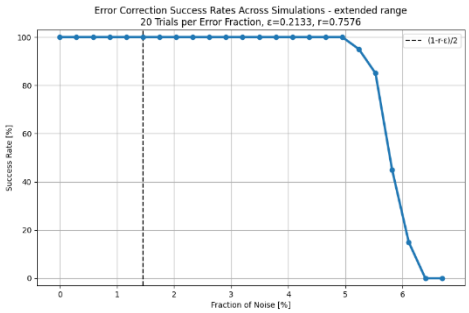
To evaluate the error correcting capability, we implemented a function that injects errors, and another function to run simulations over a range of error fractions. Typically, we performed 20 simulations for each fraction, injecting errors and then attempting to decode the received message.
The injected percentage of errors is from 0 up to slightly beyond the codes decoding capability.

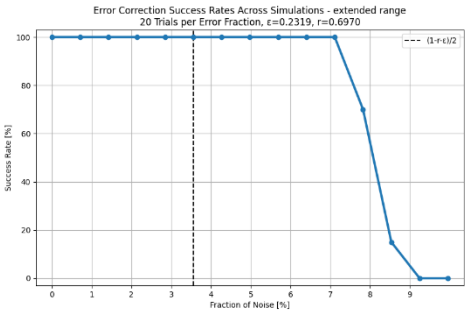
We plotted a graph where the x-axis represents the error percentage introduced, and the y-axis shows the success rate, measured as the percentage of 20 simulations that completed successfully.

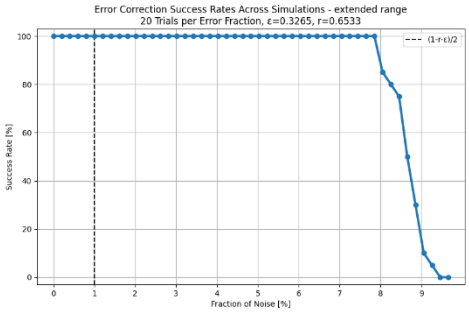
We searched for parameters of different values of r and ε (r is from 0.3 to 0.8, and ε is in range 0.1 to 0.3), and for each of them we ran three tests. The first one with only errors, the second is with only erasures, and the third one is combining both.

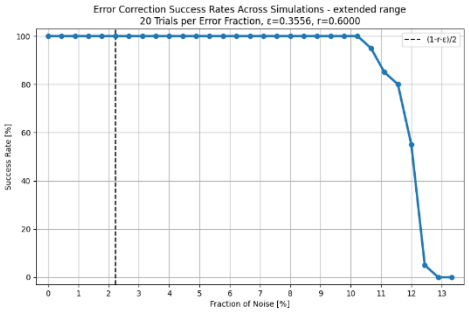
As we know, our code can correct s erasures and e errors if $s + 2e \leq 1 - r - \varepsilon$.

In the first test, injecting **only errors**, $s = 0$, and we can correct $e \leq \frac{1-r-\varepsilon}{2}$ errors.
We ran a few simulations and received results that indicate our code can correct $\frac{1-r-\varepsilon}{2}$ percent of errors:









