The Nguyen-Vidick sieve algorithm for binary codes

Slotegraaf, M.

M.M. Slotegraaf

The Nguyen-Vidick sieve algorithm for binary codes

Master thesis

July 18, 2022

Supervisor: Dr. P.J. Bruin

Universiteit Leiden
Mathematisch Instituut

Contents

1 Introduction

There are many mathematical similarities between codes and lattices. A code $\mathcal{C} \subset \mathbb{F}_q^n$ is a subspace over a finite field with the Hamming weight as the metric and a lattice $\mathcal{L} \subset \mathbb{R}^n$ is a discrete subspace of a Euclidean vector space with the Euclidean norm as the metric. For an overview of all corresponding definitions used in this thesis, see the end of the introduction. Codes and lattices also have similar applications in cryptography.

In cryptography, decoding systems can be based on finding closest codewords of lattice points from a specific element. Both the closest codeword problem and closest vector problem are NP-hard [APY09; Hof08] and they seem to be exponentially hard for a larger dimension n. Therefore, a lot of research has been based on finding algorithms that solve these problems. An algorithm that finds a codeword of a specific Hamming weight [Ste88], an algorithm that tries to validate whether the nearest codeword is the original message [LB88] and an algorithm for finding the nearest codeword [MO15] are examples of the research that has been done.

Sometimes a technique or algorithm used for lattices is used as a base for an algorithm in coding theory or the other way around. For example, there is the Blum-Kalai-Wasserman algorithm [BKW03] for solving the Learning Parity with Noise problem for lattices. It was fundamental for finding an algorithm for the Learning with Errors problem [Alb+15] for q-ary codes. Another example is the use of locality-sensitive hashing for random binary linear codes [MO15] was the foundation for locality-sensitive hashing for lattices [Laa15b]. And the LLL algorithm for lattices [LLL82] had been adapted to find an analogous algorithm for binary codes [DDW20].

In this thesis, we will give an algorithm to find a codeword with a small Hamming weight based on the Nguyen-Vidick sieve algorithm [NV08]. This is an algorithm that tries to solve the shortest vector problem, by having a list of vectors of a lattice $\mathcal{L}$ and reducing the maximum size of these vectors in every iteration with a sieve factor $\frac{2}{3} < \gamma < 1$. The original Nguyen-Vidick sieve algorithm for lattices is described in section 2.2.

Section 3 consists of some definitions and preliminaries for binary codes. It also describes the adaptation of the LLL algorithm for binary codes. This section forms the groundwork before proceeding to the adaptation of Nguyen-Vidick sieve algorithm for binary codes.

Our analogous algorithm for binary codes is described in section 4 It tries to find a codeword of minimum Hamming weight. It starts with a list of codewords of the code $\mathcal{C}$ and reduces the maximum Hamming weight of the codewords with a sieve factor $\frac{2}{3} < \gamma < 1$. The running time of our algorithm is $\mathcal{O}\left(1.0944\right)^n$.

Below is an overview of corresponding definitions, that are used in this thesis, for codes and for lattices. These definitions are written down more explicitly in the sections 2 and 3. For a more expanded list of analogous definitions between codes and lattices, see [DDW20].

	Lattice $L \subset \mathbb{R}^n$	Code $\mathcal{C} \subset \mathbb{F}_2^n$
Ambient space	$\mathbb{R}^n$	$\mathbb{F}_2^n$
Metric	Euclidean norm: $\|\|x\|\|^2 = \sum x_i^2$	Hamming weight: $\|x\| = \#\{i \mid x_i \neq 0\}$
Support of an element	$\mathbb{R} \cdot x$	$\{i \mid x_i \neq 0\}$
Auxiliary matrix	Gram-Schmidt Orthogonalisation: $\mathbf{b}_i^* = \mathbf{b}_i - \sum_{j<i} \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle} \cdot \mathbf{b}_j^*$	Epipodal matrix: $\mathbf{b}_i^+ = \mathbf{b}_i \wedge \overline{(\mathbf{b}_1^+ \vee \cdots \vee \mathbf{b}_{i-1}^+)}$

Table 1: A dictionary for analogous definitions between codes and lattices.

2 The original algorithms for lattices

We base the sieve algorithm for binary linear codes on the existing Nguyen-Vidick sieve algorithm for lattices [NV08], which is a heuristic variant of the sieve algorithm by Ajtai-Kumar-Sivakumar [AKS01]. Therefore, we first give the definitions, algorithms and theorems that apply to the Nguyen-Vidick sieve algorithm. In this section, the foundation is given before we continue to give the analogous definitions, algorithms and theorems for binary linear codes. We start by giving the definitions, followed by the LLL-algorithm and we end with the Nguyen-Vidick sieve algorithm.

Definition 2.1 (Lattice)**.** Let n be a positive integer. A subset L of the n-dimensional real vector space $\mathbb{R}^n$ is called a *lattice* if there exists a basis $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n$ of $\mathbb{R}^n$ such that

$$L = \sum_{i=1}^{n} \mathbb{Z}\mathbf{b}_i = \left\{ \sum_{i=1}^{n} r_i \mathbf{b}_i \mid r_i \in \mathbb{Z} \right\}.$$

Definition 2.2 (Inner product)**.** For any two vectors $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ and $\mathbf{y} = (y_1, y_2, \ldots, y_n)$ in $\mathbb{R}^n$ we define the *inner product* of $\mathbf{x}$ and $\mathbf{y}$ as

$$\langle \mathbf{x}, \mathbf{y} \rangle = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n.$$

The intuitive notion of length of a vector is called the Euclidean norm.

Definition 2.3 (Euclidean norm)**.** On the n-dimensional Euclidean space $\mathbb{R}^n$, the *Euclidean norm* of the vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is defined by the formula

$$||\mathbf{x}|| := \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}.$$

Definition 2.4 (Gram-Schmidt orthogonalization, [LLL82])**.** Given a basis $\mathbf{B} = \begin{pmatrix} \mathbf{b}_1 \\ \vdots \\ \mathbf{b}_n \end{pmatrix}$ of $\mathbb{R}^n$. For $1 \leq j < i \leq n$, the vectors $\mathbf{b}_i^*$ and the real numbers $\mu_{i,j}$ are inductively defined by

$$\mathbf{b}_i^* = \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \mathbf{b}_j^* \tag{1}$$

$$\mu_{i,j} = \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle}. \tag{2}$$

Now $\mathbf{B}^* = (\mathbf{b}_1^*, \mathbf{b}_2^*, \ldots, \mathbf{b}_n^*)$ is the *Gram-Schmidt orthogonalization* of $\mathbf{B}$.

Note that for every next $\mathbf{b}_i^*$ in equation (1), we know $\mathbf{b}_j^*$ for all $j < i$ and therefore we are also able to calculate $\mu_{i,j}$ defined in equation (2) before calculating equation (1).

2.1 The LLL algorithm for lattices

The well-known LLL algorithm runs in polynomial time and computes a so-called LLL reduced matrix. It has several different applications in, for example, the algorithmic number theory, integer programming or cryptology [NV10]. In the Nguyen-Vidick sieve algorithm, the LLL reduced matrix is used as the input for the sieve algorithm. Hence, the LLL algorithm is used as a first step to start with a 'better' representation of the lattice before starting the sieving.

Definition 2.5 (LLL reduced, [LLL82])**.** A basis $\mathbf{B} = (\mathbf{b}_1^*, \mathbf{b}_2^*, \ldots, \mathbf{b}_n^*)$ is *LLL reduced* if $|\mu_{i,j}| \leq \frac{1}{2}$ for $1 \leq j < i \leq n$ and

$$||\mathbf{b}_i^* + \mu_{i,i-1}\mathbf{b}_{i-1}^*|| \geq ||\mathbf{b}_{i-1}^*|| \quad \text{for} \quad 1 < i \leq n.$$

Algorithm 1 The LLL algorithm, [Hof08]

Input : A basis $(\mathbf{b}_1, \ldots, \mathbf{b}_n)$ of a lattice L.
Output: An LLL reduced basis $(\mathbf{b}_1, \ldots, \mathbf{b}_n)$.

$k \leftarrow 2$;
$\mathbf{b}_1^* \leftarrow \mathbf{b}_1$;
while $k \leq n$ **do**
  for $j = 1$ *to* $k-1$ **do**
    $\mathbf{b}_k \leftarrow \mathbf{b}_k - \lfloor \mu_{k,j} \rceil \mathbf{b}_j^*$;
  end
  if $||\mathbf{b}_k^*|| \geq (\frac{3}{4} - \mu_{k,k-1}^2)||\mathbf{b}_{k-1}||$ **then**
    $k \leftarrow k+1$;
  else
    $\mathbf{b}_{k-1} \leftrightarrow \mathbf{b}_k$;
    $k = \max\{k-1, 2\}$;
  end
end
return $(\mathbf{b}_1, \ldots, \mathbf{b}_n)$

Let $(\mathbf{b}_1, \ldots, \mathbf{b}_n)$ be a basis for a lattice $\mathcal{L}$. Let $\mathbf{b}_{\max} = \max\{||\mathbf{b}_1||, \ldots, ||\mathbf{b}_n||\}$. Then the running time of the LLL algorithm is $\mathcal{O}(n^2 \log(n) + n^2 \log(\mathbf{b}_{\max}))$ [Hof08].

2.2 The Nguyen-Vidick sieve algoritm for lattices

Now we look at the original sieve algorithm by Nguyen and Vidick [NV08]. It is split into two algorithms, where Algorithm 3 uses Algorithm 2. For a lattice $L \subset \mathbb{R}^n$, we define

$$\mathbf{B}_L(R) := \{\mathbf{v}_i \in L \mid \mathbf{v}_i \leq R\}.$$

Algorithm 2 LatticeSieve(S, γ), [NV08]

Input : A subset S of vectors in a lattice L and a sieve factor $\frac{2}{3} < \gamma < 1$.
Output: A subset $S' \subseteq \mathbf{B}_L(\gamma R)$.

$R \leftarrow \max_{\mathbf{v} \in S} ||\mathbf{v}||$;
$C \leftarrow \emptyset$, $S' \leftarrow \emptyset$;
for $\mathbf{v} \in S$ **do**
  if $||\mathbf{v}|| \leq \gamma R$ **then**
    $S' \leftarrow S' \cup \{\mathbf{v}\}$;
  else
    if $\exists \mathbf{c} \in C : ||v - c|| \leq \gamma R$ **then**
      $S' \leftarrow S' \cup \{\mathbf{v} - \mathbf{c}\}$;
    else
      $C \leftarrow C \cup \{\mathbf{v}\}$;
    end
  end
end
return S';

Algorithm 2 is the sieve algorithm. It reduces the maximal Euclidean norm of the vectors in the set S of the lattice L with a sieve factor $\frac{2}{3} < \gamma < 1$. Since the algorithm only sieves vectors with a norm between γR and R, we make the following heuristic assumption:

Heuristic 2.6. We assume that at any stage in Algorithm 3, the vectors in $S \cap C_n(\gamma, R)$ are uniformly distributed in $C_n(\gamma, R) = \{\mathbf{x} \in \mathbb{R}^n, \gamma R \leq ||x|| \leq R\}$.

Algorithm 3 Finding short lattice vectors based on sieving, [NV08]

Input : An LLL-reduced basis $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n]$ of a lattice L, a sieve factor γ such that $\frac{2}{3} < \gamma < 1$, and a positive integer N.

Output: A short non-zero vector of L.

1 $S \leftarrow \emptyset$;
2 **for** $j = 1$ *to* N **do**
3 | $S \leftarrow S \cup \text{Sampling}(\mathbf{B})$;
4 **end**
5 Remove all zero vectors from S;
6 $S_0 \leftarrow S$;
7 **repeat**
8 | $S_0 \leftarrow S$;
9 | $S \leftarrow \text{LatticeSieve}(S, \gamma)$ using Algorithm 2;
10 | Remove all zero vectors from S;
11 **until** $S = \emptyset$;
12 Compute $\mathbf{v}_0 \in S_0$ such that $||\mathbf{v}_0|| = \min\{||\mathbf{v}||, v \in S_0\}$;
13 **return** $\mathbf{v}_0$;

In steps 2-5 of Algorithm 3, we start with sampling N non-zero vectors of reasonable length from the lattice L. The sampling of these vectors is done with some polynomial-time algorithm $\mathcal{K}$, which is described in section 4.2.1 of [NV08]. Under the assumption that Heuristic 2.6 is true, the vectors are uniformly distributed in the spherical shell $C_n(\gamma, R)$. Hence, they shouldn't be biased into any single direction.

