JACOBS
UNIVERSITY

# Implementation of Locally Decodable Expander Codes and Affine Permutation Codes

by

**Horia Ionut Turcuman**

Bachelor Thesis in Computer Science

Submission: May 17, 2021 Supervisor: Prof. Keivan Mallahi-Karai

Jacobs University Bremen | Department of Computer Science and Electrical Engineering

## English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

## German: Erklrung der Autorenschaft (Urheberschaft)

Ich erklre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschlielich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zuknftigen Arbeiten verglichen werden zu knnen. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfltigung.

Diese Arbeit wurde noch keiner anderen Prfungsbehrde vorgelegt noch wurde sie bisher verffentlicht.

17.05.2021 [illegible]

Date, Signature

## Abstract

Error-correcting codes are essential for many applications that people depend on. There are several very good codes that gained much popularity in the past, their success is partly due to their practicality since implementation and actual usage is the final goal. Other schemes are very interesting from a theoretical point of view since they are built on smart ideas and they make use of tools from other fields of study. This thesis aims to spark interest in implementation of a specific family of codes that may seem rather theoretical but might lead to a good, practical scheme. Attempting to implement this code reveals a number of challenges and a few solution are proposed. A second goal is to introduce a generalisation of permutation codes and study the notion of basis since it has potential to lead to encoding or decoding algorithms.

I would like to thank my advisor, Keivan Mallahi-Karai for guiding me through the study of error-correcting codes and to Lorenzo Rota who provided me with the implementation of the Reed-Muller code.

# Contents

# 1 Introduction

Error correction of corrupted data is achieved through introduction of redundancy. The procedure is called encoding and the result is a codeword. The codeword suffers slight modification (errors) and an operation named decoding is performed with the hope that the original data can be reconstructed. Some error correction schemes guarantee that if no more than a certain number of errors are introduced, the decoding is always successful. Other non-deterministic schemes return the correct result only with high probability. Depending on the application, different combinations of parameters such as rate, correction capability, codeword length and size of the code may be more desirable and therefore various schemes have been developed and are currently in use.

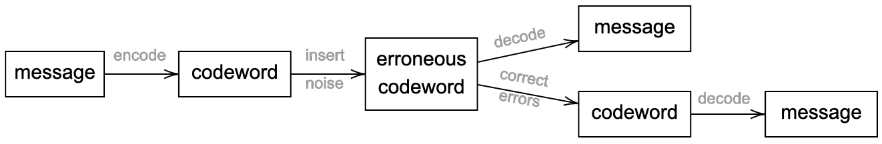


Figure 1: General coding scheme.

Figure 1 illustrates the main operations of a coding scheme. Through encoding, a message (or just data) is mapped to a codeword. Through decoding, a slightly changed version of the codeword can be mapped back to the message. An intermediate step called correction, that retrieves the original codeword from its perturbed version, might be used. Developing coding schemes with good parameters is very important for many applications in data transmission, mass storage, probabilistically-checkable proofs, private information retrieval schemes and many more, since some of the basic things people depend on nowadays wouldn't be possible without noise and error correction.

There are a few different models for the way a codeword is perturbed. The original work of *Shannon* [5] proposes a rather weak model in which the errors are introduced at random with some probability. A stronger model is the *Hamming adversarial model* in which the channel (or the occurrence of errors in general) is considered to be an adversary that aims to hinder correct decoding by introducing errors in a smart way and it is only limited by the number of errors they can produce and not by the available time they can take to decide. Many codes provide error correction for models in which the adversary's time is limited to a polynomial function, but the codes described in this paper provide local correction in extreme Hamming case which is achieved randomisation.

This work aims to provide an implementation for the *locally correctable expander code* algorithm presented in [4]. Aside from its implementation, the algorithm is also explained leaving out most of the technical details trying to facilitate an intuitive understanding for the potential user of this implementation. We start with an introduction into error-correction and necessary background in graph theory. Section 4 and 5 describe the expander code, the algorithm and its implementation. Section 6 contains an example of usage and creation of a good graph and an outer codeword. Lastly, section 7 represents a second component of the thesis, which describes a generalisation of permutation codes and linear codes and studies its basis.

## 2 Preliminaries and Notation

This section aims to introduce the mathematical objects that come up in the paper and settle the notation. Only things that are necessary are defined since the purpose is to make the paper accessible to the reader and not to provide an elaborate introduction into the fields. Different papers define things in slightly different ways, hence the definitions given here might not coincide with others entirely, although the tendency is to be as general as possible.

### 2.1 Introduction to error-correction

**Definition 2.1. (Coding Scheme)** A coding scheme is a tuple $(E, C, D)$ where E and D are the encoding and decoding functions i.e. $E : \Sigma^k \to C$, $D : C \to \Sigma^k$ and $C \subseteq \Sigma^n$ is a subset of words over the alphabet $\Sigma$. $C$ is a code and elements of $C$ are called codewords. The value $k/n$ is the rate of the code and it measures the fraction of information encoded in a codeword.

A very important class of codes are *linear codes*:

**Definition 2.2. (Linear Codes)** Let $F$ be the field and n, k, d be integers such that $1 \leq k \leq n$. A $[n, k, d]_F$ linear code is a subspace $C$ of $F^n$ which has dimension k and minimum Hamming distance d. The Hamming distance is the number of positions where two elements differ.

