Implementing a Toolkit for Ring-LWE Based Cryptography in Arbitrary Cyclotomic Number Fields

Christoph M. Mayer*

January 21, 2016

Abstract

Recent research in the field of lattice-based cryptography, especially on the topic of the ring-based primitive ring-LWE, provided efficient and practical ring-based cryptographic schemes, which can compete with more traditional number-theoretic ones. In the case of ring-LWE these cryptographic schemes operated mainly in power-of-two cyclotomics, which vastly restricted the variety of possible applications. Due to the toolkit for ring-LWE of Lyubashevsky, Peikert and Regev, there are now cryptographic schemes that operate in arbitrary cyclotomics, with no loss in their underlying hardness guarantees, and only little loss computational efficiency.

Next to some further refinements and explanations of the theory and additional implementation notes, we provide an implementation of the toolkit of Lyubashevsky, Peikert and Regev written in C++. This includes a complete framework with fast and modular algorithms that can be used to build cryptographic schemes around ring-LWE. Our framework is easy to use, open source and has only little third party dependencies. For demonstration purposes we implemented two public-key cryptographic schemes using our framework. The complete source code is available at `https://github.com/CMMayer/Toolkit-for-Ring-LWE.git`.

Introduction

The rise of *quantum* computers is one of the biggest challenges Internet security will be facing in the near future. It has already been shown in theory that a quantum computer can compute prime factorizations and discrete logarithms efficiently, using Shor's algorithm [Sho97]. This implies that every cryptographic system which is based on these classical problems (e.g. RSA) is no longer secure in the presence of a quantum computer. The research on practicable quantum computers makes great progress - look for example at the D-Wave systems - and the daily usage of such computers might become real in around ten years. Therefore, time is running in order to provide new secure cryptosystems for quantum computers.

Over the last fifteen years there has been a broad spectrum of research concerning *post-quantum cryptography* [BBD09]. One of the most promising approaches is the *lattice-based*

*TU Darmstadt, contact mail: c.m.mayer@gmx.de

cryptography. A well known basic lattice problem is the "*Shortest Vector Problem*" (SVP), which asks for a shortest vector in a given lattice. A closely related problem is the approximate version GapSVP_γ, which asks only for an approximation of the length of the shortest vector to within a certain approximation factor γ. Although the shortest vector problem is known to be NP-hard under randomized reductions [MG02], the complexity of GapSVP_γ is not yet very explicit. The complexity is known for certain values of γ, but we still lack a complete classification. However, it is conjectured that there is no *quantum* polynomial time algorithm that approximates GapSVP_γ to within any polynomial factor γ. In [Reg09] Oded Regev first introduced the "*Learning with Errors*" (LWE) problem for lattices, which enjoys a quantum reduction to GapSVP_γ (That is, an efficient algorithm for LWE would imply an efficient quantum algorithm for GapSVP_γ) and therefore, based on the above conjecture, can be seen as a hard quantum problem. In the LWE problem we are given a list of arbitrary length, providing elements $b_i \in \mathbb{Z}_p$ such that $\langle s, a_i \rangle + e_i = b_i \mod p$, where $s \in \mathbb{Z}_p^n$ is a fixed *secret*, the a_i are chosen independently and uniformly from $\mathbb{Z}_p^n$ and the error terms e_i are sampled from a specific distribution over $\mathbb{Z}_p$. The goal is to find s. In [Reg09] Regev also provides a cryptographic scheme based on LWE, which is only one amongst many. Unfortunately, most of the cryptographic schemes based on LWE tend to be not efficient enough for practical applications. The key sizes are at least *quadratic* in the primary security parameter, which needs to be in the several hundreds to fulfill the security constraints. To overcome this problem, Lyubashevsky, Peikert and Regev developed a ring version of LWE (ring-LWE) in [LPR13a], which works with *ideal lattices* in cyclotomic number fields and can be used for cryptosystems whose key sizes are only linear in the primary security parameter.

Let $\zeta_m = \exp(2\pi i/m) \in \mathbb{C}$ for some positive integer m. In particular, ζ_m is a *primitive* m-th root of unity, i.e., m is the smallest integer such that $\zeta_m^m = 1$ and every complex root of the polynomial $X^m - 1$ can be represented by a power of ζ_m. The m-th *cyclotomic polynomial* is given by

$$\Phi_m(X) := \prod_{\substack{1 \le k \le m \\ \gcd(k,m)=1}} (X - \zeta_m^k) \in \mathbb{Z}[X].$$

The degree of $\Phi_m(X)$ is $n = \varphi(m)$, where $\varphi(\cdot)$ denotes Euler's totient function, i.e., $\varphi(m)$ is the count of number $1 \le k \le m$ that are coprime to m. Considering the polynomials with integer coefficients $\mathbb{Z}[X]$ modulo the cyclotomic polynomial $\Phi_m(X)$ yields the quotient ring $\mathbb{Z}[X]/(\Phi_m(X))$. This quotient can be seen as a $\mathbb{Z}$-vector space with the *power basis* $\{1 + \Phi_m(X), X + \Phi_m(X), \ldots, X^{n-1} + \Phi_m(X)\}$.

The main part of the ring-LWE problem takes place in the m-th *cyclotomic ring* $R = \mathbb{Z}[X]/(\Phi_m(X))$. Roughly speaking, ring-LWE tries to find a *secret* $s \in R_q = R/qR = \mathbb{Z}_q[X]/(\Phi_m(X))$, given arbitrarily many pairs $(a_i, b_i = a_i \cdot s + e_i \mod qR) \in R_q \times R_q$, where q is some integer modulus, a_i are independent and uniformly random elements in R_q, and the error terms $e_i \in R$ are sampled from a specific probability distribution. It is particularly nice to work in R, if the input m is a power of two, i.e., $m = 2^k$ for some integer $k \ge 1$. Two main reasons for that are, on the one hand, that $\deg(\Phi_m(X)) = \varphi(m) = n$ is also a power of two and, on the other hand, that $\Phi_m(X) = X^n + 1$ is maximally sparse. In particular, the latter fact provides fast $O(n \log n)$ algorithms for polynomial arithmetic modulo $\Phi_m(X)$, which is essential for the practical use of ring-LWE.

For the most cryptographic schemes based on ring-LWE, the parameter m for the cyclotomic ring $\mathbb{Z}[X]/(\Phi_m(X))$ is the main security parameter. Clearly, powers of two are sparsely distributed among the natural numbers. If we would restrict ourselves to working only with

inputs m that are powers of two, then we would face a simple problem. When a certain power of two is not sufficient for our security constraints, the next bigger power of two might be way to large, causing also the key sizes in the cryptosystem to be unnecessarily big. Furthermore, some applications like (fully) homomorphic encryption [Gen09] even *need* arbitrary m to work correctly. Therefore, it is advisable and necessary to consider arbitrary positive integers for m.

When working in the ring $R = \mathbb{Z}[X]/(\Phi_m(X))$ we represent elements via the coordinates with respect to the power basis $\{1 + \Phi_m(X), X + \Phi_m(X), \ldots, X^{n-1} + \Phi_m(X)\}$. The (general not theoretical) complexity of polynomial arithmetic modulo $\Phi_m(X)$ strongly depends on the form of $\Phi_m(X)$. If m is a powers of two we know that $\Phi_m(X)$ is maximally sparse and polynomial arithmetic modulo $\Phi_m(X)$ can be done nicely in $O(n \log n)$ time. However, in general, the m-th cyclotomic polynomial for arbitrary m might have many monomials with large coefficients. Theoretically, polynomial arithmetic modulo $\Phi_m(X)$ can still be done in $O(n \log n)$, but the generic algorithms are rather complex and hard to implement, with big hidden constants in the $O(\cdot)$ notation. Therefore, it is not convenient to work with this representation in arbitrary cyclotomic rings R.

In [LPR13b], Lyubashevsky, Peikert and Regev developed a toolkit for ring-LWE in arbitrary cyclotomic number fields and rings, which uses a different representation independent from the cyclotomic polynomial $\Phi_m(X)$. In fact, elements in R are never considered as polynomials. The m-th *cyclotomic number field* $K = \mathbb{Q}(\zeta_m)$ can be seen as a $\mathbb{Q}$-vector space with the *power basis* $\{1, \zeta_m, \ldots, \zeta_m^{\varphi(m)-1}\}$. Taking this basis as a $\mathbb{Z}$-basis yields the cyclotomic ring $R = \mathbb{Z}[\zeta_m] \cong \mathbb{Z}[X]/(\Phi_m(X))$, which coincides with the ring of integers $\mathcal{O}_K$ in K. Besides the power basis, [LPR13b] considers several $\mathbb{Q}$-bases for K, which are also $\mathbb{Z}$-bases for R. Elements in K are represented by coordinate vectors in these bases. Furthermore, K is embedded into $\mathbb{C}^n$ via the *canonical embedding*, a classical concept from algebraic number theory. Both representations provide fast algorithms, not only for arithmetical tasks, but also for other operations needed in ring-LWE based cryptography, like discretization and decoding or sampling of discrete and discretized Gaussians. Overall, the toolkit provides efficient algorithms that can be used to implement a variety of cryptographic schemes in arbitrary cyclotomics, with only a little loss in efficiency compared to power-of-two cyclotomics.

Most recently Crockett and Peikert published a work for their $\Lambda \circ \lambda$ project [CP15], which is a general-purpose library for lattice-based and ring cryptography written in the functional language Haskell. It provides in particular an implementation of the toolkit [LPR13b]. The paper was first received at the ePrint archive at November 23, so our implementation and [CP15] were developed simultaneously.

Contribution. The main contribution of this work is an implementation of the toolkit [LPR13b] for ring-LWE based cryptography in *arbitrary* cyclotomic number fields. The source code is available at `https://github.com/CMMayer/Toolkit-for-Ring-LWE.git`. The original paper provides algorithms for ring-LWE in an abstract mathematical manner without any specific instructions for an implementation. We took these abstract algorithms and translated them for an implementation. The outcome is a program in C++ that provides all the presented features from the paper and can be used to implement further cryptographic schemes based on the ring-LWE problem. In particular, we implemented both cryptosystems presented in Section 2.1, which can also be seen as small examples for the usage of our implementation.

Our program consists of two main classes representing the m-th cyclotomic number field $K = \mathbb{Q}(\zeta_m)$ and elements living in K. The elements are represented by integral coordinate vectors with respect to a specific basis. Depending on this basis, an element can belong to several ideals of K. To be more precise, we consider the ring of integers R in K, powers of the dual ideal $(R^\vee)^k \subset K$ for $k \geq 1$, and the quotients $R_q = R/qR$ and $(R_q^\vee)^k = (R^\vee/qR^\vee)^k$. Furthermore, the implementation provides a variety of algorithms for cryptographic tasks, all working only on the coordinate vector. Among other, these include addition and multiplication of the ideal elements, conversion between the representation of elements in different bases, sampling of Gaussians in R and $R^\vee$, discretization of elements in K, and a decoding procedure for error terms, which is needed in decryption.

In order to make this a self standing work, we present and review the toolkit from [LPR13b] again in Chapter 2. Some features are only explained roughly and can not be implemented straightaway. Therefore, in Chapter 3 we explain these features in more detail, but still in an implementation independent manner. Our program is only one possibility to realize the toolkit.

The original paper provides some analytical tools which are mainly used for correctness and hardness proofs of the presented cryptographic schemes. We omit these analytical tools and focus only on the content that is relevant for an implementation. Therefore, we will often refer to [LPR13b] for further details and explanations.

We emphasize that it was not our goal to provide a fully optimized and at most efficient implementation of the toolkit. The algorithms stay in their theoretical complexity classes, but might have large constants. As far as we know, our implementation is the first published implementation of this toolkit written in C++. Therefore, we focused on usability and understandability of our program, which trades off with performance. That is, we used non-native data types from the C++ standard library (STL), which greatly help the readability of the source code but also produce a slight overhead. Further, we omitted any parallelization, which is an important factor for efficiency in some algorithms, because it is highly vulnerable to errors and unexpected behavior. Also, we wanted to keep our implementation as a standalone project, in other words, tried to keep third party dependencies at a low level. In fact, our program uses two libraries other than the STL, namely the "*Number Theory Library*" (NTL) [Sho15] and the "*Boost Library*" [DA15]. The NTL is mainly used for modulo arithmetic over the integers and the Boost library for matrix representation and operations, e.g., matrix inversion. The rest is implemented from scratch, which includes in particular some fast Fourier transformation algorithms (FFTs). These FFT algorithms match the used data types for our program and some are slightly specialized versions. However, it might be useful to replace them eventually by some state-of-the-art algorithm. All in all, there is a lot to optimize in our program, but it is easy to use and provides a working implementation, which can be used for further developments.

The last chapter of this work, Chapter 4, can be seen as sort of a documentation for our implementation. All implemented classes and algorithms are presented and explained, but not in full detail. The declarations are given by code snippets from the header files. The source code itself is not quoted and our explanations do not refer to any specific positions in the source code. Therefore, it is advisable and maybe necessary to have the source code and this work, in order to fully understand the implementation.

Requirements. This work assumes that the reader has a descend knowledge in algebra, especially in field theory and the related Galois theory. Furthermore, a basic knowledge in probability theory is advisable. The presented toolkit uses several concepts from algebraic number theory, mainly specialized on the case of cyclotomic number fields. All necessary notions and facts are briefly introduced in Chapter 1. However, it might be helpful to have some background in algebraic number theory.

Organization. This work is organized as follows. Chapter 1 starts with some notations and conventions we use throughout the work. The rest of the chapter provides the necessary background in several subjects. Section 1.2 introduces the Kronecker product for matrices. Next, the notion of lattices and the decoding and discretization algorithms are presented in Section 1.3. Finally, Section 1.4 deals with algebraic number theory and cyclotomic number fields. This includes some lesser-known concepts, like tensorial decomposition in prime power cyclotomics and the dual ideal in algebraic number fields.

Chapter 2 introduces the concepts of the toolkit [LPR13b]. We review all parts that are important for an implementation. First, we give some motivation in Section 2.1 and show how the toolkit can be used. Next, we develop the necessary features and operations. This is divided into two parts. First, we deal with the ideal $R = \mathbb{Z}[\zeta_m]$ in Section 2.2, and in the following with its dual ideal $R^\vee$ in Section 2.3. We close this chapter with Section 2.4, which gives a review of the discretization algorithm in the field $\mathbb{Q}(\zeta_m)$.

Subsequently, Chapter 3 provides further implementation notes for some of the features from Chapter 2. The opening Section 3.1 introduces an algorithm for a more efficient matrix-vector-multiplication for Kronecker decomposed matrices. Section 3.2 explains how FFT algorithms can be used to multiply vectors with prime-indexed DFT and CRT matrices. Next, we describe in Section 3.3 the methods for addition and multiplication in $\mathbb{Q}(\zeta_m)$ from Chapter 2 in more detail. Thereby, the goal is to provide easy implementations for these methods. Subsequent, Section 3.4 explains an efficient algorithm for decoding in $R^\vee$. Finally, Section 3.5 deals with some pre-computations for an implementation.

The last chapter, Chapter 4 describes the implementation of the toolkit. Thereby, the Sections 4.1, 4.2 and 4.3 deals with some algorithms, which are not directly related to the toolkit. This includes the algorithm for the application of Kronecker decompositions from Section 3.1, Rader and Cooley-Tukey FFT algorithms and some further tasks from algebra and general mathematics. Next, we describe the class structure of our program in Section 4.4. The particular classes are described in the following Sections 4.5, 4.6 and 4.7.

Acknowledgments. I would like to thank Prof. Johannes Buchmann and my supervisors Florian Göpfert and Thomas Wunderer for the idea of this thesis as well as the outstanding supervision throughout the whole working period. Further, I thank all proofreaders and my fellow students for helpful discussions.

Contents

1 Preliminaries **8**
- 1.1 Notations and Conventions . . . 8
- 1.2 The Kronecker Product . . . 9
- 1.3 Lattices . . . 10
  - 1.3.1 Decoding in Lattices . . . 12
  - 1.3.2 Discretization in Lattices . . . 12
- 1.4 Algebraic Number Theory . . . 13
  - 1.4.1 Cyclotomic Number Fields . . . 13
  - 1.4.2 Embeddings and Geometry . . . 15
  - 1.4.3 Ring of Integers and its Ideals . . . 18
  - 1.4.4 Duality . . . 21
  - 1.4.5 Prime Splitting and Chinese Remainder Theorem . . . 24
- 1.5 Ring-LWE . . . 25

2 The Toolkit **28**
- 2.1 Applications of the Toolkit . . . 28
  - 2.1.1 Dual-Style Cryptosystem . . . 30
  - 2.1.2 Compact Public-Key Cryptosystem . . . 31
  - 2.1.3 Summarization of the Necessary Operations . . . 31
- 2.2 Working in the Ideal R . . . 32
  - 2.2.1 Two Specific Bases of R . . . 32
  - 2.2.2 Switching between the Powerful and the CRT Basis . . . 33
  - 2.2.3 Addition and Multiplication R . . . 35
  - 2.2.4 Sparse Decomposition of DFT and CRT . . . 38
  - 2.2.5 Sampling Discrete Gaussians in R . . . 41
- 2.3 Working in the Dual Ideal $R^\vee$ . . . 45
  - 2.3.1 Three Specific Bases of $R^\vee$ . . . 45
  - 2.3.2 Switching between the Specific Bases of $R^\vee$ . . . 46
  - 2.3.3 Addition and Multiplication in $R^\vee$ . . . 48
  - 2.3.4 Sparse Decomposition of DFT* and CRT* . . . 50
  - 2.3.5 Sampling Gaussians in the Decoding Basis . . . 51
  - 2.3.6 Decoding $R^\vee$ and its Powers . . . 53
- 2.4 Discretization in $\mathbb{Q}(\zeta_m)$ . . . 54

3 Further Implementation Notes for some Features of the Toolkit **56**
- 3.1 Multiplication with Kronecker Decomposed Matrices . . . 56
- 3.2 Efficient Application of DFT and CRT in the Prime Case . . . 59
- 3.3 Addition and Multiplication of Ideal Elements . . . 60
  - 3.3.1 Choosing the Right Basis . . . 60

3.3.2 Special Multiplication . . . 63
3.4 Efficient Decoding . . . 63
3.5 Pre-Computation . . . 67
3.5.1 The Element g . . . 67
3.5.2 Ones in Different Bases . . . 68

4 An Implementation in C++ 69
4.1 Application of Kronecker Decompositions . . . 70
4.2 Rader and Cooley-Tukey FFT . . . 70
4.2.1 Cooley-Tukey FFT . . . 72
4.2.2 Rader FFT . . . 74
4.3 Mathematics Utilities . . . 76
4.4 The Basic Class Structure . . . 78
4.5 Vector Transformation Matrices . . . 79
4.5.1 Complex Transformations . . . 80
4.5.2 Real Transformations . . . 85
4.5.3 Integer Transformations . . . 88
4.6 The Class `RingLweCryptographyField` . . . 94
4.6.1 Member Variables . . . 94
4.6.2 Constructor . . . 95
4.6.3 Available Functions . . . 96
4.6.4 Pre-Computation on Construction . . . 98
4.7 The Class `RingLweCryptographyElement` . . . 100
4.7.1 Member Variables . . . 100
4.7.2 Constructors . . . 101
4.7.3 Functions . . . 101

1 Preliminaries

We start with some notations and conventions we use throughout this work.

1.1 Notations and Conventions

- Let m be some positive integer. Then $[m]$ denotes the set $\{0, \ldots, m-1\}$.
- If D is any domain and S and T are finite index sets, we denote by D^S the set of all vectors over D indexed by S, and by $D^{S \times T}$ the set of all matrices with rows indexed by S and columns indexed by T.
- For any index set S we denote by I_S the identity matrix with rows and columns indexed by S.
- If we represent an element by its coordinate vector with respect to some basis, this coordinate vector is usually a bold letter, e.g., $\mathbf{a}$.
- By $\mathbf{a} \odot \mathbf{b}$ we denote the component-wise multiplication of two equally dimensional vectors $\mathbf{a}$ and $\mathbf{b}$.
- Bases of lattices in vector spaces are usually denoted by a upper case letter, e.g., B and refer to both, the set of basis vectors and the matrix whose columns are the basis vectors.
- If an ideal or a module in some algebraic number field has a basis, this basis is usually viewed as a vector of basis elements and denoted by the vector notation, e.g., $\vec{p} = (p_0, \ldots, p_{n-1})$.
- If V is some inner product space, we denote the inner product on V by $\langle \cdot, \cdot \rangle_V$. If it is clear which space V we mean, we drop the subscript V.
- For an any positive integer m, $\mathbb{Z}_m^*$ denotes the set of all units in $\mathbb{Z}_m$
- Euler's totient function is denoted by $\varphi(\cdot)$. That is, for a positive integer m, $\varphi(m)$ is the number of units in $\mathbb{Z}_m$, i.e., $\varphi(m) = |\mathbb{Z}_m^*|$.
- For a complex number $z \in \mathbb{C}$, $\overline{z}$ means the complex conjugation of z.
- If M is a complex matrix, then M^* means the conjugate transpose $\overline{M}^T$.
- Let P_D be some probability distribution over some domain D. Then $x \leftarrow P_D$ means that $x \in D$ is a sample from the distribution P_D.
- We denote by $N_{F,r}$ ($D_{F,r}$) the continuous (discrete) Gaussian distribution over some domain F with standard deviation r and mean zero.

- We call an algorithm *efficient* if it runs in polynomial complexity in some input parameter n.

1.2 The Kronecker Product

Definition 1.2.1 (Kronecker product)**.** Let $\mathcal{R}$ be an arbitrary ring and $A \in \mathcal{R}^{[n]\times[m]}$ and $B \in \mathcal{R}^{[s]\times[t]}$ be two matrices. The *Kronecker product* (or *tensor product*) of A and B is defined as

$$A \otimes B := \begin{bmatrix} A_{0,0}B & \cdots & A_{0,m-1}B \\ \vdots & & \vdots \\ A_{n-1,0}B & \cdots & A_{n-1,m-1}B \end{bmatrix}.$$

Remark 1.2.2. By definition, the index set of $M = A \otimes B$ is given by $([n] \times [s]) \times ([m] \times [t])$ as each entry of M is given by

$$M_{(i_0,i_1),(j_0,j_1)} := A_{i_0,j_0} \cdot B_{i_1,j_1}.$$

For two integers n_0 and n_1 we can identify the set $[n_0] \times [n_1]$ with $[n_0 n_1]$ using the bijective correspondence $(i_0, i_1) \leftrightarrow i = i_0 n_1 + i_1$. Consequently, $A \otimes B$ can be seen as a $[ns] \times [mt]$ matrix.

Next we state some useful basic properties of the Kronecker product. For a more complete list and proofs we refer to [HJ91, Chapter 4].

Proposition 1.2.3. *The Kronecker product has the following properties.*

(i) *For any scalar λ we have*

$$(\lambda A) \otimes B = A \otimes (\lambda B) = \lambda(A \otimes B).$$

(ii) *It is associative, i.e., for matrices A, B and C we have*

$$A \otimes (B \otimes C) = (A \otimes B) \otimes C.$$

(iii) *It is right and left distributive, i.e., for suitable dimensional matrices A, B and C we have*

$$(A + B) \otimes C = (A \otimes C) + (B \otimes C)$$

and

$$A \otimes (B + C) = (A \otimes B) + (A \otimes C).$$

(iv) *The Kronecker product fulfills the* mixed-product property*, i.e., for matrices A, B, C and D with suitable dimensions we have*

$$(A \otimes B)(C \otimes D) = AC \otimes BD.$$

(v) *It commutes with transposition and complex conjugation, i.e.,*

$$(A \otimes B)^T = (A^T \otimes B^T)$$

and

$$(A \otimes B)^* = (A^* \otimes B^*).$$

(vi) *For non-singular matrices we have*

$$(A \otimes B)^{-1} = (A^{-1} \otimes B^{-1}).$$

Proof. A proof can be found in e.g. [HJ91]. □

Remark 1.2.4. Taking matrices with dimensions $n \times 1$ and $m \times 1$ or $1 \times n$ and $1 \times m$ in the above definition, we see that the Kronecker product is also defined for row and column vectors. In this case, an easy computation shows that the Kronecker product has a *mixed-product property* for vectors as well, where multiplication of two vectors is given by component-wise multiplication. That is, for $\mathbf{a}, \mathbf{b} \in \mathcal{R}^n$ and $\mathbf{a}', \mathbf{b}' \in \mathcal{R}^m$ we have

$$(\mathbf{a} \odot \mathbf{b}) \otimes (\mathbf{a}' \odot \mathbf{b}') = (\mathbf{a} \otimes \mathbf{a}') \odot (\mathbf{b} \otimes \mathbf{b}'),$$

where $\odot$ is component-wise multiplication.

Assume we have a square matrix A of dimension n which is of the form $A = \bigotimes_{l=1}^{m} A_l$ where the A_l are square matrices with suitable smaller dimensions n_l. Using the mixed-product property as defined above we can rewrite this tensor product as

$$A = \prod_{l=1}^{m} (I_{n_1} \otimes \ldots \otimes I_{n_{l-1}} \otimes A_l \otimes I_{n_{l+1}} \otimes \ldots \otimes I_{n_m}).$$

Here I_n means the n dimensional identity matrix. This decomposition of A helps us to develop efficient algorithms for the application of A to a vector, which can be parallelized and only need to apply the smaller matrices A_l (see Section 3.1).

1.3 Lattices

Lattices are discrete additive subgroups in vector spaces. They are important for us, since the embeddings of all rings and ideals we consider are actual lattices in $\mathbb{C}^n$. We give a short introduction with some basic facts that we need. Most of our statements are left without a proof. For further information we refer to [Mil14, Chapter 4], or any other classical textbook on algebraic number theory and lattices.

Definition 1.3.1. Let $\mathbb{K} = \mathbb{R}$ or $\mathbb{C}$ and V a $\mathbb{K}$-vector space of dimension n. Further let $\Lambda \subset V$ be an additive subgroup of the form

$$\Lambda = v_1\mathbb{Z} + \ldots + v_m\mathbb{Z},$$

where the $v_1, \ldots, v_m$ are linearly independent vectors in V. We call Λ a *lattice* with basis $\{v_1, \ldots, v_m\}$. Λ is called a full-rank lattice if $m = n$, i.e., when $\{v_1, \ldots, v_m\}$ is a basis of V.

Notation 1.3.2. We often denote the basis $\{v_1, \ldots, v_m\}$ by an upper case letter, e.g., B and refer with $\Lambda = \mathcal{L}(B)$ to the lattice generated by B. Further it is often convenient to refer with B also to the matrix, whose columns are the basis vectors v_i.

Throughout this work we are only concerned with full-rank lattices, so the bases have always full dimension. Two bases B and B' generate the same lattice if and only if there exists a unimodular matrix U (i.e., an integral matrix with determinant ± 1) such that $BU = B'$. Further, we define the *determinant* of a lattice $\mathcal{L}(B)$ by $|\det(B)|$, which is independent of the choice of B. The *minimum distance* $\lambda_1(\Lambda)$ of a lattice Λ is defined as the smallest euclidean length for nonzero lattice vectors, i.e., $\lambda_1(\Lambda) := \min_{0 \neq x \in \Lambda} \|x\|_2$.

Another important notion is the *dual lattice* $\Lambda^\vee$ of a lattice $\Lambda \subset V$. Thereby, the dual lattice is defined as

$$\Lambda^\vee := \{y \in V \,|\forall x \in \Lambda, \langle x, y\rangle \in \mathbb{Z}\}\,.$$

Let the lattice $\Lambda = \mathcal{L}(B)$ be given by a basis $B = \{b_i\}_{i=1,\dots,m}$. Then we define the *dual basis* $D = \{d_i\}_{i=1,\dots,m}$ as the unique basis that satisfies $\mathrm{span}(B) = \mathrm{span}(D)$ and $B^T D = I$. The latter condition is equivalent to saying $\langle b_i, d_j\rangle = \delta_{ij}$, where δ_{ij} is the Kronecker delta. The dual basis D is indeed a basis for the dual lattice $\Lambda^\vee$ as the following proposition shows.

Proposition 1.3.3. *Let V be an n-dimensional vector space, $\Lambda = \mathcal{L}(B) \subset V$ be an m-dimensional lattice and D be the dual basis of B. Then $\Lambda^\vee = \mathcal{L}(D)$.*

Proof. Let $x \in \Lambda$ be an arbitrary lattice vector and write it as $\sum_{i=1}^m a_i b_i$ for suitable $a_i \in \mathbb{Z}$. Then for every $1 \leq j \leq m$ we have,

$$\langle x, d_j\rangle = \sum_{i=1}^{m} a_i \langle b_i, d_j\rangle = a_j \in \mathbb{Z}.$$

Thus, $D \subset \mathcal{L}(B)^\vee$ by definition of the dual lattice. Furthermore, it follows easily from the definition that $\mathcal{L}(B)^\vee$ is closed under addition and scalar multiplication with scalars from $\mathbb{Z}$. Hence we have $\mathcal{L}(D) \subset \mathcal{L}(B)^\vee$.

Now, for the other direction, take any $y \in \mathcal{L}(B)^\vee$. Since $y \in \mathrm{span}(B) = \mathrm{span}(D)$ we can write $y = \sum_{i=1}^m a_i d_i$ for some $a_i \in \mathbb{R}$. But again, we have for all $1 \leq j \leq m$ that

$$\langle y, b_j\rangle = \sum_{i=1}^{m} a_i \langle d_i, b_j\rangle = a_j$$

As $\langle y, b_j\rangle \in \mathbb{Z}$ we got also $a_j \in \mathbb{Z}$ and $y \in \mathcal{L}(D)$. This completes the proof. □

For full-rank lattices the dual basis is given by $\left(B^T\right)^{-1}$ which leads to two easy facts. Firstly, the passage to the dual is self inverse, i.e., $(\Lambda^\vee)^\vee = \Lambda$. A basis for $(\Lambda^\vee)^\vee$ is given by $(((B^T)^{-1})^T)^{-1} = B$ which shows this fact. Secondly, we have $\det(\Lambda^\vee) = \det(\Lambda)^{-1}$. By definition of the determinant, it holds that

$$\det(\Lambda^\vee) = |\det((B^T))^{-1}| = |\frac{1}{\det(B^T)}| = |\frac{1}{\det(B)}| = \det(\Lambda)^{-1}.$$

These two facts also apply for general lattices. Another property we have, is that for lattices Λ and $\bar{\Lambda}$, the equivalence

$$\Lambda \subset \bar{\Lambda} \Leftrightarrow \Lambda^\vee \supset \bar{\Lambda}^\vee$$

holds. For proofs and more information concerning the dual lattice see [Reg04] and [Con09].

The special case we are interested in is that, where the vector space V is some $\mathbb{C}^n$. In this case we slightly change the definition of the dual lattice to

$$\Lambda^\vee := \left\{ y \in \mathbb{C}^n \,\middle|\, \forall x \in \Lambda, \langle x, \overline{y} \rangle = \sum_i x_i y_i \in \mathbb{Z} \right\}.$$

Consequently, the dual basis is characterized by $\langle b_i, \overline{d_j} \rangle = \delta_{ij}$. All facts stated above still hold for these "conjugate" dual lattices. We do this small change, because we will later deal with ideals in rings of algebraic integers which embed to lattices in $\mathbb{C}^n$. Moreover, we will define a dual ideal which, as it turns out, embeds exactly to the dual lattice as we defined it now (see Section 1.4.4).

1.3.1 Decoding in Lattices

In the following we present a certain algorithmic task working on lattices that we will often need in our applications. Let $V \subseteq \mathbb{C}^n$ be some $\mathbb{R}$ vector space, $\Lambda \subset V$ a fixed lattice and $x \in V$ be an unknown short vector. Given the input $t = x \mod \Lambda$, the algorithm should recover x. Amongst several possible solutions for this task, we choose a slight extension of the "rounding off" algorithm originally due to Babai (see [Bab86]). We refer to it as a decoding algorithm, since in applications, it will play the main role when decrypting (decoding) elements. The algorithm works as follows. Let $\{v_i\}$ be a fixed set of n linearly independent vectors in the dual lattice $\Lambda^\vee$ and denote the dual basis of $\{v_i\}$ by $\{b_i\}$. Since $\mathrm{span}(\{v_i\}) \subset \Lambda^\vee$, the lattice Λ' generated by the dual basis $\{b_i\}$ is a superlattice of Λ. Thus, given $t = x \mod \Lambda$, we can express $t \bmod \Lambda'$ in the basis $\{b_i\}$ as $\sum_{i=1}^n c_i b_i$ for suitable $c_i \in \mathbb{R}/\mathbb{Z}$. To be more precise, by the definition of the dual basis, the coefficients are given by $c_i = \langle t, \overline{v_i} \rangle \mod 1$. Then, we output $\sum_i [\![c_i]\!] b_i$, where $[\![c]\!] \in \mathbb{R}$ for $c \in \mathbb{R}/\mathbb{Z}$ denotes the unique representative $\mathbf{c} \in (c + \mathbb{Z}) \cap [-1/2, 1/2)$.

Claim 1.3.4. *With the above settings, the "rounding off" algorithm on input $x \bmod \Lambda$ outputs x, if and only if, all the coefficients $a_i = \langle x, \overline{v_i} \rangle \in \mathbb{R}$ in the linear combination $x = \sum_{i=1}^n a_i b_i$ are in $[-1/2, 1/2)$.*

Remark 1.3.5. By the Cauchy-Schwarz inequality we have that

$$|a_i| = |\langle x, \overline{v_i} \rangle| \leq \|x\|_2 \cdot \|\overline{v_i}\|_2 = \|x\|_2 \cdot \|v_i\|_2.$$

Thus, if $M = \max_{i=1,\ldots,n} \|v_i\|_2$ is the maximum length of the elements v_i, each coefficient a_i is bounded by $M \cdot \|x\|_2$. Together with Claim 1.3.4 this implies that the length of the unknown vectors x we can successfully decode depends inversely on the maximum length of the vectors v_i from the dual lattice $\Lambda^\vee$.

1.3.2 Discretization in Lattices

Discretization is an algorithmic task working on lattices. Let V be some $\mathbb{R}$ vector space and $\Lambda = \mathcal{L}(B) \subset V$ a lattice represented by a basis $B = \{b_i\}_{i=1,\ldots,n}$. Given a point $x \in V$, and a point $c \in V$ representing a lattice coset $c + \Lambda$, we want to compute a discretized point $y \in c + \Lambda$, written $y \leftarrow \lfloor x \rceil_{c+\Lambda}$, such that the length of $y - x$ is small. This is done by sampling a short offset vector $f \in c' + \Lambda = (c - x) + \Lambda$ and output $y = x + f$. There are several ways of sampling the vector f. We call a procedure for sampling f *valid*, if it is efficient and depends

only on the desired coset $c' + \Lambda$ and not on any particular representative. For our purposes we will only use the "coordinate-wise randomized rounding" method which is a valid method.

The "coordinate-wise randomized rounding" is a simple and efficient way of sampling f. First, we represent the coset representative c' in the basis B as $c' = \sum_{i=1}^{n} a_i b_i \mod \Lambda$ for some coefficients $a_i \in [0, 1)$. Then, we choose f_i randomly and independently from $\{a_i - 1, a_i\}$ such that the expectation of f_i is zero. That is, if $p := P[f_i = a_i]$ is the probability that $f_i = a_i$, then $p \cdot a_i + (1 - p) \cdot (a_i - 1)$ has to be zero. Hence, we have $p = 1 - a_i$. Now define $f := \sum_i f_i b_i \in c' + \Lambda$. Since the values a_i are independent from the representative c', the validity of this method follows immediately.

1.4 Algebraic Number Theory

In this section we give a short introduction to the topics of algebraic number theory we will need throughout this work. In particular, we will focus on cyclotomic number fields as these are the only fields we use later on. For a more detailed and complete treatment of this topic, see e.g. [Lan94] or any other introductory book on the subject. For more insight, in particular on cyclotomic fields, see [Was82].

1.4.1 Cyclotomic Number Fields

Definition 1.4.1. Let K be any finite field extension over $\mathbb{Q}$. Then we call K an *algebraic number field*. Further let m be any positive integer and ζ_m an element of multiplicative order m, i.e., a primitive m-th root of unity. Then the algebraic number field $K = \mathbb{Q}(\zeta_m)$, obtained by adjoining ζ_m to $\mathbb{Q}$, is called the m-th *cyclotomic number field*.

Remark 1.4.2. Formally speaking, ζ_m is some abstract element in an algebraic closure of $\mathbb{Q}$. However, one can show that every algebraic closure of $\mathbb{Q}$ can be embedded into $\mathbb{C}$. Thus, ζ_m can also be viewed as some particular value in $\mathbb{C}$, which is often the better way to think of ζ_m in terms of intuition and understanding.

The minimal polynomial of ζ_m is the m-th *cyclotomic polynomial*

$$\Phi_m(X) = \prod_{i \in \mathbb{Z}_m^*} (X - \omega_m^i) \in \mathbb{Z}[X],$$

where ω_m is any primitive m-th root of unity in $\mathbb{C}$, e.g., $\omega_m = \exp(2\pi\sqrt{-1}/m)$. This yields a natural isomorphism between K and $\mathbb{Q}[X]/(\Phi_m(X))$, given by $\zeta_m \mapsto X$. The degree of $\Phi_m(X)$ is $n = |\mathbb{Z}_m^*| = \varphi(m)$, where φ is the Euler totient function. Thus, we can view K as an n-dimensional vector space over $\mathbb{Q}$. A canonical basis is given by the *power basis* $(\zeta_m^j)_{j \in [n]} = (1, \zeta_m, \ldots, \zeta_m^{n-1}) \in K^{[n]}$, which corresponds to the monomial basis $(1, X, \ldots, X^{n-1})$ of $\mathbb{Q}[X]/(\Phi_m(X))$.

A common way to represent an element $a \in K \cong \mathbb{Q}[X]/(\Phi_m(X))$ is to take the coordinate vector, in the monomial basis, of the polynomial $p_a \in \mathbb{Q}[X]/(\Phi_m(X))$ corresponding to a. Depending on the form of the minimal polynomial $\Phi_m(X)$, modulo reduction in $\mathbb{Q}[X]/(\Phi_m(X))$ might become rather complex. For m being a prime or a prime power the minimal polynomial looks "nice" and has small coefficients, i.e., for primes p we have $\Phi_p(X) = 1 + X + X^2 + \cdots + X^{p-1}$ and for prime powers m we have $\Phi_m(X) = \Phi_p(X^{m/p})$.

However, for arbitrary m, generally speaking, the range of the coefficient grow with the number of prime divisors of m. For example, $\Phi_6(X) = X^2 - X + 1$, $\Phi_{3\cdot5\cdot7}(X)$ has 33 monomials with coefficients $-2, -1$ and 1 and $\Phi_{3\cdot5\cdot7\cdot11\cdot13}(X)$ has coefficients of magnitude up to 22. This behavior causes some operations, like addition and multiplication, to be inefficient for a wide choice of integers m. Fortunately, this will never be a concern for us, because we choose a different approach to represent elements in K, which is independent of the minimal polynomial. Still we want to reduce our considerations to the case of prime power cyclotomics. This is done via the tensor product.

Definition 1.4.3. Let K and L be two algebraic number fields. For arbitrary $a \in K$ and $b \in L$ we say that $a \otimes_{\mathbb{Q}} b$ is a *pure tensors*. The *tensor product of the fields* $K \otimes_{\mathbb{Q}} L$ is defined as the set of all $\mathbb{Q}$-linear combinations of pure tensors $a \otimes_{\mathbb{Q}} b$ for $a \in K$ and $b \in L$. Thereby, addition and scalar multiplication are defined as follows. For pure tensors $(a_1 \otimes_{\mathbb{Q}} b_1)$ and $(a_2 \otimes_{\mathbb{Q}} b_2)$ the sum is defined as the element

$$(a_1 \otimes_{\mathbb{Q}} b_1) + (a_2 \otimes_{\mathbb{Q}} b_2)$$

and can not be further simplified. In the special cases $b_1 = b_2$ or $a_1 = a_2$ we define

$$\begin{aligned}(a_1 \otimes_{\mathbb{Q}} b) + (a_2 \otimes_{\mathbb{Q}} b) &= (a_1 + a_2) \otimes_{\mathbb{Q}} b,\\ (a \otimes_{\mathbb{Q}} b_1) + (a \otimes_{\mathbb{Q}} b_2) &= a \otimes_{\mathbb{Q}} (b_1 + b_2).\end{aligned}$$

For any scalar $\lambda \in \mathbb{Q}$ and a pure tensor $(a \otimes_{\mathbb{Q}} b)$ we define scalar multiplication by

$$\lambda(a \otimes_{\mathbb{Q}} b) = (\lambda a) \otimes_{\mathbb{Q}} b = a \otimes_{\mathbb{Q}} (\lambda b).$$

Using the mixed-product property we can also define a multiplication on $K \otimes_{\mathbb{Q}} L$ via

$$(a_1 \otimes_{\mathbb{Q}} b_1)(a_2 \otimes_{\mathbb{Q}} b_2) = (a_1 a_2) \otimes_{\mathbb{Q}} (b_1 b_2).$$

With these definitions it is also possible to view $\otimes_{\mathbb{Q}}$ as a $\mathbb{Q}$-bilinear form

$$\begin{aligned}\otimes_{\mathbb{Q}} : K \times L &\to K \otimes_{\mathbb{Q}} L\\ (a, b) &\mapsto a \otimes_{\mathbb{Q}} b\end{aligned}$$

which satisfies the mixed-product property.

In the following, if it is clear what we mean, we will drop the index $\mathbb{Q}$ of $\otimes_{\mathbb{Q}}$ for the sake of simplicity.

Note that $K \otimes L$ is not always a field, because it may lack multiplicative inverses. However, it will always be an algebra. In the case of cyclotomic number fields, the tensor product is again a field. Indeed we have the following results.

Proposition 1.4.4. *Let m be any positive integer with the prime power factorization $m = \prod_{l=0}^{s} m_l$. Then $\zeta_m = \prod_{l=0}^{s} \zeta_{m_l}$ and*

$$\mathbb{Q}(\zeta_m) = \mathbb{Q}\left(\prod_{l=0}^{s} \zeta_{m_l}\right) = \mathbb{Q}(\zeta_{m_0}, \ldots, \zeta_{m_s}) = \prod_{l=0}^{s} \mathbb{Q}(\zeta_{m_l}).$$

Proof. A proof can be found in e.g. [Lan94]. □

Proposition 1.4.5. *Let m be any positive integer with the prime power factorization $m = \prod_{l=0}^{s} m_l$. Then $K = \mathbb{Q}(\zeta_m) = \prod_{l=0}^{s} \mathbb{Q}(\zeta_{m_l})$ is isomorphic as a field to the tensor product $\bigotimes_{l=0}^{s} \mathbb{Q}(\zeta_{m_l})$ via the correspondence*

$$\prod_{l=0}^{s} a_l \leftrightarrow \left(\bigotimes_{l=0}^{s} a_l \right),$$

where $a_l \in \mathbb{Q}(\zeta_{m_l})$.

Proof. From Proposition 1.4.4 we know that each $a \in K$ can be uniquely rewritten to a product $\prod_{l=0}^{s} a_l$, for suitable $a_l \in \mathbb{Q}(\zeta_{m_l})$. Hence, in order to prove this result, one must only check that the given map is indeed a isomorphism of fields. It is clearly bijective and the homomorphic properties are checked by straightforward computations. □

Suppose we have two cyclotomic fields K and L and let $\vec{a} = (a_0, \ldots, a_{n-1})$ and $\vec{b} = (b_0, \ldots, b_{m-1})$ be the respective power bases. Any tensor $c \otimes d \in K \otimes L$ might be rewritten to

$$c \otimes d = \left(\sum_{i \in [n]} c_i a_i \right) \otimes \left(\sum_{j \in [m]} d_j b_j \right),$$

where $(c_i)_{i \in [n]}$ and $(d_j)_{j \in [m]}$ are the respective coefficients of the representations of c and d in the respective bases. By bilinearity of $\otimes$, this can be further transformed into

$$\left(\sum_{i \in [n]} c_i a_i \right) \otimes \left(\sum_{j \in [m]} d_j b_j \right) = \sum_{i \in [n]} \sum_{j \in [m]} c_i d_j (a_i \otimes b_j).$$

The latter is a unique representation of $c \otimes d$ implying that the set of tensors $a_i \otimes b_j$ for $i \in [n], j \in [m]$ is a basis of $K \otimes L$.

The tensor product can also be defined more general for arbitrary $\mathcal{R}$-modules and $\mathcal{R}$-algebras where $\mathcal{R}$ is some commutative ring with one. Then, the tensor product consists of $\mathcal{R}$-linear combinations and $\otimes_{\mathcal{R}}$ is $\mathcal{R}$-bilinear. In the case of $\mathcal{R}$-modules, the mixed-product property is omitted since there exists no multiplication for arbitrary module elements. In this setting the tensor product is itself again a $\mathcal{R}$-module or $\mathcal{R}$-algebra respectively. Also, if there exist some $\mathcal{R}$-bases for the modules or algebras, then we get a basis for the respective tensor product in the same way as above in the cyclotomic field case. We need this more general definition for the special case, where $\mathcal{R} = \mathbb{Z}$, since we will consider several $\mathbb{Z}$-modules and their tensor product later in this work.