Then, in steps 7-11 of Algorithm 3 the maximum Euclidean norm of the vectors in the set S is reduced by the sieve factor γ with the use of Algorithm 2. The first iteration uses the sampled vectors from steps 2-5, the next iterations uses the sieved set from the previous iteration.

At the end of Algorithm 3, we are left with a non-zero vector of our lattice L. However, it is not clear how short this vector is. This is dependent on the number of iterations of steps 7-11. Since we have to repeat these steps until the set S is empty, we have to know how the size of the set S decreases. This can be done through Algorithm 2, when vectors are added to the set of centers C, or by eliminating zero vectors in step 10 of Algorithm 3. Under the assumption that Heuristic 2.6 is true, the number of eliminated zero vectors is negligible until R is too small. Hence, the critical point to determine the complexity of Algorithm 3 is the estimation of the number of codewords in the set centers C [NV08].

Therefore, the next lemma assesses the number of points in the set of centers C. For the proof of this lemma see [NV08, Lemma 4.1].

Lemma 2.7. *Let $n \in \mathbb{N}$ and $\frac{2}{3} < \gamma < 1$. Define $c_{\mathcal{H}} = \frac{1}{\gamma\sqrt{1-\frac{\gamma^2}{4}}}$ and $N_C = c_{\mathcal{H}}^n \lceil 3\sqrt{2\pi}(n+1)^{\frac{3}{2}} \rceil$. Let N be an integer. and S a subset of $C_n(\gamma, R)$ of cardinality N whose points are independently picked at random with uniform distribution.*

1. *If $N_C < N < 2^n$, then for any subset $C \leq S$ of size at least N_C whose points are picked independently at random with uniform distribution, with overwhelming probability for all $\mathbf{v} \in S$, there exists a $\mathbf{c} \in C$ such that $||\mathbf{v} - \mathbf{c}|| \leq \gamma$.*
2. *If $N < 4\sqrt{\frac{\pi}{2n}}\sqrt{\frac{4}{3}}^n$, the expected number of points in S that are at distance at least γ from all the other points in S at least $(1 - \frac{1}{n})N$.*

With the use of Lemma 2.7 the complexity of the Nguyen-Vidick sieve algorithm can be calculated. This gives that the algorithm runs in $\mathcal{O}\left(\left(\frac{4}{3}+\epsilon\right)^n\right)$ time and uses $\mathcal{O}\left(\left(\frac{4}{3}+\epsilon\right)^{\frac{n}{2}}\right)$ bits of memory (see [NV08, page 195] for details and the proof of this complexity).

3 Definitions and preliminaries for binary codes

Before we start adapting the Nguyen-Vidick sieve algorithm from lattices for binary codes, we have to introduce the analogous definitions. In the rest of this section, the LLL algorithm for binary codes is explained and some estimations and heuristics are made which are used for the analysis of the sieve algorithm for binary codes.

Definition 3.1 (Binary linear code)**.** A *binary linear code* $\mathcal{C}$ of length n and dimension k, also written as an $[n, k]$-code, is a linear subspace of $\mathbb{F}_2^n$ of dimension k.

All linear codes can be described by a set of linearly independent generators (the *generator matrix*) or by a system of modular equations (*parity-check representation*). The vectors in $\mathcal{C}$ are called *codewords*.

In this thesis, we will use a generator matrix to represent a code. The codewords are linear combinations of the row vectors of a generator matrix. There are some different binary linear codes that are constructed in a specific way, for example a Hamming code or a Reed-Muller code. For more information about these codes, see [Lin12].

For binary codewords, elements of the vector space $\mathbb{F}_2^n$, we use the standard boolean notations. So $\bar{\mathbf{x}}$ is the bitwise negation, $\mathbf{x} \oplus \mathbf{y}$ is the bitwise XOR, $\mathbf{x} \wedge \mathbf{y}$ is the bitwise AND and $\mathbf{x} \vee \mathbf{y}$ is the bitwise OR.

To define the metric on a lattice, we use the Euclidean norm. For codes we use the Hamming weight for the metric.

Definition 3.2 (Hamming weight)**.** The *support* $\mathrm{Supp}(\mathbf{x})$ of a codeword $\mathbf{x} \in \mathbb{F}_2^n$ is the set of indices of its non-zero coordinates, and its *Hamming weight* $|\mathbf{x}| \in [\![0, n]\!]$ is the cardinality of its support:

$$\mathrm{Supp}(\mathbf{x}) := \{i \in [\![1, n]\!] \mid x_i \neq 0\}, \qquad |\mathbf{x}| := \#\mathrm{Supp}(\mathbf{x}).$$

The *minimum weight* or *minimum distance* of a code $\mathcal{C}$ is

$$\min\{|\mathbf{x}| : \mathbf{x} \in \mathcal{C}, \mathbf{x} \neq 0\}.$$

Now, we look at two examples of binary linear codes. These two examples will be used throughout this thesis.

Example 3.3 ($[7, 4]$-Hamming code)**.** A generator matrix of the Hamming code $\mathcal{C} \subset \mathbb{F}_2^7$ is

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

The dimension of this code is 4 and the minimum weight is 3 [Lin12]. So $(1,0,0,0,0,1,1)$ is a codeword of minimum weight.

Example 3.4 (Random Linear $[10,5]$-code)**.** A generator matrix a random linear code $\mathcal{C} \subset \mathbb{F}_2^{10}$ is

$$G = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \end{pmatrix}.$$

The dimension of this code is 5 and the minimum weight is 3.

3.1 The LLL algorithm for binary codes

Here we give an adaptation of the LLL algorithm for binary codes, as decribed by [DDW20]. To find a LLL reduced basis for a lattice, we first need to find the Gram-Schmidt orthogonalisation. The equivalent definition we use for codes is the epipodal matrix.

Definition 3.5 (Epipodal matrix, [DDW20])**.** Let $\mathbf{B} = \begin{pmatrix} \mathbf{b}_1 \\ \vdots \\ \mathbf{b}_k \end{pmatrix}$ be a matrix of binary codewords. The i-th projection associated to this matrix is defined as

$$\begin{aligned} \pi_i : \mathbb{F}_2^n &\to \mathbb{F}_2^n \\ c &\mapsto c \wedge \overline{(\mathbf{b}_1 \vee \cdots \vee \mathbf{b}_{i-1})}. \end{aligned}$$

where π_1 denotes the identity. The i-th epipodal vector is then defined as:

$$\mathbf{b}_i^+ := \pi_i(\mathbf{b}_i).$$

The matrix $\mathbf{B}^+ := \begin{pmatrix} \mathbf{b}_1^+ \\ \dots \\ \mathbf{b}_k^+ \end{pmatrix} \in \mathbb{F}_2^{k \times n}$ is called the *epipodal matrix* of $\mathbf{B}$.

Example 3.6 ($[7,4]$-Hamming code)**.** The epipodal matrix of the given basis of the Hamming code $\mathcal{C}$ as in Example 3.3 is

$$\mathbf{B}^+ = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}.$$

Example 3.7 (Random Linear $\big[10, 5\big]$-code)**.** The epipodal matrix of the given basis of the random linear code as in Example 3.4 is

$$\mathbf{B}^+ = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Definition 3.8 (Proper basis, [DDW20])**.** A basis $\mathbf{B}$ is called *proper* if all its epipodal vectors $\mathbf{b}_i^+$ are non-zero.

For the LLL reduction algorithm for binary codes, we need the following tie breaking function [DDW20]:

$$\mathrm{TB}_{\mathbf{p}}(\mathbf{y}) = \begin{cases} 0 & \text{if } |\mathbf{p}| \text{ is odd,} \\ 0 & \text{if } y_j = 0 \text{ where } j = \min(\mathrm{Supp}(\mathbf{p})), \\ \frac{1}{2} & \text{otherwise.} \end{cases} \tag{3}$$

Definition 3.9 (LLL reduced basis, [DDW20])**.** A basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_k)$ of an $[n, k]$-code is said to be *LLL reduced* if it is a proper basis, and if $\mathbf{b}_i^+$ is a shortest non-zero codeword of the projected subcode $\pi_i(\mathcal{C}(\mathbf{b}_i, \mathbf{b}_{i+1}))$ for all $i \in [\![1, k-1]\!]$.

Now that we have given the definitions similar to those used in the LLL algorithm for lattices, we finally can formulate the LLL reduction algorithm for codes.

Algorithm 4 LLL reduction algorithm for codes, [DDW20]

Input : A proper basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_k) \in \mathbb{F}_2^{k \times n}$ of a code $\mathcal{C}$
Output: An LLL reduced basis for $\mathcal{C}$

while $\exists i \in [\![0, k-1]\!]$ *such that* $\min(|\pi_i(\mathbf{b}_{i+1})|, |\mathbf{b}_i^+ \oplus \pi_i(\mathbf{b}_{i+1})|) < |\mathbf{b}_i^+|$ **do**
  if $|\pi_i(\mathbf{b}_{i+1}) \wedge \mathbf{b}_i^+| + \mathrm{TB}_{\mathbf{b}_i^+}(\pi_i(\mathbf{b}_{i+1})) > |\mathbf{b}_i^+|/2$ **then**
    $\mathbf{b}_{i+1} \leftarrow \mathbf{b}_{i+1} \oplus \mathbf{b}_i$
  end
  $\mathbf{b}_i \leftrightarrow \mathbf{b}_{i+1}$
end
return $(\mathbf{b}_1, \ldots, \mathbf{b}_n)$

The LLL reduction algorithm for binary codes runs in polynomial time. With $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_k)$ as input the algorithm performs at most $k(n - \frac{k-1}{2}) \max_i |\mathbf{b}_i^+|$ vector operations over $\mathbb{F}_2^n$ [DDW20].

Example 3.10 ($[7,4]$-Hamming code)**.** The LLL reduced matrix of the given basis of the Hamming code as in Example 3.3 is

$$\mathbf{B}_{\mathrm{LLL}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

Note that this LLL reduced basis is the same as the given general matrix in Example 3.3. Hence, the given generator matrix is already LLL reduced.

Example 3.11 (Random Linear $[10,5]$-code)**.** The LLL reduced matrix of the given basis of the random linear code as in Example 3.4 is

$$\mathbf{B}_{\mathrm{LLL}} = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \end{pmatrix}.$$

3.2 Preliminaries for $\mathbb{F}_2^n$

In this subsection, we will look at the code $\mathcal{C} = \mathbb{F}_2^n$. The results in this subsection will be used when we look at codes that are not equal to $\mathbb{F}_2^n$.

For each positive number R and $x \in \mathbb{F}_2^n$, we define the following set:

$$\mathcal{B}_R[x] = \{y \in \mathbb{F}_2^n : |x \oplus y| \leq R\}.$$

In the next lemma and proposition, we give a bound on the cardinality of the set $\mathcal{B}_R[x]$ and of the intersection between the sets $\mathcal{B}_R[0]$ and $\mathcal{B}_{R'}[x]$.

Lemma 3.12. *Let $0 \leq R \leq \frac{n}{2}$ be some integer. A upper bound for the cardinality of the set $\mathcal{B}_R[0]$ is*

$$\#\mathcal{B}_R[0] \leq \binom{n}{R} \frac{n-(R-1)}{n-(2R-1)}.$$

Proof. We have $\mathcal{B}_R[0] = \{y \in \mathbb{F}_2^n : |y| \leq R\}$. So

$$\#\mathcal{B}_R[0] = \sum_{i=0}^{R} \binom{n}{i}.$$

We know that

$$\begin{aligned}\frac{\binom{n}{R}+\binom{n}{R-1}+\binom{n}{R-2}+\cdots}{\binom{n}{R}} &= 1+\frac{R}{n-R+1}+\frac{R(R-1)}{(n-R+1)(n-R+2)}+\cdots \\ &\leq 1+\frac{R}{n-R+1}+\left(\frac{R}{n-R+1}\right)^2+\cdots \\ &= \frac{1}{1-\frac{R}{n-(R-1)}} \\ &= \frac{n-(R-1)}{n-(2R-1)}\end{aligned}$$

Hence, we have

$$\#\mathcal{B}_R[0] \leq \binom{n}{R}\frac{n-(R-1)}{n-(2R-1)}.$$

□

Let n be a positive integer and $k \in \mathbb{Q} \setminus \mathbb{Z}$, then we define

$$\binom{n}{k} := 0.$$

The next proposition gives the cardinality of the intersection between the sets $\mathcal{B}_R[0]$ and $\mathcal{B}_{R'}[x]$ for some codeword x. It is an adaptation of Proposition 2.4.1 from [Dib13].