A family $\{C_t = [n_t, k_t, d_t]_F\}_t$ of linear codes with $n_t \to \infty$ is called *good* if there exists a constant $c > 0$ such that for all $t$ it holds:

$$\frac{k_t}{n_t} \geq c, \frac{d_t}{n_t} \geq c$$

It is proved that the minimum distance of any linear code $C$ equals it's minimum *weight* which is the minimal distance of a codeword from the 0 element of $F^n$. The *relative distance* $\Delta = d/n$ is helpful when talking about asymptotic properties of codes.

A code (together with an encoder) is locally decodable if there is an algorithm which can recover a symbol of an encoded message from an erroneous codeword by reading just a few positions from the codeword. The following definitions belong to [4]:

**Definition 2.3. (Locally Decodable Codes (LDC))** A pair $(C, E)$ where $C \in \Sigma^n$, $|C| = k$ and $E : \Sigma^k \Rightarrow \Sigma^n$ is called $(q, \rho)$-locally decodable with error probability $\eta$, there is a randomised algorithm $R$ so that all $w \in \Sigma^n, m \in \Sigma^k$ with relative distance $\Delta(w, E(m)) < \rho$ and for each $i \in [k] = \{1, \ldots, k\}$ it holds that: $\mathbb{P}[R(w, i) = m[i]] \geq 1 - \eta$ and $R$ accesses at most $q$ symbols of $w$.

A **Locally Correctable Code (LCC)** is defined similarly to a **LDC**, the only difference being that $R(w, i)$ results in $E(m)[i]$: $\mathbb{P}[R(w, i)] = E(m)[i] \geq 1 - \eta$. The constants $\rho, \eta, q$ are referred to as the *errors rate*, *failure probability* and the *query complexity*. The positions queried by $R$ form the the *query set*.

**Definition 2.4. (Smooth Reconstruction Algorithm)** Given $C \subseteq \Sigma^n$, consider the pair $(Q, A)$ where $Q$ is a randomised query algorithm with inputs in $[n]$ which outputs a random ordered subset of $q$ elements from $[n]$ and $A : \Sigma^q \times [n] \Rightarrow \Sigma$ is a deterministic reconstruction algorithm. With the notation $c_{|Q(i)} = (c[Q(i)[1]], \dots, c[Q(i)[q]])$, $(Q, A)$ is said to be a smooth reconstruction algorithm of query complexity $q$ if:

- Smoothness: the queries in $Q$ are uniformly distributed in $[n]$.
- Reconstruction: for all $i \in [n]$ and all legitimate codewords $c \in C$, $A(c_{|Q(i)}, i) = c[i]$.

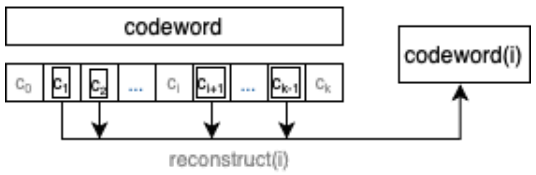


Figure 2: Reconstruction algorithm.

**Definition 2.5. (Permutation Codes)** Permutation codes are subgroups $S$ of $S_n$.

The distance function is the *Hamming distance* for permutations i.e. the number elements of the set $[n]$ where two permutations act differently. The minimum distance of a permutation code is proved to be equal to its minimum weight which is defined as before.

## 2.2 Introduction to graphs

Informally, an *expander graph* is a finite, undirected graph in which every subset of vertices has a large boundary. Every connected graph is an expander but only the ones that have low degrees and high expansion parameters are *good expander.* Different types of expansion exist but here the notion of *spectral expansion* is used:

**Definition 2.6. (Ramanujan Graph)** Let $G$ be a $d$-regular and $\alpha$ be the second highest eigenvalue of it's adjacency matrix (in absolute value). $G$ is a *Ramanujan graph* iff. $\alpha \geq 2\sqrt{d-1}$.

Intuitively, an expander is good because a random walk through the graph arrives at a random position which is uniformly distributed in the graph in just a few steps. A Ramanujan graphs is a very good expander because the number of steps of such a random walk is very small.

The *bipartite double cover* of a graph is it's bipartite version and it is defined as follows:

**Definition 2.7. (Bipartite Double Cover)** The bipartite double cover of a graph $G$ with $n$ vertices is the bipartite graph $H$ of $2n$ vertices where $V_0(H) = V(G)$ are the vertices on the left side and $V_1(H) = V(G)$ are the ones on the right. To distinguish between vertices on the right side and vertices on the left, denote for every $v \in V(G)$ by $v_0$ the one on the left and by $v_1$ the one on the right. An edge $(v_0, u_1)$ is in the graph iff $(v, u) \in G$.

**Definition 2.8. (Cayley Graph)** Let $G$ be a group and $S = \{s_1, s_1^{-1}, \dots, s_n, s_n^{-1}\}$ a set of generators together with their inverses. The *Cayley graph* $\Gamma(G, S)$ is the graph with the vertex set $V(\Gamma)$ identified with $G$ and which contains all and only edges of the type $(g, gs)$ with $g \in G$ and $s \in S$.

For a given graph $G$, we will use a bijective function $\mathcal{O} : E(G) \Rightarrow [|E(G)|]$ which assigs to each edge a different number. We call this a *global ordering* of the edges of $G$ since it allows us to say that the $i^{th}$ edge is $e_i = E(G)[i] = \mathcal{O}^{-1}(i)$. Similarly, one can define the *local ordering* at a vertex $v$: $\mathcal{O}_v : E(v) \Rightarrow [|E(v)|]$ where $E(v)$ are the edges adjacent to $v$. The $i^{th}$ edge in the local ordering at $v$ is $e_i^v = E(v)[i] = \mathcal{O}_v^{-1}(i)$.