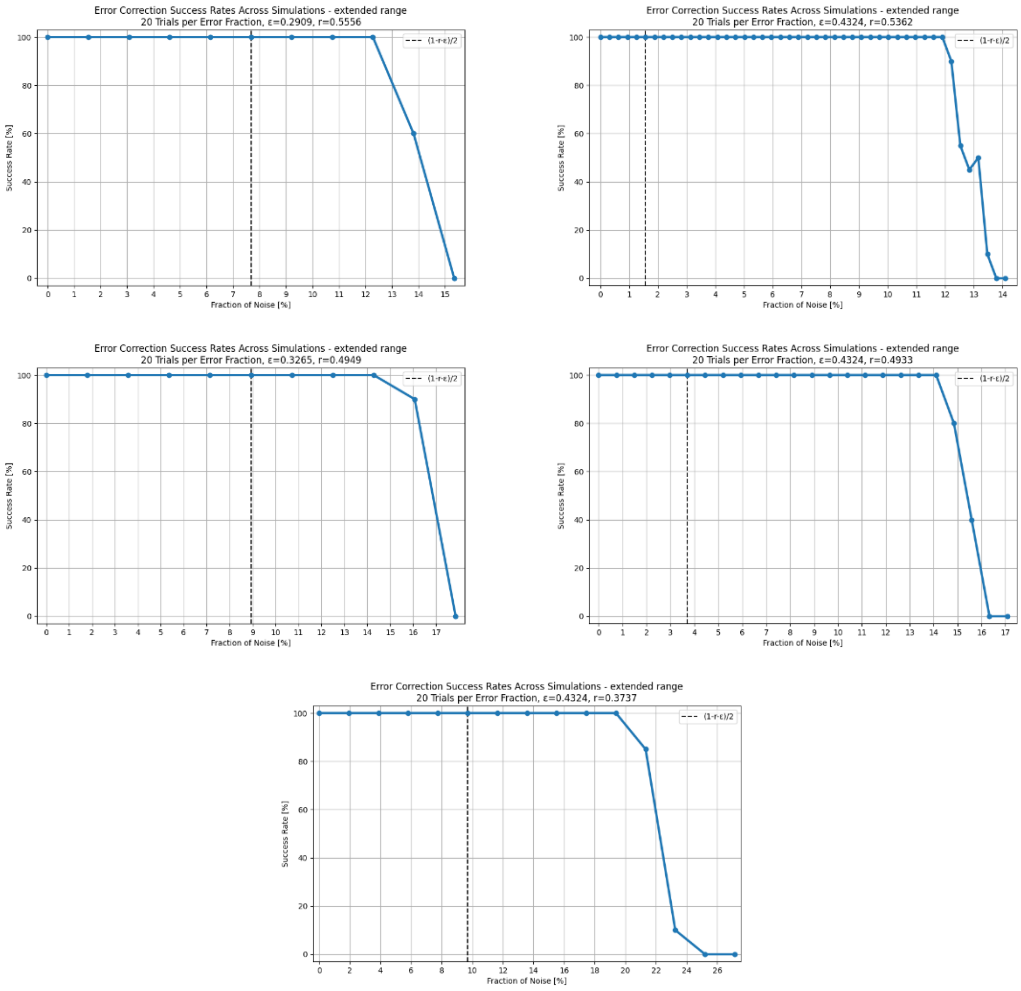
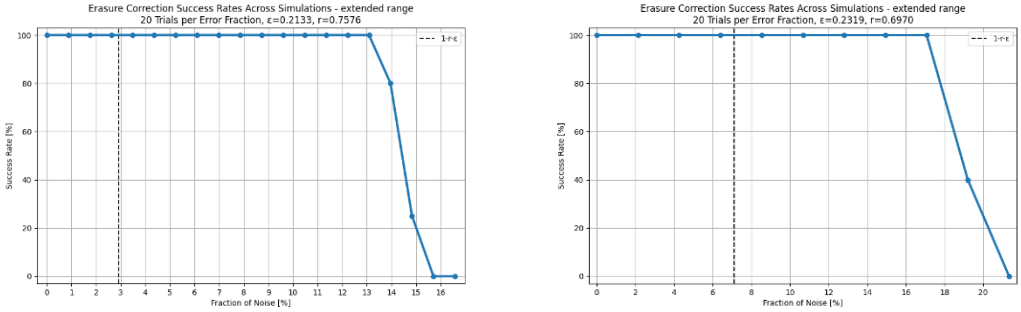


Figure 10: Plots of fraction of noise [%] vs success rate [%] of error correcting capability.
The dotted line is the maximum fraction of errors that the code should be able to correct.

In the second test, injecting **only erasures**, $e = 0$, and we can correct $s \leq 1 - r - \varepsilon$. We ran a few simulations and received results that indicate our code can correct $1 - r - \varepsilon$ percent of erasures:



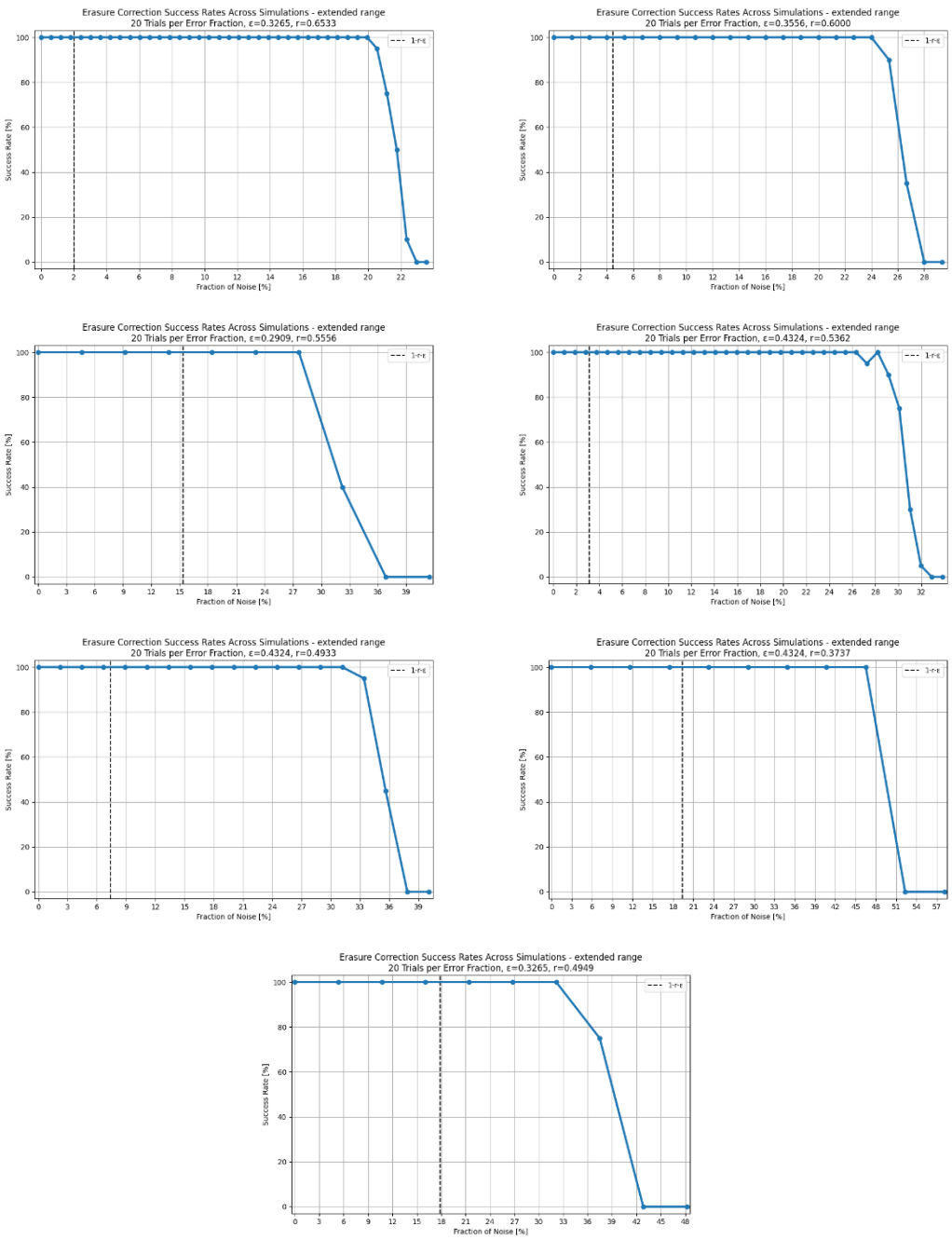


Figure 11: Plots of fraction of noise [%] vs success rate [%] of erasure correcting capability. The dotted line is the maximum fraction of errors that the code should be able to correct.

In the third test, injecting **both errors and erasures**, we set $s = e$, and we can correct if $3e \leq 1 - r - \varepsilon$, and that means we can correct $\frac{2}{3}(1 - r - \varepsilon)$ errors and erasures.