Proposition 3.13. *Let R, R' be positive integers. Let x be a codeword of length n and w its Hamming weight. Then the size of the subset $\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x]$ is*

$$\sum_{0\leq\delta_1\leq R}\sum_{0\leq\delta_2\leq R'}\binom{w}{\frac{\delta_1-\delta_2+w}{2}}\binom{n-w}{\frac{\delta_1+\delta_2-w}{2}}.$$

Proof. Let y be a codeword. Then $y \in \mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x]$ if and only if

$$|y| = \delta_1 \quad \text{and} \quad |y \oplus x| = \delta_2, \tag{4}$$

where $\delta_1 \in [0, R]$ and $\delta_2 \in [0, R']$. Now we want to know how many codewords there are that satisfy these conditions. For this, we look at the number of possible supports of a codeword y with Hamming weight δ_1. This is divided into the numbers a_1, the number of bits where y and x both have value 1, and a_0, the number of bits where y has value 1 and x has value 0. Hence, we define

$$a_i := \#\{j \in [1, n] : y_j = 1 \text{ and } x_j = i\}, \quad i \in \{0, 1\}. \tag{5}$$

When we combine equations (4) and (5), we get

$$\delta_1 = a_0 + a_1 \quad \text{and} \quad \delta_2 = a_0 + (w - a_1).$$

Therefore,

$$a_0 = \frac{\delta_1 + \delta_2 - w}{2} \quad \text{and} \quad a_1 = \frac{\delta_1 - \delta_2 + w}{2}.$$

Now, we can give an expression to calculate the number of elements in $\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x]$:

$$\sum_{0 \le \delta_1 \le R} \sum_{0 \le \delta_2 \le R'} \binom{w}{\frac{\delta_1 - \delta_2 + w}{2}} \binom{n-w}{\frac{\delta_1 + \delta_2 - w}{2}}$$

□

We can generalize Proposition 3.13 to give the cardinality of the intersection between the sets $\mathcal{B}_R[x]$ and $\mathcal{B}_{R'}[y]$.

Corollary 3.14. *Let R, R' be some integers. Let x, y be codewords of length n and $|x|$ and $|y|$ their respective Hamming weights. Then the size of the subset $\mathcal{B}_R[x] \cap \mathcal{B}_{R'}[y]$ is*

$$\sum_{0 \le \delta_1 \le R} \sum_{0 \le \delta_2 \le R'} \binom{|t|}{\frac{\delta_1 - \delta_2 + |t|}{2}} \binom{n-|t|}{\frac{\delta_1 + \delta_2 - |t|}{2}}.$$

Proof. The cardinality of intersection of the two subsets $\mathcal{B}_R[x]$ and $\mathcal{B}_{R'}[y]$ of $\mathbb{F}_2^n$ can be written as

$$\begin{aligned}
\#(\mathcal{B}_R[x] \cap \mathcal{B}_{R'}[y]) &= \{z \in \mathbb{F}_2^n : |z \oplus x| \le R \text{ and } |z \oplus y| \le R'\} \\
&\overset{\text{substitute } z = x \oplus w}{=} \{w \in \mathbb{F}_2^n : |w| \le R \text{ and } |x \oplus w \oplus y| \le R'\} \\
&\overset{\text{substitute } t = y \oplus x}{=} \{w \in \mathbb{F}_2^n : |w| \le R \text{ and } |w \oplus t| \le R'\} \\
&\overset{\text{Proposition 3.13}}{=} \sum_{0 \le \delta_1 \le R} \sum_{0 \le \delta_2 \le R'} \binom{|t|}{\frac{\delta_1 - \delta_2 + |t|}{2}} \binom{n-|t|}{\frac{\delta_1 + \delta_2 - |t|}{2}}
\end{aligned}$$

□

Now, we want to give a lower bound for the intersection between the sets $\mathcal{B}_R[0]$ and $\mathcal{B}_{R'}[x]$. First, we give the following lemma (see [Car06, Lemma 1]).

Lemma 3.15. *Let n be positive integer and $k \le \frac{1}{2}n$. Then*

$$\binom{n}{k} \le 2^n \cdot \exp\left(-\frac{(n-2k)^2}{2n}\right).$$

We want to be able to give a better asymptotic estimate for a binomial coefficient. Based on Lemma 2.2.2 from [Dib13], we will use the following approximation:

Heuristic 3.16. Let n, k be a positive integers with $k \leq \frac{1}{2}n$. Then

$$\binom{n}{k} \approx \frac{2^n}{\sqrt{\frac{1}{2}\pi \cdot n}} \cdot \exp\left(-\frac{(n-2k)^2}{2n}\right).$$

Lemma 3.17. *Let R be an integer with $R \leq \frac{1}{2}n$. Let $\frac{2}{3} < \gamma < 1$ and $R' = \gamma R$. Let x be a codeword of length n and $w \leq R$ be its Hamming weight. Let $r_- = \frac{(5-\gamma)-\sqrt{(\gamma-5)^2-16}}{8}$. Assume that Heuristic 3.16 is true, then*

$$\frac{\#(\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x])}{\#\mathcal{B}_R[0]} \gtrsim \begin{cases} \frac{1}{n\sqrt{n}} \frac{\sqrt{8}}{\sqrt{\pi}} \exp(-nG_1(\gamma)) & \text{if } 0 \leq \frac{R}{n} \leq r_- \\ \frac{1}{n\sqrt{n}} \frac{\sqrt{8}}{\sqrt{\pi}} \exp(-nG_2(\gamma)) & \text{if } r_- \leq \frac{R}{n} \leq \frac{1}{2}. \end{cases}$$

where

$$G_1(\gamma) = \frac{7\gamma^2 - (\gamma^2+3\gamma-1)\sqrt{4\gamma+1} - \gamma + 1}{2(\sqrt{4\gamma+1}+1)}$$
$$G_2(\gamma) = \frac{28\gamma^2 - (4\gamma^2+6\gamma-1)\sqrt{1+8\gamma} - 2\gamma + 1}{2(\gamma^2+(\gamma^2-2\gamma+1)\sqrt{1+8\gamma} - 2\gamma + 1)}$$

Proof. First, we give a lower bound

$$\begin{aligned} \#(\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x]) &= \sum_{0 \leq \delta_1 \leq R} \sum_{0 \leq \delta_2 \leq R'} \binom{w}{\frac{\delta_1 - \delta_2 + w}{2}} \binom{n-w}{\frac{\delta_1 + \delta_2 - w}{2}} \\ &\geq \sum_{0 \leq \delta_2 \leq R'} \binom{w}{\frac{R - \delta_2 + w}{2}} \binom{n-w}{\frac{R + \delta_2 - w}{2}} \\ &\geq \binom{w}{\frac{R - R' + w}{2}} \binom{n-w}{\frac{R + R' - w}{2}} \end{aligned} \tag{6}$$

We can combine Lemma 3.12 with equation (6) to give a lower bound for the fraction $\frac{\#(\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x])}{\#\mathcal{B}_R[0]}$. This gives

$$\frac{\#(\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x])}{\#\mathcal{B}_R[0]} \geq \frac{n-(2R-1)}{n-(R-1)} \frac{\binom{w}{\frac{R-R'+w}{2}}\binom{n-w}{\frac{R+R'-w}{2}}}{\binom{n}{R}}.$$

Note that $\frac{n-(2R-1)}{n-(R-1)} \geq \frac{1}{n}$. With the use of Heuristic 3.16, we get

$$\frac{\#(\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x])}{\#\mathcal{B}_R[0]} \gtrsim \frac{1}{n} \frac{\sqrt{2}}{\sqrt{\pi}} \frac{\sqrt{n}}{\sqrt{w(n-w)}} \exp\left(-\frac{(R-R')^2}{2w} - \frac{(n-R-R')^2}{2(n-w)} + \frac{(n-R)^2}{2n}\right).$$

Now, we substitute $R = rn$, $R' = \gamma rn$ and $w = un$, then we get

$$\frac{\#(\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x])}{\#\mathcal{B}_R[0]} \gtrsim \frac{1}{n\sqrt{n}} \frac{\sqrt{2}}{\sqrt{\pi}} \frac{1}{\sqrt{u(1-u)}} \exp\left(n \left(-\frac{(1-\gamma)^2 r^2}{2u} - \frac{(1-(1+\gamma)r)^2}{2(1-u)} + \frac{(1-2r)^2}{2} \right) \right).$$

We define

$$F_\gamma(u,r) := \frac{(1-\gamma)^2 r^2}{2u} + \frac{(1-(1+\gamma)r)^2}{2(1-u)} - \frac{(1-2r)^2}{2}.$$

Since $0 \le r \le \frac{1}{2}$ and $(1-\gamma)r \le u \le \frac{1}{2}$, we know that $u(1-u) \le \frac{1}{4}$. Therefore

$$\frac{\#(\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x])}{\#\mathcal{B}_R[0]} \gtrsim \frac{1}{n\sqrt{n}} \frac{\sqrt{8}}{\sqrt{\pi}} \exp\left(-nF_\gamma(u,r)\right). \tag{7}$$

We want to find an upper bound for $F_\gamma(u,r)$. First, we determine the roots of $\frac{\partial F_\gamma}{\partial u}$.

$$\begin{aligned} \frac{\partial}{\partial u} F_\gamma &= \frac{(2\gamma r u - \gamma r + r - u)(\gamma r + 2ru - r - u)}{2u^2(1-u)^2} \\ &= \frac{((2\gamma r - 1)u + (1-\gamma r))((2r-1)u - (1-\gamma r))}{2u^2(1-u)^2} \end{aligned}$$

Therefore the roots of $\frac{\partial F}{\partial u}$ are at $u_1 = \frac{(1-\gamma)r}{1-2\gamma r}$ and $u_2 = \frac{(1-\gamma)r}{2r-1}$. Since $0 < r < \frac{1}{2}$, we have $u_2 < 0$. Therefore the root u_2 is not in the interval. So the global maximum of $F_\gamma(u,r)$ is at the root u_1 or at the edges r or $(1-\gamma)r$.

Since

$$F_\gamma(r,r) - F_\gamma(u_1,r) \ge \frac{-\gamma^2 r(r^2 - r + 1)}{2(r-1)} > 0$$

for all r, the global maximum of $F_\gamma(u,r)$ is never at the root u_1. So the global maximum of $F_\gamma(u,r)$ is at the edges. Finally, we want to know whether $F_\gamma(r,r)$ or $F_\gamma((1-\gamma)r,r)$ is bigger for $0 < r < \frac{1}{2}$. Since neither one of those is the biggest for all $0 < r < \frac{1}{2}$, we want to determine a turning point for the maximum of $F_\gamma(u,r)$.

$$F_\gamma(r,r) - F_\gamma((1-\gamma)r,r) = -\frac{-\gamma^2 r(4r^2 - (5-\gamma)r + 1)}{2(\gamma r - r + 1)(r-1)} = 0.$$

So the turning point is at $r_\pm = \frac{(5-\gamma) \pm \sqrt{(\gamma-5)^2 - 16}}{8}$. Since $r_- < \frac{1}{2}$ and $r_+ > \frac{1}{2}$ for $\frac{2}{3} \le \gamma \le 1$, the turning point is at r_-. Hence, we get

$$F_\gamma(u,r) \le \begin{cases} F_\gamma(r,r) & \text{if } 0 \le r \le r_- \\ F_\gamma((1-\gamma)r,r) & \text{if } r_- \le r \le \frac{1}{2}. \end{cases}$$

Now, we want to calculate $\frac{d}{dr}F_\gamma(u,r)$ where u is at both edges and equate them to 0. First, we look at the edge r. Then we get

$$\begin{aligned}\frac{d}{dr}F_\gamma(r,r) &= \frac{d}{dr}\left(\frac{(1-\gamma)^2 r}{2} + \frac{(1-(1+\gamma)r)^2}{2(1-r)} - \frac{(1-2r)^2}{2}\right)\\ &= -\frac{8r^3-(20-4\gamma)r^2+(16-8\gamma)r-\gamma^2+4\gamma-4}{2(r-1)^2}\\ &= -\frac{(4r^2-6r-\gamma+2)(2r+\gamma-2)}{2(1-r)^2}\end{aligned}$$