1.4.2 Embeddings and Geometry

In the following, let $K = \mathbb{Q}(\zeta_m)$ be the m-th cyclotomic number field for some positive integer m. From Galois theory we know that K has exactly $|K/\mathbb{Q}| = n = \varphi(m)$ distinct $\mathbb{Q}$-homomorphism $\{\sigma_i\}_{i \in \mathbb{Z}_m^*}$ from K to $\mathbb{C}$ that fix every element of $\mathbb{Q}$. We call the σ_i also embeddings, since they are injective. To be more precise, let ω_m be any fixed primitive m-th root of unity in $\mathbb{C}$. Then for each $i \in \mathbb{Z}_m^*$ there is an embedding σ_i defined by $\sigma_i(\zeta_m) = \omega_m^i$.[1] Note that $\omega_m^{m-i} = \omega_m^{-i} = \overline{\omega_m^i}$, so the embeddings come in complex conjugated pairs $\sigma_i = \overline{\sigma_{m-i}}$.

[1] In all our implementations, ω_m will always be the "first" primitive root of unity on the unit circle in $\mathbb{C}$, i.e., $\omega_m = \exp(2\pi i/m)$.

Definition 1.4.6 (Canonical Embedding)**.** Let $K = \mathbb{Q}(\zeta_m)$ for some positive integer m. Further, for $i \in \mathbb{Z}_m^*$ let $\sigma_i : K \to \mathbb{C}$ be the $n = \varphi(m)$ distinct $\mathbb{Q}$-homomorphisms from K to $\mathbb{C}$. Then we define the *canonical embedding* $\sigma : K \to \mathbb{C}^{\mathbb{Z}_m^*}$ by

$$\sigma(a) := (\sigma_i(a))_{i \in \mathbb{Z}_m^*}.$$

The fact that the $\mathbb{Q}$-homomorphisms σ_i come in complex conjugated pairs gives motivation to define a certain subspace of $\mathbb{C}^{\mathbb{Z}_m^*}$, which has also a complex conjugated paired structure.

Definition 1.4.7. For any integer m define the subspace $H \subset \mathbb{C}^{\mathbb{Z}_m^*}$ as

$$H := \left\{ \mathbf{x} \in \mathbb{C}^{\mathbb{Z}_m^*} \,|x_i = \overline{x_{m-i}}, \forall i \in \mathbb{Z}_m^* \right\}.$$

Remark 1.4.8. As a subspace H inherits the standard inner product and the ℓ_2 and ℓ_∞ norms of $\mathbb{C}^{\mathbb{Z}_m^*}$. Concretely, we have $\|\mathbf{x}\|_2 = \sum_i (|x_i|)^{1/2} = \sqrt{\langle \mathbf{x}, \mathbf{x} \rangle}$ and $\|\mathbf{x}\|_\infty = \max_i |x_i|$ for all $\mathbf{x} \in H$. Furthermore, H is isomorphic to $\mathbb{R}^{[n]}$ as an inner product space. This can be seen via the $\mathbb{Z}_m^* \times [n]$ unitary basis matrix $B = \frac{1}{\sqrt{2}} \begin{pmatrix} I & \sqrt{-1}J \\ J & -\sqrt{-1}I \end{pmatrix}$ of H, where I is the identity matrix and J is the reversed identity matrix, both of dimension $n/2$ (note that the result of Euler's totient function is always even, in particular, $n = \varphi(m)$ is even).

The space H turns into a ring, if we equip it with component-wise addition and multiplication. The canonical embedding σ now maps into H and is a ring homomorphism. Indeed for arbitrary $a, b \in K$ we have

$$\sigma(a+b) = (\sigma_i(a+b))_{i \in \mathbb{Z}_m^*} = (\sigma_i(a) + \sigma_i(b))_{i \in \mathbb{Z}_m^*} = (\sigma_i(a))_{i \in \mathbb{Z}_m^*} + (\sigma_i(b))_{i \in \mathbb{Z}_m^*} = \sigma(a) + \sigma(b)$$

and

$$\sigma(a \cdot b) = (\sigma_i(a \cdot b))_{i \in \mathbb{Z}_m^*} = (\sigma_i(a) \cdot \sigma_i(b))_{i \in \mathbb{Z}_m^*} = (\sigma_i(a))_{i \in \mathbb{Z}_m^*} \odot (\sigma_i(b))_{i \in \mathbb{Z}_m^*} = \sigma(a) \odot \sigma(b)$$

where $\odot$ means component-wise multiplication. Further, since all σ_i are injective, the canonical embedding σ is also injective.

Through the canonical embedding we can endow K with a canonical geometry. For $a \in K$ we define the ℓ_2 norm as $\|a\|_2 = \|\sigma(a)\|_2 = \sum_{i \in \mathbb{Z}_m^*} (|\sigma_i(a)|)^{1/2}$ and the ℓ_∞ norm as $\|a\|_\infty = \max_{i \in \mathbb{Z}_m^*} |\sigma_i(a)|$. Since multiplication in H is component-wise we have that for any $a, b \in K$

$$\|a \cdot b\| \leq \|a\|_\infty \cdot \|b\|,$$

where $\|\cdot\|$ is either the ℓ_2 or ℓ_∞ norm. This means that we have a bound on how much an element expands any other by multiplication. For example, multiplying with any power $\zeta = \zeta_m^k$ of ζ_m does not expand the element at all. As $\sigma_i(\zeta) = \sigma_i(\zeta_m^k) = \sigma_i(\zeta_m)^k = \omega_m^{ik}$ is still a root of unity in $\mathbb{C}$, we have $\|\zeta\|_\infty = 1$ and $\|\zeta\|_2 = \sqrt{n}$.

Two important mappings from field theory are the trace and the norm. They are defined as follows.

Definition 1.4.9. Let $K = \mathbb{Q}(\zeta_m)$ for some positive integer m. Further, for $i \in \mathbb{Z}_m^*$ let $\sigma_i : K \to \mathbb{C}$ be the $n = \varphi(m)$ distinct $\mathbb{Q}$-homomorphisms from K to $\mathbb{C}$. Then we define the

trace $\mathrm{Tr} = \mathrm{Tr}_{K/\mathbb{Q}} : K \to \mathbb{Q}$ of $K/\mathbb{Q}$ as the sum off all existing $\mathbb{Q}$-homomorphisms from K to $\mathbb{C}$, i.e.,

$$\mathrm{Tr}_{K/\mathbb{Q}}(a) = \sum_{i \in Z_m^*} \sigma_i(a).$$

The *norm* $\mathrm{N} = \mathrm{N}_{K/\mathbb{Q}} : K \to \mathbb{Q}$ is defined as the product of all embeddings σ_i, i.e.,

$$\mathrm{N}_{K/\mathbb{Q}}(a) = \prod_{i \in Z_m^*} \sigma_i(a).$$

Remark 1.4.10. The fact that the trace and the norm map into $\mathbb{Q}$ although all embeddings map into $\mathbb{C}$ is not a trivial one. It needs a decent proof, which we omit at this point.

The trace is a $\mathbb{Q}$-linear map, since all embeddings σ_i are $\mathbb{Q}$-linear: $\mathrm{Tr}(a+b) = \mathrm{Tr}(a) + \mathrm{Tr}(b)$ and $\mathrm{Tr}(\lambda \cdot a) = \lambda \cdot \mathrm{Tr}(a)$ for all $a, b \in K$ and $\lambda \in \mathbb{Q}$. For a product in K we have

$$\mathrm{Tr}(a \cdot b) = \sum_{i \in \mathbb{Z}_m^*} (\sigma_i(a) \cdot \sigma_i(b)) = \left\langle \sigma(a), \overline{\sigma(b)} \right\rangle_{\mathbb{C}}, \tag{1.1}$$

so via $\mathrm{Tr}(a, b) := \mathrm{Tr}(a \cdot b)$ we can define a symmetric bilinear form that equals the inner product of the embedding and the complex conjugate embedding of a and b respectively. Since all embeddings are multiplicative, the norm is also multiplicative, i.e., $\mathrm{N}(a \cdot b) = \mathrm{N}(a) \cdot \mathrm{N}(b)$.

When viewing $K = \mathbb{Q}(\zeta_m)$ as the tensor product $\bigotimes_{l=0}^{s} K_l$ as in Proposition 1.4.5, where $m = \prod_{l=0}^{s} m_l$ is the prime power factorization of m and $K_l = \mathbb{Q}(\zeta_{m_l})$, we can also view the embedding σ of K as the tensor (Kronecker) product of the canonical embeddings $\sigma^{(l)}$ of K_l. Here we define σ for the tensor product K via the natural isomorphism from Proposition 1.4.5. Then, for a tensor $\bigotimes_{l=0}^{s} a_l$ for $a_l \in K_l$ we have

$$\sigma\left(\bigotimes_{l=0}^{s} a_l\right) = \sigma\left(\prod_{l=0}^{s} a_l\right) = \prod_{l=0}^{s} \sigma(a_l) = \bigodot_{l=0}^{s} (\sigma_i(a_l))_{i \in Z_m^*}.$$

Now via the definition of the Kronecker product and the Chinese remainder theorem, which implies that $\mathbb{Z}_m^* \cong \prod_{l=0}^{s} \mathbb{Z}_{m_l}^*$, we get

$$\bigodot_{l=0}^{s} (\sigma_i(a_l))_{i \in Z_m^*} = \bigotimes_{l=0}^{s} (\sigma_j(a_l))_{j \in Z_{m_l}^*} = \bigotimes_{l=0}^{s} \sigma^{(l)}(a_l).$$

Together, we get the relation

$$\sigma\left(\bigotimes_{l=0}^{s} a_l\right) = \bigotimes_{l=0}^{s} \sigma^{(l)}(a_l). \tag{1.2}$$

Here $\bigodot$ again means coordinate-wise multiplication. A nice implication of this fact is the decomposition of the trace. Let $\mathbf{1} \in \mathbb{R}^{[n]}$ be the all ones vector. Then we have

$$\begin{aligned} \mathrm{Tr}_{K/\mathbb{Q}}\left(\bigotimes_{l=0}^{s} a_l\right) &= \left\langle \sigma\left(\bigotimes_{l=0}^{s} a_l\right), \mathbf{1} \right\rangle = \left\langle \bigotimes_{l=0}^{s} \sigma^{(l)}(a_l), \mathbf{1} \right\rangle \\ &= \prod_{l=0}^{s} \left\langle \sigma^{(l)}(a_l), \mathbf{1} \right\rangle = \prod_{l=0}^{s} \mathrm{Tr}_{K_l/\mathbb{Q}}(a_l). \end{aligned} \tag{1.3}$$

In later applications we will sample Gaussians from the continuous Gaussian distribution N_r over K for some standard deviation r. This is done via sampling from N_r over H and a pull back to K with the canonical embedding σ. Actually, the pull back of a continuous Gaussian from H will live in the field tensor product $K_{\mathbb{R}} = K \otimes \mathbb{R}$. Here tensoring with $\mathbb{R}$ is essentially an expansion of the scalar field from $\mathbb{Q}$ to $\mathbb{R}$. This means that taking any $\mathbb{Q}$-basis of K and treating it as a $\mathbb{R}$-basis yields a field isomorphic to $K_{\mathbb{R}}$. Also, the fact that we are now allowed to take arbitrary elements from $\mathbb{R}$ as coefficients in basis representations of elements in $K_{\mathbb{R}}$ implies that the natural continuation $\sigma : K_{\mathbb{R}} \to H$ is surjective, hence an isomorphism. Alternatively, one can argue that $K_{\mathbb{R}}$ is isomorphic to $\mathbb{R}^{[n]}$ as well as H, so they have the same cardinality. The embedding $\sigma : K_{\mathbb{R}} \to H$ is still injective and a ring homomorphism, since all continuations $\sigma_i : K_{\mathbb{R}} \to \mathbb{C}$ for $i \in \mathbb{Z}_m^*$ are injective $\mathbb{R}$-homomorphisms. Consequently, σ has to be surjective and therefore an isomorphism.

1.4.3 Ring of Integers and its Ideals

Definition 1.4.11. Let K be an arbitrary algebraic number field. We denote with $R \subset K$ the set of all algebraic integers in K. This set is a subring of K and is called the *ring of integers* of K (often also denoted as $\mathcal{O}_K$).

One can show that the trace and the norm of an algebraic integer in K is again an algebraic integer in the image $\mathbb{Q}$. Thus, we have the induced maps $\mathrm{Tr}, \mathrm{N} : R \to \mathbb{Z}$.

Let K be the m-th cyclotomic number field $\mathbb{Q}(\zeta_m)$ for some positive integer m. Then the ring of integers is given by $R = \mathbb{Z}[\zeta_m] \cong \mathbb{Z}[X]/\Phi_m(X)$. R has the power basis $\{\zeta_m^j\}_{j\in[n]}$, where $n = \varphi(m)$, as a $\mathbb{Z}$-basis. Recall that we often view K as the tensor product $K = \bigotimes_{l=0}^{s} K_l$, where $K_l = \mathbb{Q}(\zeta_{m_l})$ and $m_0, \ldots, m_s$ are the distinct prime power divisors of m (cf. Proposition 1.4.5). Similarly, we can view R as the tensor product $R = \bigotimes_{l=0}^{s} R_l$ for the rings of algebraic integers R_l in K_l.

Next we introduce the notion of ideals in rings. One can think of ideals in arbitrary rings $\mathcal{R}$ as some sort of counterpart to primes and numbers in $\mathbb{Z}$. Just as there are prime numbers in $\mathbb{Z}$, there are prime ideals in $\mathcal{R}$. Furthermore, one can define arithmetic operations on ideals, similarly to usual arithmetic in $\mathbb{Z}$. In particular, if $\mathcal{R}$ is a ring of integers (or more general a Dedekind ring), this correspondence gets even stronger. In this case there is a unique prime ideal factorization for ideals.

Definition 1.4.12. Let $\mathcal{R}$ be a commutative ring with one and $\mathcal{I} \subseteq \mathcal{R}$ be any non-trivial subgroup that is closed under multiplication with $\mathcal{R}$, i.e., $0 \in \mathcal{I}$, $a - b \in \mathcal{I}$ and $r \cdot a \in \mathcal{I}$ for all $a, b \in \mathcal{I}$ and $r \in \mathcal{R}$. Then $\mathcal{I}$ is called an *ideal* in $\mathcal{R}$.[2]

Remark 1.4.13. For a given subset $X \subseteq \mathcal{R}$ we can define an ideal $\mathcal{I}$ by the set

$$\mathcal{I} := \left\{ \sum_{i\in I} r_i x_i \,\middle|\, I \subset \mathbb{N} \text{ finite}, r_i \in \mathcal{R}, x_i \in X \right\}.$$

We say that $\mathcal{I}$ is the *ideal generated by* X and write $\mathcal{I} = \langle X \rangle$.

Definition 1.4.14. Let $\mathcal{R}$ be a commutative ring with one. Any ideal $\mathcal{I} \subseteq \mathcal{R}$, which is generated by a single element $u \in \mathcal{R}$ is called *principle*. Next to $\mathcal{I} = \langle u \rangle$ we also write $\mathcal{I} = u\mathcal{R}$.

[2] Many authors define ideals including the trivial subgroup $\{0\}$. We decided to not define $\{0\}$ as an ideal, since many of the result concerning ideals we use, do only hold if the trivial ideal is excluded.

A generator of a principle ideal $\mathcal{I} = \langle u \rangle$ is in general not unique. Indeed, if $e \in \mathcal{R}$ is a unit, then $\mathcal{I} = \langle u \rangle = \langle eu \rangle$. So, the generator u is only unique up to multiplication with units in $\mathcal{R}$.

Now we return to the special case where $K = \mathbb{Q}(\zeta_m)$ is the m-th cyclotomic number field and the ring $\mathcal{R}$ is the ring of integers $R \subset K$. Here we distinguish two kinds of ideals.

Definition 1.4.15. Let $K = \mathbb{Q}(\zeta_m)$ be the m-th cyclotomic number field and $R \subset K$ its ring of integers. Any ideal $\mathcal{I} \subseteq R$ is called *integral*.

An integral ideal $\mathcal{I}$ in R is always finitely generated. In fact, one can show that there exist two elements which generate $\mathcal{I}$. Another theorem from number theory states that there is a $\mathbb{Z}$-basis of $\mathcal{I}$ of size $n = \varphi(m)$ for each ideal $\mathcal{I}$ in R. For example, if $\mathcal{I} = \langle u \rangle$ and B is any $\mathbb{Z}$-basis of R, then uB is a $\mathbb{Z}$-basis for $\mathcal{I}$.

Definition 1.4.16. Let $K = \mathbb{Q}(\zeta_m)$ be the m-th cyclotomic number field and $R \subset K$ its ring of integers. A *fractional ideal* $\mathcal{I} \subseteq K$ is a finitely generated R-submodule in K.

The name fractional ideal might be misleading, since fractional ideals are in general no ideals. However, for each fractional ideal $\mathcal{I}$ in K there exists some element $d \in R$ such that $d\mathcal{I} \subseteq R$ is an integral ideal. A fractional ideal $\mathcal{I}$ is called *principal*, if there is some element $u \in K$ such that $\mathcal{I} = uR$. Note that any integral ideal is also a fractional ideal. A key fact from algebraic number theory is that any fractional ideal $\mathcal{I}$ embeds under the canonical embedding σ as a full-rank lattice $\sigma(\mathcal{I})$ in H. We call $\sigma(\mathcal{I})$ an *ideal lattice*. For convenience, we often identify the ideal $\mathcal{I}$ with its embedded lattice $\sigma(\mathcal{I})$ and define in this way the usual lattice quantities for fractional ideals.

The notion of addition and multiplication can be adapted for fractional ideals. We define the sum $\mathcal{I} + \mathcal{J}$ of two fractional ideals as the set of all $a + b$ for $a \in \mathcal{I}, b \in \mathcal{J}$, and the product $\mathcal{I}\mathcal{J}$ as the fractional ideal generated by all ab for $a \in \mathcal{I}, b \in \mathcal{J}$, i.e., the set of all finite sums of all ab. The set of all fractional ideals in K is an abelian group under multiplication. Thereby, we define the neutral element by $\langle 1 \rangle = R$ and the inverse as $\mathcal{I}^{-1} := \{x \in K \,|\, x\mathcal{I} \subseteq R\}$.

Before we continue with the norm for fractional ideals we introduce another important invariant for algebraic number fields K. The (absolute) *discriminant* $\Delta_K := \det(\sigma(R))^2$ of K is defined as the squared determinant of the lattice $\sigma(R)$. If $K = \mathbb{Q}(\zeta_m)$ is the m-th cyclotomic number field, the discriminant is given by

$$\Delta_K = \left(\frac{m}{\prod\limits_{\text{prime } p \mid m} p^{1/(p-1)}} \right)^n \leq n^n, \tag{1.4}$$

where $n = \varphi(m)$. This inequality is tight, if and only if, m is a power of two.

The notion of the field norm $\mathrm{N}_{K/\mathbb{Q}}$ is generalized to ideals in R by defining the norm of any integral ideal $\mathcal{I} \subseteq R$ as $\mathrm{N}(\mathcal{I}) = |R/\mathcal{I}|$, the index of $\mathcal{I}$ as a subgroup in R. This is a generalization in the sense that $\mathrm{N}(\langle u \rangle) = |\mathrm{N}(u)|$ for any $u \in R$ and $\mathrm{N}(\mathcal{I}\mathcal{J}) = \mathrm{N}(\mathcal{I})\mathrm{N}(\mathcal{J})$ for integral ideals $\mathcal{I}, \mathcal{J} \subseteq R$. For a fractional ideal $\mathcal{I} \subseteq K$ and an element $d \in R$ such that $d\mathcal{I} \subseteq R$, we define the norm as $\mathrm{N}(\mathcal{I}) = \mathrm{N}(d\mathcal{I})/|\mathrm{N}(d)|$. Note that the norm is always finite. One can show that the norm of a fractional ideal $\mathcal{I}$ and the determinant of the lattice $\sigma(\mathcal{I})$ are related by

$$\det(\sigma(\mathcal{I})) = \mathrm{N}(\mathcal{I}) \cdot \det(\sigma(R)).$$

Together with the definition of the discriminant this implies that

$$\det(\sigma(\mathcal{I})) = \mathrm{N}(\mathcal{I}) \cdot \sqrt{\Delta_K}. \tag{1.5}$$

Lemma 1.4.17. *Let $K = \mathbb{Q}(\zeta_m)$ be the m-th cyclotomic number field for some positive integer m. Denote by $n = \varphi(m)$ the degree of $K/\mathbb{Q}$. For any fractional ideal $\mathcal{I}$ in K the following inequality holds,*

$$\sqrt{n} \cdot \mathrm{N}^{1/n}(\mathcal{I}) \leq \lambda_1(\mathcal{I}) \leq \sqrt{n} \cdot \mathrm{N}^{1/n}(\mathcal{I}) \cdot \sqrt{\Delta_K^{1/n}}.$$

Proof. First, we prove the lower bound. Let x_{min} denote an element $\mathcal{I}$ where the minimum distance is taken, i.e., $\|x_{min}\|_2 = \lambda_1(\mathcal{I})$. For every nonzero element $a \in \mathcal{I}$ we have $|\mathrm{N}(a)| \geq \mathrm{N}(\mathcal{I})$. Therefore, the inequality

$$\sqrt{n} \cdot \mathrm{N}^{1/n}(\mathcal{I}) \leq \sqrt{n} \cdot |\mathrm{N}^{1/n}(x_{min})|$$

holds. By definition of the Norm, the right hand side of the above inequality is equal to

$$\sqrt{n} \cdot \left(\prod_{i \in \mathbb{Z}_m^*} |\sigma_i(x_{min})| \right)^{1/n}.$$

As this product contains n factors, the n-th root of it is the arithmetic mean of the elements $|\sigma_i(x_{min})|$. By the well known arithmetic-geometric-mean-inequality the above term can be further estimated to

$$\sqrt{n} \cdot \left(\prod_{i \in \mathbb{Z}_m^*} |\sigma_i(x_{min})| \right)^{1/n} \leq \sqrt{n} \cdot \frac{1}{n} \sum_{i \in \mathbb{Z}_m^*} |\sigma_i(x_{min})| = \frac{\sqrt{n}}{n} \|x_{min}\|_1.$$

Using the Cauchy-Schwarz inequality one can show that $\|x\|_1 \leq \sqrt{n}\|x\|_2$ for any $x \in \mathcal{I}$ (in fact, this is a common fact for norms in $\mathbb{R}^n$), so that

$$\frac{\sqrt{n}}{n} \|x_{min}\|_1 \leq \|x_{min}\|_2 = \lambda_1(\mathcal{I}).$$

For the upper bound we make use of Minkowski's theorem, which states that a symmetric convex set S with volume $\mathrm{vol}(S) > 2^n \det(\mathcal{I})$ must contain at least one element $x \in \mathcal{I} \setminus 0$.[3] Consider first the ℓ_∞ norm $\|x\|_\infty = \max_i |x_i|$. Let $\alpha := \min_{0 \neq x \in \mathcal{I}} \|x\|_\infty$ and suppose that $\alpha > \det(\mathcal{I})^{1/n}$. Consider the hypercube $C := \{x \in K \,|\, \|x\|_\infty < \alpha\}$ and notice that C is symmetric, convex and has th volume $\mathrm{vol}(C) = (2\alpha)^n > 2^n \det(\mathcal{I})$. Hence, by Minkowski's theorem, C contains a nonzero element x. But, by definition of C, we have $\|x\|_\infty < \alpha$, contradicting the minimality of α. Consequently, α cannot be smaller than $\det(\mathcal{I})^{1/n}$ and we have

$$\|x\|_\infty \leq \det(\mathcal{I})^{1/n}$$

for at least one $x \in \mathcal{I}$. Since the ℓ_∞ and the ℓ_2 norm are related by a factor $\sqrt{n}$, i.e., $\|x\|_2 \leq \sqrt{n}\|x\|_{2\infty}$, this implies

$$\|x\|_2 \leq \sqrt{n} \cdot \det(\mathcal{I})^{1/n}.$$

Now we conclude the proof by observing that Equation (1.5) just states the missing equality $\det(\mathcal{I}) = \mathrm{N}(\mathcal{I}) \cdot \sqrt{\Delta_K}$. □

[3] For more details on Minskowski's theorem, especially in relation with the minimum distance of lattices see [Mic14].

Just like we have a unique prime factorization for numbers in $\mathbb{Z}$, we have a prime ideal factorization for integral ideals in R. Any integral ideal $\mathfrak{p} \subsetneq R$ is called *prime* if whenever $ab \in \mathfrak{p}$ then $a \in \mathfrak{p}$ or $b \in \mathfrak{p}$. Now, one can show that each integral ideal $\mathcal{I} \subseteq R$ has a unique factorization into powers of prime ideals, i.e., $\mathcal{I} = \mathfrak{p}_1^{k_1} \cdot \ldots \cdot \mathfrak{p}_r^{k_r}$ for prime ideals $\mathfrak{p}_1, \ldots, \mathfrak{p}_r$ and positive integers $k_1, \ldots, k_r$. This prime ideal factorization can be extended to fractional ideals $\mathcal{I} \subseteq K$ similarly as we can extend the prime factorization from $\mathbb{Z}$ to $\mathbb{Q}$. This means, that each fractional ideal $\mathcal{I} \subseteq K$ has a unique factorization of prime ideals

$$\mathcal{I} = \mathfrak{p}_1^{k_1} \cdot \ldots \cdot \mathfrak{p}_r^{k_r},$$

where this time $k_1, \ldots, k_r$ are arbitrary integers (in particular, negative integers are allowed). Recall that we defined the inverse of some fractional ideal $\mathcal{I}$ as $\mathcal{I}^{-1} \coloneqq \{x \in K \,|x\mathcal{I} \subseteq R\}$.

Another property of integral ideals in R is that any ideal $\mathfrak{p} \in R$ is prime if and only if it is *maximal*. That is, $\mathfrak{p}$ has only R itself as a proper superideal of R, which in turn implies that the quotient $R/\mathfrak{p}$ is a finite field. Finally, we call two ideals $\mathcal{I}, \mathcal{J} \subseteq R$ *coprime* if $\mathcal{I} + \mathcal{J} = R$.

1.4.4 Duality

In Section 1.3 we defined the dual lattice for lattices in a vector space V. The dual lattice consists of elements in V whose inner products with elements in the initial lattice are integral. Furthermore, we mentioned in Section 1.4.3 that fractional ideals embed as lattices under the canonical embedding and in Section 1.4.2 that the bilinear form induced by the trace corresponds to the inner product in $\mathbb{C}^n$. Following this correspondence, we can define the notion of a dual ideal for fractional ideals in cyclotomic fields. We will see that dual ideals are closely related to dual lattices. For more details and further remarks we refer to [Con09].

For the rest of this section let $K = \mathbb{Q}(\zeta_m)$ be the m-th cyclotomic number field for some positive integer m and $R \subset K$ be its ring of integers. Further let $m = \prod_{l=0}^{s} m_l$ be the prime power factorization of m and $n = \varphi(m)$.

Definition 1.4.18 (Dual Ideal)**.** Let $\mathcal{I} \subseteq K$ be a fractional ideal in K. Then the *dual ideal* $\mathcal{I}^\vee$ is defined as

$$\mathcal{I}^\vee \coloneqq \{a \in K \,|\mathrm{Tr}(a\mathcal{I}) \subset \mathbb{Z}\}.$$

The dual ideal has a similar behavior to the dual lattice. The passage to the dual ideal is self inverse, i.e., $(\mathcal{I}^\vee)^\vee = \mathcal{I}$. Analogously to the dual basis for lattices, the dual basis $B^\vee = \{b_j^\vee\}_{j\in[n]}$ for any $\mathbb{Q}$-basis $B = \{b_j\}_{j\in[n]}$ of K is characterized by $\mathrm{Tr}(b_i \cdot b_j^\vee) = \delta_{ij}$. We have that $(B^\vee)^\vee = B$ and if B is a $\mathbb{Z}$-basis for any fractional ideal $\mathcal{I}$, then $B^\vee$ is a $\mathbb{Z}$-basis for the dual ideal $\mathcal{I}^\vee$. In particular, $\mathcal{I}^\vee$ is again a fractional ideal. A quick computation using Equation (1.1) shows that the dual ideal $\mathcal{I}^\vee$ actually embeds to the dual lattice $\sigma(\mathcal{I})^\vee$, i.e., $\sigma(\mathcal{I}^\vee) = \sigma(\mathcal{I})^\vee$.

When viewing K as the tensor product $K = \bigotimes_{l=0}^{s} K_l$ as in Proposition 1.4.5, so $K_l = \mathbb{Q}(\zeta_{m_l})$, linearity and the tensorial decomposition of the trace as in Equation (1.3) imply that $(\bigotimes_{l=0}^{s} B_l)^\vee = \bigotimes_{l=0}^{s} B_l^\vee$ for any $\mathbb{Q}$-bases B_l of K_l. In other words, the passage to the dual commutes with the tensor product. For fractional ideals $\mathcal{I}_l$ in K_l this means that $(\bigotimes_{l=0}^{s} \mathcal{I}_l)^\vee = \bigotimes_{l=0}^{s} \mathcal{I}_l^\vee$.

For integral ideals $\mathcal{I} \subseteq R$ we always have $\mathrm{Tr}(\mathcal{I}) \subset \mathbb{Z}$. Together with the fact that $R\mathcal{I} = \mathcal{I}$, this implies $R \subset \mathcal{I}^\vee$. In particular, we have $R \subset R^\vee$. Moreover, the dual ideal $R^\vee$ yields a nice relation between $\mathcal{I}^\vee$ and $\mathcal{I}^{-1}$ for arbitrary fractional ideals $\mathcal{I}$, namely $\mathcal{I}^\vee = \mathcal{I}^{-1} \cdot R^\vee$.

The factor $R^\vee$ is often called the *codifferent* and plays an important role in ring-LWE and its applications. Since $R^\vee$ embeds to the dual lattice of $\sigma(R)$ we have that $\det(\sigma(R^\vee)) = \det(\sigma(R))^{-1}$. Together with Equation (1.5) we get

$$\mathrm{N}(R^\vee) = \det(\sigma(R^\vee)) \cdot \det(\sigma(R))^{-1} = \det(\sigma(R))^{-2} = \Delta_K^{-1}. \tag{1.6}$$

In the following we want to show that the codifferent is actually a principal ideal and that $(R^\vee)^{-1} \subset R$ is an integral ideal. We start with a useful Lemma.

Lemma 1.4.19. *Let m be a prime power of some prime p and j an integer. Further let $m' = m/p$. Then we can characterize the trace of the j-th power of ζ_m as*

$$\mathrm{Tr}(\zeta_m^j) = \begin{cases} \varphi(p) \cdot m', & \text{if } j = 0 \mod m, \\ -m', & \text{if } j = 0 \mod m',\ j \neq 0 \mod m, \\ 0, & \text{otherwise.} \end{cases}$$

Proof. In the first case we have $j = 0 \mod m$, thus $\zeta_m^j = 1$. Then, $\mathrm{Tr}(\zeta_m^j) = |K/\mathbb{Q}| = \varphi(m) = \varphi(p) \cdot m'$.

Otherwise, let $d = \gcd(j, m)$ and $\tilde{m} = m/d$. Then we have the field extensions

$$K \ \supset \ \mathbb{Q}(\zeta_{\tilde{m}}) \ \supset \ \mathbb{Q}.$$

In this case, we know for all $a \in K$ that $\mathrm{Tr}_{K/\mathbb{Q}}(a) = [K : \mathbb{Q}(\zeta_{\tilde{m}})] \cdot \mathrm{Tr}_{\mathbb{Q}(\zeta_{\tilde{m}})/\mathbb{Q}}(a)$. Since $m = p^k$ is a prime power, we have $\varphi(m) = (p-1)p^{k-1}$. Now d can itself be only a prime power, say $d = p^s$ for some $0 \leq s \leq (k-1)$. Then $\tilde{m} = p^{k-s}$ and we can write

$$\varphi(m) = (p-1)p^{k-s-1}p^s = \varphi(p^{k-s}) \cdot p^s = \varphi(\tilde{m}) \cdot d.$$

Moreover, we know that $[K : \mathbb{Q}] = \varphi(m)$ and $[\mathbb{Q}(\zeta_{\tilde{m}})] = \varphi(\tilde{m})$ which yields $[K : \mathbb{Q}(\zeta_{\tilde{m}})] = d$. If we rewrite $\zeta_m = \zeta_{\tilde{m}}^{\tilde{m}/m} = \zeta_{\tilde{m}}^{1/d}$, we have

$$\mathrm{Tr}(\zeta_m^j) = d \cdot \mathrm{Tr}_{\mathbb{Q}(\zeta_{\tilde{m}})/\mathbb{Q}}(\zeta_{\tilde{m}}^{j/d}).$$

But, d is the greatest common divisor of j and m, which implies that $m/d = \tilde{m}$ and j/d are coprime. Hence $\zeta_{\tilde{m}}^{j/d}$ is a primitive $\tilde{m}$-th root of unity and the $\varphi(\tilde{m})$ distinct embeddings $\{\sigma_i\}_{i \in \mathbb{Z}_{\tilde{m}}^*}$ of $\mathbb{Q}(\zeta_{\tilde{m}})/\mathbb{Q}$ permute the primitive $\tilde{m}$-th roots of unity in $\mathbb{C}$. Consequently, $\mathrm{Tr}_{\mathbb{Q}(\zeta_{\tilde{m}})/\mathbb{Q}}(\zeta_{\tilde{m}}^{j/d})$ is given by the sum of all primitive $\tilde{m}$-th roots of unity. If $j = 0 \mod m'$ and $j \neq 0 \mod m$ this sum is exactly -1, since $d = m'$ and $\tilde{m} = p$ is prime. Otherwise, the sum is zero, as required. □

Next we show that in the case of prime powers the dual ideal $R^\vee$ is principal with a simple generator.

Lemma 1.4.20. *Let m be a prime power of some prime p and $m' = m/p$. Further define $g := 1 - \zeta_p \in R = \mathbb{Z}[\zeta_m]$. Then we have*

(i) $R^\vee = \langle g/m \rangle$,

(ii) $p/g \in R$,

(iii) $\langle g \rangle$ *and* $\langle p' \rangle$ *are coprime for every prime* $p' \neq p$.

Proof. Ad (i): First we check that g/m is indeed an element of $R^\vee$. That is the case, if $\mathrm{Tr}(\zeta_m^j \cdot g/m)$ is in $\mathbb{Z}$ for each $j \in [\varphi(m)]$. It is sufficient to check this only for powers of ζ_m, since $\{\zeta_m^j\}_{j\in[\varphi(m)]}$ is a basis of R and the trace is linear. By definition of g and linearity of the trace we have

$$\mathrm{Tr}(\zeta_m^j \cdot g/m) = \frac{1}{m}\mathrm{Tr}(\zeta_m^j \cdot (1-\zeta_p)) = \frac{1}{m}\mathrm{Tr}(\zeta_m^j - \zeta_m^j \cdot \zeta_m^{m/p}) = \frac{1}{m}\left(\mathrm{Tr}(\zeta_m^j) - \mathrm{Tr}(\zeta_m^{j+m'})\right).$$

By the previous Lemma, the above expression is zero for all $j \neq 0$. In the case $j = 0$ we have $j = 0 \mod m$ and $j + m' = m' = 0 \mod m'$ but $m' \neq 0 \mod m$. Thus, the traces evaluate to

$$\frac{1}{m}\left(\mathrm{Tr}(\zeta_m^j) - \mathrm{Tr}(\zeta_m^{j+m'})\right) = \frac{1}{m}(\varphi(p)\cdot m' + m') = \frac{(\varphi(p)+1)m'}{m} = \frac{pm'}{m} = 1.$$

Now, in order to show that $R^\vee = \langle g/m\rangle$, it suffices to prove that $\mathrm{N}(R^\vee) = \mathrm{N}(g/m)$. We already proved that $g/m \in R^\vee$ and thus $\langle g/m\rangle \subset R^\vee$. Since $R^\vee$ is a fractional ideal, there is an element $\alpha \in R$ such that $\alpha R^\vee \subset R$. But, for the same element α it also holds that $\alpha\langle g/m\rangle \subset \alpha R^\vee \subset R$. The norms of these fractional ideals are defined as

$$\mathrm{N}(R^\vee) = \frac{\mathrm{N}(\alpha R^\vee)}{|\mathrm{N}(\alpha)|} \quad \text{and} \quad \mathrm{N}(\langle g/m\rangle) = \frac{\mathrm{N}(\alpha\langle g/m\rangle)}{|\mathrm{N}(\alpha)|}.$$

If $\mathrm{N}(R^\vee) = \mathrm{N}(g/m) = \mathrm{N}(\langle g/m\rangle)$ it follows that $\mathrm{N}(\alpha R^\vee) = \mathrm{N}(\alpha\langle g/m\rangle)$. This means that $[R/\alpha R^\vee] = [R/\alpha\langle g/m\rangle]$ and from group theory we know that

$$[R/\alpha\langle g/m\rangle] = [R/\alpha R^\vee]\cdot[\alpha R^\vee/\alpha\langle g/m\rangle],$$

which then leaves no choice other than $[\alpha R^\vee/\alpha\langle g/m\rangle] = 1$. This in turn implies $R^\vee = \langle g/m\rangle$.

From Equation (1.6) we know that $\mathrm{N}(R^\vee) = \Delta_K^{-1}$. The latter is due to Equation (1.4) given by

$$\Delta_K^{-1} = \left(\frac{p^{1/(p-1)}}{m}\right)^{\varphi(m)} = \frac{p^{\varphi(m)/\varphi(p)}}{m^{\varphi(m)}}.$$

Note that $\varphi(m)/\varphi(p) = m' = m/p$, so we have $\mathrm{N}(R^\vee) = p^{m/p}/m^{\varphi(m)}$. Concerning the element g/m we have $\mathrm{N}(m) = m^{\varphi(m)}$ and $\mathrm{N}(g) = \mathrm{N}(1-\zeta_p) = \mathrm{N}_{\mathbb{Q}(\zeta_p)/\mathbb{Q}}(1-\zeta_p)^{[K:\mathbb{Q}(\zeta_p)]}$. With a similar argument as in the proof of Lemma 1.4.19, we see that $[K:\mathbb{Q}(\zeta_p)] = m' = m/p$. Let $\{\tau_i\}_{i\in\mathbb{Z}_p^*}$ be the $(p-1)$ distinct embedding of $\mathbb{Q}(\zeta_p)$. Then we have

$$\mathrm{N}_{\mathbb{Q}(\zeta_p)/\mathbb{Q}}(1-\zeta_p) = \prod_{i\in\mathbb{Z}_p^*}\tau_i(1-\zeta_p) = \prod_{i\in\mathbb{Z}_p^*} 1-\tau_i(\zeta_p).$$

The terms $\tau_i(\zeta_p)$ just traverse through the complex primitive p-th roots of unity, so that the latter expression is just $\Phi_p(1)$. In fact, the p-th cyclotomic polynomial at one evaluates to p. Consequently, we have $\mathrm{N}(g) = p^{m/p}$, thus $\mathrm{N}(g/m) = p^{m/p}/m^{\varphi(m)} = \mathrm{N}(R^\vee)$.

Ad(ii): A nice fact about cyclotomic fields for primes p is that the equation

$$1 + \zeta_p + \zeta_p^2 + \ldots + \zeta_p^{p-1} = 0$$

holds. Using this we can verify

$$p = (1-\zeta_p)\left((p-1) + (p-2)\zeta_p + \ldots + \zeta_p^{p-2}\right),$$

implying that $p/g \in R$.

Ad(iii): Note that the norm of g is a power of p. Hence, for any prime integer $p' \neq p$ we have $\langle g \rangle \subset \langle g \rangle + \langle p' \rangle$, $\langle p' \rangle \subset \langle g \rangle + \langle p' \rangle$ and the norm $\mathrm{N}(\langle g \rangle + \langle p' \rangle)$ divides both p and p'. Thus, $\mathrm{N}(\langle g \rangle + \langle p' \rangle) = 1$ is the only possibility, implying $\langle g \rangle + \langle p' \rangle = R$.

□

Definition 1.4.21. Let m be an arbitrary positive integer. For $R = \mathbb{Z}[\zeta_m]$, we define $g := \prod_p (1 - \zeta_p) \in R$, where p runs over all *odd* primes dividing m. Further, define $t := \hat{m}/g \in R$, where $\hat{m} := m/2$ if m is even, $\hat{m} = m$ otherwise.

With this definition and the above lemma we can describe $R^\vee$ also for arbitrary m very precisely.

Corollary 1.4.22. *With the above definition and notation we have* $R^\vee = \langle g/\hat{m} \rangle = \langle t^{-1} \rangle$. *Furthermore,* $\langle g \rangle$ *is coprime with* $\langle p' \rangle$ *for each prime integer* p'*, except those* odd *primes dividing* m*.*

Proof. Let $m = \prod_{l=0}^{s} m_l$ be the prime power factorization of m, where m_l is a power of some prime p_l. Consider R via the ring isomorphism $R = \bigotimes_{l=0}^{s} R_l$, where $R_l = \mathbb{Z}[\zeta_{m_l}]$. Then we can express g as $g = (\hat{m}/m)(\bigotimes_{l=0}^{s} g_l)$, where $g_l = (1 - \zeta_{p_l})$. Note that the factor $\hat{m}/m$ is either 1 or $1/2$ depending on m being odd or not. In the case where m is even it thus eliminates the extra term $(1 - \zeta_2) = 2$. Now, together with Lemma 1.4.20 we have

$$\left(\bigotimes_{l=0}^{s} R_l \right)^\vee = \bigotimes_{l=0}^{s} R_l^\vee = \bigotimes_{l=0}^{s} (g_l/m_l) R_l = (g/\hat{m}) \cdot \left(\bigotimes_{l=0}^{s} R_l \right),$$

which proves the first claim.

To prove the second part, note that p' is coprime to all *odd* primes dividing m and the norm of g is a product of powers of these primes. Thus, the proof is analogously to the proof of Lemma 1.4.20 (iii). □

1.4.5 Prime Splitting and Chinese Remainder Theorem

A central topic in algebraic number theory is the splitting of primes in an algebraic number field K and its ring of integers R. That means, if we have a prime p in $\mathbb{Z}$, how does the prime ideal factorization of $\langle p \rangle \subset R$ looks like. The following theorem, which we leave without a proof, gives us some insight about that.

Theorem 1.4.23. *Let* m *be any positive integer and write* $m = \prod_p p^{k_p}$ *for primes* p*, where* $k_p \geq 0$*. Further let* $m'_p := m/p^{k_p}$ *and let* f_p *be the smallest integer such that*

$$p^{f_p} = 1 \mod m'_p,$$

i.e., f_p *is the multiplicative order of* p *in* $\mathbb{Z}^*_{m'_p}$*. Then* p *splits into*

$$\langle p \rangle = (\mathfrak{p}_1 \cdot \ldots \cdot \mathfrak{p}_r)^{\varphi(p^{k_p})},$$

where $\mathfrak{p}_1, \ldots, \mathfrak{p}_r$ *are distinct prime ideals in* R *each of norm* p^{f_p} *and* $r = \varphi(m'_p)/f_p$*.*

For later purposes we are interested in the special case, where $K = \mathbb{Q}(\zeta_m)$ and q is a prime integer such that $q = 1 \mod m$. Then $k_q = 0, f_q = 1$ and $m'_q = m$. Hence, $r = \varphi(m) = n$ and q splits into n distinct prime ideals $\mathfrak{q}_i$ all of norm q. Furthermore, since

$$q = 1 \mod m \Leftrightarrow (q-1) = 0 \mod m,$$

m divides $q - 1 = \varphi(q)$ which is the order of $\mathbb{Z}_q^*$. Now, from group theory we know that there are $\varphi(m) = n$ elements of order m in $\mathbb{Z}_q^*$, i.e., n distinct primitive roots of unity ω_m. These roots of unity are permuted by the action of the group $\mathbb{Z}_m^*$. So, if we fix one root ω_m, the remaining $n-1$ roots are given by ω_m^i for $i \in \mathbb{Z}_m^*$. Then algebraic number theory tells us that the prime ideal factors of $\langle q \rangle$ are given by $\mathfrak{q}_i = \langle q \rangle + \langle \zeta_m - \omega_m^i \rangle$. This again implies that the quotient ring $R/\mathfrak{q}_i$ is isomorphic to $\mathbb{Z}_q$ as a field via the map $\zeta_m \mapsto \omega_m^i$.

The Chinese remainder theorem states that for k distinct, pairwise coprime ideals $\{\mathfrak{p}_i\}_{i\in[k]}$ in R, the natural ring homomorphism from $R/\prod_{i\in[k]} \mathfrak{p}_i$ to $\prod_{i\in[k]} (R/\mathfrak{p}_i)$ is in fact an isomorphism. In order to define a special $\mathbb{Z}_q$-basis for the quotient $R_q = R/qR$ we will need the following specialization.