We ran a few simulations and received results that indicate our code can correct $\frac{2}{3}(1 - r - \varepsilon)$ percent of errors and erasures :

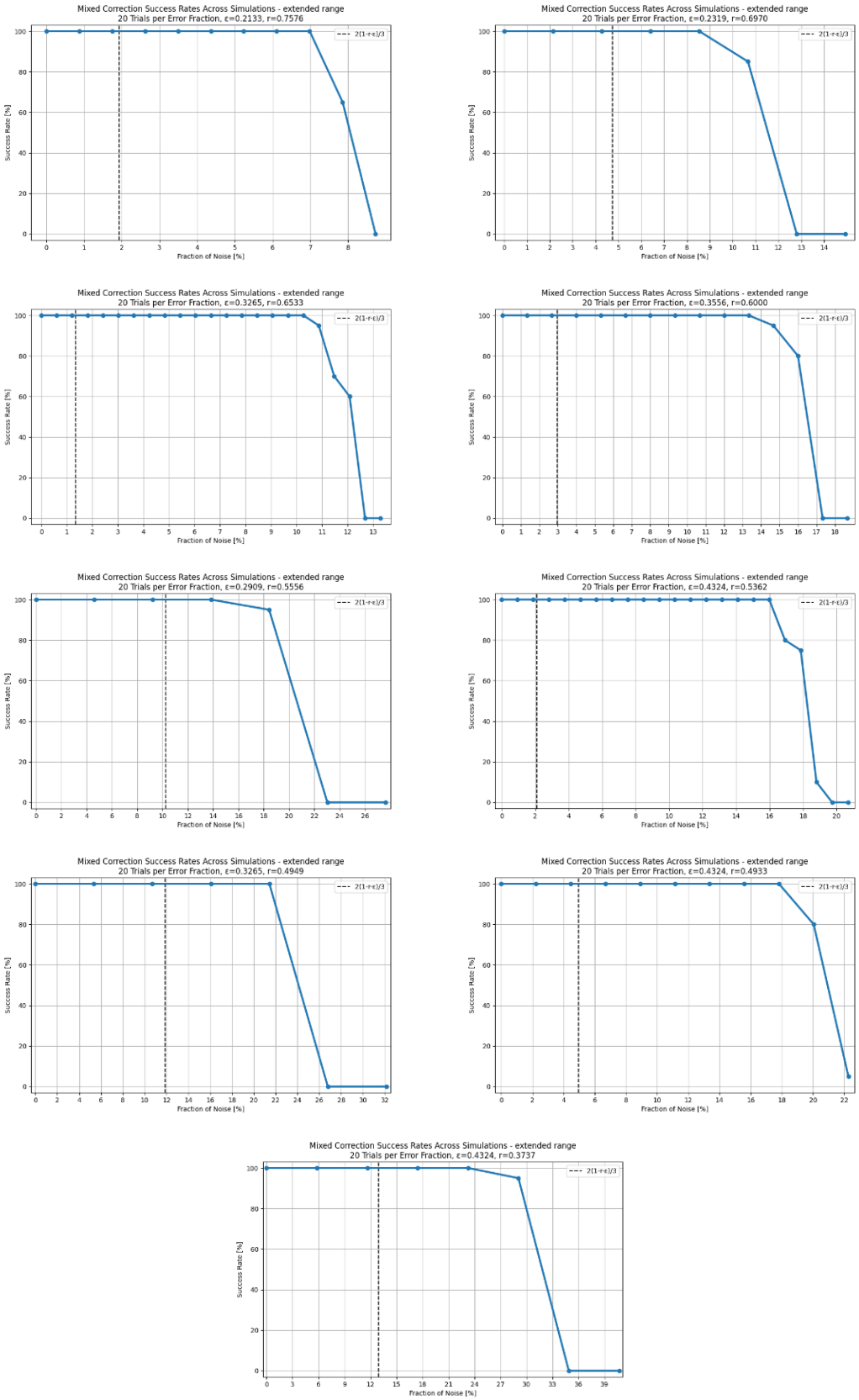


Figure 12: Plots of fraction of noise [%] vs success rate [%] of error and erasures correcting capability. The dotted line is the maximum fraction of errors that the code should be able to correct.

To conclude, our code corrects as many errors and erasers as intended.
As we can see, in fact it corrects even more than the expected value, there are a few possible reasons for this:
1. Since the error positions are random, the probability of completely corrupting the word may be low. With only 20 simulations, it's possible we haven't observed scenarios where the word cannot be decoded.
2. In the Left Code construction, there’s a part where the paper instructed to use Spielman’s code, but we used Reed-Solomon code. The error correcting capability of RS is optimal, and is probably better than the Spielman’s code.

