

Diploma thesis

A Lattice Attack on the McEliece Public Key
Cryptosystem

December 2006



by
Zaharina Velikova

Supervisors: Prof. Dr. Johannes A. Buchmann,
Dr. Ulrich Vollmer

Darmstadt University of Technology
Department of Mathematics

Table of Contents

	Page
Table of Contents	i
List of Algorithms	ii
Chapter	
1 Introduction	1
2 Reducing the Decoding Problem to a Lattice Problem	3
2.1 A Brief Introduction to Error-Correcting Codes	3
2.2 Binary Goppa Codes	4
2.3 The McEliece Cryptosystem	5
2.4 Reducing the Problem of Finding the Error Vector to the Problem of Finding the Shortest Lattice Vector	6
3 Block Basis Reduction for the Euclidean Norm	13
3.1 Introduction to Lattices	13
3.2 The L^3 Basis Reduction Algorithm	14
3.3 The Block Basis Reduction Algorithm	21
4 Block Basis Reduction for Arbitrary Norm	31
4.1 The L^3 Algorithm for Arbitrary Norm	31
4.2 The Block Basis Reduction Algorithm for Arbitrary Norm	36
4.3 Computing the Values of the Distance Functions for the l_p Norm	44
4.3.1 The case $p = 2$	45
4.3.2 The case $p > 2$	48
5 Tests and Results	51
6 Implementation Details	59
6.1 Lattice Basis Reduction Algorithms	59
6.2 Test Program for the Basis Reduction Algorithms	60
6.3 Generating the Test Input Files	62
Bibliography	66
Appendix	68
A Source Code	69
A.1 Basis Reduction Algorithms	69
A.1.1 L^3 Basis Reduction Algorithm for the l_p Norm	69

A.1.2	Block Basis Reduction Algorithm for the l_p Norm	72
A.1.3	Calculation of the Distance Functions for the l_p Norm	79
A.1.4	Utility Functions	85
A.2	Test Program for the Basis Reduction Algorithms	90
A.2.1	The Main Function	90
A.2.2	The Test Function	91
A.3	Program for Generation of Test Input Files	94
A.3.1	The Main Function	94
A.3.2	Test File Generation Functions	96
A.3.3	Generation of Random Error and Message Vectors	102
A.3.4	Generation of Random Binary Invertible Matrices and Random Permutation Matrices	103
A.3.5	Gaussian Elimination and Column Permutation	109
A.3.6	Utility Functions	115

List of Algorithms

3.1	Gauß Basis Reduction Algorithm for the Euclidean Norm	16
3.2	L^3 Basis Reduction Algorithm for the Euclidean Norm	19
3.3	Block Basis Reduction Algorithm for the Euclidean Norm	25
3.4	BASIS - Basis Transformation	26
3.5	ENUM - Enumeration Subroutine for the Block Basis Reduction Algorithm for the Euclidean Norm	28
4.1	LS Basis Reduction Algorithm - The L^3 Basis Reduction Algorithm for Arbitrary Norm	35
4.2	Block Basis Reduction Algorithm for Arbitrary Norm	41
4.3	ADFS - Enumeration Subroutine for the Block Basis Reduction Algorithm for Arbitrary Norm	43
4.4	Algorithm for the Least Squares Problem	48
4.5	Algorithm for the Linear l_p Approximation Problem	49
4.6	Algorithm for the Calculation of the Distance Functions	50

Chapter 1

Introduction

In this thesis we propose an attack on the McEliece public key cryptosystem [McE78]. This cryptosystem was proposed by R. McEliece in 1978 and it is the first public key cryptosystem based on coding theory. In its original version it is based on Goppa codes. For appropriate system parameters it remains unbroken. However, the private and public keys are large matrices, which is one of the main disadvantages of the cryptosystem. In order to overcome this disadvantage several ideas were presented to modify the McEliece's original proposal. Most of them involve replacing the Goppa codes with other codes. But still, most of them turned out to be insecure or inefficient compared to the original McEliece's proposal.

Since the McEliece cryptosystem was created, various ideas for attacks on it were presented. Some attacks aim to get the private key from the public key. For a detailed description of such attacks see [EOS06]. Other attacks aim to recover the plaintext message from a given ciphertext. Such attacks are the Information-Set-Decoding attack, Known-Partial-Plaintext attack, Related-Message attack, Reaction attack, Finding-Low-Weight-Codeword attack. A detailed overview of these attacks can be found in [BBD⁺05].

The purpose of the attack, proposed in this thesis, is to recover the error vector from a given ciphertext. This attack is based on the observation that the error vector is the codeword with minimal weight in the code generated by the matrix, which is composed by the public key matrix and the ciphertext. The problem of finding the low-weight codeword reminds of the problem of determining the shortest vector in a lattice. Using a construction, presented by Conway and Sloane [CS88], we were able to reduce the problem of recovering the error vector to one of the main lattice problems, namely the shortest lattice vector problem (SVP). The shortest vector problem is, given a lattice, to find the shortest non-zero vector in this lattice. The best provable algorithm for solving SVP exactly is the AKS sieving algorithm [AKS01], which is a probabilistic algorithm running in time $2^{O(n)}$. However it makes sense to use more efficient but not so exact algorithms and hope that the output is the shortest lattice vector. Such algorithms are the lattice basis reduction algorithms.

Given a basis of a lattice \mathcal{L} the aim of the lattice basis reduction algorithms is to transform the given basis into a basis of the same lattice, which contains vectors with smallest possible norm. Till recently the lattice basis reduction concepts and algorithms were based on the Euclidean norm. The first concepts of Lagrange and Gauß were developed for lattices with low dimensions. Then Hermite and Korkine and Zolotarev (HKZ) presented a reduction concept, which gives a very good approximation of the successive

minima, but is rather impractical for higher dimensions. A decisive breakthrough in the area of lattice basis reduction was introduced by H. Lenstra, A. Lenstra and Lovász [LLL82]. They proposed the L^3 lattice basis reduction algorithm, which extends in a natural way the concept of Gauß to lattices of arbitrary dimensions. Then Schnorr linked Lovász and Hermite's definitions and obtained a flexible hierarchy of reductions w.r.t. the Euclidean norm, named block reduction. Later Lovász and Scarf presented a generalization of the L^3 reduction concept to arbitrary norms. After that Kaib and Ritter generalized the block reduction to arbitrary norms. In theory, the lattice basis reduction algorithms guarantee only an approximation up to a factor, exponential in the lattice dimension.

In the attack, presented in this thesis, we use L^3 and block basis reduction algorithms w.r.t a non-standard norm. We have chosen the l_p norm. Since the basis reduction algorithms provide only an approximation of the length of the shortest lattice vector, we cannot expect this attack to work in all cases. The purpose of this thesis is to test, with specific reduction parameters and different dimensions of the input matrix (see Chapter 5), whether the approximation, which the reduction algorithms provide in practice, will be good enough to attack the McEliece cryptosystem in its original version. In order to do that we provide an implementation of the L^3 and the block basis reduction algorithms w.r.t. the l_p norm for $p > 2$, based on the Gnu Scientific Library [GSL] (see Chapter 6). The time used by the reduction algorithms increases considerably and the success probability decreases with increasing dimension of the input basis vectors. Our tests have shown that the attack becomes impractical for Goppa codes of length 127, since the success rate remained 0%.

The structure of this thesis is as follows. Chapter 2 is devoted to the reduction of the decoding problem to the shortest lattice vector problem. In this chapter we also present in details the construction of a Goppa code and its generation matrix and the McEliece cryptosystem in its original version. In Chapter 3 we present the L^3 and the block basis algorithms for the Euclidean norm (l_2). There we discuss the main properties of the L^3 -reduced and block reduced bases and we provide detailed proofs for the approximation of the length shortest lattice vector, which they provide. In Chapter 4 we describe the generalized version of the L^3 algorithm for arbitrary norms, provided by Lovász and Scarf [LS92], and the block basis reduction algorithm for arbitrary norms, due to Ritter [Rit97]. We also prove the quality of the approximation of the successive minima, which the algorithms give. In order to be able to generalize the reduction concepts to arbitrary norms we use specific distance functions w.r.t an arbitrary norm, introduced by Lovász and Scarf. In Section 4.3 we discuss an algorithm for calculating these distance functions w.r.t. the l_p norm. In Chapter 5 we present the results of the tests for different Goppa codes and reduction parameters. In Chapter 6 we present details for the implementation of the reduction algorithms and in the Appendix we give the source code of our implementation.

Chapter 2

Reducing the Decoding Problem to a Lattice Problem

In this chapter we present the lattice attack against the McEliece cryptosystem. We first give some basic definitions from the theory of linear and error-correcting codes. Then we define in details the irreducible binary Goppa code and describe the construction of its generator matrix. In the third section we present the original version of the McEliece cryptosystem, based on Goppa codes. In the last section we give a detailed description of the reduction of the problem of finding the error vector of a given ciphertext to the problem of finding the shortest non-zero vector in a lattice.

2.1 A Brief Introduction to Error-Correcting Codes

We begin with the definition of an (n, k) -code.

Definition 2.1. An (n, k) code \mathcal{C} over a finite field \mathbb{F} is a k -dimensional vector subspace of the vector space \mathbb{F}^n . We call \mathcal{C} an (n, k, d) code if d is the minimum distance in \mathcal{C} , i.e. $d = \min_{\mathbf{x}, \mathbf{y} \in \mathcal{C}} \text{dist}(\mathbf{x}, \mathbf{y})$, where "dist" is a distance function. With $\text{wt}(\mathbf{x}) := \text{dist}(\mathbf{0}, \mathbf{x})$ we denote the distance of $\mathbf{x} \in \mathbb{F}^n$ to the zero vector. We call $\text{wt}(\mathbf{x})$ weight of \mathbf{x} .

Throughout this chapter we use the *Hamming distance* as distance function. The Hamming distance between $\mathbf{x}, \mathbf{y} \in \mathcal{C}$ is defined as the number of indices i , such that $x_i \neq y_i$. Since \mathcal{C} is a k -dimensional vector subspace of \mathbb{F}^n , we can represent the code \mathcal{C} by a generator matrix $G \in \mathbb{F}^{k \times n}$. The rows of G form a basis of the linear code \mathcal{C} . The matrix $H \in \mathbb{F}^{(n-k) \times n}$ is called *check matrix* for the code \mathcal{C} if $\mathcal{C} = \{\mathbf{c} \in \mathbb{F}^n \mid H\mathbf{c}^T = \mathbf{0}\}$. The code generated by H is called *dual code* of \mathcal{C} and is denoted by \mathcal{C}^\perp .

Let \mathcal{C} be an (n, k, d) code. Let $t := \lfloor \frac{d-1}{2} \rfloor$ and $\mathbf{y} \in \mathbb{F}^n$ be a ciphertext with at most t corrupted entries. Then the original message can be uniquely recovered by choosing the codeword in \mathcal{C} that is closest to the received message \mathbf{y} and we say that \mathcal{C} is *t-error correcting code*. The minimum distance of a linear code \mathcal{C} determines how many errors \mathcal{C} can correct. Codes with bigger minimum distance can correct more errors.

There are codes, which are equivalent w.r.t a permutation.

Definition 2.2. Two (n, k) codes \mathcal{C} and \mathcal{C}' over a field \mathbb{F} are called permutation equivalent if there exists permutation π over n elements, such that

$$\mathcal{C}' = \pi(\mathcal{C}) = \{(x_{\pi^{-1}(1)}, \dots, x_{\pi^{-1}(n)}) \mid \mathbf{x} \in \mathcal{C}\}.$$

Throughout the chapter we use the following notation. For a $k \times n$ matrix M and any ordered subset $\{j_1, \dots, j_m\} = J \subset \{1, \dots, n\}$ we denote by $M_{.J}$ the submatrix of M consisting of the columns corresponding to the indices of J and $M_{J'} = (M^t)_{.J'}$.

2.2 Binary Goppa Codes

In this section we give definition of binary Goppa Codes. Then we describe how to construct a generator and parity check matrix for Goppa codes.

Definition 2.3. *Let m and t be positive integers and let*

$$g(X) = \sum_{i=0}^t g_i X^i \in \mathbb{F}_{2^m}[X]$$

be a monic polynomial of degree t . We call $g(X)$ Goppa polynomial. Let

$$\mathbf{L} = (\gamma_0, \dots, \gamma_{n-1}) \in \mathbb{F}_{2^m}^n$$

be a tuple of n distinct elements, such that

$$g(\gamma_i) \neq 0, \text{ for all } 0 \leq i < n.$$

For any vector $\mathbf{c} = (c_0, \dots, c_{n-1}) \in \mathbb{F}_2^n$,

$$S_{\mathbf{c}}(X) = - \sum_{i=0}^{n-1} \frac{c_i}{g(\gamma_i)} \frac{g(X) - g(\gamma_i)}{X - \gamma_i}$$

defines the syndrome of \mathbf{c} . Then the binary Goppa code $\mathcal{G}(\mathbf{L}, g(X))$ over \mathbb{F}_2 is the set of all vectors $\mathbf{c} = (c_0, \dots, c_{n-1}) \in \mathbb{F}_2^n$, for which

$$S_{\mathbf{c}}(X) = 0$$

holds in the polynomial ring $\mathbb{F}_{2^m}[X]$ or equivalently if

$$S_{\mathbf{c}}(X) \equiv \sum_{i=0}^{n-1} \frac{c_i}{X - \gamma_i} \equiv 0 \pmod{g(X)}.$$

So formally we get

$$\mathcal{G}(\mathbf{L}, g(X)) = \{\mathbf{c} \in \mathbb{F}_2^n \mid S_{\mathbf{c}}(X) = 0\} = \{\mathbf{c} \in \mathbb{F}_2^n \mid S_{\mathbf{c}}(X) \equiv 0 \pmod{g(X)}\}.$$

If $g(X)$ is irreducible over \mathbb{F}_{2^m} then we call $\mathcal{G}(\mathbf{L}, g(X))$ irreducible binary Goppa code. If $g(X)$ is irreducible, then $g(\gamma) \neq 0$ for all $\gamma \in \mathbb{F}_{2^m}$. Thus, in such case, \mathbf{L} may contain all elements of \mathbb{F}_{2^m} .

Now we will show how to construct the generator matrix G and the parity check matrix H of a Goppa code $\mathcal{G}(\mathbf{L}, g(X))$. A vector $\mathbf{c} \in \mathbb{F}_2^n$ is contained in $\mathcal{G}(\mathbf{L}, g(X))$ if and only if $S_{\mathbf{c}}(X) = 0$. For all $i = 0, \dots, n-1$ we have

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

Therefore, we get that $\mathbf{c} \in \mathcal{G}(\mathbf{L}, g(X))$ if for all $s = 0, \dots, t-1$

$$\sum_{i=0}^{n-1} \frac{1}{g(\gamma_i)} \sum_{j=s+1}^t g_j \gamma_i^{j-1-s} c_i = 0. \quad (2.1)$$

is fulfilled. So for $s = 0, \dots, t-1$ (2.1) can be rewritten as

$$H\mathbf{c}^T = 0, \text{ where } H = \begin{pmatrix} g_t g(\gamma_0)^{-1} & \cdots & g_t g(\gamma_{n-1})^{-1} \\ (g_{t-1} + g_t \gamma_0) g(\gamma_0)^{-1} & \cdots & (g_{t-1} + g_t \gamma_{n-1}) g(\gamma_{n-1})^{-1} \\ \vdots & \ddots & \vdots \\ \left(\sum_{j=1}^t g_j \gamma_0^{j-1}\right) g(\gamma_0)^{-1} & \cdots & \left(\sum_{j=1}^t g_j \gamma_{n-1}^{j-1}\right) g(\gamma_{n-1})^{-1} \end{pmatrix} \quad (2.2)$$

From (2.2) it follows that

$$\mathcal{G}(\mathbf{L}, g(X)) = \{\mathbf{c} \in \mathbb{F}_2^n \mid H\mathbf{c}^T = 0\}.$$

Therefore H is the *parity check matrix* of $\mathcal{G}(\mathbf{L}, g(X))$. Since $H\mathbf{c}^T = 0$ holds for all $c \in \mathcal{G}(\mathbf{L}, g(X))$, the matrix H can be used to determine whether a vector $\mathbf{c} \in \mathbb{F}_2^n$ is a codeword from \mathcal{G} or not. The entries of the matrix H are elements of the extension field \mathbb{F}_{2^m} over \mathbb{F}_2 . So we can extend H to a $mt \times n$ matrix H' over \mathbb{F}_2 by writing each entry as the corresponding column vector of length m from \mathbb{F}_2 . The rows of the matrix H' generate a vector subspace V of \mathbb{F}_2^n . Therefore, we can obtain a generator matrix G of a Goppa code by computing the basis of the vector space dual to V . Since H' has dimension $mt \times n$, then G is a $n \times k$ matrix with $k \geq n - mt$. So the matrix G defines a (n, k) Goppa code.

The minimum distance of two vectors $\mathbf{x}, \mathbf{y} \in \mathcal{G}(\mathbf{L}, g(X))$, where $g(X)$ is an irreducible polynomial of degree t , is at least $2t + 1$. Therefore, the Goppa code \mathcal{G} can correct up to t errors. More detailed discussion about the minimum distance of error-correcting codes can be found in [Bay97].

2.3 The McEliece Cryptosystem

The McEliece Cryptosystem is the first one, which uses error-correcting codes as a trapdoor. The original version of the McEliece cryptosystem, which uses Goppa codes, remains unbroken. The trapdoor for the McEliece cryptosystem, using Goppa codes, is the knowledge of the Goppa polynomial. The McEliece cryptosystem can be described as follows:

System Parameters: $(n, t) \in \mathbb{N}$, $t \ll n$

Key Generation: Given the parameters n and t generate the following matrices:

G' : $k \times n$ generator matrix of a binary irreducible (n, k) Goppa code \mathcal{G} , which can correct up to t errors, where k is chosen maximal.

S : $k \times k$ random binary non-singular matrix

P : $n \times n$ random permutation matrix.

Public Key: (G, t) with $G = SG'P$.

Private Key: $(S, D_{\mathcal{G}}, P)$, where $D_{\mathcal{G}}$ is an efficient decoding algorithm for $D_{\mathcal{G}}$.

Encryption: To encrypt a plaintext $\mathbf{m} \in \{0, 1\}^k$ choose randomly a vector $\mathbf{e} \in \{0, 1\}^n$ of weight t and compute the ciphertext $\mathbf{c} \in \{0, 1\}^n$ in the following way:

$$\mathbf{c} = \mathbf{m}G \oplus \mathbf{e}.$$

Decryption: To decrypt a ciphertext \mathbf{c} calculate

$$\mathbf{c}P^{-1} = (\mathbf{m}S)G' \oplus \mathbf{e}P^{-1}$$

first and apply the decoding algorithm $D_{\mathcal{G}}$ for \mathcal{G} to it. Since $\mathbf{c}P^{-1}$ has a Hamming distance of t to the Goppa code, we obtain the codeword

$$\mathbf{m}SG' = D_{\mathcal{G}}(\mathbf{c}P^{-1}).$$

Let $J \subset 1, \dots, n$ be a set, such that $G'_{.J}$ is invertible, then we can compute the plaintext as follows:

$$\mathbf{m} = (\mathbf{m}SG')_J (G'_{.J})^{-1} S^{-1}.$$

2.4 Reducing the Problem of Finding the Error Vector to the Problem of Finding the Shortest Lattice Vector

According to Conway and Sloane's "Construction A" [CS88], we can associate with any linear code \mathcal{C} a lattice $\mathcal{L}_{\mathcal{C}} \subset \mathbb{Z}^n$, where $\mathcal{L}_{\mathcal{C}}$ is the preimage of \mathcal{C} under the reduction map $\mathbb{Z}^n \rightarrow \mathbb{F}_2^n$. With the following lemma we show in details how the lattice associated with a linear code \mathcal{C} is constructed.

Lemma 2.4. *Let*

$$\sigma : \mathbb{F}_2^n \rightarrow \mathbb{Z}^n$$

be a map, which sends each zero entry of a vector $v \in \mathbb{F}_2^n$ to 0 and each non-zero entry to 1 or -1. Let

$$\pi : \mathbb{Z}^n \rightarrow \mathbb{F}_2^n, \quad \pi(\mathbf{v}) = (v_1 \bmod 2, \dots, v_n \bmod 2).$$

Let \mathcal{C} be an (n, k) binary linear code with generator matrix G , where G is in systematic form, i.e.

$$G = [\text{Id}(k) \mid A],$$

where $\text{Id}(k)$ is the $k \times k$ identity matrix and A is a $k \times (n - k)$ binary matrix. Let

$$\hat{G} := \begin{pmatrix} \sigma(\mathbf{g}_1) \\ \vdots \\ \sigma(\mathbf{g}_k) \end{pmatrix} = \begin{pmatrix} \hat{\mathbf{g}}_1 \\ \vdots \\ \hat{\mathbf{g}}_k \end{pmatrix}$$

be the matrix obtained by applying σ on the rows of the matrix G and $\mathcal{L}_{\mathcal{C}} := \pi^{-1}(\mathcal{C})$. Then $\mathcal{L}_{\mathcal{C}}$ is the lattice generated by the vectors $\hat{\mathbf{g}}_1, \dots, \hat{\mathbf{g}}_k, 2 \cdot \mathbf{e}_{k+1}, \dots, 2 \cdot \mathbf{e}_n$.

Proof of Lemma 2.4: Let

$$M := \langle \hat{\mathbf{g}}_1, \dots, \hat{\mathbf{g}}_k, 2 \cdot \mathbf{e}_{k+1}, \dots, 2 \cdot \mathbf{e}_n \rangle = \left\{ \sum_{i=1}^k \alpha_i \hat{\mathbf{g}}_i + \sum_{j=k+1}^n \alpha_j \cdot 2\mathbf{e}_j : \alpha \in \mathbb{Z} \right\}$$

be the space generated by the vectors $\hat{\mathbf{g}}_1, \dots, \hat{\mathbf{g}}_k, 2 \cdot \mathbf{e}_{k+1}, \dots, 2 \cdot \mathbf{e}_n$ in \mathbb{Z}^n .

1. $M \subset \pi^{-1}(\mathcal{C})$

Let $\mathbf{v} \in M$. Then there exist $\alpha_i \in \mathbb{Z}$, $i = 1, \dots, n$, such that

$$\mathbf{v} = \sum_{i=1}^k \alpha_i \hat{\mathbf{g}}_i + \sum_{j=k+1}^n \alpha_j \cdot 2\mathbf{e}_j.$$

Applying π on the vector \mathbf{v} we get

$$\pi(\mathbf{v}) = \sum_{i=1}^k (\alpha_i \bmod 2) \begin{pmatrix} \hat{g}_{i,1} \bmod 2 \\ \vdots \\ \hat{g}_{i,n} \bmod 2 \end{pmatrix} = \sum_{i=1}^k \beta_i \begin{pmatrix} g_{i,1} \\ \vdots \\ g_{i,n} \end{pmatrix} \in \mathcal{C},$$

where $\beta_i \in \mathbb{F}_2$ and $\beta_i = (\alpha_i \bmod 2)$. So there exists a codeword $\mathbf{c} \in \mathcal{C}$, such that $\pi(\mathbf{v}) = \mathbf{c}$ and therefore $\mathbf{v} \in \pi^{-1}(\mathcal{C}) = \mathcal{L}_{\mathcal{C}}$.

2. $\pi^{-1}(\mathcal{C}) \subset M$

Let $\mathbf{w} \in \mathcal{L}_{\mathcal{C}}$. Then there exists a codeword $\mathbf{c} \in \mathcal{C}$, such that $\pi(\mathbf{w}) = \mathbf{c}$. Therefore \mathbf{w} is congruent to \mathbf{c} modulo 2 and can be written as

$$\mathbf{w} = \sum_{i=1}^k \beta_i \hat{\mathbf{g}}_i + \sum_{j=1}^n \gamma_j \cdot 2\mathbf{e}_j \text{ for } \beta_i, \gamma_j \in \mathbb{Z}.$$

If the vectors $2\mathbf{e}_1, \dots, 2\mathbf{e}_k \in M$, we can write the vector \mathbf{w} as

$$\mathbf{w} = \sum_{i=1}^k \nu_i \hat{\mathbf{g}}_i + \sum_{j=k+1}^n \nu_j \cdot 2\mathbf{e}_j \text{ for } \nu_i \in \mathbb{Z},$$

which proves lemma. So the only thing left is to show that $2\mathbf{e}_1, \dots, 2\mathbf{e}_k \in M$.

Since G is in systematic form, for the entries of $\hat{\mathbf{g}}_1, \dots, \hat{\mathbf{g}}_k$ we have that $\hat{g}_{i,i} = 1$ or $\hat{g}_{i,i} = -1$ and $\hat{g}_{i,j} = 0$ for $i, j = 1, \dots, k$. Let

$$\hat{\mathcal{T}}_i = \{j : k+1 \leq j \leq n, \hat{g}_{i,j} = 1\} \quad \text{and} \quad \hat{\mathcal{S}}_i = \{j : k+1 \leq j \leq n, \hat{g}_{i,j} = -1\}$$

for $i = 1, \dots, k$. For $i = 1, \dots, k$ we get

$$2\mathbf{e}_i = \begin{cases} 2 \cdot \hat{\mathbf{g}}_i + (-1) \sum_{j \in \hat{\mathcal{T}}_i} 2 \cdot \mathbf{e}_j + \sum_{j \in \hat{\mathcal{S}}_i} 2 \cdot \mathbf{e}_j & \text{if } \hat{g}_{i,i} = 1 \\ (-2) \cdot \hat{\mathbf{g}}_i + \sum_{j \in \hat{\mathcal{T}}_i} 2 \cdot \mathbf{e}_j + (-1) \cdot \sum_{j \in \hat{\mathcal{S}}_i} 2 \cdot \mathbf{e}_j & \text{if } \hat{g}_{i,i} = -1, \end{cases}$$

which finishes the proof of the lemma. \square

Let \mathcal{C} be an (n, k, d) error-correcting binary linear code with generator matrix G , which can correct up to $t := \lfloor \frac{d-1}{2} \rfloor$ errors. Let $\mathbf{m} \in \mathbb{F}_2^k$ be a k -dimensional vector, $\mathbf{e} \in \mathbb{F}_2^n$ be a vector with weight smaller or equal to t and $\mathbf{c} = \mathbf{m}G \oplus \mathbf{e}$. Let

$$\tilde{G} := \begin{pmatrix} G \\ \mathbf{c} \end{pmatrix} = \begin{pmatrix} G \\ \mathbf{m}G \oplus \mathbf{e} \end{pmatrix} = \begin{pmatrix} \mathbf{g}_1 \\ \vdots \\ \mathbf{g}_m \\ \mathbf{m}G \oplus \mathbf{e} \end{pmatrix} \quad (2.3)$$

and $\tilde{\mathcal{C}}$ be the linear code generated by \tilde{G} in \mathbb{F}_2^n . The minimum weight of any non-zero codeword in \mathcal{C} is $2t + 1$. With the following lemma we will show that the only non-zero codeword in $\tilde{\mathcal{C}}$ with weight smaller or equal to t is the vector \mathbf{e} .

Lemma 2.5. *The only non-zero codeword with weight smaller or equal to t in $\tilde{\mathcal{C}}$ is \mathbf{e} .*

Proof: Since $\mathbf{c} = \mathbf{m}G \oplus \mathbf{e}$ it is obvious that $\tilde{\mathcal{C}}$ is generated by the matrix

$$\tilde{G}' := \begin{pmatrix} G \\ \mathbf{e} \end{pmatrix}.$$

Every vector \mathbf{v} generated by the rows of \tilde{G}' has the following form

$$\mathbf{v} = \mathbf{v}_1 \oplus \alpha \mathbf{e} \text{ or equivalently } \mathbf{v} \oplus \alpha \mathbf{e} = \mathbf{v}_1 \quad (2.4)$$

where $\alpha \in \{0, 1\}$ and \mathbf{v}_1 is codeword from the linear code generated by G . Therefore $\text{wt}(\mathbf{v}_1) \geq 2t + 1$. If $\alpha = 0$, then $\mathbf{v} = \mathbf{v}_1$ and $\text{wt}(\mathbf{v}) = \text{wt}(\mathbf{v}_1) \geq 2t + 1$. Let $\alpha = 1$. Since the Hamming distance is a metric over the vector space generated by the rows of \tilde{G}' and the triangle inequality

$$\text{wt}(\mathbf{u} + \mathbf{w}) \leq \text{wt}(\mathbf{u}) + \text{wt}(\mathbf{w})$$

is fulfilled for all $u, w \in \tilde{\mathcal{C}}$, from (2.4) we get

$$\begin{aligned} \text{wt}(\mathbf{v}) + \text{wt}(\mathbf{e}) &\geq \text{wt}(\mathbf{v} + \mathbf{e}) = \text{wt}(\mathbf{v}_1) \\ \text{wt}(\mathbf{v}) &\geq \text{wt}(\mathbf{v}_1) - \text{wt}(\mathbf{e}) \geq (2t + 1) - t = t + 1, \end{aligned}$$

which proves the lemma. \square

Let $\mathcal{L}_{\tilde{\mathcal{C}}}$ be the lattice in \mathbb{Z}^n associated with the code $\tilde{\mathcal{C}}$. From Lemma 2.5 we know that the codeword with the smallest weight in $\tilde{\mathcal{C}}$ is the error vector \mathbf{e} and \mathbf{e} is the only such codeword in $\tilde{\mathcal{C}}$. So we would like to choose a norm $\|\cdot\|$, such that the shortest non-zero vector w.r.t $\|\cdot\|$ is a vector \mathbf{v} , for which $\mathbf{v} \bmod 2 = \mathbf{e}$. We choose the l_p norm. With the following lemma we give a lower bound for p , so that for the shortest non-zero vector \mathbf{v} w.r.t l_p norm in $\mathcal{L}_{\tilde{\mathcal{C}}}$ holds:

$$\mathbf{v} \bmod 2 = \mathbf{e} \text{ and } \|\mathbf{v}\|_p = \text{wt}(\mathbf{e})^{1/p} = t^{1/p}.$$

Lemma 2.6. *Let \mathbf{v} be a shortest non-zero lattice vector in $\mathcal{L}_{\tilde{\mathcal{C}}}$ w.r.t the l_p norm. If*

$$p > \log_2 t,$$

where t is the weight of the error vector \mathbf{e} in $\tilde{\mathcal{C}}$, then $v_i \in \{-1, 0, 1\}$ and $v_i = e_i \bmod 2$ for $i = 1, \dots, n$.

Proof: First we will prove that the vector \mathbf{v} has entries only from $\{-2, -1, 0, 1, 2\}$. Suppose there exists an entry v_j of \mathbf{v} , such that $v_j \bmod 2 = 1$ and $v_j \notin \{-1, 1\}$. Then the vector

$$\mathbf{w} = (v_1, \dots, v_{j-1}, 1, v_{j+1}, \dots, v_n) \in \mathcal{L}_{\tilde{\mathcal{C}}}$$

has smaller l_p norm than the norm of \mathbf{v} , which is a contradiction to the fact that \mathbf{v} is a shortest vector in $\mathcal{L}_{\tilde{\mathcal{C}}}$. Therefore for all entries of \mathbf{v} , such that $v_j \bmod 2 = 1$, holds $v_j \in \{-1, 1\}$. Suppose that there exists an entry v_k of \mathbf{v} , such that $v_k \bmod 2 = 0$ and $|v_k| > 2$. Then the vector

$$\mathbf{w} = (v_1, \dots, v_{k-1}, 2, v_{k+1}, \dots, v_n) \in \mathcal{L}_{\tilde{\mathcal{C}}}$$

has smaller l_p norm than the l_p norm of \mathbf{v} , which is a contradiction to the fact that \mathbf{v} is a shortest vector in $\mathcal{L}_{\tilde{\mathcal{C}}}$. Therefore for all entries of \mathbf{v} , such that $v_k \bmod 2 = 0$, holds $v_j \in \{-2, 0, 2\}$.

Next we will show that the vector \mathbf{v} has either entries only from $\{-1, 0, 1\}$ or $\mathbf{v} \in \pm 2\text{Id}(n)$. Suppose that there exists an entry v_j of \mathbf{v} , such that $v_j = 2$ or $v_j = -2$. Consider the vectors $\mathbf{w}_1 = 2\mathbf{e}_j$ and $\mathbf{w}_2 = -2\mathbf{e}_j$. Obviously $\|\mathbf{w}_i\|_p \leq \|\mathbf{v}\|_p$ for $i = 1, 2$, where the equality is reached if and only if $\mathbf{v} = \mathbf{w}_i$ for some $i \in \{1, 2\}$. Therefore, if there exists an entry v_j of \mathbf{v} , such that $v_j = 2$ or $v_j = -2$, then $\mathbf{v} \in \pm 2\text{Id}(n)$. If there is no entry of \mathbf{v} , such that $v_j = 2$ or $v_j = -2$, then all entries of \mathbf{v} are only from $\{-1, 0, 1\}$. Since the lattice $\mathcal{L}_{\tilde{\mathcal{C}}}$ contains all vectors in \mathbb{Z}^n , which are congruent modulo 2 to a codeword in $\tilde{\mathcal{C}}$, all the vectors \mathbf{e}' congruent to \mathbf{e} modulo 2 with entries only from $\{-1, 0, 1\}$ belong to $\mathcal{L}_{\tilde{\mathcal{C}}}$ and $\|\mathbf{e}'\|_p = t^{1/p}$. Therefore all shortest vectors in $\mathcal{L}_{\tilde{\mathcal{C}}}$ have l_p norm equal or smaller than $t^{1/p}$.

Let $\mathbf{e}' \in \mathcal{L}_{\tilde{\mathcal{C}}}$ be a vector congruent to \mathbf{e} modulo 2 with entries only from $\{-1, 0, 1\}$. Assume $\mathbf{v} \in \pm 2 \cdot \text{Id}(n)$. From $p > \log_2 t$ we get

$$\|\mathbf{e}'\|_p = t^{1/p} < 2 = \|\mathbf{v}\|_p,$$

which is a contradiction to the fact that \mathbf{v} is a shortest vector in $\mathcal{L}_{\tilde{\mathcal{C}}}$.

Therefore $\mathbf{v} \neq 2\mathbf{e}_k$ for all $k = 1, \dots, n$, $v_i \in \{-1, 0, 1\}$ for all $k = 1, \dots, n$ and $\|\mathbf{v}\|_p \leq t^{1/p}$. Since $\mathbf{v} \in \mathcal{L}_{\tilde{\mathcal{C}}}$ there exists a codeword $c \in \tilde{\mathcal{C}}$ such that

$$c_i = \begin{cases} 0 & v_i = 0 \\ 1 & v_i = \pm 1 \end{cases}$$

and $\text{wt}(\mathbf{c}) \leq t$. From Lemma 2.5 we know that the only such codeword in $\tilde{\mathcal{C}}$ is \mathbf{e} . So it follows that for the entries of \mathbf{v} holds

$$v_i = \begin{cases} 0 & e_i = 0 \\ \pm 1 & e_i = 1 \end{cases}.$$

□

Using these lemmas we propose the following attack on the McEliece cryptosystem: Let (G, t) be the public key of the cryptosystem, where $G \in \mathbb{F}_2^{k \times n}$ and $\mathbf{c} = \mathbf{m}G \oplus \mathbf{e}$ be a ciphertext. First we construct the matrix \tilde{G} as in (2.3). Since the rank of the matrix \tilde{G} is k , it is always possible, using gaussian elimination and column permutation, to find matrices $R \in \mathbb{F}_2^{(k+1) \times (k+1)}$, \tilde{G}^{sys} and $W \in \mathbb{F}_2^{n \times n}$, such that

$$R \cdot \tilde{G}^{\text{sys}} \cdot W = \tilde{G},$$

where R is the transformation matrix, corresponding to the row permutations and row sums used in the gaussian elimination, W is a column permutation matrix and

$$\tilde{G}^{\text{sys}} = [\text{Id}(k+1) | A].$$

for a matrix $A \in \{0, 1\}^{(k+1) \times (n-k-1)}$. The codes generated by \tilde{G} and \tilde{G}^{sys} are by Definition 2.2 permutation equivalent w.r.t to the permutation map

$$\phi : \mathbb{F}_2^n \rightarrow \mathbb{F}_2^n, \phi(\mathbf{v}) = v \cdot W.$$

Therefore the error vector \mathbf{e} for the ciphertext \mathbf{c} in \tilde{G} corresponds to the vector $\phi^{-1}(\mathbf{e})$ in \tilde{G}^{sys} .