Lemma 1.4.24. *Let $q = 1 \mod m$ be a prime number. Further, consider $\omega_m \in \mathbb{Z}_q$ and ideals $\mathfrak{q}_i$ as above. Then the natural ring homomorphism $R/\langle q \rangle \to \prod_{i\in\mathbb{Z}_m^*} (R/\mathfrak{q}_i) \cong \mathbb{Z}_q^n$ is an isomorphism.*

1.5 Ring-LWE

We close the preliminaries with a short introduction to the ring variant of the "Learning with Errors" problem (ring-LWE). The LWE problem was first introduced by Regev [Reg09] in 2005 and works on $\mathbb{Z}_p^n$ for some integers p and n. Suppose we have a list of equations

$$\begin{aligned} \langle s, a_1 \rangle + e_1 &= b_1 \mod p \\ \langle s, a_2 \rangle + e_2 &= b_2 \mod p \\ \langle s, a_3 \rangle + e_3 &= b_3 \mod p \\ &\vdots \end{aligned}$$

where $s \in \mathbb{Z}_p^n$ is a secret element, the a_i are chosen independently and uniformly from $\mathbb{Z}_p^n$ and $b_i \in \mathbb{Z}_p$. The small errors e_i are sampled from a specific probability distribution χ over $\mathbb{Z}_p$. Now we denote the problem of finding s given a list of elements b_i of arbitrary length by $\mathrm{LWE}_{p,\chi}$. Next, we explain how this problem can be translated to rings. Thereby, we follow the presentation of [LPR13a] and [LPR13b].

Definition 1.5.1 (Ring-LWE Distribution)**.** Let $K = \mathbb{Q}(\zeta_m)$ be the m-th cyclotomic number field for some positive integer m. Further let $R \subset K$ be the ring of integers in K and $R^\vee$ its dual ideal. Let ψ be a probability distribution over $K_\mathbb{R}$ and $s \in R_q^\vee = R^\vee/qR^\vee$ (or just $R^\vee$) a "secret", where q is a prime such that $q = 1 \mod m$. We often refer to ψ as a *ring-LWE error distribution*. A sample from the *ring-LWE distribution* $A_{s,\psi}$ over $R_q \times (K_\mathbb{R}/qR^\vee)$ is generated by choosing $a \leftarrow R_q$ uniformly at random, $e \leftarrow \psi$ from the distribution ψ and defining the sample by

$$(a, b = a \cdot s + e \mod qR^\vee).$$

Remark 1.5.2. Usually the ring-LWE error distribution ψ will be some continuous Gaussian distribution N_r for a standard deviation r. We can view a single dimensional Gaussian distribution over $K_{\mathbb{R}}$ as an n-dimensional Gaussian distribution over $H \cong \mathbb{R}^{[n]}$, where each entry is a single dimensional Gaussian distribution with standard deviation r. We call a multivariate Gaussian distribution *spherical*, if each dimension has the same standard deviation. Throughout the toolkit and our applications we will always regard only spherical Gaussians.

Now we can define the search variant of LWE over the ring R.

Definition 1.5.3 (Search Variant of Ring-LWE)**.** Let $K, R, R^\vee$ and q be as in Definition 1.5.1. For a ring-LWE error distribution ψ over $K_{\mathbb{R}}$ and a secret $s \in R_q^\vee$ we define the *ring-LWE* problem in R as follows. Assume we have access to arbitrarily many independent samples from the ring-LWE distribution $A_{s,\psi}$, the goal is to find s. We denote this problem by $R\text{-LWE}_{q,\psi}$.

We can also define a decision variant of ring-LWE which is provably equivalent to $R\text{-LWE}_{q,\psi}$.

Definition 1.5.4 (Decision Variant of Ring-LWE)**.** Let $K, R, R^\vee$ and q be as in Definition 1.5.1. For a ring-LWE error distribution ψ over $K_{\mathbb{R}}$ and a uniformly random secret $s \in R_q^\vee$ we define the *decision variant* of the ring-LWE problem in R as follows. Let $l \geq 1$ be an arbitrary integer. Given l independent samples from the ring-LWE distribution $A_{s,\psi}$ and l uniformly random and independent samples from $R_q \times (K_{\mathbb{R}} \times qR^\vee)$, the goal is to distinguish with non-negligible advantage between these samples. We denote this problem by $R\text{-DLWE}_{q,\psi}$

Another classical problem is the shortest vector problem (SVP), which asks for the shortest vector in some given lattice $\mathcal{L}(B)$ for some known basis B. It can be shown that the shortest vector problem is NP-hard, see e.g. [MG02]. Now, Lyubashevsky, Peikert and Regev show in [LPR13a] that there is a reduction from SVP to ring-LWE. We cite only an informal statement. For details see [LPR13a].

Theorem 1.5.5 (Informal)**.** *Assume it is hard for polynomial-time* quantum *algorithms to approximate the shortest vector problem* (SVP) *on ideal lattices in* $K = \mathbb{Q}(\zeta_m)$ *to within a fixed* $\text{poly}(m)$ *factor. Then any* $\text{poly}(m)$ *number of samples drawn from the ring-LWE distribution are pseudorandom (i.e., indistinguishable from uniformly random samples) for any polynomial-time (even quantum) attacker.*

In our applications in Section 2.1 we use a different variant of ring-LWE. Instead of an error distribution ψ over $K_{\mathbb{R}}$ we use a discretized version, i.e., a distribution χ over $R^\vee$. For such a distribution we define the ring-LWE distribution $A_{s,\chi}$ as in Definition 1.5.1, where this time the element b is in $R_q^\vee$ instead of $(K_{\mathbb{R}}/qR^\vee)$. Therefore, the distribution $A_{s,\chi}$ is over $R_q \times R_q^\vee$. We can define the problems $R\text{-}LWE_{q,\chi}$ and $R\text{-}DLWE_{s,\chi}$ similar as in Definitions 1.5.3 and 1.5.4. Then, we use independent samples from $A_{s,\chi}$ and independent and uniform samples from $R_q \times R_q^\vee$.

Similarly, we can ask the secret s to be sampled from a discrete distribution χ over $R_q^\vee$ instead of being uniformly at random. In this way, we have more control over s, since we can choose distributions which produce elements with specific properties. For example, in our applications we use a discretization of the continuous error distribution ψ, which produces small elements. To be more precise, let p be coprime to q and $\lfloor \cdot \rceil$ be a valid discretization to (cosets of) $pR^\vee$ (cf. Section 1.3.2). Then we set $\chi = \lfloor p \cdot \psi \rceil_{w+pR^\vee}$, where w is an arbitrary

element, which can vary from sample to sample. In the distribution χ, we first choose a sample from ψ, scale it by p, and finally discretize it to $w + pR^\vee$.

The following lemmas show that, if we set $\chi = \lfloor p \cdot \psi \rceil_{w+pR^\vee}$ as above, the ring-LWE problem together with the above modifications is as hard as the original one.

Lemma 1.5.6 (Lemma 2.23. from [LPR13b])**.** *Let m be an arbitrary positive integer and $K = \mathbb{Q}(\zeta_m)$ the m-th cyclotomic number field. Further let $R \subset K$ be the ring of integers in K and $R^\vee$ its dual ideal. Denote by $\lfloor \cdot \rceil$ a valid discretization to (cosets of) $pR^\vee$, where p is coprime to q and $q = 1 \mod m$ is prime. There exists an efficient transformation that on input $w \in R_p^\vee$ and a pair $(a', b') \in R_q \times (K_\mathbb{R}/qR^\vee)$, outputs a pair $(a, b) \in R_q \times R_q^\vee$, where $a := pa' \mod qR$ and $b := \lfloor pb' \rceil_{w+pR^\vee} \mod qR^\vee$. The transformation fulfills the following guarantees:*

(i) *If the input pair is uniformly distributed, then so is the output pair.*

(ii) *If the input pair is distributed according to the ring-LWE distribution $A_{s,\psi}$, for some (unknown) $s \in R^\vee$ and an error distribution ψ over $K_\mathbb{R}$, then the output pair is distributed according to $A_{s,\chi}$, where $\chi = \lfloor p \cdot \psi \rceil_{w+pR^\vee}$.*

Proof. For a proof see [LPR13b]. □

Lemma 1.5.7 (Lemma 2.24. from [LPR13b])**.** *We use the notation from Lemma 1.5.6. Let $w \in R_p^\vee$ be an arbitrary element and ψ a ring-LWE error distribution over $K_\mathbb{R}$. If R-DLWE$_{q,\psi}$ is hard given some number l of samples, then so is the variant of R-DLWE$_{q,\psi}$ in which the secret s is sampled from $\chi := \lfloor p \cdot \psi \rceil_{w+pR^\vee}$, given $l - 1$ samples.*

Proof. For a proof see [LPR13b]. □

2 The Toolkit

The goal of this chapter is to develop a toolkit for ring-LWE based cryptography that provides efficient algorithms for basic operations and cryptographic tasks. In a general ring-LWE setting we are working with elements in a cyclotomic number field $K = \mathbb{Q}(\zeta_m)$ of dimension $n = \varphi(m)$ for some positive integer m. In particular, these elements live in certain ideals of K, most prominent the ring of integers $R = \mathcal{O}_K$ and its dual ideal $R^\vee$. All considered ideals are represented by $\mathbb{Z}$-bases, so we represent an element a by its coordinate vector $\mathbf{a} \in \mathbb{Z}^{[n]}$ with respect to a specific basis. Consequently, our algorithms work only on the coordinate vector while keeping track of the basis.

The developed toolkit is originally due to [LPR13b]. We follow closely the explanations of [LPR13b], but at the same time use a different structure that the original one. One might say that we use a "top-down" presentation, whereas [LPR13b] uses more or less the opposite "bottom-up" approach. We feel like our presentation helps the understanding, especially looking to our implementation of the toolkit later on.

The chapter starts in Section 2.1 with a motivational section, where we present two applications of the toolkit. Following these applications, we summarize the necessary operations, which are then developed through the rest of this chapter. The Sections 2.2 and 2.3 deal with some basic arithmetic and the representation of elements in R and $R^\vee$ in different bases. We define several specific bases, which have some nice properties to simplify the necessary operations. We develop efficient methods to change the representation between these bases and show how multiplication and addition can be performed. Then, in Section 2.3.6 and Section 2.4 we adapt the algorithmic tasks of decoding and discretization to our ring-LWE setting. Finally, the Sections 2.3.5 and 2.2.5 give some insight on our methods of sampling Gaussians in $R^\vee$ and R. As an overview, Figure 2.1 summarizes and briefly explains the central algebraic objects and notations used in the toolkit.

For the rest of this chapter, if not stated differently, let $K = \mathbb{Q}(\zeta_m)$ be the m-th cyclotomic number field for some positive integer m and $R \subset K$ its ring of integers. $R^\vee$ is the dual ideal of R as defined in Section 1.4.4. The dimension of K is $n = \varphi(m)$. Further, let $H \subset \mathbb{C}^{\mathbb{Z}_m^*}$ be the subspace of $\mathbb{C}^{\mathbb{Z}_m^*}$ as defined in Definition 1.4.7. Finally, let q be a prime such that $q = 1 \mod m$.

2.1 Applications of the Toolkit

We start this chapter with a motivational section. We will describe two ring-LWE based cryptosystems that use the toolkit. The purpose of this section is to give the reader an idea and some motivation for the different operations and tasks we need and which will be later on efficiently handled by the toolkit. Therefore, we recommend to read this section twice. Once at the beginning of the study of this work and once afterwards. In the second run, the way the toolkit and our implementation works for these applications and how things work together will become much clearer.

Notation	Description	See
$m, n = \varphi(m), \hat{m}$	A positive integer m with prime power factorization $m = \prod_{l=0}^{s} m_l$. Consequently, $n = \prod_{l=0}^{s} \varphi(m_l)$. Furthermore, $\hat{m} = m/2$ if m is *even*, otherwise $\hat{m} = m$	
ζ_m, ω_m	A *primitive* m-th root of unity viewed as an abstract element ζ_m in some algebraic closure of $\mathbb{Q}$. ω_m is a primitive m-th root in $\mathbb{C}$ or $\mathbb{Z}_q$ for a prime $q = 1 \mod m$.	§ 1.4.1
$K = \mathbb{Q}(\zeta_m) \cong \mathbb{Q}[X]/(\Phi_m(X)) \cong \bigotimes_{l=0}^{s} \mathbb{Q}(\zeta_{m_l})$	The m-th *cyclotomic number field* K. Elements in K are often viewed as polynomials in the quotient $\mathbb{Q}[X]/(\Phi_m(X))$, where $\Phi_m(X)$ is the m-th cyclotomic polynomial. In this work, we view K as the tensor product of the prime power indexed $\mathbb{Q}(\zeta_{m_l})$.	§ 1.4.1
$\sigma : K \to \mathbb{C}^n$	The *canonical embedding* of K to $\mathbb{C}^n$. It endows K with a canonical geometry, e.g., $\|a\|_2 = \|\sigma(a)\|_2$ for $a \in K$. Further, addition and multiplication in K correspond via σ to their component-wise counterparts in $\mathbb{C}^n$.	§ 1.4.2
$R = \mathbb{Z}[\zeta_m] \cong \mathbb{Z}[X]/(\Phi_m(X)) \cong \bigotimes_{l=0}^{s} \mathbb{Z}[\zeta_{m_l}]$	The *ring of integers* in K (often denoted as $\mathcal{O}_K$). Similar to K we view R as the tensor product of the subrings $R_l = \mathbb{Z}[\zeta_{m_l}]$.	§ 1.4.3
$R^\vee = \left\langle t^{-1} \right\rangle$, $g, t \in R$	The *dual* fractional ideal of R. It is generated by the element $t^{-1} = g/\hat{m}$. In particular, we have $R \subseteq R^\vee$. Each of $R^\vee, g$, and t can be seen as the tensor product of their counterparts in $\mathbb{Q}(\zeta_{m_l})$.	§ 1.4.4
$\vec{p} \subset R$, $t^{-1}\ \vec{p}\ \subset\ R^\vee$	The "*powerful*" $\mathbb{Z}$-basis $\vec{p}$ of R and $t^{-1}\vec{p}$ of $R^\vee$. For prime powers m, $\vec{p}$ coincides with the power basis of $\mathbb{Z}[\zeta_m]$. For non prime powers m it is the tensor product of the power(ful) bases of each $\mathbb{Z}[\zeta_{m_l}]$ and differs from the usual power basis.	§ 2.2.1, § 2.3.1
$\vec{c} \subset R_q$, $t^{-1}\ \vec{c}\ \subset\ R_q^\vee$	The "*Chinese remainder*" $\mathbb{Z}_q$-basis $\vec{c}$ of $R_q = R/qR$ and $t^{-1}\vec{c}$ of $R_q^\vee = R^\vee/qR^\vee$ for a prime $q = 1 \mod m$. Yields linear time algorithms for addition and multiplication in R_q and can be efficiently swapped with the powerful $\mathbb{Z}_q$-basis $\vec{p}$ of R_q.	§ 2.2.1, § 2.3.1
$\vec{d} \subset R^\vee$	The "*decoding*" $\mathbb{Z}$-basis of $R^\vee$. It is the dual basis of the conjugate of the powerful basis $\vec{p}$. Coincides with the tensor product of the decoding bases of $R_{m_l}^\vee$ Can be efficiently swapped with $t^{-1}\vec{p}$.	§ 2.3.1

Figure 2.1: Central Algebraic Objects and Notations

Although we will later use the toolkit to realize these cryptosystems, which implies the use of specific bases and algorithms to perform most of the operations, the cryptosystems are almost entirely described in a implementation- and basis-independent manner. We use abstract mathematical objects, operations and tasks (like ideals in rings, basic ring arithmetic or probability distributions over ideals) which could be represented and realized also through different approaches.

We state the cryptosystems without any correctness or hardness proofs. Such proofs can be found in [LPR13b] as well as a third application and further details.

2.1.1 Dual-Style Cryptosystem

The first cryptosystem is a ring-based variant of the commonly called "dual" LWE encryption from [GPV08].

Let $K = \mathbb{Q}(\zeta_m)$ be the m-th cyclotomic number field for some positive integer m and $R \subset K$ be the ring of integers of K. Let p and q be coprime integers, where p defines the message space $R_p = R/pR$ and q is a prime such that $q = 1 \mod m$. We call q the ring-LWE modulus. Let ψ be a continuous LWE error distribution over $K_{\mathbb{R}}$ and $\lfloor \cdot \rceil$ a valid discretization to cosets of $R^\vee$ or $pR^\vee$. Then, an expression like $\lfloor p \cdot \psi \rceil_{pR^\vee}$ means that we sample an element from ψ, scale it by p and discretize it to $pR^\vee$. Further, denote by $D_{R,r}$ the discrete Gaussian distribution over R for some standard deviation $r \geq \sqrt{n} \cdot \omega(\sqrt{\log n})$, where $n = \varphi(m)$. For some parameter $l \geq 2$ we define the cryptosystem as follows.

- Gen: Let $a_0 = -1 \in R_q = R/qR$ and choose uniformly random and independent elements $a_1, \ldots, a_{l-1} \in R_q$. Further, let $x_0, \ldots, x_{l-1} \leftarrow D_{R,r}$ be independent samples from the discrete Gaussian distribution $D_{R,r}$ over R. Now define $a_l := -\sum_{i \in [l]} a_i x_i$ and $x_l := 1$ and output
$$\mathbf{a} = (a_1, \ldots, a_l) \in R_q^{\{1,\ldots,l\}}$$
as the public key and
$$\mathbf{x} = (x_1, \ldots, x_l) \in R^{\{1,\ldots,l\}}$$
as the secret key. By construction we have $\langle \mathbf{a}, \mathbf{x} \rangle = x_0 \in R_q$.

- $\mathrm{Enc}_{\mathbf{a}}(\mu \in R_p)$: Choose independent samples $e_0, e_1, \ldots, e_{l-1} \leftarrow \lfloor p \cdot \psi \rceil_{pR^\vee}$ and hide the message in $e_l \leftarrow \lfloor p \cdot \psi \rceil_{t^{-1}\mu + pR^\vee}$. Let
$$\mathbf{e} = (e_1, \ldots, e_l) \in (R^\vee)^{\{1,\ldots,l\}},$$
where the e_i are the representatives of cosets $e_i + pR^\vee$ and can thus be viewed in $R^\vee$. Now define the ciphertext
$$\mathbf{c} := e_0 \cdot \mathbf{a} + \mathbf{e} \in (R_q^\vee)^{\{1,\ldots,l\}}.$$

- $\mathrm{Dec}_{\mathbf{x}}(\mathbf{c})$: Decode $\langle \mathbf{c}, \mathbf{x} \rangle$ to $d = [\![\langle \mathbf{c}, \mathbf{x} \rangle]\!] \in R^\vee$ and retrieve the message
$$\mu = t \cdot d \mod pR \in R_p.$$
Recall that $R^\vee = \langle t^{-1} \rangle$, so $t \cdot d \in R$ and the reduction modulo pR is indeed in R_p.

2.1.2 Compact Public-Key Cryptosystem

The second cryptosystem is again a public-key scheme, but with generally more compact keys and ciphertexts. To be more precise, the keys and ciphertexts consist only of one or two ring element, whereas in the first cryptosystem the parameter l, which controls the size of the keys and ciphertexts, is at least 2.

As in the previous section let $K = \mathbb{Q}(\zeta_m)$ be the m-th cyclotomic number field for some positive integer m and $R \subset K$ be the ring of integers of K. Let p and q be coprime integers, where p defines the message space $R_p = R/pR$ and q is a prime such that $q = 1 \mod m$. Let ψ be a continuous LWE error distribution over $K_{\mathbb{R}}$ and $\lfloor \cdot \rceil$ a valid discretization to cosets of $R^\vee$ or $pR^\vee$. We define the cryptosystem as follows.

- Gen : First, choose a uniformly random $a \in R_q$. Further, choose samples $x \leftarrow \lfloor \psi \rceil_{R^\vee}$ and $e \leftarrow \lfloor p \cdot \psi \rceil_{pR^\vee}$. Define $b := \hat{m}(a \cdot x + e) \mod qR \in R_q$ and output

  $$(a, b) \in R_q \times R_q$$

  as the public key. The secret key is $x \in R^\vee$. [Recall that $\hat{m} = t \cdot g$ and $R^\vee = \langle t^{-1} \rangle$. We have $a \cdot x + e \in R^\vee/qR^\vee$. Multiplication with $\hat{m}$ yields $\hat{m}(a \cdot x + e) \in gR/gqR$. Now, reducing $\hat{m}(a \cdot x + e) \bmod qR$ leads to $\hat{m}(a \cdot x + e) \mod qR \in gR/gqR \Big/ gR/qR \cong gqR/qR$, which can then be viewed in R_q.]

- $\mathrm{Enc}_{(a,b)}(\mu \in R_p)$: Choose samples $z \leftarrow \lfloor \psi \rceil_{R^\vee}, e' \leftarrow \lfloor p \cdot \psi \rceil_{pR^\vee}$ and $e'' \leftarrow \lfloor p \cdot \psi \rceil_{t^{-1}\mu + pR^\vee}$. Define $u := \hat{m}(z \cdot a + e') \mod qR \in R_q$ and $v := z \cdot b + e'' \in R_q^\vee$. Output

  $$(u, v) \in R_q \times R_q^\vee$$

  as the ciphertext.

- $\mathrm{Dec}_x(u, v)$: Compute $v - u \cdot x = \hat{m}(e \cdot z - e' \cdot x) + e'' \mod qR^\vee$ and decode it to $d = [\![v - u \cdot x]\!] \in R^\vee$. Retrieve the message

  $$\mu = t \cdot d \mod pR.$$

  Recall that $R^\vee = \langle t^{-1} \rangle$, so $t \cdot d \in R$ and the reduction modulo pR is indeed in R_p.

2.1.3 Summarization of the Necessary Operations

Now that we saw the two applications, we will summarize which features and operations our toolkit has to provide to realize these applications. To begin with, all functions and operations take place in the ring of integers R of the cyclotomic number field $K = \mathbb{Q}(\zeta_m)$, in its dual ideal $R^\vee \subset K$ or in the quotients R_q and $R_q^\vee$. Therefore, we need a way to represent elements in R and $R^\vee$ which is suitable for a computer. We already saw in Section 1.4 that there exist $\mathbb{Z}$-bases of size $n = \varphi(m)$ for R and $R^\vee$ (in fact for every fractional ideal in K). We use these bases to represent an element in R or $R^\vee$ by an integral coordinate vector of size n with respect to a specific basis. In Section 2.2.1 and Section 2.3.1 we will define several specific bases which have some nice properties, in particular concerning the functions and algorithms we provide.

As we are dealing with element in K, which is in particular a field, we should be able to perform basic arithmetic tasks, namely addition and multiplication. In particular, we are

interested in basic arithmetic in R and $R^\vee$. In Sections 2.2.3 and 2.3.3 we will develop efficient methods for addition and multiplication in R and $R^\vee$, which could also be used more generally in K.

In both applications we need to sample elements from different probability distributions over different domains. While a uniformly random sample can be achieved fairly easy, the computation of a discrete Gaussian or a discretized Gaussian needs some non-trivial algorithms. In Section 2.2.5 we describe an algorithm that samples directly from the discrete Gaussian distribution $D_{R,r}$. An efficient way of sampling Gaussians in $K_\mathbb{R}$ is developed in Section 2.3.5, which can be used or the error distribution ψ. These Gaussian can then be discretized using the algorithm from Section 2.4.

Finally, we need an algorithm that decodes certain elements in $R^\vee$. First, we introduce the decoding problem for our ring-LWE setting in Section 2.3.6 and then develop an efficient algorithm in Section 3.4.

2.2 Working in the Ideal R

Let m be an arbitrary positive integer and $K = \mathbb{Q}(\zeta_m)$ be the m-th cyclotomic number field. The ring of integers $R \subset K$ is also an integral ideal in K. In our toolkit applications and, more generally, in ring-LWE based cryptography a lot of the action takes place in the ideal R or the quotient $R_q = R/qR$ for some prime $q = 1 \mod m$. In the first place, the arithmetic tasks of addition and multiplication are performed. The complexity of these operations depends heavily on the basis, which is used for the representation of elements in R and R_q. In the following we will define two specific bases, a $\mathbb{Z}$-basis for R and a $\mathbb{Z}_q$-basis for R_q, which provide efficient algorithms not only for the arithmetic tasks, but also for discretization and sampling of Gaussians over R. Furthermore, we can switch the representation between these two bases efficiently, which is an important feature for many of our algorithms.

As it will turn out, the essence of many of our algorithms is the Chinese remainder transform CRT. It is a specific transformation matrix, which will be used very often in our algorithms, especially in the basis switching and arithmetic operations. Therefore, it is useful to optimize the efficiency of an application of CRT. At the end of this section we provide a sparse decomposition of CRT, which can be used for exactly this purpose.

2.2.1 Two Specific Bases of R

The Powerful Basis of R

First, we start with a $\mathbb{Z}$-basis of R, hence a $\mathbb{Q}$-basis of K, called the "powerful" basis.

Definition 2.2.1. For $K = \mathbb{Q}(\zeta_m)$ and $R = \mathbb{Z}[\zeta_m]$ we define the *powerful basis* $\vec{p}$ of R as follows:

- For prime powers m, we define $\vec{p}$ to be the usual power basis $(\zeta_m^j)_{j \in [\varphi(m)]}$, seen as a vector over R.
- For arbitrary $m \in \mathbb{N}$ with prime power factorization $m = \prod_{l=0}^{s} m_l$, we define $\vec{p} = \bigotimes_{l=0}^{s} \vec{p}_l$ as the Kronecker (tensor) product of the power(ful) bases $\vec{p}_l$ of each $K_l = \mathbb{Q}(\zeta_{m_l})$.

Remark 2.2.2. Recall from the definition of the Kronecker product (see. Definition 1.2.1) that the index set of $\vec{p}$ is given by the product $\prod_{l=0}^{s} [\varphi(m_l)]$. Thus, if we want to specify an

entry of $\vec{p}$, we need one index $j_l \in [\varphi(m_l)]$ per prime power m_l. The desired entry is then given by $p_{(j_l)} = \prod_{l=0}^{s} \zeta_{m_l}^{j_l}$. If we use the fact that $\zeta_{m_l} = \zeta_m^{m/m_l} \in K$, it is possible to convert this index set into a subset of $[m]$ of size $\varphi(m)$. To do this, we map the index tuple (j_l) to $j = \sum_{l=0}^{s} (m/m_l) j_l \mod m$ and access the entry via $p_j = \zeta_m^j$. This index set will in general *not* be equal to $[\varphi(m)]$, which implies that the powerful basis differs from the power bases. The only exception is the case, where m is a prime power. As an example, take $m = 15$ and set $\zeta = \zeta_{15}$. Then the powerful basis is given by $\zeta^0, \zeta^3, \zeta^5, \zeta^6, \zeta^8, \zeta^9, \zeta^{11}$ and ζ^{14}. Usually, we will stick to the first, structured index set, because the other one tends to be somewhat irregular.

The Chinese Remainder Basis of R

The second special basis we define is the Chinese remainder basis. It is a $\mathbb{Z}_q$-basis for R_q, where q is a prime integer such that $q = 1 \mod m$. This basis yields very fast algorithms for multiplication using only arithmetic in $\mathbb{Z}_q$.

Let $m \in \mathbb{N}$ be arbitrary with prime power factorization $m = \prod_{l=0}^{s} m_l$ and $q = 1 \mod m$ a prime. Then $q = 1 \mod m_l$ also holds for each m_l. Furthermore, we have that

$$R_q = R/qR = \bigotimes_{l=0}^{s} R_l / \bigotimes_{l=0}^{s} qR_l = \bigotimes_{l=0}^{s} R_l/qR_l,$$

so that we can focus on the case where m is a prime power.

Recall from Section 1.4.5 that $\langle q \rangle$ factorizes in R into $\langle q \rangle = \prod_{i \in \mathbb{Z}_m^*} \mathfrak{q}_i$, where $\mathfrak{q}_i = \langle q \rangle + \left\langle \zeta_m - \omega_m^i \right\rangle$ is prime in R and ω_m is some fixed primitive m-th root of unity in $\mathbb{Z}_q$.

Definition 2.2.3. For a positive integer m let $K = \mathbb{Q}(\zeta_m)$, $R = \mathbb{Z}[\zeta_m]$ and $q = 1 \mod m$ be a prime integer. We define the *Chinese remainder* (CRT) $\mathbb{Z}_q$-basis $\vec{c}$ of $R_q = R/qR$ as follows:

- If m is a prime power, then $\vec{c} = (c_i)_{i \in \mathbb{Z}_m^*}$ is characterized by $c_i = 1 \mod \mathfrak{q}_i$ and $c_i = 0 \mod \mathfrak{q}_j$ for $i \neq j$, where $\mathfrak{q}_i$ are the prime ideal factors of $\langle q \rangle$ in R. (Its existence is guaranteed by Lemma 1.4.24.)
- For arbitrary $m \in \mathbb{N}$ with prime power factorization $m = \prod_{l=0}^{s} m_l$, $\vec{c}$ is defined as the tensor product of the CRT bases $\vec{c}_l$ of each R_l/qR_l, i.e., $\vec{c} := \bigotimes_{l=0}^{s} \vec{c}_l$.

2.2.2 Switching between the Powerful and the CRT Basis

Recall that every $\mathbb{Z}$-basis of R is also a $\mathbb{Z}_q$-basis of R_q. In particular, the powerful basis $\vec{p}$ is a $\mathbb{Z}_q$-basis of R_q. In the following we will show that we can switch between the powerful and the CRT basis using a specific basis transformation matrix. That is, if we have an element $a \in R_q$ represented in the powerful basis we can obtain a representation in the CRT matrix via a linear transformation of the coordinate vector, and vice versa using the inverse transformation.

Definition 2.2.4. Let m be a prime power and q a prime such that $q = 1 \mod m$. Further, let ω_m be a primitive root of unity in $\mathbb{Z}_q^*$. We define the following transformations:

- The *discrete Fourier transform* $\mathrm{DFT}_{m,q}$ over $\mathbb{Z}_q^*$ is the $\mathbb{Z}_m \times \mathbb{Z}_m$ square matrix with entries $\mathrm{DFT}_{m,q}(i,j) = \omega_m^{ij}$.

- The *Chinese remainder transform* $\mathrm{CRT}_{m,q}$ over $\mathbb{Z}_q^*$ is the square submatrix of $\mathrm{DFT}_{m,q}$ with rows restricted to the index set $\mathbb{Z}_m^*$ and columns restricted to the index set $[\varphi(m)]$.

Now let m be an arbitrary positive integer with prime power factorization $\prod_{l=0}^{s} m_l$ and $q = 1 \mod m$ a prime. Then we have $q = 1 \mod m_l$ for each $0 \leq l \leq s$ and we define

$$\mathrm{DFT}_{m,q} = \bigotimes_{l=0}^{s} \mathrm{DFT}_{m_l,q} \quad \text{and} \quad \mathrm{CRT}_{m,q} = \bigotimes_{l=0}^{s} \mathrm{CRT}_{m_l,q}.$$

Remark 2.2.5. For prime powers m, the fact that $q = 1 \mod m$ is very important, since it guarantees the existence of a primitive m-th root of unity in $\mathbb{Z}_q$. Instead of $q = 1 \mod m$ we can equivalently say that $q - 1 = 0 \mod m$. In particular, this implies that m divides $q - 1$. Since q is prime, we have that $q - 1 = \varphi(q)$, which is also the order of the group $\mathbb{Z}_q^*$. Now we know that $\mathbb{Z}_q^*$ contains exactly $\varphi(d)$ elements of order d for every divisor d of $q - 1$. In particular, $\mathbb{Z}_q^*$ contains an element ω_m of order m, i.e., a primitive m-th root of unity. If $m \in \mathbb{N}$ is arbitrary with prime power factorization $m = \prod_{l=0}^{s} m_l$, the same reasoning implies that $\mathbb{Z}_q$ contains a primitive m-th root of unity as well as primitive m_l-th roots of unity for each m_l.

The discrete Fourier transform $\mathrm{DFT}_{m,q}$ gets important when we decompose the Chinese remainder transform $\mathrm{CRT}_{m,q}$ in Section 2.2.4. Right now we are only interested in $\mathrm{CRT}_{m,q}$, but it useful to define $\mathrm{DFT}_{m,q}$ and $\mathrm{CRT}_{m,q}$ at the same time, since they are so closely related.

Proposition 2.2.6. *Let m be an arbitrary positive integer and $K = \mathbb{Q}(\zeta_m)$. Further let $R \subset K$ be the ring of integers of K and $q = 1 \mod m$ a prime. Let $\vec{p}$ be the powerful basis of R_q as defined in Definition 2.2.1 and $\vec{c}$ the* CRT *basis of R_q as defined in Definition 2.2.3. Then $\vec{p}$ and $\vec{c}$ are related by*

$$\vec{p}^{\,T} = \vec{c}^{\,T} \cdot \mathrm{CRT}_{m,q},$$

where $\mathrm{CRT}_{m,q}$ is the Chinese remainder transformation over $\mathbb{Z}_q$ from Definition 2.2.4.

Proof. Assume that m is a prime power. Recall from Section 1.4.5 that $R_q = R/\langle q \rangle \cong \prod_{i \in \mathbb{Z}_m^*} (R/\mathfrak{q}_i)$, where each $R/\mathfrak{q}_i$ is isomorphic to $\mathbb{Z}_q$ via the map $\zeta_m \mapsto \omega_m^i$ for some element ω_m of order m in $\mathbb{Z}_q$. Now, by definition of $\vec{c}$, viewing the elements of $\vec{c}$ in $\prod_{i \in \mathbb{Z}_m^*} (R/\mathfrak{q}_i)$ converts them into

$$c_i = (0, \ldots, 0, \underbrace{1}_{i\text{-th entry}}, 0, \ldots, 0) \in \prod_{i \in \mathbb{Z}_m^*} (R/\mathfrak{q}_i).$$

Moreover, for any $j \in [n]$, $\zeta_m^j \mod \mathfrak{q}_i$ is given by ω_m^{ij}. Combining these two facts yields that

$$\zeta_m^j = \sum_{i \in \mathbb{Z}_m^*} \omega_m^{ij} \cdot c_i$$

for any $j \in [n]$. For arbitrary $m \in \mathbb{N}$, the bases $\vec{p}$ and $\vec{c}$ as well as the matrix $\mathrm{CRT}_{m,q}$ are defined via tensor products of prime power cases. The desired relation then follows from the mixed-product property of the tensor product. □

Remark 2.2.7. Let $a \in R_q$ be given by a coordinate vector $\mathbf{a} \in \mathbb{Z}_q^{[n]}$ in the basis $\vec{p}$. Then by Proposition 2.2.6 we have

$$a = \langle \vec{p}, \mathbf{a} \rangle = \left\langle \vec{c}^{\,T} \cdot \mathrm{CRT}_{m,q}, \mathbf{a} \right\rangle = \left\langle \vec{c}, \mathrm{CRT}_{m,q} \cdot \mathbf{a} \right\rangle. \tag{2.1}$$

Hence, the coordinate vector of a with respect to the basis $\vec{c}$ is given by $\mathrm{CRT}_{m,q} \cdot \mathbf{a} \in \mathbb{Z}_q^{\mathbb{Z}_m^*}$. Similarly, we can switch from the basis $\vec{c}$ to $\vec{p}$ by multiplication with $\mathrm{CRT}_{m,q}^{-1}$.

2.2.3 Addition and Multiplication R

Basic arithmetic in the ideal R is a core feature of the toolkit. Addition is, generally speaking, not an issue. By linearity, addition of two elements in R corresponds to the component-wise addition of the respective coordinate vectors, at least if both elements are represented in the same basis. The same holds for R_q. Now, for R we regard only the powerful basis $\vec{p}$, so all elements will be represented with respect to $\vec{p}$. For R_q we regard also the Chinese remainder basis $\vec{c}$. In the previous section we saw that we can easily switch between representations in $\vec{p}$ and $\vec{c}$. Thus, we can always make sure that also elements in R_q are represented in the same basis.

The multiplication operation is not performed so easily. If we have two elements a, b in R or R_q given by some coordinate vectors $\mathbf{a}$ and $\mathbf{b}$, we do not know per se how the coordinate vector of the product ab can be expressed in terms of $\mathbf{a}$ and $\mathbf{b}$. We will se that the complexity of this expression depends on the basis, in which the elements are represented. First we deal with elements in R in the powerful basis $\vec{p}$.

Multiplication in the Powerful Basis

Recall from Section 1.4.2 that the canonical embedding σ embeds the ideal R to a lattice $\sigma(R)$ in H. Now, we want to take advantage of the fact that σ is a ring homomorphism, where multiplication in H is given by coordinate-wise multiplication. Instead of multiplying two elements $a, b \in R$ directly in R, we first embed them via σ, multiply them in H and finally pull the result back via σ^{-1}. This works because of the homomorphic property of σ, i.e., $\sigma(ab) = \sigma(a) \odot \sigma(b)$. In mathematical terms we multiply via the equation

$$ab = \sigma^{-1}(\sigma(ab)) = \sigma^{-1}(\sigma(a) \odot \sigma(b)). \tag{2.2}$$

Next, we have to ask how the embedding σ is applied to elements in R using the coordinate vector representation. As we will show it is possible to view σ as a vector transformation of the coordinate vector in the powerful basis. First, we define the vector transformation matrix, which is similar to the Chinese remainder transformation $\mathrm{CRT}_{m,q}$ over Z_q^* from Definition 2.2.4.

Definition 2.2.8. Let m be a prime power and $\omega_m \in \mathbb{C}$ be any primitive m-th root of unity in $\mathbb{C}$. We define the following transformations:

- The *discrete Fourier transform* DFT_m over $\mathbb{C}$ is the $\mathbb{Z}_m \times \mathbb{Z}_m$ square matrix with entries $\mathrm{DFT}_m(i,j) = \omega_m^{ij}$.
- The *Chinese remainder transform* CRT_m over $\mathbb{C}$ is the square submatrix of DFT_m with rows restricted to the index set $\mathbb{Z}_m^*$ and columns restricted to the index set $[\varphi(m)]$.

Now let m be an arbitrary positive integer with prime power factorization $\prod_{l=0}^{s} m_l$. Then we define

$$\mathrm{DFT}_m = \bigotimes_{l=0}^{s} \mathrm{DFT}_{m_l} \quad \text{and} \quad \mathrm{CRT}_m = \bigotimes_{l=0}^{s} \mathrm{CRT}_{m_l}.$$

Remark 2.2.9. First, we observe that DFT_m is a symmetric matrix, which implies that $\mathrm{DFT}_m^* = \overline{\mathrm{DFT}_m}$. Furthermore, a straightforward computation shows that the inverse of DFT_m is given by

$$\mathrm{DFT}_m^{-1} = \frac{1}{m} \mathrm{DFT}_m^* .$$

The discrete Fourier transform DFT_m gets important when we decompose the Chinese remainder transform CRT_m in Section 2.2.4. Right now we are only interested in CRT_m, but it useful to define DFT_m and CRT_m at the same time, since they are so closely related.

Proposition 2.2.10. *Let $K = \mathbb{Q}(\zeta_m)$ be the m-th cyclotomic number field for some positive integer m and $R \subset K$ its ring of integers. Further let $\vec{p}$ be the powerful basis of R and $\sigma : K \to H$ the canonical embedding of K. If we apply σ entry wise to the row vector $\vec{p}^T$ and view the image as a matrix whose columns are the images of the basis elements, then we have the relation*

$$\sigma(\vec{p}^T) = \mathrm{CRT}_m,$$

where CRT_m is the Chinese remainder transform over $\mathbb{C}$ from Definition 2.2.8.

Proof. Let m be a prime power. Recall from Definition 2.2.1 that the powerful basis $\vec{p}$ is indexed by the set $[\varphi(m)]$. Since m is a prime power, the powerful basis equals the usual power basis and the row vector $\vec{p}^T$ is given by

$$\vec{p}^T = (1, \zeta_m, \ldots, \zeta_m^{\varphi(m)-1}).$$

Hence, applying σ to the j-th entry of $\vec{p}^T$ for some $j \in [\varphi(m)]$ yields a column vector containing the powers $\sigma_i(\zeta_m^j) = \omega_m^{ij}$, where i runs over $\mathbb{Z}_m^*$. Combining these column vectors to a matrix leads exactly to CRT_m.

Now, if m is an arbitrary positive integer with prime power factorization $m = \prod_{l=0}^{s} m_l$, the powerful basis is defined as the tensor product $\vec{p} = \bigotimes_{l=0}^{s} \vec{p}_l$, where $\vec{p}_l$ are the powerful bases of the rings of integers $R_l \subset K_l = \mathbb{Q}(\zeta_{m_l})$. Also the Chinese remainder transform is defined via the tensor product, i.e., $\mathrm{CRT}_m = \bigotimes_{l=0}^{s} \mathrm{CRT}_{m_l}$. From Equation (1.2) it follows that

$$\sigma\left(\bigotimes_{l=0}^{s} \vec{p}_l^T\right) = \bigotimes_{l=0}^{s} \sigma^{(l)}(\vec{p}_l^T) = \bigotimes_{l=0}^{s} \mathrm{CRT}_{m_l},$$

where $\sigma^{(l)}$ are the canonical embeddings of K_l. This finishes the proof. □

Corollary 2.2.11. *Let $a \in R$ be an element given by a coordinate vector $\mathbf{a} \in \mathbb{Z}^{[n]}$ in the powerful basis, i.e., $a = \langle \vec{p}, \mathbf{a} \rangle$. Then we can compute the embedding of a via multiplication of $\mathbf{a}$ with CRT_m, i.e.,*

$$\sigma(a) = \mathrm{CRT}_m \cdot \mathbf{a}.$$

Proof. Applying σ to a yields

$$\sigma(a) = \sigma(\langle \vec{p}, \mathbf{a} \rangle) = \sigma(\vec{p}^T \cdot \mathbf{a}) = \sigma(\vec{p}^T) \cdot \mathbf{a},$$

where we used in the last step that each entry of σ is a $\mathbb{Q}$-homomorphism and fixes in particular elements in $\mathbb{Z}$. Now with Proposition 2.2.10 it follows

$$\sigma(a) = \mathrm{CRT}_m \cdot \mathbf{a}.$$

□

Observation 2.2.12. *Proposition 2.2.10 and Corollary 2.2.11 imply direct relations for the inverse embedding σ^{-1}. If we want to retrieve the coordinate vector $\mathbf{a} \in \mathbb{Z}^{[n]}$ of an element $a \in R$, when we are given $\sigma(a) \in H$, it is sufficient to multiply $\sigma(a)$ with the inverse CRT_m^{-1}. Indeed we have*

$$\mathrm{CRT}_m^{-1} \cdot \sigma(a) = \mathrm{CRT}_m^{-1} \cdot \mathrm{CRT}_m \cdot \mathbf{a} = \mathbf{a}.$$

Combining these results with Equation (2.2) from Remark 2.2.7 yields a nice way of multiplying two elements in R. Let $a, b \in R$ and $\mathbf{a}, \mathbf{b} \in \mathbb{Z}^{[n]}$ their respective coordinate vectors with respect to the powerful basis. Then we can compute the coordinate vector $\mathbf{c}$ of ab in the powerful basis by multiplying $\mathbf{a}$ and $\mathbf{b}$ with CRT_m, component-wise multiplying the results and finally retrieving $\mathbf{c}$ via a multiplication with CRT_m^{-1}. In mathematical terms this means

$$\mathbf{c} = \mathrm{CRT}_m^{-1}((\mathrm{CRT}_m \cdot \mathbf{a}) \odot (\mathrm{CRT}_m \cdot \mathbf{b})), \tag{2.3}$$

where $ab = \langle \vec{p}, \mathbf{c} \rangle$.

We can extract a second corollary from Proposition 2.2.10 giving some insight about the length the powerful basis elements.

Corollary 2.2.13. *For each entry p_j in the powerful basis $\vec{p}$ we have that $\|p_j\|_2 = \sqrt{n}$ and $\|p_j\|_\infty = 1$.*

Proof. Recall from Section 1.4.2 that we defined the ℓ_2 andℓ_∞ norm in K via the canonical embedding σ. Thus, by Proposition 2.2.10, we have that $\|p_j\| = \|(\mathrm{CRT}_m)_{\cdot j}\|$, where $(\mathrm{CRT}_m)_{\cdot j}$ is the j-th column of CRT_m. Since each entry of CRT_m is a root of unity, hence has norm 1, we get that

$$\|(\mathrm{CRT}_m)_{\cdot j}\|_2 = \sqrt{\varphi(m)} = \sqrt{n} \quad \text{and} \quad \|(\mathrm{CRT}_m)_{\cdot j}\|_\infty = 1.$$

□

Multiplication in the CRT Basis

When regarding multiplication of elements in the quotient $R_q = R/qR$ for some prime $q = 1 \mod m$, we could try to operate in the powerful basis with a similar approach as for elements in R. However, as it turns out multiplication in the Chinese remainder basis $\vec{c}$ is much more efficient. Since we can easily switch from the powerful to the CRT basis, we will perform all multiplications in R_q with respect to $\vec{c}$.