7.3 Comparisons

Finally, we compared between Reed-Solomon to our code.

This comparison is not intuitive, since the size of the alphabet in the final codeword is different, meaning if there's one bit-flip, in our code it affects one symbol over $GF(q^{\Delta})$, and in the RS code it affects only one symbol over $GF(q)$.

There is an implementation of binary code described in the article, that would allow comparisons between error correcting capability, we didn't implement it.

For this reason, we decided to compare only between the running time of each code.

There are two types of comparisons:

1. The decoding ability between the codes is the same.
2. The rates of the codes are the same.

We increased the block size (by increasing the alphabet size, since $n \leq q-1$) of the Reed-Solomon, so it won't split the data to a few blocks.

We encoded and decoded the same word ('AAAA…A') using the different codes.

1. The first comparison:

The parameters of the linear-time code are:

$k = 80808, r = 0.745, \varepsilon = 0.216, pr = 73, qr = 13, pe = 97, qe = 13, b = 77$

The parameters of Reed-Solomon code are:

$$k = 80808, r = 0.745, nsize = 2^{17} - 1$$

Results:

```
LT Encoding time:  0.4544193744659424
LT Decoding time:  1.2137353420257568
RS Encoding time:  603.5952854156494
RS Decoding time:  1179.936364889145
```

Figure 13: Time comparison between the codes, same rate

2. The second comparison (same error correcting capability)

The parameters of the linear-time code are:

$k = 80808, r = 0.745, \varepsilon = 0.216, pr = 73, qr = 13, pe = 97, qe = 13, b = 77$

The parameters of the second Reed-Solomon code are:

$$k = 80808, r = 0.961, nsize = 2^{17} - 1$$

Results:

```
LT Encoding time:  0.501396656036377
LT Decoding time:  1.1035878658294678
RS Encoding time:  60.554606199264526
RS Decoding time:  85.77251601219177
```

Figure 14: Time comparisons between the codes, same error correcting capability

As we can see, both the encoding and decoding times for our code were notably shorter than the time it took for Reed-Solomon code. It was shorter when the rate

was smaller, since less redundancy is used – less check symbols are calculated.
There are a few reasons for these results:
1. Time Complexity - our code has linear running time, and the reedsolo package we use to encode and decode has quadratic running time.
2. In addition, since in the RS higher field size is used, the reedsolo algorithms will be slower, because its implementation cannot then use the optimized bytearray() structure but only array.array('i', …).

Note that in our code we also use the same Reed-Solomon implementation, but since we operate the code on small values (d - the Ramanujan degree, b - the block size), we can use smaller block sizes, and thus small field sizes, which reduces the running time drastically.

To conclude, we can see that both codes can correct similar error fractions, but our code encodes and decodes faster, though there is not much meaning to the comparison, because of the field size differences.
It's difficult to make the comparisons, but it is clear that our code has good performances.

8 Conclusion

In this project we implemented a fast near-MDS Error-Correcting Code as described in the article [1], tested its performances, and published it.

The code functions as expected, correcting the intended fraction of errors and erasures within the codeword, and appears to run in linear time.
It is designed to be user-friendly and easy to use.
Upon comparisons with Reed-Solomon code, our code encodes and decodes long codewords significantly faster than RS codes of equivalent block size.

However, there are limitations in the code, some of them are due to our implementation, and can be addressed, while others are the result of the code construction.

In our implementation, due to limited sizes of Ramanujan Graph, the code dimension must be large, at least tens of thousands bytes long. Encoding can't be done for all code dimensions and block sizes.
Additionally, our program supports only a constant alphabet size - 2^8.

The typical distance from MDS-codes, ε, is ≈ 0.25, which is relatively high. To reduce ε, the graphs would get larger, leading to larger code dimensions and block sizes, which would make the encoding and decoding processes more resource-intensive in terms of memory and time.

While the code outperforms other codes in terms of running time and error correction capability, due to the constrains mentioned above, we believe that most users would prefer to use other codes.
However, for users needing to encode large streams of data quickly, our code could be a useful solution.

Future plans

Further research can be done to improve the method of selecting parameters, as there may be more effective approaches to determine them.

The construction used to build the Ramanujan graph could be expanded to handle a wider range of graph sizes, revealing additional parameters.