## 3 Motivation

Local decodability is an interesting feature because it has and may still have more practical applications, but also because it is not easy to come up with local decoding algorithms with good parameters and time complexities low enough to actually be implemented and used. This is a challenge that researchers in the field are facing, which might have a large impact on some technologies once a very good solution is found. There seems to be an increasing interest in the use of expander graphs for building codes with locality notably being [3] which also introduces and makes use of a few other very interesting tools. Even if this schemes might seem theoretically good but practically unfeasible, it is still important to implement them in order to understand what difficulties need to be overcome and see how the research improves in this direction of practicality.

The studied code and algorithm achieve local correction through a few very smart ideas. I believe that further improvement or reuse of these ideas might be possible in other schemes, hence the main motivation of this thesis is to point out the major problems that need solutions and implementations before similar codes or local correction algorithms can be of practical use.

## 4 Description of the Locally Correctable Expander Codes

This section provides an intuitive description of the construction of the expander code and local decoding algorithm leaving out most of the technical details. The interested readers are encouraged to read the reference paper [?] if they try to understand everything in depth. Moreover, this description does not provide insights into how good parameters can be selected and does not prove any of the main statements but merely resumes to ease the understanding of the main ideas.

### 4.1 Construction of the Expander Codes

The construction of the code is based on a $d$-regular expander graph $G$ with $n$ vertices and a locally reconstructible code $C_0 \in \Sigma^d$. Note that the degree of the graph and length of the code need to match and the number of edges in this graph is $nd/2$. Let $H$ be the double cover of $G$ containing $2n$ vertices and $N = nd$ edges. Further, fix a global ordering $\mathcal{O}$ of the edges of $H$ and local orderings $\mathcal{O}_v$ at all vertices $v \in V(H)$. The *expander code* $C \in \Sigma^N$ is:

$$C = C(C_0, H, \mathcal{O}, \{\mathcal{O}_v\}) = \{c \in \Sigma^N : c_{|E(v)} \in C_0 \text{ for all } v \in V(H)\}$$

Here $c_{|E(v)} = (c[\mathcal{O}(e_1^v)], \ldots, c[\mathcal{O}(e_d^v)])$ represents the symbols of $c$ at all global positions of the edges adjacent to $v$, taken in the order given by the local ordering at $v$.

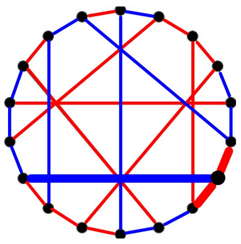


Figure 3: In this example $\Sigma = \{'r', 'b'\}$ and each edge is labeled as shown. Looking at the lower right thickened vertex, the thick edges ordered in the local order, need to form an inner codeword.

To generalise the example, imagine that a symbol $c[i]$ of $c$ is assigned to the edge $e_i$ . Then $c$ is a correct codeword of the code $C$ if and only if looking at any vertex $v$, the symbols assigned to the edges adjacent to $v$, taken in the local order at $v$, form a codeword in $C_0$.

The code $C$ is called the outer code while $C_0$ is the inner code. If the graph $G$ has very high expansion (ex. a Ramanujan graph) and the inner code has has good rate and distance, so does the outer code and it is locally correctable. If the fraction of errors $\rho$ in a word $w \in \Sigma^n$ is small enough, the successful correction probability is very high. If a sequence of such codes with $N \to \infty$ is considered, the query complexity $q$ is sublinear and the algorithm runs in linear time in $q$.

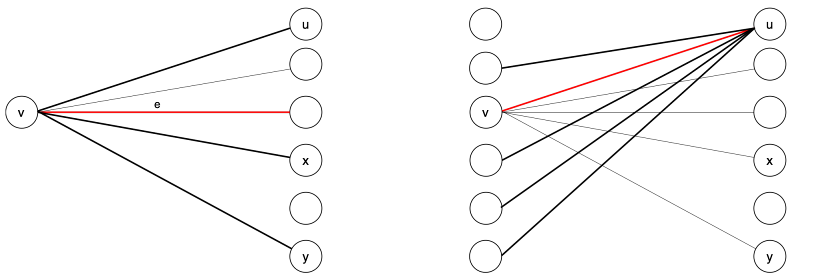
## 4.2 Local Correction Algorithm

The input of the correction algorithm is a corrupted codeword $w \in \Sigma^N$ that differs from a correct codeword $w^*$ in at most $\rho$ fraction of errors, and an edge $e \in H$. The goal is to return the symbol $w^*[\mathcal{O}(e)]$. Note that $w^*$ is not known.

The algorithm starts by building a $q_0$-ary tree $T$ where each vertex represents an edge of $H$, hence given a codeword, to each vertex of $T$ we can associate a symbol. First, it initialises the tree by letting the input edge $e = (v_0, u_1)$ be the root of the tree. The second level of the tree is constructed by considering the vertex $v_0$ and a query set given by the reconstruction algorithm of the inner code: $Q_0(\mathcal{O}_{v_0}(e))$, with the target position being the position of $e$ in the local ordering at $v_0$. 3 illustrates this idea. Once the second level is constructed, the procedure is repeated for each vertex on this level, each vertex on the next level, . . . , until the edges represented by the leaves are uniformly distributed in $H$. Since the graph has good expansion, only a small number of iterations is required.