Proposition 2.2.14. *Let $K = \mathbb{Q}(\zeta_m)$ be the m-th cyclotomic number field for some positive integer m and $R \subset K$ its ring of integers. Further let q be a prime such that $q = 1 \mod m$ and $R_q = R/qR$. Let $a, b \in R_q$ be two elements given by the coordinate vectors $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_q^{\mathbb{Z}_m^*}$ with respect to the Chinese remainder basis $\vec{c}$ of R_q. Then the coordinate vector $\mathbf{c}$ of the product $ab \in R_q$ is given by the coordinate-wise multiplication of $\mathbf{a}$ and $\mathbf{b}$, i.e.,*

$$ab = \langle \vec{c}, \mathbf{a} \rangle \cdot \langle \vec{c}, \mathbf{b} \rangle = \langle \vec{c}, \mathbf{a} \odot \mathbf{b} \rangle = \langle \vec{c}, \mathbf{c} \rangle .$$

Proof. Assume m is a prime power. For any fixed $i \in \mathbb{Z}_m^*$ and each $j \in \mathbb{Z}_m^*$ such that $i \neq j$, we have that

$$c_i^2 \mod \mathfrak{q}_i = (c_i \mod \mathfrak{q}_i) \cdot (c_i \mod \mathfrak{q}_i) = 1 \mod \mathfrak{q}_i,$$
$$c_i^2 \mod \mathfrak{q}_j = (c_i \mod \mathfrak{q}_j) \cdot (c_i \mod \mathfrak{q}_j) = 0 \mod \mathfrak{q}_j.$$

Recall from Definition 2.2.3 that the elements of the CRT basis are uniquely determined by those equations. Consequently, we have $c_i^2 = c_i \in R_q$ for each $i \in Z_m^*$. Furthermore, for an arbitrary $k \in \mathbb{Z}_m^*$ it holds that

$$(c_i \cdot c_j) \mod \mathfrak{q}_k = (c_i \mod \mathfrak{q}_k) \cdot (c_j \mod \mathfrak{q}_k) = 0 \mod \mathfrak{q}_k,$$

because either $c_i = 0 \mod \mathfrak{q}_k$ or $c_j = 0 \mod \mathfrak{q}_k$. Hence, we have that $c_i \cdot c_j = 0 \in R_q$ for any $i, j \in Z_m^*$ such that $i \neq j$.

Now, if we write $a = \sum_{i \in \mathbb{Z}_m^*} a_i \cdot c_i$ and $b = \sum_{i \in \mathbb{Z}_m^*} b_i \cdot c_i$, then the product $ab \in R_q$ is given by

$$ab = \left(\sum_{i \in \mathbb{Z}_m^*} a_i \cdot c_i \right) \cdot \left(\sum_{i \in \mathbb{Z}_m^*} b_i \cdot c_i \right) = \sum_{i \in \mathbb{Z}_m^*} \sum_{j \in \mathbb{Z}_m^*} a_i b_j \cdot (c_i c_j) = \sum_{i \in \mathbb{Z}_m^*} a_i b_i \cdot c_i,$$

since $c_i c_j = c_i$ for $i = j$, and $c_i c_j = 0$ otherwise.

Now suppose that m is an arbitrary positive integer with prime power factorization $m = \prod_{l=0}^{s} m_l$. By the tensorial decomposition $R_q = \bigotimes_{l=0}^{s} R_l / qR_l$, where R_l are the rings of integers in $K_l = \mathbb{Q}(\zeta_{m_l})$, we can represent two elements $a, b \in R_q$ as $a = \bigotimes_{l=0}^{s} a_l$ and $b = \bigotimes_{l=0}^{s} b_l$ for suitable $a_l, b_l \in R_l / qR_l$. Let $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_q^{\mathbb{Z}_m^*}$ be the respective coordinate vectors of a and b in $\vec{c}$. We can decompose $\mathbf{a}$ and $\mathbf{b}$ via the Kronecker product of vectors to $\mathbf{a} = \bigotimes_{l=0}^{s} \mathbf{a}_l$ and $\mathbf{b} = \bigotimes_{l=0}^{s} \mathbf{b}_l$, where $\mathbf{a}_l$ and $\mathbf{b}_l$ are the respective coordinate vectors of a_l and b_l in $\vec{c}_l$. Here, each $\vec{c}_l$ is the CRT basis of R_l / qR_l. Then, we have that

$$a = \bigotimes_{l=0}^{s} a_l = \bigotimes_{l=0}^{s} \langle \vec{c}_l, \mathbf{a}_l \rangle \quad \text{and} \quad b = \bigotimes_{l=0}^{s} b_l = \bigotimes_{l=0}^{s} \langle \vec{c}_l, \mathbf{b}_l \rangle.$$

By definition of the multiplication in the tensor product space, it follows that

$$ab = \bigotimes_{l=0}^{s} \langle \vec{c}_l, \mathbf{a}_l \rangle \cdot \langle \vec{c}_l, \mathbf{b}_l \rangle = \bigotimes_{l=0}^{s} \langle \vec{c}_l, \mathbf{a}_l \odot \mathbf{b}_l \rangle.$$

Thus, the coordinate vector $\mathbf{c}$ of ab in the CRT basis is given by $\mathbf{c} = \bigotimes_{l=0}^{s} \mathbf{a}_l \odot \mathbf{b}_l$. Finally, by Remark 1.2.4, we can interchange tensoring and component-wise multiplication and get

$$\mathbf{c} = \bigotimes_{l=0}^{s} \mathbf{a}_l \odot \mathbf{b}_l = \left(\bigotimes_{l=0}^{s} \mathbf{a}_l \right) \odot \left(\bigotimes_{l=0}^{s} \mathbf{b}_l \right) = \mathbf{a} \odot \mathbf{b}.$$

□

2.2.4 Sparse Decomposition of DFT and CRT

In Section 2.2.2 we saw that we can switch between representations in the powerful and in the CRT bases using the transformation matrix $\mathrm{CRT}_{m,q}$ from Definition 2.2.4 and its inverse. Further, in Section 2.2.3, we showed that multiplication in the powerful basis of R can be performed using multiplications with CRT_m and CRT_m^{-1} from Definition 2.2.8. In an implementation of the toolkit, the memory usage and complexity of application algorithms for CRT transformations are not very efficient, if we treat them as usual m-dimensional square matrices. The tensorial decomposition

$$\mathrm{CRT}_{m,q} = \bigotimes_{l=0}^{s} \mathrm{CRT}_{m_l,q} \quad \text{and} \quad \mathrm{CRT}_m = \bigotimes_{l=0}^{s} \mathrm{CRT}_{m_l}$$

of both transformations already allows us store only the matrices of prime power dimensions m_l, where $m = \prod_{l=0}^{s} m_l$ is the prime power factorization of m. This saves a good amount

of memory and in Section 3.1 we further develop an algorithm that speeds up computations taking advantage of the tensorial decomposition via parallelization. However, we can optimize the memory usage and efficiency of an application of the transformations even more. This is done via a sparse decomposition of the transformations for prime powers m, which holds for both, CRT_m and $\mathrm{CRT}_{m,q}$. These decompositions are developed in [PM08] for a more general algebraic framework and we present only the special case.

Recall that we defined not only the Chinese remainder transformations CRT_m and $\mathrm{CRT}_{m,q}$ in Definitions 2.2.4 and 2.2.8, but also the discrete Fourier transformations DFT_m and $\mathrm{DFT}_{m,q}$. These discrete Fourier transformations will play an essential role in the sparse decomposition and in fact we start with the decomposition of DFT_m.

Both, the discrete Fourier transformation DFT_m and the Chinese remainder transformation CRT_m will be decomposed in a way, such that in an application we do only need to apply matrices DFT_p and CRT_p for primes p. We can apply these matrices in the standard way using $O(p^2)$ operations. However, the special structure of DFT_p and CRT_p allows us to use Cooley-Tukey and Rader FFT algorithms, performing an application of the matrices in $O(p \log p)$. This is explained in more detail in Section 3.2.

Throughout this section ω_m will denote a m-th primitive root of unity in $\mathbb{C}$, which is used to define DFT_m and CRT_m. Further, we will describe the sparse decomposition only for DFT_m and CRT_m. If we view ω_m as a primitive m-th root of unity in $\mathbb{Z}_q$, which do exist, since $q = 1 \mod m$, every definition and equation that we develop can be adopted directly to $\mathrm{DFT}_{m,q}$ and $\mathrm{CRT}_{m,q}$ and their inverses.

Decomposition of DFT

Let m be a prime power of some prime p, and let $m' = m/p$. We can express DFT_m in terms of DFT_p and $\mathrm{DFT}_{m'}$, and by iterating even in terms of DFT_p alone. This is done by a Cooley-Tukey decomposition.

Proposition 2.2.15. *Let m be a prime power of some prime p, and let $m' = m/p$. Let ω_m be a primitive m-th root of unity in $\mathbb{C}$, which was used to define DFT_m. Define the "twiddle" matrix T_m as the m-dimensional diagonal matrix, whose entries are $\omega_m^{i,j}$ for $(i,j) \in [p] \times [m']$ in it's k-th diagonal position for $k = i \cdot m' + j$. Further let $L^m_{m'}$ be the permutation matrix representing the "bit-reversal" or "stride" permutation. Generally, the permutation L^m_d is defined for all $d|m$ by $i \mapsto i \cdot d \mod m - 1$ for $0 \leq i < m-1$ and $m-1 \mapsto m-1$. Then we can decompose the discrete Fourier transform to*

$$\mathrm{DFT}_m = L^m_{m'} \cdot (I_{[p]} \otimes \mathrm{DFT}_{m'}) \cdot T_m \cdot (\mathrm{DFT}_p \otimes I_{[m']}),$$

where all terms are square matrices of size $m = p \cdot m'$.

Proof. In order to verify the desired equation, it suffices to compare the action of both sides on the standard basis. For a more convenient computation we reindex the columns of DFT_m on the left side of the equation with pairs $(j_0, j_1) \in [p] \times [m']$, using the standard bijection $j = m'j_0 + j_1 \in [m]$. This corresponds to the index set of the Kronecker products on the right side of the equation (cf. Remark 1.2.2). Similarly we reindex the rows of DFT_m with pairs $(i_0, i_1) \in [p] \times [m']$, but this time with the non-standard bijection $i = pi_1 + i_0$. Note that this reindexation actually permutes the rows of DFT_m in the manner of $L^m_{m'}$. So in the computation we ignore the permutation $L^m_{m'}$.

Now, assume we take the (j_0, j_1)-th unit vector for any index $(j_0, j_1) \in [p] \times [m']$. Multiplication with $\mathrm{DFT}_p \otimes I_{[m']}$ results in a vector with entries $\omega_p^{i_0 j_0}$ in the (i_0, j_1)-th position for each $i_0 \in [p]$, and zero elsewhere. Applying the diagonal twiddle matrix changes these entries to $\omega_p^{i_0 j_0} \cdot \omega_m^{i_0 j_1}$. Note that the twiddle matrix is also reindexed by the standard correspondence. Finally, multiplication with $I_{[p]} \otimes \mathrm{DFT}_{m'}$ yields the vector with entries

$$\omega_p^{i_0 j_0} \cdot \omega_m^{i_0 j_1} \cdot \omega_{m'}^{i_1 j_1} = \omega_m^{m' i_0 j_0 + i_0 j_1 + p i_1 j_1} = \omega_m^{(p i_1 + i_0)(m' j_0 + j_1)}$$

for each $i_1 \in [m']$ and thus in any desired location $(i_0, i_1) \in [p] \times [m']$. In other words, applying the (j_0, j_1)-th unit vector to the left side of the equation yields exactly the (j_0, j_1)-th column of the DFT_m matrix on the left side. Consequently, the equation is correct. In the last step we used the fact that for any divisor n of m we have $\omega_n = \omega_m^{m/n}$. □

Decomposition of CRT

If we adjust the sparse decomposition of DFT_m from Proposition 2.2.15 slightly we get a similar decomposition for CRT_m.

Proposition 2.2.16. *Let m be a prime power of some prime p, and let $m' = m/p$. Further define the twiddle matrix T_m and the stride permutation $L_{m'}^{\varphi(m)}$ as in Proposition 2.2.15. [Note that $\varphi(m) = \varphi(p^k) = (p-1)p^{k-1} = \varphi(p) \cdot m'$ for some $k \geq 1$, thus $m' | \varphi(m)$ and the permutation $L_{m'}^{\varphi(m)}$ is well defined.] Denote by $\hat{T}_m$ the submatrix of T_m with columns restricted to $[\varphi(p)] \times [m']$ and rows restricted to $\mathbb{Z}_p^* \times [m']$. Then we can decompose the Chinese remainder transform to*

$$\mathrm{CRT}_m = L_{m'}^{\varphi(m)} \cdot (I_{\mathbb{Z}_p^*} \otimes \mathrm{DFT}_{m'}) \cdot \hat{T}_m \cdot (\mathrm{CRT}_p \otimes I_{[m']}),$$

where all terms are square matrices of size $\varphi(m) = \varphi(p) \cdot m'$.

Proof. Similar as in the proof of Proposition 2.2.15 we reindex the columns of CRT_m on the left side of the equation with pairs $(j_0, j_1) \in [\varphi(p)] \times [m']$ and the rows with pairs $(i_0, i_1) \in \mathbb{Z}_p^* \times [m']$. Again, the reindexation of the rows actually permutes them in the manner of $L_{m'}^{\varphi(m)}$, so we can ignore the permutation $L_{m'}^{\varphi(m)}$ in our computations.

If we perform the same steps as in the proof of Proposition 2.2.15, while keeping in mind that the rows of $\hat{T}_m$ and CRT_p are now indexed by pairs in $\mathbb{Z}_p^* \times [m']$ and the columns are indexed by pairs in $[\varphi(p)] \times [m']$, we get that the product on the right side of the equation has entries

$$\omega_m^{((p-1)i_1 + i_0)(m' j_0 + j_1)}$$

for any $(i_0, i_1) \in \mathbb{Z}_p^* \times [m']$ and $(j_0, j_1) \in [\varphi(p)] \times [m']$. This are exactly the entries of CRT_m with the above reindexation. □

Decomposition of the Inverses

Recall from Proposition 1.2.3 that $(A \otimes B)^{-1} = (A^{-1} \otimes B^{-1})$ for invertible and suitable dimensional matrices A and B. This together with the inversion rules of standard matrix multiplication leads to the following decompositions,

$$\mathrm{DFT}_m^{-1} = \left(\mathrm{DFT}_p^{-1} \otimes I_{[m']}\right) \cdot T_m^{-1} \cdot \left(I_{[p]} \otimes \mathrm{DFT}_{m'}^{-1}\right) \cdot (L_{m'}^m)^{-1},$$
$$\mathrm{CRT}_m^{-1} = \left(\mathrm{CRT}_p^{-1} \otimes I_{[m']}\right) \cdot \hat{T}_m^{-1} \cdot \left(I_{\mathbb{Z}_p^*} \otimes \mathrm{DFT}_{m'}^{-1}\right) \cdot \left(L_{m'}^{\varphi(m)}\right)^{-1}.$$

Note that all non-zero entries of the twiddle matrices are roots of unity and thus have magnitude one. Consequently, we have $T_m^{-1} = \overline{T_m}$, since T_m is diagonal and $\overline{z} \cdot z = ||z||^2$ for all $z \in \mathbb{C}$. The same holds for $\hat{T}_m$. Thus, we can restate the above equations as

$$\mathrm{DFT}_m^{-1} = \left(\mathrm{DFT}_p^{-1} \otimes I_{[m']}\right) \cdot \overline{T_m} \cdot \left(I_{[p]} \otimes \mathrm{DFT}_{m'}^{-1}\right) \cdot (L_{m'}^m)^{-1}, \tag{2.4}$$

$$\mathrm{CRT}_m^{-1} = \left(\mathrm{CRT}_p^{-1} \otimes I_{[m']}\right) \cdot \overline{\hat{T}_m} \cdot \left(I_{\mathbb{Z}_p^*} \otimes \mathrm{DFT}_{m'}^{-1}\right) \cdot \left(L_{m'}^{\varphi(m)}\right)^{-1}. \tag{2.5}$$

If we multiply two elements in R via CRT_m, we will always apply CRT_m^{-1} after we applied CRT_m. This causes the stride permutations to cancel each other out with the inverses. Consequently, in an implementation, we can omit the stride permutation completely. When we switch between the powerful an the CRT basis using $\mathrm{CRT}_{m,q}$, the stride permutation is important, since we apply only $\mathrm{CRT}_{m,q}$ or $\mathrm{CRT}_{m,q}^{-1}$ and not both. An easy computation shows that the inverse $\left(L_{m'}^{\varphi(m)}\right)^{-1}$ is given by

$$\left(L_{m'}^{\varphi(m)}\right)^{-1} = L_{\varphi(p)}^{\varphi(m)}.$$

2.2.5 Sampling Discrete Gaussians in R

A well known algorithm from linear algebra that generates an orthogonal basis out of any basis for a vector space V is the Gram-Schmidt orthogonalization. Let V be an n-dimensional vector space. For an ordered basis $B = \{b_i\}_{i\in[n]}$ of V, the Gram-Schmidt orthogonalization $\widetilde{B} = \{\tilde{b}_{i\in[n]}\}$ is defined iteratively by the following schema.

- First, define $\tilde{b}_0 := b_0$.
- Then, for $i = 1, 2, \ldots, n-1$, define $\tilde{b}_i$ as the component of b_i orthogonal to the linear span of $\tilde{b}_0, \ldots, \tilde{b}_{i-1}$, i.e.,

$$\tilde{b}_i := b_i - \sum_{j\in[i]} \tilde{b}_j \cdot \frac{\left\langle b_i, \tilde{b}_j \right\rangle}{\left\langle \tilde{b}_j, \tilde{b}_j \right\rangle}.$$

Viewing B as a matrix whose columns are the basis vectors b_i, its orthogonalization corresponds to the unique matrix factorization $B = QDU$. Thereby Q is unitary with columns $\tilde{b}_i/\|\tilde{b}_i\|_2$, D is real diagonal with positive entries $\|\tilde{b}_i\|_2 > 0$ and U is real upper unitriangular with entries $w_{j,i} = \left\langle b_i, \tilde{b}_j \right\rangle \Big/ \left\langle \tilde{b}_j, \tilde{b}_j \right\rangle$. With these matrices, the Gram-Schmidt orthogonalization is $\widetilde{B} = QD$ and thus $B = \widetilde{B}U$. The real positive definite Gram matrix of B is $B^*B = U^T D^2 U$. Since U is upper triangular, this is exactly the Cholesky decomposition of B^*B, which is unique. Therefore, the decomposition uniquely determines the matrices D and U in the Gram-Schmidt orthogonalization of B. Also, using the definitions, one may prove that D^2 and U are both rational if the Gram matrix was rational already.

The Gram-Schmidt orthogonalization becomes important for us, when we are dealing with the discrete Gaussian distribution over R. In [GPV08], Gentry, Peikert and Vaikuntanathan develop an efficient algorithm for sampling discrete Gaussians over any lattice Λ in H with standard deviation s, i.e., sampling from $D_{\Lambda+c,s}$. This algorithm uses a fixed basis B of the lattice Λ and, in particular, also its Gram-Schmidt orthogonalization $\widetilde{B}$.

Lemma 2.2.17 ([GPV08, Theorem 4.1])**.** *There is a probabilistic polynomial-time algorithm that samples to within small* $(\mathrm{negl}(n))$ *statistical distance of* $D_{\Lambda+c,s}$*, given* $c \in H$*, a basis* B *of* Λ*, and a parameter* $s \geq \max_j \|\tilde{b}_j\| \cdot \omega(\sqrt{\log n})$*, where* $\widetilde{B}$ *is the Gram-Schmidt orthogonalization of* B*.*

Remark 2.2.18. The statistical distance is a measure for the "difference" between two probability distributions. Further, $\mathrm{negl}(n)$ means a negligible function in n. That is, $\mathrm{negl}(n)$ is asymptotically smaller than n^{-c} for any constant $c > 0$, i.e., $\lim_{n\to\infty} n^c \cdot \mathrm{negl}(n) = 0$ for any constant $c > 0$. So, if we sample to within $\mathrm{negl}(n)$ statistical distance of $D_{\Lambda+c,s}$, means that we sample from a distribution that is not exactly the same, but indistinguishable from $D_{\Lambda+c,s}$.

Remark 2.2.19. Given an input c, B and s as in Lemma 2.2.17, the algorithm creates a Gaussian distributed element in the coset $\mathcal{L}(B) + c \subset H$. Further, the algorithm needs a subroutine (oracle) for sampling discrete Gaussians from $D_{\mathbb{Z},s,c'}$, i.e., Gaussians over $\mathbb{Z}$ centered at c' with standard deviation s. Given such a subroutine, the algorithm does the following. Let $c_{n-1} = c$. Then, for $i = n-1, \ldots, 0$, do:

- Define $c_i' := \left\langle c_i, \tilde{b}_i \right\rangle \Big/ \left\langle \tilde{b}_i, \tilde{b}_i \right\rangle$ and $s_i := s/\|\tilde{b}_i\| > 0$.
- Sample z_i from $D_{\mathbb{Z},s_i,c_i'}$.
- Let $c_{i-1} = c_i - z_i b_i$.

Finally, output c_0 as the sampled element.

Assume that the given basis B has a rational Gram matrix B^*B and that the element c can be represented by a rational coordinate vector in the basis B. From the above considerations about the Gram-Schmidt orthogonalization we know that the matrices D, U of the Cholesky decomposition of B^*B are also rational in this case. Recalling the definition of D and U, we observe that these matrices provide all the necessary information we need, in order to compute s_i and c_i'. The norm $\|\tilde{b}_i\|$ is just the i-th diagonal entry of D and the center c_i' is given by the inner product of the coordinate vector of c_i and the i-th row of U. Thus, in an implementation, D and U should be pre-computed to save further performance.

We can easily translate this algorithm into a sampling procedure for discrete Gaussians over R using the powerful basis $\vec{p}$ and the canonical embedding σ. Recall that $\sigma(\vec{p}^T) = \mathrm{CRT}_m$, so the given basis matrix B is actually CRT_m. The following lemma will give some insight about the Gram-Schmidt orthogonalization of CRT_m, or to be more precise, about the QDU decomposition of CRT_m. Indeed, as it follows from the lemma, the Gram matrix of CRT_m is rational and hence $\vec{p}$ is a good choice for sampling Gaussians in R.

The mixed-product property of the Kronecker product and the fact that QD uniquely determines the Gram-Schmidt orthogonalization imply that $\widetilde{A \otimes B} = \widetilde{A} \otimes \widetilde{B}$. Therefore, by the tensor structure of CRT_m, it suffices to consider the case of prime powers m.

Lemma 2.2.20. *Let* m *be a power of some prime* p *and* $m' = m/p$*. Then we have the decomposition*

$$\mathrm{CRT}_m = Q_m \cdot \left(\sqrt{m'} D_p \otimes I_{[m']}\right) \cdot \left(U_p \otimes I_{[m']}\right),$$

where Q_m *is a unitary matrix,* D_p *is the real diagonal* $[\varphi(p)] \times [\varphi(p)]$ *matrix with*

$$\sqrt{(p-1) - \frac{j}{(p-j)}}$$

in its j-th diagonal entry, and U_p is the upper unitriangular $[\varphi(p)] \times [\varphi(p)]$ matrix with

$$\frac{-1}{p-i-1}$$

in its (i,j)-th entry for $0 \leq i < j < \varphi(p)$.

Proof. Recall the sparse decomposition of CRT_m from Proposition 2.2.16

$$\mathrm{CRT}_m = \left(I_{[p]} \otimes \mathrm{DFT}_{m'}\right) \cdot \hat{T}_m \cdot \left(\mathrm{CRT}_p \otimes I_{[m']}\right).$$

By the fact that $\hat{T}_m$ and $\frac{1}{\sqrt{m'}} \cdot \mathrm{DFT}_{m'}$ are unitary matrices, it follows that

$$\mathrm{CRT}_m = \sqrt{m'} Q' \cdot \left(\mathrm{CRT}_p \otimes I_{[m']}\right)$$

for some unitary Q'. Hence, it suffices to show that CRT_p decomposes as $\mathrm{CRT}_p = Q_p \cdot D_p \cdot U_p$ for some unitary Q_p.

Let $G = \mathrm{CRT}_p^* \cdot \mathrm{CRT}_p$ be the Gram matrix of CRT_p. Since p is a prime, we have that $\mathbb{Z}_p^* = \{1, \ldots, p-1\}$ and $[\varphi(p)] = \{0, \ldots, p-2\}$. Thus for any $j \in [\varphi(p)]$ the j-th column of CRT_p is given by $(\omega_p^{1 \cdot j}, \ldots, \omega_p^{(p-1) \cdot j})^T$. For roots of unity in $\mathbb{C}$, inversion and complex conjugation coincide, hence for any $i \in \mathbb{Z}_p^*$ the i-th row of CRT_p^* is given by $(\omega_p^{-1 \cdot i}, \ldots, \omega_p^{-(p-1) \cdot i})$. Multiplying these vectors yields the (i,j)-th entry of G

$$G_{(i,j)} = \sum_{k=1}^{p-1} \omega_p^{k \cdot i - k \cdot j} = \sum_{k=1}^{p-1} (\omega_p^{i-j})^k.$$

But ω_p^{i-j} is just another primitive p-th root of unity $\omega_p' \in \mathbb{C}$, except when $i = j$. So, if $i \neq j$, we have $G_{(i,j)} = (\omega_p')^1 + \ldots + (\omega_p')^{p-1} = -1$, and $G_{(i,i)} = p - 1$ otherwise. Therefore, the matrix G is given as

$$G = \begin{pmatrix} p-1 & -1 & \cdots & -1 \\ -1 & \ddots & \ddots & \cdots \\ \vdots & \ddots & \ddots & -1 \\ -1 & \cdots & -1 & p-1 \end{pmatrix}.$$

By uniqueness of the Cholesky decomposition it suffices to show that

$$G = U_p^T \cdot D^2 \cdot U_p.$$

This is done by elementary calculations. For $k \geq 2$ define

$$T(k) := \frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \cdots + \frac{1}{(k-1) \cdot k}.$$

Adding $1/k$ to the above sum leads to

$$T(k) + \frac{1}{k} = T(k-1) + \frac{1}{(k-1) \cdot k} + \frac{1}{k} = T(k-1) \frac{1 + (k-1)}{(k-1) \cdot k} = T(k-1) + \frac{1}{k-1}.$$

Since $T(2) + 1/2 = 1$, by induction we have that $T(k) = 1 - 1/k$. Closely examining the multiplication $U_p^T \cdot D^2 \cdot U_p$, we observe that for any $i \in [\varphi(p)]$ the i-th diagonal entry is given by

$$p - 1 - \frac{i}{p-i} + \sum_{k=0}^{i-1} \frac{1}{(p-k-1)^2} \left(p - 1 - \frac{k}{p-k} \right).$$

Considering only the latter summation in the above expression, we see that

$$\begin{aligned}
&\sum_{k=0}^{i-1} \frac{1}{(p-k-1)^2} \left(p - 1 - \frac{k}{p-k} \right) \\
&= \sum_{k=0}^{i-1} \frac{p-1}{(p-k-1)^2} - \frac{k}{(p-k-1)^2 \cdot (p-k)} \\
&= \sum_{k=0}^{i-1} \frac{(p-1)(p-k) - k}{(p-k-1)^2 \cdot (p-k)} \\
&= \sum_{k=0}^{i-1} \frac{p(p-k-1)}{(p-k-1)^2 \cdot (p-k)} \\
&= p \sum_{k=0}^{i-1} \frac{1}{(p-k-1)(p-k)}.
\end{aligned}$$

Now, observe that the elements in the last sum are exactly those who lie "between" $T(p)$ and $T(p-i)$. Thus,

$$\begin{aligned}
p \sum_{k=0}^{i-1} \frac{1}{(p-k-1)(p-k)} &= p(T(p) - T(p-i)) \\
= p \left(1 - \frac{1}{p} - 1 + \frac{1}{p-i} \right) &= \frac{p}{p-i} - 1 = \frac{i}{p-i},
\end{aligned}$$

and the i-th diagonal entry of $U_p^T \cdot D^2 \cdot U_p$ is $p-1$, as required. All other entries are calculated similarly and the calculation is left to the reader. □

Remark 2.2.21. Let m be an arbitrary positive integer with prime power factorization $m = \prod_{l=0}^{s} m_l$, where each m_l is the power of some prime p_l, and $m_l' = m_l / p_l$. From the above lemma it follows that the matrices D and U in the decomposition of CRT_m are given by

$$D = \bigotimes_{l=0}^{s} \left(\sqrt{m'} D_{p_l} \otimes I_{[m_l']} \right)$$

and

$$U = \bigotimes_{l=0}^{s} \left(U_{p_l} \otimes I_{[m_l']} \right).$$

According to the above discussion, knowing the QDU decomposition of CRT_m allows us to efficiently sample discrete Gaussians over R in the powerful basis. When doing so, we have to compute inner products with the rows of U. Through the Kronecker decomposition, an inner

product with a row of U can be reduced to inner products with the respective rows in all U_{p_l}. Note that each row of U_{p_l} has a small common denominator. Thus, if the element c has an integral coordinate vector (which will be the case), one could compute the inner product using integer arithmetic and divide the result by the common denominator. However, due to the structure of our implementation we have to use another approach. In the end, we will compute the inner product using real arithmetic, which is only a small disadvantage. See Section 4.5.2 for details.

2.3 Working in the Dual Ideal $R^\vee$

For our applications, arithmetic operations in the dual ideal $R^\vee$ of R are equally important to those in R itself. In Section 2.2 we developed efficient methods for addition and multiplication in the ideal R, which depend heavily on the specific bases we defined for R. In this Section we will develop similar methods for arithmetic operations in $R^\vee$ and power $\mathcal{I}_k = (R^\vee)^k$ for some $k \geq 2$. Again, we define several specific bases and show that they are a good choice for usage in our algorithms. Those bases are closely related to the specific bases of R and provide similar algorithms for bases switching as well as for addition and multiplication.

2.3.1 Three Specific Bases of $R^\vee$

The Powerful and the Chinese Remainder Basis of $R^\vee$

Recall from Section 1.4.4 that the dual ideal $R^\vee$ is generated by the element t^{-1} as defined in Definition 1.4.21, i.e., $R^\vee = \langle t^{-1} \rangle = t^{-1}R$. This implies that each $\mathbb{Z}$-basis $\vec{b}$ of R can be transformed into a $\mathbb{Z}$-basis of $R^\vee$ via multiplication of the basis elements by t^{-1}, i.e., $t^{-1}\vec{b}$ is a $\mathbb{Z}$-basis of $R^\vee$. The same holds for $\mathbb{Z}_q$-bases $\vec{b}'$ of R_q, which then get $\mathbb{Z}_q$-bases of $R_q^\vee$. Using this fact we can define the powerful and the Chinese remainder bases for $R^\vee$ as follows.

Definition 2.3.1. Let $K = \mathbb{Q}(\zeta_m)$ be the m-th cyclotomic number field for some positive integer m and $R \subset K$ its ring of integers. Further, let $R^\vee$ be the dual ideal of R and $\mathcal{I}_k = (R^\vee)^k$ a power of $R^\vee$ for some integer $k \geq 1$. For a prime $q = 1 \mod m$ let $R_q = R/qR$, $R_q^\vee = R^\vee/qR^\vee$ and $\mathcal{I}_{k,q} = (R_q^\vee)^k$. Let $\vec{p}$ be the powerful basis of R from Definition 2.2.1 and $\vec{c}$ be the Chinese remainder basis of R from Definition 2.2.3. Then we define the *powerful basis of* the power $\mathcal{I}_k$ as $t^{-k}\vec{p}$ and the *Chinese remainder basis of* the power $\mathcal{I}_{k,q}$ as $t^{-k}\vec{c}$.

The Decoding Basis of $R^\vee$

For an arbitrary positive integer m and $K = \mathbb{Q}(\zeta_m)$ define the $\mathbb{Q}$-homomorphism $\tau : K \to K$ as the map that maps the primitive root of unity ζ_m to its inverse, i.e., $\tau(\zeta_m) = \zeta_m^{-1} = \zeta_m^{m-1}$. The image of the power basis $(1, \zeta_m, \ldots, \zeta_m^{\varphi(m)})$ under τ is again a basis, so τ is an automorphism. Furthermore, τ is an involution, since $\tau(\tau(\zeta_m)) = (\zeta_m^{-1})^{-1} = \zeta_m$. Let $\{\sigma_i\}_{i \in \mathbb{Z}_m^*}$ be the $n = \varphi(m)$ distinct $\mathbb{Q}$-homomorphisms from K to $\mathbb{C}$ that define the canonical embedding σ of K (cf. Section 1.4.2). For each $i \in \mathbb{Z}_m^*$ we have that

$$\sigma_i(\tau(\zeta_m)) = \sigma_i(\zeta_m^{-1}) = \omega_m^{-i} = \omega_m^{m-i} = \overline{\omega_m^i} = \overline{\sigma(\zeta_m)},$$

where $\omega_m \in \mathbb{C}$ is a primitive m-th root of unity in $\mathbb{C}$. The computation shows that under the canonical embedding σ, τ corresponds to complex conjugation, i.e., $\sigma(\tau(a)) = \overline{\sigma(a)}$ for each

$a \in K$. That is why we refer to τ as the conjugation map. Note that τ actually corresponds to σ_{m-1}. For any m' dividing m, τ still maps $\zeta_{m'} = \zeta_m^{m/m'}$ to its inverse $\zeta_{m'}^{-1} = \zeta_m^{-m/m'}$. Further, the image of the powerful basis $\tau(\vec{p})$ is a $\mathbb{Z}$-basis of R, because τ is an automorphism and hence $\tau(R) = R$.

Definition 2.3.2. Let m be an arbitrary positive integer and $K = \mathbb{Q}(\zeta_m)$ be the m-th cyclotomic number field. Further let $R \subset K$ be the ring of integers in K and $R^\vee$ its dual ideal. Then we define the $\mathbb{Z}$-basis $\vec{d}$ for $R^\vee$ as the dual of the conjugate powerful basis, i.e., $\vec{d} := \tau(\vec{p})^\vee$. We call $\vec{d}$ the *decoding basis* of $R^\vee$.

Remark 2.3.3. By definition, the decoding basis has the same index set as the powerful basis. For prime powers m we have $\tau(\vec{p}) = (\zeta_m^{-j})_{j \in [\varphi(m)]}$. Hence, $\vec{d}$ is the dual of the conjugate power basis of R. For arbitrary $m \in \mathbb{N}$ with prime power factorization $m = \prod_{l=0}^{s} m_l$ it is immediate that

$$\tau(\vec{p}) = \tau\left(\bigotimes_{l=0}^{s} \vec{p}_l\right) = \bigotimes_{l=0}^{s} \tau(\vec{p}_l),$$

where $\vec{p}_l$ are the powerful bases of the rings of integers $R_l \subset K_l$ and $K_l = \mathbb{Q}(\zeta_{m_l})$. Furthermore, recall from Section 1.4.4 that $(\vec{a} \otimes \vec{b})^\vee = (\vec{a}^\vee \otimes \vec{b}^\vee)$. These two facts imply that the decoding basis $\vec{d}$ for arbitrary m is the tensor product of the decoding bases of each $R_l^\vee$.

2.3.2 Switching between the Specific Bases of $R^\vee$

Similar to the results from Section 2.2.2 we want to develop a method to switch between the bases of $R^\vee$ that we defined above. A first thing to observe is that by linearity of the inner product in K we can switch between the powerful and the Chinese remainder basis of $R_q^\vee$ just as we did in R_q. That is, for an element $a \in R_q^\vee$ given by some coordinate vector $\mathbf{a} \in \mathbb{Z}_q^{[n]}$ in the powerful basis $t^{-1}\vec{p}$ we have that

$$a = \left\langle t^{-1}\vec{p}, \mathbf{a} \right\rangle = t^{-1} \left\langle \vec{p}, \mathbf{a} \right\rangle = t^{-1} \left\langle \vec{c}, \mathrm{CRT}_{m,q} \cdot \mathbf{a} \right\rangle = \left\langle t^{-1}\vec{c}, \mathrm{CRT}_{m,q} \cdot \mathbf{a} \right\rangle, \tag{2.6}$$

where we used Equation (2.1) in the third step. Similarly, we can switch from $t^{-1}\vec{c}$ to $t^{-1}\vec{p}$ using $\mathrm{CRT}_{m,q}^{-1}$.

The following proposition states how the powerful and the decoding basis are related. We can use this relation to switch efficiently between those bases.

Proposition 2.3.4. *Let m be some positive integer and $K = \mathbb{Q}(\zeta_m)$ be the m-th cyclotomic number field. Further let $R \subset K$ be the ring of integers in K and $R^\vee$ its dual ideal. Let $t^{-1}\vec{p}$ be the powerful basis of $R^\vee$ and $\vec{d}$ the decoding basis. For a prime p denote by $L_p \in \mathbb{Z}^{[\varphi(p)] \times [\varphi(p)]}$ the lower-triangular matrix, whose (i,j)-th entry is one for $i \geq j$, and zero otherwise. Let $m = \prod_{l=0}^{s} m_l$ be the prime power factorization of m such that m_l are prime powers of some primes p_l. Define $m_l' = m_l/p_l$ and $L_m = \bigotimes_{l=0}^{s} L_{m_l} = \bigotimes_{l=0}^{s} (L_{p_l} \otimes I_{[m_l']})$. Then we have the relation*

$$\vec{d}^T = t^{-1}\vec{p}^T \cdot L_m. \tag{2.7}$$

Proof. We can view both, $t^{-1}\vec{p}^T$ and $\vec{d}^T$, as Kronecker decompositions

$$t^{-1}\vec{p}^T = \bigotimes_{l=0}^{s} t^{-1}\vec{p}_l^T \quad \text{and} \quad \vec{d}^T = \bigotimes_{l=0}^{s} \vec{d}_l^T,$$

where $t^{-1}\vec{p}_l^T$ and $\vec{d}_l^T$ are the powerful and decoding bases of the respective dual ideals $R_l^\vee$ of the ring of integers $R_l \subset K_l = \mathbb{Q}(\zeta_{m_l})$. Then, by the mixed-product property of the Kronecker product it suffices to prove that

$$\vec{d}_l^T = t^{-1}\vec{p}_l^T(L_{p_l} \otimes I_{[m'_l]}),$$

i.e., we can reduce our focus to prime powers.

Let m be a prime power of some prime p and $m' = m/p$. In order to simplify the computation, instead of $[n]$, we use the bijective index set $[\varphi(p)] \times [m']$. The bijection is given by $(j_0, j_1) \mapsto j = j_0 m' + j_1$. For the powerful basis this means that

$$p_j = p_{(j_0,j_1)} = \zeta_m^j = \zeta_m^{j_0 m' + j_1} = (\zeta_m^{m'})^{j_0} \cdot \zeta_m^{j_1} = \zeta_p^{j_0} \cdot \zeta_m^{j_1}.$$

With this indexation and the way that the Kronecker product is defined, observe that the matrix $(L_p \otimes I_{[m']})$ consists of pure one diagonals starting at each row indexed by $(j_0, 0)$ for $j_0 \in [\varphi(p)]$. Therefore, we can rewrite Equation (2.7) as

$$d_{(j_0,j_1)} = t^{-1} \cdot (\zeta_p^{j_0} + \zeta_p^{j_0+1} + \ldots + \zeta_p^{p-2}) \cdot \zeta_m^{j_1}.$$

Recalling from Definition 1.4.21 that $t^{-1} = (1 - \zeta_p)/m$ yields

$$\begin{aligned}
&\frac{1-\zeta_p}{m} \cdot (\zeta_p^{j_0} + \zeta_p^{j_0+1} + \ldots + \zeta_p^{p-2}) \cdot \zeta_m^{j_1} \\
&= \frac{1}{m} \cdot ((\zeta_p^{j_0} - \zeta_p^{j_0+1}) + (\zeta_p^{j_0+1} - \zeta_p^{j_0+2}) + \ldots + (\zeta_p^{p-2} - \zeta_p^{p-1})) \cdot \zeta_m^{j_1} \\
&= \frac{1}{m} \cdot (\zeta_p^{j_0} - \zeta_p^{p-1}) \cdot \zeta_m^{j_1}.
\end{aligned}$$

To verify this equation we need to prove that the right-hand side meets the characterization of the dual basis. Thus, since $d_{(j_0,j_1)} = \tau(p_{(j_0,j_1)})^\vee$, the trace of the product of the right-hand side with $\tau(p_{(j'_0,j'_1)})$ has to be one if $(j_0, j_1) = (j'_0, j'_1)$ and zero otherwise. Note that with the indexation we use, the element $\tau(p_{(j'_0,j'_1)})$ is given by $\zeta_p^{-j'_0} \cdot \zeta_m^{-j'_1}$. For any $(j'_0, j'_1) \in [\varphi(p)] \times [m']$ we have

$$\left(\frac{1}{m} \cdot (\zeta_p^{j_0} - \zeta_p^{p-1}) \cdot \zeta_m^{j_1}\right) \cdot (\zeta_p^{-j'_0} \cdot \zeta_m^{-j'_1}) = \frac{1}{m} \cdot (\zeta_p^{j_0-j'_0} - \zeta_p^{p-1-j'_0}) \cdot \zeta_m^{j_1-j'_1}.$$

Next, we need to compute the trace of this term. Using $\zeta_p = \zeta_m^{m'}$ and the linearity of the trace, it holds that

$$\mathrm{Tr}\left(\frac{1}{m}(\zeta_p^{j_0-j'_0} - \zeta_p^{p-1-j'_0})\zeta_m^{j_1-j'_1}\right) = \frac{1}{m}\left(\mathrm{Tr}\left(\zeta_m^{m'(j_0-j'_0)+(j_1-j'_1)}\right) - \mathrm{Tr}\left(\zeta_m^{m'(p-1-j'_0)+(j_1-j'_1)}\right)\right).$$

By Lemma 1.4.19, if $j_1 \neq j'_1$, then $j_1 - j'_1 \neq 0 \mod m'$ and both traces are zero. Further, if $j_1 = j'_1$ but $j_0 \neq j'_0$, then both $j_0 - j'_0$ and $p - 1 - j'_0$ are not equal to zero mod p, which implies that for both traces we are in the second case of Lemma 1.4.19. Thus, the whole term is zero. Finally, if $(j_0, j_1) = (j'_0, j'_1)$, we have

$$\begin{aligned}
&\frac{1}{m}\left(\mathrm{Tr}\left(\zeta_m^{m'(j_0-j'_0)+(j_1-j'_1)}\right) - \mathrm{Tr}\left(\zeta_m^{m'(p-1-j'_0)+(j_1-j'_1)}\right)\right) \\
&= \frac{1}{m}\left(\mathrm{Tr}\left(\zeta_m^0\right) - \mathrm{Tr}\left(\zeta_m^{m'(p-1-j'_0)}\right)\right) = \frac{1}{m}\left(\mathrm{Tr}\left(1\right) - (-m')\right) \\
&= \frac{1}{m}\left(\varphi(m) + m'\right) = \frac{1}{m}\left(\varphi(p)m' + m'\right) = \frac{pm'}{m} = 1,
\end{aligned}$$

which ends the proof. □

Remark 2.3.5. Let $a \in R^\vee$ be given by a coordinate vector $\mathbf{a} \in \mathbb{Z}^{[n]}$ in the basis $\vec{d}$. Then, we have that

$$a = \left\langle \vec{d}, \mathbf{a} \right\rangle = \left\langle t^{-1}\vec{p}^T \cdot L_m, \mathbf{a} \right\rangle = \left\langle t^{-1}\vec{p}, L_m \cdot \mathbf{a} \right\rangle. \tag{2.8}$$

Thus, the coordinate vector of a with respect to the basis $t^{-1}\vec{p}$ is given by $L_m \cdot \mathbf{a}$. Similarly, we can switch from the basis $t^{-1}\vec{p}$ to $\vec{d}$ by multiplication with L_m^{-1}. Since L_p is defined as

$$L_p = \begin{pmatrix} 1 & & \\ \vdots & \ddots & \\ 1 & \cdots & 1 \end{pmatrix}$$

the inverse is given by

$$L_p^{-1} = \begin{pmatrix} 1 & & & \\ -1 & \ddots & & \\ & \ddots & \ddots & \\ & & -1 & 1 \end{pmatrix}.$$

Both matrices can be applied using $O(\varphi(p))$ scalar operations via successive sums and partial differences (cf. Section 4.5.3), which makes the switch between the bases efficient.

2.3.3 Addition and Multiplication in $R^\vee$

Addition and multiplication in the dual ideal $R^\vee$ is equally important to the same operations in R. In Section 2.2.3 we described how we add and multiply elements in R. As before, by linearity, addition of two elements corresponds to component-wise addition of the coordinate vectors, if they are with respect to the same bases. Since we can switch efficiently between all considered bases of $R^\vee$, we can always assume that the elements are represented in the same basis. The same holds for addition in $R_q^\vee$.