Then we fix a map σ (see Lemma 2.4) and apply σ to the rows of \tilde{G}^{sys} . We choose the map σ , so that

$$\begin{pmatrix} \sigma(\mathbf{g}_1^{\text{sys}}) \\ \vdots \\ \sigma(\mathbf{g}_{k+1}^{\text{sys}}) \end{pmatrix} = [\text{Id}(k+1) | D], \quad (2.5)$$

for $D \in \{-1, 0, 1\}^{(k+1) \times (n-k-1)}$. Then we construct the rows of the input matrix for the lattice basis reduction algorithm B^{in} from the generator vectors of the lattice $\mathcal{L}_{\tilde{\mathcal{C}}^{\text{sys}}}$, associated with the code $\tilde{\mathcal{C}}^{\text{sys}}$, generated by \tilde{G}^{sys} , as in Lemma 2.4. The matrix B^{in} has the following form

$$B^{\text{in}} = \begin{pmatrix} \text{Id}(k+1) & D \\ \mathbf{0} & 2 \cdot \text{Id}(n-k-1) \end{pmatrix}.$$

The output of the basis reduction algorithms depend on the order of the rows in the input matrix. In our tests we consider the following constructions of \tilde{G}

$$\tilde{G}_1 = \begin{pmatrix} G \\ \mathbf{c} \end{pmatrix} = \begin{pmatrix} G \\ \mathbf{m}G \oplus \mathbf{e} \end{pmatrix} \quad \text{or} \quad \tilde{G}_2 = \begin{pmatrix} \mathbf{c} \\ G \end{pmatrix} = \begin{pmatrix} \mathbf{m}G \oplus \mathbf{e} \\ G \end{pmatrix}.$$

Let $\tilde{G}^{\text{sys}_i} = [\text{Id}(k+1) | A_i]$, $i = 1, 2$ be the corresponding to \tilde{G}_i systematic matrix and

$$\begin{pmatrix} \sigma(\mathbf{g}_1^{\text{sys}_i}) \\ \vdots \\ \sigma(\mathbf{g}_{k+1}^{\text{sys}_i}) \end{pmatrix} = [\text{Id}(k+1) | D_i], \quad (2.6)$$

Then we consider the following permutations of the rows of B^{in}

$$B_{i,1}^{\text{in}} = \begin{pmatrix} \text{Id}(k+1) & D_i \\ \mathbf{0} & 2 \cdot \text{Id}(n-k-1) \end{pmatrix} \text{ or } B_{i,2}^{\text{in}} = \begin{pmatrix} \mathbf{0} & 2 \cdot \text{Id}(n-k-1) \\ \text{Id}(k+1) & D_i \end{pmatrix}. \quad (2.7)$$

Finally we apply lattice basis reduction algorithms on the matrix B^{in} , where B^{in} is one of the matrices in (2.7). The results of the attack for the different constructions of the input matrix are given in Chapter 5. Let B^{out} be the output matrix of the lattice basis reduction. The lattice basis reduction algorithms can find short vectors, which are only approximation of the length of the shortest lattice vector. Therefore we cannot expect that this attack will always work. In order to determine whether the attack is successful or not, we consider the following cases:

- There is a row vector \mathbf{b}_j in B^{out} with l_p norm smaller or equal to $t^{1/p}$, such that $\mathbf{b}_j \notin \pm 2\text{Id}(n)$. Then from Lemma 2.6 we know that $\mathbf{b}_j \bmod 2 = \phi^{-1}(\mathbf{e})$ and therefore $\phi(\mathbf{b}_j \bmod 2)$ is the error vector, we were looking for. In most cases we expect that this vector is the first row vector of B^{out} , but this is not necessarily so.
- There is no vector with l_p norm smaller or equal to $t^{1/p}$, but there exists a row vector $\mathbf{b}_j \notin \pm 2\text{Id}(n)$ in B^{out} with l_p norm $(t+k)^{1/p}$ for some $k = 1, \dots, t$. By construction all rows of B^{out} are congruent modulo 2 to a codeword from \mathcal{C}^{sys} . From the proof of Lemma 2.5 we know that the codewords with weight bigger than t and smaller than $2t+1$ can be represented as

$$\mathbf{v} = \mathbf{v}_1 \oplus \alpha \tilde{\mathbf{e}},$$

with $\alpha = 1$ and $\text{wt}(\mathbf{v}_1) = 2t+1$. Hence from such basis vector \mathbf{b}_j we can get the following information about the entries of the vector $\tilde{\mathbf{e}} = \phi^{-1}(\mathbf{e})$:

$$t - k + 1 \leq |\{1 \leq i \leq n \mid \tilde{e}_i = 1\} \cap \{1 \leq i \leq n \mid b_{j,i} = 0\}| \leq t.$$

- In B^{out} there are no vectors with weight smaller than $2t+1$. Then the attack was totally unsuccessful.

Chapter 3

Block Basis Reduction for the Euclidean Norm

In this chapter we describe in details the L^3 and the block basis reduction algorithm for the Euclidean norm. First we give a brief introduction to the lattice theory. Then we describe the L^3 basis reduction algorithm, which is a part of the block basis reduction algorithm. We consider in details the approximation of the length shortest lattice vector, which these algorithms give.

3.1 Introduction to Lattices

We begin with the definition of a lattice:

Definition 3.1. A lattice $\mathcal{L} \subset \mathbb{R}^n$ is a discrete additive subgroup of \mathbb{R}^n , such that

$$\mathcal{L} = \{t_1 \mathbf{b}_1 + \dots + t_m \mathbf{b}_m \mid t_1, \dots, t_m \in \mathbb{Z}\},$$

where $\mathbf{b}_1, \dots, \mathbf{b}_m \in \mathbb{R}^n$ are linearly independent vectors. We call $(\mathbf{b}_1, \dots, \mathbf{b}_m)$ a basis of the lattice $\mathcal{L} = \mathcal{L}(\mathbf{b}_1, \dots, \mathbf{b}_m)$. The number of the basis vectors is called rank or dimension of \mathcal{L} . A lattice \mathcal{L} is called integral if $\mathcal{L} \subseteq \mathbb{Z}^n$. For $m = n$, \mathcal{L} is called full-dimensional.

The basis of a lattice is not uniquely determined. If $\bar{\mathbf{b}}_1, \dots, \bar{\mathbf{b}}_m \in \mathbb{R}^n$ is another basis of $\mathcal{L}(\mathbf{b}_1, \dots, \mathbf{b}_m)$, then there exists an invertible matrix $T \in \mathbb{Z}^{m \times m}$, such that

$$|\det(T)| = 1 \quad \text{and} \quad (\mathbf{b}_1, \dots, \mathbf{b}_m) = (\bar{\mathbf{b}}_1, \dots, \bar{\mathbf{b}}_m)T.$$

With $S_n(M, r, \|\cdot\|) := \{x \in \mathbb{R}^n : \|x - M\| \leq r\}$ we denote the n -dimensional sphere with center the point M and radius r , where $\|\cdot\|$ is an arbitrary norm over \mathbb{R}^n .

Definition 3.2. Let $\|\cdot\|$ be an arbitrary norm over \mathbb{R}^n , $\mathcal{L} \subset \mathbb{R}^n$ be an m -dimensional lattice and $i \in \mathbb{N}$ with $1 \leq i \leq m$. Then the minimal number $r \in \mathbb{R}_{>0}$, such that the sphere $S_n(0, r, \|\cdot\|)$ contains i linearly independent vectors $\mathbf{v}_1, \dots, \mathbf{v}_i \in \mathcal{L}$ is called the i -th successive minimum of the lattice \mathcal{L} (w.r.t the norm $\|\cdot\|$). We denote the i -th successive minimum of \mathcal{L} with $\lambda_{i, \|\cdot\|}(\mathcal{L})$. Formally, the i -th successive minimum of \mathcal{L} is given by

$$\lambda_{i, \|\cdot\|}(\mathcal{L}) = \inf\{r \mid \dim(\text{span}(\mathcal{L}) \cap S(0, r, \|\cdot\|)) \geq i\}.$$

In particular, $\lambda_1(\mathcal{L})$ is the length (w.r.t. $\|\cdot\|$) of the shortest non-zero vector in \mathcal{L} and a vector x with $\|x\| = \lambda_1(\mathcal{L})$ is called a *shortest vector* in \mathcal{L} . The successive minima are lattice invariants, i.e. they do not depend on the choice of the basis of the lattice.

Definition 3.3. Let $B = (\mathbf{b}_1, \dots, \mathbf{b}_m)$ be a basis of the lattice $\mathcal{L} \subset \mathbb{R}^n$. The fundamental parallelepiped for the basis B is defined as

$$P(B) = \left\{ x = \sum_{i=1}^m x_i \mathbf{b}_i \mid 0 \leq x_i < 1 \right\}.$$

The determinant $\det(\mathcal{L})$ of the lattice $\mathcal{L}(B)$ is the volume of $P(B)$

$$\det(\mathcal{L}) = \text{vol}_m P(B) = \det(B^t B)^{\frac{1}{2}}.$$

The determinant doesn't depend on the choice of the basis of \mathcal{L} . Let B, \bar{B} be different bases of \mathcal{L} with $\bar{B} = BT$. Then, due to $|\det(T)| = 1$, we obtain $\det(\mathcal{L}) = \det(B^t B) = \det(T^t B^t B T) = \det(\bar{B}^t \bar{B})$.

3.2 The L^3 Basis Reduction Algorithm

We denote with $\|\cdot\|_2$ the Euclidean norm over \mathbb{R}^n and with $\langle \cdot, \cdot \rangle_2$ the corresponding inner product.

Definition 3.4. For a basis $(\mathbf{b}_1, \dots, \mathbf{b}_m)$ of a lattice \mathcal{L} the associated orthogonal basis $(\mathbf{b}_1^*, \dots, \mathbf{b}_m^*)$ can be computed using the Gram-Schmidt method:

$$\mathbf{b}_1^* = \mathbf{b}_1, \tag{3.1}$$

$$\mathbf{b}_i^* = \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \mathbf{b}_j^* \quad \text{for } 2 \leq i \leq m, \tag{3.2}$$

$$\mu_{i,j} = \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle_2}{\|\mathbf{b}_j^*\|_2^2} \quad \text{for } 1 \leq j < i \leq m. \tag{3.3}$$

We call the coefficients $\mu_{i,j}$ Gram-Schmidt coefficients. The resulting Gram-Schmidt orthogonalization $(\mathbf{b}_1^*, \dots, \mathbf{b}_m^*)$ strongly depends on the order of the given basis vectors in $(\mathbf{b}_1, \dots, \mathbf{b}_m)$. With $\mu_{i,i} = 1$ for $1 \leq i \leq m$ and $\mu_{i,j} = 0$ for $i < j$, we can construct a lower triangular matrix $M = (\mu_{i,j})_{1 \leq i, j \leq m}$, such that

$$(\mathbf{b}_1, \dots, \mathbf{b}_m) = (\mathbf{b}_1^*, \dots, \mathbf{b}_m^*) M^t.$$

The vectors $(\mathbf{b}_1^*, \dots, \mathbf{b}_m^*)$ are linearly independent, but they are not necessarily in the lattice. If the vectors $(\mathbf{b}_1, \dots, \mathbf{b}_m)$ are integral, then the vectors $(\mathbf{b}_1^*, \dots, \mathbf{b}_m^*)$ and the coefficients $\mu_{i,j}$ are rational. We denote the orthogonal projections onto $\text{span}(\mathbf{b}_1, \dots, \mathbf{b}_{i-1})^\perp$ with π_i :

$$\begin{aligned} \pi_i : \mathbb{R}^n &\rightarrow \text{span}(\mathbf{b}_1, \dots, \mathbf{b}_{i-1})^\perp, \\ \pi_i(\mathbf{b}_i) &= \mathbf{b}_i^* && \text{for } i = 1, \dots, m, \\ \pi_i(\mathbf{b}_j) &:= \sum_{t=i}^j \mu_{j,t} \mathbf{b}_t^* && \text{for } i < j \text{ and } i = 1, \dots, m. \end{aligned}$$

Therefore $\|\pi_i(\mathbf{b}_j)\|_2$ gives the minimal distance of the vector \mathbf{b}_j to $\text{span}(\mathbf{b}_1, \dots, \mathbf{b}_{i-1})$ with respect to the Euclidean norm. With the following lemma we give some useful properties of the associated orthogonal bases:

Lemma 3.5. *Let $B = (\mathbf{b}_1, \dots, \mathbf{b}_m)$ be a basis of the lattice $\mathcal{L} \subset \mathbb{R}^n$ and $B^* = (\mathbf{b}_1^*, \dots, \mathbf{b}_m^*)$ be the associated orthogonal basis with B . Then we have:*

$$\begin{aligned}\langle \mathbf{b}_i^*, \mathbf{b}_i^* \rangle &= \langle \mathbf{b}_i^*, \mathbf{b}_i \rangle, \\ \|\mathbf{b}_i^*\|_2 &\leq \|\mathbf{b}_i\|_2.\end{aligned}$$

For a vector $\mathbf{v} = \sum_{i=1}^m x_i \mathbf{b}_i$, where $x_i \in \mathbb{R}$ for $1 \leq i \leq m$, we have:

$$\mathbf{v} = \sum_{i=1}^m \frac{\langle \mathbf{v}, \mathbf{b}_i^* \rangle}{\langle \mathbf{b}_i^*, \mathbf{b}_i^* \rangle} \mathbf{b}_i^*.$$

Furthermore, we have

$$\det(B^t B) = \det(B^{*t} B^*) = \prod_{i=1}^m \|\mathbf{b}_i^*\|_2^2 \leq \prod_{i=1}^m \|\mathbf{b}_i\|_2^2.$$

Corollary 3.6. *For a lattice $\mathcal{L} \subset \mathbb{R}^n$ with basis $(\mathbf{b}_1, \dots, \mathbf{b}_m)$ and associated orthogonal basis $(\mathbf{b}_1^*, \dots, \mathbf{b}_m^*)$, holds*

$$\det(\mathcal{L}) = \prod_{i=1}^m \|\mathbf{b}_i^*\|_2.$$

The following corollary gives us a lower bound for the shortest non-zero vector of a lattice.

Corollary 3.7. *Let $\mathcal{L} \subset \mathbb{R}^n$ be a lattice with basis $(\mathbf{b}_1, \dots, \mathbf{b}_m)$ and associated orthogonal basis $(\mathbf{b}_1^*, \dots, \mathbf{b}_m^*)$. Then the following holds:*

$$\min_{i=1, \dots, m} \{\|\mathbf{b}_i^*\|_2\} \leq \lambda_{1, \|\cdot\|_2}(\mathcal{L}).$$

Proof: Let \mathbf{v} be a shortest non-zero vector in \mathcal{L} : $\|\mathbf{v}\|_2 = \lambda_{1, \|\cdot\|_2}(\mathcal{L})$. Then it holds:

$$\mathbf{v} = \sum_{i=1}^m x_i \mathbf{b}_i = \sum_{i=1}^m x'_i \mathbf{b}_i^*,$$

where $x_i \in \mathbb{Z}$ and $x'_i \in \mathbb{R}$. Let k be the largest index with $x_k \neq 0$. Then $x_k = x'_k \in \mathbb{Z}$ and

$$\|\mathbf{v}\|_2^2 = \left\| \sum_{i=1}^k x'_i \mathbf{b}_i^* \right\|_2^2 = \sum_{i=1}^k (x'_i)^2 \|\mathbf{b}_i^*\|_2^2 \geq (x'_k)^2 \|\mathbf{b}_k^*\|_2^2 = x_k^2 \|\mathbf{b}_k^*\|_2^2 \geq \|\mathbf{b}_k\|_2^2.$$

□

In other words Corollary 3.7 implies that the Euclidean length of the shortest vector in \mathcal{L} is bigger or equal to the shortest Euclidean distance of a basis vector \mathbf{b}_i to $\text{span}(\mathbf{b}_1, \dots, \mathbf{b}_{i-1})$.

As an introduction to the lattice basis reduction algorithms we present first the most simple lattice basis reduction, introduced by Gauß. It allows computation of a minimal basis for 2-dimensional lattices.

Definition 3.8. We call a lattice basis $(\mathbf{b}_1, \mathbf{b}_2) \in \mathbb{R}^n$ Gauß-reduced (w.r.t $\|\cdot\|_2$) if

$$\|\mathbf{b}_1\|_2 \leq \|\mathbf{b}_2\|_2 \quad \text{and} \quad 0 \leq \mu_{2,1} \leq \frac{1}{2}.$$

The Gauß reduction algorithm can be presented as follows:

Algorithm 3.1 Gauß Basis Reduction Algorithm for the Euclidean Norm

INPUT: Lattice basis $B = (\mathbf{b}_1, \mathbf{b}_2) \in \mathbb{R}^n$ with $\|\mathbf{b}_1\| \leq \|\mathbf{b}_2\|$

OUTPUT: Gauß-reduced lattice basis

```

Calculate the Gram-Schmidt coefficient  $\mu_{2,1}$ 
while  $|\mu_{2,1}| > \frac{1}{2}$  do
     $\mathbf{b}_2 := \mathbf{b}_2 \cdot \text{sign}(\mu_{2,1})$ 
     $\mathbf{b}_2 := \mathbf{b}_2 - \lceil \mu_{2,1} \rceil \mathbf{b}_1$ 
    if  $\|\mathbf{b}_1\|_2 > \|\mathbf{b}_2\|_2$  then
        swap  $\mathbf{b}_1$  and  $\mathbf{b}_2$ 
    end if
    Calculate the Gram-Schmidt coefficient  $\mu_{2,1}$ 
end while
return  $(\mathbf{b}_1, \mathbf{b}_2)$ 

```

The L^3 algorithm, due to A.K. Lenstra, H.W. Lenstra and L. Lovász, extends the Gauß-reduction algorithm to lattices of arbitrary dimensions.

Definition 3.9. We call a lattice basis $(\mathbf{b}_1, \dots, \mathbf{b}_m) \in \mathbb{R}^n$ L^3 -reduced with δ for $\delta \in (0, 1]$ if

1. $|\mu_{i,j}| \leq \frac{1}{2}$ for $1 \leq j < i \leq m$,
2. $\delta \|\mathbf{b}_{k-1}^*\|_2^2 \leq \|\pi_{k-1}(\mathbf{b}_k)\|_2^2$ for $k = 2, \dots, m$,
or equivalently
 $\delta \|\mathbf{b}_{k-1}^*\|_2^2 \leq \|\mathbf{b}_k^* + \mu_{k,k-1} \mathbf{b}_{k-1}^*\|_2^2$.

Bases, which fulfill only the first condition, are called *size-reduced*. The first condition in Definition 3.9 means that the basis vectors are "almost orthogonal". Furthermore, if this condition holds, then the length of the basis vectors cannot be shortened any more by adding an integer multiple of \mathbf{b}_j to \mathbf{b}_i ($1 \leq j < i \leq m$). The parameter δ determines the quality of the approximation. With smaller δ the L^3 basis provides weaker approximation.

For $\delta > \frac{1}{4}$ the L^3 -reduced bases approximate the successive minima up to an exponential factor in the dimension of the lattice.

Theorem 3.10. [LLL82] Let $\delta \in (\frac{1}{4}, 1]$, $(\mathbf{b}_1, \dots, \mathbf{b}_m)$ be an L^3 -reduced basis with δ and $(\mathbf{b}_1^*, \dots, \mathbf{b}_m^*)$ be the orthogonal basis associated with $(\mathbf{b}_1, \dots, \mathbf{b}_m)$. Then for $\alpha := \frac{1}{\delta - \frac{1}{4}}$ and for $j = 1, \dots, m$ holds

$$\|\mathbf{b}_k\|_2^2 \leq \alpha^{j-1} \|\mathbf{b}_j^*\|_2^2 \quad \text{for } k \leq j, \quad (3.4)$$

$$\alpha^{1-j} \leq \|\mathbf{b}_j^*\|_2^2 \lambda_{j,\|\cdot\|_2}^{-2}(\mathcal{L}), \quad (3.5)$$

$$\alpha^{m-1} \geq \|\mathbf{b}_j\|_2^2 \lambda_{j,\|\cdot\|_2}^{-2}(\mathcal{L}). \quad (3.6)$$

In the proof of Theorem 3.10 we follow the proof given in [LLL82]. First we consider the following proposition:

Proposition 3.11. Let $(\mathbf{b}_1, \dots, \mathbf{b}_m)$ be a L^3 -reduced basis with δ . Then for $\alpha := \frac{1}{\delta - \frac{1}{4}}$ for $j = 1, \dots, m$ and $\delta \in (1, \frac{1}{4}]$ holds:

$$\|\mathbf{b}_k^*\|_2^2 \leq \alpha^{j-k} \|\mathbf{b}_j^*\|_2^2 \quad \text{for } 1 \leq k \leq j \leq m.$$

Proof: $(\mathbf{b}_1, \dots, \mathbf{b}_m)$ is an L^3 reduced basis. This implies:

$$\delta \|\mathbf{b}_{j-1}^*\|_2^2 \leq \|\mathbf{b}_j^* + \mu_{j,j-1} \mathbf{b}_{j-1}^*\|_2^2 \leq \|\mathbf{b}_j^*\|_2^2 + \mu_{j,j-1}^2 \|\mathbf{b}_{j-1}^*\|_2^2 \leq \|\mathbf{b}_j^*\|_2^2 + \frac{1}{4} \|\mathbf{b}_{j-1}^*\|_2^2.$$

This shows that

$$\begin{aligned} \|\mathbf{b}_j^*\|_2^2 &\leq \left(\delta - \frac{1}{4}\right) \|\mathbf{b}_{j-1}^*\|_2^2, \\ \|\mathbf{b}_{j-1}^*\|_2^2 &\leq \underbrace{\frac{1}{\delta - \frac{1}{4}}}_{\alpha} \|\mathbf{b}_j^*\|_2^2 \end{aligned}$$

holds for $1 < j \leq m$. By induction we obtain

$$\|\mathbf{b}_k^*\|_2^2 \leq \alpha^{j-k} \|\mathbf{b}_j^*\|_2^2$$

for $1 \leq k \leq j \leq m$. □

Proof of Theorem 3.10: Applying Proposition 3.11 we obtain that for all k and j , such that $k \leq j$ we have

$$\begin{aligned} \|\mathbf{b}_k\|_2^2 &= \|\mathbf{b}_k^* + \sum_{i=1}^{k-1} \mu_{k,i} \mathbf{b}_i^*\|_2^2 \\ &\leq \|\mathbf{b}_k^*\|_2^2 + \frac{1}{4} \sum_{i=1}^{k-1} \|\mathbf{b}_i^*\|_2^2 \\ &\leq \|\mathbf{b}_j^*\|_2^2 (\alpha^{j-k} + \frac{1}{4} \sum_{i=1}^{k-1} \alpha^{j-i}) \quad (\text{from Proposition 3.11}) \\ &\leq \|\mathbf{b}_j^*\|_2^2 \alpha^{j-1} (\alpha^{1-k} + \frac{1}{4} \sum_{i=1}^{k-1} \alpha^{1-i}). \end{aligned}$$

So it remains to show that

$$\alpha^{1-k} + \frac{1}{4} \sum_{i=1}^{k-1} \alpha^{1-i} \leq 1.$$

For $k = 1$, the inequality is obviously true. For $k \geq 2$ and $\alpha^{-1} = \delta - \frac{1}{4} \leq \frac{3}{4}$, we have

$$\alpha^{1-k} + \frac{1}{4} \underbrace{\sum_{i=1}^{k-1} \alpha^{1-i}}_{\text{geom. progression}} \leq \left(\frac{3}{4}\right)^{k-1} + \frac{1}{4} \frac{1 - \left(\frac{3}{4}\right)^{k-1}}{1 - \frac{3}{4}} = \frac{1}{4} \frac{1}{1 - \frac{3}{4}} = 1.$$

With this we have shown (3.4). Let $1 < j \leq m$. Then there exists an index $k \leq j$ such that $\lambda_{j, \|\cdot\|_2}(\mathcal{L}) \leq \|\mathbf{b}_k\|_2$. Then with the following inequality we obtain (3.5):

$$\lambda_{j, \|\cdot\|_2}^2(\mathcal{L}) \leq \|\mathbf{b}_k\|_2^2 \leq \alpha^{j-1} \|\mathbf{b}_j^*\|_2^2.$$

From Corollary 3.7 we have that there exists an index $k \geq j$, such that $\lambda_{j, \|\cdot\|_2}^2 \geq \|\mathbf{b}_k^*\|_2^2$. Then it follows:

$$\begin{aligned} \lambda_{j, \|\cdot\|_2}^2 &\geq \|\mathbf{b}_k^*\|_2^2 \geq \alpha^{-k+j} \|\mathbf{b}_j^*\|_2^2 && \text{(from Proposition 3.11)} \\ &\geq \alpha^{-k+1} \|\mathbf{b}_j\|_2^2 && \text{(from (3.4) with } k = j\text{)} \\ &\geq \alpha^{-m+1} \|\mathbf{b}_j\|_2^2 && (k \leq m \text{ and } \alpha \geq 1). \end{aligned}$$

□

A.K. Lenstra, H.W. Lenstra and L. Lovász [LLL82] presented an L^3 lattice basis reduction algorithm for $\delta = \frac{3}{4}$. Algorithm 3.2 describes the L^3 algorithm for $\frac{1}{4} < \delta < 1$. In the beginning the L^3 algorithm initializes $\|\mathbf{b}_1^*\|_2^2 := \|\mathbf{b}_1\|_2^2$ and sets the stage index k to 2. The algorithm finishes when the stage index k reaches $m + 1$. At the beginning of every stage k it calculates $\|\mathbf{b}_k^*\|_2^2$ and the corresponding Gram-Schmidt coefficients $\mu_{k,j}$ for $j = 1, \dots, k - 1$. Then, if needed, it performs size-reduction of \mathbf{b}_k and updates the coefficients $\mu_{k,i}$ for $1 \leq i \leq k - 1$. After that it checks the condition

$$\delta \|\mathbf{b}_{k-1}^*\|_2^2 > \|\mathbf{b}_k^*\|_2^2 + \mu_{k,k-1}^2 \|\mathbf{b}_{k-1}^*\|_2^2.$$

If it is fulfilled, k is set to $k - 1$ and the previous steps are repeated. If the condition is not fulfilled, then exchanging the vectors will not shorten the length of $\|\mathbf{b}_k^*\|_2$ any more, and the stage index is set to $k + 1$. Therefore upon entry of stage k the basis $\mathbf{b}_1, \dots, \mathbf{b}_{k-1}$ is L^3 reduced with δ , since the conditions

$$\begin{aligned} |\mu_{i,j}| &< \frac{1}{2} && \text{for } 1 \leq j < i \leq k - 1 \\ \delta \|\mathbf{b}_{i-1}^*\|_2^2 &\leq \|\mathbf{b}_i^*\|_2^2 + \mu_{i,i-1}^2 \|\mathbf{b}_{i-1}^*\|_2^2 && \text{for } 1 < i \leq k - 1 \end{aligned}$$

are fulfilled.

Algorithm 3.2 L^3 Basis Reduction Algorithm for the Euclidean Norm

INPUT: Lattice basis $(\mathbf{b}_1, \dots, \mathbf{b}_m) \in \mathbb{R}^n$ **OUTPUT:** L^3 -reduced lattice basis

```
 $\|\mathbf{b}_1^*\|_2^2 := \|\mathbf{b}_1\|_2^2$ 
 $k := 2$ 
while  $k \leq m$  do
  /* Compute  $\|\mathbf{b}_k^*\|_2^2$  and the Gram-Schmidt coefficients  $\mu_{k,j}$  for  $j = 1, \dots, k$  */
  for  $j = 1, \dots, k - 1$  do
     $\mu_{k,j} := \frac{\langle \mathbf{b}_k, \mathbf{b}_j^* \rangle - \sum_{i=1}^{j-1} \mu_{k,i} \mu_{j,i} \|\mathbf{b}_i^*\|_2^2}{\|\mathbf{b}_j^*\|_2^2}$ 
  end for
   $\mu_{k,k} := 1$ 
   $\|\mathbf{b}_k^*\|_2^2 := \langle \mathbf{b}_k, \mathbf{b}_k \rangle - \sum_{j=1}^{k-1} \mu_{k,j}^2 \|\mathbf{b}_j^*\|_2^2$ 

  /* Size-reduce  $\mathbf{b}_k$  and update  $\mu_{k,i}$  for  $i = 1, \dots, k - 1$  */
  for  $j = k - 1, \dots, 1$  do
    if  $|\mu_{k,j}| < \frac{1}{2}$  then
       $\mathbf{b}_k := \mathbf{b}_k - \lceil \mu_{k,j} \rceil \mathbf{b}_j$ 
      for  $i = 1, \dots, k - 1$  do
         $\mu_{k,i} := \mu_{k,i} - \lceil \mu_{k,j} \rceil \mu_{j,i}$ 
      end for
    end if
  end for

  if  $\delta \|\mathbf{b}_{k-1}^*\|_2^2 > \|\mathbf{b}_k^*\|_2^2 + \mu_{k,k-1}^2 \|\mathbf{b}_{k-1}^*\|_2^2$  then
     $\mathbf{b}_{k-1} \longleftrightarrow \mathbf{b}_k$ 
    if  $k = 2$  then
      update  $\|\mathbf{b}_1^*\|_2^2$ 
    end if
     $k := \max(k - 1, 2)$ 
  else
     $k := k + 1$ 
  end if

end while

return  $(\mathbf{b}_1, \dots, \mathbf{b}_m)$ 
```

From theoretical point of view the L^3 algorithm performs well, since it outputs relatively short vectors within polynomial time. In practice, however, the L^3 algorithm become slow as a result of long integer arithmetic, used in its subroutines. For speeding up the algorithm, it has been proposed to do the operations in floating point arithmetic. But then the algorithm becomes unstable and may result in changing the lattice. Therefore Schnorr and Euchner [SE94] have rewritten the original L^3 algorithm to minimize

floating point errors. They proposed a practical floating point variation (L³FP) of the L³ algorithm, in which the floating point precision arithmetic decreases the number of swaps and results in a faster algorithm. More details for this variation of the L³ algorithm can be found in [SE94].

Next we will analyse the runtime of Algorithm 3.2. Let

$$D_i := \det(\mathcal{L}(\mathbf{b}_1, \dots, \mathbf{b}_i))^2 = \prod_{j=1}^i \|\mathbf{b}_j^*\|_2^2 \quad \text{and} \quad D := \prod_{j=1}^{m-1} D_j$$

Theorem 3.12. [SE94] *Let $\mathbf{b}_1, \dots, \mathbf{b}_m \in \mathbb{Z}^n$ be a basis of a lattice \mathcal{L} . Let $M := \max_{i=1, \dots, m} \|\mathbf{b}_i^{\text{Start}}\|_2^2$. Then Algorithm 3.2 stops after maximum*

$$\binom{m}{2} \log_{\frac{1}{\delta}} M$$

swaps $\mathbf{b}_{k-1} \longleftrightarrow \mathbf{b}_k$ and

$$\mathcal{O}\left(m^2 n \left(1 + m \log_{\frac{1}{\delta}} M\right)\right)$$

arithmetic operations.

We follow the proof of Schnorr [SE94].

Proof: During the execution of the algorithm the values of D_j remain always positive integers for $j = 1, \dots, m$. In the beginning of the algorithm we have that

$$D^{\text{Start}} \leq M \binom{m}{2}. \tag{3.7}$$

First we show that after each swap $D^{\text{new}} \leq \delta \cdot D^{\text{old}}$ holds. When we swap \mathbf{b}_{k-1} and \mathbf{b}_k the lattices $\mathcal{L}(\mathbf{b}_1, \dots, \mathbf{b}_j)$ for $j \neq k-1$ and the determinants D_j remain unchanged. In the algorithm we swap \mathbf{b}_{k-1} and \mathbf{b}_k if we have

$$\delta \cdot \|\mathbf{b}_{k-1}^{\text{old}}\|_2^2 > \|\mathbf{b}_k^{\text{old}}\|_2^2 + (\mu_{k,k-1}^{\text{old}})^2 \cdot \|\mathbf{b}_{k-1}^{\text{old}}\|_2^2.$$

Since

$$(\mu_{k,k-1}^{\text{old}})^2 \cdot \|\mathbf{b}_{k-1}^{\text{old}}\|_2^2 = \|\mathbf{b}_{k-1}^{\text{new}}\|_2^2,$$

we get that $\delta \cdot \|\mathbf{b}_{k-1}^{\text{old}}\|_2^2 > \|\mathbf{b}_{k-1}^{\text{new}}\|_2^2$. Therefore from

$$\frac{D^{\text{new}}}{D^{\text{old}}} = \frac{D_{k-1}^{\text{new}}}{D_{k-1}^{\text{old}}} = \frac{\|\mathbf{b}_{k-1}^{\text{new}}\|_2^2}{\|\mathbf{b}_{k-1}^{\text{old}}\|_2^2} \leq \delta$$

follows $D^{\text{new}} \leq \delta D^{\text{old}}$.

$D^{\text{end}} \in \mathbb{N}$ implies

$$D^{\text{start}} \geq D^{\text{end}} \cdot \left(\frac{1}{\delta}\right)^{\#\text{Swaps}} \geq \left(\frac{1}{\delta}\right)^{\#\text{Swaps}}$$

Then from (3.7) follows that

$$\#\text{Swaps} \leq \binom{m}{2} \log_{\frac{1}{\delta}} M = \mathcal{O}(m^2 \log_2 M).$$

In the beginning of the algorithm $k = 2$ and at the end $k = m + 1$. Therefore

$$\#\text{Iterations} \leq m - 1 + 2 \cdot \#\text{Swaps}.$$

Every iteration of the while-loop requires $\mathcal{O}(mn)$. Then the number of the arithmetic operations in the whole algorithm is at most $\mathcal{O}\left(m^2 n \left(1 + m \log_{\frac{1}{\delta}} M\right)\right)$. \square

3.3 The Block Basis Reduction Algorithm

In the following section we present the concept of block reduced bases and give an algorithm for block basis reduction.

Definition 3.13. *Let $\beta \geq 2$ be an integer and $\delta \in (0, 1]$ be a real number. We call a basis $\mathbf{b}_1, \dots, \mathbf{b}_m$ of a lattice $\mathcal{L} \subset \mathbb{R}^n$ β -block reduced with δ or (β, δ) -reduced (w.r.t. $\|\cdot\|_2$) if for $i = 1, \dots, m$*

1. $|\mu_{i,j}| \leq \frac{1}{2}$ for all $j < i$,
2. $\delta \|\mathbf{b}_i^*\|_2^2 \leq \lambda_{1,\|\cdot\|_2}^2(\mathcal{L}(\pi_i(\mathbf{b}_i), \dots, \pi_i(\mathbf{b}_{\min(i+\beta-1, m)})))$.

In order to estimate the quality of the approximation of the (β, δ) -reduced bases, we define first the Hermite constants. The *Hermite constant* γ_β is defined as:

$$\gamma_\beta = \sup\{\lambda_{1,\|\cdot\|_2}^2(\mathcal{L}) \det(\mathcal{L})^{-\frac{2}{\beta}} : \mathcal{L} \text{ is a lattice of rank } \beta\}.$$

Its exact value is known for $\beta \leq 8$. The following upper bound for the Hermite constant was proved by Blichfeld [Bli14]:

$$\gamma_\beta \leq \frac{2}{\pi} \Gamma\left(2 + \frac{\beta}{2}\right)^{\frac{2}{\beta}}.$$

It is also known that for $\beta \rightarrow \infty$ we have the following asymptotic bounds for γ_β :

$$\frac{\beta}{2\pi e}(1 + o(1)) \leq \gamma_\beta \leq \frac{\beta}{1.14\pi e}(1 + o(1)),$$

where the lower bound follows from the Minkowski-Hlawka theorem [Cas71] and the upper bound is due to Kabatyansky and Levenshtein [KL78].

The (β, δ) -reduced bases approximate the successive minima up to an exponential factor (in m/β). The approximation becomes better with increased block size β and bigger δ . If β is a fixed fraction of m , then the successive minima will be approximated up to a polynomial in m factor.

Lemma 3.14. [Sch94] Let $\mathbf{b}_1, \dots, \mathbf{b}_m$ be a (β, δ) -reduced basis of a lattice $\mathcal{L} \subset \mathbb{R}^n$ with $2 \leq \beta \leq m$ and $0 < \delta \leq 1$. Let $M := \max\{\|\mathbf{b}_{m-\beta+2}^*\|_2, \dots, \|\mathbf{b}_m^*\|_2\}$. Then

$$\|\mathbf{b}_1\|_2 \leq \left(\frac{\gamma_\beta}{\delta}\right)^{\frac{m-1}{\beta-1}} M.$$

Proof [Sch94]: We expand the basis $\mathbf{b}_1, \dots, \mathbf{b}_m$ with $\beta-2$ linearly independent vectors to

$$\mathbf{b}_{-\beta+3}, \dots, \mathbf{b}_{-1}, \mathbf{b}_0, \mathbf{b}_1, \dots, \mathbf{b}_m, \quad (3.8)$$

so that the following conditions are fulfilled

1. $\|\mathbf{b}_i\|_2 = \|\mathbf{b}_1\|_2$ for $i \leq 0$,
2. $\langle \mathbf{b}_i, \mathbf{b}_j \rangle = 0$ for $i \leq 0$, $i \neq j$ and $j = -\beta+3, \dots, m-\beta+1$.