**Algorithm 1:** makeTree: Uses the local reconstruction property of $C_0$ to build a tree of edges.

```
Data: edge $e \in H$ representing the root of a tree $T$, requested tree depth $L$
initialise tree $T$ with a single node $e$
for $l$ from 1 to $L$ do
    for $edge = (v_0, u_1) \in$ current leaves of $T$ do
        if $l$ mod 2 == 0 then
            $vertex = v_0$
        else
            $vertex = u_1$
        end
        $local\_pos = \mathcal{O}_{vertex}(edge)$
        for $query\_pos \in Q_0(local\_pos)$ do
            $new\_edge = \mathcal{O}_{vertex}(query\_pos)$
            add a vertex $new\_edge$ as a child of $e$ in $T$
        end
    end
end
```



(a) Red is to be reconstructed, thick black is the query set.

(b) In the second step, the query set is chosen from the other side.

Figure 4: Running the algorithm for the initial edge $e$, the query set represented by the thick black edges in (a) might be found. For each vertex on these edges that is on the other side, the procedure is repeated. (b) shows a new query set at $u$ for the edge connecting $u$ to $v$.

Once the tree $T$ has been built, one could read all symbols assigned to the edges on it's leaves and use the reconstruction algorithm $A_0$ of the inner code to iteratively reconstruct symbols. This would be done in a bottom up approach and the final result would be the symbol at the root. The success probability of this approach is small ($(1-\rho)^{\#leaves}$) since even one wrong symbol on a leaf might yield an incorrect result.

To solve the issue, the algorithm tries to correct instead of reconstruct. Since the correction (explained later) uses the symbols of $w$ on every vertex of $T$, an adversary might change just symbols on edges close to the target edge $e$ on $H$, preventing in this way the correction. To fix this problem, new trees $T_1, \ldots, T_m$ rooted at the leaves of $T$ are constructed using the *makeTree* procedure. Since they are rooted at edges uniformly distributed in the graph, an adversary can not do worse than random errors against this procedure. Therefore, the full algorithm will first correct the trees $T_1, \ldots, T_m$ and only afterwards reconstruct $T$ using the corrected symbols on its leaves.

To understand the correction function *correct_lower_tree*, let us define the labelling of a tree to be and assignment of symbols to the tree's vertices. Consider the definition of the distance between two labelings of the same tree, $\sigma_1$ and $\sigma_2$:

$$D(\sigma_1, \sigma_2) = \max_{P \in \text{Paths from root to leaves}} \Delta(\sigma_{1|P}, \sigma_{2|P})$$

In words, the distance is the highest relative distance between words defined by paths from roots to leaves. Suppose that $\tau, \tau^*$ denote the labelings of the same tree by symbols from $w$ and $w^*$ (i.e. $\tau(e) = w[\mathcal{O}(e)]$) and $\sigma_1^*, \sigma_2^*$ are labelings of the same tree arising from two different legitimate codewords. It is very likely that:

1. $\tau_{|P}$ contains less than $\gamma < 1/2$ errors on any path $P$ from root to leaves.
2. The previous point implies $D(\tau, \tau^*) < \gamma < 1/2$.
3. $D(\sigma_1^*, \sigma_2^*) = 1$, which implies $D(\tau, \sigma_1^*) > 1 - \gamma > 1/2$ iif. $\sigma_1^* \neq \tau^*$.

These proposition can be used to show that there is a unique symbol $a$ that maximises:

$$Score(a) = \min_{\sigma^* : root(\sigma^*) = a} D(\sigma^*, \tau)$$

The *correct_lower_tree* function computes this score for every symbol $a$ by going through the tree bottom to top and using the reconstruction algorithm of the inner code. At each vertex $e = (v_0, u_1)$, the subtree rooted at $e$ is considered, the score for each symbol $a$ is computed using the results for the direct descendants $e_1, \dots, e_{q_0}$ and these scores are stored in order to be used when the scores for the parent vertex are computed. The following reccurrence formula is followed:

$$best_a(e) = \begin{cases} \begin{cases} 1 & \text{if } \tau[e] \neq a \\ 0 & \text{if } \tau[e] = a \end{cases} & \text{if e is a leaf of } \tau \\ \min_{(a_0, \dots, a_{q0}) \in S_a} \max_{r \in [q_0]} best_{a_r}(e_r) & \text{if e is not a leaf and } \tau[e] = a \\ \min_{(a_0, \dots, a_{q0}) \in S_a} \max_{r \in [q_0]} best_{a_r}(e_r) + 1 & \text{otherwise} \end{cases} \tag{1}$$

This is computing the distance function defined above. The first part is the base case. For each leaf, the smallest distance between $\tau$ and a correct labelling (labelling arising from a correct codeword) that has $a$ on $e$, is either 0 if $\tau[e] = a$ or 1 otherwise. Moving on to the second part, the set $S_a$ is the set of all ordered sets of $q_0$ symbols for which the reconstruction algorithm evaluates to $a$ on the local position of $e$. Since any of these sets would yield $a$, the children of $e$ may be any of them if $e$ is fixed to $a$. Therefore, the best such set is the one that minimises the distance. For each set $(a_0, \dots, a_{q_0})$, the minimal path distance at every child $e_r$ of $e$ (which is now fixed to the symbol $a_r$) is given by $best_{a_r}(e_r)$. In the path distance formula, the maximum over all paths root to leaves is considered, so one needs to take the maximum over all children of $e_r$. In the last part a 1 is added because the minimised value does not consider the root $e$ for which $\tau[e] \neq a$.