The implementation can be upgraded to support a larger range of alphabet sizes.
In addition, the article describes a construction of a binary code, which addresses the limitation of the alphabet size – since the alphabet size grows exponentially in $1/\varepsilon$, the smaller ε is, the larger the alphabet size. Implementing it could lead to interesting and valuable results.

Finally, implementing the code in a low level language, would improve running time.

9 Reflection

As well as proper conclusions about the project, we gained insights about ourselves, and the way an engineering project is conducted. We would like to take a step back and acknowledge the journey we have been through in this project, that concludes four years of study and development.

First of all, we learned about team work, how to work efficiently together, while each of us contributes to the project.
There were a few stages during the work, and each of them required us to adjust to the new situation and make the changes necessary in the way of operation.
In the beginning, we needed to study the theoretical background about ECC, this was done separately, in case one of us didn't understand anything, the other one helped. Then, we read the article together and separately, and tried to understand the scheme described in the article. This was one of our first times reading an academic article, and we realized along the way how to read it (slowly), and understand the information.

Then, we approached the implementation, immediately opening PyCharm and start programming. This wasn't a wise decision, and we realized we must have at least a pseudo code before starting to program, so we wrote it and then started programming.
We believe this part should've included more preparation, we didn't define what exactly should the code include, we started to write encoding and decoding functions, and along the way encountered challenges that took us back, and solved them.
In future project, we would be more organized, and plan in detail what should the code include, divide it to stages, and be more tight to a schedule, regarding each part. When we know we have two months to write a code, it's too vague, and it's difficult to understand if we're on schedule, or need to invest more time in the work. We learned that schedules must be as specific as possible: what do we want to accomplish every week in this two months, of course with some room for hold ups and unexpected events or difficulties.
In addition, along the way we learned how to share the work efficiently, the strengths and weaknesses of each of us, what should be done together and what separately to maintain efficient work, while both of us are in the loop.

After the code was written we went on to the parameters part, this was quite difficult, there are many parameters to determine, and we got a bit lost.
We tried processing the information in a few ways – by a written summary, a table, and eventually by a drawn scheme. Writing the information in different ways, helped us to fully understand it, and eventually we were able to find fitting parameters.

In this stage, we got to the simulations, initially, the code was very slow. This happened mainly because of inaccurate coding, and also raw algorithms that could have been improved. We had to find the bottleneck in the code, this was a thing we haven't experienced before, and it was quite interesting and fun, along the way we made more and more improvements, some were neglect in terms of running time, and some were very important, and drastically improved our code, which was very satisfying.

When we started getting actual results, they were not always the expected results. A major challenge was to understand where exactly is the problem – is it in the way we wrote the specific simulation, in the parameters chosen, or in the implementation of the ECC itself.
We managed this by changing different things every time, we mainly focused on the parameters, because this was the part we were the least comfortable with. This took a long time, and the results were still not exact, so we checked the decoding process, and found out it had some inaccuracies. The whole way was not organized, and very time consuming.
This is an important lesson we learned, instead of juggling in the code trying to understand where is the problem – it's better to check all of the stages in the process properly, in a planned strategy, and not waste time checking the same function over and over again (it’s also a valuable lesson for life in general).

To conclude, this project was very interesting, not only because we explored the subject matter in depth, but also because we gained valuable insights into project management and teamwork. Additionally, we discovered our strengths and areas for growth, enhancing our overall skill set for future work.

Bibliography

[1] V. Guruswami and I. Piotr, "Linear Time Encodable/Decodeble Codes With Near-Optimal Rate," *IEEE,* 2005.

[2] T. Filiba, "reedsolo: A pure Python Reed-Solomon encoder/decoder library," PyPI, 2023. [Online]. Available: https://pypi.org/project/reedsolo. [Accessed October 2024].

[3] M. Hostetter, "A performant NumPy extension for Galois fields and their applications," PyPI, 2024. [Online]. Available: https://pypi.org/project/galois/. [Accessed October 2024].

[4] A. Lubotzky, R. Phillips and P. Sarnak, "Ramanujan Graphs," *Combinatorica,* vol. 8, no. 3, pp. 261-277, 1988.

[5] W. C. Huffman, Fundamentals of Error-Correcting Codes, Cambridge University Press, 2003.