This derivative is equal to 0 if $2r+\gamma-2=0$ or $4r^2-6r-\gamma+2=0$. Since $r<\frac{1}{2}$ and $\frac{2}{3}<\gamma<1$ the equation $2r+\gamma-2=0$ will not take place. Hence $4r^2-6r-\gamma+2=0$. So we get the roots $\rho_\pm=\frac{6\pm\sqrt{4+16\gamma}}{8}$. Since $\rho_+>\frac{1}{2}$ for $\frac{2}{3}<\gamma<1$, the root ρ_+ is not in our interval. So the root $\rho_1(\gamma)=\frac{6-\sqrt{4+16\gamma}}{8}$ is the only root for the derivative of $F_\gamma(r,r)$ that satisfies the conditions $\frac{2}{3}<\gamma<1$ and $0<r<\frac{1}{2}$. Now, we substitute $r=\rho_1(\gamma)=\frac{6-\sqrt{4+16\gamma}}{8}$ in $F_\gamma(r,r)$:

$$G_1(\gamma)=F_\gamma(\rho_1(\gamma),\rho_1(\gamma))=\frac{7\gamma^2-(\gamma^2+3\gamma-1)\sqrt{4\gamma+1}-\gamma+1}{2(\sqrt{4\gamma+1}+1)}.$$

Second, we take the edge $(1-\gamma)r$. Then we get

$$\begin{aligned}\frac{d}{dr}F_\gamma((1-\gamma)r,r) &= \frac{d}{dr}\left(\frac{1}{2}(1-\gamma)r + \frac{(1-(1+\gamma)r)^2}{2(1-(1-\gamma)r)} - \frac{1}{2}(1-2r)^2\right)\\ &= -\frac{(4\gamma^2-8\gamma+4)r^3-(4\gamma^2-14\gamma+10)r^2-(8\gamma-8)r+2\gamma-2}{(\gamma r-r+1)^2}\\ &= -\frac{2(\gamma-1)(2(\gamma-1)r^2+3r-1)(r-1)}{(\gamma r-r+1)^2}\end{aligned}$$

This derivative is equal to 0 if $r-1=0$ or $2(\gamma-1)r^2+3r-1=0$. Since $r<\frac{1}{2}$, the first case will not occur. Hence $2(\gamma-1)r^2+3r-1=0$. So we get the roots $\rho_\pm=\frac{-3\pm\sqrt{1+8\gamma}}{4(\gamma-1)}$. Since $\rho_->\frac{1}{2}$ for $\frac{2}{3}<\gamma<1$. So the root $\rho_2(\gamma)=\frac{3-\sqrt{1+8\gamma}}{4(1-\gamma)}$ is the only root for the derivative of $F_\gamma((1-\gamma)r,r)$ that satisfies the conditions $\frac{2}{3}<\gamma<1$ and $0<r<\frac{1}{2}$. Now we substitute $r=\rho_2(\gamma)=\frac{3-\sqrt{1+8\gamma}}{4(1-\gamma)}$ in $F_\gamma((1-\gamma)r,r)$:

$$G_2(\gamma)=F_\gamma((1-\gamma)\rho_2(\gamma),\rho_2(\gamma))=\frac{28\gamma^2-(4\gamma^2+6\gamma-1)\sqrt{1+8\gamma}-2\gamma+1}{2(\gamma^2+(\gamma^2-2\gamma+1)\sqrt{1+8\gamma}-2\gamma+1)}$$

Concluding, we get

$$\frac{\#(\mathcal{B}_R[0]\cap\mathcal{B}_{R'}[x])}{\#\mathcal{B}_R[0]} \gtrsim \begin{cases}\frac{1}{n\sqrt{n}}\frac{\sqrt{8}}{\sqrt{\pi}}\exp(-nG_1(\gamma)) & \text{if } 0\le r\le r_-\\ \frac{1}{n\sqrt{n}}\frac{\sqrt{8}}{\sqrt{\pi}}\exp(-nG_2(\gamma)) & \text{if } r_-\le r\le\frac{1}{2}.\end{cases}$$

□

Remark 3.18. When we look at the values for $G_1(\gamma)$ and $G_2(\gamma)$ from Lemma 3.17, we see that $G_1(\gamma) > G_2(\gamma)$ for all $\frac{2}{3} < \gamma < 1$. Hence, from here we continue to use that

$$\frac{\#(\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x])}{\#\mathcal{B}_R[0]} \gtrsim \frac{1}{n\sqrt{n}} \frac{\sqrt{8}}{\sqrt{\pi}} \exp(-nG_1(\gamma)).$$

In Figure 1, we can see the numerical evidence that $\exp(G_1(\gamma)) > \exp(G_2(\gamma))$ is true for all $\frac{2}{3} < \gamma < 1$. Furthermore, $1.0461 < \exp(G_1(\gamma)) < 1.1235$ for $\frac{2}{3} < \gamma < 1$.

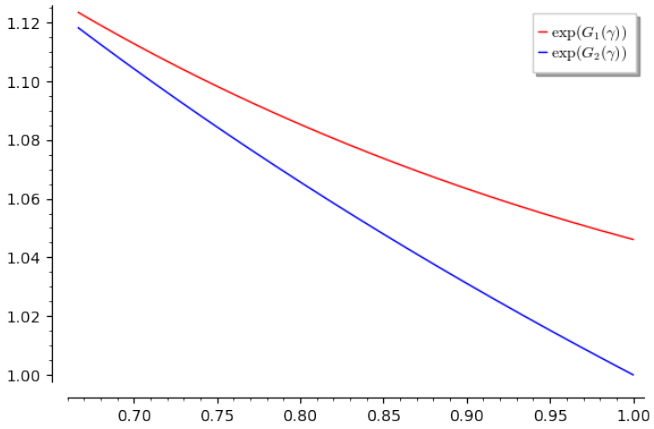


Figure 1: The numerical evaluation of $\exp(G_1(\gamma))$ and $\exp(G_2(\gamma))$ for $\frac{2}{3} < \gamma < 1$.

3.3 Preliminaries for codes $\mathcal{C}$

In this subsection, we will look at binary linear codes $\mathcal{C}$ and the relation between a binary linear $[n, k]$-code $\mathcal{C}$ and $\mathbb{F}_2^n$. For every positive real number R, we define the following set:

$$\mathcal{B}_R^{\mathcal{C}}[x] = \{y \in \mathcal{C} : |x \oplus y| \leq R\}.$$

The cardinality of a code $\mathcal{C}$ with dimension k is 2^k. Hence $\#\mathcal{C} = 2^{k-n}\#\mathbb{F}_2^n$.

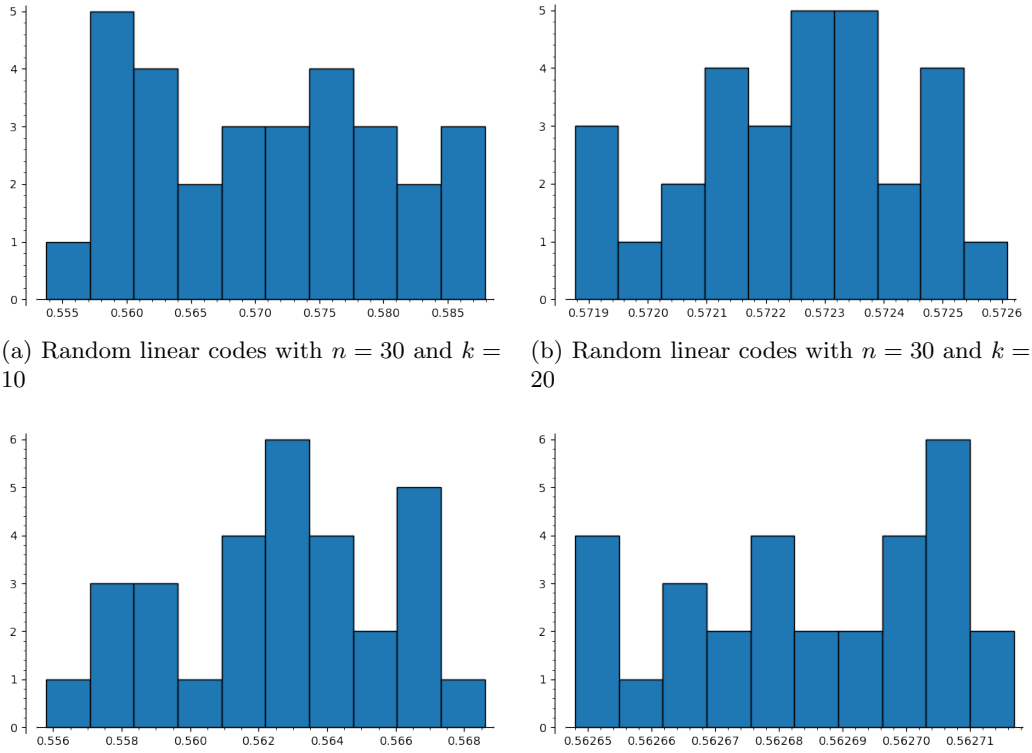
Since Algorithm 6 at steps 2-4 only samples codewords of maximum Hamming weight $\frac{1}{2}n$ and in Lemma 3.17 the integer R is smaller than $\frac{1}{2}n$, we make the following assumption:

Heuristic 3.19. Let $\mathcal{C}$ be a binary linear $[n, k]$-code. Then

$$\frac{\#\mathcal{B}_{\frac{n}{2}}^{\mathcal{C}}[0]}{\#\mathcal{C}} \approx \frac{\#\mathcal{B}_{\frac{n}{2}}[0]}{\#\mathbb{F}_2^n} \approx \frac{1}{2}.$$

In Figure 2, the fraction $\frac{\#\mathcal{B}_{n/2}^{\mathcal{C}}[0]}{\#\mathcal{C}}$ is plotted for 30 randomly generated linear codes. In these plots, we can see that Heuristic 3.19 is a useful and accurate estimation. For the

Hamming $[15, 11]$-code and $[31, 26]$-code, the fraction $\frac{\#\mathcal{B}^{\mathcal{C}}_{n/2}[0]}{\#\mathcal{C}}$ is equal to $\frac{1}{2}$ in both cases. The fraction $\frac{\#\mathcal{B}^{\mathcal{C}}_{n/2}[0]}{\#\mathcal{C}}$ for the Reed Muller $[32, 16]$-code and $[16, 11]$-code are respectively 0.7786 and 0.7124. What these numerical examples show is that most codewords in a code have Hamming weight smaller than $\frac{1}{2}n$. Hence, when we sample N random codewords in Algorithm 6, the number of codewords that satisfy the condition that the Hamming weight is smaller than or equal to $\frac{1}{2}n$ is large enough.



(a) Random linear codes with $n = 30$ and $k = 10$

(b) Random linear codes with $n = 30$ and $k = 20$

(c) Random linear codes with $n = 40$ and $k = 13$

(d) Random linear codes with $n = 40$ and $k = 26$

Figure 2: Histogram of $\frac{\#\mathcal{B}^{\mathcal{C}}_{n/2}[0]}{\#\mathcal{C}}$ with 30 different random linear codes

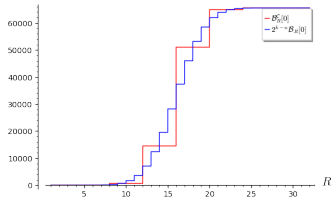
Now, we want an estimation for the ratio between the cardinality of the sets $\mathcal{B}^{\mathcal{C}}_R[0]$ and $\mathcal{B}_R[0]$. Therefore, the following assumption is needed.

Heuristic 3.20. Let $\mathcal{C}$ be a binary linear $[n, k]$-code. Let R be an integer. Then

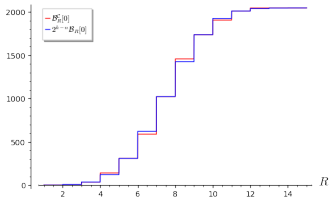
$$\#\mathcal{B}^{\mathcal{C}}_R[0] \approx 2^{k-n}\#\mathcal{B}_R[0].$$

For some numerical evidence for this heuristic see Figure 3. In these plots, $\#\mathcal{B}^{\mathcal{C}}_R[0]$ (in red) and $2^{k-n}\#\mathcal{B}_R[0]$ (in blue) is shown for all $1 < R < n$ for the Reed-Muller $[32, 16]$-code, the Hamming $[15, 11]$-code, a random linear $[30, 10]$-code and a random linear

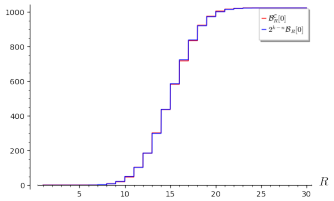
$[30, 20]$-code. We can see that Heuristic 3.20 is an accurate assumption for the size of $\mathcal{B}_R^{\mathcal{C}}[0]$.



(a) Reed-Muller code with $n = 32$, $k = 16$ and minimum weight 8.



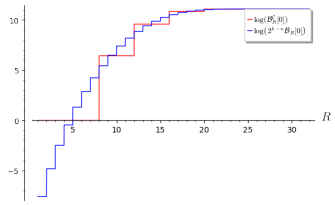
(b) Hamming code with $n = 15$, $k = 11$ and minimum weight 3.