# 5 Implementation

The previously described algorithm was implemented in *sagemath*, but extensive use of the *sagemath* functionality is made in the given example were tools from graph and group theory modules are needed. The implementation does not make use or follow the abstract classes and structure of the Coding Theory *sagemath* module. We will first discuss the structure of the code so that it is clear how it relates to the correction algorithm given in the previous section, and afterwards present the implemented optimisations and chosen parameters.

## 5.1 Code Structure

The main class is *LCECode*. The constructor has two arguments: a *ReconstructibleCode* and a *BipartiteGraph* from *sagemath*. The first is an abstract class meant to define the functionality of a locally reconstructible code, and the second is a double cover with edges labeled in the following way: *(left_local_pos, right_local_pos, global_pos)*. The vertices should be labeled by *(anything, side of the graph)*.

The starting point of the correction algorithm is *correct(pos, codeword_reader)* method which aims to find the correct symbol at position *pos*. The *codeword_reader* exists in order to limit the algorithm's access to the codeword, to allow different storing formats and to count the number of queries made (for performance analysis). The function build_reconstruction_tree creates the top and bottom trees as instances of the internal class *LCECode.Tree* using the *makeTree* procedure. These are labeled by symbols read from the codeword in *fill_bottom_trees_with_symbols*. Next, *correct_lower_tree* computes with high probability the correct symbols on the leaves of $T$, and *reconstruct_top_tree* yields the only symbol that fits the root of $T$ given the corrected symbols at the leaves.

## 5.2 Optimisations

One of the big issues faced during the implementation stage was that basic operations on the *sagemath* graph object like iterating over the edges or finding the edge of a given local position was computationally expensive and testing was not possible. As a solution, an internal representation of the graph that facilitates exactly the required operations was developed and the running time drastically decreased. This representation stores all edges in an array in their global order. Each edge stores the adjacent vertices together with the corresponding sides of the graph. Each vertex stores it's edges in an array in local order and therefore all required indexing and position-to-edge conversions are constant time.

A second optimisation used in the code tries to reduce the number of queries on the codeword during the labelling of the bottom trees phase. In the original algorithm, if two edges appear multiple times in the bottom edge trees (or even multiple times in one of them), a new query on the codeword is made even if the symbol is already known. In the implementation, all global positions are stored and one query is released for each unique position. Assuming a scenario in which each query costs time or another type of penalty and they need to be minimised, this optimisation might be significant.

One big drawback in the scheme is the need of generating, for each local position an edge in the tree and each symbol $a$, all ordered subsets of symbols for which the reconstruction algorithm of the inner code returns $a$. The time complexity of this generation is $|\Sigma|^d$ and it needs to be done for every vertex and every symbol. Instead of recomputing each time, A pre-generation for all possible pairs (*local position*, $a$) could be performed at instantiatin time, but that would make the make the running time of the constructor be unfeasible large. The implemented approach stores the symbol sets after every generation and reuses them when possible. The result is that the more corrections are performed the faster the algorithm works, but also the more memory it uses. A further improvement would be to drop the least used results to keep the memory usage in certain bounds.

### 5.3 Parameters

Two important parameters are the depth of the top and bottom trees. The depth of the top tree needs to be high enough such that the edges corresponding to its leaves are uniformly distributed in $H$. IT should also not be greater than that because unnecessary queries would be performed and the time complexity would grow. Similarly, the depth of the bottom trees to be large enough such that there are enough levels in order to have successful correction, but not larger than that because it would be nonoptimal. If the graph $G$ has good expansion, the depth of the first tree will stay low, and if it is Ramanujan, one can easily compute a very good value. Studying the proof of correctness in [4], one can observe that the second depth should be chosen to be a fraction that is not too large, of the first depth.

Other parameters that need to bee chosen are the number of vertices of the initial graph $G$, the degree $d$ and hence the length of the inner code. The higher the parameter, the longer is going to take to correct. Intuitively speaking, if the number of vertices is larger, there would be more codewords in the code hence more different messages can be transmitted. We have seen that the query complexity of the inner code and the size of the alphabet have a large impact on the running time. The inner code might have other parameters as well. As an example, the *Reed-Muller* code is used and it requires 3 parameters: number of indeterminates, maximum degree of each monomial and the finite field order which directly determine the length, query complexity and the alphabet used.

## 6 Usage and Final Comments

### 6.1 Test construction

A reconstructible code that is simple to implement on top of *sagemath* tools is the *Reed-Muller* code. The example uses this code but doesn't explain its details since it is viewed as a blackbox by the expander code and the required functionality has been described in section 2.

One major problem with testing and usage of this scheme is that an expander graph (or merely one with good expansion) needs to be constructed. There are procedures to generate even Ramanujan graphs but these are either very complex, do not fully specify the resulting graph (number of vertices, degree) or are not implemented. It is also important that the degree of the graph matches the length of the inner codewords, and since the inner code has its own restrictions, the problem becomes difficult. Thankfully, the Cayley graph of the group $SL_2(F_9)$ described in section 2 has good expansion, allow for degree 16 that will perfectly match the inner code's length and give an easy method for choosing a correct outer codeword.