For this purpose we embed the given basis in $\mathbb{R}^{n+\beta-2}$ and we set for $\mathbf{b}_{-\beta+3}, \dots, \mathbf{b}_{-1}, \mathbf{b}_0$ the $\|\mathbf{b}_1\|_2$ multiples of the unit vectors in the additional directions. With this we have that the basis (3.8) is (β, δ) -reduced basis. This means that for $i = -\beta+3, \dots, m-\beta+1$ we have

$$\begin{aligned} \delta \|\mathbf{b}_i^*\|_2^2 &\leq \lambda_{1, \|\cdot\|_2}^2(\mathcal{L}(\pi_i(\mathbf{b}_i), \dots, \pi_i(\mathbf{b}_{i+\beta-1}))) \\ &\leq \gamma_\beta \det(\mathcal{L}(\pi_i(\mathbf{b}_i), \dots, \pi_i(\mathbf{b}_{i+\beta-1}))). \end{aligned}$$

So we get

$$\delta^{\frac{\beta}{2}} \|\mathbf{b}_i^*\|_2^\beta \leq \gamma_\beta^{\frac{\beta}{2}} \|\mathbf{b}_i^*\|_2 \|\mathbf{b}_{i+1}^*\|_2 \cdots \|\mathbf{b}_{i+\beta-1}^*\|_2$$

for $i = -\beta+3, \dots, m-\beta+1$. Then we multiply these $m-1$ inequalities and get

$$\begin{aligned} \delta^{\frac{(m-1)\beta}{2}} \|\mathbf{b}_{-\beta+3}^*\|_2^\beta \|\mathbf{b}_{-\beta+4}^*\|_2^\beta \cdots \|\mathbf{b}_{m-\beta+1}^*\|_2^\beta \\ \leq \gamma_\beta^{\frac{(m-1)\beta}{2}} \|\mathbf{b}_{-\beta+3}^*\|_2^1 \|\mathbf{b}_{-\beta+4}^*\|_2^2 \cdots \|\mathbf{b}_1^*\|_2^{\beta-1} \|\mathbf{b}_2^*\|_2^\beta \cdots \\ \|\mathbf{b}_{m-\beta+1}^*\|_2^\beta \|\mathbf{b}_{m-\beta+2}^*\|_2^{\beta-1} \cdots \|\mathbf{b}_{m-1}^*\|_2^2 \|\mathbf{b}_m^*\|_2^1. \end{aligned}$$

This implies

$$\delta^{\frac{(m-1)\beta}{2}} \|\mathbf{b}_{-\beta+3}^*\|_2^{\beta-1} \|\mathbf{b}_{-\beta+4}^*\|_2^{\beta-2} \cdots \|\mathbf{b}_0^*\|_2^2 \|\mathbf{b}_1^*\|_2^1 \leq \gamma_\beta^{\frac{(m-1)\beta}{2}} \|\mathbf{b}_{m-\beta+2}^*\|_2^{\beta-1} \cdots \|\mathbf{b}_{m-1}^*\|_2^2 \|\mathbf{b}_m^*\|_2^1.$$

Then from $\|\mathbf{b}_i\|_2 = \|\mathbf{b}_i^*\|_2 = \|\mathbf{b}_1\|_2$ for $i \leq 0$ follows that

$$\delta^{\frac{(m-1)\beta}{2}} \|\mathbf{b}_1\|_2^{\binom{\beta}{2}} \leq \gamma_\beta^{\frac{(m-1)\beta}{2}} M^{\binom{\beta}{2}}.$$

So finally we get:

$$\|\mathbf{b}_1\|_2 \leq \left(\frac{\gamma_\beta}{\delta}\right)^{\frac{m-1}{\beta-1}} M.$$

□

Theorem 3.15. [Sch94] *Let $\mathbf{b}_1, \dots, \mathbf{b}_m$ be a (β, δ) -reduced basis of a lattice $\mathcal{L} \subset \mathbb{R}^n$ with $2 \leq \beta \leq m$ and $0 < \delta \leq 1$. Then for $i = 1, \dots, m$ holds:*

$$\frac{\|\mathbf{b}_i^*\|_2^2}{\lambda_{i, \|\cdot\|_2}^2(\mathcal{L})} \leq \frac{1}{\delta} \left(\frac{\gamma_\beta}{\delta}\right)^{2 \frac{m-i}{\beta-1}}, \quad (3.9)$$

$$\frac{\|\mathbf{b}_i\|_2^2}{\lambda_{i, \|\cdot\|_2}^2(\mathcal{L})} \leq \frac{i+3}{4\delta} \left(\frac{\gamma_\beta}{\delta}\right)^{2 \frac{m-1}{\beta-1}}, \quad (3.10)$$

$$\frac{\lambda_{i, \|\cdot\|_2}^2(\mathcal{L})}{\|\mathbf{b}_i\|_2^2} \leq \frac{1}{\delta} \left(\frac{\gamma_\beta}{\delta}\right)^{2 \frac{i-1}{\beta-1}}. \quad (3.11)$$

(3.10) implies that the approximation factor of the upper bound of the Euclidean length of the reduced basis vector \mathbf{b}_i worsens linearly with increasing i . (3.11) implies that the approximation factor of the lower bound of the Euclidean length of the reduced basis vector \mathbf{b}_i worsens exponentially with increasing i .

Proof [Sch94]:

(3.9):

Case $i = 1$:

For $i = 1$ we will prove (3.9) by induction on m .

- Let $m = \beta$. Then by Definition 3.13 we have

$$\sqrt{\delta} \|\mathbf{b}_1\|_2 \leq \lambda_{1, \|\cdot\|_2}(\mathcal{L})$$

and (3.9) follows directly from $\frac{\gamma_\beta}{\delta} \geq 1$.

- Let $m > \beta$ and \mathbf{v} be a shortest lattice vector in $\mathcal{L}(B)$. Here we consider two cases:

- $\mathbf{v} \in \mathcal{L}(\mathbf{b}_1, \dots, \mathbf{b}_{m-1})$.

In this case we can reduce m by 1 and (3.9) follows from the induction hypothesis.

- $\mathbf{v} \notin \mathcal{L}(\mathbf{b}_1, \dots, \mathbf{b}_{m-1})$.

Since $\pi_i(\mathbf{v}) \neq \mathbf{0}$ we have

$$\lambda_{1, \|\cdot\|_2}(\mathcal{L}) = \|\mathbf{v}\|_2 \geq \|\pi_j(\mathbf{v})\|_2 \geq \lambda_{1, \|\cdot\|_2}(\mathcal{L}(\pi_j(\mathbf{b}_j), \dots, \pi_j(\mathbf{b}_m))) \geq \sqrt{\delta} \|\mathbf{b}_j^*\|_2$$

for $j = m - \beta + 1, \dots, m$. From this we get

$$\begin{aligned} \lambda_{1, \|\cdot\|_2}(\mathcal{L}) &\geq \sqrt{\delta} \max \{ \|\mathbf{b}_j^*\|_2 : j = m - \beta + 1, \dots, m \} \\ &\geq \sqrt{\delta} \max \{ \|\mathbf{b}_j^*\|_2 : j = m - \beta + 2, \dots, m \} \end{aligned}$$

and (3.9) follows from Lemma 3.14.

Case $i > 1$: For $i > 1$ we have that $\mathcal{L}_i = \mathcal{L}(\pi_i(\mathbf{b}_i), \dots, \pi_i(\mathbf{b}_m))$ is also (β, δ) -reduced and therefore we get

$$\|\mathbf{b}_i^*\|_2 \leq \frac{1}{\sqrt{\delta}} \left(\frac{\gamma_\beta}{\delta}\right)^{\frac{m-i}{\beta-1}} \lambda_{1, \|\cdot\|_2}(\mathcal{L}_i). \quad (3.12)$$

In the lattice \mathcal{L} there exist i linear independent vectors $\mathbf{a}_1, \dots, \mathbf{a}_i$ with length not greater than $\lambda_{i, \|\cdot\|_2}(\mathcal{L})$. Therefore there exists at least one $j < i$ with $\|\pi_i(\mathbf{a}_j)\|_2 \neq 0$. So we get

$$\lambda_{1, \|\cdot\|_2}(\mathcal{L}_i) \leq \|\pi_i(\mathbf{a}_j)\|_2 \leq \lambda_{i, \|\cdot\|_2}(\mathcal{L}).$$

Then (3.9) follows directly from (3.12).

(3.10): Every (β, δ) -reduced basis is also size-reduced. So we get:

$$\begin{aligned} \|\mathbf{b}_i\|_2^2 &\stackrel{(3.2)}{=} \|\mathbf{b}_i^*\|_2^2 + \sum_{j=1}^{i-1} \mu_{i,j}^2 \|\mathbf{b}_j^*\|_2^2 \leq \|\mathbf{b}_i^*\|_2^2 + \frac{1}{4} \sum_{j=1}^{i-1} \|\mathbf{b}_j^*\|_2^2 \\ &\stackrel{(3.9)}{\leq} \frac{1}{\delta} \left(\frac{\gamma\beta}{\delta}\right)^{2\frac{m-i}{\beta-1}} \lambda_{i, \|\cdot\|_2}^2(\mathcal{L}) + \frac{1}{4} \sum_{j=1}^{i-1} \frac{1}{\delta} \left(\frac{\gamma\beta}{\delta}\right)^{2\frac{m-j}{\beta-1}} \lambda_{j, \|\cdot\|_2}^2(\mathcal{L}) \\ &\leq \frac{1}{\delta} \left(\frac{\gamma\beta}{\delta}\right)^{2\frac{m-i}{\beta-1}} \left(\left(\frac{\gamma\beta}{\delta}\right)^{2\frac{-i}{\beta-1}} + \frac{1}{4} \sum_{j=1}^{i-1} \left(\frac{\gamma\beta}{\delta}\right)^{2\frac{-j}{\beta-1}} \right) \lambda_{i, \|\cdot\|_2}^2(\mathcal{L}). \end{aligned}$$

We approximate the terms of the sum from above with $\frac{\gamma\beta}{\delta} 2^{\frac{-1}{\beta-1}}$ and we get

$$\|\mathbf{b}_i\|_2^2 \leq \left(\frac{\gamma\beta}{\delta}\right)^{2\frac{m-i}{\beta-1}} \left(\frac{\gamma\beta}{\delta}\right)^{2\frac{-1}{\beta-1}} \left(1 + \frac{i-1}{4}\right) \lambda_{i, \|\cdot\|_2}^2(\mathcal{L}) \leq \frac{i+3}{4\delta} \left(\frac{\gamma\beta}{\delta}\right)^{2\frac{m-1}{\beta-1}} \lambda_{i, \|\cdot\|_2}^2(\mathcal{L}).$$

(3.11): By the definition of successive minima we have

$$\lambda_{i, \|\cdot\|_2}^2(\mathcal{L}) \leq \max_{j=1, \dots, i} \|\mathbf{b}_j\|_2^2.$$

From $\|\mathbf{b}_j\|_2^2 = \|\mathbf{b}_j^*\|_2^2 + \sum_{k=1}^{j-1} \mu_{j,k}^2 \|\mathbf{b}_k^*\|_2^2$ and $\mu_{j,k}^2 \leq \frac{1}{4}$ follows that

$$\lambda_{i, \|\cdot\|_2}^2(\mathcal{L}) \leq \frac{i+3}{4} \max_{j=1, \dots, i} \|\mathbf{b}_j^*\|_2^2. \quad (3.13)$$

Applying Lemma 3.14 to the (β, δ) -reduced basis $\pi_j(\mathbf{b}_j), \dots, \pi_j(\mathbf{b}_i)$ gives

$$\|\mathbf{b}_j^*\|_2^2 \leq \left(\frac{\gamma\beta}{\delta}\right)^{2\frac{i-j}{\beta-1}} \max_{k=i-\beta+2, \dots, i} \|\mathbf{b}_k^*\|_2^2 \quad (1 \leq j \leq i - \beta + 1). \quad (3.14)$$

On the other hand by the definition of (β, δ) -reduced basis we have

$$\delta \|\mathbf{b}_k^*\|_2^2 \leq \lambda_{1, \|\cdot\|_2}^2(\mathcal{L}(\pi_k(\mathbf{b}_k), \dots, \pi_k(\mathbf{b}_i))) \leq \|\pi_k(\mathbf{b}_i)\|_2^2 \leq \|\mathbf{b}_i\|_2^2 \quad (3.15)$$

for $i - \beta + 2 \leq k \leq i$. From (3.14) and (3.15) follows that

$$\|\mathbf{b}_j^*\|_2^2 \leq \frac{1}{\delta} \left(\frac{\gamma\beta}{\delta}\right)^{2\frac{i-j}{\beta-1}} \|\mathbf{b}_i\|_2^2 \quad (1 \leq j \leq i).$$

Now (3.11) follows from (3.13). □

The following theorem shows that the L^3 reduction is a special case of the (β, δ) -reduction with $\beta = 2$.

Theorem 3.16. [SE94] *A basis $\mathbf{b}_1, \dots, \mathbf{b}_m$ is $(2, \delta)$ -reduced for $\frac{1}{3} \leq \delta \leq 1$ if and only if it is L^3 reduced with δ .*

Next we describe the block basis reduction algorithm. Let $c_j := \|\mathbf{b}_j^*\|_2^2$. The block basis reduction algorithm uses the L^3 algorithm (Algorithm 3.2) and an enumeration subroutine $ENUM(j, k)$, which finds an integer non-zero vector (u_j, \dots, u_k) , such that (u_j, \dots, u_k) minimizes the expression

$$c_j(\tilde{u}_j, \dots, \tilde{u}_k) := \left\| \pi_j \left(\sum_{i=j}^k \tilde{u}_i \mathbf{b}_i \right) \right\|_2^2 = \sum_{s=j}^k \left(\sum_{i=s}^k \tilde{u}_i \mu_{i,s} \right)^2 c_s. \quad (3.16)$$

We will denote with \bar{c}_j the minimal value of $c_j(\tilde{u}_j, \dots, \tilde{u}_k)$ and with $\mathbf{b}_j^{\text{new}}$ the vector $\sum_{i=j}^k u_i \mathbf{b}_i$ corresponding to \bar{c}_j .

Algorithm 3.3 Block Basis Reduction Algorithm for the Euclidean Norm

INPUT: $(\mathbf{b}_1, \dots, \mathbf{b}_m) \in \mathbb{Z}^n$, $\frac{1}{2} < \delta < 1$, $2 < \beta \leq m$

OUTPUT: (β, δ) -reduced basis $(\mathbf{b}_1, \dots, \mathbf{b}_m)$

```

z := 0
j := m - 1
L3((b1, ..., bm), δ)

while z < m - 1 do
  j := j + 1
  if j = m then
    j := 1
  end if
  k := min{j + β - 1, m}

  (c̄j, (uj, ..., uk), bjnew) := ENUM(j, k)
  h := min{k + 1, m}
  if c̄j < δcj then
    BASIS((uj, ..., uk), (b1, ..., bk))
    L3((b1, ..., bh), δ)
    z := 0
  else
    L3((b1, ..., bh), δ)
    z := z + 1
  end if
end while

return (b1, ..., bm)

```

Throughout the algorithm the integer j is cyclically shifted through the integers $1, 2, \dots, m-1$. The variable z counts the number of positions j that satisfy the inequality

$$\delta \|\mathbf{b}_j^*\|_2^2 \leq \lambda_{1, \|\cdot\|_2}^2(L(\pi_j(\mathbf{b}_j), \dots, \pi_j(\mathbf{b}_k))). \quad (3.17)$$

If this inequality does not hold for index j , then we extend $\mathbf{b}_1, \dots, \mathbf{b}_{j-1}, \mathbf{b}_j^{\text{new}}$ to a basis $\mathbf{b}_1, \dots, \mathbf{b}_{j-1}, \mathbf{b}_j^{\text{new}}, \mathbf{b}'_{j+1}, \dots, \mathbf{b}'_k, \dots, \mathbf{b}_m$ of the lattice \mathcal{L} , using the subroutine BASIS, defined below. Then we call the L^3 algorithm, and reset z to 0. If (3.17) holds, then we call the L^3 algorithm for $\mathbf{b}_1, \dots, \mathbf{b}_{\min\{k+1, m\}}$ and we increase the index z .

According to Definition 3.13, a basis $\mathbf{b}_1, \dots, \mathbf{b}_m$ is (β, δ) -reduced, if it is size-reduced and (3.17) holds for $j = 1, \dots, m$. Since for $j = m$ (3.17) is obviously true, then the basis $\mathbf{b}_1, \dots, \mathbf{b}_m$ is (β, δ) -reduced, if it is size-reduced and $z = m - 1$. The calls of the L^3 algorithm at the end of each iteration guarantee, that the output basis will be size-reduced. Since for every j $c_j \geq 0$, the value of \bar{c}_j cannot endlessly decrease. This guarantees that the block basis algorithm terminates after finite number of iterations. Since it terminates when $z = m - 1$, the basis $\mathbf{b}_1, \dots, \mathbf{b}_m$ will be (β, δ) -reduced.

Algorithm 3.4 BASIS - Basis Transformation

INPUT: $(u_j, \dots, u_k) \in \mathbb{Z}^{k-j+1} \setminus \{\mathbf{0}\}$, $(\mathbf{b}_1, \dots, \mathbf{b}_k)$

```

 $\mathbf{b}_j^{\text{new}} = \sum_{i=j}^k u_i \mathbf{b}_i$ 
 $g := \max\{t : j \leq t \leq k, u_t \neq 0\}$ 
while  $|u_g| > 1$  do
   $i := \max\{t : j \leq t \leq g, u_t \neq 0\}$  (such  $i$  always exists since  $\gcd(u_j, \dots, u_k) = 1$ )
   $q = \lceil u_g / u_i \rceil$ 
   $u := u_i$ 
   $u_i := u_g - q \cdot u_i$ 
   $u_g := u$ 
   $\mathbf{b} := \mathbf{b}_g$ 
   $b_g := q \cdot b_g + b_i$ 
   $b_i := b$ 
end while
for  $i = g, \dots, j + 1$  do
   $\mathbf{b}_i := \mathbf{b}_{i-1}$ 
end for
 $\mathbf{b}_j := \mathbf{b}_j^{\text{new}}$ 

```

In the subroutine BASIS we extend $\mathbf{b}_1, \dots, \mathbf{b}_{j-1}, \mathbf{b}_j^{\text{new}} = \sum_{i=j}^k u_i \mathbf{b}_i$ to a basis of \mathcal{L} . Therefore we distinguish two cases. Let $g := \max\{i : j \leq i \leq k, u_i \neq 0\}$.

1. $|u_g| = 1$: Then $\mathbf{b}_1, \dots, \mathbf{b}_{j-1}, \mathbf{b}_j^{\text{new}}, \mathbf{b}_j, \dots, \mathbf{b}_{g-1}, \mathbf{b}_{g+1}, \dots, \mathbf{b}_m$ is a basis of \mathcal{L} , i.e. we can remove the vector \mathbf{b}_g and insert the vector $\mathbf{b}_j^{\text{new}}$ at position j .
2. $|u_g| > 1$: In this case we transform the basis $\mathbf{b}_1, \dots, \mathbf{b}_m$ into $\mathbf{b}_1, \dots, \mathbf{b}_{j-1}, \mathbf{b}'_j, \dots, \mathbf{b}'_g, \mathbf{b}_{g+1}, \dots, \mathbf{b}_m$, so that $\mathbf{b}_j^{\text{new}} = \sum_{i=j}^g u'_i \mathbf{b}'_i$ with $|u'_g| = 1$. This is always possible, since ENUM(j, k) outputs a vector (u_j, \dots, u_k) with $\gcd(u_j, \dots, u_k) = 1$. Otherwise the vector

$$\mathbf{b} := \frac{\mathbf{b}_j^{\text{new}}}{\gcd(u_j, \dots, u_k)}$$

would be a vector in \mathcal{L} , such that

$$\|\pi_j(\mathbf{b})\|_2^2 = \frac{\bar{c}_j}{\gcd(u_j, \dots, u_k)^2}$$

and therefore we would have $\bar{c}_j > \lambda_{1, \|\cdot\|_2}^2(\mathcal{L}(\pi_j(\mathbf{b}_j), \dots, \pi_j(\mathbf{b}_k)))$. After this transformation we have $|u_g| = 1$, so we can remove \mathbf{b}'_g and insert $\mathbf{b}_j^{\text{new}}$ at position j .

Next we present the enumeration subroutine $ENUM(j, k)$, which finds an integer non-zero vector (u_j, \dots, u_k) , such that (u_j, \dots, u_k) minimizes the expression

$$c_j(\tilde{u}_j, \dots, \tilde{u}_k) := \left\| \pi_j \left(\sum_{i=j}^k \tilde{u}_i \mathbf{b}_i \right) \right\|_2^2 = \sum_{s=j}^k \left(\sum_{i=s}^k \tilde{u}_i \mu_{i,s} \right)^2 c_s. \quad (3.18)$$

The algorithm $ENUM$ enumerates in Depth-First-Order all integer vectors $(\tilde{u}_t, \dots, \tilde{u}_k)$ that satisfy $c_t(\tilde{u}_t, \dots, \tilde{u}_k) < \bar{c}_j$ for $t = j, \dots, k$, where \bar{c}_j is the current minimum for the function (3.18) and the current minimal place is (u_j, \dots, u_k) . Throughout the enumeration we always have that $\tilde{c}_t = c_t(\tilde{u}_t, \dots, \tilde{u}_k)$ for the current vector $(\tilde{u}_t, \dots, \tilde{u}_k)$. More precisely,

$$\begin{aligned} \tilde{c}_t = c_t(\tilde{u}_t, \dots, \tilde{u}_k) &= \sum_{s=t}^k \left(\sum_{i=s}^k \tilde{u}_i \mu_{i,s} \right)^2 c_s = \left(\sum_{i=t}^k \tilde{u}_i \mu_{i,t} \right)^2 c_t + \underbrace{\sum_{s=t+1}^k \left(\sum_{i=s}^k \tilde{u}_i \mu_{i,s} \right)^2 c_s}_{\tilde{c}_{t+1}} \\ &= \left(\tilde{u}_t + \sum_{i=t+1}^k \tilde{u}_i \mu_{i,t} \right)^2 c_t + \tilde{c}_{t+1}. \end{aligned}$$

The uniquely determined real minimal place of $c_t(\tilde{u}_t, \dots, \tilde{u}_k)$ for fixed $(\tilde{u}_{t+1}, \dots, \tilde{u}_k)$ is $-y_t$, where $y_t := \sum_{i=t+1}^k \tilde{u}_i \mu_{i,t}$. So we can get the integer minimal place at $v_t := \lceil -y_t \rceil$ due to

$$c_t(-y_t + x, \tilde{u}_{t+1}, \dots, \tilde{u}_k) = c_t(-y_t - x, \tilde{u}_{t+1}, \dots, \tilde{u}_k).$$

The index s denotes the maximal previous value for t . When initially we arrive at level t from level $t-1$ we have $\Delta_t = 0$ and $s = t$. Then we set $\Delta_t = 1$ and $\tilde{u}_t = 1$. So the vector $(\tilde{u}_j, \dots, \tilde{u}_k)$ fulfills the following condition: the largest index i , for which $\tilde{u}_i \neq 0$ satisfies $\tilde{u}_i > 0$. When level t is reached from level $t+1$ we set $\Delta_t = 0$ and we assign to δ_t the sign of $y_t - \lceil -y_t \rceil$. When subsequently level t is reached from level $t-1$ we take for Δ_t the next value in either the order $1, -1, 2, -2, 3, -3, \dots$ or in the order $-1, 1, -2, 2, -3, 3, \dots$ and correspondingly for \tilde{u}_t the next value in either the order $v_t, v_t + 1, v_t - 1, v_t + 2, v_t - 2, v_t + 3, v_t - 3, \dots$ or in the order $v_t, v_t - 1, v_t + 1, v_t - 2, v_t + 2, v_t - 3, v_t + 3, \dots$ as long as $\tilde{c}_t \geq c_j$. The choice of the order depends on δ_t . At this point t is incremented to $t+1$.

Algorithm 3.5 ENUM - Enumeration Subroutine for the Block Basis Reduction Algorithm for the Euclidean Norm

INPUT: j, k ($1 \leq j \leq k \leq m$), $c_i = \|\mathbf{b}_i'\|_2^2$ ($i = j, \dots, k$), $\mu_{i,t}$ ($j \leq t < i \leq k$)

OUTPUT: • the minimal place (u_j, \dots, u_k) of $c_j(\tilde{u}_j, \dots, \tilde{u}_k)$
 • the minimum \bar{c}_j of $c_j(\tilde{u}_j, \dots, \tilde{u}_k)$
 • $\mathbf{b}_j^{\text{new}} = \sum_{i=j}^k u_i \mathbf{b}_i$

/ Initialization: */*

$\bar{c}_j := \delta c_j$

$\tilde{u}_j := u_j := 1, y_j := \Delta_j := 0, s := t := j$

$\delta_j := 1$

for $i = j + 1, \dots, k + 1$ **do**

$\tilde{c}_i := u_i := \tilde{u}_i := y_i := \Delta_i := 0, \delta_i := 1$

end for

while $t \leq k$ **do**

/ Calculate $c_t(\tilde{u}_t, \dots, \tilde{u}_k)$ for the current stage t */*

$\tilde{c}_t := \tilde{c}_t + 1 + (y_t + \tilde{u}_t)^2 c_t$

if $\tilde{c}_t < \bar{c}_j$ **then**

if $t > j$ **then**

/ Decrease t by 1 and set \tilde{u}_t such that $c_t(\tilde{u}_t, \dots, \tilde{u}_k)$ for fixed $(u, \tilde{u}_{t+1}, \dots, \tilde{u}_k)$ is minimal. */*

$t := t - 1, y_t := \sum_{i=t+1}^s \tilde{u}_i \mu_{i,t}, \tilde{u}_t := v_t := \lceil -y_t \rceil, \Delta_t := 0$

if $\tilde{u}_j > -y_t$ **then**

$\delta_t := -1$

else

$\delta_t := 1$

end if

else

$\bar{c}_j := \tilde{c}_j$ */* If $t = j$ we have found a shorter vector */*

for $i = j, \dots, k$ **do**

$u_i := \tilde{u}_i$

end for

end if

else

/ The function $c_t(\tilde{u}_t, \dots, \tilde{u}_k)$ is monoton increasing and its minimum w.r.t to u_t is not smaller than \bar{c}_j . Therefore we can stop the enumeration of the vectors $(u, \tilde{u}_{t+1}, \dots, \tilde{u}_k)$ and increase t by 1 */*

$t := t + 1, s := \max(s, t)$

if $t < s$ **then**

$\Delta_t := -\Delta_t$

end if

if $\Delta_t \delta_t \geq 0$ **then**

$\Delta_t := \Delta_t + \delta_t$

end if

$\tilde{u}_t := v_t + \Delta_t$

end if

end while

return $\bar{c}_j, (u_j, \dots, u_k), \mathbf{b}_j^{\text{new}} = \sum_{i=j}^k u_i \mathbf{b}_i$

In the algorithm ENUM the initialization step will be executed exactly once and it needs $O(k - j + 1)$ arithmetic operations. For each iteration of the while-loop ENUM executes $O(k - j + 1) = O(n\beta)$ arithmetic operations. With the following theorem we give an upper bound of the iterations of ENUM in the while-loop.

Theorem 3.17. Proof [Rit97]: Let $\delta \in (\frac{1}{4}, 1]$. For block size $\beta = k - j + 1$ and b_j, \dots, b_k - an L^3 -reduced basis with δ , the algorithm ENUM makes in the while-loop at most $(1/(\delta - \frac{1}{4}))^{O(\beta^2)}$ iterations. The number of arithmetic operations is at most $O(\beta)(1/(\delta - \frac{1}{4}))^{O(\beta^2)}$

Proof [Rit97]: Let $A(t, x)$ be the number of enumerated vectors $\tilde{u}_t, \dots, \tilde{u}_k$ with $\tilde{c}_t < x$. The number of iterations in Step 2 A is bounded from

$$A \leq (k - j + 1) + A(j, c_j) + 2 \sum_{t=j+1}^k A(t, c_j). \quad (3.19)$$

Let for fixed $(\tilde{u}_{t+1}, \dots, \tilde{u}_k)$, $\tilde{\mathbf{b}}_{t+1} := \sum_{i=t+1}^k \tilde{u}_i \mathbf{b}_i$ and $\tilde{c}_{t+1} := \|\pi_{t+1}(\mathbf{b}_{t+1})\|_2^2 < c_j$. Then for all $\tilde{u}_t \in \mathbb{Z}$ with $\tilde{c}_t := \|\pi_t(\mathbf{b}_{t+1} + \tilde{u}_t \mathbf{b}_t)\|_2^2 < c_j$ holds

$$c_j > \tilde{c}_t = \tilde{c}_{t+1} + \left(\tilde{u}_t + \sum_{i=t+1}^s \tilde{u}_i \mu_{i,t} \right)^2 c_t \geq \left(\tilde{u}_t + \sum_{i=t+1}^s \tilde{u}_i \mu_{i,t} \right)^2 c_t$$

and we get

$$|\tilde{u}_t + \sum_{i=t+1}^s \tilde{u}_i \mu_{i,t}| < \sqrt{\frac{c_j}{c_t}}.$$

So we obtain

$$A(t, c_j) < \prod_{i=t}^k \left(2\sqrt{\frac{c_j}{c_i}} + 1 \right). \quad (3.20)$$

Since $\mathbf{b}_j, \dots, \mathbf{b}_k$ is L^3 -reduced with δ , we have that

$$\frac{c_j}{c_t} \geq \left(\delta - \frac{1}{4} \right)^{j-t}.$$

Applying this to (3.20) gives

$$\begin{aligned} A(t, c_j) &< \prod_{i=t}^k \left(2 \left(\delta - \frac{1}{4} \right)^{(j-i)/2} + 1 \right), \\ &< 3^{k-t+1} \left(\delta - \frac{1}{4} \right)^{\frac{1}{2} \sum_{i=t}^k (j-i)}, \\ &= 3^{k-t+1} \left(\delta - \frac{1}{4} \right)^{\frac{1}{2}(k-t+1)(j-(k+t)/2)} = \left(\delta - \frac{1}{4} \right)^{O(\beta^2)}. \end{aligned}$$

The theorem follows from (3.19). □

Now we analyze the running time of the block basis reduction. We follow the analysis, presented by Ritter [Rit97].

Let

$$D_i := \det(\mathcal{L}(\mathbf{b}_1, \dots, \mathbf{b}_i))^2 = \prod_{j=1}^i \|\mathbf{b}_j^*\|_2^2 \quad \text{and} \quad D := \prod_{j=1}^{m-1} D_j$$

and $\mathbf{b}_j^{\text{new}} \in \mathcal{L}(\mathbf{b}_j, \dots, \mathbf{b}_{\min\{j+\beta-1, m\}})$ be a vector, shorter than \mathbf{b}_j , found from the ENUM subroutine. Using the BASIS subroutine, $\mathbf{b}_j^{\text{new}}$ is inserted in the basis $\mathbf{b}_1, \dots, \mathbf{b}_{\min\{j+\beta-1, m\}}$. After this we have $D_j^{\text{new}} < \delta D_j$, but the values of $D_{j+1}, \dots, D_{\min\{j+\beta-1, m\}}$ are also changed. So we cannot be sure, that the value of D decreases with δ for every insertion of such vector $\mathbf{b}_j^{\text{new}}$. Therefore, on the heuristic assumption, that the value of D decreases at least with δ for every insertion of a vector $\mathbf{b}_j^{\text{new}}$, is proven that, for fixed block size β and $\delta \in (\frac{1}{4}, 1)$, the block basis reduction algorithm runs in polynomial time. But in the worst case this statement is wrong. Without this heuristic assumption and with small modifications of the algorithm Ritter [Rit97] proved, for $\beta = 3$ and $\frac{1}{2} \leq \delta < \frac{1}{2}\sqrt{3}$, that the block basis reduction algorithm runs in polynomial time.

Chapter 4

Block Basis Reduction for Arbitrary Norm

In this chapter we describe the generalization of the block basis reduction algorithm for arbitrary norm. In the first section we present the L^3 algorithm for arbitrary norm, due to Lovász and Scarf [LS92]. In order to do that we introduce special distance functions, which minimize the distance to $\text{span}(\mathbf{b}_1, \dots, \mathbf{b}_{i-1})$ not with respect to the Euclidean, but with respect to an arbitrary norm. In the second subsection we present the block basis reduction algorithm and the associated enumeration method. In the last section of the chapter we present an algorithm for calculating the minimal distances w.r.t the l_p norm. We also prove the quality of the approximation of the successive minima, which give the described reduced bases.

4.1 The L^3 Algorithm for Arbitrary Norm

In many applications of lattice basis reduction we are looking for lattice vectors, which are short not with respect to the Euclidean norm but with respect to a given arbitrary norm. Special meaning for our decoding problem has the l_p norm on \mathbb{R}^n defined as:

$$\|\cdot\|_p : \mathbb{R}^n \rightarrow \mathbb{R}, \quad \|\mathbf{x}\|_p = \left(\sum_{i=1}^n |x_i|^p \right)^{\frac{1}{p}}. \quad (4.1)$$

In order to be able to generalize the reduction concepts to an arbitrary norm, we minimize the distance to $\text{span}(\mathbf{b}_1, \dots, \mathbf{b}_{i-1})$ with respect to this norm. Therefore we use the distance functions defined by Lovász and Scarf [LS92].

Definition 4.1. *Let $\mathbf{b}_1, \dots, \mathbf{b}_m \in \mathbb{R}^n$ be linearly independent vectors, $m \leq n$, and $\|\cdot\|$ be a norm on \mathbb{R}^n . Then the i -th distance function w.r.t the norm $\|\cdot\|$ for $1 \leq i \leq m$*

$$F_i(\mathbf{b}_1, \dots, \mathbf{b}_{i-1}) : \mathbb{R}^n \rightarrow \mathbb{R}$$

is defined as

$$F_1(\mathbf{v}) := \|\mathbf{v}\|$$
$$F_i(\mathbf{v}) := F_i(\mathbf{b}_1, \dots, \mathbf{b}_{i-1})(\mathbf{v}) = \min_{\alpha_1, \dots, \alpha_{i-1} \in \mathbb{R}} \left\| \mathbf{v} + \sum_{j=1}^{i-1} \alpha_j \mathbf{b}_j \right\|, \quad 1 \leq i \leq m.$$

Since $F_i(\mathbf{v}) = 0$ if and only if $\mathbf{v} \in \text{span}(\mathbf{b}_1, \dots, \mathbf{b}_{i-1})$, it can be easily shown that the distance functions F_i are norms on $\text{span}(\mathbf{b}_1, \dots, \mathbf{b}_{i-1})^\perp$. With respect to the Euclidean norm we have $F_i(x) = \|\mathbf{b}_i^*\|_2$. Analog to the Euclidean case (see Corollary 3.7) the distance functions give a lower bound for the length of the shortest non-zero vector in a lattice \mathcal{L} .

Lemma 4.2. *Let $\mathbf{b}_1, \dots, \mathbf{b}_m$ be a basis of the lattice $\mathcal{L} \subset \mathbb{R}^n$. Then the following inequality holds:*

$$\min_{i=1, \dots, m} F_i(\mathbf{b}_i) \leq \lambda_{1, \|\cdot\|}(\mathcal{L})$$

Proof: Let $\mathbf{b} = \sum_{i=1}^m t_i \mathbf{b}_i$ with $\|\mathbf{b}\| = \lambda_{1, \|\cdot\|}(\mathcal{L})$. Let $s := \max\{i \mid t_i \neq 0\}$. Then from $t_s \in \mathbb{Z} \setminus \{0\}$ we get:

$$\lambda_{1, \|\cdot\|}(\mathcal{L}) = \|\mathbf{b}\| \geq \min_{\xi_1, \dots, \xi_{s-1}} \left\| \mathbf{b} + \sum_{j=1}^{s-1} \xi_j \mathbf{b}_j \right\| = F_s(\mathbf{b}) = F_s(t_s \mathbf{b}_s) = \underbrace{|t_s|}_{\geq 1} \cdot F_s(\mathbf{b}_s) \geq F_s(\mathbf{b}_s).$$

Then Lemma 4.2 follows from

$$\min_{i=1, \dots, m} F_i(\mathbf{b}_i) \leq F_s(\mathbf{b}_s) \leq \lambda_{1, \|\cdot\|}(\mathcal{L}).$$

□

Lovász and Scarf [LS92] generalize the concept of L^3 -reduced basis to an arbitrary norm. They give the following definition of a reduced basis:

Definition 4.3. *Let $\|\cdot\|$ be an arbitrary norm on \mathbb{R}^n and $0 < \Delta \leq 1$. Let $\mathbf{b}_1, \dots, \mathbf{b}_m$ be a basis of the lattice $\mathcal{L} \subset \mathbb{R}^n$. Then we call $\mathbf{b}_1, \dots, \mathbf{b}_m$ Lovász-Scarf-reduced with Δ or LS-reduced with Δ if*

$$F_j(\mathbf{b}_i) \leq F_j(\mathbf{b}_i \pm \mathbf{b}_j) \quad \text{for all } 1 \leq j < i \leq m, \quad (4.2)$$

$$\Delta F_i(\mathbf{b}_i) \leq F_i(\mathbf{b}_{i+1}) \quad \text{for } 1 \leq i < m. \quad (4.3)$$

The lattice basis is called size-reduced if the first condition is fulfilled.

Remark 4.4. *The condition (4.2) is equivalent to*

$$F_j(\mathbf{b}_i) = \min_{\mu \in \mathbb{Z}} F_j(\mathbf{b}_i + \mu \cdot \mathbf{b}_j) \quad \text{for all } 1 \leq j < i \leq m.$$

This equivalence follows from the convexity of the distance functions F_j , $1 \leq j \leq m$.

For the Euclidean norm the notions of L^3 -reduced basis with δ and LS-reduced basis with $\Delta = \sqrt{\delta}$ are identical. For $\Delta > \frac{1}{2}$ the LS-reduced bases approximate the successive minima up to a factor exponential in the dimension of the lattice.

Theorem 4.5. [LS92] *Let $\mathbf{b}_1, \dots, \mathbf{b}_m$ be a LS-reduced basis with $\Delta \in (1/2, 1]$ of the lattice $\mathcal{L} \subset \mathbb{R}^n$. Then for $i = 1, \dots, m$,*

$$F_i(\mathbf{b}_i) \left(\Delta - \frac{1}{2} \right)^{m-i} \leq \lambda_{i, \|\cdot\|}(\mathcal{L}) \leq F_i(\mathbf{b}_i) / \left(\Delta - \frac{1}{2} \right)^{i-1}.$$

We follow the proof given by Lovász and Scarf [LS92].