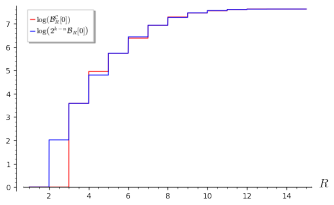
(c) Random Linear code with $n = 30$, $k = 10$ and minimum weight 8.

Figure 3: $\#\mathcal{B}_R^{\mathcal{C}}[0]$ and $2^{k-n}\#\mathcal{B}_R[0]$ for three different codes $\mathcal{C}$

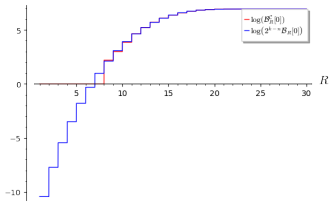
Since $\#\mathcal{B}_R^{\mathcal{C}}[0]$ and $2^{k-n}\#\mathcal{B}_R[0]$ is small for R close to 1, Figure 3 does not give a clear view on the differences between the two plots. To more clearly show this, we have also plot the natural logarithm in Figure 4.



(a) Reed-Muller code with $n = 32$, $k = 16$ and minimum weight 8.



(b) Hamming code with $n = 15$, $k = 11$ and minimum weight 3.



(c) Random Linear code with $n = 30$, $k = 10$ and minimum weight 8.

Figure 4: $\log\left(\#\mathcal{B}_R^{\mathcal{C}}[0]\right)$ and $\log\left(2^{k-n}\#\mathcal{B}_R[0]\right)$ for three different codes $\mathcal{C}$

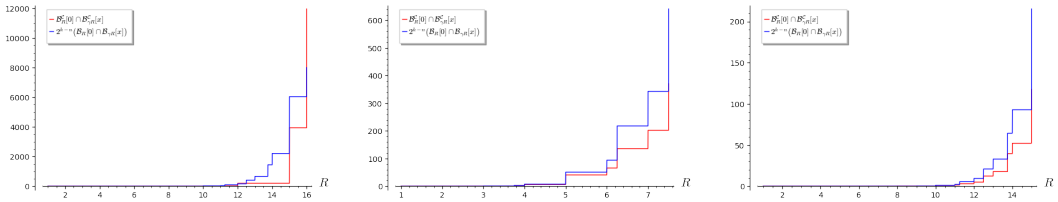
Now, we want an estimate for the relation between the cardinalities of $\mathcal{B}_R^{\mathcal{C}}[0] \cap \mathcal{B}_{R'}^{\mathcal{C}}[x]$ and $\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x]$. For this, we make the following assumption:

Heuristic 3.21. Let $\mathcal{C}$ be a binary linear $[n, k]$-code and $x \in \mathcal{C}$. Let R be an integer with $R \leq \frac{1}{2}n$. Let $\frac{2}{3} < \gamma < 1$ and $R' = \gamma R$. Then

$$\#(\mathcal{B}_R^{\mathcal{C}}[0] \cap \mathcal{B}_{R'}^{\mathcal{C}}[x]) \approx 2^{k-n}\#(\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x]).$$

For some numerical evidence for this heuristic see Figure 5. In these plots, we have taken 10 random codewords of the code $\mathcal{C}$ with a maximum Hamming weight $\frac{1}{2}n$. Then we calculated $\#\left(\mathcal{B}_R^{\mathcal{C}}[0] \cap \mathcal{B}_{R'}^{\mathcal{C}}[x]\right)$ and $\#\left(\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x]\right)$ for all these 10 codewords and

took the average of these cardinalities. We can see that the lines for a code $\mathcal{C}$ (in red) and for $\mathbb{F}_2^n$ differs more than in Figure 3, but still is an accurate assumption.



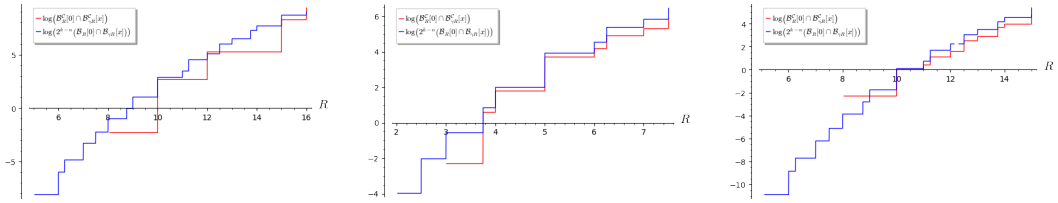
(a) Reed-Muller code with $n = 32$, $k = 16$ and minimum weight 8.

(b) Hamming code with $n = 15$, $k = 11$ and minimum weight 3.

(c) Random Linear code with $n = 30$, $k = 10$ and minimum weight 8.

Figure 5: $\#(\mathcal{B}_R^{\mathcal{C}}[0] \cap \mathcal{B}_{\gamma R}^{\mathcal{C}}[x])$ and $2^{k-n}\#(\mathcal{B}_R[0] \cap \mathcal{B}_{\gamma R}[x])$ for three different codes $\mathcal{C}$ with $\gamma = 0.8$

For small R, Figure 6 will shows more clearly how $\#(\mathcal{B}_R^{\mathcal{C}}[0] \cap \mathcal{B}_{R'}^{\mathcal{C}}[x])$ and $2^{k-n}\#(\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x])$ differ. Note that the lines stop at the point where $\mathcal{B}_R^{\mathcal{C}}[0] \cap \mathcal{B}_{R'}^{\mathcal{C}}[x] = \emptyset$. In this case, it is at the point where R is equal to the minimum weight of the code $\mathcal{C}$. Hence, for least one of the ten codewords that are generated, the cardinality $\#(\mathcal{B}_R^{\mathcal{C}}[0] \cap \mathcal{B}_{R'}^{\mathcal{C}}[x])$ is not equal to zero for R equal to the minimum weight of the code.



(a) Reed-Muller code with $n = 32$, $k = 16$ and minimum weight 8.

(b) Hamming code with $n = 15$, $k = 11$ and minimum weight 3.

(c) Random Linear code with $n = 30$, $k = 10$ and minimum weight 8.

Figure 6: $\log\left(\#(\mathcal{B}_R^{\mathcal{C}}[0] \cap \mathcal{B}_{\gamma R}^{\mathcal{C}}[x])\right)$ and $\log\left(2^{k-n}\#(\mathcal{B}_R[0] \cap \mathcal{B}_{\gamma R}[x])\right)$ for three different codes $\mathcal{C}$ with $\gamma = 0.8$

Now that we have settled some estimations for the ratio between the cardinality of the codes $\mathcal{C}$ and $\mathbb{F}_2^n$, we are able to give a lower bound for the fraction $\frac{\#(\mathcal{B}_R^{\mathcal{C}}[0] \cap \mathcal{B}_{R'}^{\mathcal{C}}[x])}{\#\mathcal{B}_R^{\mathcal{C}}[0]}$ analogous to Lemma 3.17.

Corollary 3.22. *Let $\mathcal{C}$ be a binary linear $[n,k]$-code and $x \in \mathcal{C}$. Let R be an integer with $R \leq \frac{1}{2}n$. Let $\frac{2}{3} < \gamma < 1$ and $R' = \gamma R$. Let $r_- = \frac{(5-\gamma)-\sqrt{(\gamma-5)^2-16}}{8}$. Assume that the heuristics 3.16, 3.20 and 3.21 are true, then*

$$\frac{\#(\mathcal{B}_R^{\mathcal{C}}[0] \cap \mathcal{B}_{R'}^{\mathcal{C}}[x])}{\#\mathcal{B}_R^{\mathcal{C}}[0]} \gtrsim \begin{cases} \frac{1}{n\sqrt{n}} \frac{\sqrt{8}}{\sqrt{\pi}} \exp(-nG_1(\gamma)) & \text{if } 0 \leq \frac{R}{n} \leq r_- \\ \frac{1}{n\sqrt{n}} \frac{\sqrt{8}}{\sqrt{\pi}} \exp(-nG_2(\gamma)) & \text{if } r_- \leq \frac{R}{n} \leq \frac{1}{2}. \end{cases}$$

where

$$G_1(\gamma) = \frac{7\gamma^2 - (\gamma^2+3\gamma-1)\sqrt{4\gamma+1} - \gamma + 1}{2(\sqrt{4\gamma+1}+1)}$$
$$G_2(\gamma) = \frac{28\gamma^2 - (4\gamma^2+6\gamma-1)\sqrt{1+8\gamma} - 2\gamma + 1}{2(\gamma^2+(\gamma^2-2\gamma+1)\sqrt{1+8\gamma} - 2\gamma + 1)}$$

Proof. By Lemma 3.17, we know that

$$\frac{\#(\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x])}{\#\mathcal{B}_R[0]} \gtrsim \begin{cases} \frac{1}{n\sqrt{n}} \frac{\sqrt{8}}{\sqrt{\pi}} \exp(-nG_1(\gamma)) & \text{if } 0 \leq r \leq r_- \\ \frac{1}{n\sqrt{n}} \frac{\sqrt{8}}{\sqrt{\pi}} \exp(-nG_2(\gamma)) & \text{if } r_- \leq r \leq \frac{1}{2}. \end{cases}$$

From the heuristics 3.20 and 3.21, we can conclude that

$$\begin{aligned} \frac{\#(\mathcal{B}_R^{\mathcal{C}}[0] \cap \mathcal{B}_{R'}^{\mathcal{C}}[x])}{\#\mathcal{B}_R^{\mathcal{C}}[0]} &\approx \frac{2^{k-n}\#(\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x])}{2^{k-n}\#\mathcal{B}_R[0]} \\ &= \frac{\#(\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x])}{\#\mathcal{B}_R[0]} \\ &\gtrsim \begin{cases} \frac{1}{n\sqrt{n}} \frac{\sqrt{8}}{\sqrt{\pi}} \exp(-nG_1(\gamma)) & \text{if } 0 \leq r \leq r_- \\ \frac{1}{n\sqrt{n}} \frac{\sqrt{8}}{\sqrt{\pi}} \exp(-nG_2(\gamma)) & \text{if } r_- \leq r \leq \frac{1}{2}. \end{cases} \end{aligned}$$

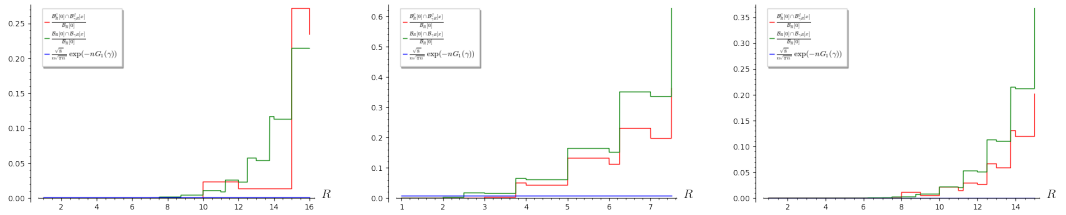
□

Since $G_1(\gamma)$ and $G_2(\gamma)$ are the same for Corollary 3.22 and Lemma 3.17, it still holds, as shown in Figure 1, that $\exp(G_1(\gamma)) > \exp(G_2(\gamma))$ for all $\frac{2}{3} < \gamma < 1$. Hence, we get

Remark 3.23. Similar to Remark 3.18, we can conclude that

$$\frac{\#(\mathcal{B}_R^{\mathcal{C}}[0] \cap \mathcal{B}_{R'}^{\mathcal{C}}[x])}{\#\mathcal{B}_R^{\mathcal{C}}[0]} \gtrsim \frac{\sqrt{8}}{n\sqrt{\pi n}} \exp(-nG_1(\gamma)).$$

In Figure 7, the fractions $\frac{\#(\mathcal{B}_R^{\mathcal{C}}[0] \cap \mathcal{B}_{R'}^{\mathcal{C}}[x])}{\#\mathcal{B}_R^{\mathcal{C}}[0]}$ and $\frac{\#(\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x])}{\#\mathcal{B}_R[0]}$ and the approximate lower bound $\frac{\sqrt{8}}{n\sqrt{\pi n}} \exp(-nG_1(\gamma))$ from Lemma 3.17 are plotted. This gives a view on what the difference is between the fractions for a code $\mathcal{C}$ and $\mathbb{F}_2^n$.