When dealing with multiplication we have to distinguish between the modulo case, where we multiply in $R_q^\vee$, and the normal case in $R^\vee$. In the modulo case, multiplication works nearly as in R_q. Let $a, b \in R_q^\vee$ be two elements and $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_q^{\mathbb{Z}_m^*}$ their respective coordinate vectors in the CRT basis $t^{-1}\vec{c}$ of $R_q^\vee$. By linearity and Proposition 2.2.14, we have that

$$ab = \left\langle t^{-1}\vec{c}, \mathbf{a} \right\rangle \cdot \left\langle t^{-1}\vec{c}, \mathbf{b} \right\rangle = t^{-2} \left\langle \vec{c}, \mathbf{a} \odot \mathbf{b} \right\rangle = \left\langle t^{-2}\vec{c}, \mathbf{a} \odot \mathbf{b} \right\rangle.$$

Now, $t^{-2}\vec{c}$ is a $\mathbb{Z}_q$-basis of the power $\mathcal{I}_{2,q} = (R_q^\vee)^2 \subset R_q^\vee$. In a more general setting, where $a \in \mathcal{I}_{k_1,q}$ and $b \in \mathcal{I}_{k_2,q}$ for some $k_1, k_2 \geq 1$, the same computation yields that

$$ab = \left\langle t^{-(k_1+k_2)}\vec{c}, \mathbf{a} \odot \mathbf{b} \right\rangle \in \mathcal{I}_{k_1+k_2,q} \tag{2.9}$$

i.e., $\mathbf{a} \odot \mathbf{b}$ is the coordinate vector of $ab \in \mathcal{I}_{k,q}$ in the CRT basis $t^{-k}\vec{c}$, where $k = k_1 + k_2$. Together with Proposition 2.2.14 this is even true for $k_1, k_2 \geq 0$.

For normal multiplication in $R^\vee$ we use the same approach as in R. That is, we multiply two elements $a, b \in R^\vee$ via the canonical embedding σ, i.e.,

$$ab = \sigma^{-1}(\sigma(a) \odot \sigma(b)).$$

To make this operation efficient we need an efficient computation of σ in the decoding basis.

Proposition 2.3.6. *Let $K = \mathbb{Q}(\zeta_m)$ be the m-th cyclotomic number field for some positive integer m and $R \subset K$ its ring of integers. Further let $\vec{d}$ be the powerful basis of the dual ideal $R^\vee$ and $\sigma : K \to H$ the canonical embedding of K. For any element $a \in K_\mathbb{R} = K \otimes \mathbb{R}$ such that $a = \left\langle \vec{d}, \mathbf{a} \right\rangle$ for a suitable coordinate vector $\mathbf{a} \in \mathbb{R}^{[n]}$ (such a coordinate vector exists and is unique, since any $\mathbb{Q}$-basis of K is a $\mathbb{R}$-basis of $K_\mathbb{R}$) we have that*

$$\sigma(a) = (\mathrm{CRT}_m^*)^{-1} \cdot \mathbf{a},$$

where CRT_m is the Chinese remainder transform over $\mathbb{C}$ from Definition 2.2.8. In particular, we have that

$$\sigma(\vec{d}^T) = (\mathrm{CRT}_m^*)^{-1}.$$

Proof. Recall the characterization of the dual basis from Section 1.4.4. Given the dual basis $\{d_j^\vee\}_{j \in [n]}$, each entry a_j of $\mathbf{a}$ for $j \in [n]$ can be computed as

$$\mathrm{Tr}(a \cdot d_j^\vee) = \mathrm{Tr}\left(\sum_{i \in [n]} a_i d_i \cdot d_j^\vee \right) = \sum_{i \in [n]} \left(a_i \cdot \mathrm{Tr}(d_i d_j^\vee) \right) = a_j,$$

since $\mathrm{Tr}(d_i d_j^\vee) = \delta_{ij}$. Keeping the definition of the decoding basis in mind, it follows that

$$a_j = \mathrm{Tr}(a \cdot d_j^\vee) = \mathrm{Tr}(a \cdot \tau(p_j)) = \sum_{i \in \mathbb{Z}_m^*} (\sigma_i(a)\sigma_i(\tau(p_j))) = \sum_{i \in \mathbb{Z}_m^*} (\sigma_i(a)\overline{\sigma_i}(p_j)) = \langle \sigma(a), \sigma(p_j) \rangle_H .$$

Note that the second entry of $\langle \cdot, \cdot \rangle_H$ gets conjugated by definition. Since $\sigma(\vec{p}) = \mathrm{CRT}_m$, the latter equation is equivalent to

$$\mathbf{a} = \mathrm{CRT}_m^* \cdot \sigma(a).$$

Applying $(\mathrm{CRT}_m^*)^{-1}$ leads to the desired equation. □

The latter result is of greatest interest for us, if we look at elements $a, b \in R^\vee$, whose coordinate vectors $\mathbf{a}$ and $\mathbf{b}$ are in $\mathbb{Z}^{[n]}$. Then we can perform a multiplication of a and b via

$$ab = \mathrm{CRT}_m^*(((\mathrm{CRT}_m^*)^{-1} \cdot \mathbf{a}) \odot ((\mathrm{CRT}_m^*)^{-1} \cdot \mathbf{b})). \tag{2.10}$$

"Mixed" Addition and Multiplication

A difficulty arises if we want to add or multiply two elements a, b, where $a \in R$ and $b \in R^\vee$. We know that $R \subset R^\vee$, so we could view a as an element in $R^\vee$ and perform the desired operation in $R^\vee$. However, we have no efficient procedure that computes the coordinate vector of a in $t^{-1}\vec{p}$ or $\vec{d}$ given the coordinate vector of a in $\vec{p}$. A possibility would be a multiplication with t. If $\mathbf{b}$ is the coordinate vector of ta in $\vec{p}$, then we have

$$\left\langle t^{-1}\vec{p}, \mathbf{b} \right\rangle = t^{-1} \left\langle \vec{p}, \mathbf{b} \right\rangle = t^{-1}ta = a,$$

i.e., $\mathbf{b}$ would be the coordinate vector of a in $t^{-1}\vec{p}$. For an efficient multiplication with t we would need its coordinate vector in the basis $\vec{p}$. Unfortunately, the computation of this coordinate vector is rather complex and we would like to omit it completely.

Without a coordinate vector of a in some basis of $R^\vee$ we are not able to perform any operation in $R^\vee$. We can solve this problem by the following observation. In both, the powerful

and the decoding bases, we can easily embed elements under the canonical embedding σ into H via the CRT matrices. If we represent a in $\vec{p}$ and b in $\vec{d}$, we can compute the coordinate vector $\mathbf{c}$ of $a+b$ or ab in $\vec{d}$ as

$$\mathbf{c} = \mathrm{CRT}_m^*(\mathrm{CRT}_m \cdot \mathbf{a} \diamond (\mathrm{CRT}_m^*)^{-1} \cdot \mathbf{b}),$$

where $\mathbf{a}$ and $\mathbf{b}$ are the respective coordinate vectors of a and b and $\diamond \in \{+, \odot\}$.

2.3.4 Sparse Decomposition of DFT^* and CRT^*

The previous section introduced an efficient method for the multiplication of elements in $R^\vee$. The method uses the conjugate transpose of the Chinese remainder transformation CRT_m^* and its inverse. Similar to CRT_m we can decompose CRT_m^* to

$$\mathrm{CRT}_m^* = \bigotimes_{l=0}^{s} \mathrm{CRT}_{m_l}^*,$$

where $m = \prod_{l=0}^{s} m_l$ is the prime power factorization of m. To speed up algorithmic applications of $\mathrm{CRT}_{m_l}^*$, we can adopt the sparse decomposition of CRT_{m_l} from Section 2.2.4. A sparse decomposition of $\mathrm{CRT}_{m_l}^*$ also needs a sparse decomposition of $\mathrm{DFT}_{m_l}^*$. The same holds for the inverse $(\mathrm{CRT}_m^*)^{-1}$.

Proposition 2.3.7. *Let m be a prime power of some prime p, and let $m' = m/p$. Let ω_m be a primitive m-th root in $\mathbb{C}$, which was used to define DFT_m. Define the "twiddle" matrix T_m as the m-dimensional diagonal matrix, whose entries are $\omega_m^{i,j}$ for $(i,j) \in [p] \times [m']$ in it's k-th diagonal position for $k = i \cdot m' + j$. Further, let $L_{m'}^m$ be the permutation matrix representing the "bit-reversal" or "stride" permutation. Generally, the permutation L_d^m is defined for all $d|m$ by $i \mapsto i \cdot d \mod m-1$ for $0 \le i < m-1$ and $m-1 \mapsto m-1$. Then we can decompose the conjugate transpose of the discrete Fourier transform to*

$$\mathrm{DFT}_m^* = \left(\overline{\mathrm{DFT}_p} \otimes I_{[m']}\right) \cdot \overline{T_m} \cdot \left(I_p \otimes \overline{\mathrm{DFT}_{m'}}\right) \cdot (L_{m'}^m)^{-1},$$

where all terms are square matrices of size $m = p \cdot m'$.

Proposition 2.3.8. *Let m be a prime power of some prime p, and let $m' = m/p$. Further define the twiddle matrix T_m and the stride permutation $L_{m'}^{\varphi(m)}$ as in Proposition 2.3.7. [Note that $\varphi(m) = \varphi(p^k) = (p-1)p^{k-1} = \varphi(p) \cdot m'$ for some $k \ge 1$, thus $m' | \varphi(m)$ and the permutation $L_{m'}^{\varphi(m)}$ is well defined.] Denote by $\hat{T}_m$ the submatrix of T_m with columns restricted to $[\varphi(p)] \times [m']$ and rows restricted to $\mathbb{Z}_p^* \times [m']$. Then we can decompose the conjugate transpose of the Chinese remainder transform to*

$$\mathrm{CRT}_m^* = \left(\mathrm{CRT}_p^* \otimes I_{[m']}\right) \cdot \overline{\hat{T}_m} \cdot \left(I_{\mathbb{Z}_p^*} \otimes \overline{\mathrm{DFT}_{m'}}\right) \cdot \left(L_{m'}^{\varphi(m)}\right)^{-1},$$

where all terms are square matrices of size $\varphi(m) = \varphi(p) \cdot m'$.

Proof of Propositions 2.3.7 and 2.3.8. Both propositions follow from the sparse decomposition of DFT_m and CRT_m from Propositions 2.2.15 and 2.2.16. Using Proposition 1.2.3 and the usual rules for conjugate matrices we get

$$\mathrm{DFT}_m^* = \left(\mathrm{DFT}_p^* \otimes I_{[m']}\right) \cdot T_m^* \cdot (I_p \otimes \mathrm{DFT}_{m'}^*) \cdot (L_{m'}^m)^*$$

and

$$\mathrm{CRT}_m^* = \left(\mathrm{CRT}_p^* \otimes I_{[m']}\right) \cdot \hat{T}_m^* \cdot \left(I_{\mathbb{Z}_p^*} \otimes \mathrm{DFT}_{m'}^*\right) \cdot \left(L_{m'}^{\varphi(m)}\right)^*.$$

First, we observe that DFT_m is a symmetric matrix for all m, and thus $\mathrm{DFT}_m^* = \overline{\mathrm{DFT}_m}$. Furthermore, the twiddle matrices T_m and $\hat{T}_m$ are diagonal matrices and the transposition in the conjugate transposes can be skipped. Finally, $L_{m'}^m$ and $L_{m'}^{\varphi(m)}$ are permutation matrices, so complex conjugation changes nothing and the transpose equals the inverse. □

In the same way as above, we get decompositions of $(\mathrm{DFT}_m^*)^{-1}$ and $(\mathrm{CRT}_m^*)^{-1}$ from Equations (2.4) and (2.5)

$$(\mathrm{DFT}_m^*)^{-1} = L_{m'}^m \cdot \left(I_p \otimes \left(\overline{\mathrm{DFT}_{m'}}\right)^{-1}\right) \cdot \hat{T}_m \cdot \left(\left(\mathrm{DFT}_p^*\right)^{-1} \otimes I_{[m']}\right), \tag{2.11}$$

$$(\mathrm{CRT}_m^*)^{-1} = L_{m'}^{\varphi(m)} \cdot \left(I_{\mathbb{Z}_p^*} \otimes \left(\overline{\mathrm{DFT}_{m'}}\right)^{-1}\right) \cdot \hat{T}_m \cdot \left(\left(\mathrm{CRT}_p^*\right)^{-1} \otimes I_{[m']}\right). \tag{2.12}$$

Similar to CRT_m we will always apply the matrices CRT_m^* and $(\mathrm{CRT}_m^*)^{-1}$ in a way such that the stride permutations cancel each other out. Therefore we omit them completely in our implementation.

2.3.5 Sampling Gaussians in the Decoding Basis

This section deals with the sampling of continuous Gaussians over $K_\mathbb{R} = K \otimes \mathbb{R}$ represented in the decoding basis. We will show how the somewhat efficient way of sampling via the canonical embedding σ can be further optimized under certain circumstances. Our goal is to achieve a real coordinate vector $\mathbf{a}$ of some Gaussian distributed $a \in K_\mathbb{R}$ such that $a = \left\langle \vec{d}, \mathbf{a} \right\rangle$. By Proposition 2.3.6 this can be done by sampling $\sigma(a)$ from the n-dimensional Gaussian distribution over H and then left multiply by CRT_m^*. The latter is done somewhat efficiently using the sparse-decomposition of CRT_m^* from Proposition 2.3.8. Further, recall from Remark 1.4.8 that H has the unitary basis matrix

$$B = \frac{1}{\sqrt{2}} \begin{pmatrix} I & \sqrt{-1}J \\ J & -\sqrt{-1}I \end{pmatrix} \in \mathbb{C}^{\mathbb{Z}_m^* \times [\varphi(m)]},$$

which implies that the sample $\sigma(a)$ is obtained by sampling n independent *real* Gaussians used as coefficients for the basis B.

Definition 2.3.9. For a positive integer m define the *radical* $\mathrm{rad}(m)$ of m as the product of all primes dividing m.

The above procedure can be optimized when $rad(m) \ll m$, which is the case for basically all m of interest. Multiplication by CRT_m^* via the sparse decomposition from Proposition 2.3.8 results in multiplication with a DFT followed by a twiddle matrix and then a CRT. Noticing that the first two matrices are scaled unitary matrices and that Gaussian distributions are invariant under unitary transformations let us observe that we can effectively skip those multiplications. Since the leftover application of the CRT matrix is in general of much lower dimension, this is significantly faster. The following proposition provides more details.

Proposition 2.3.10. *Let m be a positive integer and $m = \prod_{l=0}^{s} m_l$ be the prime power factorization of m, where each m_l is the power of some prime p_l. Let $K = \mathbb{Q}(\zeta_m)$ be the m-th*

cyclotomic number field, $R \subset K$ the ring of integers in K and $R^\vee$ its dual ideal. Define the matrices

$$C^* := \bigotimes_{l=0}^{s} \left(\mathrm{CRT}^*_{p_l} \otimes I_{[m'_l]} \right)$$

and

$$B' = \bigotimes_{l=0}^{s} \left(B'_{p_l} \otimes I_{[m'_l]} \right),$$

where B'_{p_l} is the $[\varphi(p)] \times [\varphi(p)]$ matrix

$$B'_{p_l} = \frac{1}{\sqrt{2}} \begin{pmatrix} I & \sqrt{-1}J \\ J & -\sqrt{-1}I \end{pmatrix}$$

and $m'_l = m_l/p_l$ [In the case $p_l = 2$ we define $B'_{p_l} := 1$]. Here I denotes the $\varphi(p_l)/2$-dimensional identity matrix and J the $\varphi(p_l)/2$-dimensional reverse identity matrix. Then, in order to sample a continuous Gaussian of standard deviation r over $K_\mathbb{R} = K \otimes \mathbb{R}$ in the decoding basis of $R^\vee$, it is sufficient to sample $n = \varphi(m)$ independent real *Gaussians of standard deviation $r\sqrt{m/\mathrm{rad}(m)}$ in a vector $\mathbf{c}$ and left multiply $\mathbf{c}$ by C^*B'. The resulting vector is the coordinate vector of the desired Gaussian in the decoding basis.*

Proof. Recall the sparse decomposition of $\mathrm{CRT}^*_{m_l}$,

$$\mathrm{CRT}^*_{m_l} = \left(\mathrm{CRT}^*_{p_l} \otimes I_{[m'_l]} \right) \cdot \hat{T}^*_{m_l} \cdot \left(I_{\mathbb{Z}^*_{p_l}} \otimes \mathrm{DFT}^*_{m'_l} \right).$$

The twiddle matrix $\hat{T}^*_{m_l}$ and the scaled discrete Fourier transform $1/\sqrt{m'_l} \cdot \mathrm{DFT}^*_{m'_l}$ are both unitary matrices. Therefore, we can combine them in the sparse decomposition to some unitary Q_l and write

$$\mathrm{CRT}^*_{m_l} = \left(\mathrm{CRT}^*_{p_l} \otimes I_{[m'_l]} \right) \cdot \sqrt{m'_l} \cdot Q_l.$$

Using the mixed-product property of the Kronecker product yields

$$\mathrm{CRT}^*_m = \bigotimes_{l=0}^{s} \mathrm{CRT}^*_{m_l} = \bigotimes_{l=0}^{s} (\mathrm{CRT}^*_{p_l} \otimes I_{[m'_l]}) \cdot \sqrt{\frac{m}{\mathrm{rad}(m)}} \bigotimes_{l=0}^{s} Q_l.$$

Now, $Q = \bigotimes_{l=0}^{s} Q_l$ is unitary and therefore sends a Gaussian distribution over $H \subset \mathbb{C}^{\mathbb{Z}^*_m}$ to a Gaussian distribution with the same standard deviation over $H' = QH \subset \mathbb{C}^{\mathbb{Z}^*_m}$. Hence, in order to produce a continuous Gaussian of standard deviation r over H, it suffices to sample a Gaussian of standard deviation $r\sqrt{m/\mathrm{rad}(m)}$ over H' and left multiply the result by

$$C^* = \bigotimes_{l} \left(\mathrm{CRT}^*_{p_l} \otimes I_{[m'_l]} \right) = \mathrm{CRT}^*_{\mathrm{rad}(m)} \otimes I_{[m/\mathrm{rad}(m)]}.$$

It remains to show that $B'\mathbf{c}$ is a Gaussian of standard deviation $r\sqrt{m/\mathrm{rad}(m)}$ over H'. To do so, it is sufficient to show that B' is a unitary basis matrix for H'. From Proposition 2.3.6 and the fact that H is an n-dimensional real vector space it follows that $\mathrm{CRT}^*_m H$ is isomorphic to $\mathbb{R}^{[n]}$ and hence $C^* \cdot H' = C^* \cdot QH$ is also isomorphic to $\mathbb{R}^{[n]}$, since the difference between

$\mathrm{CRT}_m^* H$ and $C^* \cdot QH$ is only the factor $\sqrt{m/\mathrm{rad}(m)} \in \mathbb{R}$. Therefore, we can characterize H' as

$$H' = \left\{ x \in \mathbb{C}^{\mathbb{Z}_m^*} \,\middle|\, C^* x \in \mathbb{R}^{[n]} \right\}.$$

Now, it suffices to show that B' is a unitary matrix such that $C^* B'$ is real, in order to proof that B' is a unitary basis matrix of H'.

Clearly B' is unitary. By the mixed-product property it is sufficient to check that the product $\mathrm{CRT}_{p_l}^* \cdot B'_{p_l}$ is real, in order to prove that $C^* \cdot B'$ is real. Letting $D_{p_l} = \mathrm{CRT}_{p_l}^* \cdot B'_{p_l}$ we can compute the entries of D_{p_l} using the somewhat symmetric structure of B'_{p_l}. By definitions of $\mathrm{CRT}_{p_l}^*$ and B'_{p_l} we have

$$D_{p_l}(j_0, j_1) = \frac{1}{\sqrt{2}} \left(\overline{\omega_{p_l}}^{j_0 \cdot j_1} + \overline{\omega_{p_l}}^{j_0 \cdot (p_l - j_1)} \right) = \frac{1}{\sqrt{2}} \left(\overline{\omega_{p_l}}^{j_0 \cdot j_1} + \omega_{p_l}^{j_0 \cdot j_1} \right) = \sqrt{2} \cdot \mathrm{Re}(\omega_{p_l}^{j_0 \cdot j_1})$$

and

$$D_{p_l}(j_0, \varphi(p_l) - j_1) = \frac{1}{\sqrt{2}} \left(i\overline{\omega_{p_l}}^{j_0 \cdot j_1} - i\overline{\omega_{p_l}}^{j_0 \cdot (p_l - j_1)} \right) = \frac{1}{\sqrt{2}} \left(i\overline{\omega_{p_l}}^{j_0 \cdot j_1} - i\omega_{p_l}^{j_0 \cdot j_1} \right) = \sqrt{2} \cdot \mathrm{Im}(\omega_{p_l}^{j_0 \cdot j_1}),$$

where $i = \sqrt{-1}$ is the imaginary unit, $j_0 \in [\varphi(p_l)]$ and $1 \leq j_1 \leq \varphi(p_l)/2$. Note that we used the fact that $\overline{\omega_p}^{p-j} = \overline{\omega_p}^{-j} = \omega_p^j$ for any prime p and integer j. Clearly, both computed values are in $\mathbb{R}$ implying that $C^* B'$ is a real matrix. □

2.3.6 Decoding $R^\vee$ and its Powers

In this section we want to adopt the decoding algorithm from Section 1.3.1 to our general ring-LWE setting. Recall that the goal of the decoding algorithm is to recover a sufficiently short element $x \in V$, given $x \mod \Lambda$, where V is some vector space and Λ a lattice in V. The algorithm uses short linearly independent vectors in the dual lattice $\Lambda^\vee$ to recover x. Now, for our applications we have to decode elements from the quotient $K/R^\vee$. To do so, we use the decoding basis $\vec{d}$ of $R^\vee$, whose dual basis in $(R^\vee)^\vee = R$ is the conjugate powerful basis $\tau(\vec{p})$. By Remark 1.3.5 the length of elements we can successfully decode, depends inversely on the maximum length of the dual basis elements. Therefore, we would like to have dual basis elements that are as short as possible. Since τ maps roots of unity again to roots of unity, the basis elements in the conjugate powerful basis have the same length as the powerful basis elements, i.e., $\|\tau(p_j)\|_2 = \|p_j\|_2$. By Observation 2.2.13 the latter norm is $\|p_j\|_2 = \sqrt{n}$. Further, by Lemma 1.4.17 and the fact that $\mathrm{N}(R) = 1$, we get that every nonzero element in R has length at least $\sqrt{n}$. This implies that the decoding basis is a optimal choice, since there cannot exist any set of shorter dual elements.

Unfortunately, in the more general case, when we decode $K/\mathcal{I}_k$ for some power $\mathcal{I}_k = (R^\vee)^k = \langle t^{-k} \rangle$, $k \geq 1$, this is not the case any more. Here the "decoding" basis would be $t^{1-k}\vec{d}$ and we could perform the rounding off algorithm in this basis. But, as it turns out, the elements of the dual basis $t^{k-1}\tau(\vec{p})$ might be much longer than the shortest nonzero elements in $\mathcal{I}_k^\vee = \langle t^{k-1} \rangle$. This is why we use the *scaled decoding basis* $\hat{m}^{1-k}\vec{d}$, where we recall from Definition 1.4.21 that $\hat{m} = t \cdot g$. Also, keeping in mind that $R^\vee = t^{-1}R$, the scaled decoding basis generates the superideal $\mathcal{J}_k = \hat{m}^{1-k} R^\vee = g^{1-k} t^{-k} R \supseteq \mathcal{I}_k$ and has the dual elements $\hat{m}^{k-1}\tau(\vec{p}) \in \mathcal{I}_k^\vee$ of length $\hat{m}^{k-1}\sqrt{n}$. The scaled decoding basis is still not optimal, but nearly so. Lemma 1.4.17 together with Equation (1.6) imply that the minimum distance

of $\mathcal{I}_k^\vee = (R^\vee)^{1-k}$ is at least

$$\sqrt{n} \cdot \mathrm{N}^{(1-k)/n}(R^\vee) = \sqrt{n} \cdot \Delta_K^{(k-1)/n}.$$

Then, using Equation (1.4), we can estimate the ratio between the basis elements and the shortest nonzero vector as

$$\frac{\|\hat{m}^{k-1}\tau(\vec{p})\|_2}{\lambda_1(\mathcal{I}_k^\vee)} = \frac{\hat{m}^{k-1}\sqrt{n}}{\lambda_1(\mathcal{I}_k^\vee)} \leq \frac{\hat{m}^{k-1}\sqrt{n}}{\sqrt{n} \cdot \Delta_K^{(k-1)/n}} = \left(\prod_{\text{odd prime } p \mid m} p^{1/(p-1)} \right)^{k-1},$$

which is for almost all relevant choices of m and k quite small. For example, if we take all odd primes up to 17, the term inside the parenthesis is ≈ 6.73. The product of these primes is 255255, so this corresponds already to choices of $m \geq 255255$.

In the following we consider a "scaled up and discretized" version of the rounding off algorithm, due to reasons of convenience in applications and implementations. In this version we decode from $\mathcal{I}_{k,q}$ to $\mathcal{I}_k$ for some integer $q \geq 1$. That is, the unknown vector in $\mathcal{I}_k$ is given modulo $q\mathcal{I}_k$, and the output is expected to be in $\mathcal{I}_k$. For $k \geq 2$, the scaled decoding basis $\hat{m}^{1-k}\vec{d}$ may generate a strict superideal $\mathcal{J}_k \supset \mathcal{I}_k$ and the decoded element might be in $\mathcal{J}_k$ but not in $\mathcal{I}_k$. In this case we define the output as undefined. But as long as the unknown element in $\mathcal{I}_k$ is short enough, it will be decoded correctly.

Notation 2.3.11. For any $a \in \mathbb{Z}_q$, let $[\![a]\!]$ denote the unique representative $a' \in a + q\mathbb{Z} \cap [-q/2, q/2)$. Further, for vectors $\mathbf{a} \in \mathbb{Z}_q^n$ we extend $[\![\cdot]\!]$ component-wise such that $[\![\mathbf{a}]\!] \in [-q/2, q/2)^n$.

Definition 2.3.12. Let $\mathcal{I}_k = (R^\vee)^k$ for some $k \geq 1$. Then we define the *decoding function* $[\![\cdot]\!] : \mathcal{I}_{k,q} \to \mathcal{I}_k$ as follows. For any element $a \in \mathcal{I}_{k,q}$, write $a = \left\langle \hat{m}^{1-k}\vec{d}, \mathbf{a} \right\rangle \mod q\mathcal{J}_k$ for some coordinate vector $\mathbf{a}$ over $\mathbb{Z}_q$, where $\mathcal{J}_k = \hat{m}^{1-k}R^\vee \supseteq \mathcal{I}_k$. Then, define $[\![a]\!] := \left\langle \hat{m}^{1-k}\vec{d}, [\![\mathbf{a}]\!] \right\rangle$ if this value is in $\mathcal{I}_k$, otherwise $[\![a]\!]$ is undefined.

Lemma 2.3.13. *Let $\mathcal{I}_k = (R^\vee)^k$ for some $k \geq 1$ and $a \in \mathcal{I}_k$ some element. Write $a = \left\langle \hat{m}^{1-k}\vec{d}, \mathbf{a} \right\rangle$ for some integral coordinate vector $\mathbf{a}$. Further let $q \geq 1$ be some integer. Then the following holds:*

(i) *If every coefficient a_j is in the range $[-q/2, q/2)$, then $[\![a \bmod q\mathcal{I}_k]\!] = a$.*

(ii) *For every coefficient a_j we have $|a_j| \leq \hat{m}^{k-1}\sqrt{n} \cdot \|a\|_2$.*

Proof. The first claim is just a reformulation of Claim 1.3.4 for the ring-LWE setting. The second claim follows from Remark 1.3.5 and the fact that $\|\hat{m}^{k-1}\tau(\vec{p})\|_2 = \hat{m}^{k-1}\sqrt{n}$. □

In Section 3.4 we describe how this procedure can be implemented efficiently. For further analysis of this topic we refer to [LPR13b].

2.4 Discretization in $\mathbb{Q}(\zeta_m)$

An important step in our cryptosystems from Section 2.1 is the conversion of a continuous Gaussian into a discrete Gaussian-like distribution. In our usual setting, where $K = \mathbb{Q}(\zeta_m)$ is

the m-th cyclotomic number field for some positive integer m and $R \subset K$ is its ring of integers, this means that we have to discretize a certain point $x \in K_{\mathbb{R}} = K \otimes \mathbb{R}$ to a point $y \in c + pR^\vee$ where p is an integer scaling factor and $R^\vee$ is the dual ideal of R. In Section 1.3.2 we described the discretization problem for general lattices in a vector space. Via the canonical embedding σ we could convert the discretization problem in $K_{\mathbb{R}}$ to a discretization problem in the vector space H. However, the discretization algorithm works only on the coordinate vectors of the involved elements. The coordinate vector of an element $a \in R^\vee$ with respect to a basis $\vec{b}$ is the same as for the element $\sigma(a) \in \sigma(R^\vee) \subset H$ with respect to the basis $\sigma(\vec{b})$. Consequently, we can view all concerned elements directly in $K_{\mathbb{R}}$ when performing the algorithm.

To begin with, we take a closer look at the special case mentioned at the beginning of this section, namely discretizing a point $x \in K_{\mathbb{R}}$ to a coset $c + pR^\vee$. Let $n = \varphi(m)$. Suppose the points $x \in K_{\mathbb{R}}$ and $c \in R^\vee$ are given by some coordinate vectors $\mathbf{x} \in \mathbb{R}^{[n]}$ and $\mathbf{c} \in \mathbb{Z}^{[n]}$. We assume that both points are represented in the decoding basis $\vec{d}$. The basis for our lattice $pR^\vee$ is then given by $p\vec{d}$. Thus, we have to represent $c' = c - x \mod pR^\vee$ in the basis $p\vec{d}$. Since our points are given in the decoding basis, this is simply done by cutting of the integer parts of each coordinate in $\frac{1}{p}(\mathbf{c} - \mathbf{x})$. Indeed, we have

$$c - x = \left\langle \vec{d}, \mathbf{c} - \mathbf{x} \right\rangle = \left\langle p\vec{d}, \frac{1}{p}(\mathbf{c} - \mathbf{x}) \right\rangle,$$

where $\frac{1}{p}(\mathbf{c} - \mathbf{x}) \in \mathbb{R}^n$. Cutting of the integer parts leads to a vector in $[0,1)^{[n]}$. Let $\mathbf{z} = (z_i) \in [0,1)^{[n]}$ be this resulting vector. Then, we have to choose randomly and independently values f_i from $\{z_i - 1, z_i\}$ where $P[f_i = z_i] = 1 - z_i$. We have $\mathbf{f} = (f_i) \in (-1,1)^{[n]}$ and by construction $f := \left\langle p\vec{d}, \mathbf{f} \right\rangle \in c' + pR^\vee$. Now, the output of the algorithm is $y := x + f$, where $y \in R^\vee$ and $y + pR^\vee = c + pR^\vee$. To validate the latter two conditions, we have to check two properties. On the one hand, y needs to have an integer coordinate vector with respect to the decoding basis and, on the other hand, $\frac{1}{p}(\mathbf{y} - \mathbf{c})$ has to be an integral vector.

Let $i \in [n]$ be an arbitrary index. We have to distinguish two cases for the build-up of the entry y_i in the coordinate vector of y. Either $f_i = z_i$ or $f_i = z_i - 1$. Note that we compute in the decoding basis, so the coordinates of f are actually $p\mathbf{f}$. Let n_i denote the integer such that $\frac{1}{p}(c_i - x_i) - n_i \in [0,1)$. Then, in the first case we have

$$y_i = x_i + pf_i = x_i + pz_i = x_i + p\left(\frac{1}{p}(c_i - x_i) - n_i\right) = c_i - pn_i \in \mathbb{Z},$$

and in the second case we have

$$y_i = x_i + pf_i = x_i + p(z_i - 1) = x_i + pz_i - p = c_i - pn_i - p \in \mathbb{Z}.$$

The above computation shows both properties. Clearly, y has an integer coordinate vector. Furthermore, either $\frac{1}{p}(y_i - c_i) = -n_i \in \mathbb{Z}$ or $\frac{1}{p}(y_i - c_i) = -n_i - 1 \in \mathbb{Z}$, which implies that $\frac{1}{p}(\mathbf{y} - \mathbf{c})$ is an integral vector. So y is indeed our desired element.

In a more general setting, let $\vec{b}$ be a $\mathbb{Q}$-basis of K and say we want to discretize a point $x \in K_{\mathbb{R}}$ to a coset $c + p\left\langle \vec{b} \right\rangle$, where $c \in H$. Note that by allowing real coefficients in the coordinate vectors, $\vec{b}$ turns into an $\mathbb{R}$-basis of $K_{\mathbb{R}}$. Therefore, if the elements x and c are given with respect to $\vec{b}$, we can proceed as in the special case described above. Moreover, in our applications, the coset representative c will actually be an element in $\left\langle \vec{b} \right\rangle$ itself. Hence, it is sufficient to represent c by an integral coordinate vector as it is done above.

3 Further Implementation Notes for some Features of the Toolkit

This chapter gives some further implementation notes of some central features of the toolkit. A majority of the operations in the toolkit rely on the transformation of certain coordinate vectors. These transformations are given as matrices, which are decomposed via the Kronecker product. We can take advantage of this decomposition to provide a more efficient way of multiplying the matrices with the coordinate vectors. This algorithm is developed in Section 3.1. In particular, the sparse decomposition of DFT and CRT from Sections 2.2.4 and 2.3.4 use this algorithm. This implies that in the end only the prime-indexed transformations will be applied to a suitable subvector. Section 3.2 shows how this can be done via FFT algorithms, providing $O(n \log n)$ algorithms instead of $O(n^2)$. Subsequently, Section 3.3 explains in more detail, compared to Chapter 2, how addition and multiplication in R and $R^\vee$ can be implemented. Next, in Section 3.4 we describe an algorithm that efficiently decodes elements in $R^\vee$, if possible. Finally, Section 3.5 deals with some pre-computations that can be done for better performance. In particular, we explain how the element $g \in R$ and the neutral element in the different bases can be pre-computed.

Throughout this chapter, if not stated differently, let $K = \mathbb{Q}(\zeta_m)$ be the m-th cyclotomic number field for some positive integer m and $R \subset K$ its ring of integers. $R^\vee$ is the dual ideal of R as defined in Section 1.4.4. The dimension of K is $n = \varphi(m)$. Further let $K_\mathbb{R} = K \otimes \mathbb{R}$ be the tensor product of K and $\mathbb{R}$, whose image under the canonical embedding σ is the vector space H (cf. Definition 1.4.7).

3.1 Multiplication with Kronecker Decomposed Matrices

A common task in all our applications and also crucial for the efficiency is the multiplication of certain square matrices with vectors. Thereby, the matrices are decomposed via the Kronecker product. That is, for a square matrix A of dimension m we have $A = \bigotimes_{l=0}^{s} A_l$ where A_l are suitable square matrices with smaller dimensions m_l. Recall the *mixed-product property* for the Kronecker product: $(A \otimes B)(C \otimes D) = (AC) \otimes (BD)$. Using this property we can rewrite the decomposition as

$$A = \prod_{l=0}^{s} (I_{m_1} \otimes \ldots \otimes I_{m_{l-1}} \otimes A_l \otimes I_{m_{l+1}} \otimes \ldots \otimes I_{m_s}),$$

where I_{m_l} means the identity matrix of dimension m_l. Since the Kronecker product of two identity matrices is again an identity matrix, namely $I_{m_l} \otimes I_{m_{l'}} = I_{m_l m_{l'}}$, we can combine the identity matrices on the left and the right side into a bigger one respectively,

$$A = \prod_{l=0}^{s} (I_{m_1 \cdots m_{l-1}} \otimes A_l \otimes I_{m_{l+1} \cdots m_s}).$$

To see how we can use this to our advantage we need to take a closer look at the structure of each factor.

Let n_0 and n_1 be arbitrary positive integers. Applying the n_0-dimensional identity matrix via the Kronecker product from the left side to A_l leads to a diagonal block matrix with n_0 diagonal blocks A_l. A further application of the n_1-dimensional identity matrix via the Kronecker product from the right side scatters the entries of each block A_l in an $n_1 \times n_1$ pattern, leading to a sparse matrix. We illustrate this on an example.

Example 3.1.1. Given the matrix

$$A = \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix},$$

we compute $I_2 \otimes A \otimes I_2$. By definition of the Kronecker product we have that

$$I_2 \otimes A \otimes I_2 = \begin{pmatrix} 2 & 2 & 0 & 0 \\ 2 & 2 & 0 & 0 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 2 & 2 \end{pmatrix} \otimes I_2 = \begin{pmatrix} 2 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 2 & 0 & 0 & 0 & 0 \\ 2 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 & 2 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 2 \end{pmatrix}.$$

Let us analyze this example. If we want to apply this matrix to a vector of dimension 8, the only thing we have to do is to apply the matrix A to the subvectors indexed by $(0, 2), (1, 3), (4, 6)$ and $(5, 7)$. So instead of a matrix multiplication with $I_2 \otimes A \otimes I_2$ we have only four multiplications with A which can be parallelized. This saves real performance on modern architectures. Back in the general setting this means that the application of a matrix $A = \bigotimes_{l=0}^{s} A_l$ reduces to m/m_l parallel multiplications by A_l, in sequence for each m_l.

Having this insight, we can construct an algorithm for multiplication with a Kronecker decomposed matrix. The input will be a list, or any other suitable container, of matrices A_l of dimensions m_l representing $A = \bigotimes_{l=0}^{s} A_l$ and a vector x of suitable dimension. Then the algorithm outputs $y = A \cdot x$. To do so, we simply traverse through the list and for each A_l do the m/m_l multiplications and update the resulting vector. Notice that the order in which we apply the matrices A_l does not matter, since $(B \otimes I_n)(I_{n'} \otimes C) = B \otimes C = (I_{n'} \otimes C)(B \otimes I_n)$ for appropriate B, C, n and n'. But, as we assume an ordered list anyway, it makes sense to traverse linear through the list.

Let k be a fixed position in the list, so the algorithm has already made $k - 1$ loops. Let x_{k-1} be the updated input vector. Define $n_0 := \prod_{l=0}^{k-1} m_l$ and $n_1 := \prod_{l=k+1}^{s} m_l$ (for $k = 1$ or $k = s$ set $n_0 = 1$ or $n_1 = 1$ respectively). Then we want to compute $(I_{n_0} \otimes A_l \otimes I_{n_1}) \cdot x_{l-1}$. To do so we have to compute the correct indexes to build subvectors of dimension m_l. This is best implemented via a subroutine, because often we also multiply vectors with matrices of the form $I \otimes A$ or $A \otimes I$, which then can be realized directly through this subroutine.

Let us again take a look at Example 3.1.1, in particular at the structure of $I_2 \otimes A \otimes I_2$. We have a diagonal block-matrix with 2 blocks of size $4 = 8/2$. The number and size of the blocks only depend on the dimension of unity matrix which is applied from the left side. Generally speaking, this means that we have n_0 blocks of size m/n_0. So, once we know how to compute the indexes inside of one block we can derive all indexes through shifting by m/n_0.

Each block in our example is of the form $A \otimes I_2$, so the entries of A are scattered in a 2×2 pattern. We are only interested in the first two rows, because we just need to compute two subvectors and the rest is repetition. The indexes in the first row where we have non-zero entries are given by $0 \cdot 2 + 1$ and $1 \cdot 2 + 1$ and in the second row they are only shifted by 1. In the general setting this translates to an $n_1 \times n_1$ scattered matrix $A_l \otimes I_{n_1}$ and we have to compute n_1 different subvectors $z_1, \ldots, z_{n_1}$, where the indexes of a subvector z_i are given by $0 \cdot n_1 + i, \ldots, (m-1) \cdot n_1 + i$. So, in each step the indexes are shifted by 1, n times in total. Now we can multiply A_l with each subvector and the updated vector x_l is just the merging of all resulting subvectors.

Algorithm 3.1.2. Let $A = \bigotimes_{l=0}^{s} A_l$ be a Kronecker decomposed square matrix of dimension $m = \prod_{l=0}^{s} m_l$, where $\dim(A_l) = m_l$. Then the following Algorithm provides an efficient way of multiplying A with a vector x of dimension m.

ApplyKronDecomp: Efficiently compute $y = (\bigotimes_{l=0}^{s} A_l) \cdot x$

Require: A list $A_0, \ldots, A_s$ and an m dimensional vector x.
Ensure: $y = (\bigotimes_{l=0}^{s} A_l) \cdot x$.

```
1: y := x {initiate resulting vector}
2: for all A_l do
3:     n_0 := ∏_{i=0}^{l-1} m_i
4:     n_1 := ∏_{j=l+1}^{s} m_j
5:     y = ApplySingleKronDecomp(A_l, n_0, n_1, y)
6: end for
7: return y
```

The subroutine ApplySingleKronDecomp(A_l, n_0, n_1, x) is given by

---

ApplySingleKronDecomp: Efficiently compute $y = (I_{n_0} \otimes A_l \otimes I_{n_1}) \cdot x$

Require: A square matrix A_l of size m_l, dimensions n_0, n_1, such that $m = n_0 \cdot m_l \cdot n_1$, and a vector x of dimension m.
Ensure: $y = (I_{n_0} \otimes A_l \otimes I_{n_1}) \cdot x$.

```
1: y := 0 {m dimensional vector}
2: y', x' := 0 {m_l dimensional vectors}
3: for i = 0 to m − 1 step (m/n_0) do
4:     for j = 0 to n_1 − 1 do
5:         for k = 0 to m_l − 1 do
6:             x'_k ← x_{k·n_1+j+i}
7:         end for
8:         y' ← A_l · x'
9:         for k = 0 to m_l − 1 do
10:            y_{k·n_1+j+i} ← y'_k
11:        end for
12:    end for
13: end for
```

Remark 3.1.3. Notice that, for the sake of simplicity, we do not compute all subvectors first, multiply them with A_l and then update the resulting vector. Instead we make these

steps ad hoc by first computing one subvector, multiplying it with A_l and finally updating the resulting vector at the corresponding indexes.

3.2 Efficient Application of DFT and CRT in the Prime Case

Recall from Section 2.2.4 that we decomposed the discrete Fourier transform DFT_m and the Chinese remainder transform CRT_m from Definition 2.2.8 such that, combined with the algorithms for the application of Kronecker decompositions from Section 3.1, an application of DFT_m or CRT_m only depends on DFT_p and CRT_p for prime divisor p of m, and some twiddle matrices. While all twiddle matrices are diagonal and can be applied in linear time, DFT_p and CRT_p are dense matrices using quadratic time for an application to a vector. However, the special form of DFT_p and CRT_p allows us to optimize the runtime of an application to $O(p \log p)$.

In several fields of research, especially in signal processing, the discrete Fourier transform for a complex sequence $(x_0, \ldots, x_{N-1})$ of length $N \in \mathbb{N}$ is usually defined as

$$X_k = \sum_{j=0}^{N-1} x_j \omega_N^{jk} \quad \text{for } 0 \leq k \leq N-1, \tag{3.1}$$

where $\omega_N = \exp(2\pi i/N) \in \mathbb{C}$. If we view the resulting values X_k in a vector $(X_0, \ldots, X_{N-1})$, the above equations correspond to

$$\begin{pmatrix} X_0 \\ \vdots \\ X_{N-1} \end{pmatrix} = \mathrm{DFT}_N \cdot \begin{pmatrix} x_0 \\ \vdots \\ x_{N-1} \end{pmatrix}.$$

In particular, we can view an application of DFT_p as a discrete Fourier transform of prime length in the sense of Equation (3.1).

In general, an algorithm that efficiently performs a discrete Fourier transformation on an input $(x_0, \ldots, x_{N-1})$ is called a *Fast Fourier Transformation* (FFT). There are several FFT algorithms we could use for an application of DFT_p, for example Bluestein, Winograd or Rader FFT. All of these algorithms are suited for inputs of prime length p and need $O(p \log p)$ time. Detailed explanations and analysis of these and more algorithms can be found in [Nus82]. For our implementation we chose to use Rader's FFT algorithm. It uses a Cooley-Tukey FFT for power of two input length as a subroutine. In Section 4.2 we briefly describe our implementation of these FFT algorithms.

Recall from Definition 2.2.8 that CRT_p is a submatrix of DFT_p. In particular, since p is a prime number, CRT_p is the submatrix of DFT_p where the first row and the last column are removed. Therefore it is possible, via a small adjustment in the algorithm, to apply CRT_p also via Rader's FFT algorithm as we will see in Section 4.2.

The inverse discrete Fourier transformation for an input $(X_0, \ldots, X_{N-1})$ of length N is defined as

$$x_k = \frac{1}{N} \sum_{j=0}^{N-1} X_j \omega_N^{-jk} \quad \text{for } 0 \leq k \leq N-1$$

and corresponds to an application of DFT_N^{-1}. An application of $\mathrm{DFT}_N^* = \overline{\mathrm{DFT}_N}$ corresponds to the equations

$$X_k = \sum_{j=0}^{N-1} x_j \overline{\omega_N}^{jk} \quad \text{for } 0 \leq k \leq N-1.$$

Finally, an application of $(\mathrm{DFT}_m^*)^{-1}$ corresponds to

$$x_k = \frac{1}{N} \sum_{j=0}^{N-1} X_j \overline{\omega_N}^{-jk} \quad \text{for } 0 \leq k \leq N-1.$$

Note that for the primitive root of unity ω_N we have that $\overline{\omega_N} = \omega_N^{-1}$. Consequently, we can interchange these terms in the above equations. Using the techniques of Rader's FFT algorithm, we can perform applications of $\mathrm{DFT}_p^{-1}, \mathrm{DFT}_p^*$ and $(\mathrm{DFT}_p^*)^{-1}$ in $O(p \log p)$. Since CRT_p^* is a submatrix of DFT_p^*, we can apply it similar to CRT_p via Rader's algorithm.