Proof: First we prove $\lambda_{i,\|\cdot\|}(\mathcal{L}) \leq F_i(\mathbf{b}_i) / (\Delta - \frac{1}{2})^{i-1}$:

We first argue that for $1 \leq j < i \leq m$

$$F_j(\mathbf{b}_i) \leq F_i(\mathbf{b}_i) + \frac{1}{2} \sum_{t=j}^{i-1} F_t(\mathbf{b}_t). \quad (4.4)$$

Let ξ_0 be the minimal place of $F_t(\mathbf{b}_i + \xi \mathbf{b}_t)$ i.e

$$F_t(\mathbf{b}_i + \xi_0 \mathbf{b}_t) = \min_{\xi \in \mathbb{R}} F_t(\mathbf{b}_i + \xi \mathbf{b}_t) = F_{t+1}(\mathbf{b}_i).$$

Therefore we have

$$\begin{aligned} F_t(\mathbf{b}_i) &\stackrel{(4.2)}{\leq} \min_{\mu \in \mathbb{Z}} F_t(\mathbf{b}_i + \mu \mathbf{b}_t) \leq F_t(\mathbf{b}_i + \lceil \xi_0 \rceil \mathbf{b}_t) \\ &= F_t(\mathbf{b}_i + \xi_0 \mathbf{b}_t + \underbrace{(\lceil \xi_0 \rceil - \xi_0)}_{\leq \frac{1}{2}} \mathbf{b}_t) \leq F_{t+1}(\mathbf{b}_i) + \frac{1}{2} F_t(\mathbf{b}_t). \end{aligned}$$

Then (4.4) follows from the consecutive application of the last inequality for $j \leq t \leq (i-1)$. From $\Delta F_i(\mathbf{b}_i) \stackrel{(4.3)}{\leq} F_i(\mathbf{b}_{i+1}) \leq \min F_i(\mathbf{b}_{i+1} + \mu \mathbf{b}_i) \leq F_{i+1}(\mathbf{b}_{i+1}) + \frac{1}{2} F_i(\mathbf{b}_i)$ we get

$$F_i(\mathbf{b}_i) \stackrel{\Delta \in (1/2, 1]}{\leq} F_{i+1}(\mathbf{b}_{i+1}) \frac{1}{(\Delta - \frac{1}{2})}.$$

Recursive application of this relation gives

$$F_{i-j}(\mathbf{b}_{i-j}) \leq \frac{F_i(\mathbf{b}_i)}{(\Delta - \frac{1}{2})^j}. \quad (4.5)$$

So we get that

$$\sum_{j=1}^{i-1} F_j(\mathbf{b}_j) = \sum_{j=1}^{i-1} F_{i-j}(\mathbf{b}_{i-j}) \leq F_i(\mathbf{b}_i) \sum_{j=1}^{i-1} \frac{1}{(\Delta - \frac{1}{2})^j}$$

and we can substitute this relation back into (4.4) for $j = 1$ to get

$$F_1(\mathbf{b}_i) \leq F_i(\mathbf{b}_i) \left(1 + \frac{1}{2} \sum_{j=1}^{i-1} \frac{1}{(\Delta - \frac{1}{2})^j} \right). \quad (4.6)$$

Let $r = (\Delta - \frac{1}{2})$. Then we have

$$1 + \frac{1}{2} \underbrace{\sum_{j=1}^{i-1} \left(\frac{1}{r}\right)^j}_{\text{geom. progression}} = 1 + \frac{1 - \frac{1}{r^i}}{2(r-1)} = \frac{1}{r^{i-1}} \left(\frac{2r^i - r^{i-1} - 1}{2(r-1)} \right) \underset{0 \leq r \leq \frac{1}{2}}{\leq} \frac{1}{r^{i-1}}. \quad (4.7)$$

So substituting (4.7) in (4.6) gives us

$$F_1(\mathbf{b}_i) \leq \frac{F_i(\mathbf{b}_i)}{r^{i-1}} = \frac{F_i(\mathbf{b}_i)}{\left(\Delta - \frac{1}{2}\right)^{i-1}}$$

By Definition 3.2 it follows that $\lambda_{i,\|\cdot\|}(\mathcal{L}) \leq F_1(\mathbf{b}_i)$, which proves the right side of the inequality of Theorem 4.5.

Now we prove $\lambda_{i,\|\cdot\|}(\mathcal{L}) \geq F_i(\mathbf{b}_i) \left(\Delta - \frac{1}{2}\right)^{m-i}$:

Let \mathbf{v}_i for $i = 1, \dots, m$ be linearly independent vectors in \mathcal{L} , which realize the successive minima of \mathcal{L} , i.e., $\|\mathbf{v}_i\| = F_1(\mathbf{v}_i) = \lambda_{i,\|\cdot\|}(\mathcal{L})$. Then there exist $c_{i,j} \in \mathbb{Z}$ ($1 \leq i, j \leq m$) such that

$$\begin{aligned} \mathbf{v}_1 &= c_{1,1}\mathbf{b}_1 + c_{1,2}\mathbf{b}_2 + \cdots + c_{1,m}\mathbf{b}_m \\ &\vdots \\ \mathbf{v}_i &= c_{i,1}\mathbf{b}_1 + c_{i,2}\mathbf{b}_2 + \cdots + c_{i,m}\mathbf{b}_m \\ &\vdots \\ \mathbf{v}_m &= c_{m,1}\mathbf{b}_1 + c_{m,2}\mathbf{b}_2 + \cdots + c_{m,m}\mathbf{b}_m. \end{aligned}$$

For each index i , there exists a pair of indices j and k with $j \leq i \leq k$ such that $c_{j,k} \neq 0$, since otherwise

$$\begin{aligned} \mathbf{v}_1 &= c_{1,1}\mathbf{b}_1 + \cdots + c_{1,i-1}\mathbf{b}_{i-1} \\ &\vdots \\ \mathbf{v}_i &= c_{i,1}\mathbf{b}_1 + \cdots + c_{i,i-1}\mathbf{b}_{i-1}. \end{aligned}$$

and the vectors $\mathbf{v}_1, \dots, \mathbf{v}_i$ would be linearly dependent. Then for each i , let k be the largest index such that $c_{j,k} \neq 0$ for some $j \leq i \leq k$. Let $\alpha_{j,i} \in \mathbb{R}$, ($1 \leq j \leq m$, $1 \leq i \leq k-1$) satisfy the condition

$$F_k(\mathbf{v}_j) = \min_{\xi_1, \dots, \xi_{i-1} \in \mathbb{R}} \left\| \mathbf{v}_j + \sum_{t=1}^{k-1} \xi_t \mathbf{b}_t \right\| = \left\| \mathbf{v}_j + \sum_{t=1}^{k-1} \alpha_{j,t} \mathbf{b}_t \right\|.$$

Then we have that

$$F_1(\mathbf{v}_j) = F_1\left(\sum_{t=1}^k c_{j,t} \mathbf{b}_t\right) \geq F_1\left(\sum_{t=1}^k c_{j,t} \mathbf{b}_t + \sum_{t=1}^{k-1} \alpha_{j,t} \mathbf{b}_t\right) = F_k(\mathbf{v}_j). \quad (4.8)$$

But then, since $|c_{j,k}| \geq 1$, we get

$$\begin{aligned} \lambda_i \geq \lambda_j &= F_1(\mathbf{v}_j) \stackrel{(4.8)}{\geq} F_k(\mathbf{v}_j) = |c_{j,k} F_k(\mathbf{b}_k)| \stackrel{(4.5)}{\geq} F_i(\mathbf{b}_i) \left(\Delta - \frac{1}{2}\right)^{k-i} \\ &\stackrel{(\Delta - \frac{1}{2}) \leq 1}{\geq} F_i(\mathbf{b}_i) \left(\Delta - \frac{1}{2}\right)^{m-i}, \end{aligned}$$

which proves the inequality on the left-hand side of Theorem 4.5. \square

The following algorithm, presented by Lovász and Scarf [LS92], performs LS-reduction with $\frac{1}{2} < \Delta < 1$ for a given lattice basis $\mathbf{b}_1, \dots, \mathbf{b}_m$ and Δ .

Algorithm 4.1 LS Basis Reduction Algorithm - The L^3 Basis Reduction Algorithm for Arbitrary Norm

INPUT: $(\mathbf{b}_1, \dots, \mathbf{b}_m) \in \mathbb{Z}^n$, $\frac{1}{2} < \Delta < 1$

```

k := 2
while k ≤ m do
  /* Size-reduction of  $\mathbf{b}_k$  */
  for j = k - 1, ..., 1 do
     $\mu_j = \{\mu \in \mathbb{Z} \mid F_j(\mathbf{b}_k + \mu\mathbf{b}_j) \text{ is minimal}\}$ 
     $\mathbf{b}_k := \mathbf{b}_k + \mu_j\mathbf{b}_j$ 
  end for
  /* Swap  $\mathbf{b}_{k-1}$  and  $\mathbf{b}_k$  or increment k */
  if  $F_{k-1}(\mathbf{b}_k) < \Delta F_{k-1}(\mathbf{b}_{k-1})$  then
     $\mathbf{b}_k \longleftrightarrow \mathbf{b}_{k-1}$ 
     $k := \max(k - 1, 2)$ 
  else
     $k := k + 1$ 
  end if
end while

```

Assume we have reached level k , $2 \leq k \leq m$ of the algorithm. In the size-reduction step we transform the vector b_k in the following way: For $j = k - 1, \dots, 1$ we calculate a real number μ , such that

$$F_j(\mathbf{b}_k + \mu\mathbf{b}_j) \tag{4.9}$$

is minimal. Since F_j is a convex function, the integral minimum of (4.9) μ_j is one of the integers $\lfloor \mu \rfloor$ or $\lceil \mu \rceil$ and then we set

$$\mathbf{b}_k := \mathbf{b}_k + \mu_j\mathbf{b}_j.$$

At the end of the stage we get

$$\mathbf{b}_k = \mathbf{b}_k + \sum_{j=1}^{k-1} \mu_j\mathbf{b}_j,$$

which obviously assures the condition (4.2). If after this replacement, the condition (4.3) is fulfilled then we move to the next level. Otherwise, we interchange \mathbf{b}_k and \mathbf{b}_{k-1} and move to the preceding level $k - 1$, unless $k = 2$, in which case we remain at level 2. Hence at every stage k the conditions (4.2) and (4.3) hold for the basis $\mathbf{b}_1, \dots, \mathbf{b}_{k-1}$, and therefore it is LS-reduced with Δ . So the basis $\mathbf{b}_1, \dots, \mathbf{b}_m$ is LS-reduced with Δ when $k = m + 1$.

Theorem 4.6. [LS92] For arbitrary norm $\|\cdot\|$ and $\frac{1}{2} < \Delta < 1$ Algorithm 4.1 terminates in polynomial number of steps for fixed m .

4.2 The Block Basis Reduction Algorithm for Arbitrary Norm

In this section we present the concept of block reduced bases for arbitrary norm and we give an algorithm for calculating such bases.

Definition 4.7. Let $\beta \in \mathbb{Z}$, $\beta \geq 2$ and $\Delta \in \mathbb{R}$, $\Delta \in (0, 1]$. We call a basis $\mathbf{b}_1, \dots, \mathbf{b}_m$ of a lattice $\mathcal{L} \subset \mathbb{R}^n$ β -block reduced with Δ or (β, Δ) -reduced (w.r.t. a norm $\|\cdot\|$) if for $i = 1, \dots, m$

1. $F_j(\mathbf{b}_i) \leq F_j(\mathbf{b}_i \pm \mathbf{b}_j)$ for all $j < i$,
2. $\Delta F_i(\mathbf{b}_i) \leq \lambda_{1, F_i}(\mathcal{L}(\mathbf{b}_i, \dots, \mathbf{b}_{\min(i+\beta-1, m)}))$.

Remark 4.8. A lattice basis $\mathbf{b}_1, \dots, \mathbf{b}_m$ is (β, δ) -reduced (in the sense of Definition 3.13) if and only if it is (β, Δ) -reduced (in the sense of Definition 4.7) and $\Delta = \sqrt{\delta}$.

In order to generalize the Hermite constant γ_β for an arbitrary norm $\|\cdot\|$ on \mathbb{R}^m we define the constants

$$\begin{aligned} \kappa_{m, \|\cdot\|} &:= \sup \left\{ \lambda_{1, \|\cdot\|}(\mathcal{L}(\mathbf{b}_i, \dots, \mathbf{b}_m)) \left(\prod_{i=1}^m F_i(\mathbf{b}_i) \right)^{-\frac{1}{m}} : \mathbf{b}_1, \dots, \mathbf{b}_m \text{ is basis of } \mathcal{L} \subset \mathbb{R}^m \right\} \\ \kappa_m &:= \sup \{ \kappa_{m, \|\cdot\|} : \|\cdot\| \text{ is a norm on } \mathbb{R}^m \} \end{aligned} \quad (4.10)$$

The following Theorem shows that κ_m is well-defined.

Theorem 4.9. [Kai94] Every basis $\mathbf{b}_1, \dots, \mathbf{b}_m$ of a lattice $\mathcal{L} \subset \mathbb{R}^n$ satisfies

$$\min_{i=1, \dots, m} F_i(\mathbf{b}_i) \leq \lambda_{1, \|\cdot\|}(\mathcal{L}) \leq \left(m! \prod_{i=1}^m F_i(\mathbf{b}_i) \right)^{\frac{1}{m}}.$$

From $\gamma_m = \kappa_{m, \|\cdot\|_2}^2$, $\kappa_{m, \|\cdot\|} \leq \kappa_m$ and Theorem 4.9 we obtain the following bounds for κ_m :

$$\sqrt{\gamma_m} \leq \kappa_m \leq m!^{\frac{1}{m}}.$$

For $m \rightarrow \infty$ the asymptotic bounds for κ_m are:

$$\sqrt{\frac{m}{2\pi e}}(1 + o(1)) \leq \kappa_m \leq m!^{\frac{1}{m}} = \frac{m}{e}(1 + o(1)).$$

Remark 4.10. For $m \geq 1$ holds $\kappa_m^m \leq \frac{1}{2}\kappa_{m+1}^{m+1}$ [Kai94]. From $\kappa_1 = 1$ follows that $\kappa_m^m \geq 2^{m-1}$ and $\kappa_m^2 \geq 2$ for $m \geq 2$. From $2!^{1/2} = \sqrt{2}$ follows that $\kappa_2 = \sqrt{2}$.

Analog to the (β, δ) -reduced bases w.r.t. the Euclidean norm, the (β, Δ) -reduced bases w.r.t. an arbitrary norm approximate the successive minima up to an exponential factor (in m/β), although this factor is bigger for arbitrary norm. From Theorem 4.5 follows that the LS-reduced bases approximate the successive minima up to an exponential factor (in m). Therefore the (β, δ) -reduced bases give better approximation than the LS-reduced bases. The approximation of the (β, δ) -reduced bases becomes better with increased block size and bigger Δ . If β is a fixed fraction of m , then the successive minima will be approximated up to a polynomial factor in m .

Theorem 4.11. *Let $2 \leq \beta \leq m$ and $0 < \Delta \leq 1$. For every (β, Δ) -reduced basis $\mathbf{b}_1, \dots, \mathbf{b}_m$ of a lattice $\mathcal{L} \subset \mathbb{R}^n$ holds:*

$$\frac{\|\mathbf{b}_i\|}{\lambda_{i,\|\cdot\|}(\mathcal{L})} \leq \frac{i+1}{4} \left(\frac{\kappa_\beta}{\Delta}\right)^{2\frac{m-1}{\beta-1}} \quad \text{for } 1 \leq i \leq m, \quad (4.11)$$

$$\frac{\|\mathbf{b}_i\|}{\lambda_{i,\|\cdot\|}(\mathcal{L})} \geq \frac{2\Delta}{i+1} \quad \text{for } 1 \leq i \leq \beta, \quad (4.12)$$

$$\frac{\|\mathbf{b}_i\|}{\lambda_{i,\|\cdot\|}(\mathcal{L})} \geq \frac{4}{i+1} \left(\frac{\Delta}{\kappa_\beta}\right)^{2\frac{i-1}{\beta-1}} \quad \text{for } \beta \leq i \leq m. \quad (4.13)$$

Kaib [Kai94] proved this theorem for $\Delta = 1$. We follow his proof and prove first the following lemma:

Lemma 4.12. *Let $2 \leq \beta \leq m$ and $0 < \Delta \leq 1$. For every (β, Δ) -reduced basis $\mathbf{b}_1, \dots, \mathbf{b}_m$ of a lattice $\mathcal{L} \subset \mathbb{R}^n$ holds:*

$$\|\mathbf{b}_1\| \leq \frac{\Delta}{2} \left(\frac{\kappa_\beta}{\Delta}\right)^{2\frac{m-1}{\beta-1}} \max_{i=m-\beta+1, \dots, m-1} F_i(\mathbf{b}_i), \quad (4.14)$$

$$\|\mathbf{b}_1\| \leq \frac{1}{2} \left(\frac{\kappa_\beta}{\Delta}\right)^{2\frac{m-1}{\beta-1}} \lambda_{1,\|\cdot\|}(\mathcal{L}). \quad (4.15)$$

Proof:

(4.14): Let $h_i := F_i(\mathbf{b}_i)$. From Definition 4.7 we obtain

$$\begin{aligned} \Delta^i h_1^i &\leq \lambda_{i,\|\cdot\|}^i(\mathcal{L}(\mathbf{b}_1, \dots, \mathbf{b}_i)) && \text{for } i = 1, \dots, \beta - 1, \\ \Delta^\beta h_i^\beta &\leq \lambda_{i, F_i}^\beta(\mathcal{L}(\mathbf{b}_i, \dots, \mathbf{b}_{i+\beta-1})) && \text{for } i = 1, \dots, m - \beta. \end{aligned}$$

Then using (4.10) we get

$$\begin{aligned} \Delta^i h_1^i &\leq \kappa_i^i h_1 \cdots h_i && \text{for } i = 1, \dots, \beta - 1, \\ \Delta^\beta h_i^\beta &\leq \kappa_\beta^\beta h_i \cdots h_{i+\beta-1} && \text{for } i = 1, \dots, m - \beta. \end{aligned}$$

Multiplication of these inequalities gives

$$\Delta^{\binom{\beta}{2} + \beta(m-\beta)} h_1^{\binom{\beta+1}{2}} h_2^\beta \cdots h_{m-\beta}^\beta \leq \kappa_1^1 \kappa_2^2 \cdots \kappa_{\beta-1}^{\beta-1} \kappa_\beta^{\beta(m-\beta)} h_1^\beta h_2^\beta \cdots h_{m-\beta}^\beta h_{m-\beta+1}^{\beta-1} \cdots h_{m-1}^1,$$

which is equivalent to

$$\begin{aligned} \Delta^{\binom{\beta}{2} + \beta(m-\beta)} h_1^{\binom{\beta}{2}} &\leq \kappa_1^1 \kappa_2^2 \cdots \kappa_{\beta-1}^{\beta-1} \kappa_\beta^{\beta(m-\beta)} h_{m-\beta+1}^{\beta-1} \cdots h_{m-1}^1 \\ &\leq \kappa_1^1 \kappa_2^2 \cdots \kappa_{\beta-1}^{\beta-1} \kappa_\beta^{\beta(m-\beta)} (\max_{i=m-\beta+1, \dots, m-1} F_i(\mathbf{b}_i))^{\binom{\beta}{2}}. \end{aligned}$$

From $\kappa_m^m \leq \frac{1}{2} \kappa_{m+1}^{m+1}$ for $m \geq 1$ (see Remark 4.10) and the last inequality follows

$$\kappa_1^1 \kappa_2^2 \cdots \kappa_{\beta-1}^{\beta-1} \kappa_\beta^{\beta(m-\beta)} \leq \left(\frac{1}{2}\right)^{\binom{\beta}{2}} \kappa_\beta^{\beta(m-1)},$$

which finishes the proof of (4.14).

(4.15): We prove (4.15) by induction on m for $m \geq \beta$:

(i) *Induction basis: $m = \beta$*

From Definition 4.7 follows that $\Delta \|\mathbf{b}_1\| \leq \lambda_{1,\|\cdot\|}(\mathcal{L})$. From $\kappa_\beta^2 \geq 2$ (see Remark 4.10) and $1 \leq \Delta^{-1} \leq \Delta^{-2}$ follows the inequality (4.15) for $m = \beta$.

(ii) *Induction hypothesis:*

Let assume that (4.15) holds for $m - 1$.

(iii) *Induction step:*

Let $\mathbf{v} = \sum_{i=1}^m u_i \mathbf{b}_i$ and $\|\mathbf{v}\| = \lambda_{1,\|\cdot\|}(\mathcal{L})$.

For $u_m = 0$ $\|\mathbf{v}\| = \lambda_{1,\|\cdot\|}(\mathcal{L}) = \lambda_{1,\|\cdot\|}(\mathcal{L}(\mathbf{b}_1, \dots, \mathbf{b}_{m-1}))$ and (4.15) follows from the induction hypothesis.

Let $u_m \neq 0$. Then for $m - \beta + 1 \leq i \leq m$ hold the inequality

$$\lambda_{1,\|\cdot\|}(\mathcal{L}) = \|\mathbf{v}\| \geq F_i(\mathbf{v}) \geq \lambda_{1,F_i}(\mathcal{L}(\mathbf{b}_i, \dots, \mathbf{b}_{m-1})) \geq \Delta F_i(\mathbf{b}_i).$$

Therefore we obtain

$$\max_{i=m-\beta+1, \dots, m-1} F_i(\mathbf{b}_i) \leq \lambda_{1,\|\cdot\|}(\mathcal{L}) \Delta^{-1}$$

and (4.15) follows from (4.14). □

Proof of Theorem 4.11:

(4.11): For $j = 1, \dots, m$ the basis vectors $\mathbf{b}_j, \dots, \mathbf{b}_m$ are by Definition 4.7 (β, Δ) -reduced w.r.t. the norm F_j . So from Lemma 4.12 follows

$$F_j(\mathbf{b}_j) \leq \frac{1}{2} \left(\frac{\kappa_\beta}{\Delta} \right)^{2 \frac{m-j}{\beta-1}} \lambda_{1,F_j}(\mathcal{L}(\mathbf{b}_j, \dots, \mathbf{b}_m)) \text{ for } 1 \leq j \leq m - \beta + 1.$$

Since $\kappa_\beta^{2 \frac{m-1}{\beta-1}} \geq 2$ and $\Delta^{-1} \leq \Delta^{-2 \frac{m-1}{\beta-1}}$, for $m - \beta + 1 \leq j \leq m$ holds

$$F_j(\mathbf{b}_j) \leq \Delta^{-1} \lambda_{1,F_j}(\mathcal{L}(\mathbf{b}_j, \dots, \mathbf{b}_m)) \leq \frac{1}{2} \left(\frac{\kappa_\beta}{\Delta} \right)^{2 \frac{m-1}{\beta-1}} \lambda_{1,F_j}(\mathcal{L}(\mathbf{b}_j, \dots, \mathbf{b}_m)).$$

From (4.4) and $\lambda_{1,F_j}(\mathcal{L}(\mathbf{b}_j, \dots, \mathbf{b}_m)) \leq \lambda_{i,\|\cdot\|}(\mathcal{L})$ (for $j \leq i$) follows

$$\begin{aligned} \|\mathbf{b}_i\| &\leq F_i(\mathbf{b}_i) + \frac{1}{2} \sum_{j=1}^{i-1} F_j(\mathbf{b}_j) \\ &\leq \frac{1}{2} \left(\frac{\kappa_\beta}{\Delta} \right)^{2 \frac{m-1}{\beta-1}} \left(1 + \frac{1}{2}(i-1) \right) \lambda_{i,\|\cdot\|}(\mathcal{L}) \\ &= \frac{i+1}{4} \left(\frac{\kappa_\beta}{\Delta} \right)^{2 \frac{m-1}{\beta-1}} \lambda_{i,\|\cdot\|}(\mathcal{L}) \text{ for } 1 \leq i \leq m. \end{aligned}$$

(4.12): From the definition of successive minima and inequality (4.4) follows

$$\lambda_{i,\|\cdot\|}(\mathcal{L}) \leq \max_{j=1, \dots, m} \|\mathbf{b}_j\| \leq \max_{j=1, \dots, m} F_j(\mathbf{b}_j) + \frac{i-1}{2} \max_{j=1, \dots, m} F_j(\mathbf{b}_j) = \frac{i+1}{2} \max_{j=1, \dots, m} F_j(\mathbf{b}_j). \quad (4.16)$$

Then from Definition 4.7 we obtain for $\max(1, i - \beta + 1) \leq j \leq i$

$$F_j(\mathbf{b}_j) \leq \Delta^{-1} \lambda_{1, F_j}(\mathcal{L}(\mathbf{b}_j, \dots, \mathbf{b}_{\min(j+\beta-1, m)})) \leq \Delta^{-1} F_j(\mathbf{b}_i) \leq \Delta^{-1} \|\mathbf{b}_i\|.$$

Therefore for $1 \leq i \leq \beta$ holds

$$\max_{j=1, \dots, i} F_j(\mathbf{b}_j) \leq \Delta^{-1} \|\mathbf{b}_i\|.$$

So (4.12) follows from the last inequality and (4.16).

(4.13): For $i \geq \beta$ and $1 \leq j \leq i - \beta + 1$ the basis vectors $\mathbf{b}_j, \dots, \mathbf{b}_i$ are (β, Δ) -reduced w.r.t. the norm F_j . So from Lemma 4.12 we get

$$\begin{aligned} F_j(\mathbf{b}_j) &\leq \frac{1}{2} \left(\frac{\kappa_\beta}{\Delta} \right)^{2 \frac{i-j}{\beta-1}} \lambda_{1, F_j}(\mathcal{L}(\mathbf{b}_j, \dots, \mathbf{b}_m)) \\ &\leq \frac{1}{2} \left(\frac{\kappa_\beta}{\Delta} \right)^{2 \frac{i-j}{\beta-1}} F_j(\mathbf{b}_i) \\ &\leq \frac{1}{2} \left(\frac{\kappa_\beta}{\Delta} \right)^{2 \frac{i-1}{\beta-1}} \|\mathbf{b}_i\|. \end{aligned}$$

For $i - \beta + 1 \leq j \leq i$ holds

$$F_j(\mathbf{b}_j) \leq \Delta^{-1} \|\mathbf{b}_i\| \leq \frac{1}{2} \left(\frac{\kappa_\beta}{\Delta} \right)^{2 \frac{i-1}{\beta-1}} \|\mathbf{b}_i\|.$$

Then from (4.16) follows (4.13). □

Remark 4.13. All norms over \mathbb{R}^n are equivalent, i.e. for every norm $\|\cdot\|$ over \mathbb{R}^n there exist positive constants $r = r(\|\cdot\|)$ and $R = R(\|\cdot\|)$, such that

$$r(\|x\|) \leq \|x\|_2 \leq R(\|x\|)$$

holds for all $x \in \mathbb{R}^n$. Therefore for every lattice $\mathcal{L} \subset \mathbb{R}^n$ holds the inequality

$$\lambda_{1, \|\cdot\|_2}(\mathcal{L}) \leq R(\|\cdot\|) \lambda_{1, \|\cdot\|}(\mathcal{L}).$$

From Theorem 3.15 we get:

For every (β, δ) -reduced basis (in the sense of Definition 3.13) $\mathbf{b}_1, \dots, \mathbf{b}_m \in \mathbb{Z}^n$ with $2 \leq \beta \leq m$ and $0 < \delta \leq 1$ and for every norm $\|\cdot\|$ over \mathbb{R}^n holds

$$\|\mathbf{b}_1\| \leq \frac{R(\|\cdot\|)}{r(\|\cdot\|)} \frac{1}{\sqrt{\delta}} \left(\frac{\gamma_\beta}{\delta} \right)^{\frac{m-1}{\beta-1}} \lambda_{1, \|\cdot\|}(\mathcal{L}) \quad (4.17)$$

The first vector of (β, δ) -reduced basis w.r.t. the Euclidean norm approximates the first successive minimum also w.r.t. an arbitrary norm.

For the l_p norm we have that

$$r(\|\cdot\|_p) = \begin{cases} n^{\frac{1}{2} - \frac{1}{p}} & \text{for } p < 2 \\ 1 & \text{for } p > 2 \end{cases} \quad \text{and} \quad R(\|\cdot\|_p) = \begin{cases} 1 & \text{for } p < 2 \\ n^{\frac{1}{2} - \frac{1}{p}} & \text{for } p > 2 \end{cases}$$

and therefore $\frac{R(\|\cdot\|_p)}{r(\|\cdot\|_p)} = \sqrt{n}$.

Let $\Delta = \sqrt{\delta}$. From (4.17) we obtain that for (β, Δ^2) -block reduced basis w.r.t. the Euclidean norm holds

$$\|\mathbf{b}_1\|_p \leq \frac{\sqrt{n}}{\Delta} \left(\frac{\gamma_\beta}{\Delta^2} \right)^{\frac{m-1}{\beta-1}} \lambda_{1, \|\cdot\|_p}(\mathcal{L}).$$

From (4.15) we obtain that for (β, Δ^2) -block reduced basis w.r.t. l_p norm holds

$$\|\mathbf{b}_1\|_p \leq \frac{1}{2} \left(\frac{\kappa_\beta^2}{\Delta^2} \right)^{\frac{m-1}{\beta-1}} \lambda_{1, \|\cdot\|_p}(\mathcal{L}).$$

Therefore for β close to m and big dimensions n the (β, Δ) -reduction algorithm achieves better approximation than the (β, δ) -reduction algorithm with $\delta = \Delta^2$.

The following algorithm performs a (β, Δ) -reduction for $2 \leq \beta \leq m$. It uses LS-reduction Algorithm (see Algorithm 4.1) and a subroutine $ADFS(j, k)$ (see Algorithm 4.3) which finds an integer non-zero vector (u_j, \dots, u_k) , such that the vector $\mathbf{b}_j^{\text{new}} = \sum_{i=j}^k u_i \mathbf{b}_i$ has minimal norm \bar{F}_j in $\mathcal{L}(\mathbf{b}_j, \dots, \mathbf{b}_k)$, i.e., the vector $\mathbf{b}_j^{\text{new}}$ satisfies the following condition

$$\bar{F}_j := F_j(\mathbf{b}_j^{\text{new}}) = F_j\left(\sum_{i=j}^k u_i \mathbf{b}_i\right) = \lambda_{1, F_j}(\mathcal{L}(\mathbf{b}_j, \dots, \mathbf{b}_k)). \quad (4.18)$$

Throughout the algorithm the integer j is cyclically shifted through the integers $1, 2, \dots, m-1$. The variable z counts the number of positions j that satisfy the inequality $\Delta F_j(\mathbf{b}_j) \leq \bar{F}_j$. If this inequality does not hold for some j , then we insert $\mathbf{b}_j^{\text{new}}$ in the basis, using the same subroutine BASIS as in the block reduction algorithm for the Euclidean norm (see Algorithm 3.4). Then we size-reduce the new basis using LS-reduction (see Algorithm 4.1), and we reset z to 0. The integer $j = m$ is always skipped since the inequality obviously holds for $j = m$. Since for every j the value of the j -th distance function is non-negative, the value of \bar{F}_j cannot endlessly decrease. This guarantees that the block basis algorithm terminates after finite number of iterations. Obviously a basis $(\mathbf{b}_1, \dots, \mathbf{b}_m)$ is (β, Δ) -reduced if it is size reduced and $z = m - 1$. Therefore the algorithm produces a basis that is (β, Δ) -reduced.

Algorithm 4.2 Block Basis Reduction Algorithm for Arbitrary Norm

INPUT: $(\mathbf{b}_1, \dots, \mathbf{b}_m) \in \mathbb{Z}^n$, $\frac{1}{2} < \Delta < 1$, $2 \leq \beta \leq m$
OUTPUT: (β, Δ) -reduced basis $(\mathbf{b}_1, \dots, \mathbf{b}_m)$

```

 $z := 0$ 
 $j := m - 1$ 
 $LS\_REDUCTION((\mathbf{b}_1, \dots, \mathbf{b}_m), \Delta)$ 
while  $z < m - 1$  do
   $j := j + 1$ 
  if  $j = m$  then
     $j := 1$ 
  end if
   $k := \min(j + \beta - 1, m)$ 
   $(\bar{F}_j, (u_j, \dots, u_k), \mathbf{b}_j^{\text{new}}) := ADFS(j, k, (\mathbf{b}_1, \dots, \mathbf{b}_k))$ 
   $h := \min(k + 1, m)$ 
  if  $\bar{F}_j < \Delta F_j(\mathbf{b}_j)$  then
     $BASIS((u_j, \dots, u_k), (\mathbf{b}_1, \dots, \mathbf{b}_k))$ 
     $LS\_REDUCTION((\mathbf{b}_1, \dots, \mathbf{b}_h), \Delta)$ 
     $z := 0$ 
  else
     $LS\_REDUCTION((\mathbf{b}_1, \dots, \mathbf{b}_h), \Delta)$ 
     $z := z + 1$ 
  end if
end while

return  $(\mathbf{b}_1, \dots, \mathbf{b}_m)$ 

```

The ADFS algorithm (Algorithm 4.3) finds an integer non-zero vector (u_j, \dots, u_k) , such that the vector $\mathbf{b}_j^{\text{new}} = \sum_{i=j}^k u_i \mathbf{b}_i$ satisfies the following condition

$$\bar{F}_j := F_j(\mathbf{b}_j^{\text{new}}) = F_j\left(\sum_{i=j}^k u_i \mathbf{b}_i\right) = \lambda_{1, F_j}(\mathcal{L}(\mathbf{b}_j, \dots, \mathbf{b}_k)). \quad (4.19)$$

The minimum F_j is searched in the sublattices $\mathcal{L}(\mathbf{b}_j, \dots, \mathbf{b}_s)$ with increasing s . With (u_j, \dots, u_k) we denote the current minimal place of the norm $F_j(\sum_{i=j}^k \tilde{u}_i \mathbf{b}_i)$ and with \bar{F}_j we denote its current minimum. In the beginning we set $\bar{F}_j = F_j(\mathbf{b}_j)$ and

$$(u_j, u_{j+1}, \dots, u_k) = (\tilde{u}_j, \tilde{u}_{j+1}, \dots, \tilde{u}_k) := (1, 0, \dots, 0).$$

Throughout the algorithm we enumerate the integers \tilde{u}_t in order of the ascending values of $F_t(\sum_{i=t}^s \tilde{u}_i \mathbf{b}_i)$. For this purpose we calculate a real number z_t for which $F_t(z \mathbf{b}_t + \sum_{i=t+1}^s \tilde{u}_i \mathbf{b}_i)$ is minimal. Since F_t is a convex function, the integral minimum \tilde{u}_t is one of the integers $l_t = \lfloor z_t \rfloor$ or $r_t = \lceil z_t \rceil$. In every stage of the algorithm we have $\tilde{u}_s > 0$, i.e. the vector \mathbf{b}_s is used always with a positive sign for the search. This prevents redundancies during the enumeration. If $F_t(l_t \mathbf{b}_t + \sum_{i=t+1}^s \tilde{u}_i \mathbf{b}_i) \geq F_j(\sum_{i=j}^k u_i \mathbf{b}_i)$, then this inequality holds for all $\tilde{u}_t < l_t$ and we can stop the enumeration to the left. Similarly we

can stop the enumeration to the right whenever $F_t(r_t \mathbf{b}_t + \sum_{i=t+1}^s \tilde{u}_i \mathbf{b}_i) \geq F_j(\sum_{i=j}^k u_i \mathbf{b}_i)$.

In the algorithm ADFS the initialization step will be executed exactly once and it needs $O(k - j + 1)$ arithmetic operations. For each iteration of the while-loop ADFS executes $O(k - j + 1) = O(n\beta)$ arithmetic operations and $O(1)$ calculations of the distance functions. With the following theorem we give an upper bound of the number of iterations in the while-loop of ADFS.

Theorem 4.14. [Rit97] *For block size $\beta = k - j + 1$ and an LS-reduced basis b_1, \dots, b_k the algorithm ADFS makes at most $(1/(\Delta - \frac{1}{2}))^{O(\beta^2)}$ iterations in the while-loop.*

We follow the proof from Ritter [Rit97].

Proof: Let $A(t, x)$ be the number of enumerated vectors $\tilde{u}_t, \dots, \tilde{u}_k$ with $F_t(\sum_{i=t}^s \tilde{u}_i \mathbf{b}_i) < x$.

When ADFS enters stage t always holds $F_{t+1}(\sum_{i=t+1}^k \tilde{u}_i \mathbf{b}_i) < F_j(\mathbf{b}_j)$. Therefore the stage index will be decreased from $t + 1$ to t at most $A(t + 1, F_j(\mathbf{b}_j))$ times and will be increased from $t - 1$ to t at most $A(t, F_j(\mathbf{b}_j)) + 1$ times and $t = j$ stays unchanged at most $A(j, F_j(\mathbf{b}_j))$ times. So we get that the number of iterations A in the while-loop is bounded from

$$A \leq (k - j + 1) + A(j, F_j(\mathbf{b}_j)) + 2 \sum_{t=j+1}^k A(t, F_j(\mathbf{b}_j)). \quad (4.20)$$

Since $F_t(x) \geq F_{t+1}(x)$, we have

$$A(t, F_j(\mathbf{b}_j)) \leq A(t + 1, F_j(\mathbf{b}_j)) \max\# \left\{ \tilde{u}_t \in \mathbb{Z} \mid F_t \left(\sum_{i=t}^k \tilde{u}_i \mathbf{b}_i \right) < F_j(\mathbf{b}_j) \right\},$$

where we take the maximum over all non-zero vectors $(\tilde{u}_{t+1}, \dots, \tilde{u}_k)$.