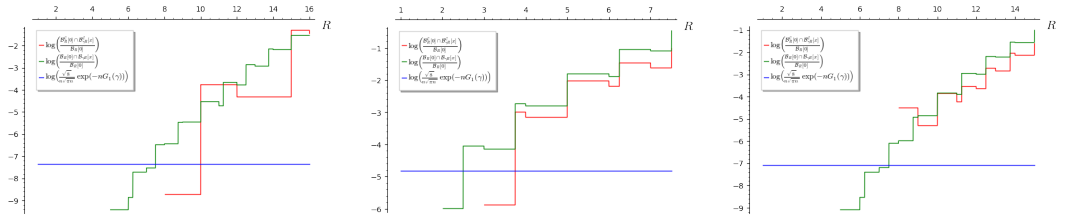
(a) Reed-Muller code with $n = 32$, $k = 16$ and minimum weight 8.

(b) Hamming code with $n = 15$, $k = 11$ and minimum weight 3.

(c) Random Linear code with $n = 30$, $k = 10$ and minimum weight 8.

Figure 7: $\frac{\#(\mathcal{B}_R^{\mathcal{C}}[0] \cap \mathcal{B}_{R'}^{\mathcal{C}}[x])}{\#\mathcal{B}_R^{\mathcal{C}}[0]}$, $\frac{\#(\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x])}{\#\mathcal{B}_R[0]}$ and $\frac{\sqrt{8}}{n\sqrt{\pi n}} \exp(-nG_1(\gamma))$ for $\gamma = 0.8$.

Figure 7 isn't clear on whether the estimation $\frac{\sqrt{8}}{n\sqrt{\pi n}} \exp(-nG_1(\gamma))$ is really a lower bound for the fractions $\frac{\#(\mathcal{B}_R^{\mathcal{C}}[0] \cap \mathcal{B}_{R'}^{\mathcal{C}}[x])}{\#\mathcal{B}_R^{\mathcal{C}}[0]}$ and $\frac{\#(\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x])}{\#\mathcal{B}_R[0]}$. To see whether the lower bound is a good lower bound, we once again plot the natural logarithm of each in Figure 8. For the Reed-Muller $[32, 16]$-code and the Hamming $[15, 11]$-code, we see that the red line at some point drops under the blue line before R is greater than the minimum weight. This is the case, because we randomly generated ten codewords x and then calculated the average of $\#(\mathcal{B}_R^{\mathcal{C}}[0] \cap \mathcal{B}_{R'}^{\mathcal{C}}[x])$ and $\#(\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x])$ over these ten codewords. Since for some of these codewords the cardinality of the set was 0, it is possible that the average is lower than $\frac{\sqrt{8}}{n\sqrt{\pi n}} \exp(-nG_1(\gamma))$.



(a) Reed-Muller code with $n = 32$, $k = 16$ and minimum weight 8.

(b) Hamming code with $n = 15$, $k = 11$ and minimum weight 3.

(c) Random Linear code with $n = 30$, $k = 10$ and minimum weight 8.

Figure 8: $\log\left(\frac{\#(\mathcal{B}_R^{\mathcal{C}}[0] \cap \mathcal{B}_{R'}^{\mathcal{C}}[x])}{\#\mathcal{B}_R^{\mathcal{C}}[0]}\right)$, $\log\left(\frac{\#(\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x])}{\#\mathcal{B}_R[0]}\right)$ and $\log\left(\frac{\sqrt{8}}{n\sqrt{\pi n}} \exp(-nG_1(\gamma))\right)$ for $\gamma = 0.8$.

4 The Nguyen-Vidick sieve algorithm for binary codes

4.1 The algorithm

Now, we finally are able to give an adaptation of the Nguyen-Vidick sieve algorithm for codes. Similarly to the algorithm for lattices, the goal of this algorithm is to reduce the maximum Hamming weight of the codewords in the set S of our code $\mathcal{C}$. Before we look into these algorithms, we define

$$B_{\mathcal{C}}(R) := \{c \in \mathcal{C} \mid |c| \leq R\}.$$

Analogous to the lattice sieve algorithm, we now define two algorithms. One that sieves a list of codewords (Algorithm 5) and one that outputs a codeword of low Hamming weight (Algorithm 6).

Algorithm 5 CodeSieve(S, γ)

Input : A subset S of codewords of $\mathcal{C}$ and a sieve factor $\frac{2}{3} < \gamma < 1$.
Output: A subset $S' \subseteq B_n(\gamma R)$.

```
R ← max_{c∈S} |c|
C ← ∅, S' ← ∅
for c ∈ S do
    if |c| ≤ γR then
        S' ← S' ∪ {c}
    else
        if ∃w ∈ C such that |c ⊕ w| ≤ γR then
            S' ← S' ∪ {c ⊕ w}
        else
            C ← C ∪ {c}
        end
    end
end
return S'
```

Note that the matrix for the input of Algorithm 6 isn't specified which matrix is used. In section 4.2.1, we check experimentally whether the LLL reduced matrix as an input matrix gives a better result versus the generator matrix.

The equivalent of Algorithm 3 for codes is Algorithm 6. Similarly as for the algorithm for lattices, this algorithm starts with sampling N non-zero codewords. First we add the codewords that are represented in the basis matrix for which the Hamming weight is smaller than or equal to $\frac{1}{2}n$. Then we take linear combinations of two codewords of the basis matrix, again for which the Hamming weight is smaller than or equal to $\frac{1}{2}n$. We continue to take an increasing number linear combinations of the basis matrix for which

Algorithm 6 Finding codes of low Hamming weight based on sieving.

Input : A basis matrix G of a code $\mathcal{C}$, a sieve factor γ such that $\frac{2}{3} < \gamma < 1$, and a number N.

Output: A non-zero codeword of $\mathcal{C}$ with a low Hamming weight.

1 $S \leftarrow \emptyset$
2 **for** $j = 1$ *to* N **do**
3 | $S \leftarrow S \cup \text{Sampling}(G)$
4 **end**
5 Remove all zero codewords from S.
6 **repeat**
7 | $S_0 \leftarrow S$
8 | $S \leftarrow \text{CodeSieve}(S, \gamma)$ using Algorithm 5
9 | Remove all zero codewords from S.
10 **until** $S = \emptyset$;
11 Compute $c_0 \in S_0$ such that $|c_0| = \min\{|c|, c \in S_0\}$.
12 **return** c_0

the Hamming weight is smaller than or equal to $\frac{1}{2}n$ until we have a list of N non-zero codewords.

Under the assumption that Heuristic 3.19 is true, we know that there are enough codewords that satisfy this condition. Hence, after steps 2-4 of Algorithm 6 we have a set S with N non-zero codewords all with a maximum length of $\frac{1}{2}n$.

Then, with the use of Algorithm 5, at every iteration of steps 6-10 of Algorithm 6, the maximum Hamming weight of the codewords in S is reduced by a factor γ. For the first iteration of steps 6-10, we use the sampled set S from steps 2-4. In Algorithm 5, we start with two empty lists C and S'. Then for every codeword c in S, we check whether the Hamming weight of this codeword c is smaller than γR. If this is the case, then the codeword c is added to the list S'. If this is not the case, we check whether there exists a codeword w in the set C such that the codeword $c \in \mathcal{B}^{\mathcal{C}}_{\gamma R}[w]$. Then we add the codeword $c + w$ to the set S', else we add the codeword c to the set C. For every new iteration of the steps 6-10 of Algorithm 6, we use the set S' as the set S. This is done until the set S' is empty or only contains zero codewords at the end of Algorithm 5.

Now, we define $C_n(\gamma R, R) := \{x \in \mathbb{F}_2^n \mid \gamma R \leq |x| \leq R\}$. Since Algorithm 5 only sieves the codewords that have a Hamming weight between γR and R, we make the following assumption:

Heuristic 4.1. At any stage in Algorithm 6, the codewords in $S \cap C_n(\gamma R, R)$ are uniformly distributed in $C_n(\gamma R, R)$.

Lemma 4.2. *Let $n \in \mathbb{N}$ and $\frac{2}{3} < \gamma < 1$. Define $\alpha_\gamma := \exp(G_1(\gamma))$ with*

$$G_1(\gamma) = \frac{7\gamma^2 - (\gamma^2 + 3\gamma - 1)\sqrt{4\gamma + 1} - \gamma + 1}{2(\sqrt{4\gamma + 1} + 1)}$$

and $N_C = \frac{n^2\sqrt{\pi n}}{\sqrt{8}}\alpha_\gamma^n$. Let N be an integer, and S a subset of $C_n(\gamma R, R)$ of cardinality N whose points are picked independently at random with uniform distribution.

If $\frac{1}{n\sqrt{n}}\frac{\sqrt{8}}{\sqrt{\pi}}\alpha_\gamma^n < N < 2^n$, then for any subset centers $C \subseteq S$ of size at least N_C whose points are picked independently at random with uniform distribution, with probability $1 - e^{-n}$, for all $y \in S$, there exists a $c \in C$ such that $|y \oplus c| \leq \gamma$.

Proof. If α_γ is as described, then by Corollary 3.22 we have

$$\Omega(\gamma) = \frac{\#(\mathcal{B}_R[0] \cap \mathcal{B}_{R'}[x])}{\#\mathcal{B}_R[0]} \gtrsim \frac{\sqrt{8}}{n\sqrt{n\pi}}\alpha_\gamma^{-n}.$$

Let x be a codeword in the set of centers C. Then the expected fraction of $\mathcal{B}_R[0]$ that is not covered by $\mathcal{B}_{R'}[x]$ is at most $1 - \Omega(\gamma)$. Hence for N_C uniformly distributed centers, the expected fraction of $\mathcal{B}_R[0]$ that is not covered by any of these centers is at most $(1 - \Omega(\gamma))^{N_C}$. We have

$$\begin{aligned} N_C \log(1 - \Omega(\gamma)) &\leq -N_C\Omega(\gamma) \\ &\leq -\frac{n^2\sqrt{\pi n}}{\sqrt{8}}\alpha_\gamma^n \frac{\sqrt{8}}{n\sqrt{n\pi}}\alpha_\gamma^{-n} \\ &= -n \end{aligned}$$

So the expected fraction that is not covered by ant of the N_C centers is at most e^{-n}. Hence, the expected fraction that is covered by N_C centers is $1 - e^{-n}$. □

With the result of Lemma 4.2, we can estimate the complexity of the sieve algorithm. Under the assumption that Heuristic 4.1 is true, we expect at steps 7-11 that the size of S will decrease by roughly α_γ^n. Then at every iteration of the steps 7-11, the maximum Hamming weight of the codewords in S is reduced by a factor γ. Since the maximum Hamming weight of the initial sampled codewords is $\lfloor \frac{n}{2} \rfloor$, and if the number of sampled codewords is $poly(n) \cdot N_C$, then after a linear number of iterations we expect to be left with a large number of codewords with a low Hamming weight. At the limit $\gamma \to 1$, we get $\alpha_\gamma = \exp\left(\frac{7-3\sqrt{5}}{2(\sqrt{5}+1)}\right)$. Since the running time of the sieve is quadratic, the total running time of the algorithm is expected to be of order $\left(\exp\left(\frac{7-3\sqrt{5}}{\sqrt{5}+1}\right) + \epsilon\right)^n \approx (1.0944 + \epsilon)^n$. The quadratic running time of the sieve does not influence the space complexity, so the space complexity is expected to be of order $(\alpha_\gamma + \epsilon)^n \approx (1.0461 + \epsilon)^n$.

Since the Nguyen-Vidick sieve algorithm for lattices has $(\frac{4}{3} + \epsilon)^n \approx (1.3333 + \epsilon)^n$ as running time, hence we see that the sieve algorithm for codes has a lower heuristic complexity.

4.2 Numerical experiments

The minimum Hamming weight of a Hamming code is 3, independent of the length n and the dimension d of the code. The binary Reed Muller code of length $n = 2^m$ has 2^{m-2} as its minimum Hamming weight [Lin12]. Since the minimum Hamming weight of these types of codes is already known, using this sieve algorithm isn't necessary to find this minimum Hamming weight. Furthermore, the LLL reduced basis consisted of a codeword of minimum Hamming weight in the examples we tried. Therefore, we only looked at random linear codes for the numerical experiments.

4.2.1 LLL reduced basis versus generator matrix

As mentioned before, we wanted to find out whether using the LLL reduced basis matrix increases the chance to find the codeword of minimum Hamming weight versus using the generator matrix as the input matrix. To analyze this, we have performed a hundred experiments for an increasing number of sampled codewords N with for different values for γ. So for the sieve factor γ equal to 0.7, 0.8, 0.9 and 0.97 (the value Nguyen-Vidick [NV08] used for their experiments) with an increasing integer N, we checked for a hundred random linear binary codes whether the output of Algorithm 6 was a codeword of minimum Hamming weight.