Lyubashevsky, Peikert and Regev state in their paper [LPR13b] that CRT_p^{-1} could also be applied in $O(p \log p)$. There is no specific statement that this should be done by an FFT algorithm, but one might suggest so. However, CRT_p^{-1} lacks the nice structure of $\mathrm{DFT}_p^{-1} = 1/p\,\mathrm{DFT}_p^*$ and is in particular not a submatrix of DFT_p^{-1}. Therefore we were not able to apply CRT_p^{-1} in $O(p \log p)$ via Rader's algorithm or any other FFT algorithm. Additionally, we were not yet able to specify the structure of CRT_p^{-1} and therefore could not find any other method for an application than the naive $O(p^2)$ way. The same problematic applies for $(\mathrm{CRT}_m^*)^{-1}$.

3.3 Addition and Multiplication of Ideal Elements

Probably the most important feature of our toolkit is an efficient method for addition and multiplication in the various ideals we consider. Each element in such an ideal is represented by an integral coordinate vector with respect to a specific basis. Depending on the constellation of the elements and the ideals they live in, the actual operation takes place in different ideals. For example, multiplying two elements $a, b \in R$ is done in R, but multiplication of $a \in R$ and $b \in R^\vee$ is done in $R^\vee$. Now, our algorithm only knows the coordinate vector and the basis of each element. Depending on the basis our algorithm has to recognize the different ideals that are involved, adjust the bases correctly and then do the actual operation. Since we only consider three different bases with some optional scaling factors, we only have to distinguish a manageable amount of cases.

3.3.1 Choosing the Right Basis

In our applications we only have to consider a few ideals, namely $R, R_q, (R^\vee)^k$ and $(R_q^\vee)^k$ for $k \geq 1$. Table 3.1 shows which bases we might use for each ideal. Note that we do not use $t^{-(k-1)}\vec{d}$ as a basis for $(R^\vee)^k$ due to some bad properties in the decoding algorithm. See Section 3.4 for details.

Suppose we want to compute $x = a \cdot b$ where a and b are represented in the bases $\vec{b}_1$ and $\vec{b}_2$ respectively. As we consider up to three different bases for the same ideal, there is no unique way of determining a basis in which x is represented. Therefore, we have to choose

Ring	$\mathbb{Z}$-bases	$\mathbb{Z}_q$-bases
R	$\vec{p}$	
R_q		$\vec{p}, \vec{c}$
$R^\vee$	$\vec{d}$	
$R_q^\vee$		$\vec{d}$
$\mathcal{I}^k = (R^\vee)^k$	$t^{-k}\vec{p}$	
$\mathcal{I}_q^k = (R_q^\vee)^k$		$t^{-k}\vec{p}, t^{-k}\vec{c}$

Table 3.1: The considered bases

one appropriately. Thereby, we want to perform as few basis transformations as possible, while trying to avoid computation with high precision reals.

In the following, let $(\vec{b}_1, \vec{b}_2)$ denote that the elements a, b are represented in the bases $\vec{b}_1$ and $\vec{b}_2$ respectively. We now describe which output basis we choose in the different cases and sketch how to compute the coordinate vector $\mathbf{x}$ of x. Thereby, we distinguish between addition and multiplication.

Addition

Adding two elements is in the most cases done by normal vector addition of the coordinate vectors. If the bases b_1 and b_2 are equal, there is nothing else to do. If the elements are in the same ideal, but represented in different bases, we first need to adjust the basis of one element. In the case, where we do not compute modulo q, i.e., in the ideals R and $R^\vee$, the preferred bases are always $\vec{p}$ and $\vec{d}$ respectively. When computing modulo q, $\vec{c}$ and $t^{-k}\vec{c}$ seem the best choices, since these bases have the best properties concerning addition and multiplication (cf. Proposition 2.2.14 and Equation (2.9)).

In some rare cases it might happen that we want to add two elements $a \in R$ and $b \in R^\vee$. This is possible since $R \subset R^\vee$. Unfortunately, we have no efficient way to compute a representation of a in a basis of $R^\vee$. We saw already in Section 2.3.3 that we can evade this problem via the canonical embedding σ. To be more precise, we can compute the coordinate vector $\mathbf{x}$ of ab as

$$\mathbf{x} = \mathrm{CRT}_m^*(\mathrm{CRT}_m\,\mathbf{a} + (\mathrm{CRT}_m^*)^{-1}\mathbf{b}).$$

Multiplication

Here we describe the cases separately, since unlike to addition, they have not much in common. First we consider the cases where our elements are in R and $R^\vee$ so the coordinate vectors are in $\mathbb{Z}^{[n]}$.

i. $(\vec{p}, \vec{p})$: We have $a \in R$ and $b \in R$. Consequently we compute $a \cdot b$ via the embedding σ in the basis $\vec{p}$, i.e.,

$$\mathbf{x} = \mathrm{CRT}_m^{-1}\left(\mathrm{CRT}_m\,\mathbf{a} \odot \mathrm{CRT}_m\,\mathbf{b}\right),$$

and output x with respect to $\vec{p}$.

ii. $(\vec{p}, t^{-k}\vec{p})$: Here we have $a \in R$ and $b \in (R^\vee)^k$. By linearity of the inner product it holds that

$$a \cdot b = \langle \vec{p}, \mathbf{a} \rangle \cdot \left\langle t^{-k}\vec{p}, \mathbf{b} \right\rangle = t^{-k} \cdot \langle \vec{p}, \mathbf{a} \rangle \cdot \langle \vec{p}, \mathbf{b} \rangle .$$

Thus we can compute $\mathbf{x}$ as in (i.) and the output basis is $t^{-k}\vec{p}$.

iii. $(\vec{p}, \vec{d})$: Again we have $a \in R$ and $b \in R^\vee$ and we can multiply via the canonical embedding σ, where a is embedded via CRT_m and b via $(\mathrm{CRT}_m^*)^{-1}$. The product $x = ab \in R^\vee$ will be represented in the decoding basis, thus we have to pull back the element from H via CRT_m^*. Together we get

$$\mathbf{x} = \mathrm{CRT}_m^* \left(\mathrm{CRT}_m \mathbf{a} \odot (\mathrm{CRT}_m^*)^{-1}\mathbf{b}\right),$$

where $\mathbf{a}$ and $\mathbf{b}$ are the coordinate vectors of a and b respectively.

iv. $(t^{-k_1}\vec{p}, t^{-k_2}\vec{p})$: Now we have $a \in \mathcal{I}^{k_1}$ and $b \in \mathcal{I}^{k_2}$. Again by linearity we can compute $\mathbf{x}$ as in (i.) and the output basis then is $t^{-(k_1+k_2)}\vec{p}$, i.e., $x = ab \in \mathcal{I}^{k_1+k_2}$.

v. $(\vec{d}, t^{-k_2}\vec{p})$: If we switch the basis of a to $t^{-1}\vec{p}$ via L_m, the computation is as in (iv.) with $k_1 = 1$.

vi. $(\vec{d}, \vec{d})$: Note that $R^\vee$ is by definition an R-module. In particular, $R^\vee$ is not closed under multiplication, hence $x = ab$ might not be in $R^\vee$ (only products ab, where $a \in R$ and $b \in R^\vee$, will always be in $R^\vee$). However it will be in $\mathcal{I}^2$. So we switch both bases to $t^{-1}\vec{p}$ and proceed as in (iv.) with $k_1 = k_2 = 1$.

Now we consider the cases where the coordinate vectors are in $\mathbb{Z}_q^{[n]}$, i.e., we have elements in R_q or $(R_q^\vee)^k$ for $k \geq 1$. These computations are often even more efficient, since we can avoid to use high precision doubles or floats, but instead compute only with integers. Note that if we have, for example, elements $a \in R$ and $b \in R_q^\vee$, then we have $a \cdot b \in R_q^\vee$. If we take instead of $a \in R$ the coset $a \in R_q$, i.e., reduce a modulo qR, and compute the product $a \cdot b$, the result would be the same. This means that we can always assume both elements to be in R_q, i.e., the coordinate vectors are in $\mathbb{Z}_q^{[n]}$.

vii. $(\vec{p}, \vec{p}), (\vec{p}, \vec{c}), (\vec{c}, \vec{c})$: We have $a, b \in R_q$. The output basis will be $\vec{c}$ in all cases, so we need both coordinate vectors in the basis $\vec{c}$. We can achieve this by the basis transformations $\mathrm{CRT}_{m,q}$ and $\mathrm{CRT}_{m,q}^{-1}$ (see Equation (2.1)). Then we get $\mathbf{x}$ with respect to $\vec{c}$ by coordinate-wise multiplication.

viii. $(\vec{p}, \vec{d}), (\vec{p}, t^{-k}\vec{p}), (\vec{p}, t^{-k}\vec{c}), (\vec{c}, \vec{d}), (\vec{c}, t^{-k}\vec{p}), (\vec{c}, t^{-k}\vec{c})$: Now we have $a \in R_q$ and $b \in (R_q^\vee)^k$ (k might be 1). In all cases we choose $t^{-k}\vec{c}$ as the output basis. Therefore we have to represent a in the basis $\vec{c}$ and b in $t^{-k}\vec{c}$. Again multiplication with $\mathrm{CRT}_{m,q}$ and $\mathrm{CRT}_{m,q}^{-1}$ switches between $\vec{p}$ and $\vec{c}$. Similarly, multiplication with L_m and L_m^{-1} switches between $t^{-1}\vec{p}$ and $\vec{d}$. Linearity allows us to perform all needed basis transformations with these matrices only. Having the elements represented in the right basis, coordinate-wise multiplication leads to $\mathbf{x}$ with respect to $t^{-k}\vec{c}$.

ix. $(\vec{d}, \vec{d}), (\vec{d}, t^{-k_2}\vec{p}), (\vec{d}, t^{-k_2}\vec{c}), (t^{-k_1}\vec{p}, t^{-k_2}\vec{p}), (t^{-k_1}\vec{p}, t^{-k_2}\vec{c}), (t^{-k_1}\vec{c}, t^{-k_2}\vec{c})$: As discussed in Section 2.3.3, multiplication in powers $\mathcal{I}^k$ can also be done coordinate wise in the CRT basis. Thus, letting $k_1 = 1$ in the cases where $b_1 = \vec{d}$, we will output the element $ab \in \mathcal{I}^{k_1+k_2}$ in the basis $t^{-(k_1+k_2)}\vec{c}$. As in (viii.) we transform the coordinate vectors such that a and b are represented in the correct bases, and then multiply them coordinate wise to get $\mathbf{x}$.

Following these standards, an algorithm for addition or multiplication just has to check the bases of a and b and determine in which ideal they live. Having these informations is sufficient to choose one of the above cases and perform the necessary operations.

Using these algorithms, one should always try to avoid arithmetic via the canonical embedding. Not only are arithmetic operations over $\mathbb{Z}$ and $\mathbb{Z}_q$ much faster, there is also a subtlety with precision of the computed elements. Note that the transformation CRT_m consists of complex roots of unity. A common representation for complex values $z \in \mathbb{C}$ are pairs $(a, b) \in \mathbb{R}^2$ such that $z = a + ib$. In the case of roots of unity, often either a or b will be an irrational number like $\sqrt{2}$ or $\sqrt{3}$. Unfortunately we can only approximate such numbers via rationales given as doubles or floats. Thus, if we embed two integral coordinate vectors via CRT_m, perform some arithmetic operations over H, and finally retrieve the resulting element via CRT_m^*, the resulting vector will contain complex numbers that are only very close to integers. The error we make, i.e., the distance between the resulting vector and the closest integral vector, will increase with the number of arithmetic operations we perform over H. We did not yet run any error analysis on this topic, and therefore do not know exactly how the error behaves and grows. However, one should be aware of this fact and take it into account in an implementation.

3.3.2 Special Multiplication

Recall from Section 1.4.4 that the dual Ideal $R^\vee$ is generated by t^{-1} as defined in Definition 1.4.21. Thus each element $a \in R^\vee$ can be rewritten as $t^{-1}a'$ for some $a' \in R$. Consequently, multiplying a with t results in $a' \in R$. Although R is a subset of $R^\vee$ and a' could also be represent in any $\mathbb{Z}$-basis of $R^\vee$, for example the decoding basis, it is quite handy to represent a' again in the powerful basis. The situation is similar if a is an element in $R_q^\vee$. In this case we would like to represent $a' \in R_q$ in the CRT basis. However, this sort of multiplication is not covered by our normal multiplication function and has to be done separately. Fortunately, it turn out to be a very simple task. If we represent a in $t^{-1}\vec{p}$ or $t^{-1}\vec{c}$, a' has exactly the same coordinates with respect to $\vec{p}$ or $\vec{c}$. Thus, it is sufficient to manually change the basis without any further computations.

A multiplication of some element $a \in R$ with t is not so easy. Here we would need the coordinates of t in the powerful basis. Although we can pre-compute g (see. Section 3.5.1) it is not possible to simply compute $t = \hat{m}/g$, since we have no function for division, or to be more precise, for inversion of elements in K. However, this kind of multiplication is not needed in our applications, which is why our program will lack this feature.

Multiplication with t^{-1} behaves similar as with t. Again for an element $a \in R$, the product $t^{-1}a \in R^\vee$ has the same coordinates, if a is given in the powerful basis $\vec{p}$ and $t^{-1}a$ is represented in $t^{-1}\vec{p}$. Furthermore, this time we have a simple way to compute the coordinates of $t^{-1} \in R^\vee$. We know that $1 \in R$ has the coordinate vector $(1, 0, \ldots, 0)$ in the powerful basis. Consequently t^{-1} has the same coordinate vector in $t^{-1}\vec{p}$ and we can multiply elements in $R^\vee$ with t^{-1} via the normal multiplication function. Clearly it is also possible to switch the bases to the CRT or decoding basis and perform the multiplication there.

3.4 Efficient Decoding

In this section we describe an efficient implementation of the decoding procedure from Section 2.3.6. Recall that we have to recover a short element $a \in \mathcal{I}$ which is given in the form

$a' = a \mod q\mathcal{I}$, where $\mathcal{I} = (R^\vee)^k$ is a power for some $k \geq 1$. The input is given as a coordinate vector $\mathbf{a}'$ over $\mathbb{Z}_q$ such that $a' = \left\langle t^{1-k}\vec{b}, \mathbf{a}' \right\rangle \mod q\mathcal{I}$. Thereby $\vec{b}$ is some of the $\mathbb{Z}_q$-basis of $R_q^\vee$ we consider, i.e., the powerful, CRT or decoding basis. The output is a coordinate vector $\mathbf{a}$ over $\mathbb{Z}$ with respect to the basis $t^{1-k}\vec{d}$ of $\mathcal{I}$.

First, we treat the simpler case where $k = 1$, i.e., we decode $R_q^\vee$. Since we can switch efficiently from both, the powerful and CRT bases, to the decoding basis, we assume that the input coordinate vector is given with respect to $\vec{d}$. Thus, for an input $\mathbf{a}' \in \mathbb{Z}_q^{[n]}$ we have that $R_q^\vee \ni a' = \left\langle \vec{d}, \mathbf{a}' \right\rangle \mod qR^\vee$. To decode this element we define the coordinate vector of the output as $\mathbf{a} = [\![\mathbf{a}']\!]$ (cf. Notation 2.3.11) so that $a = \left\langle \vec{d}, \mathbf{a} \right\rangle \in R^\vee$ is the desired element.

For $k \geq 2$ recall from Section 2.3.6 that we use the scaled decoding basis $\hat{m}^{1-k}\vec{d}$ which generates the superideal $\mathcal{J} = \hat{m}^{1-k}R^\vee \supset \mathcal{I}$. In fact, each scaled basis $\hat{m}^{1-k}\vec{b}$, where $\vec{b}$ is any $\mathbb{Z}$-basis of $R^\vee$, generates this superideal. Assume this time that the input $a' \in \mathcal{I}_q$ is given with respect to the CRT basis. The decoding procedure then consists of three steps:

(i) First compute the representation of $a'' = a' \mod q\mathcal{J}$ in the $\mathbb{Z}_q$-basis $\hat{m}^{1-k}\vec{c}$ of $\mathcal{J}_q$.

(ii) Then decode a'' to an element $a \in \mathcal{J}$ in the scaled decoding basis $\hat{m}^{1-k}\vec{d}$, similarly to the case $k = 1$.

(iii) Finally compute the representation of a in the $\mathbb{Z}$-basis $t^{1-k}\vec{d}$ of $\mathcal{I}$.

For the first step recall from Definition 1.4.21 that $\hat{m} = g \cdot t$ for $g \in R$. In particular, we have $g^{k-1} \in R$. Let $\mathbf{a}'$ be the coordinate vector of the product $g^{k-1}a' \in \mathcal{I}_q$ in the basis $t^{1-k}\vec{c}$, i.e., $g^{k-1}a' = \left\langle t^{1-k}\vec{c}, \mathbf{a}' \right\rangle \mod q\mathcal{I}$. Multiplying both sides with g^{1-k} yields the equivalence

$$g^{k-1}a' = \left\langle t^{1-k}\vec{c}, \mathbf{a}' \right\rangle \mod q\mathcal{I} \Leftrightarrow a' = \left\langle \hat{m}^{1-k}\vec{c}, \mathbf{a}' \right\rangle \mod q\mathcal{J}.$$

Thus we can compute the coordinates of $a'' = a' \mod q\mathcal{J}$ in the basis $\hat{m}^{1-k}\vec{c}$ by multiplication of a' with g^{k-1} in the basis $t^{1-k}\vec{c}$. In Section 3.5.1 we describe how to pre-compute the coordinates of g in the CRT basis and hence also for g^{k-1}, which makes the first step very efficient.

The second step works analogously to the case $k = 1$. First we switch the basis of $a'' \in \mathcal{J}_q$ to the scaled decoding basis. Hence, we have $a'' = \left\langle \hat{m}^{1-k}\vec{d}, \mathbf{a}'' \right\rangle$ for some $\mathbf{a}''$ over $\mathbb{Z}_q$. The decoded element is then given by $a = \left\langle \hat{m}^{1-k}\vec{d}, [\![\mathbf{a}'']\!] \right\rangle \in \mathcal{J}$, which matches the outcome of the decoding function from Definition 2.3.12 if it is in $\mathcal{I}$.

If a is not in $\mathcal{I}$ then the conversion into a representation of a in the basis $t^{1-k}\vec{d}$ is impossible, which indicates decoding failure. Otherwise, the computation in step (iii) is somewhat inverse to step (i). Multiplying a with g^{1-k} yields the coordinate vector of a in $t^{1-k}\vec{d}$ which can be seen by the following equivalence

$$g^{1-k} \cdot a = \left\langle \hat{m}^{1-k}\vec{d}, \mathbf{a} \right\rangle \in \mathcal{J} \Leftrightarrow a = \left\langle t^{1-k}\vec{d}, \mathbf{a} \right\rangle \in \mathcal{I}.$$

As we are now working in the decoding basis, multiplication by g cannot be performed as efficiently as in the first step. At least there is a better way than performing a usual multiplication via the canonical embedding σ, which includes computation with high precision

complex numbers. To be more precise, there is an integral matrix A such that $g \cdot \vec{d}^T = \vec{d}^T \cdot A$, which means that we can multiply an element with g in the decoding basis by applying A to the respective coordinate vector. Similarly we can divide by g using the inverse A^{-1}.

Proposition 3.4.1. *Let $K = \mathbb{Q}(\zeta_m)$ be the m-th cyclotomic number field for some positive integer m with prime power factorization $m = \prod_{l=0}^{s} m_l$, where each m_l is a power of some prime p_l. Further let $R \subset K$ be the ring of integers in K and $R^\vee$ its dual ideal. Denote by $\vec{d}$ the decoding basis of $R^\vee$. For each prime power m_l, let $n_l = \varphi(m_l)$, $m_l' = m_l/p_l$ and define the $[n_l] \times [n_l] \cong ([\varphi(p_l)] \times [m_l']) \times ([\varphi(p_l)] \times [m_l'])$ matrices A by*

$$A_l = \begin{pmatrix} 2 & 1 & 1 & \dots & 1 \\ -1 & 1 & 0 & \cdots & 0 \\ 0 & -1 & 1 & \ddots & \vdots \\ \vdots & \ddots & & \ddots & 0 \\ 0 & \cdots & 0 & -1 & 1 \end{pmatrix} \otimes I_{[m_l']}.$$

Then we have that

$$g \cdot \vec{d}^T = \vec{d}^T \cdot \bigotimes_{l=0}^{s} A_l,$$

where $g \in R$ is defined as in Definition 1.4.21.

Proof. By the tensorial decomposition of $\vec{d}$ and the multilinearity of the tensor product, it suffices to show that

$$g_l \cdot \vec{d_l}^T = \vec{d_l}^T \cdot A_l,$$

where $g_l = 1 - \zeta_{p_l}$ and $\vec{d_l}$ is the decoding basis of $R_l^\vee$, the dual ideal of the ring of integers $R_l \subset K_l = \mathbb{Q}(\zeta_{m_l})$.

For better readability we drop the subscript l, i.e., let m be a prime power of some prime p and $m' = m/p$. Define $n := \varphi(m)$ and recall from the proof of Proposition 2.3.4 that we have the relation

$$d_{(j_0,j_1)} = \frac{1}{m} \cdot (\zeta_p^{j_0} - \zeta_p^{p-1}) \cdot \zeta_m^{j_1},$$

where $j_0 \in [\varphi(p)]$ and $j_1 \in [m']$. Now multiplication of $g = (1 - \zeta_p)$ with $d_{(j_0-j_1)}$ yields

$$\begin{aligned} g \cdot d_{(j_0,j_1)} &= (1 - \zeta_p) \cdot \frac{1}{m} \cdot (\zeta_p^{j_0} - \zeta_p^{p-1}) \cdot \zeta_m^{j_1} \\ &= \frac{1}{m} \cdot \zeta_m^{j_1} \cdot (\zeta_p^{j_0} - \zeta_p^{p-1} - \zeta_p^{j_0+1} + \zeta_p^{p}) \\ &= \frac{1}{m} \cdot \zeta_m^{j_1} \cdot (\zeta_p^{j_0} - \zeta_p^{p-1}) - \frac{1}{m} \cdot \zeta_m^{j_1} \cdot (\zeta_p^{j_0+1} - 1) \\ &= d_{(j_0,j_1)} - \frac{1}{m} \cdot \zeta_m^{j_1} \cdot (\zeta_p^{j_0+1} - 1). \end{aligned}$$

Further computations with the last term lead to

$$\begin{aligned} \frac{1}{m} \cdot \zeta_m^{j_1} \cdot (\zeta_p^{j_0+1} - 1) &= \frac{1}{m} \cdot \zeta_m^{j_1} \cdot (\zeta_p^{j_0+1} - 1 + \zeta_p^{p-1} - \zeta_p^{p-1}) \\ &= \frac{1}{m} \cdot \zeta_m^{j_1} \cdot (\zeta_p^{j_0+1} - \zeta_p^{p-1}) - \frac{1}{m} \cdot \zeta_m^{j_1} \cdot (1 - \zeta_p^{p-1}) \\ &= d_{(j_0+1,j_1)} - d_{(0,j_1)}. \end{aligned}$$

Together we get the equality

$$g \cdot d_{(j_0,j_1)} = d_{(j_0,j_1)} - d_{(j_0+1,j_1)} + d_{(0,j_1)}. \tag{3.2}$$

Note for the last column that the greatest index in $[\varphi(p)]$ is $p-2$ and that $d_{(p-1,j_1)} = 0$. □

Remark 3.4.2. It follows immediately from Proposition 3.4.1 that we can divide an element by g in the decoding basis if we multiply the coordinate vector with $A^{-1} = \bigotimes_{l=0}^{s} A_l^{-1}$. For a prime power m_l of some prime p_l, the inverse A_l^{-1} is given by

$$A_l^{-1} = \frac{1}{p_l} \begin{pmatrix} 1 & 2-p_l & 3-p_l & \cdots & -1 \\ 1 & 2 & 3-p_l & \cdots & -1 \\ 1 & 2 & 3 & \cdots & -1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 2 & 3 & \cdots & p_l-1 \end{pmatrix} \otimes I_{[m_l']}.$$

All matrices, A_l and A_l^{-1} for each $0 \leq l \leq s$, can be applied in linear time, as described in Section 4.5.3. Furthermore, it useful to avoid rational arithmetic and therefore division by g should be done by first multiplying the coordinate vector with the integral matrix $\mathrm{rad}(m)A^{-1} = \bigotimes_{l=0}^{s} p_l A_l$ and then evenly divide the result by $\mathrm{rad}(m) = \prod_{l=0}^{s} p_l$. If the latter step is not possible, the decoded element is not in $\mathcal{I}$ and decoding failure is detected.

We can also multiply by g in the powerful basis. Let m be a prime power of a prime p and assume a setting as in Proposition 3.4.1. The powerful basis of R is given by

$$p_{(j_0,j_1)} = \zeta_p^{j_0} \cdot \zeta_m^{j_1}$$

for pairs $(j_0, j_1) \in [\varphi(p)] \times [m']$. Multiplying with $g = (1 - \zeta_p)$ yields

$$g \cdot p_{(j_0,j_1)} = (1-\zeta_p) \cdot \zeta_p^{j_0} \cdot \zeta_m^{j_1} = \zeta_p^{j_0} \cdot \zeta_m^{j_1} - \zeta_p^{j_0+1} \cdot \zeta_m^{j_1} = p_{(j_0,j_1)} - p_{(j_0+1,j_1)},$$

for $0 \leq j_0 \leq \varphi(p) - 2$. In the case $j_0 = \varphi(p) - 1 = p - 2$ we have

$$\begin{aligned} \zeta_p^{j_0} \cdot \zeta_m^{j_1} - \zeta_p^{j_0+1} \cdot \zeta_m^{j_1} &= \zeta_p^{p-2} \cdot \zeta_m^{j_1} - \zeta_p^{p-1} \cdot \zeta_m^{j_1} \\ &= \zeta_p^{p-2} \cdot \zeta_m^{j_1} + (1 + \ldots + \zeta_p^{p-2}) \cdot \zeta_m^{j_1} \\ &= 2\zeta_p^{p-2} \cdot \zeta_m^{j_1} + \zeta_p^{p-3} \cdot \zeta_m^{j_1} + \ldots + 1 \cdot \zeta_m^{j_1} \\ &= 2p_{(j_0,j_1)} + p_{(j_0-1,j_1)} + \ldots + p_{(0,j_1)}, \end{aligned}$$

where we used the equality $1 + \zeta_p + \ldots + \zeta_p^{p-1} = 0$. These equalities show that multiplication with g is given by the matrix JA^TJ, where $J = J_{[n]}$ is the $[n] \times [n]$ reversal identity matrix, i.e.,

$$J = \begin{pmatrix} & & 1 \\ & \iddots & \\ 1 & & \end{pmatrix}.$$

We have $J = J^{-1}$, $J_{[n]} = J_{[\varphi(p)]} \otimes J_{[m']}$ and $(JA^TJ)^{-1} = J(A^T)^{-1}J$, which is used for division by g. Thus, multiplication by g in the powerful basis is also done in linear time.

Regarding the above considerations, we could also use the decoding or powerful basis in the first step of the decoding procedure for $k \geq 2$, since we can efficiently multiply with g in both bases. But still, the CRT basis remains the best choice here, as it produces the easiest computations overall.

3.5 Pre-Computation

In order to speed up our computations and algorithms, it is useful to pre-compute certain elements and transformations. Usually, in an application, the toolkit will be initialized only once and pre-computation really makes great sense. Concerning transformation matrices there is nothing to say about the pre-computation, since all our transformations are defined in a direct way and can be pre-computed straight forwardly. A different situation arises when pre-computing certain elements. All special elements of our interest are defined in an algebraic way. In an implementation they are represented by a coordinate vector with respect to a certain basis. These coordinates might not be available directly from the definition, but have to be computed, which is what we explain in the following sections.

3.5.1 The Element g

As mentioned in Section 3.4, the decoding procedure can be speed up by pre-computing the element $g = \prod_{2 \neq p \mid m} (1 - \zeta_p)$ in the CRT basis. It can be convenient, to first compute g in the powerful basis, and then swap the basis to CRT via $\mathrm{CRT}_{m,q}$. Recall the definition of the powerful basis from Definition 2.2.1. We need to represent every ζ_p in the powerful basis, where p is an odd prime dividing m. An element is represented by a coordinate vector with the same index set as the powerful basis, which is $\prod_{l=0}^{s} [\varphi(m_l)]$ for prime power factors $m_0, \ldots, m_s$ dividing m. The element of the powerful basis indexed by $(j_l)_{l=0,\ldots,s}$ is given by $p_{(j_l)} = \prod_{l=0}^{s} \zeta_{m_l}^{j_l}$. Let m_l be a power of some prime p_l and $m_l' := m_l/p_l$, then we have $\zeta_{p_l} = \zeta_{m_l}^{m_l'}$. Thus, if we have a coordinate vector which is one in the entry indexed by $(0, \ldots, 0, m_l', 0, \ldots, 0)$, for a fixed l, and zero otherwise, the represented element would be ζ_{p_l}. Note that $\varphi(m_l) = \varphi(p_l) \cdot m_l'$, so m_l' is indeed a valid entry for this index. Since an implementation will most likely work with linear index sets, we have to translate the index $(0, \ldots, 0, m_l', 0, \ldots, 0)$ into an index in $[\varphi(m)]$. This is done by the bijection

$$\prod_{l=0}^{s} [\varphi(m_l)] \to [\varphi(m)]$$

$$(j_l)_l \mapsto j = j_0 \cdot \varphi(m_1) \cdots \varphi(m_s) + \ldots + j_{s-1} \cdot \varphi(m_s) + j_s.$$

Having this insight, we can pre-compute g in the following manner:

- Compute the prime power factorization $m = \prod_{l=0}^{s} m_l$ of m.
- Initialize $g = 1 \in R$ in the powerful basis, i.e., g is represented by the vector $(1, 0, \ldots, 0)$.
- For each m_l, except when $p_l = 2$, represent the element ζ_{p_l} by the vector having entry 1 at the index $m_l' \cdot \ \varphi(m_{l+1}) \cdots \varphi(m_s)$ and update $g = g \cdot (1 - \zeta_{p_l})$.
- Finally switch the basis of g to $\vec{c}$.

If the element $g \in R$ is not needed, but only $g \in R_q$, then we can directly represent all elements in the CRT basis. In more detail, we can compute ζ_{p_l} in the powerful basis, switch the basis to $\vec{c}$ and then multiply also in $\vec{c}$, which is just coordinate-wise multiplication. In this way we can omit the embedding σ and computation with high precision complex numbers.

3.5.2 Ones in Different Bases

While working with our toolkit, it can be quite convenient to have access to ones represented in different bases. For the powerful basis, this is not worth mentioning, since it is clear by definition that the coordinate vector of $1 \in R$ in the powerful basis is given by $v = (1, 0, \ldots, 0)$. Since the powerful and CRT bases are related via $\mathrm{CRT}_{m,q}$, it is also clear that $1 \in R_q$ is represented in $\vec{c}$ by $\mathrm{CRT}_{m,q} \cdot v = (1, \ldots, 1)$. However, this does not help us to determine the coordinate vector of $1 \in R^\vee$ in the decoding basis. Moreover, there is no trivial way to specify this vector from the definition of $\vec{d}$. Fortunately there is a way to compute the desired coordinate vector via the canonical embedding σ.

Let $K = \mathbb{Q}(\zeta_m)$ and $n = \varphi(m)$ for some positive integer m. Further let $\{\sigma_i\}_{i \in \mathbb{Z}_m^*}$ be the n distinct ring-homomorphisms from K to $\mathbb{C}$. For each $i \in \mathbb{Z}_m^*$ we have that $\sigma_i(1) = 1 \in \mathbb{C}$. Thus, the canonical embedding σ maps the one in K to the all ones vector in H, i.e.,

$$\sigma(1) = (\sigma_i(1))_{i \in \mathbb{Z}_m^*} = (1)_{i \in \mathbb{Z}_m^*}.$$

Since the pull back σ^{-1} in the decoding basis $\vec{d}$ is done by multiplication with CRT_m^*, the coordinate vector $\mathbf{1}$ of $1 \in R^\vee$ in $\vec{d}$ is given by

$$\mathbf{1} = \mathrm{CRT}_m^* \cdot (1, \ldots, 1)^T.$$

Now we can also switch the bases and represent $1 \in R^\vee$ in powerful and CRT bases of $R^\vee$.

4 An Implementation in C++

In this chapter we present our C++ implementation of the toolkit [LPR13b], which we described in the beginning of this work. The complete source code is available at `https://github.com/CMMayer/Toolkit-for-Ring-LWE.git`. In their most recent work [CP15] (which was written simultaneously to this one) Crockett and Peikert introduce a general-purpose library for lattice-based and ring cryptography written in the functional language Haskell. In particular, this library includes an implementation of the toolkit [LPR13b]. Next to [CP15] this is - as far as we know - the first implementation written in C++, which is why we decided to focus on functionality rather than efficiency. That is, we use several classes from the C++ standard library to aid readability and handling of the source code. Using these classes often trades off with efficiency compared to native structures that we could have used as well. We also omitted parallel programming and multi threading, since it is highly liable for errors and unexpected behavior. There is still much work to do in order to get our program practicable and maybe usable for on-time cryptography. Nevertheless, a working program is a good start to build upon.

Dealing with arrays in C++ is often very unhandy, especially when working with dynamic arrays. Therefore, we chose the class `std::vector` from the C++ standard library (STL) for the representation of the coordinates of our elements. Despite the fact that arrays are faster we gain some advantages like range checks for indexes, better usability with containers and functions as well as compatibility to many useful algorithms from the STL. Another subtlety occurs while working with pointers in C++. Linking pointers to temporary elements can cause unexpected behavior. Moreover, the lack of a garbage collector in C++ can lead to unnecessary memory leaks, when pointers are treated incorrectly. We can avoid the latter by the use of smart pointers provided by the STL. Smart pointers manage their memory usage automatically and we do not have to worry about that issue any more. There are three different sorts of smart pointers implemented in the STL. We will use only two of them, namely *shared pointers* and *unique pointers*. As the names suggest, shared pointers are used for object, where more than one pointer might point to. They keep track of the number of different pointers pointing to the same object. Unique pointers, on the other hand, are intended for objects where at all times only one pointer points to.

Using these STL classes together with STL algorithms and local functions, we try to help the reader in terms of the understanding of our source code. Studying the source code together with the following introduction to our program should be sufficient to understand why and how everything works.

Before we present the different classes of our program, we start with some algorithms and utility functions we throughout the program. In Section 4.1 we describe our implementation of the algorithm for applications of Kronecker-decompositions from Section 3.1. Furthermore, in Section 4.2 we explain our implementation of Rader and Cooley-Tukey FFT, which is used for the application of prime-indexed CRT matrices. Finally, in Section 4.3 we provide and explain several useful mathematical functions.

4.1 Application of Kronecker Decompositions

In Section 3.1 we described an algorithm that efficiently multiplies a given coordinate vector with a matrix that is decomposed via the Kronecker product. The presented algorithms can be implemented straightforward and there is no further explanation necessary. Therefore, we only list the declarations of the functions which implement the algorithms and present the involved data types. Our program provides two algorithms, `applyKroneckerDecomposition` for the application of a decomposed matrix

$$A = \bigotimes_{l=0}^{s} A_l$$

and `applySingleKroneckerDecomposedMatrix` for the application of a single matrix $I_n \otimes A \otimes I_{n'}$, where the latter is used as a subroutine in the first routine, but it is also used separately by other classes. The declarations are as follows.

Listing 4.1: Transformation Algorithms

```
template<typename T> using avt_uptr =
    std::unique_ptr<AbstractVectorTransformation<T>>;

template<typename T> void
    applySingleKroneckerDecomposedMatrix(std::vector<T>& x,
    AbstractVectorTransformation<T> const* matrix, std::vector<T> const&
    vec, int const dim1, int const dim2)

template<typename T> void applyKroneckerDecomposition(std::vector<T>& x,
    std::list<avt_uptr<T>> const& matrices, std::vector<T> const& vec)
```

The class `AbstractVectorTransformation` represents all vector transformations we work with and is defined in Section 4.5. Since we consider integer, real and complex vector transformations, the functions are actual template functions, so there is a separate function for each required data type. As mentioned above `applyKroneckerDecomposition` calls `applySingleKroneckerDecomposition` with the transformations in the input list. Although the list consists of unique pointers to vector transformations, the application function `applySingleKroneckerDecomposition` expects a normal pointer to a constant vector transformation. This is due to convenience and will be no problem, since the unique pointers also provide a normal pointer to the element they point to.

The class `AbstractVectorTransformation` provides a function that applies itself to a vector, which is used inside `applySingleKroneckerDecomposition` for matrix multiplication. On the other hand, some transformations will themselves call again the procedure `applyKroneckerDecomposition` in their application function which causes a class overlapping recursion. This behavior is described in more detail in Section 4.5.

4.2 Rader and Cooley-Tukey FFT

As described in Section 3.2 we use some Fast Fourier Transformations (FFT) in order to efficiently apply the matrices DFT_p and CRT_p from Definition 2.2.8 for some prime p. We will use Rader's FFT algorithm, which has a Cooley-Tukey FFT as a subroutine. Detailed

descriptions and explanations of both algorithm can be found in [Nus82]. For a more vivid introduction to Rader's FFT with examples and implementation notes see [Ros13].

Our implementation provides the following functions for efficient FFT algorithms.

Listing 4.2: FFT algorithms

```
typedef std::vector<std::complex<double>> complex_vec;

void cooley_tukey_fft(complex_vec& output, complex_vec const& input);
void cooley_tukey_ifft(complex_vec& output, complex_vec const& input);

void cooley_tukey_fft(complex_vec& output, complex_vec const& input,
   int N, int start = 0, int step = 1);

void cooley_tukey_ifft(complex_vec& output, complex_vec const& input,
   int N, int start = 0, int step = 1);

void rader_dft_for_primes(complex_vec& output, complex_vec const& input,
    complex_vec const& precomp_DFT_omega_p, int generator);

void rader_crt_for_primes(complex_vec& output, complex_vec const& input,
    complex_vec const& precomp_DFT_omega_p, int generator);

void rader_crt_star_for_primes(complex_vec& output,
   complex_vec const& input, complex_vec const& precomp_DFT_omega_p,
   int generator);

void unitsGeneratorPermutation(complex_vec& output,
   complex_vec const& input, int generator, int p);

void unitsGeneratorPermutationInverse(complex_vec& output,
   complex_vec const& input, int generator, int p);

void zeroPadding(complex_vec& output, complex_vec const& input,
   int newSize);

void cyclicPadding(complex_vec& output, complex_vec const& input,
   int newSize);
```

In the following we will explain our implementations of Rader and Cooley-Tukey FFT. We start with the Cooley-Tukey algorithm.

4.2.1 Cooley-Tukey FFT

Recall from Section 3.2 that the discrete Fourier transformation (DFT) for a complex sequence $(x_0, \ldots, x_{N-1})$ of length N is defined as

$$X_k = \sum_{j=0}^{N-1} x_j \omega_N^{jk} \quad \text{for } 0 \leq k \leq N-1,$$

where $\omega_N = \exp(2\pi i/N) \in \mathbb{C}$. Assume that the input length N is a power of two, i.e., $N = 2^s$ for some $s \in \mathbb{N}$. In this case we can split the above sum into two sums, one for the elements indexed by odd numbers and one for the numbers indexed by even numbers. For any $0 \leq k \leq N-1$ we have the equality

$$X_k = \sum_{j=0}^{N/2-1} x_{2j} \omega_N^{(2j)k} + \sum_{j=0}^{N/2-1} x_{2j+1} \omega_N^{(2j+1)k}.$$

Now, on the one hand, we have that

$$\omega_N^2 = \left(e^{\frac{2\pi i}{N}}\right)^2 = e^{\frac{2\pi i}{N/2}} = \omega_{N/2}$$

and on the other hand we can factor out ω_N^k in the second sum. Hence we can rewrite the above equality to

$$X_k = \sum_{j=0}^{N/2-1} x_{2j}\omega_{N/2}^{jk} + \omega_N^k \sum_{j=0}^{N/2-1} x_{2j+1}\omega_{N/2}^{jk}.$$

Define $E_k := \sum_{j=0}^{N/2-1} x_{2j}\omega_{N/2}^{jk}$ and $O_k := \sum_{j=0}^{N/2-1} x_{2j+1}\omega_{N/2}^{jk}$. Both, E_k and O_k are discrete Fourier transforms of length $N/2$ and we have

$$X_k = E_k + \omega_N^k O_k.$$

Consequently, we can compute X_k recursively via E_k and O_k. Since $\omega_{N/2}^{N/2} = 1$, we have that $E_{k+N/2} = E_k$ and $O_{k+N/2} = O_k$. Further we have that

$$e^{\frac{2\pi i}{N}(k+N/2)} = e^{\frac{2\pi ik}{N}+\pi i} = e^{\pi i}e^{\frac{2\pi ik}{N}} = -e^{\frac{2\pi ik}{N}}.$$

Summarizing the facts, we can restate the discrete Fourier transformation as

$$\begin{aligned} X_k &= E_k + \omega_N^k O_k, \\ X_{k+N/2} &= E_k - \omega_N^k O_k, \end{aligned}$$

for $0 \leq k \leq N/2 - 1$. This is the basic idea of the Cooley-Tukey FFT for powers of two.

We will also need the inverse discrete Fourier transformation (DFT^{-1}), which is defined as

$$x_k = \frac{1}{N}\sum_{j=0}^{N-1} X_j\omega_N^{-jk} \quad \text{for } 0 \leq k \leq N-1.$$

Assuming that N is a power of two, we can split the sum analogously to the discrete Fourier transform. The only difference we will get in the end is that for $0 \leq k \leq N/2 - 1$ we have

$$\begin{aligned} x_k &= \frac{1}{N}(e_k + \omega_N^k o_k), \\ x_{k+N/2} &= \frac{1}{N}(e_k - \omega_N^k o_k), \end{aligned}$$

where $e_k := \sum_{j=0}^{N/2-1} x_{2j}\omega_{N/2}^{-jk}$ and $o_k := \sum_{j=0}^{N/2-1} x_{2j+1}\omega_{N/2}^{-jk}$. Thus, we can compute $N \cdot x_k$ for $0 \leq k \leq N-1$ recursively via e_k and o_k and divide the result by N to get x_k. This is the Cooley-Tukey IFFT algorithm.

Now our implementations of Cooley-Tukey FFT and IFFT are straightforward using the above results. We only emphasize that our algorithms work on the same vector throughout the complete recursion. That is, instead of splitting the input vector of length N into two subvectors of length $N/2$, we want to traverse through the input vector such that we only meet the elements of a specific subvector. To do this, we provide the length of the currently regarded subvector as well as a start position and a step width to traverse through the original vector. To illustrate this, let $\mathbf{x} = (x_0, \ldots, x_{N-1})$ be the original input for a power of two N.

In order to get the subvector of all entries indexed by odd numbers, we provide the length $N/2$, the starting position $s_0 = 0$ and the step width $s_1 = 2$. Now, if we traverse through x starting at position s_0 and performing $N/2 - 1$ steps of width s_1 we will meet exactly the elements indexed by odd numbers. If we change s_0 to 1 and do the same again, we will meet all elements indexed by even numbers. We can use the same method to get the subvectors containing every 4-th element or every 8-th element and so on. In particular, we can achieve every subvector that occurs in the whole recursion of a Cooley-Tukey FFT.

4.2.2 Rader FFT

Our implementation of Rader's FFT algorithm follows the descriptions of [Ros13]. We leave the following statements without any proof. For a detailed and complete explanation see e.g. [Nus82].