Let for fixed $(\tilde{u}_{t+1}, \dots, \tilde{u}_k)$, $\tilde{\mathbf{b}}_{t+1} := \sum_{i=t+1}^k \tilde{u}_i \mathbf{b}_i$ and $F_{t+1}(\tilde{\mathbf{b}}_{t+1}) := F_t(\tilde{\mathbf{b}}_{t+1} + z \mathbf{b}_t) < F_j(\mathbf{b}_j)$. Then for all $\tilde{u}_t \in \mathbb{Z}$ with $F_t(\tilde{\mathbf{b}}_{t+1} + \tilde{u}_t \mathbf{b}_t) < F_j(\mathbf{b}_j)$ holds

$$\begin{aligned} F_j(\mathbf{b}_j) &> F_t(\tilde{\mathbf{b}}_{t+1} + \tilde{u}_t \mathbf{b}_t) \\ &= F_t((z - \tilde{u}_t) \mathbf{b}_t - (z \mathbf{b}_t + \tilde{\mathbf{b}}_{t+1})) \\ &\geq F_t((z - \tilde{u}_t) \mathbf{b}_t) - F_t(z \mathbf{b}_t + \tilde{\mathbf{b}}_{t+1}) \\ &= |z - \tilde{u}_t| F_t(\mathbf{b}_t) - F_{t+1}(\tilde{\mathbf{b}}_{t+1}) \\ &> |z - \tilde{u}_t| F_t(\mathbf{b}_t) - F_j(\mathbf{b}_j). \end{aligned}$$

Therefore we get

$$|z - \tilde{u}_t| < 2 \frac{F_j(\mathbf{b}_j)}{F_t(\mathbf{b}_t)},$$

from which we obtain

$$A(t, F_j(\mathbf{b}_j)) < \prod_{i=t}^k \left(4 \frac{F_j(\mathbf{b}_j)}{F_t(\mathbf{b}_t)} + 1 \right). \quad (4.21)$$

Algorithm 4.3 ADFS - Enumeration Subroutine for the Block Basis Reduction Algorithm for Arbitrary Norm

INPUT: j, k ($1 \leq j \leq k \leq m$), $(\mathbf{b}_j, \dots, \mathbf{b}_k)$

OUTPUT: • a vector (u_j, \dots, u_k) which satisfies $F_j(\sum_{i=j}^k u_i \mathbf{b}_i) = \lambda_{1, F_j}(\mathcal{L}(\mathbf{b}_j, \dots, \mathbf{b}_k))$

• $\bar{F}_j := F_j(\sum_{i=j}^k u_i \mathbf{b}_i)$

• $\mathbf{b}_j^{\text{new}} := \sum_{i=j}^k u_i \mathbf{b}_i$

*/*Initialization:*/*

$(u_j, u_{j+1}, \dots, u_k) := (1, 0, \dots, 0)$

$(\tilde{u}_j, \tilde{u}_{j+1}, \dots, \tilde{u}_k) := (1, 0, \dots, 0)$

$\bar{F}_j := F_j(\mathbf{b}_j)$

$s := t := j$

while $t \leq k$ **do**

if $F_t(\sum_{i=t}^s \tilde{u}_i \mathbf{b}_i) < \bar{F}_j$ **then**

if $t > j$ **then**

$t := t - 1$

*/*With fixed $\tilde{u}_{t+1}, \dots, \tilde{u}_s$ find an integer z_t : $F_t(\sum_{i=t}^s \tilde{u}_i \mathbf{b}_i)$ is minimal and set $\tilde{u}_t = z_t$ */*

$z_t := \{z \in \mathbb{R} : F_t(z \mathbf{b}_t + \sum_{i=t+1}^s \tilde{u}_i \mathbf{b}_i) \text{ is minimal}\}$, $l_t := \lfloor z_t \rfloor$, $r_t := l_t + 1$

if $F_t(l_t \mathbf{b}_t + \sum_{i=t}^s \tilde{u}_i \mathbf{b}_i) < F_t(r_t \mathbf{b}_t + \sum_{i=t+1}^s \tilde{u}_i \mathbf{b}_i)$ **then**

$\tilde{u}_t := l_t$

$l_t := l_t - 1$

else

$\tilde{u}_t := r_t$

$r_t := r_t + 1$

end if

else

/ If $t = j$ we have found a shorter (w.r.t F_j) vector*/*

$(u_j, u_{j+1}, \dots, u_k) := (\tilde{u}_j, \tilde{u}_{j+1}, \dots, \tilde{u}_k)$

$\bar{F}_j := F_j(\sum_{i=j}^k u_i \mathbf{b}_i)$

end if

else

/ We cannot find a shorter vector $\sum_{i=t}^s \tilde{u}_i \mathbf{b}_i$ with fixed $\tilde{u}_{t+1}, \dots, \tilde{u}_s$. Therefore we can stop the enumeration of the vectors $(u, \tilde{u}_{t+1}, \dots, \tilde{u}_s)$ and increase t by 1*/*

$t := t + 1$

if $t \geq s$ **then**

$\tilde{u}_t := \tilde{u}_t + 1$

$s := t$

else if $F_t(l_t \mathbf{b}_t + \sum_{i=t+1}^s \tilde{u}_i \mathbf{b}_i) < F_t(r_t \mathbf{b}_t + \sum_{i=t+1}^s \tilde{u}_i \mathbf{b}_i)$ **then**

$\tilde{u}_t := l_t$, $l_t := l_t - 1$

else

$\tilde{u}_t := r_t$

$r_t := r_t + 1$

end if

end if

end while

return \bar{F}_j , (u_j, \dots, u_k) , $\mathbf{b}_j^{\text{new}} = \sum_{i=j}^k u_i \mathbf{b}_i$

Since $\mathbf{b}_1, \dots, \mathbf{b}_k$ is LS-reduced, we have that

$$\Delta F_{t-1}(\mathbf{b}_{t-1}) \leq F_{t-1}(\mathbf{b}_t) \leq F_t(\mathbf{b}_t) + \frac{1}{2} F_{t-1}(\mathbf{b}_{t-1}).$$

Using induction by t we get

$$\frac{F_j(\mathbf{b}_j)}{F_t(\mathbf{b}_t)} \leq \left(\Delta - \frac{1}{2} \right)^{j-t}.$$

Applying this to (4.21) results in

$$\begin{aligned} A(t, F_j(\mathbf{b}_j)) &< \prod_{i=t}^k \left(4 \left(\Delta - \frac{1}{2} \right)^{j-i} + 1 \right) \\ &< 5^{k-t+1} \left(\Delta - \frac{1}{2} \right)^{\sum_{i=t}^k j-i} \\ &= 5^{k-t+1} \left(\Delta - \frac{1}{2} \right)^{(k-j+1)(j-(k+t)/2)} = \left(\Delta - \frac{1}{2} \right)^{O(\beta^2)}. \end{aligned}$$

The theorem follows from (4.20). □

There doesn't exist a polynomial bound for the runtime of the presented block basis reduction algorithm.

4.3 Computing the Values of the Distance Functions for the l_p Norm

Let $\mathbf{b}_1, \dots, \mathbf{b}_m \in \mathbb{R}^n$ be linearly independent vectors, $m \leq n$. Then, according to Definition 4.1, the i -th distance function for $1 \leq i \leq m$ w.r.t the norm $\|\cdot\|_p$ is defined as

$$\begin{aligned} F_i &: \mathbb{R}^n \rightarrow \mathbb{R} \\ F_1(\mathbf{x}) &:= \|\mathbf{x}\|_p \\ F_i(\mathbf{x}) &:= \min_{\xi_1, \dots, \xi_{i-1} \in \mathbb{R}} \left\| \mathbf{x} + \sum_{j=1}^{i-1} \xi_j \mathbf{b}_j \right\|_p, \quad 1 \leq i \leq m. \end{aligned}$$

For $i = 1$ the value of the distance function can be calculated directly from (4.1). For $1 < i \leq m$ we can write $F_i(x)$ as

$$F_i(\mathbf{x}) = \min_{\mathbf{a} \in \mathbb{R}^{i-1}} \|\mathbf{x} - B\mathbf{a}\|_p$$

with

$$B = \begin{pmatrix} \vdots & \vdots & \cdots & \vdots \\ -\mathbf{b}_1 & -\mathbf{b}_2 & \cdots & -\mathbf{b}_{i-1} \\ \vdots & \vdots & \cdots & \vdots \end{pmatrix} \text{ and } \mathbf{a} = \begin{pmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_{i-1} \end{pmatrix}. \quad (4.22)$$

So we can transform the problem of finding the value of the distance functions into the basic approximation problem of finding the l_p solution of a system of linear equations, which can be stated as follows:

Problem 4.15. Let B be a matrix in $\mathbb{Z}^{n \times k}$, $\mathbf{x} \in \mathbb{R}^n$ and $1 < p < \infty$. Let $f(\mathbf{a})$ be the function:

$$f : \mathbb{R}^n \longrightarrow \mathbb{R}^n, \quad f(\mathbf{a}) = \mathbf{x} - B\mathbf{a}.$$

Then find $\mathbf{a} \in \mathbb{R}^n$ to minimize $\|f(\mathbf{a})\|_p$.

For a particular $\mathbf{a} \in \mathbb{R}^k$, let

$$D_1 = \begin{pmatrix} |f_1(\mathbf{a})|^{p-1} & & & \\ & |f_2(\mathbf{a})|^{p-1} & & \\ & & \ddots & \\ & & & |f_n(\mathbf{a})|^{p-1} \end{pmatrix} \text{ and } \boldsymbol{\theta} = \begin{pmatrix} \text{sign}(f_1(\mathbf{a})) \\ \text{sign}(f_1(\mathbf{a})) \\ \vdots \\ \text{sign}(f_n(\mathbf{a})) \end{pmatrix}$$

Let $r : \mathbb{R}^n \longrightarrow \mathbb{R}$, $r(\mathbf{a}) = \|f(\mathbf{a})\|_p^p$. Then r is a convex differentiable function and $r'(\mathbf{a}) = pB^t D_1 \boldsymbol{\theta}$. Therefore we obtain the following the following theorem:

Theorem 4.16. A vector $\mathbf{a} \in \mathbb{R}^n$ solves Problem 4.15 if and only if

$$B^t D_1 \boldsymbol{\theta} = \mathbf{0}. \quad (4.23)$$

For the solution of this problem we propose an algorithm in case $n \geq k$, described by Watson [Wat80]. If $n < k$, then we can append to \mathbf{x} $n - k$ zero entries and to B $n - k$ zero rows. The iterative algorithm, described by Watson, is a variant of the Newton method, which in each iteration uses the solution of Problem 4.15 w.r.t $p = 2$, for certain \mathbf{x} and B , in order to determine the increment to be added to the current iteration point. So in Section 4.3.1 we present an algorithm for Problem 4.15 for $p = 2$ and then in Section 4.3.2 we describe the algorithm for $p > 2$. The described methods for both cases can be found in [Wat80].

4.3.1 The case $p = 2$

For $p = 2$ the Problem 4.15 is called *Least Squares Problem*. Let $\mathbf{a}^* \in \mathbb{R}^n$ solves 4.15 for $p = 2$. Then the system of equations (4.15) can be written as

$$B^t f(\mathbf{a}^*) = \mathbf{0},$$

which gives

$$B^t B \mathbf{a}^* = B^t \mathbf{x}, \quad (4.24)$$

If B has rank k , $B^t B$ is symmetric positive definite matrix, and the solution to (4.16) can be obtained using Cholesky decomposition of the matrix $B^t B$. However, the matrix $B^t B$ can be ill-conditioned with respect to inversion and information can be lost even during the formation of $B^t B$.

Example 4.17. Take

$$B = \begin{pmatrix} 1 & 1 & 1 \\ \varepsilon_1 & 0 & 0 \\ 0 & \varepsilon_2 & 0 \\ 0 & 0 & \varepsilon_3 \end{pmatrix}$$

If $|\varepsilon_i| < 1 =^{-6}$ for $i = 1, \dots, 3$ and the computation is performed on a computer whose word length is such that no more than 10 decimal places are retained, then the matrix

$$B^t B = \begin{pmatrix} 1 + \varepsilon_1^2 & 1 & 1 \\ 1 & 1 + \varepsilon_2^2 & 1 \\ 1 & 1 & 1 + \varepsilon_3^2 \end{pmatrix}$$

will be saved in the computer as

$$B^t B = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

An alternative procedure for the least squares problem, which avoids the formation of $B^t B$, involves the computation of the *Singular Value Decomposition* of a matrix.

Theorem 4.18. Let $B \in \mathbb{R}^{n \times k}$ be a non-zero matrix of rank r . Then there exist an $n \times n$ orthogonal matrix U , a $k \times k$ orthogonal matrix V , and a diagonal matrix

$$W = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_r \end{bmatrix}$$

with $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r > 0$ such that

$$B = USV^t, \text{ with } S = \left[\begin{array}{c|c} W & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{0} \end{array} \right] = \left[\begin{array}{cccc|c} \sigma_1 & & & & \mathbf{0}_{r \times (k-r)} \\ & \sigma_2 & & & \\ & & \ddots & & \\ & & & \sigma_r & \\ \hline \mathbf{0}_{(n-r) \times r} & & & & \mathbf{0}_{(n-r) \times (k-r)} \end{array} \right]$$

We call $\sigma_1, \sigma_2, \dots, \sigma_r$ singular values of B and the decomposition USV^t of B Singular Value Decomposition of B .

Let $r = \text{rank}(B)$ and let the Singular Value Decomposition of B be

$$B = USV^t, \text{ with } S = \left[\begin{array}{c|c} W & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{0} \end{array} \right]$$

as defined in Theorem 4.18. Then

$$\|\mathbf{x} - B\mathbf{a}\|_2 = \|U^t(\mathbf{x} - B\mathbf{a})\|_2 = \|U^t\mathbf{x} - SV^t\mathbf{a}\|_2.$$

Let

$$\mathbf{c} = U^t\mathbf{x} = \begin{pmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{pmatrix} \text{ and } \mathbf{y} = V^t\mathbf{a} = \begin{pmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{pmatrix}, \text{ where } \mathbf{c}_1, \mathbf{y}_1 \in \mathbb{R}^r.$$

Then

$$\begin{aligned}\min_{\mathbf{a}} \|\mathbf{x} - B\mathbf{a}\|_2^2 &= \min_{\mathbf{y}} \left\| \begin{pmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{pmatrix} - \begin{pmatrix} W & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{pmatrix} \right\|_2^2 \\ &= \min_{\mathbf{y}} \left\| \begin{pmatrix} \mathbf{c}_1 - W\mathbf{y}_1 \\ \mathbf{c}_2 \end{pmatrix} \right\|_2^2 \\ &= \min_{\mathbf{y}} \{ \|\mathbf{c}_1 - W\mathbf{y}_1\|_2^2 + \|\mathbf{c}_2\|_2^2 \}.\end{aligned}$$

This minimum clearly occurs when

$$\mathbf{c}_1 - W\mathbf{y}_1 = \mathbf{0}. \quad (4.25)$$

Since

$$W = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_r \end{bmatrix} \quad (4.25) \text{ is equivalent to the condition } \mathbf{y}_{1,i} = \frac{\mathbf{c}_{1,i}}{\sigma_i} \text{ for } 1 \leq i \leq r.$$

Case $r = n$: This the unique solution.

Case $r < n$: \mathbf{y}_2 does not occur in (4.25) and thus it has no effect on the residual. So the components of \mathbf{y}_2 can be chosen arbitrarily. However, there is a unique least squares solution with $\|\mathbf{y}\|_2$ minimized. This occurs when the $\mathbf{y}_2 = \mathbf{0}$. Note that $\mathbf{a} = V\mathbf{y}$ and $\|\mathbf{a}\|_2 = \|\mathbf{y}\|_2$. Thus $\|\mathbf{a}\|_2$ is minimized if and only if $\|\mathbf{y}\|_2$ is minimized. So setting $\mathbf{y}_2 = \mathbf{0}$ we get the unique least squares solution of Problem 4.15 with $\|\mathbf{a}\|_2$ minimized. We call it *minimum norm solution*.

If a matrix B has k "large" singular values and all of the other singular values are "small", then we say that the matrix B has *numerical rank* k . That is, given some tolerance ε , if

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_k \geq \varepsilon \geq \cdots \quad (4.26)$$

then we say that B has numerical rank k .

Example 4.19. *Let*

$$B = \begin{pmatrix} 1/3 & 1/3 & 2/3 \\ 2/3 & 2/3 & 4/3 \\ 1/3 & 2/3 & 3/3 \\ 2/5 & 2/5 & 4/5 \\ 3/5 & 1/5 & 4/5 \end{pmatrix}.$$

Clearly, B has rank 2 since the last column is the sum of the first two columns. But MATLAB, for example, computes its singular values as

$$\sigma_1 = 2.5987 \quad \sigma_2 = 0.3682 \quad \sigma_3 = 8.6614 \times 10^{-17}.$$

Since there are 3 non-zero singular values, we can conclude that the rank of B is 3. However, one of the singular values is extremely small and is non-zero only because of roundoff errors.

So in the above computations it is best to use for r the numerical rank of B . All "small" singular values should be set to 0, right after decomposing the matrix and before calculating the least squares solution.

The algorithm for finding the solution of Problem 4.15 can be stated as follows:

Algorithm 4.4 Algorithm for the Least Squares Problem

INPUT: $B \in \mathbb{R}^{n \times k}$, $\mathbf{x} \in \mathbb{R}^n$

OUTPUT: $\mathbf{a} \in \mathbb{R}^k : \mathbf{a} := \{ \mathbf{z} \in \mathbb{R}^n : \|\mathbf{x} - B\mathbf{z}\|_2 \text{ is minimal} \}$

if $n < k$ **then**

 Append $k - n$ zero rows to B and $k - n$ zero entries to \mathbf{x} .

end if

$r = \text{rank}(B)$

$\{U, S, V\} = \text{SINGULAR_VALUE_DECOMPOSITION}(B)$

Zero all "small" singular values.

$\mathbf{c} = U^t \mathbf{x}$

$\mathbf{y} \in \mathbb{R}^n$, $\mathbf{y} = \mathbf{0}$

for $i = 1, \dots, r$ **do**

$$\mathbf{y}_i = \frac{\mathbf{c}_i}{S_{i,i}}$$

end for

$\mathbf{a} = V\mathbf{y}$

return \mathbf{a}

4.3.2 The case $p > 2$

If $p > 2$, then the system of equations (4.23) is no longer linear in the components of \mathbf{a} , and thus a solution cannot generally be obtained in a finite number of operations. Since $p > 2$, $|r_i|^{p-2}$ exists for $1 \leq i \leq m$. So we can define the diagonal matrix D as

$$D = \begin{pmatrix} |f_1|^{p-2} & & & \\ & |f_2|^{p-2} & & \\ & & \ddots & \\ & & & |f_n|^{p-2} \end{pmatrix}.$$

Then the system (4.23) can be rewritten as

$$B^t D f = \mathbf{0} \text{ or as } B^t D B \mathbf{a} = B^t D \mathbf{x}. \quad (4.27)$$

This system of normal equations differs from equations (4.24) by the presence of the weighting matrix D . The following algorithm for solving (4.27) is a variant of the Newton's method.

Now consider the system of equations (4.27) as

$$g(\mathbf{a}) = \mathbf{0}$$

Then in order to apply the Newton's method to this system we have to determine for each iteration i the increment \mathbf{d}_i , which has to be added to the approximation \mathbf{a}_i . The increment d_i is given from

$$\nabla g(\mathbf{a}_i)\mathbf{d}_i = -g(\mathbf{a}_i)$$

or equivalently from

$$(p-1)B^t D_i B \mathbf{d}_i = B^t D_i f(\mathbf{a}_i), \quad (4.28)$$

where

$$D_i = \begin{pmatrix} |f_1(\mathbf{a}_i)|^{p-2} & & & \\ & |f_2(\mathbf{a}_i)|^{p-2} & & \\ & & \ddots & \\ & & & |f_n(\mathbf{a}_i)|^{p-2} \end{pmatrix}.$$

We can obtain the \mathbf{d}_i through the solution of the problem of finding the vector $\mathbf{z} \in \mathbb{R}^n$ to minimize the $\|\cdot\|_2$ -norm of

$$\mathbf{s}_i = D_i^{\frac{1}{2}} \mathbf{x} - D_i^{\frac{1}{2}} A \mathbf{z}.$$

Let \mathbf{z}_i minimizes $\|\mathbf{s}_i\|_2$. Then \mathbf{z}_i satisfies

$$B^t D_i B \mathbf{z}_i = B^t D_i \mathbf{x} = B^t D_i (f(\mathbf{a}_i) + B \mathbf{a}_i) = B^t D_i f(\mathbf{a}_i) + B^t D_i B \mathbf{a}_i.$$

Thus

$$B^t D_i B (\mathbf{z}_i - \mathbf{a}_i) = B^t D_i f(\mathbf{a}_i).$$

So it follows that we can take

$$\mathbf{d}_i = \frac{1}{p-1} (\mathbf{z}_i - \mathbf{a}_i).$$

After these considerations we present the algorithm for solving (4.27) as follows:

Algorithm 4.5 Algorithm for the Linear l_p Approximation Problem

INPUT: $B \in \mathbb{R}^{n \times k}$, $\mathbf{x} \in \mathbb{R}^n$, $p \in \mathbb{Z}$, $p > 2$

OUTPUT: $\mathbf{a} \in \mathbb{R}^k$: $\mathbf{a} = \{\mathbf{z} \in \mathbb{R}^k : \|\mathbf{x} - B\mathbf{z}\|_p \text{ is minimal}\}$

$i = 0$, $D_0 = Id(n)$

while true **do**

$\mathbf{z}_i = \text{LEAST_SQUARES}(D_i^{\frac{1}{2}} B, D_i^{\frac{1}{2}} \mathbf{x})$

if $i == 0$ **then**

$\mathbf{a}_0 = \mathbf{z}_0$

end if

$\mathbf{a}_{i+1} = \frac{p-2}{p-1} \mathbf{a}_i + \frac{1}{p-1} \mathbf{z}_i$

if the iteration \mathbf{a}_i has converged **then**

return \mathbf{a}_{i+1}

else

$i = i + 1$

end if

end while

Finally, using (4.22), we can describe the algorithm for the calculating of the i -th distance function as follows:

Algorithm 4.6 Algorithm for the Calculation of the Distance Functions

INPUT: $i \in \mathbb{Z}$, $B \in \mathbb{R}^{n \times (i-1)}$ (constructed from the basis vectors $\mathbf{b}_1, \dots, \mathbf{b}_{i-1}$ as in 4.22), $\mathbf{x} \in \mathbb{R}^n$, $p \in \mathbb{Z}$, $p > 2$

OUTPUT: The value of $F_i(x)$ and the corresponding coefficients $\xi_1, \dots, \xi_{i-1} \in \mathbb{R}$

if $i == 1$ **then**

return $\|x\|_p$ and an empty vector ξ

else

$\xi = (\xi_1, \dots, \xi_{i-1}) = \text{NEWTON}(B, \mathbf{x}, p)$

end if

return $\|\mathbf{x} + B\xi\|_p$ and ξ

Chapter 5

Tests and Results

In this chapter we present the test cases with which we tested the algorithms, described in Chapter 3 and Chapter 4, and the corresponding results of these tests. For the case $p = 2$ we used the implementation of the L^3 and block basis reduction algorithm provided by the NTL library[NTL]. The success rate of these tests was 0% even for Goppa codes of length 15. So till the end in this chapter we present the test results from the implementation of the LS and the block basis reduction algorithms for $p > 2$, given in the Appendix. All tests were executed on *Athlon XP 1800+* with *512MB RAM* under *SUSE Linux 9.2*.

The following test results showed that the time, used by the reduction algorithms, increases considerably and the success probability decreases with increasing dimension of the input basis vectors. For Goppa codes of length 15 the success rate was 100% and the reduction algorithms used not more than 2 sec. in order to output the error vector. Our tests have shown that the attack becomes impractical for Goppa codes of length 127, since the success rate remained 0%. The tests showed that increasing p has influence only on the execution time of the algorithms, but not on the success rate. They also showed that in most test cases the block basis reduction does not provide better results than the LS reduction algorithm. For these reasons in this chapter we give mainly the results from the LS reduction algorithm for Goppa codes of length 31 and 63. Before presenting the test cases and results we will briefly describe the different constructions of the input matrices.

Let G' be an $k \times n$ generator matrix for binary irreducible Goppa code \mathcal{C} with minimal distance d , which can correct up to t errors, $\mathbf{m} \in \mathbb{F}_2^k$ and $\mathbf{e} \in \mathbb{F}_2^n$ be an error vector with weight smaller or equal to t . Then we choose either to construct a matrix $G := SG'P$, as in the McEliece public key, or to use directly $G := G'$. Let $\mathbf{c} = \mathbf{m}G \oplus \mathbf{e}$, where $G = SG'P$ or $G := G'$. Then we have the following possibilities for the construction of \tilde{G} (see (2.3))

$$\tilde{G}_1 := \begin{pmatrix} \mathbf{c} \\ G \end{pmatrix} \text{ or } \tilde{G}_2 := \begin{pmatrix} G \\ \mathbf{c} \end{pmatrix}. \quad (5.1)$$

In the following tables we will refer to the construction of G_1 as *code_mode=0* and to the construction of G_2 as *code_mode=1*. Using gaussian elimination and column permutation we find matrices $R \in \mathbb{F}_2^{(k+1) \times (k+1)}$, \tilde{G}^{sys_i} in systematic form and $W \in \mathbb{F}_2^{n \times n}$, such that

$$R \cdot \tilde{G}^{\text{sys}_i} \cdot W = \tilde{G}_i,$$

for $i = 1$ or $i = 2$. Then we fix a map σ and apply $\sigma : \mathbb{F}_2^n \rightarrow \mathbb{Z}^n$ to the rows of \tilde{G}^{sys_i} . For the map σ we choose either the map $\sigma_{\text{unsigned}} : \mathbb{F}_2^n \rightarrow \mathbb{Z}^n$, which maps 0 to 0 and 1 to 1, or a map $\sigma_{\text{signed}} : \mathbb{F}_2^n \rightarrow \mathbb{Z}^n$, such that

$$\begin{pmatrix} \sigma_{\text{signed}}(\mathbf{g}_1^{\text{sys}_i}) \\ \vdots \\ \sigma_{\text{signed}}(\mathbf{g}_{k+1}^{\text{sys}_i}) \end{pmatrix} = [\text{Id}(k+1) | D_i],$$

with $D_i \in \{-1, 0, 1\}^{(k+1) \times (n-k-1)}$ and

$$d_{l,s} = \begin{cases} 0 & \text{if } g_{l,s+k+1}^{\text{sys}_i} = 0, \\ 1 \text{ or } -1 & \text{if } g_{l,s+k+1}^{\text{sys}_i} = 1. \end{cases}$$

Then we consider the following constructions of the input matrix B^{in}

$$B_{i,1}^{\text{in}} = \begin{pmatrix} \mathbf{0} & 2 \cdot \text{Id}(n-k-1) \\ \text{Id}(k+1) & D_i \end{pmatrix} \text{ or } B_{i,2}^{\text{in}} = \begin{pmatrix} \text{Id}(k+1) & D_i \\ \mathbf{0} & 2 \cdot \text{Id}(n-k-1) \end{pmatrix} \quad (5.2)$$

In the following tables we will refer to the construction of $B_{i,1}$ as $id_mode=0$ and to the construction of $B_{i,2}$ as $id_mode=1$.

We can categorize the different test parameters as parameters defining the different constructions of the input matrix, reduction parameters and parameters of the Goppa codes. The parameters for the different constructions of the input matrix are:

- *id_mode* - specifies whether the input matrix B^{in} is constructed as $B_{i,1}^{\text{in}}$ or as $B_{i,2}^{\text{in}}$ (see (5.2)).
- *sigma_mode* - specifies whether the input matrix B^{in} is constructed with σ_{unsigned} or σ_{signed} .
- *code_mode* - specifies whether the input matrix B^{in} is constructed with \tilde{G}_1 or with \tilde{G}_1 (see (5.1)).
- *g_mode* - specifies whether the input matrix B^{in} was constructed with G' or with $SG'P$.

The reduction parameters are:

- p - defines the norm to be used. By all test cases, except one, we test with $p = 3$ and use Goppa codes which are suitable (according to Lemma 2.6) for $p = 3$.
- δ - the parameter $0 < \delta < 1$ of the reduction algorithms. Since δ determines the quality of the approximation and with smaller δ the LS reduction provides weaker approximation for all the test cases we set $\delta = 0.99$.
- β - the block size for the block basis reduction.

The parameters of the Goppa codes are:

- n - the length of the Goppa code.
- k - the dimension of the Goppa code.
- d - the minimal distance of the Goppa code.
- t - the number of errors, which the Goppa code can correct. For all the test cases, except one, we test with error vectors, which have maximal weight t . Only in test case 6 we test with error vectors with weight smaller than the maximal possible t for the corresponding Goppa code.

In the first three test cases we test with different construction parameters for the input matrices. In the other test cases we fix the input matrix according to the best results shown in the first three test cases and test with different reduction parameters and different parameters of the Goppa code.

Test Case 1: G' vs $SG'P$

In these tests we used Goppa codes, for which $k \approx \frac{n}{2}$. In the following tables we give the success rate and the time, used by the LS reduction algorithm, for $p = 3$ with the following construction parameters for the input matrices:

- $id_mode = 0$ with $code_mode = 0$, σ_{signed} (Table 1),
- $id_mode = 1$ with $code_mode = 0$, σ_{unsigned} (Table 2).

Table 1: $id_mode = 0$ with $code_mode = 0$, σ_{signed}

n	k	d	t	# Tests	G'		$SG'P$	
					# Found	Av. Time(sec)	# Found	Av. Time(sec)
31	16	7	3	25	14	4.6s	15	5s
63	33	11	5	25	4	95s	3	105s

Table 2: $id_mode = 1$ with $code_mode = 0$, σ_{unsigned}

n	k	d	t	# Tests	G		SGP	
					# Found	Av. Time(sec)	# Found	Av. Time(sec)
31	16	7	3	25	22	17.7s	16	19.3s
63	33	11	5	25	12	1874.3s	13	1892.4s

Test Case 2: σ_{unsigned} vs σ_{signed}

In these tests we used Goppa codes, for which $k \approx \frac{n}{2}$. In the following tables we give the success rate and the time, used by the LS reduction algorithm, for $p = 3$ with the following construction parameters for the input matrices:

- $id_mode = 0$ with $code_mode = 0$, $g_mode = G'$ (Table 1),
- $id_mode = 1$ with $code_mode = 0$, $g_mode = G'$ (Table 2).

Table 1: $id_mode = 0$ with $code_mode = 0$, $g_mode = G'$

n	k	d	t	# Tests	σ_{signed}		σ_{unsigned}	
					# Found	Av. Time(sec)	# Found	Av. Time(sec)
31	16	7	3	25	16	4.8s	15	4.8s
63	33	11	5	25	5	94.3s	4	98s

Table 2: $id_mode = 1$ with $code_mode = 0$, $g_mode = G'$

n	k	d	t	# Tests	σ_{signed}		σ_{unsigned}	
					# Found	Av. Time(sec)	# Found	Av. Time(sec)
31	16	7	3	25	17	35.8s	21	18s
63	33	11	5	25	7	3248s	13	1622s

Test Case 3: $code_mode = 0$ vs $code_mode = 1$

In these tests we used Goppa codes, for which $k \approx \frac{n}{2}$. In the following tables we give the success rate and the time, used by the LS reduction algorithm, for $p = 3$ with the following construction parameters for the input matrices:

- $id_mode = 0$ with σ_{signed} , $g_mode = G'$ (Table 1),
- $id_mode = 1$ with σ_{unsigned} , $g_mode = G'$ (Table 2).

Table 1: $id_mode = 0$ with σ_{signed} , $g_mode = G'$

n	k	d	t	# Tests	$code_mode = 0$		$code_mode = 1$	
					# Found	Av. Time(sec)	# Found	Av. Time(sec)
31	16	7	3	25	15	5s	13	5.1s
63	33	11	5	25	6	98s	6	105s

Table 2: $id_mode = 1$ with σ_{unsigned} , $g_mode = G'$

n	k	d	t	# Tests	$code_mode = 0$		$code_mode = 1$	
					# Found	Av. Time(sec)	# Found	Av. Time(sec)
31	16	7	3	25	21	14.4s	21	15s
63	33	11	5	25	16	1569s	16	1527s

Test Case 4: $p = 3$, $d = 15$ vs $p = 3$, $d = 17$ vs $p = 4$, $d = 15$ vs $p = 4$, $d = 17$

In this test case we tested what is the success rate and the time, used by the LS reduction algorithm, for suitable Goppa codes (according to Lemma 2.6) for $p = 3$ and $p = 4$. We use the following construction parameters for the input matrices:

- $id_mode = 0$ with $code_mode = 0$, σ_{signed} , $g_mode = G'$ (Table 1),
- $id_mode = 1$ with $code_mode = 0$, σ_{unsigned} , $g_mode = G'$ (Table 2).

Table 1: $id_mode = 0$ with $code_mode = 0$, σ_{signed} , $g_mode = G'$

n	k	d	t	# Tests	$p = 3$		$p = 4$	
					# Found	Av. Time(sec)	# Found	Av. Time(sec)
63	21	15	7	25	7	148s	6	526s
63	15	17	8	25	14	227s	18	809s

Table 2: $id_mode = 1$ with $code_mode = 0$, $\sigma_{unsigned}$, $g_mode = G'$

n	k	d	t	# Tests	$p = 3$	
					# Found	Av. Time(sec)
63	21	15	7	25	11	1554.5s
63	15	17	8	25	22	1365.2s

Test Case 5: Different k and Different d

In these tests we used Goppa codes of the same length n , which have different dimensions k and can correct different number of errors t . In the following tables we give the success rate and the time, used by the LS reduction algorithm, for $p = 3$ with the following construction parameters for the input matrices:

- $id_mode = 0$ with $code_mode = 0$, σ_{signed} , $g_mode = G'$ (Table 1: $n = 31$, Table2: $n = 63$),
- $id_mode = 1$ with $code_mode = 0$, $\sigma_{unsigned}$, $g_mode = G'$ (Table 3: $n = 31$, Table4: $n = 63$).

Table 1: $n = 31$, $id_mode = 0$ with $code_mode = 0$, σ_{signed} , $g_mode = G'$

n	k	d	t	# Tests	# Found	Av. Time(sec)
31	26	5	2	5	2	12s
31	21	5	2	5	4	1.8s
31	21	7	3	5	1	2.4s
31	16	7	3	5	2	4s
31	11	9	4	5	1	8.8s

Table 2: $n = 63$, $id_mode = 0$ with $code_mode = 0$, σ_{signed} , $g_mode = G'$

n	k	d	t	# Tests	# Found	Av. Time(sec)
63	39	9	4	5	1	59s
63	33	11	5	5	1	80.2s
63	33	13	6	5	0	148s
63	27	13	6	5	0	117s
63	27	15	7	5	0	165s
63	21	15	7	5	1	101.8s

Table 3: $n = 31$ $id_mode = 1$ with $code_mode = 0$, σ_{unsigned} , $g_mode = G'$

n	k	d	t	# Tests	# Found	Av. Time(sec)
31	26	5	2	5	2	22.6s
31	21	5	2	5	4	21.6s
31	21	7	3	5	4	18.6s
31	16	7	3	5	5	5s
31	11	9	4	5	5	20s

Table 4: $n = 63$ $id_mode = 1$ with $code_mode = 0$, σ_{unsigned} , $g_mode = G'$

n	k	d	t	# Tests	# Found	Av. Time(sec)
63	39	9	4	5	4	1342.4s
63	33	11	5	5	3	1481.2s
63	33	13	6	5	1	2474.3s
63	27	13	6	5	1	2593.2s
63	27	15	7	5	0	3151.2s
63	21	15	7	5	3	1542.6s

Test Case 6: Error Vectors with Smaller than the Maximal Possible Weight

In these tests we used Goppa codes, for which $k \approx \frac{n}{2}$ and with error vectors with smaller than the maximal possible weight. The column t corresponds to the maximal number of errors, which the corresponding Goppa code can correct and the column t_1 corresponds to the weight of the error vector, used in the construction of input matrix. In the following tables we give the success rate and the time, used by the LS reduction algorithm for $p = 3$, with the following construction parameters for the input matrices:

- $id_mode = 0$ with $code_mode = 0$, σ_{signed} , $g_mode = G'$ (Table 1),
- $id_mode = 1$ with $code_mode = 0$, σ_{unsigned} , $g_mode = G'$ (Table 2).

Table 1: $id_mode = 0$ with $code_mode = 0$, σ_{signed} , $g_mode = G'$

n	k	d	t	t_1	# Tests	Found	Av. Time(sec)
31	16	7	3	1	10	10	5.5s
31	16	7	3	2	10	7	4.6s
31	16	7	3	3	10	5	5.7s
63	33	11	5	1	10	10	93.6s
63	33	11	5	2	10	6	73.9s
63	33	11	5	3	10	5	88.4s
63	33	11	5	4	10	2	94.4s
63	33	11	5	5	10	1	89.8s

Table 2: $id_mode = 1$ with $code_mode = 0$, σ_{unsigned} , $g_mode = G'$

n	k	d	t	t_1	# Tests	Found	Av.Time(sec)
31	16	7	3	1	10	10	52.6s
31	16	7	3	2	10	10	17.7s
31	16	7	3	3	10	9	16.1s
63	33	11	5	1	10	10	489.3s
63	33	11	5	2	10	10	500s
63	33	11	5	3	10	8	682.8s
63	33	11	5	4	10	7	1437.7s
63	33	11	5	5	10	5	1865s

Test Case 7: Block Basis Reduction Algorithm

In these tests we used Goppa codes, for which $k \approx \frac{n}{2}$ and input matrices, for which the LS reduction algorithm has not succeeded to output the error vector. We give the success rate and the time, used by the block basis reduction algorithm for $p = 3$ and different block sizes. We use the following construction parameters for the input matrices:

- $id_mode = 1$ with $code_mode = 0$, σ_{unsigned} , $g_mode = G'$ (Table1).

For this test we have chosen exactly these construction parameters for the input matrices since with them we have achieved the best success results with the LS reduction for all other test cases.

Table 1: $id_mode = 1$ with $code_mode = 0$, σ_{unsigned} , $g_mode = G'$

n	k	d	t	β	# Tests	Found	Av.Time(sec)
31	16	7	3	4	10	0	88.3s
31	16	7	3	8	10	0	136.1s
31	16	7	3	12	10	0	208s
31	16	7	3	16	10	0	391.5s
63	33	11	5	4	10	1	3124.5s
63	33	11	5	8	10	1	4053.7s
63	33	11	5	12	10	1	5164.2s
63	33	11	5	16	10	1	7572.9s

Chapter 6

Implementation Details

We provide an implementation in C of the LS basis reduction and the block basis reduction algorithms w.r.t the l_p -norm, which uses the GNU Scientific Library [GSL]. The source code of the basis reduction algorithms can be found in Section A.1. In order to check that our implementation works correctly we use a small program, which uses NTL.