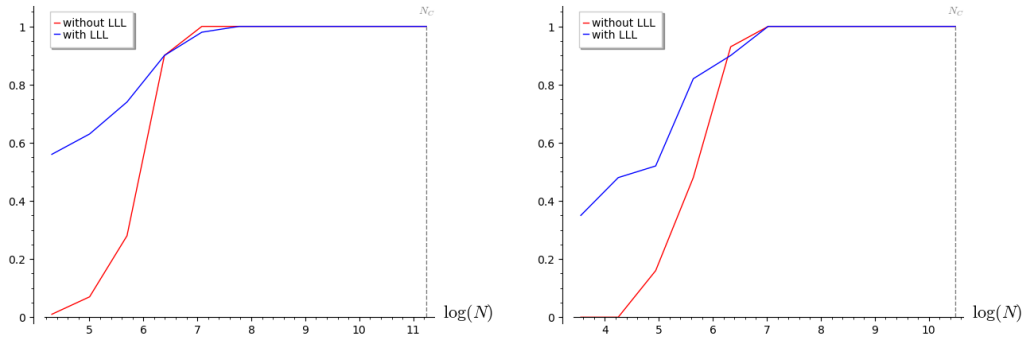
In Figure 9, the fraction where a codeword of minimum Hamming weight was found for random linear codes with $n = 30$ and $k = 20$ is represented. The indicated value N_C is calculated the same as in Lemma 4.2.

As we can see in Figure 9, the chance that you find a codeword of minimum Hamming weight at $N = N_C$ is larger for a smaller γ. We also see that for γ closer to one, using the LLL reduced basis matrix as input matrix for the algorithm improves the result more than for smaller γ.

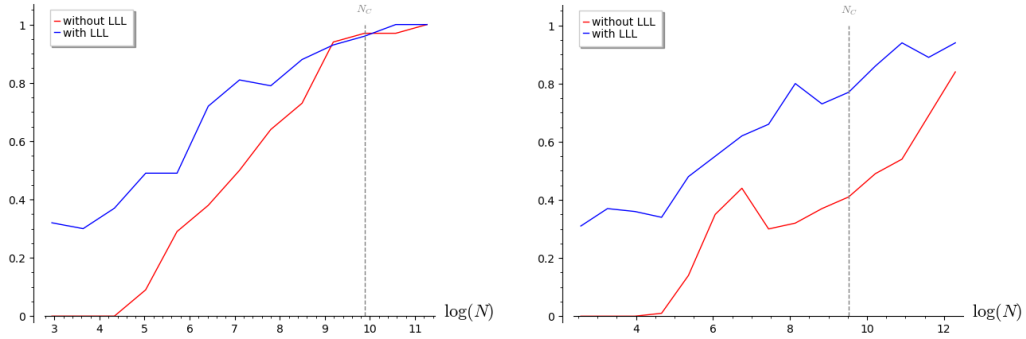
For $\gamma = 0.97$, we see that when the number of sampled codewords N is equal to N_C, that the chance that we found a codeword of minimum Hamming weight is approximately 0.4 without the use of LLL and approximately 0.8 with the use of LLL. When we increase N, then the chance that we find a codeword of minimum Hamming weight gets closer to one with or without using LLL as the input matrix.

If $\gamma = 0.9$, the chance that the algorithm outputs a codeword of minimum Hamming weight is almost equal at $N = N_C$ when the LLL reduced basis or the generator matrix is used as input. When N becomes a lot smaller than N_C, the chance that a codeword of minimum weight is returned after the algorithm is bigger when the LLL reduced basis is used as input.

When γ is equal to 0.7 or 0.8, we see that the chance that the algorithm finds a codeword of minimum weight is 1 when the number of sampled codewords N is equal to N_C and also if N is several times smaller than N_C. If the generator matrix is used as the input for the algorithm, we see that the chance that a codeword of minimum weight is found drops faster to zero than if the LLL reduced basis is used as the input.



(a) $\gamma = 0.7$ (b) $\gamma = 0.8$



(c) $\gamma = 0.9$ (d) $\gamma = 0.97$

Figure 9: The chance that a codeword of minimum Hamming weight was found for a random $[30, 20]$-code for four different values of γ with N sampled codewords.

Overall, we can conclude from Figure 9 that LLL does improves the chance that a codeword of minimum Hamming weight will be the output of Algorithm 6.

4.2.2 Number of iterations

The maximum number of iterations the algorithm uses is bounded and can easily be calculated. At the beginning of the algorithm, the maximum Hamming weight of the sampled codewords is $\lfloor \frac{1}{2}n \rfloor$. The minimum Hamming weight of a code is at least 1. Since the maximum Hamming weight of the list of codewords in every iteration is reduced by γ, we know in which range the Hamming weights of the list of codewords is at each iteration. Hence, the number of iterations can be calculated with Algorithm 7.

In most cases when the algorithm is used, the maximum number of iterations will not be needed. This could be the case if the maximum Hamming weight of the sampled codewords is smaller than $\lfloor \frac{1}{2}n \rfloor$ or if the minimum Hamming weight of the code is larger than 1. Therefore, it is interesting to know what the average number of iterations is.

Algorithm 7 Maximum number of iterations needed for a binary linear code $\mathcal{C} \subset \mathbb{F}_2^n$.

Input : The length of the code n and a sieve factor $\frac{2}{3} < \gamma < 1$.
Output: Maximum number of iterations.

```
w = ⌊½n⌋
i = 0
while w > 1 do
    w = ⌊w · γ⌋
    i = i + 1
end
return i
```

To calculate the average number of iterations, we performed twenty experiments for codes of length $20 < n < 40$ for four different values of γ. We used the the algorithm with the LLL reduced basis matrix as the input matrix and the number of sampled codewords is $N = N_C$. Then we calculated the average number of iterations needed in the algorithm.

In Figure 10, the average number of iterations the algorithm uses is displayed in the solid lines. The dotted lines show the maximum number of iterations calculated with Algorithm 7.

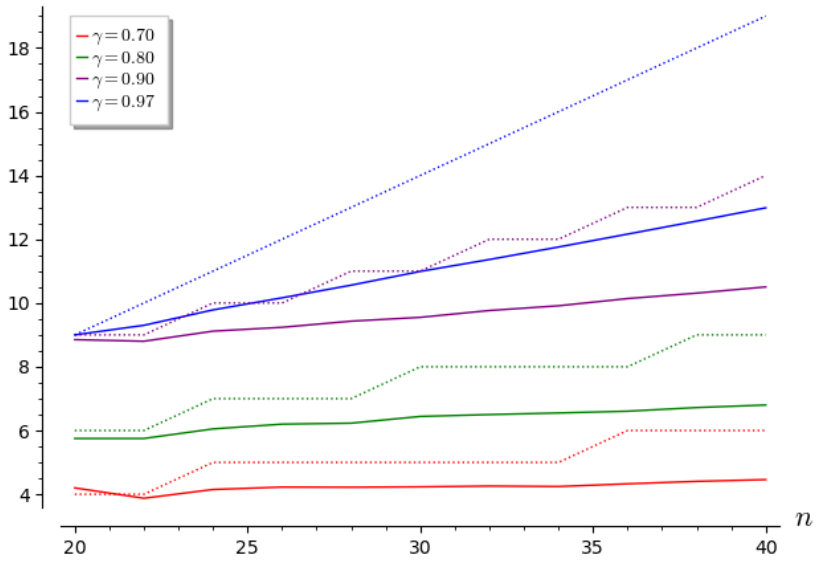


Figure 10: Average number of iterations needed for codes with length $20 < n < 40$ for four different values of γ compared with the maximum number of iterations (dotted lines).

5 Discussion and further research

5.1 Discussion

For the calculation of the lower bound for the fraction $\frac{\#(\mathcal{B}_R^{\mathcal{C}}[0] \cap \mathcal{B}_{R'}^{\mathcal{C}}[x])}{\#\mathcal{B}_R^{\mathcal{C}}[0]}$ in 3.3, we have used several heuristics. These heuristics are substantiated with numerical evidence, but are not exact. Therefore some differences could still occur in the given estimations. The lower bound of this fraction is used to analyze the running time of Algorithm 6. For this running time, we also assumed that Heuristic 4.1 is true.

Second, in section 3.3 we have only given an upper bound for the fraction $\frac{\#(\mathcal{B}_R^{\mathcal{C}}[0] \cap \mathcal{B}_{R'}^{\mathcal{C}}[x])}{\#\mathcal{B}_R^{\mathcal{C}}[0]}$ and no lower bound for this fraction. Therefore, we were not able to give a tight estimate for the running time of the algorithm.

5.2 Further research

As we could see in section 4.2.1, the chance to find a codeword of minimum Hamming weight is larger for a smaller γ than for γ closer to one. The numerical calculations in the section suggests that for γ equal to 0.7 or 0.8, the number of sampled codewords at the beginning of the algorithm can be lower than for γ equal to 0.9 or 0.97. Hence, it could be interesting to look what the optimal value for γ is for the running time and space requirements.

q-ary codes

In this thesis, we only looked at binary linear codes $\mathcal{C}$. The estimations, heuristics and the LLL algorithm in section 3 are all only applicable for binary codes. This is the case, because the i-th position of a codeword can only have two values, namely it can be 0 or 1. Therefore we were able to use binomial coefficients to find the cardinality of $\mathcal{B}_R^{\mathcal{C}}[x]$.

It is possible that algorithms 5 and 6, can be adapted directly for q-ary codes with $q > 2$. However, the analysis of the running time has to be adapted.

Multiple level sieving

The traditional Nguyen-Vidick lattice sieve, which was the basis for this thesis, is sometimes also considered as the 1-level sieve [Laa15a]. This 1-level sieve has been improved by Wang-Liu-Tian-Bi into a 2-level sieve [Wan+11] and by Zhang-Pan-Hu into a 3-level sieve [ZPH13]. These algorithms work by adding respectively one or two sieve factors. These algorithms have a better time complexity, but use more space.

It could be interesting to look if these algorithms will also improve the 1-level sieve algorithm for binary codes.

References

[AKS01] Miklós Ajtai, Ravi Kumar, and Dandapani Sivakumar. "A sieve algorithm for the shortest lattice vector problem". In: *Proceedings of the thirty-third annual ACM symposium on Theory of computing*. 2001, pp. 601–610.

[Alb+15] Martin Albrecht et al. "On the complexity of the BKW algorithm on LWE". In: *Designs, Codes and Cryptography* 74.2 (2015), p. 26. DOI: 10.1007/s10623-013-9864-x. URL: https://hal.inria.fr/hal-00921517.

[APY09] Noga Alon, Rina Panigrahy, and Sergey Yekhanin. "Deterministic approximation algorithms for the nearest codeword problem". In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. Springer, 2009, pp. 339–351.

[BKW03] Avrim Blum, Adam Kalai, and Hal Wasserman. "Noise-Tolerant Learning, the Parity Problem, and the Statistical Query Model". In: *J. ACM* 50.4 (2003), pp. 506–519. ISSN: 0004-5411. DOI: 10.1145/792538.792543. URL: https://doi.org/10.1145/792538.792543.

[Car06] Claude Carlet. "The complexity of Boolean functions from cryptographic viewpoint". In: *Complexity of Boolean Functions*. Ed. by Matthias Krause et al. Dagstuhl Seminar Proceedings 06111. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006. URL: http://drops.dagstuhl.de/opus/volltexte/2006/604.

[DDW20] Thomas Debris-Alazard, Léo Ducas, and Wessel P.J. van Woerden. *An Algorithmic Reduction Theory for Binary Codes: LLL and more*. Cryptology ePrint Archive, Report 2020/869. https://eprint.iacr.org/2020/869. 2020.

[Dib13] Stephanie Dib. "Distribution de la non-linéarité des fonctions booléennes". PhD thesis. Aix-Marseille, 2013, pp. 9–23. URL: http://www.theses.fr/2013AIXM4090.

[Hof08] Jeffrey Hoffstein. "Lattices and Cryptography". In: *An Introduction to Mathematical Cryptography*. New York, NY: Springer New York, 2008, pp. 1–87. ISBN: 978-0-387-77994-2. DOI: 10.1007/978-0-387-77993-5_6. URL: https://doi.org/10.1007/978-0-387-77993-5_6.

[Laa15a] Thijs Laarhoven. "Search problems in cryptography". PhD thesis. PhD thesis, Eindhoven University of Technology, 2015, pp. 91–219.

[Laa15b] Thijs Laarhoven. "Sieving for shortest vectors in lattices using angular locality-sensitive hashing". In: *Annual Cryptology Conference*. Springer. 2015, pp. 3–22.