The basic idea of Rader was that we can view the discrete Fourier transformation for prime length p as a convolution. To do this, he took advantage of a different indexation induced by the group Z_p^*. We know from group theory that $\mathbb{Z}_p^*$ contains at least one generator g, i.e., g is of order $\varphi(p) = p - 1$ and for each $a \in \mathbb{Z}_p^*$ there is a unique $j \in [p-1]$ such that $a = g^j$. The inverse g^{-1} in $\mathbb{Z}_p^*$ is also a generator and we can equivalently say that for each $a \in \mathbb{Z}_p^*$ there is a unique $i \in [p-1]$ such that $a = g^{-i}$. Let $\mathbf{x} = (x_0, \dots, x_{p-1})$ be an input vector and $\mathbf{X} = (X_0, \dots, X_{p-1})$ the result of a discrete Fourier transformation. We can use a generator g of $\mathbb{Z}_p^*$ to rewrite the discrete Fourier transformation

$$X_a = x_0 + \sum_{b=1}^{p-1} x_a \omega_p^{ba}, \quad a \in \{0\} \cup \mathbb{Z}_p^*$$

to

$$X_{g^{-k}} = x_0 + \sum_{j=0}^{p-2} x_{g^j} \omega_p^{g^{-(k-j)}}.$$

Note that the reindexation via g actually permutes the entries of $\mathbf{x}$ and $\mathbf{X}$ and we have to take this into account in our implementation. The latter sum in the above equation is in particular a convolution, which can be computed via discrete Fourier transformations. Let $\widetilde{X} = (X_{g^{-j}})_{j \in [p-1]}$, $\widetilde{x} = (x_{g^j})_{j \in [p-1]}$ and $\widetilde{\omega}_p = (\omega_p^{g^{-j}})_{j \in [p-1]}$. Then we have that

$$\widetilde{X} - x_0 = \mathrm{DFT}^{-1}(\mathrm{DFT}(\widetilde{x}) \odot \mathrm{DFT}(\widetilde{\omega}_p)),$$

where $\odot$ denotes component-wise multiplication. Our goal is to use the Cooley-Tuckey FFT for powers of two to compute $\mathrm{DFT}^{-1}(\mathrm{DFT}(\widetilde{x}) \odot \mathrm{DFT}(\widetilde{\omega}_p))$. Well, we know that the vectors $\widetilde{x}$ and $\widetilde{\omega}_p$ are of length $p-1$, which is in general not a power of two. Nevertheless, there is a way to solve this problem using different padding methods. Let $M \geq 2p - 3$ be a power of two. We transform both, $\widetilde{x}$ and $\widetilde{\omega}_p$ into vectors of length M. For $\widetilde{x}$ let $\widetilde{x}'$ be the vector that we get if we insert the necessary amount of zeros between the first and second entry of $\widetilde{x}$ so that the length of $\widetilde{x}'$ is M. For $\widetilde{\omega}_p$ we denote by $\widetilde{\omega}'_p$ the vector of length M that contains the entries of $\widetilde{\omega}_p$ in a cyclic loop. On can show that the first $p-1$ elements of $\mathrm{DFT}^{-1}(\mathrm{DFT}(\widetilde{x}') \odot \mathrm{DFT}(\widetilde{\omega}'_p))$ are equal to $\mathrm{DFT}^{-1}(\mathrm{DFT}(\widetilde{x}) \odot \mathrm{DFT}(\widetilde{\omega}_p))$. The vector $\mathrm{DFT}(\widetilde{\omega}'_p)$ should be pre-computed, since it is equal for every DFT of length p.

Recall that we use Rader's algorithm to apply the transformation matrix DFT_m from Definition 2.2.8. In Section 3.2 we described, how applications of $\mathrm{DFT}_m^*, \mathrm{DFT}_m^{-1}$ and $(\mathrm{DFT}_m^*)^{-1}$

can be viewed as equations similar to DFT_m. These applications can also be done via Rader FFT. If we compare the equations for $\mathrm{DFT}_m^*, \mathrm{DFT}_m^{-1}$ and $(\mathrm{DFT}_m^*)^{-1}$

$$X_a = x_0 + \sum_{b=1}^{p-1} x_a \overline{\omega_p}^{ba}, \quad a \in \{0\} \cup \mathbb{Z}_p^*,$$

$$X_a = \frac{1}{p}(x_0 + \sum_{b=1}^{p-1} x_a \omega_p^{-ba}), \quad a \in \{0\} \cup \mathbb{Z}_p^*,$$

$$X_a = \frac{1}{p}(x_0 + \sum_{b=1}^{p-1} x_a \overline{\omega_p}^{-ba}), \quad a \in \{0\} \cup \mathbb{Z}_p^*,$$

we see that they differ mainly in the multiplication with ω_p. First, in the inverse cases we have to divide the result by p. Besides that, by the above considerations it is sufficient to adjust the pre-computed vector $\mathrm{DFT}(\widetilde{\omega}_p')$ in order to perform an application of $\mathrm{DFT}_m^*, \mathrm{DFT}_m^{-1}$ or $(\mathrm{DFT}_m^*)^{-1}$. To be more precise, the usage of $\overline{\omega_p}$ in the computation of $\mathrm{DFT}(\widetilde{\omega}_p')$ corresponds to an application of DFT_m^*; the usage of $\omega_p^{-1} = \overline{\omega_p}$ in the computation of $\mathrm{DFT}(\widetilde{\omega}_p')$ corresponds to an application of DFT_m^{-1}; and the usage of $\overline{\omega_p}^{-1} = \omega_p$ in the computation of $\mathrm{DFT}(\widetilde{\omega}_p')$ corresponds to an application of $(\mathrm{DFT}_m^*)^{-1}$.

The implementation of `rader_dft_for_primes` follows closely the explanations from above. For an input $(x_0, \ldots, x_{p-1})$, we permute the size $p-1$ subvector $(x_1, \ldots, x_{p-1})$ according to the permutation induced by the generator g. If $p-1$ is not a power of two we zero pad the permuted vector to achieve a vector whose size is a power of two. Then we perform Cooley-Tukey FFT and multiply the result component-wise with the pre-computed $\mathrm{DFT}(\widetilde{\omega}_p')$ vector, which is given as an input parameter. Now we perform Cooley-Tukey IFFT and extract the first $p-1$ entries of the result. Lastly, we permute the $p-1$ entries according to the permutation induced by g^{-1} and add the offset x_0.

Finally, we want to use Rader FFT also for the application of CRT_p and CRT_p^*. Recall from Definition 2.2.8 that CRT_p is a submatrix of DFT_p. This implies that CRT_p^* is a submatrix of DFT_p^*. In particular, we receive CRT_p if we remove the first row and the last column of DFT_p. Therefore, in the algorithm for CRT we do not need to compute the first entry X_0. The remaining entries $X_1, \ldots, X_{p-1}$ can be computed as in the DFT case, when we add a zero at the end of the input vector. The matrix CRT_p^* can be achieved from DFT_p^* if the first column and the last row are removed. The algorithm for CRT_p^* is again a slight adjustment of Rader FFT. First note that the input vector is of size $p-1$. We can treat it as the vector $(x_1, \ldots, x_{p-1})$ from above, i.e., we do not need the offset x_0, since this corresponds to the first column of DFT_m^*. If we perform the same steps as in Rader FFT for DFT_p^* we get a result vector $(X_1, \ldots, X_{p-1})$. The entry X_{p-1} is the multiplication of the input vector with the last row of DFT_p^*, which we do not need. Additionally, the first entry X_0, which corresponds to the multiplication of the input vector with the first row of DFT_p^*, is missing. Therefore, we can shift the result vector to the right and replace the first entry to get $(X_0, \ldots, X_{p-2})$. The entry X_0 can be computed at the beginning of the algorithm.

As we mentioned already in Section 3.2 we were not able to apply CRT_p^{-1} and $(\mathrm{CRT}_p^*)^{-1}$ via Rader's FFT algorithm. Instead we use standard matrix-vector multiplication, which needs $O(p^2)$ time instead of $O(p \log p)$.

4.3 Mathematics Utilities

Throughout our implementation we need to solve some tasks from general mathematics. We provide algorithms for these tasks in a separate file. To keep dependencies on third-party libraries low, we implemented most algorithms from scratch, where we only use the boost library [DA15] for some tasks concerning primes. Also in this way, the algorithms match the data types of the rest of the program. Here is a list of available functions in the mathematics utility file.

Listing 4.3: Mathematics Utilities

```
int findGeneratorOfZZpUnits(int p);

int getNextPowerOfTwo(int n);

bool isPowerOfTwo(int n);

int fastModPow(int base, int exponent, int modulus);

unsigned int const eulerTotient(unsigned int const n);

std::list<std::pair<unsigned int, unsigned int>>
    primeFactorization(pos_int const m);

unsigned int getPrimeMod1(pos_int const m, pos_int const min);

std::complex<double> const computeRootOfUnity(unsigned int m);

NTL::ZZ_p const findElementOfOrder(unsigned int order, unsigned int p);
```

`findGeneratorOfZZpUnits`. The first three functions are used in the FFT algorithms from Section 3.2. The algorithm `findGeneratorOfZZpUnits` computes a generator g of the unit group $\mathbb{Z}_p^*$ for some prime integer p. It is due to [Buc08] and checks for some random element in $\mathbb{Z}_p^*$, if its order is $\varphi(p)$. Since the generator of $\mathbb{Z}_p^*$ is in general not unique and we use random numbers that we test, the results of several calls of our algorithm with the same input parameters may vary. For further details on the way how the algorithm works, see [Buc08].

`getNextPowerOfTwo` and `isPowerOfTwo`. In particular for the Cooley-Tuckey FFT algorithm from Section 4.2.1 we need to compute powers of 2 that are bigger but as close as possible to some integer n. That is, for a given n the goal is to find a power 2^k such that $2^{k-1} < n \leq 2^k$. For a solving algorithm it is convenient to be able to check if a given n is already a power of 2. We can do this on the bit level. If n is a power of 2 its bit code can be seen as a unit vector, lets say $(0,0,0,1,0,0,0,0)$. Subtracting 1 from n leads to a bit vector having only 1s up to the position previous to the only non-zero entry before. Our example vector would look like $(0,0,0,0,1,1,1,1)$. If we perform a bit-wise AND with $n-1$ and n, the result is exactly the zero bit vector. It is easy to see that the same procedure results in some non-zero vector if n was not a power of 2 in the first place. Thus, if we convert the resulting integer, that is represented by the bit vector, into a boolean, the negation of this

boolean indicates whether n was a power of 2 or not. Now, if we want to compute the next bigger power of 2 for n, we can check if it is already a power of 2 or not. If not, we use again the bit code to our advantage. Clearly, the bit representation of the next bigger power of 2 has a 1 next to the last 1 in the bit vector of n. Unfortunately, C++ does not allow us to manipulate single bits of an integer. An equivalent solution using the same idea is to shift the bits of n one step to the right until it is zero. For each performed step we take the next power of 2 starting at the exponent $k = 1$. The resulting number is the desired power of 2.

fastModPow. If we compute with integers in $\mathbb{Z}_q$ for some prime q, we usually use the class `NTL::ZZ_p`, which makes sure that all arithmetic is done modulo q. However, the data type we use for the coordinates is `int`, so sometimes it might be convenient to perform the modulo arithmetic manually. In particular, this avoids unnecessary conversions between `int` and `NTL::ZZ_p`. Of course the laws of modulo arithmetic allow us to use standard addition and multiplication followed by a call of the modulo operator to get a correct result. But especially when it comes to exponentiation we can use some facts from group theory to speed up computations. There is a well known algorithm for fast modulo exponentiation, which can be found in e.g. [Buc08] including further details on how the algorithm works. Our implementation is just the translation of the pseudocode from [Buc08] to C++.

eulerTotient. Another function from general mathematics that we need is Euler's totient function $\varphi(\cdot)$. The computation of $\varphi(n)$ for some positive integer n is not trivial, but we can use several properties of the Euler totient, known from algebra, to make computations more efficient via recursion. As an anchor we can use that $\varphi(p) = p - 1$ for primes p. Further we know that φ is multiplicative for coprime integers m, n, i.e., $\varphi(mn) = \varphi(m)\varphi(n)$. Finally, for prime powers p^k we have $\varphi(p^k) = (p-1)p^{k-1}$, which for the case $p = 2$ combined with the previous result yields $\varphi(n) = 2^{k-1}\varphi(m)$, where $n = 2^k m$ for an odd m. Christian Stigen Larsen provides an algorithm for Euler's totient in [Lar12] using all these tricks. To complete this algorithm we need a prime test, a list of primes up to a certain prime and a binary greatest common divisor (gcd) function. All these dependencies are implemented via the boost library, using Miller-Rabin for primality testing, a fast look up table providing the first 10000 primes and some gcd function. We note that all relevant input values wont have prime divisors greater than the 10000-th prime.

primeFactorization. Our function for the computation of prime factorizations also uses the look up table from the boost library. For our purposes it is convenient to compute rather a prime power factorization, i.e., directly produce a list of maximal prime powers p^k dividing some input m. We will produce pairs (p, k) of primes p and integers k, such that p^k divides m, but p^{k+1} does not. Since the values of the input m will be rather small it is sufficient to use a simple algorithm, which traverses through a list of primes and counts how often it is possible to evenly divide m by the respective prime. Clearly, since the list of primes we use provides only the first 10000 primes, our algorithm will fail to compute the factorization for inputs with huge prime factors (bigger than the 10000-th prime). As already mentioned this will be no problem for us.

getPrimeMod1. The ring-LWE modulus q is always a prime, such that $q = 1 \mod m$, where the considered field K is the m-th cyclotomic field. Therefore, it is nice to have a

simple procedure that computes such a prime $q \geq M$ greater than some bound M. Again we use the look up table for primes from the boost library. First, we search for a prime that is greater or equal to M. Then, starting at this prime, we traverse through the list until we found a prime $q = 1 \mod m$.

computeRootOfUnity. The function `computeRootOfUnity` is a simple convenience function computing the complex value of the "first" root of unity ω_m. "First" means the first in the unit circle, i.e., $\omega_m = \exp(2\pi i/m)$.

findElementOfOrder. Finally `findElementOfOrder` searches for an element $g \in \mathbb{Z}_p^*$ of a given multiplicative order. The return type is `NTL::ZZ_p`, since we use this type almost always for elements in $\mathbb{Z}_p$ throughout the rest of the program. We assume that the input p is a prime number, such that the order of $\mathbb{Z}_p^*$ is $\varphi(p) = p-1$. Group theory tells us, that the order of each element $g \in \mathbb{Z}_p^*$ divides the group order, which is $p-1$. Furthermore, for each divisor d of $p-1$ $\mathbb{Z}_p^*$ contains exactly $\varphi(d)$ many distinct elements of order d. Consequently, our algorithm will only succeed in finding the desired element g, if and only if, the given order divides $p-1$. Johannes Buchmann describes in [Buc08], how to test algorithmically if a given element has a specific order. Note that this algorithm is similar to `findGeneratorOfZZpUnits`, which is indeed a special case, where we look for an element of order $p-1$. We separated these functions nevertheless, because they are used in different contexts and their output types differ.

4.4 The Basic Class Structure

As the basis for our program, we need a useful class structure that is capable of properly managing all informations we got and need. As we saw in the previous chapters, we work with elements a that live in a specific field $K = \mathbb{Q}(\zeta_m)$ for some positive integer m. More precise, the elements a live in some ideals $\mathcal{I} \subset K$ and are represented by a coordinate vector of size $n = \varphi(m)$ with respect to a specific basis. Also, we saw that we need a lot of transformation matrices, whose definitions are dependent on the input parameter m, i.e., on the prime power factorization of m. So these transformations depend on the same parameter as the field K. We divide our representation into two main parts. We define one class that represents the field K and manages the transformations and another class that represents the elements in K. Thereby, the element class has to be rather "dynamic", because the elements can switch their bases or live in another ideal after a multiplication and so on. On the contrary, the field class has to be more "static" or "constant", since the field and the transformation do not change after their initialization.

As already mentioned there are several vector transformations we use throughout all our operations in the toolkit. These transformations are used to change the basis in which an element is represented in, or to embed an element into the vector space H. We use our field class to manage the different transformations, since they depend on the same input parameter m as the field does. Arising from the tensor structure of the ring R and its $\mathbb{Z}$ bases we consider, all vector transformations we use are decomposed via the Kronecker product. To be more precise, the transformations are decomposed in smaller matrices of prime or prime power dimensions. As we saw in Section 3.1, this yields more efficient algorithms for matrix-vector

multiplication, where only certain subvectors are multiplied with the smaller dimensional matrices. Also, nearly all of the transformation of prime or prime power dimension have special structures allowing us to apply them in a more efficient way than the standard matrix-vector multiplication, which uses quadratic time. These facts imply that a simple matrix class with a standard matrix-vector multiplication is not suitable for our program. Instead we need a transformation class that realizes many different, but very specific matrix-vector multiplications. To do so, we make use of an abstract base class that has a pure virtual function for the application to a vector. In this way, we can realize decomposed transformations like CRT_m with its sparse decomposition (cf. Section 2.2.4), which then itself consists of other transformations of the same base class, or represent a single matrix which can be applied directly to a vector.

4.5 Vector Transformation Matrices

The representation of the vector transformations we use throughout the toolkit has to be versatile in the sense that it has to be capable of managing different transformations for integers, reals and complex numbers that are differently decomposed via the Kronecker product. The basic idea is to define an abstract base class that has only a dimension as a member variable and a pure virtual member function for the application to a vector. Furthermore, the type of the transformation is captured as a template. Using only a single dimension is sufficient, since we consider only square transformation matrices. We call this base class `AbstractVectorTransformation` and define it as follows.

Listing 4.4: The base class for all vector transformations

```
template<typename T> class AbstractVectorTransformation
{
public:
typedef T entry_type;
typedef std::vector<entry_type> entry_type_vec;

// Getter
inline int getDim() const { return dim_; };

// Purely virtual method to apply the matrix to a given vector (Param:
   vec) and store the result in x.
virtual void applyToVector(entry_type_vec& x, entry_type_vec const& vec)
   const = 0;

~AbstractVectorTransformation(){};

protected:

AbstractVectorTransformation(int dim):dim_(dim){};

int dim_; // Dimension of the transformation matrix.
};
```

The virtual function `applyToVector` provides great flexibility for all transformations we realize with this class. Many transformations use different algorithms to apply themselves

to a vector. Since we realize the transformations in different classes that all inherit from `AbstractVectorTransformation`, each of these application algorithms can be implemented through the `applyToVector` function. We can use this function without knowing the exact data type of the actual transformation, since the correct implementation of `applyToVector` is called automatically. This mechanism will become clearer in the following sections, where we describe how the different considered vector transformations are realized by this base class. Thereby we distinguish between integer, real and complex transformations.

Let m be a fixed positive integer with prime power factorization $m = \prod_{l=0}^{s} m_l$, where each m_l is the power of some prime p_l, and $m'_l = m_l/p_l$. In general, all considered transformations are of the form

$$A_m = \bigotimes_{l=0}^{s} A_{m_l}.$$

The factors A_{m_l} are transformations that can be further decomposed in different ways. We store A_m as a list of transformations in our field class. The factors A_{m_l} are then constructed when the field object is constructed. Usually, A_{m_l} is further decomposed such that it relies only on transformations A_{p_l}. The matrices A_{m_l} and A_{p_l} are represented in different classes. We describe this scenario for each transformation in detail. Thereby, we refer to A_{m_l} as the prime power case, and to A_{p_l} as the pure prime case.

4.5.1 Complex Transformations

The only complex vector transformations we consider are sorts of Chinese remainder or discrete Fourier transformations as defined in Definition 2.2.8. Recall the sparse decompositions from Section 2.2.4

$$\mathrm{DFT}_m = (I_{[p]} \otimes \mathrm{DFT}_{m'}) \cdot T_m \cdot (\mathrm{DFT}_p \otimes I_{[m']})$$

and

$$\mathrm{CRT}_m = (I_{\mathbb{Z}_p^*} \otimes \mathrm{DFT}_{m'}) \cdot \hat{T}_m \cdot (\mathrm{CRT}_p \otimes I_{[m']}),$$

where m is a prime power of some prime p, $m' = m/p$ and we omitted the stride permutation. Since m' may be a non trivial prime power itself, $\mathrm{DFT}_{m'}$ can be further decomposed until we are only left with prime-indexed transformations. Therefore, we distinguish between transformations that are already fully decomposed and those who can be further decomposed.

The Pure Prime Case

For some prime p consider the matrices $\mathrm{DFT}_p, \mathrm{DFT}_p^{-1}, \mathrm{DFT}_p^*$ and $(\mathrm{DFT}_p^*)^{-1}$ and their CRT counterparts. Recall from Section 3.2 that we can apply all of these matrices except of CRT_p^{-1} and $(\mathrm{CRT}_p^*)^{-1}$ via Rader's FFT algorithm, which we briefly described in Section 4.2.2. To use Rader FFT we need a fixed generator g of $\mathbb{Z}_p^*$ and the pre-computed discrete Fourier transform of $\widetilde{\omega}_p = (\omega_p^{g^{-j}})_{j\in[p-1]}$, where $\omega_p \in \mathbb{C}$ is a primitive p-th root of unity. If the length $p-1$ of $\widetilde{\omega}_p$ is not a power of two we use cyclic padding to transform $\widetilde{\omega}_p$ into a vector $\widetilde{\omega}'_p$ whose size is a power of two. Then we can use the Cooley-Tukey FFT from Section4.2.1 to pre-compute $\mathrm{DFT}(\widetilde{\omega}'_p)$. For an application of CRT_p^{-1} and $(\mathrm{CRT}_p^*)^{-1}$ we use the standard matrix-vector multiplication and thus have to store the actual matrix in our class.

Because of the different informations we need for CRT_p^{-1} and $(\mathrm{CRT}_p^*)^{-1}$ compared to the remaining transformations, we represent them in two separate classes. For CRT_p^{-1} and

$(\mathrm{CRT}_p^*)^{-1}$ we use the class `MatrixCompMult`. The name indicates that it represents a complex matrix, which uses standard matrix-vector multiplication for an application to a vector. The class is defined as follows.

Listing 4.5: Definition of `MatrixCompMult`

```
class MatrixCompMult : public
    AbstractVectorTransformation<std::complex<double>>
{
public:
typedef std::vector<std::complex<double>> entry_type_vec;
typedef boost::numeric::ublas::matrix<std::complex<double>> matrix_type;

MatrixCompMult(int p, bool adjoint);
MatrixCompMult(MatrixCompMult const& matrix);
~MatrixCompMult();

protected:
virtual void applyToVector(entry_type_vec& x, entry_type_vec const& vec)
    const;

private:
std::unique_ptr<matrix_type> matrix_;

void invert();
};
```

When the constructor is called the matrix CRT_p is computed according to its definition. Then we use the invert function to compute CRT_p^{-1}. If the boolean parameter `adjoint` is `true`, we compute $(\mathrm{CRT}_p^{-1})^* = (\mathrm{CRT}_p^*)^{-1}$. The resulting matrix is stored in the attribute `matrix_`. For the latter, we use the matrix class from the boost library [DA15]. Additionally, the boost library provides functions to compute the conjugate transpose and the LU-factorization of a matrix. We use the LU-factorization to implement the invert function. For an explanation on how the LU-factorization is used to compute the inverse of a matrix see [MvdGvdG15]. The `applyToVector` function also uses tools from the boost library to perform a standard matrix-vector multiplication.

For the representation of the remaining complex transformations with prime indexes we use the class `MatrixCompFFT`. The name indicates that it represents a complex matrix, which uses FFT algorithms for an application to a vector. The class is defined as follows.

Listing 4.6: Definition of `MatrixCompFFT`

```
class MatrixCompFFT : public
    AbstractVectorTransformation<std::complex<double>>
{
public:
typedef std::vector<std::complex<double>> entry_type_vec;

enum class MatrixType{
    DFT_P,
    DFT_P_INV,
    DFT_P_STAR,
```

```
    DFT_P_STAR_INV,
    CRT_P,
    CRT_P_STAR
};

MatrixCompFFT(MatrixType matrixType, int p);
~MatrixCompFFT();

protected:
virtual void applyToVector(entry_type_vec& x, entry_type_vec const& vec)
    const;

private:
MatrixType matrixType_;
int generator;
std::unique_ptr<entry_type_vec> precomp_DFT_omega_p_;
};
```

Since this class represents several matrices, which use different implementations of Rader FFT for an application (cf. Section 3.2 and Section 4.2), we distinguish them via the class enum `MatrixType`. An indicator of this type as well as the dimension p are given on construction. A call of the constructor starts with the computation of a generator g of $\mathbb{Z}_p^*$ and its inverse $g^{-1} = g^{p-2} \mod p$. To do this, we use the functions `findGeneratorOfZZpUnits` and `fastModPow` from the mathematics utilities (see Section 4.3). Depending on the given indicator we compute the vector $\widetilde{\omega}_p$ of the respective complex transformation according to the considerations of Section 4.2.2. Finally, we compute the discrete Fourier transform $\mathrm{DFT}(\widetilde{\omega}'_p)$ using Cooley-Tukey FFT and store it in `precomp_DFT_omega_p_`. The generator g is also stored, since we need it for calls of Rader's FFT algorithms.

Now, the application function simply checks the indicator of the different transformation matrices and calls the respective version of Rader FFT as described in Section 4.2.2, using the generator and the pre-computed discrete Fourier transform.

The Prime Power Case

For a prime power m of some prime p recall again the sparse decompositions of DFT_m and CRT_m

$$\mathrm{DFT}_m = (I_{[p]} \otimes \mathrm{DFT}_{m'}) \cdot T_m \cdot (\mathrm{DFT}_p \otimes I_{[m']})$$

and

$$\mathrm{CRT}_m = (I_{\mathbb{Z}_p^*} \otimes \mathrm{DFT}_{m'}) \cdot \hat{T}_m \cdot (\mathrm{CRT}_p \otimes I_{[m']}).$$

Similar decompositions apply if we consider the conjugate transposes, inverses or inverses of the conjugate transposes of DFT_m and CRT_m (cf. Section 2.2.4 and Section 2.3.4). For example we have

$$\mathrm{DFT}_m^* = \left(\overline{\mathrm{DFT}_p} \otimes I_{[m']}\right) \cdot \overline{T_m} \cdot (I_p \otimes \mathrm{DFT}_{m'}^*)$$

and

$$\mathrm{CRT}_m^* = \left(\mathrm{CRT}_p^* \otimes I_{[m']}\right) \cdot \overline{\hat{T}_m} \cdot \left(I_{\mathbb{Z}_p^*} \otimes \mathrm{DFT}_{m'}^*\right).$$

If we compare, for example, the decompositions of DFT_m and DFT_m^*, the appearing factors are somewhat similar, but in reverse order. Furthermore, since m' might be a proper prime power itself, it holds for all decompositions that the DFT part could be further decomposed.

In the following we refer only to DFT_m and CRT_m. In our explanations these matrices can be interchanged with $\mathrm{DFT}_m^{-1}, \mathrm{DFT}_m^*, (\mathrm{DFT}_m^*)^{-1}$ and their CRT counterparts.

Due to small differences we separate the representation of CRTs and DFTs. The above consideration motivate the following definition of `TransformationCompDft`, which represents all kinds of DFTs for proper prime powers.

Listing 4.7: Definition of `TransformationCompDft`

```
class TransformationCompDft : public
    AbstractVectorTransformation<std::complex<double>>
{
public:
typedef std::vector<std::complex<double>> entry_type_vec;
typedef AbstractVectorTransformation<std::complex<double>> base_type;

TransformationCompDft(int m, int p, bool reversed,
    std::shared_ptr<MatrixCompFFT const> DFT_p);
~TransformationCompDft();

protected:
virtual void applyToVector(entry_type_vec& x, entry_type_vec const& vec)
    const;

private:
bool reversed_;
std::shared_ptr<base_type const> DFT_p_; // MatrixCompFFT
std::unique_ptr<base_type const> DFT_m_prime_; // TransformationCompDFT
std::unique_ptr<entry_type_vec> twiddleMatrix_;
};
```

The boolean `reversed_` is true if we represent DFT_m^* or DFT_m^{-1} and indicates that the factors in the sparse decomposition appear in reversed order. In the end, an application of DFT_m will only depend on DFT_p. Since all iterative decompositions of DFT_m will use the same matrix DFT_p, we use a shared pointer that points to the desired DFT_p object. The latter will be of type `MatrixCompFFT`. The pointer `DFT_m_prime_` points to the $\mathrm{DFT}_{m'}$ object. Since $\mathrm{DFT}_{m'}$ might be further decomposed, the type of this object is again `TransformationCompDFT`. If $m' = m/p = 1$, we are in the last step of the decomposition and have $\mathrm{DFT}_m = \mathrm{DFT}_p$. In this case we leave `DFT_m_prime_` as a null pointer. The same holds for the pointer `twiddleMatrix_`, which points to a vector representing the diagonal twiddle matrix T_m.

Assume that the constructor is called with input m, p, some boolean b and some pointer d, and $m' = m/p > 1$, i.e., $m' = p^k$ for some $k \geq 1$. Then the constructor calls itself with input m', p, b, d to initialize the decomposition of $\mathrm{DFT}_{m'}$. This causes a recursive call of constructors until $m' = 1$ and DFT_m cannot be further decomposed. Also, if $m' > 1$, the twiddle matrix is computed as well. We compute T_m if $b =$ `false`, and $\overline{T_m}$ otherwise.

In the case $m' = 1$, the application function of `TransformationCompDFT` forwards its call directly to the application function of `MatrixCompFFT`, which represents DFT_p. If $m' > 1$, the application functions of the factors of the sparse decomposition are called in the correct order. That is, if `reversed_` = `true` they are called in reversed order, and in normal order otherwise.

For the representation of CRT_m we use the class `TransformationCompCRT`. It is defined as follows.

Listing 4.8: Definition of `TransformationCompCRT`

```
class TransformationCompCrt : public
    AbstractVectorTransformation<std::complex<double>>
{
public:
typedef AbstractVectorTransformation<std::complex<double>> base_type;

enum class EmbeddingType
{
   COMPLEX_CRT_M,
   COMPLEX_CRT_M_INVERSE,
   COMPLEX_CRT_M_STAR,
   COMPLEX_CRT_M_STAR_INVERSE,
};

TransformationCompCrt(int m, int p, EmbeddingType embedding);
~TransformationCompCrt();

protected:
virtual void applyToVector(entry_type_vec& x, entry_type_vec const& vec)
    const;

private:
bool reversed_;
std::unique_ptr<base_type const> CRT_p_; // MatrixCompFFT
std::unique_ptr<base_type const> DFT_m_prime_; // TransformationCompDFT
std::unique_ptr<entry_type_vec> twiddleMatrix_;
};
```

Overall, this class works very similar to `TransformationCompDFT`, although there are some minor differences. First note that the object of CRT_p is linked via a unique pointer, since in opposite to DFT_p there will be only one pointer pointing to this object. The possibly remaining decompositions of $\mathrm{DFT}_{m'}$ do again need the matrix DFT_p. The remaining attributes and the application function behave as in the class `TransformationCompDFT`.

A main difference between `TransformationCompCRT` and `TransformationCompDFT` occurs in the functioning of the constructor. Next to the integers m and p, the constructor expects an indicator for the transformation that is represented. The possible cases are $\mathrm{CRT}_m, \mathrm{CRT}_m^{-1}, \mathrm{CRT}_m^*$ and $(\mathrm{CRT}_m^*)^{-1}$. Depending on the value of the indicator we first construct the object for CRT_p. There are two possible types for these objects. If we represent CRT_m or CRT_m^*, we use `MatrixCompFFT` from Listing 4.6 as the data type for CRT_p or CRT_p^*. If the represented transformation is CRT_m^{-1} or $(\mathrm{CRT}_m^*)^{-1}$, then the data type is `MatrixCompMult` from Listing 4.5 and the object represents CRT_p^{-1} or $(\mathrm{CRT}_p^*)^{-1}$. Now, if $m' > 1$, we can initiate the construction of the respective $\mathrm{DFT}_{m'}$ of type `MatrixCompFFT` and the vector representing the twiddle matrix. Thereby the constructor of the $\mathrm{DFT}_{m'}$ object needs a shared pointer to the DFT_p object for the sparse decomposition. This shared pointer is also constructed at this stage.

4.5.2 Real Transformations

The real transformations that we need are the matrices D and U from the Gram-Schmidt orthogonalization of the CRT matrix from Remark 2.2.21 and C^*B' from Proposition 2.3.10, which is used to efficiently sample Gaussians over $K_{\mathbb{R}}$ in the decoding basis. In all cases the transformations are given in the form

$$A_m = \bigotimes_{l=0}^{s} (A_{p_l} \otimes I_{[m'_l]}),$$

where p_l are the distinct prime factors of m and $m'_l = m_l/p_l$ for all prime powers m_l dividing m. Thus, we can split the representation into three parts. As already mentioned above, the complete transformation A_m is given as a list of transformations of the type $(A_{p_l} \otimes I_{[m'_l]})$ and stored in our field class. Further, we separate $(A_{p_l} \otimes I_{[m'_l]})$ and A_{p_l} in two classes. We refer to $(A_{p_l} \otimes I_{[m'_l]})$ as the prime power case and to A_{p_l} as the pure prime case.

The Pure Prime Case

As discussed in Section 2.2.5, concerning D and U, we will only need the i-th diagonal entries of D and the i-th rows of U in our computations. Since we defined D and U as Kronecker decompositions, we do not have direct access to these values. However, remember that we have an efficient algorithm for the application of Kronecker decompositions (cf. Section 3.1 and Section 4.1). Using this algorithm with the i-th unit vector as input will result in the i-th column of the specific transformation. Thus we can access the desired values via applications of $D = \bigotimes_{l=0}^{s} (\sqrt{m'_l} D_{p_l} \otimes I_{[m'_l]})$ and $U^T = \bigotimes_{l=0}^{s} (U_{p_l}^T \otimes I_{[m'_l]})$ to the appropriate unit vectors. Further, the matrices D_{p_l} and U_{p_l} are defined in a direct way allowing us to implement specialized algorithms for their application that are faster than standard matrix-vector multiplication. Additionally, these algorithms are specialized in a way that we do not need to store the specific matrices D_{p_l} and $U_{p_l}^T$.

Unfortunately, the matrix C^*B' lacks this nice feature and has to be applied via the standard matrix-vector multiplication. At least we can easily pre-compute the prime case matrices $\mathrm{CRT}^*_{p_l} \cdot B'_{p_l}$ and store them in order to speed up the following multiplications with C^*B'.

Following these different properties of the real transformations we represent them in two distinguished classes. For $\sqrt{m'} D_{p_l}$ and $U_{p_l}^T$ we define the class `MatrixRealGS`, where "GS" stands for Gram-Schmidt.

Listing 4.9: Definition of `MatrixRealGS`

```
class MatrixRealGS : public AbstractVectorTransformation<double>
{
public:
MatrixRealGS(int dim, int m_prime, bool D_or_U);
MatrixRealGS(MatrixRealGS const& mr);

~MatrixRealGS();

protected:
virtual void applyToVector(entry_type_vec& x, entry_type_vec const& vec)
    const;
```

```
private:
int m_prime_;
bool GS_D_or_U_;
void gsU(entry_type_vec& x, entry_type_vec const& vec) const;
void gsD(entry_type_vec& x, entry_type_vec const& vec) const;
};
```

The boolean attribute `GS_D_or_U_` indicates whether $\sqrt{m'}D_{p_l}$ or $U_{p_l}^T$ is represented. The dimension of these matrices is $\varphi(p_l) = p_l - 1$ which is given on construction. The attribute `m_prime_` stores the integer m'_l, which we need for the application of $\sqrt{m'}D_{p_l}$ and is also given on construction. The procedures `gsD` and `gsU` are subroutines used in `applyToVector` and apply the matrices $\sqrt{m'}D_{p_l}$ and $U_{p_l}^T$, respectively, to a given input vector. If `GS_D_or_U_` is true, then `gsD` is called, and `gsU` otherwise. The computations in these routines are based on the definitions of U_{p_l} and D_{p_l} from Lemma 2.2.20. Multiplication with $\sqrt{m'}D_{p_l}$ can be done in linear time and an implementation is straight forward, since it is a diagonal matrix. Noticing that the increasing rows of $U_{p_l}^T$ differ in only one entry, we see that $U_{p_l}^T$ can also be applied in linear time via successive sums.

The matrices $\mathrm{CRT}_{p_l}^* \cdot B'_{p_l}$ are represented by the class `MatrixRealSampleGauss`, which we define as follows.

Listing 4.10: Definition of `MatrixRealSampleGauss`

```
class MatrixRealSampleGauss : public AbstractVectorTransformation<double>
{
public:
typedef std::vector<entry_type> entry_type_matrix;
// Use a single vector to represent a matrix.

MatrixRealSampleGauss(int dim);
MatrixRealSampleGauss(MatrixRealSampleGauss const& mr);

~MatrixRealSampleGauss();

protected:
virtual void applyToVector(entry_type_vec& x, entry_type_vec const& vec)
    const;

private:
std::unique_ptr<entry_type_matrix> matrix_;
};
```

Again the dimension $p_l - 1$ is given on construction. Additionally we have a unique pointer to a vector of doubles which stores the actual matrix $\mathrm{CRT}_{p_l}^* \cdot B'_{p_l}$. In Section 2.3.5 we showed how each entry of this matrix can be computed given only its dimension. Consequently we can pre-compute $\mathrm{CRT}_{p_l}^* \cdot B'_{p_l}$ on construction and any further applications are done via standard matrix-vector multiplication.

The Prime Power Case

Let m be a power of some prime p and $m' = m/p$. Then, the transformation we have to represent is of the form

$$A_p \otimes I_{[m']},$$

where A_p is a $[\varphi(p)] \times [\varphi(p)]$ matrix. Hence, the dimension of the transformation is $n = \varphi(m) = \varphi(p)m'$. The prime-indexed matrix A_p will be an object of type `MatrixRealGS` or `MatrixRealSampleGauss`, which do both inherit from the same base class. Consequently, we store A_p in an attribute which is a unique pointer to exactly this base class.

Listing 4.11: Definition of `TransformationRealPrimePower`

```
class TransformationRealPrimePower : public
    AbstractVectorTransformation<double>
{
public:
typedef AbstractVectorTransformation<double> base_type;

// the real vector transformation types
enum class TransformationType
{
REAL_GS_D,
REAL_GS_U,
REAL_CONVERT_GAUSSIANS
};

TransformationRealPrimePower(int m, int p,
    TransformationType transformation);

~TransformationRealPrimePower();

protected:
virtual void applyToVector(entry_type_vec& x, entry_type_vec const& vec)
    const;

private:
std::unique_ptr<base_type const> matrix_p_;
};
```

The constructor switches over the indicator of the represented transformation and constructs a unique pointer to the respective object for A_p. That is, if we represent D or U, the desired matrix object is of type `MatricRealGS`. The constructor of `MatricRealGS` asks for the dimension $\varphi(p) = p - 1$ of D and U, the value m' and a boolean indicating whether D or U is represented. The latter is `true` for D and `false` for U. On the other hand, if we represent $C * B'$, the desired matrix object is of type `MatricRealSampleGauss`. Then,we only have to pass the dimension $\varphi(p) = p - 1$ to the constructor.

The `applyToVector` function is implemented very easily. Since the given transformation has the form $A_p \otimes I_{[m']}$, we just need to call `applySingleKroneckerDecomposedMatrix` from Section 4.1 for the template type `double` and input parameters `x`, `matrix_p_.get()`, `vec`, 1 and m'. Here `matrix_p_.get()` returns a normal pointer pointing to the same object as the unique pointer does.

4.5.3 Integer Transformations

First we recall all integer transformations we considered throughout this work. Definition 2.2.4 introduced an integer version of the Chinese remainder transform $\mathrm{CRT}_{m,q}$ where all entries are over $\mathbb{Z}_q$ for some given prime $q = 1 \mod m$. Since the sparse decompositions from Section 2.2.4 also apply for $\mathrm{CRT}_{m,q}$, the representation will be similar to the one of the complex case CRT_m, described in Section 4.5.1. Further in Proposition 2.3.4 we defined the integer transformation $L_m = \bigotimes_{l=0}^{s} L_{m_l} = \bigotimes_{l=0}^{s}(L_{p_l} \otimes I_{\lfloor m'_l \rceil})$, where m_l are prime powers, of some primes p_l, dividing m and $m'_l = m_l/p_l$. Finally, in Section 3.4 we saw that multiplication with the element g in the powerful and decoding basis can be performed via the application of some integer transformations, which we now refer to as $G_m^{\vec{p}} = \bigotimes_{l=0}^{s} G_{m_l}^{\vec{p}} = \bigotimes_{l=0}^{s}(G_{p_l}^{\vec{p}} \otimes I_{\lfloor m'_l \rceil})$ and $G_m^{\vec{d}} = \bigotimes_{l=0}^{s} G_{m_l}^{\vec{d}} = \bigotimes_{l=0}^{s}(G_{p_l}^{\vec{d}} \otimes I_{\lfloor m'_l \rceil})$. Additionally, we will also need the inverses of all integer transformations.[1]

By the properties of the Kronecker product, L_m, $G_m^{\vec{p}}$, $G_m^{\vec{d}}$ and their inverses do all have the same basic format

$$A_m = \bigotimes_{l=0}^{s} A_{m_l} = \bigotimes_{l=0}^{s}(A_{p_l} \otimes I_{\lfloor m'_l \rceil}).$$

Similar to the real transformations (see Section 4.5.2) A_m will be given as a list of transformations A_{m_l}, which is stored and managed by the field class. Further A_{m_l} and A_{p_l} will be represented by separate classes.

The Pure Prime Case

Each one of the matrices L_m, $G_m^{\vec{p}}$, $G_m^{\vec{d}}$ and their inverses can be applied in linear time using a specialized algorithm. If we represent all these matrices by the same class `MatrixZZ`, this class needs to know which particular matrix it represents, in order to call the correct application algorithm. Therefore we distinguish the different matrices by a class enum.

Listing 4.12: Definition of `MatrixZZ`

```
class MatrixZZ : public AbstractVectorTransformation<int>
{
public:
enum class MatrixType
{
   MATRIX_L,
   MATRIX_L_INVERSE,
   MATRIX_G_DECODING,
   MATRIX_G_INVERSE_DECODING,
   MATRIX_G_POWERFUL,
   MATRIX_G_INVERSE_POWERFUL
};

MatrixZZ(int dim, MatrixType matrixType);
MatrixZZ(MatrixZZ const& mzz);
```

[1] Note that the inverses of $G_{p_l}^{\vec{p}}$ and $G_{p_l}^{\vec{d}}$ are actually rational transformations, since they are scaled by a factor $1/p$. Hence, in a decoding procedure, we first scale them by p, apply them and then divide the result by p. The latter step should still result in integral values as long as decoding is successful.

```
~MatrixZZ();

inline MatrixType getMultType() const { return matrixType_; };

protected:
virtual void applyToVector(entry_type_vec& x, entry_type_vec const& vec)
    const;

private:
MatrixType matrixType_;

// specialized application algorithms
void gDec(entry_type_vec& x, entry_type_vec const& vec) const;
void gInvDec(entry_type_vec& x, entry_type_vec const& vec) const;
void gPow(entry_type_vec& x, entry_type_vec const& vec) const;
void gInvPow(entry_type_vec& x, entry_type_vec const& vec) const;
void applyL(entry_type_vec& x, entry_type_vec const& vec) const;
void applyLInverse(entry_type_vec& x, entry_type_vec const& vec) const;
};
```

The dimension $\varphi(p_l) = p_l - 1$ will be given on construction as well as the type of the matrix that is represented. The `applyToVector` function just needs to switch over the attribute `matrixType_` and call the respective application algorithm.

Implementation of the Specialized Application Algorithms

Recall that L_p is the lower unitriangular matrix containing only ones and its inverse has ones on the main diagonal and values -1 on the first lower diagonal. Thus L_p can be applied in linear time using successive sum and L_p^{-1} via a simple loop computing $x_i - x_{i-1}$, where $\mathbf{x} = (x_i)$ is some input vector.

The matrices $G_p^{\vec{d}}$ and $\left(G_p^{\vec{d}}\right)^{-1}$, for some prime p, are defined as the $[\varphi(p)] \times [\varphi(p)]$ matrices

$$G_p^{\vec{d}} = \begin{pmatrix} 2 & 1 & 1 & \dots & 1 \\ -1 & 1 & 0 & \cdots & 0 \\ 0 & -1 & 1 & \ddots & \vdots \\ \vdots & \ddots & & \ddots & 0 \\ 0 & \cdots & 0 & -1 & 1 \end{pmatrix} \text{ and } \left(G_p^{\vec{d}}\right)^{-1} = \frac{1}{p} \begin{pmatrix} 1 & 2-p & 3-p & \cdots & -1 \\ 1 & 2 & 3-p & \cdots & -1 \\ 1 & 2 & 3 & \cdots & -1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 2 & 3 & \cdots & p-1 \end{pmatrix}.$$

$G_p^{\vec{d}}$ can be applied similar to L_p^{-1}, but we reverse the loop and compute also the sum $\sum_{i=p-1}^{1} x_i$, which is used for the first entry. Concerning $\left(G_p^{\vec{d}}\right)^{-1}$ we will first apply the scaled version $p\left(G_p^{\vec{d}}\right)^{-1}$. Observe that the successive rows of $p\left(G_p^{\vec{d}}\right)^{-1}$ only differ in one entry, so we first compute the sum $x_0 + \sum_{i=1}^{p-1}(i+1-p)x_i$ for the first entry an then, in a second loop, successively add px_i. In a last step, we try to evenly divide the resulting vector by p. If this is not possible we throw an error, since decoding failure is detected.