A second challenge is the lack of implementations for the expander code encoders. With an encoder algorithm it is easy to turn a message into a codeword, and there are theoretical encoders for the expander code but it was not possible to find an implementation, hence a different method for finding a proper codeword is used. The method relies on the construction of the underlying graph using an ordered set of generators $S = \{s_1, s_1^{-1}, \dots, s_d, s_d^{-1}\}$. Since the inverses also show up in the set, it contains an even number of elements and therefore the generated graph has even degree $d$. One way of making sure that the method gives a correct outer codeword is to enforce that, at every vertex, the symbols associated to their edges form an inner codeword by choosing one inner codeword $c$ and making it be the local inner codeword at every vertex. A systematic way of doing that is to consider the generators from $S$ and let them determine the local order at each vertex $v$: the edge $(v, s_1 \cdot v)$ is the first one, $(v, s_d \cdot v)$ comes last. What is important to notice is that looking at a vertex $s_i \cdot v$, the edge $(s_i^{-1} \cdot s_i \cdot v), s_i \cdot v)$ is the same as $(v, s_i \cdot v)$. Therefore, the $i^{th}$ edge will needs to have the same symbol as the $j^{th}$ one and this happens for every pair $(i, j) \Rightarrow c[i] = c[j]$ if $s_i^{-1} = s_j$. To be able to assign in this way, every symbol from the alphabet needs to show up even number of times in $c$, and the generators from $S$ need to be ordered such that the equality does hold for any pair $(i, j)$.

It is straightforward to extend the described labelling to the double cover of the Cayley graph if we still consider the vertexes to be elements from the group with an additional index that denotes the side. To be able to construct an outer code, the double cover needs to be labeled with the local positions given by the generators of the group and a random global order. After this step, an instance of the *LCECode* is created and an erroneous version of the generated codeword can be corrected.

## 6.2 Comments

The generated double cover has 1440 vertices, 11520 edges and degree 16. The query complexity of the inner code is 15. Instantiation takes a long time because of the conversion to the internal representation of the graph. The correction time is low since the depths of the constructed trees are small. A tree of depth 4 means already 3375 queries for each lower tree so it is out of the question since it is not local anymore. Because of this reason, the correction function of the lower trees makes only 2 steps and the correction. We can that the different choices and parameters influence the resulting code. With a Reed-Muller inner code, the lowest query complexity I could achieve was large but the generated graph was not large enough. Much larger graphs would not be computationally feasible so the only solution would be an inner code with lower complexity.

In conclusions, there is still work to be done in order to have a practical implementation of this scheme, both from the theoretical and implementation sides. Important challenges are highlighted throughout the paper and a few partial solutions are also presented with the aim of assessing the current situation and hope to determine more work in this direction.

# 7 Affine Permutation Codes

## 7.1 Introduction

In this second component of the thesis we will have a look at a generalisation of permutation codes and linear codes that combines them.

Currently, permutation codes have not been studied extensively even though they have more structure than other types of codes. This additional structure makes them more rigid and difficult to obtain, but it may also give new schemes and interesting results. There are nevertheless research papers [2, 6] on permutations codes that do not require the group structure but merely the fact that an element can not occur multiple times in the usual way of representing permutations i.e. $\sigma = (\sigma_1, \dots, \sigma_n)$. The fact that this condition is enough for creating new coding schemes seems promising and for this reason this section explores the notion of basis which might be beneficial when working with this type of codes.

On the other hand, linear codes are well developed and very mature meaning that there are many studies, experience and theory behind them. Many popular schemes are successfully used in a wide range of applications so far. Hence, given their success, it is natural to try to find similarities, transfer their properties and mimic their theory when developing a new class of codes. This is done in [1] where similar concepts between permutation and linear codes are discussed. All properties are potentially interesting when developing a new scheme or trying to add functionality to a code, so it is important to have good results and to carefully study them if people are to start developing permutation code schemes.

The main focus of this section is to study the notion of basis for the code emerging from the generalisation of the natural action of permutations $\sigma \in S_n$ on elements $v = (v_1, \dots, v_n) \in F_q^n$ which works as follows: $\sigma \cdot v = v^\sigma = (\sigma(v_1), \dots, \sigma(v_n))$. Properties of the permutation code are extended to the new one, so called *affine permutation code*, and results are adjusted to fit the more general framework. For an intuitive view, one can imagine this new code to be a combination of two other codes, one linear and one of permutations. We also expect that it can in special cases be either the initial linear code or initial permutation code.

## 7.2 Description of the Affine Permutation Codes

We start by giving the definition of the studied code and a short discussion on this choice:

**Definition 7.1.** Let $S \leq S_n$, $V \leq F_q^n$ and let $\varphi$ denote the action of $S$ on $F_q^n$ by permuting it's entries i.e. $\sigma \cdot v = v^\sigma = (\sigma(v_1), \dots, \sigma(v_n))$. We further assume that $V$ is invariant under the action of permutations from $S$. An *affine permutation code* is the semidirect product $H = S \ltimes_\varphi V$ where the underlying set is the Cartesian product $S \times V$ and the group operation is defined as:

$$\cdot : (S \ltimes_\varphi V) \times (S \ltimes_\varphi V) \to (S \ltimes_\varphi V) \text{ is } (\sigma, v) \cdot (\sigma', v') = (\sigma \circ \sigma', \sigma \cdot v' + v)$$

When $S = \{e\}$ the group operation becomes $(e, v) \cdot (e, v') = (e, v + v')$ and the group is isomorphic to $V$ viewed as an additive group. When $V = \{0\}$ the operation is $(\sigma, 0) \cdot (\sigma', 0) = (\sigma \circ \sigma', 0)$ so $H$ is isomorphic to $S$.