6.1 Lattice Basis Reduction Algorithms

The source code for the block basis reduction algorithm (see Algorithm 4.2) can be found in Section A.1.2.

It uses the subroutines *ls_reduction* (see Algorithm 4.1), *adfs* (see Algorithm 4.3) and *basis_transformation* (see Algorithm 3.4). *adfs* uses the subroutine *find_min_mu*. For given integers t and s and a vector $\tilde{\mathbf{u}}$, *find_min_mu* calculates the real minimum μ of $F_t(\mu \mathbf{b}_t + \sum_{i=t+1}^s \tilde{u}_i \mathbf{b}_i)$ and depending on whether $F_t(\lfloor z \rfloor \mathbf{b}_t + \sum_{i=t+1}^s \tilde{u}_i \mathbf{b}_i)$ or $F_t((\lfloor \mu \rfloor + 1) \mathbf{b}_t + \sum_{i=t+1}^s \tilde{u}_i \mathbf{b}_i)$ is smaller sets correspondingly $\tilde{u}_t := \lfloor \mu \rfloor$ or $\tilde{u}_t := \lfloor z \rfloor + 1$ (as described in Algorithm 4.3).

At the end of each iteration of the block basis reduction algorithm the vectors

$$\mathbf{b}_1, \dots, \mathbf{b}_{\min\{k+1, m\}}$$

are LS reduced. If at stage z of the block basis reduction algorithm the *adfs* finds a shorter vector in $\mathcal{L}(\mathbf{b}_j, \dots, \mathbf{b}_k)$, then the *basis_transformation* function modifies only some the vectors $\mathbf{b}_j, \dots, \mathbf{b}_k$. Therefore the vectors $\mathbf{b}_1, \dots, \mathbf{b}_{j-1}$ remain LS reduced and the LS reduction for the current iteration can start from index j , instead of starting from index 1.

If *adfs* fails to find a shorter vector, the vectors $\mathbf{b}_1, \dots, \mathbf{b}_k$ remain LS reduced and the LS reduction for the current iteration can start from index $\min\{k+1, m\}$. For this reason the *ls_reduction* function accepts an additional integer *start_index* as parameter, which specifies from which index to start the reduction of the given basis vectors.

The source code for the LS basis reduction algorithm can be found in Section A.1.1. Since we use the basis reduction algorithms for finding a basis vector with a predetermined l_p norm, it makes sense to stop the execution of the algorithm if such a vector is found. Hence the *ls_reduction* function takes an additional integer t as parameter and returns if a basis vector $\mathbf{b}_i \notin \pm \text{Id}(n)$ with l_p norm $t^{1/p}$ is found. For the size-reduction of the basis vectors *ls_reduction* uses the subroutine *find_min_mu*, which calculates, for

given j and k , an integer μ_j , such that $F_j(\mathbf{b}_k + \mu\mathbf{b}_j)$ is minimal.

The source code for the algorithm for the calculation of the values of the distance functions, described in Section 4.3, can be found in Section A.1.3. To calculate

$$F_j(\mathbf{b}_1, \dots, \mathbf{b}_{j-1})(\mathbf{v}) = \min_{\alpha_1, \dots, \alpha_{j-1} \in \mathbb{R}} \left\| \mathbf{v} + \sum_{i=1}^{j-1} \alpha_i \mathbf{b}_i \right\|_p$$

the function *p_norm* (see Algorithm 4.6) is called. In order to avoid calculating the p -th root of a real number *p_norm* always returns $F_j(\mathbf{v})^p$. If $j = 1$ *p_norm* returns $\|\mathbf{v}\|_p^p$. Otherwise it uses the *newton_method* function (see Algorithm 4.5), which calculates the coefficients $\alpha_1, \dots, \alpha_{j-1}$. In order to determine these coefficients *newton_method* uses the *least_squares* function, which solves the least squares problem (see Algorithm 4.4). The function *least_squares* uses the following built-in functions from the GSL library

```
int gsl_linalg_SV_decomp (gsl_matrix * A, gsl_matrix * V, gsl_vector * S, gsl_vector * work)
int gsl_linalg_SV_solve (gsl_matrix * U, gsl_matrix * V, gsl_vector * S, const gsl_vector * b,
                        gsl_vector * x).
```

For the floating point arithmetic we use the standard type "double" in C. The "double" type uses the standard representation of floating point numbers with double precision according to the IEEE-754 standard. For the comparison of the real numbers we use precision 10^{-15} .

The code for several small help functions, which are commonly used, is given in Section A.1.4.

6.2 Test Program for the Basis Reduction Algorithms

In order to be able to test the lattice basis reduction algorithms we have written a program *test_lattice_reduction*, which reads the matrix to be reduced and the required basis reduction parameters from an input file and saves the result of the reduction in an output file. We run *test_lattice_reduction* with the following command line parameters

```
./test_lattice_reduction < in_file > < out_file >
```

in_file the path to the input file, which contains the following data

```
algo integer specifying the basis reduction algorithm, which has to be used
      algo = 0 - test_lattice_reduction uses the LS basis reduction algorithm
      algo = 1 - test_lattice_reduction uses the block basis reduction algorithm
p integer specifying the norm to be used
δ the basis reduction parameter  $\frac{1}{4} < \delta < 1$ 
β the block size for the block basis reduction algorithm
    (β is given only if algo = 1)
n the number of columns and rows of the matrix  $B^{in}$ 
Bin the  $n \times n$  matrix to be reduced
e the random generated error vector with weight  $t$ , used in the generation of  $B^{in}$ 
```

out_file the path to the output file. The output file contains the time used for the reduction of B^{in} , the reduced basis B^{out} and whether during the reduction of B^{in} a vector

\mathbf{v} was found, such that $\|v\|_p^p = t$.

After the reduction of B^{in} the *test_lattice_reduction* program tests whether the used basis reduction algorithm works correctly and does not change the input lattice. First, if a vector \mathbf{v} , such that $\|v\|_p^p = t$, was found it is compared with the original error vector \mathbf{e} and if $\mathbf{v} \neq \mathbf{e} \pmod 2$ then an additional error message is written in the output file. Second, it checks whether the vectors from B^{out} are contained in the lattice, generated by the rows of B^{in} and vice versa. For this test, it uses the built-in function from the NTL library

```
long LatticeSolve( vec_ZZ &x, const mat_ZZ &A, const vec_ZZ &y ),
```

which tests if for a given matrix A and a vector \mathbf{y} , there exists a vector \mathbf{x} such that $\mathbf{x} \cdot A = \mathbf{y}$. Since we know that the matrix B^{in} has full rank, then if this test was successful, we can be sure that the input lattice has not been changed during the execution of the basis reduction algorithms. If this test was not successful a corresponding error message is printed in the output file. The source code for the *test_lattice_reduction* program can be found in Section A.2.

Sample Output File

At the end of the next subsection we give a sample output file *test.in* from the program for generating test input files. Giving *test.in* as an input file to the *test_lattice_reduction* program we get the following output file:

```
RUNNING TEST ...

OUTPUT:
Time used from the algorithm: 5 seconds, 825874 microseconds

FOUND in row 16
Original Error Vector:

0 0 0 1 0 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 1 0 0 0 0 0 0
The reduced matrix is:

0 -1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 1 0 -1 0 0 0 0
0 0 0 0 0 0 0 0 0 1 -1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 -1 0 1 -1 0 0 0 0
-1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 1 1 0 0 0 0 0 -1 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 0 1 0 0 1 0 1 0 0 0
-1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 -1 -1 0 1 0
0 0 0 0 0 -1 0 0 0 1 0 0 0 0 0 0 0 0 -1 1 0 0 0 0 0 -1 0 1 -1 0 0 0
-1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 1 1 -1 -1 0 0 0 0
0 0 0 -1 0 0 0 1 0 0 0 0 0 0 0 0 0 -1 0 0 -1 1 0 0 1 0 0 -1 0 0 1
0 -1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 -1 -1 0 -1
0 -1 0 -1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 -1 0 -1 0 0 0 0 1 0 0 -1 0 0 -1
1 0 0 0 0 0 1 0 0 0 0 0 0 0 -1 0 0 1 0 -1 0 -1 1 0 -1 0 1 0 0 0 0
0 0 0 -1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 -1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 -1 0 0 0 -1 -1 0 0 0 -1 0 0 0 0 -1 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 1 1 0 0 0 1 1 0 1 0 0 0 1
-1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 -1 0 1 0 0 -1 0 0 1
0 0 0 -1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 -1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 1 0 1 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0
```

```

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2

```

TEST 1: OUTPUT VECTORS ARE IN THE LATTICE GENERATED BY THE ROWS OF THE INPUT MATRIX: TRUE
TEST 2: INPUT VECTORS ARE IN THE LATTICE GENERATED BY THE ROWS OF THE OUTPUT MATRIX: TRUE

6.3 Generating the Test Input Files

The generator matrices G' for the Goppa codes are generated with MAGMA - a software for Algebra, Number Theory, and Geometry. For more information see

<http://magma.maths.usyd.edu.au/>.

In order to generate the input file for the lattice basis reduction algorithms we have written a program *generate_test_file*. The input file for the *generate_test_file* program contains the following data:

- algo* integer specifying the basis reduction algorithm, for which the test file is generated
 - algo* = 0 - generates a test file for the LS basis reduction algorithm
 - algo* = 1 - generates a test file for the block basis reduction algorithm
- δ the basis reduction parameter $\frac{1}{4} < \delta < 1$
- β the block size for the block basis reduction algorithm
(β is given only if *algo* = 1)
- p* integer specifying the norm to be used
- k* number of rows in the Goppa generation matrix G'
- n* number of columns in the Goppa generation matrix G'
- G' a $k \times n$ generator matrix for a binary irreducible Goppa code, which can correct up to *t* errors
- t_1 the weight of the error vector ($0 < t_1 \leq t$), which has to be generated

First a random message vector $\mathbf{m} \in \mathbb{F}_2^k$, a random error vector $\mathbf{e} \in \mathbb{F}_2^n$ with weight t_1 . Then, depending on the command line parameters, either $G = G'$ or the matrix $G = SG'P$ is constructed, where $S \in \mathbb{F}_2^{k \times k}$ is a random regular matrix and P is a random permutation matrix. After that a ciphertext $\mathbf{c} = \mathbf{m}G \oplus \mathbf{e}$ is calculated. Depending on the command line parameters, one of the matrices

$$\tilde{G}_1 = \begin{pmatrix} G \\ \mathbf{c} \end{pmatrix} = \begin{pmatrix} G \\ \mathbf{m}G \oplus \mathbf{e} \end{pmatrix} \quad \text{or} \quad \tilde{G}_2 = \begin{pmatrix} \mathbf{c} \\ G \end{pmatrix} = \begin{pmatrix} \mathbf{m}G \oplus \mathbf{e} \\ G \end{pmatrix}.$$

is constructed. Using Gaussian elimination and column permutation, *generate_test_file* finds a binary matrix $R \in \mathbb{F}_2^{(k+1) \times (k+1)}$, a systematic matrix \tilde{G}^{sys} and a permutation matrix $W \in \mathbb{F}_2^{n \times n}$, such that

$$R \cdot \tilde{G}^{\text{sys}} \cdot W = \tilde{G},$$

where $\tilde{G} = \tilde{G}_1$ or $\tilde{G} = \tilde{G}_2$ and

$$\tilde{G}^{\text{sys}} = [\text{Id}(k+1) \mid A].$$

Then a vector \mathbf{e}_{perm} is constructed, by permuting the entries of \mathbf{e} according to W^{-1} . After that a map σ is applied on \tilde{G}^{sys} , where σ , depending on the specified command line parameters, is either $\text{id}_{\mathbb{Z}^n}$ or a map, such that $\sigma(\tilde{G}^{\text{sys}}) = [\text{Id}_{k+1} \mid A] := M$, where $A \in \{-1, 0, 1\}^{k \times n-k-1}$ and

$$m_{i,j} = \begin{cases} 0 & \text{if } g_{i,j}^{\text{sys}} = 0 \text{ and } 1 \leq i \leq k+1, 1 \leq j \leq n, \\ 1 & \text{if } g_{i,j}^{\text{sys}} = 1 \text{ and } 1 \leq i \leq k+1, 1 \leq j \leq n, \\ 1 \text{ or } -1 & \text{if } g^{\text{sys}} s_{i,j} = 1 \text{ and } 1 \leq i \leq k+1, k+1 \leq j \leq n. \end{cases} \quad (6.1)$$

Then one of the matrices

$$B_1^{\text{in}} = \begin{pmatrix} \sigma(\mathbf{g}_1^{\text{sys}}) \\ \vdots \\ \sigma(\mathbf{g}_{k+1}^{\text{sys}}) \\ \mathbf{0} \quad 2 \cdot \text{Id}(n-k-1) \end{pmatrix} \text{ or } B_2^{\text{in}} = \begin{pmatrix} \mathbf{0} \quad 2 \cdot \text{Id}(n-k-1) \\ \sigma(\mathbf{g}_1^{\text{sys}}) \\ \vdots \\ \sigma(\mathbf{g}_{k+1}^{\text{sys}}) \end{pmatrix}.$$

is constructed. We run `./generate_test_file` with the following command line parameters

`./generate_test_file < in_file > < out_file > < cipher_mode > < id_mode >`
`< sigma_mode > < g_mode >`

`in_file` the path to the input file

`out_file` the path to the output file, which contains the following data

- `algo` integer specifying the basis reduction algorithm, which has to be used
 - `algo = 0` - `test_lattice_reduction` uses the LS basis reduction algorithm
 - `algo = 1` - `test_lattice_reduction` uses the block basis reduction algorithm
- `p` integer specifying the norm to be used
- `δ` the basis reduction parameter $\frac{1}{4} < \delta < 1$
- `β` the block size for the block basis reduction algorithm
(`β` is given only if `algo = 1`)
- `n` the number of columns and rows of the matrix B^{in}
- B^{in} the $n \times n$ matrix to be reduced
- \mathbf{e} the random generated error vector with weight t , used in the generation of B^{in}
- `cipher_mode` integer specifying where in \tilde{G} to insert the ciphertext \mathbf{c}
 - `cipher_mode = 0` - \mathbf{c} is inserted as the first row of \tilde{G}
 - `cipher_mode = 1` - \mathbf{c} is inserted as the last row of \tilde{G}
- `id_mode` integer specifying where in B^{in} to insert the rows $2\mathbf{e}_{k+1}, \dots, 2\mathbf{e}_n$
 - `id_mode = 0` - the vectors $2\mathbf{e}_{k+1}, \dots, 2\mathbf{e}_n$ are inserted as the first $n-k-1$ rows of B^{in}
 - `id_mode = 1` - the vectors $2\mathbf{e}_{k+1}, \dots, 2\mathbf{e}_n$ are inserted as the last $n-k-1$ rows of B^{in}
- `sigma_mode` integer specifying the map σ , which will be applied on \tilde{G}_{sys}
 - `sigma_mode = 0` - $\sigma = \text{id}_{\mathbb{Z}^n}$
 - `sigma_mode = 1` - σ is defined as in (6.1)
- `g_mode` integer specifying whether $G = G'$ or $G = SG'P$
 - `g_mode = 0` - $G = G'$
 - `g_mode = 1` - $G = SG'P$

The source code for the generation of the test input files can be found in Section A.3. The functions for generating random messages and random error vectors are given in Section A.3.3. The functions, which generate random regular binary matrices and random permutation matrices are given in Section A.3.4. The source code for the Gaussian Elimination Algorithm, described below, and the column permutation can be found in Section A.3.5. In the last Section of the Appendix are given several small help functions.

Description of the Gaussian Elimination Algorithm

Let M be an $m \times n$ binary matrix of rank m . The Gaussian Elimination algorithm transforms M into a matrix N , so that N contains the columns of the $m \times m$ identity matrix by means of the operations

- (O1) Addition of one row to another row.
- (O2) Swapping two rows.

At every step k ($1 \leq k \leq m$) the matrix M is transformed into a matrix M' , so that the first l ($1 \leq k < l \leq n$) columns of M' contain the first k columns of the $m \times m$ identity matrix. Suppose the algorithm is at step i and the first $j - 1$ columns contain the first $i - 1$ columns of the $m \times m$ identity matrix. Then the following cases are considered:

1. $m_{i,j} = 1$: set $\mathbf{m}_r := \mathbf{m}_i \oplus \mathbf{m}_r$ for all rows \mathbf{m}_r with index $r \neq i$ and $m_{r,j} = 1$.
2. $m_{i,j} = 0$:
 - (a) If there exists an index s ($i + 1 \leq s \leq m$), such that $m_{s,j} = 1$, swap the rows \mathbf{m}_i and \mathbf{m}_s and go to Case 1.
 - (b) If there doesn't exist an index s ($i + 1 \leq s \leq m$), such that $m_{s,j} = 1$, then set $j := j + 1$. If $m_{i,j} = 1$ go to Case 1, otherwise go to Case 2.

Sample Input File

Here we give a sample input file *test.txt* for the *generate_test_file* program. With the following input file we generate a test input *test.in* file for the LS reduction algorithm with $\delta = 0.99$, $p = 3$, 16×31 generator matrix for Goppa code and $t = 3$.

```
0
0.99
3
16
31
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 0 0 0 0 1 1 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 1 1 1 1 0 0 1 0 1 1 1
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0 1 0 0 1 0 1 1 1
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 1 0 0 1 0 1 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 0 1
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 1 1 1 0 0 1 0 1 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0 1 0 0 1 0 1 1 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 0 0 1 1 0 1 0 0 1
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1 0 1 1 0 0 0 1 0 1 0 1 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 1 1 1 0 0 1 0 0 1 0 0 1 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 1 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 1 1 1 0 1 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 1 0 1 1 1 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 0 0 1 0 1 1 1 1 0 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 1 0 1 0 1 0 0 0 0 1
3
```

Sample Output File

The following file *test.in* is generated with *generate_test_file* program with the command

```
> ./generate_test_file test.txt test.in 0 1 0 0
```

```
0
3
0.990000
31
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 0 1 1 1 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 1 0 0 1 1 1 1 1
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0 1 0 0 1 1 1 1
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 1 0 0 1 0 1 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 1 1 1 0 0 0 1
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 1 1 0 0 1 0 1 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0 1 0 0 1 0 1 1 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 1 1 0 1 1 0 0 1 0 0 0 1
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 1 0 1 1 0 0 1 0 0 1 0 0 1 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 1 1 1 0 0 1 0 0 1 0 0 1 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 1 0 1 0 1 1 1 0 1 0 1 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 0 1 1 0 1 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 1 1 1 1 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 1 1 0 1 1 1 1 0 1 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 0 1 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 0 1 0 1 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 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 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0
```


Bibliography

- [AKS01] M. Ajtai, R. Kumar, and D. Sivakumar. A sieve algorithm for the shortest lattice vector problem. *Proc. 33rd STOC*, pages 601–610, 2001.
- [Bay97] J. Baylis. *Error Correcting Codes: A Mathematical Introduction*. Chapman & Hall/CRC, 1st edition, 1997.
- [BBD⁺05] J. Buchmann, R. Blümel, M. Döring, D. Engelbert, D. Schepers, A. Schmidt, U. Vollmer, and R.P. Weinmann. übersicht über Angriffe auf relevante kryptographische Verfahren, 2005.
- [Bli14] H.F. Blichfeld. A new principle in the geometry of numbers, with some applications. *Transactions of American Mathematical Society*, 15:227–235, 1914.
- [Cas71] J.W.S. Cassels. *An introduction to the geometry of numbers*. Springer Verlag, New York, 1971.
- [CS88] J.H. Conway and N.J. Sloane. *Sphere packings, lattices and groups*. Springer Verlag, New York, 1988.
- [EOS06] D. Engelbert, R. Overbeck, and A. Schmidt. A summary of McEliece-type cryptosystems and their security. *Preprint*, 2006.
- [GSL] GSL - GNU Scientific Library. <http://www.gnu.org/software/gsl/>.
- [Kai94] M. Kaib. *Gitterbasenreduktion fuer beliebige Normen*. PhD thesis, Johann Wolfgang Goethe Univesity, 1994.
- [KL78] G.A. Kabatiansky and V.I. Levenshtein. Bounds for packings in a sphere and in space. *Problems of Information Transition*, 14, 1978.
- [LLL82] L. Lovász, H.W. Lenstra, and A.K. Lenstra. Factoring polynomials with rational coefficients. *Math. Annalen*, 261:515–534, 1982.
- [LS92] L. Lovász and H. Scarf. The generalized basis reduction algorithm. *Mathematics of Operations Research*, 17:754–764, 1992.
- [McE78] R.J. McEliece. A public key cryptosystem based on algebraic coding theory. *DSN progress report*, 42-44:114–116, 1978.
- [NTL] NTL: A library for doing Number Theory. <http://www.shoup.net/ntl/>.

- [Rit97] H. Ritter. *Aufzählung von kurzen Gittervektoren in allgemeiner Norm*. PhD thesis, Johann Wolfgang Goethe University, 1997.
- [Sch94] C.P. Schnorr. Block reduced lattice bases and successive minima. *Combinatorics, Probability and Computing*, 3:507–522, 1994.
- [SE94] C.P. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66:181–199, 1994.
- [Wat80] G.A. Watson. *Approximation Theory and Numerical Methods*. John Wiley and Sons Ltd, 1980.

Appendix A

Source Code

A.1 Basis Reduction Algorithms

A.1.1 L^3 Basis Reduction Algorithm for the l_p Norm

```
#include "gsl/gsl_blas.h"
#include "gsl/gsl_math.h"
#include "p_norms.h"
#include "util.h"
#include <math.h>

/*
 * Given a  $m \times n$  matrix  $B$ , integers  $1 \leq j \leq m$ ,  $1 \leq k \leq m$  and  $p > 2$  finds an integer
 *  $\mu$  such that
 *  $\mu = \{\alpha \in \mathbb{Z} \mid F_j(\mathbf{b}_k + \alpha \mathbf{b}_j) \text{ is minimal}\}$ ,
 * where  $F_j$  is the  $j$ -th distance function w.r.t the  $l_p$  norm (see Definition 4.1 )
 *
 * Parameters:
 * gsl_matrix_int * basis - the matrix  $B$ 
 * int j - the integer  $j$ 
 * int k - the integer  $k$ 
 * int p - the integer  $p$ 
 *
 * Returns:
 * an integer  $\mu$ , which minimizes  $F_j(\mathbf{b}_k + \alpha \mathbf{b}_j)$ 
 */
int find_min_mu( gsl_matrix_int * basis, int j, int k, int p )
{
    int n = basis->size2;

    double Fj1bk;
    gsl_vector_int * bkminus1 = gsl_vector_int_alloc( n );
    gsl_matrix_int_get_row( bkminus1, basis, k-1 );
    gsl_vector * alpha = p.f_norm( j+1, basis, bkminus1, p, Fj1bk );
    int mu_ceil = ( int ) ceil( gsl_vector_get( alpha, j-1 ) );
    int mu_floor = ( int ) floor( gsl_vector_get( alpha, j-1 ) );
    gsl_vector_free( alpha );

    gsl_vector_int * bjminus1 = gsl_vector_int_alloc( n );
    gsl_matrix_int_get_row( bjminus1, basis, j-1 );
```

```

gsl_vector_int * b_ceil = gsl_vector_int_alloc( n );
gsl_vector_int_memcpy( b_ceil, bkminus1 );
gsl_vector_int * b_floor = gsl_vector_int_alloc( n );
gsl_vector_int_memcpy( b_floor, bkminus1 );

gsl_vector_int * temp;
if ( mu_ceil != 0 )
{
    temp = gsl_vector_int_alloc( n );
    gsl_vector_int_memcpy( temp, bjminus1 );
    gsl_vector_int_scale( temp, mu_ceil );
    gsl_vector_int_add( b_ceil, temp );
    gsl_vector_int_free( temp );
}

if ( mu_floor != 0 )
{
    temp = gsl_vector_int_alloc( n );
    gsl_vector_int_memcpy( temp, bjminus1 );
    gsl_vector_int_scale( temp, mu_floor );
    gsl_vector_int_add( b_floor, temp );
    gsl_vector_int_free( temp );
}

gsl_vector_int_free( bkminus1 );
gsl_vector_int_free( bjminus1 );

double Fjbceil;
double Fjbfloor;
gsl_vector * dummy = p_f_norm( j, basis, b_floor, p, Fjbfloor );
gsl_vector_free( dummy );
gsl_vector_int_free( b_floor );

dummy = p_f_norm( j, basis, b_ceil, p, Fjbceil );
gsl_vector_free( dummy );
gsl_vector_int_free( b_ceil );

return compare_double( Fjbfloor, Fjbceil ) == 1 ? mu_floor : mu_ceil;
}

/*
* Given an  $m \times n$  matrix  $B$ ,  $\frac{1}{2} < \Delta < 1$ , an integer  $p$ , and indexes  $s$  and  $l$ ,
*  $1 \leq s < l \leq m$ , performs LS-Reduction of the row vectors  $\mathbf{b}_1, \dots, \mathbf{b}_l$ ,
* starting from  $\mathbf{b}_s$  (see Algorithm 4.1). The algorithm is executed until either
* a row vector  $\mathbf{b}_i$  which fulfills the conditions
* 1.  $\|\mathbf{b}_i\|_p^p = t$ , for a given integer  $t$ 
* 2.  $\mathbf{b}_i \notin 2 \cdot \text{Id}(n)$ 
* is found or until the specified rows of  $B$  are LS-reduced.
* If a row vector  $\mathbf{b}_i$ , which fulfills both conditions is found then it saves
* the index  $i$  in a given integer and exits.
*
* Parameters:
* gsl_matrix_int * basis - the matrix  $B$ 
* double delta - a real number  $\Delta$ ,  $\frac{1}{2} < \Delta < 1$ 
* int pos1 - the row index, from which the LS-reduction should start

```

```

* int pos2 - the row index of the last row vector of B, which should be
* reduced
* int p - the integer p
* int t - the integer t
* int &err_row - the integer, where the index of the first row which fulfills
* conditions 1 and 2 should be written ( if such vector is found during the
* reduction )
*
* Returns:
* TRUE - if a row, which fulfills condition 1 and 2, is found during the reduction
* FALSE - otherwise
*/
bool ls_reduction(
    gsl_matrix_int * basis,
    double delta,
    int pos1,
    int pos2,
    int p,
    int t,
    int & err_row )
{
    int n = basis->size2;
    int k;

    if ( pos1 <= 2 )
        k = 2;
    else
        k = pos1;

    while ( k <= pos2 )
    {
        for ( int j = k - 1; j >= 1; j-- )
        {
            int mu = find_min_mu( basis, j, k, p );
            if ( mu != 0 )
            {
                gsl_vector_int *bjminus1 = gsl_vector_int_alloc( n );
                gsl_matrix_int_get_row( bjminus1, basis, j-1 );
                gsl_vector_int *bkminus1 = gsl_vector_int_alloc( n );
                gsl_matrix_int_get_row( bkminus1, basis, k-1 );

                gsl_vector_int_scale( bjminus1, mu );
                gsl_vector_int_add( bkminus1, bjminus1 );
                gsl_matrix_int_set_row( basis, k-1, bkminus1 );

                int pnorm;
                p_norm( bkminus1, p, pnorm );
                if ( ( pnorm == t ) && !is_vector_from_2id_matrix( bkminus1 ) )
                {
                    gsl_vector_int_free( bkminus1 );
                    err_row = k-1;
                    return true;
                }
            }

            gsl_vector_int_free( bjminus1 );
        }
    }
}

```

```

    gsl_vector_int_free( bkminus1 );
  }
}

double Fk_1bk;
double Fk_1bk_1;
gsl_vector *dummy;

gsl_vector_int *bk_1 = gsl_vector_int_alloc( n );
gsl_matrix_int_get_row( bk_1, basis, k-1 );
gsl_vector_int *bk_2 = gsl_vector_int_alloc( n );
gsl_matrix_int_get_row( bk_2, basis, k-2 );

dummy = p_f_norm( k-1, basis, bk_1, p, Fk_1bk );
gsl_vector_free( dummy );
dummy = p_f_norm( k-1, basis, bk_2, p, Fk_1bk_1 );
gsl_vector_free( dummy );

gsl_vector_int_free( bk_1 );
gsl_vector_int_free( bk_2 );

if ( compare_double( Fk_1bk, delta * Fk_1bk_1 ) == 1 )
{
  gsl_matrix_int_swap_rows( basis, k-1, k-2 );
  k = GSL_MAX_INT( k-1, 2 );
}
else
  k++;
}

return false;
}

```

A.1.2 Block Basis Reduction Algorithm for the l_p Norm

```

#include "ls_reduction.h"
#include "gsl/gsl_math.h"
#include "p_norms.h"
#include "util.h"
/*
 * Implements the following part of the ADFS Algorithm (see Algorithm 4.3 )
 *
 *  $z_t := \{z \in \mathbb{R} : F_t(z\mathbf{b}_t + \sum_{i=t+1}^s \tilde{u}_i \mathbf{b}_i) \text{ is minimal}\}$ 
 *  $l_t := \lfloor z_t \rfloor$ 
 *  $r_t := l_t + 1$ 
 * if  $F_t(l_t \mathbf{b}_t + \sum_{i=t+1}^s \tilde{u}_i \mathbf{b}_i) < F_t(r_t \mathbf{b}_t + \sum_{i=t+1}^s \tilde{u}_i \mathbf{b}_i)$ 
 * then
 *    $\tilde{u}_t := l_t$ 
 *    $l_t := l_t - 1$ 
 * else
 *    $\tilde{u}_t := r_t$ 
 *    $r_t := r_t + 1$ 
 * endif
 */

```

```

*/
int find_min_coef(
    int begin,
    int end,
    int p,
    gsl_matrix_int * basis,
    gsl_vector_int * coefs,
    gsl_vector_int * floor_coefs,
    gsl_vector_int * ceil_coefs,
    int mode )
{
    int len = basis->size2;
    gsl_vector * min_coefs;
    gsl_vector_int * btminus1;
    gsl_vector_int * vec_sum = calculate_scaled_sum(
        begin+1, end, basis, coefs );

    if ( mode == 0 )
    {
        double value;
        min_coefs = p.f_norm( begin+1, basis, vec_sum, p, value );
        int mu_ceil = ( int ) ceil( gsl_vector_get( min_coefs, begin-1 ) );
        int mu_floor = ( int ) floor( gsl_vector_get( min_coefs, begin-1 ) );
        gsl_vector_free( min_coefs );
        gsl_vector_int_set( floor_coefs, begin-1, mu_floor );
        gsl_vector_int_set( ceil_coefs, begin-1, mu_ceil );
    }

    double floor_value;
    double ceil_value;
    int f_coef = gsl_vector_int_get( floor_coefs, begin-1 );
    if ( f_coef != 0 )
    {
        btminus1 = gsl_vector_int_alloc( len );
        gsl_matrix_int_get_row( btminus1, basis, begin-1 );
        gsl_vector_int_scale( btminus1, f_coef );
        gsl_vector_int_add( btminus1, vec_sum );
        min_coefs = p.f_norm( begin, basis, btminus1, p, floor_value );
        gsl_vector_int_free( btminus1 );
    }
    else
    {
        min_coefs = p.f_norm( begin, basis, vec_sum, p, floor_value );
    }
    gsl_vector_free( min_coefs );

    int c_coef = gsl_vector_int_get( ceil_coefs, begin-1 );
    if ( c_coef != 0 )
    {
        btminus1 = gsl_vector_int_alloc( len );
        gsl_matrix_int_get_row( btminus1, basis, begin-1 );
        gsl_vector_int_scale( btminus1, c_coef );
        gsl_vector_int_add( btminus1, vec_sum );
        min_coefs = p.f_norm( begin, basis, btminus1, p, ceil_value );
        gsl_vector_int_free( btminus1 );
    }
}

```

```

}
else
  min_coefs = p_f_norm( begin, basis, vec_sum, p, ceil_value );

gsl_vector_free( min_coefs );
gsl_vector_int_free( vec_sum );

if ( compare_double( floor_value, ceil_value ) == 1 )
{
  gsl_vector_int_set( floor_coefs, begin-1, f_coef-1 );
  return f_coef;
}
else
{
  gsl_vector_int_set( ceil_coefs, begin-1, c_coef+1 );
  return c_coef;
}
}
}

/*
* Implements the enumeration subroutine ADFS (see Algorithm 4.3) of the
* block basis reduction algorithm for the  $l_p$  norm. It finds an integer non-zero
* vector  $(u_j, \dots, u_k)$ , such that the vector
* 
$$\mathbf{b}_j^{new} = \sum_{i=j}^k u_i \mathbf{b}_i$$

* satisfies the following condition
* 
$$\bar{F}_j := F_j(\mathbf{b}_j^{new}) = \lambda_{1, F_j}(\mathcal{L}(\mathbf{b}_j, \dots, \mathbf{b}_k)).$$

*
* Parameters:
* int  $j$  - the integer  $j$ ,  $1 \leq j \leq m$ 
* int  $k$  - the integer  $k$ ,  $1 \leq k \leq m, j < k$ 
* gsl_matrix_int * basis - the matrix with rows  $\mathbf{b}_1, \dots, \mathbf{b}_m$ 
* double delta - a real number  $\Delta$ ,  $\frac{1}{2} < \Delta < 1$ 
* int  $p$  - the integer  $p$ 
* double &fMin - the variable, in which  $\bar{F}_j$  is returned
*
* Returns:
* the integer vector  $(u_j, \dots, u_k)$ 
*/
gsl_vector_int * adfs(
  int j,
  int k,
  gsl_matrix_int * basis,
  double delta,
  int p,
  double &fMin)
{
  int n = basis->size2;
  gsl_vector_int * u = gsl_vector_int_alloc( k );
  gsl_vector_int_set_zero( u );
  gsl_vector_int_set( u, j-1, 1 );
  gsl_vector_int * utemp = gsl_vector_int_alloc( k );
  gsl_vector_int_set_zero( utemp );
  gsl_vector_int_set( utemp, j-1, 1 );

  int s = j;

```

```

int t = j;

gsl_vector_int * l = gsl_vector_int_alloc( k );
gsl_vector_int * r = gsl_vector_int_alloc( k );
gsl_vector_int_set_zero( l );
gsl_vector_int_set_zero( r );

gsl_vector_int * bjminus1 = gsl_vector_int_alloc( n );
gsl_matrix_int_get_row( bjminus1, basis, j-1 );

gsl_vector * min_coefs = p_f_norm( j, basis, bjminus1, p, fMin );
gsl_vector_free( min_coefs );
gsl_vector_int_free( bjminus1 );
fMin = delta * fMin;

while ( t <= k )
{
    double ftsun;
    gsl_vector_int * sum = calculate_scaled_sum( t, s, basis, utemp );
    min_coefs = p_f_norm( t, basis, sum, p, ftsun );
    gsl_vector_int_free( sum );
    gsl_vector_free( min_coefs );

    if ( compare_double( ftsun, fMin ) == 1 )
    {
        if ( t > j )
        {
            t = t - 1;
            int min = find_min_coef( t, s, p, basis, utemp, l, r, 0 );
            gsl_vector_int_set( utemp, t-1, min );
        }
        else
        {
            gsl_vector_int_memcpy( u, utemp );
            gsl_vector_int * sum = calculate_scaled_sum( j, k, basis, u );
            min_coefs = p_f_norm( j, basis, sum, p, fMin );
            gsl_vector_free( min_coefs );
            gsl_vector_int_free( sum );
        }
    }
    else
    {
        t = t + 1;
        if ( t <= k )
        {
            if ( t >= s )
            {
                int coef = gsl_vector_int_get( utemp, t-1 );
                gsl_vector_int_set( utemp, t-1, coef+1 );
                s = t;
            }
            else
            {
                int min = find_min_coef( t, s, p, basis, utemp, l, r, 1 );
                gsl_vector_int_set( utemp, t-1, min );
            }
        }
    }
}

```

```

    }
  }
}
gsl_vector_int_free( utemp );
gsl_vector_int_free( l );
gsl_vector_int_free( r );
return u;
}

/*
 * Given a basis  $(\mathbf{b}_1, \dots, \mathbf{b}_m) \in \mathbb{Z}^n$  of a lattice  $\mathcal{L}$ , indexes  $1 \leq j < k \leq m$ 
 * and a non-zero integer vector  $(u_j, \dots, u_k)$ , extends the vectors
 *  $\mathbf{b}_1, \dots, \mathbf{b}_{j-1}, \mathbf{b}_j^{new} = \sum_{i=j}^k u_i \mathbf{b}_i$ 
 * to a basis of  $\mathcal{L}$ .
 *
 * Parameters:
 * int j - the integer j
 * int k - the integer k
 * gsl_matrix_int * a - the matrix with rows  $(\mathbf{b}_1, \dots, \mathbf{b}_m) \in \mathbb{Z}^n$ 
 * gsl_vector_int * coefs - the vector  $(u_j, \dots, u_k)$ 
 */
void transform_basis(
    int j,
    int k,
    gsl_matrix_int * a,
    gsl_vector_int * coefs )
{
  int n = a->size2;

  if ( gsl_vector_int_isnull( coefs ) == 1 )
    return;

  gsl_matrix_int * submatr = gsl_matrix_int_alloc( k-j+1, n );
  for ( int l = j-1; l < k; l++ )
  {
    gsl_vector_int * temp = gsl_vector_int_alloc( n );
    gsl_matrix_int_get_row( temp, a, l );
    gsl_matrix_int_set_row( submatr, l-j+1, temp );
    gsl_vector_int_free( temp );
  }

  gsl_vector_int * subvec = gsl_vector_int_alloc( k-j+1 );
  for ( int l = j-1; l < k; l++ )
  {
    gsl_vector_int_set( subvec, l-j+1, gsl_vector_int_get( coefs, l ) );
  }

  gsl_vector_int * bjnew = calculate_scaled_sum( j, k, a, coefs );

  int i;
  int g = k-j;
  for ( i = k-j; i >= 0; i-- )
  {
    int coef = gsl_vector_int_get( subvec, i );