[LB88] P. J. Lee and E. F. Brickell. "An Observation on the Security of McEliece's Public-Key Cryptosystem". In: *Advances in Cryptology — EUROCRYPT '88*. Ed. by D. Barstow et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 275–280. ISBN: 978-3-540-45961-3.

[Lin12] Jacobus Hendricus van Lint. *Introduction to coding theory*. Vol. 86. Springer Science & Business Media, 2012.

[LLL82] Arjen K Lenstra, Hendrik Willem Lenstra, and László Lovász. "Factoring polynomials with rational coefficients". In: *Mathematische Annalen* 261.4 (1982), pp. 515–534.

[MO15] Alexander May and Ilya Ozerov. "On Computing Nearest Neighbors with Applications to Decoding of Binary Linear Codes". In: *Advances in Cryptology – EUROCRYPT 2015*. Ed. by Elisabeth Oswald and Marc Fischlin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 203–228. ISBN: 978-3-662-46800-5.

[NV08] Phong Q Nguyen and Thomas Vidick. "Sieve algorithms for the shortest vector problem are practical". In: *Journal of Mathematical Cryptology* 2.2 (2008), pp. 181–207.

[NV10] Phong Q Nguyen and Brigitte Vallée. *The LLL algorithm*. Springer, 2010.

[Ste88] Jacques Stern. "A method for finding codewords of small weight". In: *International Colloquium on Coding Theory and Applications*. Springer. 1988, pp. 106–113.

[Wan+11] Xiaoyun Wang et al. "Improved Nguyen-Vidick Heuristic Sieve Algorithm for Shortest Vector Problem". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2011, pp. 1–9. ISBN: 9781450305648. DOI: 10.1145/1966913.1966915. URL: https://doi.org/10.1145/1966913.1966915.

[ZPH13] Feng Zhang, Yanbin Pan, and Gengran Hu. *A Three-Level Sieve Algorithm for the Shortest Vector Problem*. Cryptology ePrint Archive, Report 2013/536. 2013. URL: https://ia.cr/2013/536.

SageMath code

The next code is used as the Nguyen-Vidick sieve algorithm for binary codes. The code is written in SageMath.

```
V = GF(2)

"""
Set up the desired parameters
"""
gamma = 0.7
L = 30
k = floor((2/3)*L)


M = matrix.random(V, k, L)
B = codes.LinearCode(M)
G = copy(B.generator_matrix())


"""
Bitwise AND function
"""
def AND(v,w):
    V = v.parent()
    And = [v[i] and w[i] for i in range(len(v))]
    return V(And)

"""
Bitwise OR function
"""
def OR(v,w):
    V = v.parent()
    Or = [v[i] or w[i] for i in range(len(v))]
    return V(Or)

"""
Bitwise NOT function
"""
def NOT(v):
    V = v.parent()
    Not = [1-v[i] for i in range(len(v))]
    return V(Not)

"""
Function that computes a list of the negotation of the union of the
    previous calculated epipodal vector for the epipodal matrix
"""
def not_or(G):
    not_or_list = []
    r = zero_vector(G.ncols())
    not_or_list.append(NOT(r))
```

```
    for v in G.rows():
        r = OR(r,v)
        not_or_list.append(NOT(r))
    return(not_or_list)

"""
Function that computes the b_i^+ vectors of the epipodal matrix
"""
def B_plus(G, not_or_list):
    B_plus_list = []

    for i in range(G.nrows()):
        if i == 0:
            B_plus_list.append(G.row(i))
        else:
            b_i_plus = AND(G.row(i), not_or_list[i])
            B_plus_list.append(b_i_plus)
    return(B_plus_list)

"""
Function that calculates pi_i(b_{i+1})
"""
def pi_function(i,v,M):
    not_or_list = not_or(M)
    w = not_or_list[i-1]
    return(AND(v,w))

"""
Function that finds whether there is a local minimum, this is used in the
    while statement of the LLL algorithm for binary codes
"""
def find_local_minimum(M, B_plus_list):
    for i in range(1,M.nrows()):
        if min(pi_function(i,M.row(i),M).hamming_weight(),
            (B_plus_list[i-1] +
            pi_function(i,M.row(i),M)).hamming_weight()) <
            B_plus_list[i-1].hamming_weight():
            return i-1
    return None

"""
The tie breaking function used of the LLL algorithm for binary codes
"""
def TB(p,y):
    if p.hamming_weight() % 2 == 1:
        return 0
    else:
        j = min(p.support())
        if y[j] == 0:
            return 0
        else: return 1/2

"""
```

```
The LLL algorithm for binary codes
"""
def LLL_codes(G):
    while True:
        B_plus_list = B_plus(G, not_or(G))
        i = find_local_minimum(G, B_plus_list)
        if i is None:
            break
        if AND(pi_function(i+1,G.row(i+1),G),
            B_plus_list[i]).hamming_weight() + TB(B_plus_list[i],
            pi_function(i+1,G.row(i+1),G)) >
            B_plus_list[i].hamming_weight() / 2:
            G.add_multiple_of_row(i+1,i,1)
        G.swap_rows(i, i+1)
    return G

"""
Function that samples linear combinations of the codewords of the LLL
    matrix with maximal Hamming weight n/2
"""
def sampling(B,N,R):
    S = []
    k = B.nrows()

    for i in [0..k-1]:
        s = B.row(i)
        if s.hamming_weight() <= R:
            S.append(s)
    for i in [0..k-2]:
        for j in [i+1..k-1]:
            if len(S)<N:
                s = B.row(i) + B.row(j)
                if s.hamming_weight() <= R:
                    S.append(s)
            else: break
    for i1 in [0..k-3]:
        for i2 in [i1+1..k-2]:
            for i3 in [i2+1..k-1]:
                if len(S)<N:
                    s = B.row(i1) + B.row(i2) + B.row(i3)
                    if s.hamming_weight() <= R:
                        S.append(s)
                else: break
    for i1 in [0..k-4]:
        for i2 in [i1+1..k-3]:
            for i3 in [i2+1..k-2]:
                for i4 in [i3+1..k-1]:
                    if len(S)<N:
                        s = B.row(i1) + B.row(i2) + B.row(i3) + B.row(i4)
                        if s.hamming_weight() <= R:
                            S.append(s)
                    else: break
    for i1 in [0..k-5]:
```

```
    for i2 in [i1+1..k-4]:
        for i3 in [i2+1..k-3]:
            for i4 in [i3+1..k-2]:
                for i5 in [i4+1..k-1]:
                    if len(S)<N:
                        s = B.row(i1) + B.row(i2) + B.row(i3) +
                            B.row(i4) + B.row(i5)
                        if s.hamming_weight() <= R:
                            S.append(s)
                    else: break
for i1 in [0..k-6]:
    for i2 in [i1+1..k-5]:
        for i3 in [i2+1..k-4]:
            for i4 in [i3+1..k-3]:
                for i5 in [i4+1..k-2]:
                    for i6 in [i5+1..k-1]:
                        if len(S)<N:
                            s = B.row(i1) + B.row(i2) + B.row(i3) +
                                B.row(i4) + B.row(i5) + B.row(i6)
                            if s.hamming_weight() <= R:
                                S.append(s)
                        else: break
for i1 in [0..k-7]:
    for i2 in [i1+1..k-6]:
        for i3 in [i2+1..k-5]:
            for i4 in [i3+1..k-4]:
                for i5 in [i4+1..k-3]:
                    for i6 in [i5+1..k-2]:
                        for i7 in [i6+1..k-1]:
                            if len(S)<N:
                                s = B.row(i1) + B.row(i2) + B.row(i3)
                                    + B.row(i4) + B.row(i5) +
                                    B.row(i6) + B.row(i7)
                                if s.hamming_weight() <= R:
                                    S.append(s)
                            else: break
for i1 in [0..k-8]:
    for i2 in [i1+1..k-7]:
        for i3 in [i2+1..k-6]:
            for i4 in [i3+1..k-5]:
                for i5 in [i4+1..k-4]:
                    for i6 in [i5+1..k-3]:
                        for i7 in [i6+1..k-2]:
                            for i8 in [i7+1..k-1]:
                                if len(S)<N:
                                    s = B.row(i1) + B.row(i2) +
                                        B.row(i3) + B.row(i4) +
                                        B.row(i5) + B.row(i6) +
                                        B.row(i7)+B.row(i8)
                                    if s.hamming_weight() <= R:
                                        S.append(s)
                                else: break
for i1 in [0..k-9]:
```

```
    for i2 in [i1+1..k-8]:
        for i3 in [i2+1..k-7]:
            for i4 in [i3+1..k-6]:
                for i5 in [i4+1..k-5]:
                    for i6 in [i5+1..k-4]:
                        for i7 in [i6+1..k-3]:
                            for i8 in [i7+1..k-2]:
                                for i9 in [i8+1..k-1]:
                                    if len(S)<N:
                                        s = B.row(i1) + B.row(i2) +
                                            B.row(i3) + B.row(i4) +
                                            B.row(i5) + B.row(i6) +
                                            B.row(i7) + B.row(i8) +
                                            B.row(i9)
                                        if s.hamming_weight() <= R:
                                            S.append(s)
                                    else: break
for i1 in [0..k-10]:
    for i2 in [i1+1..k-9]:
        for i3 in [i2+1..k-8]:
            for i4 in [i3+1..k-7]:
                for i5 in [i4+1..k-6]:
                    for i6 in [i5+1..k-5]:
                        for i7 in [i6+1..k-4]:
                            for i8 in [i7+1..k-3]:
                                for i9 in [i8+1..k-2]:
                                    for i10 in [i9+1..k-1]:
                                        if len(S)<N:
                                            s = B.row(i1) + B.row(i2)
                                                + B.row(i3) +
                                                B.row(i4) + B.row(i5)
                                                + B.row(i6) +
                                                B.row(i7) + B.row(i8)
                                                + B.row(i9) +
                                                B.row(i10)
                                            if s.hamming_weight() <= R:
                                                S.append(s)
                                        else: break
for i1 in [0..k-11]:
    for i2 in [i1+1..k-10]:
        for i3 in [i2+1..k-9]:
            for i4 in [i3+1..k-8]:
                for i5 in [i4+1..k-7]:
                    for i6 in [i5+1..k-6]:
                        for i7 in [i6+1..k-5]:
                            for i8 in [i7+1..k-4]:
                                for i9 in [i8+1..k-3]:
                                    for i10 in [i9+1..k-2]:
                                        for i11 in [i10+1..k-1]:
                                            if len(S)<N:
                                                s = B.row(i1) +
                                                    B.row(i2) +
                                                    B.row(i3) +
```

```
                                                    B.row(i4) +
                                                    B.row(i5) +
                                                    B.row(i6) +
                                                    B.row(i7) +
                                                    B.row(i8) +
                                                    B.row(i9) +
                                                    B.row(i10) +
                                                    B.row(i11)
                                                if s.hamming_weight()
                                                    <= R:
                                                    S.append(s)
                                            else: break
    return S

"""

"""
def find_centre(v, C, X):
    for c in C:
        if (v-c).hamming_weight() <= X:
            return c
    return None

"""
The CodeSieve(S,gamma) algorithm, the same as Algorithm 5
"""
def code_sieve(M, gamma):
    R = max(v.hamming_weight() for v in M.rows())
    C = []
    S_accent = []

    for i in range(M.nrows()):
        if M.row(i).hamming_weight() <= gamma*R:
            S_accent.append(M.row(i))
        else:
            c = find_centre(M.row(i), C, gamma*R)
            if c is None:
                C.append(M.row(i))
            else:
                S_accent.append(M.row(i)-c)

    C_hammingweight = [c.hamming_weight() for c in C]

    for s in S_accent:
        s.set_immutable()
    return set(sorted(S_accent))

"""
Calculates N_C as defined in Lemma 4.2
"""
G1 = (7*gamma^2-(gamma^2+3*gamma-1)*sqrt(4*gamma+1)-gamma+1) /
    (2*(sqrt(4*gamma+1)+1))
N = ceil(((L^2*sqrt(pi*L))/sqrt(8))*exp(L*G1))
```

```
"""
The Nguyen-Vidick sieve algorithm for binary codes with the LLL reduced
    matrix as input
"""
def sieving_with_LLL(G,L,k,N):
    G = LLL_codes(G)
    B = matrix(G)
    S = sampling(B,N,L/2)

    while S!=[]:
        M = matrix(S)
        S = code_sieve(M,gamma)
        S = [v for v in S if not v.is_zero()]

    c0 = [c for c in M.rows() if c.hamming_weight() ==
        min(v.hamming_weight() for v in M.rows())]
    return c0[0]
```