With the intuition that this is a generalisation of permutations, we further define the action of $H$ on $V$:

$$\cdot : H \times V \to V \text{ is given by } (\sigma, v) \cdot w = \sigma \cdot w + v$$

The elements $(e, v)$ act just like addition on $V$ and the elements $(\sigma, 0)$ act as permutations. Now that this action is defined, one can check that the semidirect product is the right generalisation in the following way:

$$\begin{aligned} ((\sigma, v) \circ (\sigma', v')) \cdot w = (\sigma, v) \cdot ((\sigma', v') \cdot w) \quad &= (\sigma, v) \cdot (\sigma' \cdot w + v') \\ = \sigma \cdot (\sigma' \cdot w + v') + v \quad &= \sigma \cdot (\sigma' \cdot w) + \sigma \cdot v' + v \\ = (\sigma \circ \sigma') \cdot w + \sigma \cdot v' + v &= (\sigma \circ \sigma', \sigma \cdot v' + v) \cdot w \end{aligned}$$

This proves that the composition of two elements of H acts in the same way as firstly acting by the right term and secondly acting with the left one. This is one of the properties that need to be satisfied hence the the group operation can only be as defined.

## 7.3 Basis

The basis of a linear space is a set of linearly independent vectors that span the space. The basis of a permutation subgroup of $S_n$ is a subset of $[n] = \{1, \ldots, n\}$ such that only the identity permutation fixes all of its elements. We substitute vectors of $V$ for the elements of $[n]$ and come to the definition of a basis of $H$:

**Definition 7.2** (Basis)**.** A basis of an affine permutation code $H = S \ltimes V$ is a subset $B$ of $V$ such that $Stab_H(B) = \{(e, 0)\}$ i.e. $\nexists h \neq (e, 0) \in H$ with $h \cdot b = b \ \forall b \in B$.

A natural question is whether a basis always exists or not. The answer is not in general but there are cases where it does. We prove:

**Proposition 7.1.** The set $F_q^n$ is a basis for the affine permutation codes $H = S \ltimes F_q^n$.

*Proof.* Suppose that $\forall (\sigma, v) \in H, w \in F_q^n$ it holds that $(\sigma, v) \cdot w = \sigma \cdot w + v = w \Rightarrow \sigma \cdot w = v - w$.

Let $w = (1, \ldots, 1)$ and suppose $v \neq (0, \ldots, 0) \Rightarrow v_i \neq 0$ for some position $i \in [n] \Rightarrow w_i - v_i \neq 1 \Rightarrow w \neq w - v$.

It is easy to see that for any $\sigma$, $\sigma \cdot (w = (1, \ldots, 1)) = w \Rightarrow \sigma \cdot w \neq w - v$. This is a contradiction $\Rightarrow v = (0, \ldots, 0)$. It follows that $\sigma \cdot w = w \ \forall w \in F_q^n$. If $\sigma \neq e \Rightarrow \exists i \neq j \in [n]$ s.t. $\sigma(i) = j$. Take $w$ with $w_i \neq w_j$. Then $(\sigma \cdot w)_j = w_i \Rightarrow w_j = w_i$ which is again a contradiction meaning that $\sigma = e$.

□

A basis of a permutation code has the property that no two permutations act in the same way on all the elements of the basis. This is the main motivation for having a basis, a way of describing the elements of the group uniquely. We thus want to have the same result for affine permutation codes and indeed it holds that:

**Proposition 7.2.** $B$ is a basis of an affine permutation codes iff. $\nexists h, h' \in H$ with $h \cdot b = h' \cdot b$ $\forall b \in B$.

*Proof.* Suppose $B$ is a basis and $\exists h, h' \in H$ with $h \cdot b = h' \cdot b$ $\forall b \in B$. Then $b = h^{-1} \cdot (h' \cdot b) = (h^{-1} \circ h') \cdot b$ $\forall b \in B$. Since only the identity stabilizes all elements of the basis, $h^{-1} \circ h' = e \Rightarrow h = h'$.

For the other direction, suppose $\nexists h \neq h' \in H$ with $h \cdot b = h' \cdot b$ $\forall b \in B$. Let $h \in H$ be st. $h \cdot b = b$ $\forall b \in B$ which means $h \cdot b = (e, 0) \cdot b$ and $h$ needs to be $(e, 0)$. □

The whole space $F_q^n$ does not give a very good basis (in terms of number of elements). Using the previous proposition, we now show that:

**Proposition 7.3.** The canonical basis of $F_q^n$, $B = \{b_1 = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \ldots, b_n = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}\}$ is a basis for $H = S \ltimes F_q^n$ if $q > 2$ or $n > 2$. In the case $q = 2$ and $n = 2$ adding the $0^{th}$ vector to $B$ also gives a basis.

*Proof.* Let $(\sigma, v)$ act on $b_1$ to $b_n$. Sowing that knowing the results uniquely determines the $\sigma$ and $v$ would imply using the previous proposition that this is a basis. We have that:

$$w = (\sigma, v) \cdot \begin{pmatrix} 1 \\ 0 \\ \cdots \\ 0 \end{pmatrix} = \sigma \cdot \begin{pmatrix} 1 \\ 0 \\ \cdots \\ 0 \end{pmatrix} + \begin{pmatrix} v_1 \\ \cdots \\ v_n \end{pmatrix} = \begin{pmatrix} v_1 \\ \cdots \\ v_{\sigma(1)}+1 \\ \cdots \\ v_{\sigma(n)} \end{pmatrix}$$