```

```

    if ( coef != 0 )
    {
        g = i;
        break;
    }
}

int coef = gsl_vector_int_get( subvec, g );
while ( abs( coef ) > 1 )
{
    int l = g - 1;
    for ( ; l >= 0; l-- )
    {
        int c = gsl_vector_int_get( subvec, g );
        if ( c != 0 )
            break;
    }

    int coef_g = gsl_vector_int_get( subvec, g );
    int coef_l = gsl_vector_int_get( subvec, l );

    int q = (int)( (double)coef_g / (double)coef_l );
    coef = coef_l;
    int coef_temp = coef_g - q * coef_l;
    gsl_vector_int_set( subvec, l, coef_temp );
    gsl_vector_int_set( subvec, g, coef_l );

    gsl_vector_int * bg = gsl_vector_int_alloc( n );
    gsl_matrix_int_get_row( bg, submatr, g );
    gsl_vector_int * bl = gsl_vector_int_alloc( n );
    gsl_matrix_int_get_row( bl, submatr, l );
    gsl_vector_int_scale( bg, q );
    gsl_vector_int_add( bg, bl );
    gsl_matrix_int_swap_rows( submatr, g, l );
    gsl_matrix_int_set_row( submatr, g, bg );

    gsl_vector_int_free( bg );
    gsl_vector_int_free( bl );
}
gsl_vector_int_free( subvec );

for( int ind = g; ind >= 1; ind-- )
{
    gsl_vector_int * row = gsl_vector_int_alloc( n );
    gsl_matrix_int_get_row( row, submatr, ind - 1 );
    gsl_matrix_int_set_row( submatr, ind, row );
    gsl_vector_int_free( row );
}

gsl_matrix_int_set_row( submatr, 0, bjnew );
gsl_vector_int_free( bjnew );

for ( int ind = 0; ind <= k-j; ind++ )
{
    gsl_vector_int * row = gsl_vector_int_alloc( n );

```

```

    gsl_matrix_int_get_row( row, submatr, ind );
    gsl_matrix_int_set_row( a, ind + j-1, row );
    gsl_vector_int_free( row );
}

gsl_matrix_int_free( submatr );
}

/*
 * Implements the  $(\beta, \Delta)$  block basis reduction algorithm (see Algorithm 4.2 )
 *
 * Parameters:
 * gsl_matrix_int * basis - the matrix, whose rows has to be reduced
 * double delta - a real number  $\Delta$ ,  $\frac{1}{2} < \Delta < 1$ 
 * int beta - the block size
 * int p - the integer p for the  $l_p$  norm
 * int t - the integer t (see the Section A.1.1)
 * int &err_row - the index of the row, where a vector with  $l_p$  norm
 *  $t^{1/p}$  is found during the reduction (see the Section A.1.1)
 *
 * Returns:
 * TRUE - if a vector with  $l_p$  norm  $t^{1/p}$  is found during the reduction
 * (see the Section A.1.1)
 * FALSE - otherwise
 */
bool block_reduction(
    gsl_matrix_int * basis,
    double delta,
    int beta,
    int p,
    int t,
    int &err_row )
{
    int m = basis->size1;
    int n = basis->size2;

    bool found = ls_reduction( basis, delta, 1, m, p, t, err_row );
    if ( found )
        return true;

    int j = m-1;
    int z = 0;
    while ( z < m-1 )
    {
        j++;
        if ( j == m )
        {
            j = 1;
        }
        int k = GSL_MIN( j+beta-1, m );

        double f_adfs_min;
        gsl_vector_int * u = adfs( j, k, basis, delta, p, f_adfs_min);
        gsl_vector_int * bjminus1 = gsl_vector_int_alloc( n );

```

```

gsl_matrix_int_get_row( bjminus1, basis, j-1 );

double fjbjminus1;
gsl_vector * min_coefs = p_f_norm( j, basis, bjminus1, p, fjbjminus1 );
gsl_vector_free( min_coefs );
gsl_vector_int_free( bjminus1 );

int h = GSL_MIN( k+1, m );
if ( compare_double( f_adfs_min, delta*fjbjminus1 ) == 1 )
{
    transform_basis( j, k, basis, u );
    gsl_vector_int_free( u );
    found = ls_reduction( basis, delta, j, h, p, t, err_row );
    if ( found )
        return true;
    z = 0;
}
else
{
    gsl_vector_int_free( u );

    found = ls_reduction( basis, delta, h-1, h, p, t, err_row );
    if ( found )
        return true;

    z++;
}
}

return false;
}

```

A.1.3 Calculation of the Distance Functions for the l_p Norm

```

#include "gsl/gsl_math.h"
#include "gsl/gsl_blas.h"
#include "p_norms.h"
#include "newton.h"
#include "util.h"
#include <math.h>

/*
 * Calculates  $\| \mathbf{v} \|_p^p$  of a real vector  $\mathbf{v}$ .
 *
 * Parameters:
 * gsl_vector * v - the real vector  $\mathbf{v}$ 
 * int p - the integer  $p$ 
 * double &value - the variable, where  $\| \mathbf{v} \|_p^p$  is returned.
 */
void p_norm( gsl_vector * v, int p, double &value )
{
    double sum = 0;
    int n = v->size;

```

```

    for ( int i = 0; i < n; i++ )
        sum += gsl_pow_int( fabs( gsl_vector_get( v, i ) ), p );
    value = sum;
}

/*
 * Calculates  $\| \mathbf{v} \|_p^p$  of a integer vector  $\mathbf{v}$ 
 * and returns  $\| \mathbf{v} \|_p^p$  as real number.
 *
 * Parameters:
 * gsl_vector_int * v - the integer vector  $\mathbf{v}$ 
 * int p - the integer p
 * double &value - the variable, where  $\| \mathbf{v} \|_p^p$  is returned.
 */
void p_norm( gsl_vector_int * v, int p, double &value)
{
    double sum = 0;
    int n = v->size;
    for ( int i = 0; i < n; i++ )
        sum += gsl_pow_int( fabs( gsl_vector_int_get( v, i ) ), p );
    value = sum;
}

/*
 * Calculates  $\| \mathbf{v} \|_p^p$  of a integer vector  $\mathbf{v}$ 
 * and returns  $\| \mathbf{v} \|_p^p$  as integer.
 *
 * Parameters:
 * gsl_vector_int * v - the integer vector  $\mathbf{v}$ 
 * int p - the integer p
 * int &value - the variable, where  $\| \mathbf{v} \|_p^p$  is returned.
 */
void p_norm( gsl_vector_int * v, int p, int &value)
{
    double sum = 0;
    int n = v->size;
    for ( int i = 0; i < n; i++ )
        sum += gsl_pow_int( fabs( gsl_vector_int_get( v, i ) ), p );
    value = (int)sum;
}

/*
 * Given a  $m \times n$  matrix  $B$ , a vector  $\mathbf{x}$  and integers  $1 \leq j \leq m$ 
 * calculates
 *
 * 
$$F_j(\mathbf{x}) := \min_{\alpha_1, \dots, \alpha_{j-1} \in \mathbb{R}} \| \mathbf{x} + \sum_{i=1}^{j-1} \alpha_i \mathbf{b}_i \|_p^p$$

 * and the corresponding  $\alpha_1, \dots, \alpha_{j-1}$ .
 *
 * Parameters:
 * int j - the index j
 * gsl_matrix_int * a - the matrix  $B$ 
 * gsl_vector_int * b - the vector  $\mathbf{x}$ 
 * int p - the integer p
 * double & value - the variable, in which the value of  $F_j(\mathbf{x})$  is returned
 *

```

```

* Returns:
* the corresponding to  $F_j(\mathbf{x})$   $\alpha_1, \dots, \alpha_{j-1}$ 
*/
gsl_vector * p_f_norm(
    int j,
    gsl_matrix_int * a,
    gsl_vector_int * b,
    int p,
    double &value )
{
    if ( j == 1 )
    {
        p_norm( b, p, value );
        gsl_vector * min_coefs = gsl_vector_alloc( 1 );
        gsl_vector_set_zero( min_coefs );

        return min_coefs;
    }
    else
    {
        int n = a->size2;
        gsl_matrix_int_view submatr = gsl_matrix_int_submatrix( a, 0, 0, j-1, n );

        gsl_matrix_int * temp = gsl_matrix_int_alloc( n, j-1 );
        gsl_matrix_int_transpose_memcpy( temp, &submatr.matrix );
        gsl_matrix * a_double = matrix_int_to_matrix_double( temp );
        gsl_matrix_int_free( temp );
        gsl_matrix_scale( a_double, -1 );
        gsl_vector * b_double = vector_int_to_vector_double( b );

        gsl_vector * min_coefs = newton_method( a_double, b_double, p );

        gsl_vector * y = gsl_vector_alloc( n );
        gsl_vector_memcpy( y, b_double );
        gsl_blas_dgemv( CblasNoTrans, -1.0, a_double, min_coefs, 1.0, y );
        gsl_matrix_free( a_double );
        gsl_vector_free( b_double );

        p_norm( y, p, value );
        gsl_vector_free( y );

        return min_coefs;
    }
}

```

Newton Method

```

#include "gsl/gsl_linalg.h"
#include "gsl/gsl_blas.h"
#include "gsl/gsl_math.h"
#include <math.h>

/*
* The tolerance for the Newton solution.

```

```

*/
const double NEWTON_TOL = gsl_pow_int( 10, -4 );

/*
*  $\varepsilon = 10^{-10}$  (see (4.26) )
*/
double ZERO_SINGULAR = gsl_pow_int( 10, -10 );

/*
* For a matrix  $B \in \mathbb{R}^{m \times n}$ , and a vector  $\mathbf{x} \in \mathbb{R}^m$  calculates a vector
*  $\mathbf{a} \in \mathbb{R}^n$ , such that
*  $\mathbf{a} = \{ \mathbf{z} \in \mathbb{R}^n : \| \mathbf{x} - B\mathbf{z} \|_2 \text{ is minimal} \}$ 
* (see Algorithm 4.4 ).
*
* Parameters:
* gsl_matrix * b - the matrix B
* gsl_vector * x - the vector  $\mathbf{x}$ 
*
* Returns:
* the vector  $\mathbf{a}$ 
*/
gsl_vector * least_squares( gsl_matrix * b, gsl_vector * x )
{
    gsl_matrix * copyB;
    gsl_vector * copyX;
    gsl_vector * sol;

    int m = b->size1;
    int n = b->size2;
    if ( m < n )
    {
        copyB = gsl_matrix_alloc( n, n );
        copyX = gsl_vector_alloc( n );

        gsl_vector * v = gsl_vector_alloc( n );
        gsl_vector_set_zero( v );

        for ( int i = 0; i < n; i++ )
        {
            if ( i <= m )
            {
                gsl_vector * v1 = gsl_vector_alloc( n );
                gsl_matrix_get_row( v1, b, i );
                gsl_matrix_set_row( copyB, i, v1 );
                gsl_vector_free( v1 );
                gsl_vector_set( copyX, i, gsl_vector_get( x, i ) );
            }
            else
            {
                gsl_matrix_set_row( copyB, i, v );
                gsl_vector_set( copyX, i, 0 );
            }
        }
    }
}

```

```

    gsl_vector_free( v );
}
else
{
    copyB = gsl_matrix_alloc( m, n );
    gsl_matrix_memcpy( copyB, b );

    copyX = gsl_vector_alloc( m );
    gsl_vector_memcpy( copyX, x );
}

gsl_vector * s = gsl_vector_alloc( n );
gsl_matrix * v = gsl_matrix_alloc( n, n );
gsl_vector * work = gsl_vector_alloc( n );
gsl_linalg_SV_decomp( copyB, v, s, work );
gsl_vector_free( work );

//zero all "small" singular values (we have chosen  $\epsilon = 10^{-10}$ )
double max = gsl_vector_max( s );
double zero = max * ZERO_SINGULAR;
for ( int i = 0; i < n; i++ )
{
    if ( gsl_vector_get( s, i ) < zero )
    {
        gsl_vector_set( s, i, 0 );
    }
}
sol = gsl_vector_alloc( n );
gsl_linalg_SV_solve( copyB, v, s, copyX, sol );

gsl_vector_free( s );
gsl_matrix_free( v );
gsl_matrix_free( copyB );
gsl_vector_free( copyX );

return sol;
}

/*
 * Given a  $m \times n$  matrix  $B$ , a vector  $\mathbf{x}$ , and an integer  $p$ 
 * it finds a vector
 * 
$$\xi = \{ \mathbf{z} \in \mathbb{R}^n : \| \mathbf{x} - B\mathbf{z} \|_p^p \text{ is minimal} \}$$

 * ( see Algorithm 4.5 )
 *
 * Parameters:
 * gsl_matrix * b - the matrix  $B$ 
 * gsl_vector * x - the vector  $\mathbf{x}$ 
 * int p - the integer  $p$ 
 *
 * Returns:
 * the vector  $\xi = (\xi_1, \dots, \xi_n)$ 
 */
gsl_vector * newton_method(
    gsl_matrix * b,
    gsl_vector * x,

```

```

                int p )
{
    int index = 0;
    int m = b->size1;
    int n = b->size2;

    gsl_vector * prev_point = gsl_vector_alloc( n );
    gsl_vector * next_point = gsl_vector_alloc( n );
    gsl_vector * lls_min;

    while ( true )
    {
        if ( index == 0 )
        {
            lls_min = least_squares( b, x );
            gsl_vector_memcpy( prev_point, lls_min );
            index = 1;
        }
        else
        {
            gsl_blas_dcopy( next_point, prev_point );
            gsl_vector * temp = gsl_vector_alloc( m );
            gsl_vector_memcpy( temp, x );
            gsl_blas_dgemv( CblasNoTrans, -1.0, b, prev_point, 1.0, temp );

            gsl_vector * f = gsl_vector_alloc( m );
            for ( int i = 0; i < m; i++ )
            {
                double entry = gsl_pow_int( fabs( gsl_vector_get( temp, i ) ), ( p-2 ) );
                gsl_vector_set( f, i, sqrt( entry ) );
            }
            gsl_vector_free( temp );

            gsl_matrix * d = gsl_matrix_alloc( m, m );
            gsl_matrix_set_zero( d );
            for ( int i = 0; i < m; i++ )
            {
                gsl_matrix_set( d, i, i, gsl_vector_get( f, i ) );
            }
            gsl_vector_free( f );

            gsl_matrix * newB = gsl_matrix_alloc( m, n );
            gsl_matrix_set_zero( newB );
            gsl_vector * newX = gsl_vector_alloc( m );
            gsl_vector_set_zero( newX );
            gsl_blas_dgemm( CblasNoTrans, CblasNoTrans, 1, d, b, 0, newB );
            gsl_blas_dgemv( CblasNoTrans, 1, d, x, 0, newX );
            gsl_matrix_free( d );

            lls_min = least_squares( newB, newX );
            gsl_matrix_free( newB );
            gsl_vector_free( newX );

            index++;
        }
    }
}

```

```

gsl_vector * prev_copy = gsl_vector_alloc( n );
gsl_vector_memcpy( prev_copy, prev_point );

double sc = ( (double)( p-2 ) ) / (double)( p-1 );
gsl_vector_scale( prev_copy, sc );
sc = ( (double) 1 ) / ( (double)( p-1 ) );
gsl_vector_scale( lls_min, sc );

gsl_blas_dcopy( prev_copy, next_point );
gsl_vector_free( prev_copy );
gsl_vector_add( next_point, lls_min );
gsl_vector_free( lls_min );

int count = 0;
//test whether the iteration  $\mathbf{a}_i$  has converged (see Algorithm 4.5 )
while ( count < n )
{
    double entry1 = gsl_vector_get( next_point, count );
    double entry2 = gsl_vector_get( prev_point, count );
    if ( fabs( entry1 - entry2 ) <= NEWTON_TOL )
        count++;
    else
        break;
}

if ( ( count == n && index >=2 ) )
{
    gsl_vector_free( prev_point );
    return next_point;
}

//we just want to be sure that if the newton method cannot converge we will not
//run into endless loop
if ( index > 20000 )
{
    printf( "NEWTON METHOD has not converged in 20000 iterations! Exiting . . ." );
    exit( 0 );
}
}
}

```

A.1.4 Utility Functions

```

#include "gsl_math.h"
#include "gsl/gsl_matrix.h"

const double DOUBLE_TOL = gsl_pow_int( 10, -15 );

/*
 * Compares two real positive numbers a and b.
 *
 * Paramaters:

```

```

* double a - the real number a
* double b - the real number b
*
* Returns:
* -1 - if  $a \geq b$  or if  $|a - b| < \text{DOUBLE\_TOL}$ 
* 1 - otherwise
*/
int compare_double( double a, double b )
{
    if ( a == b )
        return -1;

    if ( fabs( a - b ) < DOUBLE_TOL )
        return -1;

    return a < b ? 1 : -1;
}

/*
* Given a matrix  $A \in \mathbb{Z}^{m \times n}$ , integers  $i, j$  and
* an integer vector  $\mathbf{c}$  calculates the vector
*
*  $\mathbf{v} := \sum_{k=i}^j c_k \cdot \mathbf{a}_k$ 
*
* Parameters:
* int begin - the index  $i$ 
* int end - the index  $j$ 
* gsl_matrix_int * a - the matrix  $A$ 
* gsl_vector_int * coefs - the vector  $\mathbf{c}$ 
*
* Returns:
* the vector  $\mathbf{v} := \sum_{k=i}^j c_k \cdot \mathbf{a}_k$ 
*/
gsl_vector_int * calculate_scaled_sum(
                                int begin,
                                int end,
                                gsl_matrix_int * a,
                                gsl_vector_int * coefs )
{
    int cols = a->size2;
    gsl_vector_int * sum = gsl_vector_int_alloc( cols );
    gsl_vector_int_set_zero( sum );

    for ( int i = begin-1; i <= end-1; i++ )
    {
        int coef = gsl_vector_int_get( coefs, i );
        if ( coef != 0 )
        {
            gsl_vector_int * ai = gsl_vector_int_alloc( cols );
            gsl_matrix_int_get_row( ai, a, i );

            gsl_vector_int_scale( ai, coef );
            gsl_vector_int_add( sum, ai );
            gsl_vector_int_free( ai );
        }
    }
}

```

```

    return sum;
}

/*
 * Casts a matrix  $M \in \mathbb{Z}^{m \times n}$  to a matrix  $M' \in \mathbb{R}^{m \times n}$ 
 *
 * Parameters:
 * gsl_matrix_int * src - the matrix  $M \in \mathbb{Z}^{m \times n}$ 
 *
 * Returns:
 * the matrix  $M' \in \mathbb{R}^{m \times n}$ 
 */
gsl_matrix * matrix_int_to_matrix_double( gsl_matrix_int * src )
{
    int i,j;
    int m = src->size1;
    int n = src->size2;

    gsl_matrix * dest = gsl_matrix_alloc( m, n );
    for ( i = 0; i < m; i++ )
    {
        for ( j = 0; j < n; j++ )
        {
            double entry = (double) gsl_matrix_int_get( src, i, j );
            gsl_matrix_set( dest, i, j, entry);
        }
    }
    return dest;
}

/*
 * Casts a vector  $\mathbf{v} \in \mathbb{Z}^n$  to a vector  $\mathbf{v}' \in \mathbb{R}^n$ 
 *
 * Parameters:
 * gsl_vector_int * src - the vector  $\mathbf{v} \in \mathbb{Z}^n$ 
 *
 * Returns:
 * vector  $\mathbf{v}' \in \mathbb{R}^n$ 
 */
gsl_vector * vector_int_to_vector_double( gsl_vector_int * src )
{
    int n = src->size;

    gsl_vector * dest = gsl_vector_alloc( n );
    for ( int j = 0; j < n; j++ )
    {
        gsl_vector_set( dest, j, (double)gsl_vector_int_get( src, j ));
    }
    return dest;
}

/*
 * Given the integers vectors  $\mathbf{v}$  and  $\mathbf{w}$  tests whether
 *  $\mathbf{v} = \mathbf{w} \bmod 2$ .

```

```

*
* Parameters:
* gsl_vector_int * v - the vector v
* gsl_vector_int * w - the vector w
*
* Returns:
* TRUE - if  $\mathbf{v} = \mathbf{w} \pmod{2}$ 
* FALSE - otherwise
*/
bool error_vectors_equal( gsl_vector_int * v, gsl_vector_int * w )
{
    int n = v->size;
    int m = w->size;

    if( n != m )
        return 0;

    for ( int j = 0; j < n; j++ )
    {
        int a = gsl_vector_int_get( v, j );
        int b = gsl_vector_int_get( w, j );
        if ( ( a != 0 && b == 0 ) || ( a == 0 && b != 0 ) )
            return 0;

        if( a < 0 )
            a = a * (-1);
        if( b < 0 )
            b = b * (-1);

        if( a%2 != b%2 )
            return 0;
    }
    return 1;
}

```

```

/*
* Tests whether a given vector  $\mathbf{v} \in \mathbb{Z}_n$  is a row from  $\pm 2 \cdot \text{Id}(n)$ 
*
* Parameters:
* gsl_vector_int * vec - the vector v
*
* Returns:
* TRUE - if  $\mathbf{v} \in \pm 2 \cdot \text{Id}(n)$ 
* FALSE - otherwise
*/

```

```

bool is_vector_from_2id_matrix( gsl_vector_int * vec )
{
    int n = vec->size;
    gsl_vector_int * temp = gsl_vector_int_alloc( n );
    gsl_vector_int_memcpy( temp, vec );
    if ( gsl_vector_int_max( temp ) == 2 )
    {
        int ind = gsl_vector_int_max_index( temp );
        if ( gsl_vector_int_isnull( temp ) )
            {

```

```

        gsl_vector_int_set( temp, ind, 0 );
        gsl_vector_int_free( temp );
        return 1;
    }
}
if ( gsl_vector_int_min( temp ) == -2 )
{
    int ind = gsl_vector_int_min_index( temp );
    gsl_vector_int_set( temp, ind, 0 );
    if ( gsl_vector_int_isnull( temp ) )
    {
        gsl_vector_int_free( temp );
        return 1;
    }
}

gsl_vector_int_free( temp );
return 0;
}

/*
 * Prints a matrix  $M \in \mathbb{Z}^{m \times n}$  in a file.
 *
 * Parameters:
 * FILE * ostream - the file, where the matrix M should be written
 * gsl_matrix_int * matrix - the matrix M
 */
void print_matrix_int( FILE * ostream, gsl_matrix_int * matrix )
{
    int m = matrix->size1;
    int n = matrix->size2;

    fprintf( ostream, "\n" );

    for ( int i = 0; i < m; i++ )
    {
        for ( int j = 0; j < n; j++ )
        {
            fprintf( ostream, "%d ", gsl_matrix_int_get( matrix, i, j ) );
        }
        fprintf( ostream, "\n" );
    }
}

/*
 * Prints a vector  $\mathbf{v} \in \mathbb{Z}^n$  in a file.
 *
 * Parameters:
 * FILE * ostream - the file, where the vector v should be written
 * gsl_vector_int * vector - the vector v
 */
void print_vector_int( FILE * ostream, gsl_vector_int * vector )
{
    int n = vector->size;

```

```

fprintf( ostream, "\n" );
for ( int j = 0; j < n; j++ )
{
    fprintf( ostream, "%d ", gsl_vector_int_get( vector, j ) );
}
fprintf( ostream, "\n" );
}

```

A.2 Test Program for the Basis Reduction Algorithms

A.2.1 The Main Function

```

#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "tests.h"

/*
 * The main function reads the given parameters and calls the function
 * void run_test(
 *             FILE * infile,
 *             FILE * outfile_result )
 */
int main( int argc, char * argv[] )
{
    FILE * FIN;
    FILE * FOUT_RESULT;

    if ( argc < 3 )
    {
        printf( "Undefined usage! Exiting..." );
        return 0;
    }

    FIN = fopen( argv[1], "r" );
    if ( !FIN )
    {
        printf( " Error by opening the specified input file! Exiting ..." );
        return 0;
    }

    FOUT_RESULT = fopen( argv[2], "w" );
    if ( !FOUT_RESULT )
    {
        printf( " Error by opening the specified output file! Exiting ..." );
        return 0;
    }

    run_test( FIN, FOUT_RESULT );

    fclose( FIN );
}

```

```

fclose( FOUT_RESULT );

return 0;
}

```

A.2.2 The Test Function

```

#include "block_reduction.h"
#include "ls_reduction.h"
#include "p_norms.h"
#include "tests.h"
#include "util.h"
#include "LLL.h"
#include <sys/resource.h>
#include <sys/time.h>
#include <stdio.h>

/*
 * Given matrices A and B, checks whether the row vectors of A
 * are contained in the lattice, generated by the rows of B and vice versa and
 * writes the result in a specified file.
 *
 * Parameters
 * gsl_matrix_int * bin - the matrix A
 * gsl_matrix_int * bout - the matrix B
 * FILE * outstream - the output file
 */
void test_reduced_output( gsl_matrix_int * bin, gsl_matrix_int * bout, FILE * outstream )
{
    mat_ZZ input;
    mat_ZZ output;
    int n = bin->size1;

    input.SetDims( n, n );

    for ( int i = 0; i < n; i++ )
    {
        for ( int j = 0; j < n; j++ )
            input[i][j] = gsl_matrix_int_get( bin, i, j );
    }

    output.SetDims( n, n );
    for ( int i = 0; i < n; i++ )
    {
        for ( int j = 0; j < n; j++ )
            output[i][j] = gsl_matrix_int_get( bout, i, j );
    }

    fprintf( outstream, "\nTEST 1: OUTPUT VECTORS ARE IN THE LATTICE GENERATED BY");
    fprintf( outstream, " THE ROWS OF THE INPUT MATRIX: \n" );
    bool checked1 = 1;
    for( int i = 0; i < n; i++ )
    {
        vec_ZZ x;

```

```

long res = LatticeSolve( x, input, output[i] );
if( res == 0 )
{
    checked1 = 0;
    fprintf( ostream, "\nFALSE\n" );
    fprintf( ostream, "The row %d from the output matrix is not in the lattice", i+1);
    fprintf( ostream, "generated by the rows of the input matrix:\n");
    for( int g = 0; g < n; g++ )
    {
        int b = to_int( output[i][g] );
        fprintf( ostream, "%d ", b );
    }
    fprintf( ostream, "\n" );
}
}

if( checked1 )
    fprintf( ostream, "TRUE\n" );

fprintf( ostream, "\nTEST 2: INPUT VECTORS ARE IN THE LATTICE GENERATED BY");
fprintf( ostream, " THE ROWS OF THE OUTPUT MATRIX: \n" );
bool checked2 = 1;
for( int i = 0; i < n; i++ )
{
    vec_ZZ x;
    long res = LatticeSolve( x, output, input[i] );
    if( res == 0 )
    {
        checked2 = 0;
        fprintf( ostream, "\nFALSE\n" );
        fprintf( ostream, "The row %d from the input matrix is not in the lattice", i+1);
        fprintf( ostream, " generated by the rows of the output matrix:\n");
        for( int g = 0; g < n; g++ )
        {
            int b = to_int( input[i][g] );
            fprintf( ostream, "%d ", b );
        }
        fprintf( ostream, "\n" );
    }
}

if( checked2 )
    fprintf( ostream, "TRUE\n" );
}

```

```

/*
* Reads the reduction parameters and the matrix from a specified input file and, depending
* on the specified parameters in the input file, runs either the LS algorithm or the block
* basis reduction algorithm for the input matrix. After the reduction, if a vector  $\mathbf{v}$ ,
* such that
* 1.  $\|\mathbf{v}\|_p^p = t$ , for a given integer  $t$ 
* 2.  $\mathbf{v} \notin 2 \cdot \text{Id}(n)$ 
* was found, it tests whether  $\mathbf{v}$  is equal modulo 2 to the specified in the
* input file error vector. After each reduction it calls test reduced output.

```

```

* to check whether the input lattice has been changed during the reduction and writes the
* gathered information from the reduction in the specified output file.
*
* Parameters:
* FILE * instream - the input file
* FILE * ostream - the output file
*/
void run_test(
    FILE * instream,
    FILE * ostream )
{
    int n, algo, p, beta;
    double delta;

    fscanf( instream, "%d", &algo );
    fscanf( instream, "%d", &p );
    fscanf( instream, "%lf", &delta );
    if ( algo == 1 )
        fscanf( instream, "%d", &beta );
    fscanf( instream, "%d", &n );

    gsl_matrix_int * basis = gsl_matrix_int_alloc ( n, n );
    int * a;
    for ( int i = 0; i < n; i++ )
    {
        a = new int [n];
        for ( int j = 0; j < n; j++ )
        {
            fscanf( instream, "%d", &a[j] );
            gsl_matrix_int_set( basis, i, j, a[j] );
        }
        delete [] a;
    }

    gsl_vector_int * err = gsl_vector_int_alloc( n );
    for ( int i = 0; i < n; i++ )
    {
        a = new int [n];
        fscanf( instream, "%d", &a[i] );
        gsl_vector_int_set( err, i, a[i] );
        delete [] a;
    }

    fprintf( ostream, "\nRUNNING TEST . . . \n");

    gsl_matrix_int * basis_copy = gsl_matrix_int_alloc( n, n );
    gsl_matrix_int_memcpy( basis_copy, basis );

    bool found;
    int err_row = -1;
    int t;
    p_norm( err, p, t );
    if ( algo == 0 )
        found = ls_reduction( basis, delta, 1, basis->size1, p, t, err_row );
    if ( algo == 1 )

```

```

found = block_reduction( basis, delta, beta, p, t, err_row );

fprintf( ostream, "\nOUTPUT:\n" );
struct rusage tmp;
if (getrusage(RUSAGE_SELF, &tmp) != 0)
{
    fprintf( ostream, "\nSorry, problem during rusage! EXIT\n");
    exit(0);
}
fprintf(
    ostream,
    "Time used from the algorithm: %d seconds, %d microseconds\n",
    tmp.ru_utime.tv_sec, tmp.ru_utime.tv_usec );
if ( found )
{
    gsl_vector_int * found_err = gsl_vector_int_alloc( n );
    gsl_matrix_int_get_row( found_err, basis, err_row );
    if( !error_vectors_equal( err, found_err ) )
    {
        fprintf(
            ostream,
            "\nERROR! ANOTHER VECTOR WITH WEIGHT t FOUND IN ROW %d\n",
            err_row + 1 );
    }
    else
        fprintf( ostream, "\nFOUND in row %d\n", err_row + 1 );
    gsl_vector_int_free( found_err );
}
else
    fprintf( ostream, "\nNOT FOUND\n" );
fprintf( ostream, "Original Error Vector:\n" );
print_vector_int( ostream, err );
fprintf( ostream, "The reduced matrix is:\n" );
print_matrix_int( ostream, basis );

test_reduced_output( basis_copy, basis, ostream );

gsl_matrix_int_free( basis );
gsl_vector_int_free( err );
gsl_matrix_int_free( basis_copy );
}

```

A.3 Program for Generation of Test Input Files

A.3.1 The Main Function

```

#include "generate_input.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( int argc, char **argv )
{

```

```

int id_mode;
int cipher_mode;
int sigma_mode;
int g_mode;
FILE * FIN;
FILE * FOUT;

if ( argc < 7 )
{
    printf( " Undefined Usage! Exiting . . . " );
    return 0;
}

FIN = fopen( argv[1], "r" );
if ( !FIN )
{
    printf( " The specified input file cannot be opened! Exiting . . . " );
    return 0;
}

FOUT = fopen( argv[2], "w" );
if ( !FOUT )
{
    printf( " Error by opening the specified output file! Exiting . . . " );
    return 0;
}

cipher_mode = atoi( argv[3] );
if ( !( cipher_mode == 0 ) && !( cipher_mode == 1 ) )
{
    printf( "The third parameter must be either 0 or 1! Exiting. . . " );
    return 0;
}

id_mode = atoi( argv[4] );
if ( !( id_mode == 0 ) && !( id_mode == 1 ) )
{
    printf( "The fourth parameter must be either 0 or 1! Exiting. . . " );
    return 0;
}

sigma_mode = atoi( argv[5] );
if ( !( sigma_mode == 0 ) && !( sigma_mode == 1 ) )
{
    printf( "The fifth parameter must be either 0 or 1! Exiting. . . " );
    return 0;
}

g_mode = atoi( argv[6] );
if ( !( g_mode == 0 ) && !( g_mode == 1 ) )
{
    printf( "The sixth parameter must be either 0 or 1! Exiting. . . " );
    return 0;
}

```

```

generate_input( cipher_mode, id_mode, sigma_mode, g_mode, FIN, FOUT );

fclose( FIN );
fclose( FOUT );
return 0;
}

```

A.3.2 Test File Generation Functions

```

#include "gauss_elimination_and_column_permutation.h"
#include "random_messages_and_error_vectors.h"
#include "random_binary_matrices.h"
#include "gsl/gsl_randist.h"
#include "gsl/gsl_blas.h"
#include "gsl/gsl_rng.h"
#include "util.h"
#include <stdio.h>
#include <time.h>

/*
 * Computes the vector c, such that  $\mathbf{c} = \mathbf{m}G \oplus \mathbf{e}$ .
 *
 * Parameters:
 * gsl_matrix_int * g - the matrix G
 * gsl_vector_int * mes - the vector m
 * gsl_vector_int * error_vector - the vector e
 *
 * Returns:
 * the vector  $\mathbf{c} = \mathbf{m}G \oplus \mathbf{e}$ 
 */
gsl_vector_int * get_ciphertext(
    gsl_matrix_int * g,
    gsl_vector_int * mes,
    gsl_vector_int * error_vector )
{
    int n = g->size2;

    gsl_vector * code_d = vector_int_to_vector_double( error_vector );
    gsl_vector * mes_d = vector_int_to_vector_double( mes );
    gsl_matrix * g_d = matrix_int_to_matrix_double( g );

    gsl_blas_dgemv( CblasTrans, 1, g_d, mes_d, 1, code_d );
    gsl_matrix_free( g_d );
    gsl_vector_free( mes_d );

    int entry;
    gsl_vector_int * code = gsl_vector_int_alloc( n );
    for ( int i = 0; i < n; i++ )
    {
        entry = ( int ) gsl_vector_get( code_d, i );
        entry = entry % 2;
        gsl_vector_int_set( code, i, entry );
    }
    gsl_vector_free( code_d );
}

```

```

    return code;
}

/*
 * Computes a random binary regular matrix  $S \in \mathbb{F}_2^{k \times k}$  and
 * a random permutation matrix  $P \in \mathbb{F}_2^{n \times n}$  and the matrix
 * product  $G = SG'P$ , where  $G'$  is a  $k \times n$  generator matrix of a binary
 * irreducible Goppa code.
 *
 * Parameters:
 * gsl_matrix_int * goppa_matrix - the generator matrix  $G'$ 
 * gsl_matrix_int * gpub - the matrix  $G = SG'P$ 
 *
 * Returns:
 * FALSE - if there was an error in the generation of the matrix  $S$ 
 * TRUE - otherwise
 */
bool get_mceliece_gpub( gsl_matrix_int * goppa_matrix, gsl_matrix_int * gpub )
{
    int m = goppa_matrix->size1;
    int n = goppa_matrix->size2;

    gsl_matrix * s_inverse = gsl_matrix_alloc( m, m );
    gsl_matrix * s = gsl_matrix_alloc( m, m );
    bool success = create_random_regular_matrix_and_inverse( m, s, s_inverse );
    if ( !success )
    {
        gsl_matrix_free( s );
        gsl_matrix_free( s_inverse );
        return 0;
    }
    gsl_matrix_free( s_inverse );

    int * random_perm = create_random_permutation( n );
    gsl_matrix * perm_matrix = create_permutation_matrix( n, random_perm );
    delete [] random_perm;

    gsl_matrix * res_double = gsl_matrix_alloc( m, n );
    gsl_matrix * temp = gsl_matrix_alloc( m, n );
    gsl_matrix * goppa_double = matrix_int_to_matrix_double( goppa_matrix );
    gsl_blas_dgemm( CblasNoTrans, CblasNoTrans, 1, s, goppa_double, 0, temp );
    gsl_blas_dgemm( CblasNoTrans, CblasNoTrans, 1, temp, perm_matrix, 0, res_double );
    gsl_matrix_free( goppa_double );
    gsl_matrix_free( temp );
    gsl_matrix_free( s );

    for ( int i = 0; i < m; i++ )
        for ( int j = 0; j < n; j++ )
        {
            int a = ( int ) gsl_matrix_get( res_double, i, j );
            gsl_matrix_int_set( gpub, i, j, a%2 );
        }

    return 1;
}

```

```

}

/*
 * Transforms a given  $m \times n$  matrix  $G$  in a systematic form into a matrix  $M$ ,
 * such that
 *
 *  $m_{i,j} = 0$  if  $g_{i,j} = 0$  and  $1 \leq i \leq k+1, 1 \leq j \leq n$ 
 *  $m_{i,j} = 1$  if  $g_{i,j} = 0$  and  $1 \leq i \leq k+1, 1 \leq j \leq k+1$ 
 *  $m_{i,j} = 1$  or  $m_{i,j} = -1$  (randomly!) if  $g_{i,j} = 1$  and  $1 \leq i \leq k+1, k+1 \leq j \leq n$ 
 *
 * Parameters:
 * gsl_matrix_int * mat - the matrix  $G$ 
 */
void random_sigma( gsl_matrix_int * mat )
{
    gsl_rng_env_setup();
    const gsl_rng_type * T = gsl_rng_default;
    gsl_rng * r = gsl_rng_alloc( T );
    time_t random_seed = time( NULL );
    gsl_rng_set( r, random_seed );

    int m = mat->size1;
    int n = mat->size2;

    int entry;
    double d;

    for ( int i = 0; i < m; i++ )
        for ( int j = m; j < n; j++ )
        {
            entry = gsl_matrix_int_get( mat, i, j );
            if ( entry == 1 )
            {
                d = gsl_rng_uniform_int( r, 2 );
                int sign = (int)d;
                if ( sign == 1 )
                    gsl_matrix_int_set( mat, i, j, -1 );
            }
        }

    gsl_rng_free( r );
}

/*
 * Reads the given input file and generates a input file for the LS or the
 * block basis reduction algorithm according to the given parameters.
 *
 * Parameters:
 * int cipher_mode - an integer, which specifies, whether the ciphertext  $\mathbf{c}$ 
 * will be inserted as the first row or as the last row of  $\tilde{G}$ 
 * int id_mode - an integer, which specifies whether the row vectors  $2\mathbf{e}_{k+1}, \dots, 2\mathbf{e}_n$ 
 * will be inserted as the first  $n - k - 1$  rows or as the last  $n - k - 1$  rows of the
 * matrix  $B^{\text{in}}$ .
 * int sigma_mode - an integer, which specifies the map  $\sigma$ , which will be
 * applied on  $\tilde{G}$  sys