In the same way we find $w' = (\sigma, v) \cdot b_2 = \begin{pmatrix} v_1 \\ \cdots \\ v_{\sigma(2)}+1 \\ \cdots \\ v_{\sigma(n)} \end{pmatrix}$. Knowing these two vectors, there is only one possible $v$ and it is easy to find it: for all position i where $w_i = w'_i$ we have $v_i = w_i$. This happens in all but two positions, namely $\sigma(1)$ and $\sigma(2)$. For $k \in \{1, 2\}$ it holds:

$$v_{\sigma(k)} = \begin{cases} w_{\sigma(k)}, & \text{if } w_{\sigma(k)} + 1 = w'_{\sigma(k)}. \\ w'_{\sigma(k)}, & \text{otherwise.} \end{cases}$$

This works only if $q > 2$ because $+1$ needs to be distinct then $-1$. In case $q = 2, n > 2$ one can use three such vectors by picking, for each position, the value on which at least two of them agree. If both are true, notice that $\sigma \cdot 0 + v = v$ so having $0 \in B$ would determine the value of $v$ uniquely.

Once $v$ is known, for every $i \in [n]$ the following equation holds:

$$\sigma \cdot \begin{pmatrix} 1 \\ 0 \\ \cdots \\ 0 \end{pmatrix} = \begin{pmatrix} v_1 \\ \cdots \\ v_{\sigma(i)}+1 \\ \cdots \\ v_{\sigma(n)} \end{pmatrix} - v = \begin{pmatrix} 0 \\ \cdots \\ 0 \\ 1 \\ 0 \\ \cdots \\ 0 \end{pmatrix}$$

With 1 at position $\sigma(i)$. Hence all $\sigma(i)$'s are know by inspecting these vectors $\Rightarrow \sigma$ is also known and moreover both v and $\sigma$ are uniquely determined.

□

Notice that smaller basis may still exist, depending on the chosen group of permutation. An example is:

**Proposition 7.4.** Let $S$ be the cyclic group of permutations and $H = S \ltimes F_q^n$. $B = \{0 = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}, b_1\}$ is a basis of $H$.

*Proof.* Since $\sigma \cdot 0 + v = v$, $v$ is uniquely determined by the action of the element $(\sigma, v)$ on 0. Therefore also $\sigma \cdot b_1 = \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix}$ is known $\Rightarrow \sigma(1)$ is known. This is enough to determine the whole $\sigma$ because the other positions are moved by the same displacement. $\square$

There are codes that don't have any basis. An example is $H = S_n \ltimes \langle \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \rangle$ where $\langle \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \rangle$ is the subspace of vectors with the same value in all coordinates. Using the approach from the previous proof, $v$ can still be uniquely identified but all $\sigma$'s act in the same way on all vectors so they can not be determined just by inspecting their action. Affine permutation codes for which bases exist are of course desirable.

Another interesting question is whether there is a relation between the size of the minimal bases of the permutation code $S$ and $H = S \ltimes V$. We find the following result:

**Proposition 7.5.** Let $H = S \ltimes F_q^n$. Let $I = \{i_1, \ldots, i_m\}$ be a basis of $S$. Then $B = \{b_{i_1}, \ldots, b_{i_m}, 0\}$ with $b_j =$ the unit basis vector of $F_q^n$ is a basis for $H$. Hence the size of minimal basis of $H$ is less or equal the size of the minimal basis of $S + 1$. In case $q > 2$ or $n > 2$, not adding the $0^{th}$ vector would also work.

*Proof.* As discussed before, $v$ can be uniquely determined due to having the 0 element or from it's action on 2 basis vectors $b_{i_1}, b_{i_2}$. Since $I$ is a basis of $S$, it is easy to see that $\sigma$ can be uniquely determined as well since it's action on a basis vector $b_{i_j}$ would yield another canonical basis vector of $F_q^n$, namely $b_k$ hence $\sigma(i_j) = k$. This determines the action of $\sigma$ on the basis $I$ which in turn determines $\sigma$ uniquely. $\square$

We conclude this section observing that it is in general difficult to make statements about codes $H = S \ltimes V$ where $V$ is potentially not the whole space $F_q^n$. Saying that, one may still be able to work with special choices of $V$ and $S$ when their structure is known.

## References

[1] Peter J. Cameron. Permutation codes. *Eur. J. Comb.*, 31(2):482490, February 2010. URL: https://doi.org/10.1016/j.ejc.2009.03.044, http://dx.doi.org/10.1016/j.ejc.2009.03.044 doi:10.1016/j.ejc.2009.03.044.

[2] Yeow Meng Chee and Punarbasu Purkayastha. Efficient decoding of permutation codes obtained from distance preserving maps. *IEEE International Symposium on Information Theory - Proceedings*, pages 636–640, 07 2012. http://dx.doi.org/10.1109/ISIT.2012.6284273 doi:10.1109/ISIT.2012.6284273.

[3] Yotam Dikstein, Irit Dinur, Prahladh Harsha, and Noga Ron-Zewi. Locally testable codes via high-dimensional expanders, 2020. http://arxiv.org/abs/2005.01045 arXiv:2005.01045.

[4] Brett Hemenway, Rafail Ostrovsky, and Mary Wootters. Local correctability of expander codes, 2015. http://arxiv.org/abs/1304.8129 arXiv:1304.8129.

[5] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948. http://dx.doi.org/10.1002/j.1538-7305.1948.tb01338.x doi:10.1002/j.1538-7305.1948.tb01338.x.

[6] Theo Swart and Hendrik Ferreira. Decoding distance-preserving permutation codes for power-line communications. pages 1 – 7, 10 2007. http://dx.doi.org/10.1109/AFRCON.2007.4401563 doi:10.1109/AFRCON.2007.4401563.