```

```

* int g_mode - integer, which specifies whether  $G = G'$  or  $G = SG'P$ 
* FILE * instream - the input file of the input generation program
* FILE * ostream - the file, in which the generated input should be written
*/
void generate_input(
    int cipher_mode,
    int id_mode,
    int sigma_mode,
    int g_mode,
    FILE * instream,
    FILE * ostream )
{
    int algo, k, n, t, beta, p;
    double delta;

    fscanf( instream, "%d", &algo );
    fscanf( instream, "%lf", &delta );
    if ( algo == 1 )
        fscanf( instream, "%d", &beta );
    fscanf( instream, "%d", &p );
    fscanf( instream, "%d", &k );
    fscanf( instream, "%d", &n );

    gsl_matrix_int * goppa_matrix = gsl_matrix_int_alloc( k, n );
    int * a;
    for ( int i = 0; i < k; i++ )
    {
        a = new int [n];
        for ( int j = 0; j < n; j++ )
        {
            fscanf( instream, "%d", &a[j] );
            gsl_matrix_int_set( goppa_matrix, i, j, a[j] );
        }
        delete [] a;
    }
    fscanf( instream, "%d", &t );

    gsl_vector_int * mes = create_random_message( k );
    gsl_vector_int * err = create_random_error_vector_with_weight( t, n );
    gsl_vector_int * ciphertext = get_ciphertext( goppa_matrix, mes, err );
    gsl_vector_int_free( mes );

    gsl_matrix_int * gpub = gsl_matrix_int_alloc( k, n );
    if( g_mode == 0 )
    {
        gsl_matrix_int_memcpy( gpub, goppa_matrix );
    }
    if( g_mode == 1 )
    {
        bool success = get_mceliece_gpub( goppa_matrix, gpub );
        if ( !success )
        {
            printf( "ERROR in creating RANDOM REGULAR MATRIX!" );
            return;
        }
    }
}

```

```

}

gsl_matrix_int * matrix_for_gauss = gsl_matrix_int_alloc( k+1, n );

if ( cipher_mode == 0 )
{
    gsl_matrix_int_set_row( matrix_for_gauss, 0, ciphertext );
    for ( int i = 1; i <= k; i++ )
    {
        gsl_vector_int * row = gsl_vector_int_alloc( n );
        gsl_matrix_int_get_row( row, gpub, i-1 );
        gsl_matrix_int_set_row( matrix_for_gauss, i, row );
        gsl_vector_int_free( row );
    }
}

if ( cipher_mode == 1 )
{
    for ( int i = 0; i < k; i++ )
    {
        gsl_vector_int * row = gsl_vector_int_alloc( n );
        gsl_matrix_int_get_row( row, gpub, i );
        gsl_matrix_int_set_row( matrix_for_gauss, i, row );
        gsl_vector_int_free( row );
    }
    gsl_matrix_int_set_row( matrix_for_gauss, k, ciphertext );
}

gsl_matrix_int_free( gpub );
gsl_vector_int_free( ciphertext );

gsl_matrix_int * perm_of_cols = gsl_matrix_int_alloc( n, n );
gsl_matrix_int * mat_in_sys_form = gsl_matrix_int_alloc( k+1, n );

bool success_gauss = get_matrix_systematic_form(
    matrix_for_gauss, perm_of_cols, mat_in_sys_form );
if ( !success_gauss )
{
    printf( "ERROR in GAUSS! Exiting..." );
    gsl_matrix_int_free( perm_of_cols );
    gsl_matrix_int_free( mat_in_sys_form );
    gsl_matrix_int_free( matrix_for_gauss );
    return;
}

gsl_matrix_int_free( matrix_for_gauss );

int * inv_perm = get_inverse_permutation( perm_of_cols );
gsl_vector_int * err_perm = perform_vector_permutation( err, inv_perm );
delete [] inv_perm;
gsl_vector_int_free( err );
gsl_matrix_int_free( perm_of_cols );

if ( sigma_mode == 1 )
    random_sigma( mat_in_sys_form );

```

```

gsl_matrix_int * algo_input_matrix = gsl_matrix_int_alloc( n, n );
gsl_matrix_int * id_2_times_k_n = gsl_matrix_int_alloc( n-k-1, n );
gsl_matrix_int_set_zero( id_2_times_k_n );
for ( int i = 0, s = k+1; s < n ; i++, s++ )
{
    gsl_matrix_int_set( id_2_times_k_n, i, s, 2 );
}

if ( id_mode == 0 )
{
    for ( int i = 0; i < n-k-1; i++ )
    {
        gsl_vector_int * row = gsl_vector_int_alloc( n );
        gsl_matrix_int_get_row( row, id_2_times_k_n, i );
        gsl_matrix_int_set_row( algo_input_matrix, i, row );
        gsl_vector_int_free( row );
    }
    for ( int i = n-k-1, r = 0; i < n; i++, r++ )
    {
        gsl_vector_int * row = gsl_vector_int_alloc( n );
        gsl_matrix_int_get_row( row, mat_in_sys_form, r );
        gsl_matrix_int_set_row( algo_input_matrix, i, row );
        gsl_vector_int_free( row );
    }
}

if ( id_mode == 1 )
{
    for ( int i = 0; i < k+1; i++ )
    {
        gsl_vector_int * row = gsl_vector_int_alloc( n );
        gsl_matrix_int_get_row( row, mat_in_sys_form, i );
        gsl_matrix_int_set_row( algo_input_matrix, i, row );
        gsl_vector_int_free( row );
    }
    for ( int i = k+1, r = 0; i < n; i++, r++ )
    {
        gsl_vector_int * row = gsl_vector_int_alloc( n );
        gsl_matrix_int_get_row( row, id_2_times_k_n, r );
        gsl_matrix_int_set_row( algo_input_matrix, i, row );
        gsl_vector_int_free( row );
    }
}
gsl_matrix_int_free( id_2_times_k_n );
gsl_matrix_int_free( mat_in_sys_form );

fprintf( ostream, "%d\n", algo );
fprintf( ostream, "%d\n", p );
fprintf( ostream, "%lf\n", delta );
if ( algo == 1 )
    fprintf( ostream, "%d\n", beta );
fprintf( ostream, "%d", n );
print_matrix_int( ostream, algo_input_matrix );
print_vector_int( ostream, err_perm );

```

```

gsl_vector_int_free( err_perm );
gsl_matrix_int_free( algo_input_matrix );
}

```

A.3.3 Generation of Random Error and Message Vectors

```

#include "random_binary_matrices.h"
#include "gsl/gsl_randist.h"
#include "gsl/gsl_rng.h"
#include <time.h>

/*
 * Creates a random n-dimensional binary vector. In order to decide whether to set
 * an entry of the vector as 0 or 1 we use the following GSL function, which
 * returns a random integer from 0 to n-1, for n = 2
 *
 * double gsl_rng_uniform_int (const gsl_rng * r, unsigned long int n)
 *
 * For more details see:
 * http://www.gnu.org/software/gsl/manual/html\_node/Sampling-from-a-random-number-generator.html#Sampling-from-a-random-number-generator
 *
 * Parameters:
 * int n - the length of the vector
 *
 * Returns:
 * a random n-dimensional binary vector
 */
gsl_vector_int * create_random_message( int n )
{
    gsl_vector_int * mes = gsl_vector_int_alloc( n );

    gsl_rng_env_setup();
    const gsl_rng_type * T = gsl_rng_default;
    gsl_rng * r = gsl_rng_alloc( T );
    time_t random_seed = time( NULL );
    gsl_rng_set( r, random_seed );

    for ( int j = 0; j < n; j++ )
    {
        double d = gsl_rng_uniform_int( r, 2 );
        gsl_vector_int_set( mes, j, (int) d );
    }

    gsl_rng_free( r );
    return mes;
}

/*
 * Creates a random n-dimensional binary vector with weight t. It creates a
 * random permutation  $\pi$  of the numbers  $0, \dots, n-1$  and sets 1 at the
 * entries with indexes  $\pi(0), \dots, \pi(t-1)$ 
 *

```

```

* Parameters:
* int n - the length of the vector
* int t - the weight of the vector
*
* Returns:
* a random n-dimensional binary vector with weight t
*/
gsl_vector_int * create_random_error_vector_with_weight( int t, int len )
{
    gsl_vector_int * err = gsl_vector_int_alloc( len );
    gsl_vector_int_set_zero( err );

    int * perm = create_random_permutation( len );
    for ( int i = 0; i < t; i++ )
    {
        gsl_vector_int_set( err, perm[i], 1 );
    }

    delete [] perm;
    return err;
}

```

A.3.4 Generation of Random Binary Invertible Matrices and Random Permutation Matrices

```

#include "gsl/gsl_randist.h"
#include "gsl/gsl_linalg.h"
#include "gsl/gsl_blas.h"
#include "gsl/gsl_rng.h"
#include <time.h>

/*
* Given  $n \in \mathbb{Z}$ , creates a random permutation of the numbers  $0, \dots, n - 1$ .
* We use the GSL function
*
* void gsl_ran_shuffle (const gsl_rng * r, void * base, size_t n, size_t size).
*
* For more details see:
* http://www.gnu.org/software/gsl/manual/html\_node/Shuffling-and-Sampling.html
*
* Parameters:
* int n -  $n \in \mathbb{Z}$ 
*
* Returns:
* a random permutation of the numbers  $0, \dots, n - 1$ 
*/
int * create_random_permutation( int n )
{
    int * res = new int[n];

    for ( int i = 0; i < n; i++ )
        res[i] = i;
}

```

```

gsl_rng_env_setup();
const gsl_rng_type * T = gsl_rng_default;
gsl_rng * r = gsl_rng_alloc( T );
time_t random_seed = time( NULL );
gsl_rng_set( r, random_seed );

gsl_ran_shuffle( r, res, n, sizeof( int ) );
gsl_rng_free( r );

return res;
}

/*
 * Given  $n \in \mathbb{Z}$  and a permutation  $p$  of the numbers  $0, \dots, n-1$ , creates
 * a permutation matrix, corresponding to the given permutation.
 *
 * Parameters:
 * int  $n$  -  $n \in \mathbb{Z}$ 
 * int *  $p$  - a permutation of the numbers  $0, \dots, n-1$ 
 *
 * Returns:
 * the permutation matrix corresponding to  $p$ 
 */
gsl_matrix * create_permutation_matrix( int n, int * p )
{
    gsl_matrix * res = gsl_matrix_alloc( n, n );
    gsl_matrix_set_zero( res );
    for ( int i = 0; i < n; i++ )
        gsl_matrix_set( res, i, p[i], 1 );

    return res;
}

/*
 * Creates an  $n \times n$  random binary regular upper triangular matrix. First it takes
 * the  $n \times n$  identity matrix and sets the entries  $(i, j)$  with  $i < j$  randomly
 * to 0 or 1. The so constructed upper triangular matrix is regular, since all its
 * diagonal entries are 1.
 *
 * Parameters:
 * int  $n$  - number of rows (columns) of the matrix
 *
 * Returns:
 * an  $n \times n$  random binary regular upper triangular matrix
 */
gsl_matrix * create_random_upper_triangular_matrix( int n )
{
    gsl_rng_env_setup();
    const gsl_rng_type * T = gsl_rng_default;
    gsl_rng * r = gsl_rng_alloc( T );
    time_t random_seed = time( NULL );
    gsl_rng_set( r, random_seed );

```

```

gsl_matrix * res = gsl_matrix_alloc( n, n );
gsl_matrix_set_zero( res );

for ( int i = 0; i < n; i++ )
    gsl_matrix_set( res, i, i, 1 );

for ( int i = 0; i < n; i++ )
    for ( int j = i + 1; j < n; j++ )
    {
        double d = gsl_rng_uniform_int( r, 2 );
        gsl_matrix_set( res, i, j, d );
    }

gsl_rng_free( r );

return res;
}

/*
 * Creates an  $n \times n$  random binary regular lower triangular matrix. First it takes
 * the  $n \times n$  identity matrix and sets the entries  $(i, j)$  with  $i > j$  randomly
 * to 0 or 1. The so constructed lower triangular matrix is regular, since all its
 * diagonal entries are 1.
 *
 * Parameters:
 * int n - number of rows (columns) of the matrix
 *
 * Returns:
 * an  $n \times n$  random binary regular lower triangular matrix
 */
gsl_matrix * create_random_lower_triangular_matrix( int n )
{
    gsl_rng_env_setup();
    const gsl_rng_type * T = gsl_rng_default;
    gsl_rng * r = gsl_rng_alloc( T );
    time_t random_seed = time( NULL );
    gsl_rng_set( r, random_seed );

    gsl_matrix * res = gsl_matrix_alloc( n, n );
    gsl_matrix_set_zero( res );

    for ( int i = 0; i < n; i++ )
        gsl_matrix_set( res, i, i, 1 );

    for ( int i = 0; i < n; i++ )
        for ( int j = 0; j < i; j++ )
        {
            double d = gsl_rng_uniform_int( r, 2 );
            gsl_matrix_set( res, i, j, d );
        }

    gsl_rng_free( r );

    return res;
}

```

```

/*
 * Given binary matrices M and N tests whether  $N = M^{-1}$ .
 *
 * Parameters:
 * gsl_matrix * rand_mat - the matrix M
 * gsl_matrix * rand_mat_inv - the matrix N
 *
 * Returns:
 * TRUE - if  $N = M^{-1}$ 
 * FALSE - otherwise
 */
bool test_random_regular_matrix( gsl_matrix * rand_mat, gsl_matrix * rand_mat_inv )
{
    int n = rand_mat->size1;
    gsl_matrix * id = gsl_matrix_alloc( n, n );
    gsl_matrix_set_zero( id );
    gsl_blas_dgemm( CblasNoTrans, CblasNoTrans, 1.0, rand_mat_inv, rand_mat, 0.0, id );

    for ( int i = 0; i < n; i++ )
        for ( int j = 0; j < n; j++ )
        {
            double a = gsl_matrix_get( id, i, j );
            gsl_matrix_set( id, i, j, ( (int)a )%2 );
        }

    for ( int i = 0; i < n; i++ )
    {
        gsl_vector * col = gsl_vector_alloc( n );
        gsl_matrix_get_col( col, id, i );
        double a = gsl_vector_get( col, i );
        if ( a != 1 )
            return 0;
        gsl_vector * temp = gsl_vector_alloc( n );
        gsl_vector_memcpy( temp, col );
        gsl_vector_set( temp, i, 0 );
        if ( !gsl_vector_isnull( temp ) )
        {
            gsl_matrix_free( id );
            return 0;
        }
    }
    gsl_matrix_free( id );
    return 1;
}

/*
 * Creates an  $n \times n$  random binary regular matrix M and its inverse. It creates a random
 * binary regular lower triangular L and a random binary regular upper triangular matrix U
 * and multiplies them. Since L and U are regular, then their product is also a regular
 * matrix. Then it multiplies  $L \cdot U$  with a random permutation matrix P. The so constructed
 * matrix  $M = L \cdot U \cdot P$  is regular, since every permutation matrix is regular. Then it creates
 *  $P^{-1}$ ,  $U^{-1}$  and  $L^{-1}$  and  $M^{-1} = P^{-1} \cdot U^{-1} \cdot L^{-1}$ .
 * Then it tests, whether  $M \cdot M^{-1} = \text{Id}(n)$ , just to be sure that there is no error.

```

```

*
* Parameters:
* int n number of rows (columns) of the matrix
* gsl_matrix * mat - the matrix, in which it will write M
* gsl_matrix * inv - the matrix, in which it will write  $M^{-1}$ 
*
* Returns:
* TRUE - if  $M \cdot M^{-1} = \text{Id}(n)$ 
* FALSE - otherwise
*/
bool create_random_regular_matrix_and_inverse( int n, gsl_matrix * mat, gsl_matrix * inv )
{
    //create random binary regular matrix
    gsl_matrix * lmd = create_random_lower_triangular_matrix( n );
    gsl_matrix * umd = create_random_upper_triangular_matrix( n );
    gsl_matrix * rmd = gsl_matrix_alloc( n, n );
    gsl_blas_dgemm( CblasNoTrans, CblasNoTrans, 1, lmd, umd, 0, rmd );

    int * p = create_random_permutation( n );
    gsl_matrix * pmat = create_permutation_matrix( n, p );
    gsl_blas_dgemm( CblasNoTrans, CblasNoTrans, 1, rmd, pmat, 0, mat );
    for ( int i = 0; i < n; i++ )
        for ( int j = 0; j < n; j++ )
        {
            double a = gsl_matrix_get( mat, i, j );
            gsl_matrix_set( mat, i, j, ( (int)a )%2 );
        }

    //create the inverse of the binary regular matrix
    //create the inverse of the lower triangular matrix lmd
    gsl_matrix * ilmd = gsl_matrix_alloc( n, n );
    gsl_matrix_set_identity( ilmd );

    for ( int i = 0; i < n; i++ )
    {
        for ( int j = i + 1; j < n; j++ )
        {
            if ( gsl_matrix_get( lmd, j, i ) > 0 )
            {
                for ( int k = 0; k <= i; k++ )
                {
                    double d1 = gsl_matrix_get( ilmd, j, k );
                    double d2 = gsl_matrix_get( ilmd, i, k );

                    double dset;
                    if ( d1 > 0.5 && d2 > 0.5 )
                        dset = 0;
                    else if ( d1 < 0.5 && d2 < 0.5 )
                        dset = 0;
                    else
                        dset = 1;

                    gsl_matrix_set( ilmd, j, k, dset );
                }
            }
        }
    }
}

```

```

}
}

//create the inverse of the upper triangluar matrix umd
gsl_matrix * iumd = gsl_matrix_alloc( n, n );
gsl_matrix_set_identity( iumd );

for ( int i = n-1; i >=0 ; i-- )
{
  for ( int j = i - 1; j >= 0; j-- )
  {
    if ( gsl_matrix_get( umd, j, i ) > 0 )
    {
      for ( int k = i; k < n; k++ )
      {
        double d1 = gsl_matrix_get( iumd, j, k );
        double d2 = gsl_matrix_get( iumd, i, k );

        double dset;
        if ( d1 > 0.5 && d2 > 0.5 )
          dset = 0;
        else if ( d1 < 0.5 && d2 < 0.5 )
          dset = 0;
        else
          dset = 1;

        gsl_matrix_set( iumd, j, k, dset );
      }
    }
  }
}

int * pinv = new int[n];

for ( int i = 0; i < n; i++ )
  pinv[p[i]] = i;

gsl_matrix * pinvmat = create_permutation_matrix( n, pinv );
gsl_matrix * tmp = gsl_matrix_alloc( n, n );
gsl_blas_dgemm( CblasNoTrans, CblasNoTrans, 1, pinvmat, iumd, 0, tmp );
gsl_blas_dgemm( CblasNoTrans, CblasNoTrans, 1, tmp, ilmd, 0, inv );
gsl_matrix_free( tmp );
for ( int i = 0; i < n; i++ )
  for ( int j = 0; j < n; j++ )
  {
    double a = gsl_matrix_get( inv, i, j );
    gsl_matrix_set( inv, i, j, ( (int)a )%2 );
  }

gsl_matrix_free( lmd );
gsl_matrix_free( umd );
gsl_matrix_free( rmd );
gsl_matrix_free( ilmd );
gsl_matrix_free( iumd );
gsl_matrix_free( pinvmat );

```

```

gsl_matrix_free( pmat );
delete [] pinv;
delete [] p;

bool check_regularity = test_random_regular_matrix( mat, inv );
return check_regularity;
}

```

A.3.5 Gaussian Elimination and Column Permutation

```

#include "gsl/gsl_blas.h"
#include "util.h"

/*
 * Sums the i-th and j-th row modulo 2 of a given matrix M.
 *
 * Parameters:
 * gsl_matrix_int * mat - the matrix M
 * int i - the index i
 * int j - the index j
 *
 * Returns:
 * the sum of the i-th and j-th row of M modulo 2
 */
gsl_vector_int * sum_rows_modulo_2( gsl_matrix_int * mat, int i, int j )
{
    int n = mat->size2;

    gsl_vector_int * rowi = gsl_vector_int_alloc( n );
    gsl_vector_int * rowj = gsl_vector_int_alloc( n );
    gsl_matrix_int_get_row( rowi, mat, i );
    gsl_matrix_int_get_row( rowj, mat, j );

    gsl_vector_int * sum = gsl_vector_int_alloc( n );
    for ( int k = 0; k < n; k++ )
    {
        int a = gsl_vector_int_get( rowi, k );
        int b = gsl_vector_int_get( rowj, k );
        gsl_vector_int_set( sum, k, (a+b)%2 );
    }
    gsl_vector_int_free( rowi );
    gsl_vector_int_free( rowj );

    return sum;
}

/*
 * Tests for a given vector v and an index j whether v is equal
 * to the j-th column of the identity matrix e_j.
 *
 * Parameters:
 * gsl_vector_int * vec - the vector v
 * int ind - the index j
 */

```

```

*
* Returns:
* TRUE - if  $\mathbf{v} = \mathbf{e}_j$ 
* FALSE - otherwise
*/
bool is_vector_id_column_at_ind( gsl_vector_int * vec, int ind )
{
    if ( gsl_vector_int_isnull( vec ) )
        return 0;

    int n = vec->size;
    gsl_vector_int * temp = gsl_vector_int_alloc( n );
    gsl_vector_int_memcpy( temp, vec );
    gsl_vector_int_set( temp, ind, 0 );

    if ( gsl_vector_int_isnull( temp ) )
        return 1;
    else
        return 0;
}

/*
* Saves a single transformation ( permutation of two rows or sum of two rows ) in the
* matrix R, performed in the Gaussian elimination algorithm.
*
* Parameters:
* gsl_matrix * r - the matrix R
* gsl_matrix * trans - the matrix corresponding to the last transformation performed in
* the Gaussian elimination algorithm
*/
void perform_transformation( gsl_matrix * r, gsl_matrix * trans )
{
    int m = r->size1;
    gsl_matrix * temp = gsl_matrix_alloc( m, m );
    gsl_blas_dgemm( CblasNoTrans, CblasNoTrans, 1, r, trans, 0, temp );
    for ( int i = 0; i < m; i++ )
        for ( int j = 0; j < m; j++ )
        {
            int a = ( ( int )gsl_matrix_get( temp, i, j ) )%2;
            gsl_matrix_set( r, i, j, (double)a );
        }
    gsl_matrix_free( temp );
}

/*
* Performs the Gaussian elimination algorithm for a given matrix M and generates
* a matrix R, corresponding to the row permutations and row sums, performed in the
* algorithm.
*
* Parameters:
* gsl_matrix_int * mat - the matrix M
* gsl_matrix_int * gauss_mat - the object, in which is written the transformed matrix
* gsl_matrix * r - the object, in which is written the matrix R
*/
void gauss_elimination(

```

```

        gsl_matrix_int * mat,
        gsl_matrix_int * gauss_mat,
        gsl_matrix * r )
{
    int m = mat->size1;
    int n = mat->size2;

    gsl_matrix_int_memcpy( gauss_mat, mat );
    //c_ind - the index of columns
    //r_ind - the index of rows
    for ( int c_ind = 0, r_ind = 0; r_ind < m; )
    {
        gsl_vector_int * col = gsl_vector_int_alloc( m );
        gsl_matrix_int_get_col( col, gauss_mat, c_ind );
        int ind_of_first_1 = -1;
        for ( int i = r_ind; i < m; i ++ )
        {
            int a = gsl_vector_int_get( col, i );
            if ( a == 1 )
            {
                ind_of_first_1 = i;
                break;
            }
        }
        if ( ind_of_first_1 == -1 )
        {
            c_ind++;
            gsl_vector_int_free( col );
            continue;
        }
        else
        {
            if ( ind_of_first_1 != r_ind )
            {
                gsl_matrix_int_swap_rows( gauss_mat, ind_of_first_1, r_ind );
                gsl_matrix * trans = gsl_matrix_alloc( m, m );
                gsl_matrix_set_identity( trans );
                gsl_matrix_swap_rows( trans, ind_of_first_1, r_ind );
                perform_transformation( r, trans );
                gsl_matrix_free( trans );
            }

            gsl_vector_int * row = gsl_vector_int_alloc( n );
            gsl_matrix_int_get_row( row, gauss_mat, r_ind );

            int row_to_sum = 0;
            while ( row_to_sum < m )
            {
                gsl_matrix_int_get_col( col, gauss_mat, c_ind );
                if ( is_vector_id_column_at_ind( col, r_ind ) )
                    break;
                if ( row_to_sum != r_ind )
                {
                    int b = gsl_vector_int_get( col, row_to_sum );
                    if ( b == 1 )

```

```

    {
        gsl_vector_int * new_row = sum_rows_modulo_2( gauss_mat, row_to_sum, r_ind );
        gsl_matrix * trans = gsl_matrix_alloc( m, m );
        gsl_matrix_set_identity( trans );
        gsl_matrix_set( trans, row_to_sum, r_ind, 1 );
        gsl_matrix_set( trans, row_to_sum, row_to_sum, 1 );
        perform_transformation( r, trans );
        gsl_matrix_int_set_row( gauss_mat, row_to_sum, new_row );
        gsl_vector_int_free( new_row );
    }
    row_to_sum ++;
}
else
    row_to_sum++;
}
gsl_vector_int_free( row );
r_ind++;
c_ind++;
}
}
}
}

```

```

/*
 * Tests for a given vector  $\mathbf{v}$  of length  $m$  whether  $\mathbf{v}$  is equal
 * to a column of the  $m \times m$  identity matrix.
 *

```

```

 * Parameters:

```

```

 * gsl_vector_int * vec - the vector  $\mathbf{v}$ 
 *

```

```

 * Returns:

```

```

 * TRUE - if  $\mathbf{v} = \mathbf{e}_j$  for some  $j \in \{1, \dots, m\}$ 

```

```

 * FALSE - otherwise
 */

```

```

bool is_vector_id_column( gsl_vector_int * vec )

```

```

{
    if ( gsl_vector_int_isnull( vec ) )
        return 0;

```

```

    int m = vec->size;

```

```

    gsl_vector_int * temp = gsl_vector_int_alloc( m );

```

```

    gsl_vector_int_memcpy( temp, vec );

```

```

    int max = gsl_vector_int_max_index( temp );

```

```

    gsl_vector_int_set( temp, max, 0 );

```

```

    if ( gsl_vector_int_isnull( temp ) )

```

```

        return 1;

```

```

    else

```

```

        return 0;
    }

```

```

/*
 * Transforms an  $m \times n$  matrix  $N$  ( $m < n$ ), which contains the columns of the  $m \times m$ 
 * identity matrix, into systematic form, using column permutations, and generates the
 * permutation matrix  $W$ , corresponding to the performed column permutations.
 *

```

```

* Parameters:
* gsl_matrix_int * mat_in_gauss_form - the matrix N
* gsl_matrix_int * perm_mat - the object, in which is written the matrix W
* gsl_matrix_int * mat_in_systematic_form - the object, in which is written the matrix
* N in systematic form
*/
void perform_column_permutation(
    gsl_matrix_int * mat_in_gauss_form,
    gsl_matrix_int * perm_mat,
    gsl_matrix_int * mat_in_systematic_form )
{
    int m = mat_in_gauss_form->size1;
    int n = mat_in_gauss_form->size2;

    int id_index = 0;
    int non_id_ind = m;

    for ( int col_ind = 0; col_ind < n; col_ind ++ )
    {
        gsl_vector_int * col = gsl_vector_int_alloc( m );
        gsl_matrix_int_get_col( col, mat_in_gauss_form, col_ind );
        if ( is_vector_id_column( col ) )
        {
            int ind = gsl_vector_int_max_index( col );
            gsl_matrix_int_set_col( mat_in_systematic_form, ind, col );
            gsl_vector_int * id_col = gsl_vector_int_alloc( n );
            gsl_vector_int_set_basis( id_col, ind );
            gsl_matrix_int_set_col( perm_mat, col_ind, id_col );
            id_index++;
        }
        else
        {
            gsl_matrix_int_set_col( mat_in_systematic_form, non_id_ind, col );
            gsl_vector_int * id_col = gsl_vector_int_alloc( n );
            gsl_vector_int_set_basis( id_col, non_id_ind );
            gsl_matrix_int_set_col( perm_mat, col_ind, id_col );
            gsl_vector_int_free( id_col );
            non_id_ind ++;
        }
        gsl_vector_int_free( col );
    }
}

/*
* Tests whether a given  $m \times n$  ( $m < n$ ) matrix  $M$  is in systematic form.
*
* Parameters:
* gsl_matrix_int * mat - the matrix M
*/
bool test_for_id_after_transformation( gsl_matrix_int * mat )
{
    int m = mat->size1;
    for ( int i = 0; i < m; i++ )
    {
        gsl_vector_int * col = gsl_vector_int_alloc( m );

```

```

gsl_matrix_int_get_col( col, mat, i );
int a = gsl_vector_int_get( col, i );
if ( a != 1 )
    return 0;
gsl_vector_int * temp = gsl_vector_int_alloc( m );
gsl_vector_int_memcpy( temp, col );
gsl_vector_int_set( temp, i, 0 );
if ( !gsl_vector_int_isnull( temp ) )
    return 0;
}
return 1;
}

/*
 * Transforms a  $m \times n$  ( $m < n$ ) matrix  $M$  into a systematic matrix  $N$  and
 * tests whether there were errors during the transformations and whether the transformed
 * matrix  $N$  is actually in systematic form.
 *
 * Parameters:
 * gsl_matrix_int * mat - the matrix  $M$ 
 * gsl_matrix_int * perm - the column permutation matrix, corresponding to the column
 * permutations used in the transformation.
 * gsl_matrix_int * sys_form - the object, in which is written the systematic form of the
 * matrix  $M$ 
 *
 * Returns:
 * TRUE - if there was no error during the transformation
 * FALSE - otherwise
 */
bool get_matrix_systematic_form(
    gsl_matrix_int * mat,
    gsl_matrix_int * perm,
    gsl_matrix_int * sys_form )
{
    int m = mat->size1;
    int n = mat->size2;

    gsl_matrix_int * gauss_elim_mat = gsl_matrix_int_alloc( m, n );
    gsl_matrix * s = gsl_matrix_alloc( m, m );
    gsl_matrix_set_identity( s );
    gauss_elimination( mat, gauss_elim_mat, s );

    gsl_matrix_int_set_zero( perm );
    gsl_matrix_int_set_zero( sys_form );
    perform_column_permutation( gauss_elim_mat, perm, sys_form );

    gsl_matrix * mat_double = matrix_int_to_matrix_double( mat );
    gsl_matrix * gauss_double = matrix_int_to_matrix_double( gauss_elim_mat );
    gsl_matrix * perm_double = matrix_int_to_matrix_double( perm );
    gsl_matrix * sys_double = matrix_int_to_matrix_double( sys_form );
    gsl_matrix * temp = gsl_matrix_alloc( m, n );
    gsl_matrix * temp2 = gsl_matrix_alloc( m, n );
    gsl_blas_dgemm( CblasNoTrans, CblasNoTrans, 1, s, sys_double, 0, temp );
    gsl_blas_dgemm( CblasNoTrans, CblasNoTrans, 1, temp, perm_double, 0, temp2 );
    for ( int i = 0; i < m; i++ )

```

```

{
  for ( int j = 0; j < n; j++ )
  {
    double a = gsl_matrix_get( mat_double, i, j );
    double b = (double)(( ( int )gsl_matrix_get( temp2, i, j ) ) % 2);
    if ( a != b )
    {
      gsl_matrix_free( gauss_double );
      gsl_matrix_free( perm_double );
      gsl_matrix_free( sys_double );
      gsl_matrix_free( temp );
      gsl_matrix_int_free( gauss_elim_mat );
      return 0;
    }
  }
}

gsl_matrix_free( gauss_double );
gsl_matrix_free( perm_double );
gsl_matrix_free( sys_double );
gsl_matrix_free( temp );
gsl_matrix_int_free( gauss_elim_mat );

if ( !test_for_id_after_transformation( sys_form ) )
  return 0;
return 1;
}

```

A.3.6 Utility Functions

```

#include "gsl/gsl_blas.h"
#include "util.h"

/*
 * Casts a matrix  $M \in \mathbb{Z}^{m \times n}$  to a matrix  $M' \in \mathbb{R}^{m \times n}$ 
 *
 * Parameters:
 * gsl_matrix_int * src - the matrix  $M \in \mathbb{Z}^{m \times n}$ 
 *
 * Returns:
 * the matrix  $M' \in \mathbb{R}^{m \times n}$ 
 */
gsl_matrix * matrix_int_to_matrix_double( gsl_matrix_int * src )
{
  int i,j;
  int m = src->size1;
  int n = src->size2;

  gsl_matrix * dest = gsl_matrix_alloc( m, n );
  for ( i = 0; i < m; i++ )
  {
    for ( j = 0; j < n; j++ )
    {

```

```

    double entry = (double) gsl_matrix_int_get( src, i, j );
    gsl_matrix_set( dest, i, j, entry);
}
}
return dest;
}

/*
 * Casts a vector  $\mathbf{v} \in \mathbb{Z}^n$  to a vector  $\mathbf{v}' \in \mathbb{R}^n$ 
 *
 * Parameters:
 * gsl_vector_int * src - the vector  $\mathbf{v} \in \mathbb{Z}^n$ 
 *
 * Returns:
 * vector  $\mathbf{v}' \in \mathbb{R}^n$ 
 */
gsl_vector * vector_int_to_vector_double( gsl_vector_int * src )
{
    int n = src->size;

    gsl_vector * dest = gsl_vector_alloc( n );
    for ( int j = 0; j < n; j++ )
    {
        gsl_vector_set( dest, j, (double)gsl_vector_int_get( src, j ));
    }
    return dest;
}

/*
 * Given an  $n \times n$  permutation matrix  $P$ , creates the permutation of the
 * numbers  $0, \dots, n - 1$  corresponding to the column permutation in the matrix  $P^{-1}$ .
 *
 * Parameters:
 * gsl_matrix_int * perm - the permutation matrix  $P$ 
 *
 * Returns:
 * the permutation of the numbers  $0, \dots, n - 1$  corresponding to the column
 * permutation in the matrix  $P^{-1}$ 
 */
int * get_inverse_permutation( gsl_matrix_int * perm )
{
    int n = perm->size1;
    int * p = new int[n];
    for ( int i = 0; i < n; i ++ )
    {
        gsl_vector_int * col = gsl_vector_int_alloc( n );
        gsl_matrix_int_get_col( col, perm, i );
        int max = gsl_vector_int_max_index( col );
        p[i] = max;
        gsl_vector_int_free( col );
    }
    int * p_inv = new int[n];
    for ( int i = 0; i < n; i++ )
        p_inv[p[i]] = i;
}

```

```

    return p_inv;
}

/*
 * Permutes the entries of a vector  $\mathbf{v} \in \mathbb{Z}^n$  according to a permutation
 * of the numbers  $0, \dots, n-1$ .
 *
 * Parameters:
 * gsl_vector_int * v - the vector  $\mathbf{v} \in \mathbb{Z}^n$ 
 * int * perm - a permutation of the numbers  $0, \dots, n-1$ 
 *
 * Returns:
 * the vector with permuted, according to the given permutation, entries
 */
gsl_vector_int * perform_vector_permutation( gsl_vector_int * v, int * perm )
{
    int n = v->size;

    gsl_vector_int * v_perm = gsl_vector_int_alloc( n );
    for ( int i = 0; i < n; i++ )
        gsl_vector_int_set( v_perm, i, gsl_vector_int_get( v, perm[i] ) );

    return v_perm;
}

/*
 * Prints a matrix  $M \in \mathbb{Z}^{m \times n}$  in a file.
 *
 * Parameters:
 * FILE * ostream - the file, where the matrix  $M$  should be written
 * gsl_matrix_int * matrix - the matrix  $M$ 
 */
void print_matrix_int( FILE * ostream, gsl_matrix_int * matrix )
{
    int m = matrix->size1;
    int n = matrix->size2;

    fprintf( ostream, "\n" );

    for ( int i = 0; i < m; i++ )
    {
        for ( int j = 0; j < n; j++ )
        {
            fprintf( ostream, "%d ", gsl_matrix_int_get( matrix, i, j ) );
        }
        fprintf( ostream, "\n" );
    }
}

/*
 * Prints a vector  $\mathbf{v} \in \mathbb{Z}^n$  in a file.
 *
 * Parameters:
 * FILE * ostream - the file, where the vector  $\mathbf{v}$  should be written
 * gsl_vector_int * vector - the vector  $\mathbf{v}$ 
 */

```

```
*/  
void print_vector_int( FILE * ostream, gsl_vector_int * vector )  
{  
    int n = vector->size;  
  
    for ( int j = 0; j < n; j++ )  
    {  
        fprintf( ostream, "%d ", gsl_vector_int_get( vector, j ) );  
    }  
    fprintf( ostream, "\n" );  
}
```