To see how the application of $G_p^{\vec{p}}$ and $p(G_p^{\vec{p}})^{-1}$ can be realized, recall that $G_p^{\vec{p}}$ was defined

as

$$G_p^{\vec{p}} = J \cdot (G_p^{\vec{d}})^T \cdot J = J \cdot \begin{pmatrix} 2 & -1 & 0 & \dots & 0 \\ 1 & 1 & -1 & \ddots & \vdots \\ 1 & 0 & 1 & & 0 \\ \vdots & \vdots & \ddots & \ddots & -1 \\ 1 & 0 & \dots & 0 & 1 \end{pmatrix} \cdot J,$$

where J is the reversed identity matrix. Similarly $(G_p^{\vec{p}})^{-1}$ is given as

$$(G_p^{\vec{p}})^{-1} = J \cdot \frac{1}{p} \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 2-p & 2 & 2 & \dots & 2 \\ 3-p & 3-p & 3 & \dots & 3 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ -1 & -1 & -1 & \dots & p-1 \end{pmatrix} \cdot J.$$

Both multiplications with J are done via the `std::reverse` function, which just reverses the order of a given input vector. The application of $(G_p^{\vec{d}})^T$ is implemented straight forward. In each step we just have to compute $x_0 + x_i - x_{i+1}$ except for the last one, where it is $x_0 + x_{p-2}$. The application of $p((G_p^{\vec{d}})^T)^{-1}$ is also quite easy. Let $x'_0, \dots, x'_{p-2}$ be the entries of the resulting vector. First we compute the sum $S := \sum_{i=0}^{p-2} x_i$ and set $S_0 = S$. For $0 \leq i \leq p-3$ inductively define $S_{i+1} = S_i + S - px_i$ and set $x'_i = S_i$, which is best done by a for loop. It is easy to see that for $i \geq 1$ we have

$$S_i = (i+1-p)x_0 + \dots + (i+1-p)x_{i-1} + (i+1)x_i + \dots + (i+1)x_{p-2},$$

which is just the inner product of the input vector $\mathbf{x}$ and the i-th row of $p((G_p^{\vec{d}})^T)^{-1}$. As before we do now try to evenly divide by p, which is possible as long as no decoding failure occurs.

The Prime Power Case

Now we want to represent the integer transformations of the form $A_m = (A_p \otimes I_{[m']})$, where m is a power of some prime p and A_p will be an object of type `MatrixZZ` as defined above. This is done very similar to the real transformation case described in Section 4.5.2.

Listing 4.13: Definition of `TransformationZZ_PrimePower`

```
class TransformationZZ_PrimePower : public
    AbstractVectorTransformation<int>
{
public:
typedef AbstractVectorTransformation<int> base_type;

// the integer vector transformation types
enum class TransformationType
{
    INT_L,
    INT_L_INVERSE,
    INT_G_DECODING,
```

```
    INT_G_INVERSE_DECODING,
    INT_G_POWERFUL,
    INT_G_INVERSE_POWERFUL
};

TransformationZZ_PrimePower(int m, int p,
    TransformationType transformation);

~TransformationZZ_PrimePower();

protected:
virtual void applyToVector(entry_type_vec& x, entry_type_vec const& vec)
    const;

private:
std::unique_ptr<base_type const> matrix_p_;
};
```

The constructor takes the values m, p and an indicator for the represented integer transformation. The dimension is set to $\varphi(m) = \varphi(p)m' = (p-1)m/p$. Then, the unique pointer `matrix_p_`, which points to a `MatrixZZ` object, is constructed. Thereby, the indicator of the transformation and the dimension $\varphi(p) = p-1$ are passed to the constructor of `MatrixZZ`. (Note that the passed indicator is of another data type and has a slightly different name, but indicates the same transformation.)

The `applyToVector` function is implemented analogously to the real transformation prime power case. That is, `applySingleKroneckerDecomposedMatrix` from Section 4.1 for the template type `int` is called with input parameters `x`, `matrix_p_.get()`, `vec`, 1 and m'_l. As before, `matrix_p_.get()` returns a normal pointer pointing to the same object as the unique pointer does.

Representing $\mathrm{CRT}_{m,q}$

The only integer transformations that we cannot represent by the above classes are $\mathrm{CRT}_{m,q}$ and $\mathrm{CRT}_{m,q}^{-1}$ for prime powers m. Using the sparse decomposition from Section 2.2.4 we can rewrite $\mathrm{CRT}_{m,q}$ and $\mathrm{CRT}_{m,q}^{-1}$ as

$$\mathrm{CRT}_{m,q} = L_{m'}^{\varphi(m)} \cdot \left(I_{\mathbb{Z}_p^*} \otimes \mathrm{DFT}_{m',q}\right) \cdot \hat{T}_{m,q} \cdot \left(\mathrm{CRT}_{p,q} \otimes I_{[m']}\right)$$

and

$$\mathrm{CRT}_{m,q}^{-1} = \left(\mathrm{CRT}_{p,q}^{-1} \otimes I_{[m']}\right) \cdot \hat{T}_{m,q}^{-1} \cdot \left(I_{\mathbb{Z}_p^*} \otimes \mathrm{DFT}_{m',q}^{-1}\right) \cdot \left(L_{m'}^{\varphi(m)}\right)^{-1},$$

where $\mathrm{DFT}_{m',q}$, $\hat{T}_{m,q}$ and $\mathrm{CRT}_{p,q}$ are defined as in Proposition 2.2.16 with $\omega_m, \omega_{m'}$ and ω_p being roots of unity in $\mathbb{Z}_q$. Note that we omitted the permutation $L_{m'}^{\varphi(m)}$ in the complex case, since the complex CRT transformations are applied in a way such that the permutation is always canceled out by its inverse. Concerning the integer CRT transformations, this is no longer the case. We switch between the bases $\vec{c}$ and $\vec{p}$ using these transformations and usually one switches only in one direction. Thus, the permutation is necessary and we need to take care of it.

In the following we present two classes, one representing the CRT transformation in its sparse decomposition and the other representing the matrices inside the decomposition. An

exception is made by the twiddle matrix, which is a diagonal matrix and hence we represent it by a single vector.

We start with the second class, which we call `MatrixZZq`, since all matrices represented by it are over $\mathbb{Z}_q$. In this class we store a unique pointer to a square matrix over $\mathbb{Z}_q$. For this matrix we use the class `mat_ZZ_p` from the NTL library [Sho15]. The NTL library is a free C++ library under the GNU General Public License for doing number theory. The class `mat_ZZ_p` represents matrices over $\mathbb{Z}_p$ for some arbitrary prime p. Further it provides some standard matrix features like multiplication modulo p and transposition as well as a particular nice feature, inversion modulo p. Our class `MatrixZZq` is just an interface between the NTL and our program. We do also provide inversion modulo p, which is then used to pre-compute $\mathrm{CRT}_{p,q}^{-1}$ and $\mathrm{DFT}_{m',q}^{-1}$.

Listing 4.14: Definition of `MatrixZZq`

```
class MatrixZZq : public AbstractVectorTransformation<int>
{
public:
typedef NTL::mat_ZZ_p matrix_type;

MatrixZZq(matrix_type const& entries);
MatrixZZq(MatrixZZq const& matrix);

~MatrixZZq();

void invert();

protected:
virtual void applyToVector(entry_type_vec& x, entry_type_vec const& vec)
    const;

private:
std::unique_ptr<matrix_type> matrix_;
};
```

The `invert` function is just a forwarding to the NTL inversion applied to the stored matrix. This means that the object behind `matrix_` will be overwritten and the class then represents the inverse matrix. In the application algorithm we have to translate the input vector to the NTL version. The NTL has his own data types for integers (`ZZ`) and integers modulo p (`ZZ_p`) as well as for vectors and matrices over these ground types. Since our matrix is of type `mat_ZZ_p`, we have to convert the input vector to a `vec_ZZ_p`, multiply it with the matrix via the NTL and convert the result back to a vector of integers. The NTL provides an overload of the assignment operator for the types `ZZ_P` and `long`. Hence, we can convert the input vector coordinate-wise into a `vec_ZZ_p`. The other direction is a bit more messy. We will also proceed component-wise but unfortunately there is no direct access to the `long` object that lies behind a `ZZ_P` object. Therefore, we have to do this manually via the `rep` function returning the representative of an element. Eventually the returned representative will be an array of `long` objects, which stores the desired element at its second position, i.e., at the index 1.

The implicit conversion between `int` and `long` that is performed several times, should never be a problem for us. In all applications the used numbers are small enough to be rep-

resented by the type `int`.

Next we present the first class from above, i.e., the class representing $\mathrm{CRT}_{m,q}$ in its decomposed form. We call the class `TransformationZZqCrt`, because it represents the CRT transform over $\mathbb{Z}_q$.

Listing 4.15: Definition of `TransformationZZqCrt`

```
class TransformationZZqCrt : public AbstractVectorTransformation<int>
{
public:
typedef AbstractVectorTransformation<int> base_type;

TransformationZZqCrt(MatrixZZq const& DFT, MatrixZZq const& CRT,
    std::vector<NTL::ZZ_p> const& twiddleMatrix, bool const reversed);

~TransformationZZqCrt();

protected:
virtual void applyToVector(entry_type_vec& x, entry_type_vec const& vec)
    const;

private:
void permutationL_M_D(entry_type_vec& x, entry_type_vec const& a, int d)
    const;

bool reversed_;
std::unique_ptr<base_type const> DFT_;
std::unique_ptr<base_type const> CRT_;
std::unique_ptr<std::vector<NTL::ZZ_p> const> twiddleMatrix_;
};
```

The pointers `DFT_` and `CRT_` point to the actual matrices $\mathrm{DFT}_{m',q}$ and $\mathrm{CRT}_{p,q}$, which will be of type `MatrixZZq`. Further, the diagonal twiddle matrix is represented by a single vector of `ZZ_p` objects. The boolean `reversed_` indicates whether the represented transformation is $\mathrm{CRT}_{m,q}$ (`false`) or $\mathrm{CRT}_{m,q}^{-1}$ (`true`). The name of this attribute is due to the fact, that the single matrices in the sparse decomposition are applied in reversed order. The permutation function can be implemented straight forward following the definition of the stride permutation from Proposition 2.2.15. Note that we apply only $L^{\varphi(m)}_{m'}$ and $\left(L^{\varphi(m)}_{m'}\right)^{-1}$, where the parameter m' is given as the input `d` and $\varphi(m)$ is given implicitly, since the input vector `vec` has exactly the length $\varphi(m)$. Moreover, since $\varphi(m) = \varphi(p) \cdot m'$, using the definition one can verify that

$$\left(L^{\varphi(m)}_{m'}\right)^{-1} = L^{\varphi(m)}_{\varphi(p)},$$

so a call of the permutation function with input $\mathtt{d} = \varphi(p)$ performs the inverse stride permutation.

Finally, the implementation of the `applyToVector` function again uses the application functions for Kronecker decompositions from Section 4.1. First, note that we have implicit access to $\varphi(p) = \dim(\mathrm{CRT}_{p,q})$ and $m' = \dim(\mathrm{DFT}_{m',q})$. If the boolean `reversed_` is `false`, the represented transformation is $\mathrm{CRT}_{m,q}$ and according to the sparse decomposition from Proposition 2.2.16 we first apply the matrix $(\mathrm{CRT}_{p,q} \otimes I_{[m']})$, which is done by a call

of `applySingleKroneckerDecomposedMatrix` with input parameters `x`, `CRT_.get()`, `vec`, 1 and m'. Then, we compute the inner product of the vector representing the twiddle matrix and `x`, where all arithmetic is modulo q. Next, the matrix $(I_{\mathbb{Z}_p^*} \otimes \mathrm{DFT}_{m',q})$ is applied, i.e., we call again `applySingleKroneckerDecomposedMatrix`, but this time with input parameters `x`, `DFT_.get()`, `x`, $\varphi(p)$ and 1. In a last step, we apply the permutation $L^{\varphi(m)}_{m'}$ to `x`. If, on the other hand, the boolean `reversed_` equals `true`, then the represented transformation is $\mathrm{CRT}^{-1}_{m,q}$. Now we start with the permutation $L^{\varphi(m)}_{\varphi(p)}$ and then make the exact same multiplications as before, but in reversed order.

4.6 The Class `RingLweCryptographyField`

In the following we will describe our class that represents the field $K = \mathbb{Q}(\zeta_m)$ for some positive integer m. The name of this class is `RingLweCryptographyField` (short: `rlweField`) and it is a rather "static" class, in the sense that it is instantiated once at the beginning of an application and does never change throughout the whole process. All elements that we consider live in the field K and we reflect this in our class structure. Once the field class is instantiated, all elements we construct get a pointer to this field. In this way, the field class can manage all vector transformations we consider, which are dependent on the input m, and an element can access them whenever needed. The considered transformations are pre-computed on construction and stored in the `rlweField` class. Besides this main feature, the field class can also construct some special elements, like zeros and ones in different bases or the elements t and g from Definition 1.4.21.

4.6.1 Member Variables

Let us first take a look at the member variables `rlweField` is equipped with. First we store some constants, namely the integer m, the dimension $n = \varphi(m)$ and the radical $\mathrm{rad}(m)$. In our applications of the toolkit we work a lot with elements whose coordinates are over $\mathbb{Z}_q$, where the r-LWE modulus q is some prime integer such that $q = 1 \mod m$. We also store this modulus in our field class.

As already mentioned in Section 4.5, all vector transformations we consider are given as a Kronecker decomposition $A_m = \bigotimes_{l=0}^{s} A_{m_l}$ and we will store this decomposition in a list. This is done by the field class, so for each transformation we store a unique pointer to a list containing unique pointers to objects of type `AbstractVectorTransformation` as defined in Section 4.5.

Finally, we store some coordinate vectors for special elements, which are pre-computed on construction. To be more precise, we store the coordinates of t in the decoding basis, these of g in the CRT basis and the coordinates for ones in different bases, which are held in a map, where the basis is the key and the coordinates the value. Note that we cannot construct any elements $a \in K$ at the pre-computation stage, since such an element would need a pointer to the field, which is still under construction at this stage. This is why we store the coordinate vectors instead and create the particular element only if we need it.

All together we declare the following member variables.

Listing 4.16: Member Variables of `RingLweCryptographyField`

```
class RingLweCryptographyField : public
    std::enable_shared_from_this<RingLweCryptographyField>
{
private:
unsigned int const m_;
unsigned int const n_;
unsigned int rad_m_;
unsigned int const rlwe_modulus_;

std::unique_ptr<std::vector<int>> t_inverse_decoding_coords_;
std::unique_ptr<std::vector<int>> g_crt_coords_;
std::unique_ptr< std::map<RingElement::Basis,
    std::unique_ptr<std::vector<int>>> > ones;

template<typename T>
   using avt_uptr = std::unique_ptr<AbstractVectorTransformation<T>>;
template<typename T>
   using avt_list_uptr = std::unique_ptr<std::list<avt_uptr<T>>>;

// Integer transformations
avt_list_uptr<int> Lm_;
avt_list_uptr<int> LmInverse_;

avt_list_uptr<int> crtMq_;
avt_list_uptr<int> crtMqInverse_;

avt_list_uptr<int> mgd_; // multiplication with g in decoding basis.
avt_list_uptr<int> dgd_; // multiplication with g^-1 in decoding basis.
avt_list_uptr<int> mgp_; // multiplication with g in powerful basis.
avt_list_uptr<int> dgp_; // multiplication with g^-1 in powerful basis.

// Complex transformations
avt_list_uptr<std::complex<double>> crtM_;
avt_list_uptr<std::complex<double>> crtMInverse_;
avt_list_uptr<std::complex<double>> crtMStar_;
avt_list_uptr<std::complex<double>> crtMStarInverse_;

// Real transformations
avt_list_uptr<double> sample_gauss_D_;
avt_list_uptr<double> gs_decomp_U_;
avt_list_uptr<double> gs_decomp_D_;
}
```

The meaning of the inheritance from `std::enable_shared_from_this` will be explained later.

4.6.2 Constructor

We provide only one constructor for this class. In order to instantiate a specific field we need two informations, the integer m and the ring-LWE modulus q. All other pre-computations can be done using these values.

Listing 4.17: Constructor of `RingLweCryptographyField`

```
class RingLweCryptographyField : public
    std::enable_shared_from_this<RingLweCryptographyField>
{
public:
RingLweCryptographyField(unsigned int m, unsigned int modulus);

~RingLweCryptographyField();
}
```

The constructor initiates the variables `m_`, `n_` and `rlwe_modulus_` to `m`, $\varphi(\mathtt{m})$ and `modulus`, respectively. Thereby the Euler totient $\varphi(\mathtt{m})$ is computed via a function, which is provided by a separate mathematics utility file (see Section 4.3). The variable `rad_m_` is set to 1 and will be further manipulated throughout the pre-computations, which are triggered by a call of a private init-function.

4.6.3 Available Functions

Next to pre-computation, the field class has two main features, the construction of special elements and the application of vector transformations. Besides some standard getter functions we provide procedures for the construction of the special elements g and t^{-1}, for zeros and ones in different bases as well as for functions for the application of an integer, real or complex transformation to some given vector. We indicate the different transformations we consider by a class enum.

Listing 4.18: Available Functions of `RingLweCryptographyField`

```
class RingLweCryptographyField : public
    std::enable_shared_from_this<RingLweCryptographyField>
{
public:
enum class TransformationMatrices
{
   COMPLEX_CRT_M,
   COMPLEX_CRT_M_INVERSE,
   COMPLEX_CRT_M_STAR,
   COMPLEX_CRT_M_STAR_INVERSE,
   INTEGER_L_M,
   INTEGER_L_M_INVERSE,
   INTEGER_CRT_MQ,
   INTEGER_CRT_MQ_INVERSE,
   INTEGER_MULT_G_DEC,
   INTEGER_DIV_G_DEC,
   INTEGER_MULT_G_POW,
   INTEGER_DIV_G_POW,
   REAL_SAMPLE_GAUSS_D,
   REAL_GS_DECOMP_U,
   REAL_GS_DECOMP_D
};

unsigned int getDimension() const;
unsigned int getModulus() const;
```

```
unsigned int getM() const;
unsigned int getRadM() const;

RingLweCryptographyElement getElementT_inverse() const;
RingLweCryptographyElement getElementG() const;
RingLweCryptographyElement getZero(RingLweCryptographyElement::Basis
    basis) const;
RingLweCryptographyElement getOne(RingLweCryptographyElement::Basis
    basis) const;

void applyIntTransformation(TransformationMatrices transformation,
    coordinate_vec& x, coordinate_vec const& a) const;
void applyRealTransformation(TransformationMatrices transformation,
    real_vec& x, real_vec const& a) const;
void applyCompTransformation(TransformationMatrices transformation,
    comp_vec& x, comp_vec const& a) const;
}
```

The type of the special elements is `RingLweCryptographyElement`, which is our class for the representation of elements in K. Its constructor takes a shared pointer to this field class, an indicator of the basis in which the element is represented, a coordinate vector with respect to the indicated basis and two optional integer values, which we omit for the moment. See Section 4.7 for a detailed description and further explanations concerning the functionality of the element class. When constructing a 0 or 1 in K, the basis indicator is given by the input parameter. Clearly, the coordinate vector for zeros is for all bases the zero vector. If we construct a $1 \in K$, the coordinate vector in the specific basis is not always trivial, but can be pre-computed as described in Section 3.5.2 and 4.6.4. We can access the pre-computed coordinate vectors via the map `ones`, where the basis indicators work as keys. The coordinate vectors for g and t^{-1} are also pre-computed, but already with respect to a specific basis, namely the Chinese remainder basis $\vec{c}$ for g and the decoding basis $\vec{d}$ for t^{-1}. All element constructions still lack the shared pointer to this field class. At this stage it becomes important that `RingLweCryptographyField` inherits from `std::enable_shared_from_this`. This provides a function, which returns a shared pointer similar to the `this` pointer, and takes care of the consistency of the object counter each shared pointer has. That is, if we already created a shared pointer to the instance of `RingLweCryptographyField`, the object counter will be updated correctly. Otherwise, a new object counter will be initiated. It would also be possible to create a shared pointer directly from the `this` pointer. However, this can lead to inconsistencies in the actual object count.

Recall that the field class stores a unique pointer to a list of abstract vector transformations for each transformation A_m we consider. This list represents the Kronecker decomposition $A_m = \bigotimes_{l=0}^{s} A_{m_l}$, where m_l are the different prime powers dividing m. If we call one of the three application functions, `applyIntTransformation`, `applyRealTransformation` or `applyComplexTransformation`, we have to pass an indicator for the specific transformation we want to apply. The application is done via `applyKroneckerDecomposition` from Section 4.1, using the input `x` and `vec`, as well as the list for the indicated transformation. The latter is chosen via a switch statement over the indicator. If we call an application function with a mismatching indicator, for example `applyIntTransformation` with the indicator `COMPLEX_CRT_M`, nothing will happen. One specialty arises when we divide by g, i.e., when we use the indicators `INTEGER_DIV_G_DEC` or `INTEGER_DIV_G_POW`. While

computing the result, we could detect decoding failure causing an error to be thrown. This error will be forwarded and eventually handled by the decoding procedure.

4.6.4 Pre-Computation on Construction

As mentioned above, the constructor calls a private init-function, which starts the pre-computations. We have to pre-compute the following values and objects:

(i) All the vector transformations we consider,

(ii) the radical $\mathrm{rad}(m)$,

(iii) the coordinates of the neutral element in the different bases,

(iv) the coordinates of g in the CRT basis and these of t^{-1} in the decoding basis.

Recall that the vector transformations are given in the form $A_m = \bigotimes_{l=0}^{s} A_{m_l}$, where m_l are the different prime powers dividing m. Thus, to start things off, we have to compute the prime power factorization of m. This is done by a function from the mathematics utilities from Section 4.3, which computes a list of pairs (p_l, k_l) such that $m_l = p_l^{k_l}$ for primes p_l and integers k_l. Consequently, given such a pair, we can compute the values $m'_l = p_l^{k_l - 1}$, $m_l = p_l \cdot m'_l$ and $n_l = \varphi(m_l) = (p_l - 1) \cdot m'_l$. Once we initialized all the vector transformation lists as empty lists, we can start a loop over the prime power factorization. Inside this loop we update `rad_m_` to `rad_m_` $\cdot\, p_l$, so that after the loop `rad_m_` is indeed the product of all prime divisors of m. Further, we compute the specific transformations for the prime powers m_l. This step is divided into three parts, i.e., we provide one pre-computation function for each type (integer, real and complex) of transformations we consider.

Pre-Compute the Transformations

For better readability we drop the index $l \in [s+1]$, i.e., m is now a prime power of some prime p and $m' = m/p$.

Integer Transformations. Following the class structure of the integer transformations as described in Section 4.5.3, there are two different ways how they are constructed. The "easier" way applies for L_m, $G_m^{\vec{p}}$, $G_m^{\vec{d}}$ and their inverses. For each of the latter transformations we construct a unique pointer to a `TransformationZZ_PrimePower` object. The constructor of `TransformationZZ_PrimePower` takes the values m and p, and an indicator for the represented transformation. After construction, each unique pointer is added to the respective list.

Concerning $\mathrm{CRT}_{m,q}$ and $\mathrm{CRT}_{m,q}^{-1}$ we have to put some more effort into computations. The constructor of `TransformationZZqCrt` (cf. Listing 4.15) takes two matrices of type `MatrixZZq` (cf. Listing 4.14), a vector of `NTL::ZZ_p` objects and a boolean. The constructor of `MatrixZZq` on the other hand, takes an object of type `NTL::mat_ZZ_p`. So first we construct the two NTL matrix objects for $\mathrm{CRT}_{p,q}$ and $\mathrm{DFT}_{m',q}$, which are of dimension $p-1$ and m', respectively, and a `NTL::ZZ_p` vector of size n, representing the twiddle matrix. In order to compute the entries for these objects, we need the roots of unity $\omega_p, \omega_{m'}$ and ω_m which are all over $\mathbb{Z}_q$. These roots are computed via the `findElementOfOrder` function from the mathematics utilities (see Section 4.3). Now we can compute the entries according

to the definitions from Section 2.2.4, where we use the `NTL::power` function for fast modulo exponentiation. Having this done, we can continue with the construction of the `MatrixZZq` objects, which are then passed, together with the `NTL::ZZ_p` vector and the boolean `false`, to the constructor of the unique pointer to the `TransformationZZqCrt` object. We add this pointer to the list for the CRT transform. Since all objects are copied in the latter construction, we can now invert them without any side effects. That is, the `MatrixZZq` objects are inverted via their build in function and the vector entries are component-wise inverted modulo q. Finally we repeat the above construction, where this time the boolean `true` is passed, as we construct CRT_m^{-1}, and add the pointer to the list for the inverse CRT transform.

Real Transformations. For the construction of the real transformations we have to construct unique pointers to objects of type `TransformationRealPrimePower`. The constructor of `TransformationRealPrimePower` just asks for the values p and m, and for an indicator of the respective real transformation that is represented. We call the constructor for each suitable indicator and add the pointer to the respective list.

Complex Transformations. The construction of the complex transformations works exactly as the one for the real transformations. Now, the unique pointers point to objects of type `TransformationCompCrt`. Again we call the constructor with p, m and one of the suitable indicators for complex transformations. Then, the unique pointer is added to the respective list.

Special Coordinate Vectors

After the pre-computation of the transformations is finished, we can continue with the coordinate vectors for the special elements. In Section 3.5.2 we described how the coordinate vectors of the neutral element in the different bases can be computed. The implementation follows these instructions closely, where we use the already computed transformations to switch between the bases.

Section 3.5.1 introduced an algorithm which computes the coordinates of g in the CRT basis. Actually, the algorithm works in the powerful basis and only switches the basis as a very last step. Since we do not need g in the powerful basis, we perform all operation in the CRT basis saving some performance. Recall that for each $l \in [s+1]$ we have to create elements with coordinate vectors that are 1 at the index $m'_l \cdot \ \varphi(m_{l+1}) \cdots \varphi(m_s)$, where $m_0, \ldots, m_s$ are the distinct prime power divisors of m. We already computed these prime power factors and can pass them to this computation. Then the indexes $m'_l \cdot \ \varphi(m_{l+1}) \cdots \varphi(m_s)$ are easiest computed, if we traverse the list of prime power divisors in reverse order. Start with a variable $i = 1$. Since each m_l is given in the form $m_l = p_l^{k_l}$, we can easily compute $m'_l = p_l^{k_l - 1}$ and $\varphi(m_l) = (p_l - 1) \cdot m'_l$. The first index ($l = s$) is given by $m'_s \cdot i = m'_s$, which we use to create the element $\zeta_{p_s} \in R_q$ and update $g = g \cdot (1 - \zeta_{p_s})$, where g was initialized as $1 \in R_q$. Next, we update $i = i \cdot \varphi(m_s)$ and proceed with the next step in the loop. Now, the index is $m'_{s-1} \cdot i = m'_{s-1} \cdot \varphi(m_s)$ and we can create the next element to update g. Inductively we will get $i = \varphi(m_{l+1}) \cdots \varphi(m_s)$ and indexes $m'_l \cdot \ \varphi(m_{l+1}) \cdots \varphi(m_s)$ at the l-th step in the loop, for $s \geq l \geq 0$. At the end of the loop, g will be the desired element.

As the last step in pre-computations, we construct the coordinate vector of t^{-1} in the decoding basis. This is done very easily. Note that by linearity, $t^{-1} \in R^\vee$ has the exact same coordinates in the powerful basis $t^{-1}\vec{p}$ as $1 \in R$ in $\vec{p}$. These coordinates are given by

$(1, 0, \ldots, 0)$. Applying the transformation L_m^{-1} from Proposition 2.3.4 yields the coordinate vector with respect to $\vec{d}$.

4.7 The Class `RingLweCryptographyElement`

The representation of elements $a \in K = \mathbb{Q}(\zeta_m)$ plays a central role in our program. Such elements are represented by the class `RingLweCryptographyElement`, which, roughly speaking, keeps track of the coordinates and the basis specifying a certain element. Also it has a "lives in" relation to the class `RingLweCryptographyField` representing K (see. Section 4.6). All the functions and algorithms that we developed throughout this work are available for this class. In the following we describe how this class works and how we realized the features of the toolkit.

4.7.1 Member Variables

Let us first take a look at the member variables of `RingLweCryptographyElement`. As described in the beginning of this chapter, we prefer to use the `std::vector` class for the representation of the coordinates. Instead of the vector itself, we store a unique pointer to the vector. A shared pointer to an `RingLweCryptographyField` object represents the field K in which the element lives. Next, we need a useful representation of the regarded basis. In all computations that we make, it is not important how the regarded basis actually looks like. We only need to know which basis we are looking at. Thus, it is sufficient to indicate each basis of our interest and equip our class with such an indicator. This works only if we regard a small finite amount of different basis indicators. We can achieve this, if we separate the power k for bases like $t^{-k}\vec{c}$ or $t^{-k}\vec{p}$. Unfortunately, this is still not sufficient to uniquely determine the ideal in which the element lives. For example, right now we cant distinguish R and R_q. Therefore we need another variable for q, indicating if computations are done modulo q or not.

These considerations lead to the following member variables

Listing 4.19: The Member Variables of `RingLweCryptographyElement`

```
class RingLweCryptographyElement
{
public:
enum class Basis
{
   BASIS_POWERFUL,
   BASIS_T_INVERSE_POWERFUL,
   BASIS_CRT,
   BASIS_T_INVERSE_CRT,
   BASIS_DECODING
};
private:
std::shared_ptr<RingLweCryptographyField const> field_;
std::unique_ptr<std::vector<int>> coordinates_;
Basis basis_;
unsigned int primeModulusQ_;
unsigned int k_;
}
```

Note that k can only be greater or equal to one, if the given basis actually represents $R^\vee$. Otherwise we make sure, that k is always zero.

4.7.2 Constructors

Having all member variables set, we can move on to the constructors. First we observe that, since all elements we create live in some field K, it is not useful to have a default constructor taking no arguments as input. The least we have to know is the field K. For later usage of the program it will be convenient to have a constructor that instantiates a sort of default element in a given field. To be more precise, given a shared pointer to the field K, this constructor actually creates the zero element in R, i.e., sets the basis to $\vec{p}$, the coordinate vector to $0 \in \mathbb{Z}^{[n]}$ and the values q and k to zero. Furthermore, we have a constructor that creates an element, given a shared pointer the field K, the coordinate vector, the basis and two optional arguments for the modulus q and the power k, which have the default values $q = 0$ and $k = 1$. The ideal power k is always set via an extra setter-function, which makes sure that $k = 0$, if the basis is $\vec{p}$ or $\vec{c}$ (in this cases, the represented element lives in the ideal R, where the power k has no meaning). For later purposes it is also useful the have a copy constructor. Summarizing this, we have the following constructor overloads

Listing 4.20: Class Constructors of `RingLweCryptographyElement`

```
class RingLweCryptographyElement
{
public:
typedef std::shared_ptr<RingLweCryptographyField const> rlwe_field_sptr;

// ``Default'' constructor
RingLweCryptographyElement(rlwe_field_sptr field);

// Constuctor for known elements
RingLweCryptographyElement(rlwe_field_sptr field,
   Basis basis,
   std::vector<int>& coordinates,
   unsigned int primeModulusQ = 0,
   unsigned int idealPower = 1);

// Copy constructor
RingLweCryptographyElement(RingLweCryptographyElement const& elmt);

// Default destructor
~RingLweCryptographyElement();
}
```

4.7.3 Functions

Throughout this work we developed several kinds of tasks our program has to handle. A lot of the features of the toolkit will be provided by our element class. For example, we have the basic arithmetic in K and algorithmic tasks like decoding and discretizing. In the following we give a code snippet with all available functions in our class header. Then each group of functions is explained in more detail.

Note, that in general we designed our procedures in a way, that they do not have any return value, but instead manipulate one of the input parameters. For example, the procedure for addition takes three argument, x, a and b, so that $x = a + b$. In this way, the user can decide whether he uses a new element for the result, recycles and old one, or takes one of the input elements for the output.

Listing 4.21: Available Functions of `RingLweCryptographyElement`

```
//For better raedability
typedef RingLweCryptographyElement RingElement

class RingLweCryptographyElement
{
public:
//Changing the basis
bool changeBasisTo(Basis newBasis);

//Additional operator overloads
RingElement& operator=(RingElement const& elmt);
RingElement& operator+=(RingElement const& elmt);
RingElement& operator-=(RingElement const& elmt);
RingElement& operator*=(RingElement const& elmt);
RingElement& operator*=(const int scalar);
}

//Comparison
bool operator==(RingElement const& lhs, RingElement const& rhs);
bool operator!=(RingElement const& lhs, RingElement const& rhs);

//Adition
RingElement operator+(RingElement const& a, RingElement const& b);
RingElement operator-(RingElement const& a, RingElement const& b);
RingElement operator-(RingElement const& a);
bool add(RingElement& x, RingElement const& a, RingElement const& b);
bool sub(RingElement& x, RingElement const& a, RingElement const& b);
bool neg(RingElement& x, RingElement const& a);

//Multiplication
RingElement operator*(RingElement const& a, RingElement const& b);
RingElement operator*(int const scalar, RingElement const& b);
bool mult(RingElement& x, RingElement const& a, RingElement const& b);
bool multWithScalar(RingElement& x, int const scalar,
   RingElement const& a);

//Special multiplication
bool multWithT(RingElement& x, RingElement const& a);
bool multWithT_Inverse(RingElement& x, RingElement const& a);

//Decoding
void decode(RingElement& x, RingElement const& a);
bool computeUniqueRepresentative(RingElement& x, RingElement const& a);
```

```
//Discretizing
void discretize(RingElement& x,
   std:.vector<double>& realCoordinateVector,
   RingElement const& cosetRepresentative,
   int const scalingFactor = 1);

//Sampling
void sampleGaussiansInK_RR(real_vec& x,
   RingLweCryptographyField const& field,
   double mean,
   double stddev);

void sampleDiscretizedGaussian(RingElement& x,
   double mean,
   double stddev,
   RingElement const& cosetRepresentative,
   int const scalingFactor = 1);

void sampleDiscreteGaussianInR(RingElement& x,
   RingElement const& c,
   real_type stddev);

int sampleDiscreteGaussianInZZ(double stddev,
   double center,
   double funcValue = 3);

void sampleUniformlyCoordVec(RingElement& x, int lb, int ub);
```

Changing the basis

In Sections 2.2.2 and 2.3.2 we saw how we can efficiently switch between the different bases we deal with. On the one hand, we have the relation between the powerful basis $\vec{p}$ and the Chinese remainder basis $\vec{c}$, and, on the other hand, between the decoding basis $\vec{d}$ and $t^{-1}\vec{p}$. In all cases, the coordinate vector of the element is transformed by some transformation matrix. These transformations are managed by the field class. The element class provides two private functions, one that switches the representation between $\vec{p}$ and $\vec{c}$, and another one that does the same for $\vec{d}$ and $t^{-1}\vec{p}$.

Listing 4.22: Basis Swapping

```
class RingLweCryptographyElement
{
private:
bool swapBasesPowerfulAndCrt(bool direction);
bool swapBasesTInversePowerfulAndDecoding(bool direction);
}
```

The direction is oriented at the function name, i.e., `swapBasesPowerfulAndCrt(true)` switches $\vec{p}$ to $\vec{c}$, `swapBasesTInversePowerfulAndDecoding(true)` from $t^{-1}\vec{p}$ to $\vec{d}$ and both in the opposite direction for the input `false`. The returned boolean indicates the success of the swap. Both functions call the needed transformation from the field class. We can use them internally for basis swapping, if we already know the current basis of the element.

The `changeBasisTo` routine is used for a change of bases, if we do not know the current basis. As the input parameter it takes the basis, in which we want to represent the element. Then, it checks the current basis and, if possible, changes the bases via the above mentioned private functions. If the desired change is not possible, an error is thrown.

Addition

The core function for addition is `bool add(x,a,b)`, which takes two elements a and b, computes the sum $a + b$ and stores the result in x, i.e., $x = a + b$. Furthermore, it returns a boolean, indicating if addition succeeded or failed. Reasons for a failure could be an input with inconsistent attributes, like an element with basis $\vec{c}$ but a modulus $q = 0$.

As we saw in Section 3.3.1, we have to distinguish between addition modulo some integer q and normal addition. Thus `add(x,a,b)` uses two private subroutines, `addModulo(x,a,b)` and `addNonModulo(x,a,b)`, with the same input parameters. In the modulo case, all additions are performed coordinate-wise, we only might have to switch the basis of one element to match both bases. The basis switching is done via the sub routines described above. The two involved bases are compared via a switch statement over the basis attribute. The non modulo case works similarly, but addition might be performed via the canonical embedding σ. We describe later how this works in detail.

All other functions, like `sub` or `neg` and the respective operators, just call `add` with suitable parameters.

Multiplication

Concerning multiplication we have two separate functions, `bool mult(x,a,b)` and `bool multWithScalar(x,s,a)`, both storing the result in x and returning a boolean indicating failure or success, similar to the addition routine. Since the scalar s is given as an integer, `multWithScalar` just needs to multiply the coordinate vector with s, depending on the modulus either modulo q or not. On the other hand, the normal multiplication function `mult` works similarly to the addition function. That is, we distinguish between the modulo q and the normal case, which are both handled in separate subroutines with the same input parameters, namely `multModulo(x,a,b)` and `multNonModulo(x,a,b)`. As described in Section 3.3.1, in the modulo case, we just need to check the involved bases via a switch command over the basis attributes of a and b, adjust those bases properly, i.e., switch them always to the CRT basis, and then multiply the respective coordinate vectors component-wise modulo q. Further, we have to compute the ideal power for x, which is just the sum $k_1 + k_2$, where k_1 and k_2 are the ideal powers of a and b respectively.

The non modulo case works analogously. The involved bases are compared, maybe swapped, and the elements are multiplied. Here we cannot simply multiply the coordinate vectors coordinate-wise. Instead the vectors are multiplied via the canonical embedding σ, i.e., via CRT_m (see Section 2.2.3).

Special Multiplication

Following the considerations from Section 3.3.2 we provide separate functions for the multiplication with t and t^{-1}. The procedure `bool multWithT(x,a)` returns `false`, if the basis is $\vec{c}$ or $\vec{p}$, since we do not provide multiplication of elements in R with t. Further, if the basis is $\vec{d}$, it is changed to $t^{-1}\vec{p}$, and unchanged otherwise. Then the basis is manually set to

the respective counterpart without the factor t^{-1}, i.e., $t^{-1}\vec{p}$ to $\vec{p}$ and $t^{-1}\vec{c}$ to $\vec{c}$, and the ideal power k is set to zero.

The function `bool multWithT_Inverse(x,a)` works somewhat conversely. In the cases where the basis is $\vec{p}$, we manually set the basis to $t^{-1}\vec{p}$ and then change it to $\vec{d}$, since the decoding basis is somehow the default basis for $R^\vee$. If the basis is $\vec{c}$, it is set to $t^{-1}\vec{c}$. In both cases the ideal power k is set to one. In the remaining cases where the basis represents $R^\vee$ or $R_q^\vee$, the element a is multiplied with t^{-1} in the usual way. Thereby t^{-1} is pre-computed and provided by our field class.

Decoding

The decoding procedure follows closely the descriptions from Section 3.4. As a subroutine we use the function `computeUniqueRepresentative`, which represents the map $[\![\cdot]\!]$ for elements with coordinate vectors over $\mathbb{Z}_q$. Recall that, regarding the map coordinate-wise, for $a \in \mathbb{Z}_q$ we want to output the unique representative $[\![a]\!] \in \mathbb{Z} \cap [-q/2, q/2)$. Thus the subroutine checks for a correct input and then subtracts q from every entry that is greater than $(q-1)/2$, where the last quotient is an integer division, i.e., the result is rounded in case that $q-1$ is odd. Clearly, the coordinate vector has entries in $\mathbb{Z} \cap [-q/2, q/2)$ and represents the same element after this procedure.

The decoding function just checks if k equals 0 or 1 or is greater than 1. If $k = 0$, the represented element cannot be in $R^\vee$ or some power of it and consequently cannot be decoded. The rest of the implementation is more or less straight forward. The only thing we have to keep in mind is that we do not know the value of k in advance and therefore cannot pre-compute g^{k-1}. Thus if $k > 1$ and the element is represented in the CRT basis, we first have to compute g^{k-1} using the pre-computed element g. Similarly, in the other bases, using the respective integer transformation equals only one multiplication with g and we might have to repeat this step. Further, if $q > 0$, all arithmetic is modulo q. At this stage we compute the modulo arithmetic manually, since the integer transformations for multiplication with g are not per se transformations over $\mathbb{Z}_q$, but rather over $\mathbb{Z}$.

Discretizing

Given a $\mathbb{Z}$-basis $\vec{b}$ of R or $R^\vee$, an element $x = \left\langle \vec{b}, \mathbf{x} \right\rangle \in K_\mathbb{R}$ and a coset representative $c = \left\langle \vec{b}, \mathbf{c} \right\rangle$, where $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{c} \in \mathbb{Z}^n$, discretization is the task to compute an element y, with coordinates over $\mathbb{Z}$ such that the distance of x and y is small. In Section 2.4 we explain how such an element can be computed efficiently. The presented method includes some randomness, which we get from the C++ standard library. We use the default random engine, instantiated with a random device, to sample elements from different distributions. The computations presented in Section 2.4 can all be done coordinate-wise. For the discretization function, the coordinates $\mathbf{x}$ are given as a vector of doubles and $\mathbf{c}$ is a vectors of integers. We can perform coordinate-wise manipulations of two vectors by the `std::transform` algorithm, using a locally defined function.

The computation of the coordinates for y works as follows. First, we compute the decimal part of $\frac{1}{p}(c_i - x_i)$, where the scaling factor p is given as an input. The STL provides a function that divides a double into its integer and decimal part. The decimal part of a negative double, will also be negative, which is why we might have to translate the outcome z_i by $+1$, since

we want values in $[0, 1)$. Now we can create a Bernoulli distribution that returns `true` with probability $1 - z_i$ and `false` otherwise. If the outcome is `true`, we return $y_i = x_i + pz_i \in \mathbb{Z}$ and $y_i = x_i + p(z_i - 1) \in \mathbb{Z}$ otherwise. Since all considered values here are doubles, we might face some minor computation inaccuracies. We round these errors off, in order to be able to safely convert these doubles into integers. Finally, we can create the output element, where the coordinate vector might be taken modulo q, if the representative c is actually in R_q or $R_q^\vee$.

Sampling in Several Ideals

For our applications we basically need to sample elements in K in two different ways. In both cases we want to sample some discrete Gaussians, but we can do this by sampling directly from the discrete Gaussian distribution or by discretizing some continuous Gaussians. First, we describe the discretized variant, where we sample continuous Gaussians over $K_\mathbb{R}$ and discretize them to a certain coset of some ideal $c+\mathcal{I}$. In our application we will actually regard a scaled version, i.e., if ψ is a continuous Gaussian distribution over $K_\mathbb{R}$, then we discretize samples from $p \cdot \psi$ for some scaling factor p. Moreover, we do only need to discretize to $R^\vee$, which is quite convenient, since we saw in Section 2.3.5 how we can efficiently sample elements from ψ using the decoding basis.

`sampleGaussiansInK_RR`. The efficient method of sampling continuous Gaussians over $K_\mathbb{R}$ in the decoding basis is realized in the function `sampleGaussiansInK_RR`, which computes a vector of doubles representing the coordinates of the sampled element in the decoding basis. The function expects four inputs, the double vector for the output, the field K and the mean and standard deviation for the Gaussian distribution. Note that, although we actually sample from an n-dimensional Gaussian distribution, it is sufficient to expect only a single standard deviation, since we use spherical Gaussian distributions whose standard deviations are equal for each dimension. Let s be the given standard deviation. According to Proposition 2.3.10 we can now sample a vector of doubles, where each entry is taken from the Gaussian distribution with standard deviation $s' = s \cdot \sqrt{m/\mathrm{rad}(m)}$. For these Gaussians we use the normal distribution with a default random engine from the standard library. Next, we multiply this vector with $D = C^* B'$ using the function provided by the field class, turning the single samples into the desired coordinates.

`sampleDiscretizedGaussian`. The procedure `sampleDiscretizedGaussian` samples an element from $\lfloor p \cdot \psi \rceil_{c+pR^\vee}$, where the mean and standard deviation of ψ, the coset representative c and the scaling factor p are given as input parameters. Further, the function expects a ring element x to store the sampled element. Using the above described `sampleGaussiansInK_RR` function, we can sample a coordinate vector from ψ and scale it by p, if $p > 1$. Then we discretize the element represented by these coordinates to $c + pR^\vee$ using the `discretize` function. At all times we expect that the used basis is the decoding basis, so in particular c needs to be represented in the decoding basis in order to guarantee a correct working of the function.

Next, we describe the function that samples discrete Gaussians directly from the discrete Gaussian distribution $D_{s,R}$ over R.

`sampleDiscreteGaussiansInR`. Concerning samples from the discrete Gaussian distribution over R, we saw in Section 2.2.5 how we can use the Gram-Schmidt orthogonalization of the powerful basis to our advantage. Lemma 2.2.17 restated Theorem 4.1 of [GPV08], which says that there is an efficient algorithm producing samples from the discrete Gaussian distribution $D_{s,\Lambda+c}$ for some lattice $\Lambda = \mathcal{L}(B)$ for a given basis B, a coset representative c and a standard deviation s. For our purposes, the lattice Λ is given by R and the powerful basis. We will only sample elements from R, so c is zero. The standard deviation s has to fulfill the constraint $s \geq \max_j \|\tilde{b}_j\| \cdot \omega(\sqrt{\log n})$, where $\tilde{b}_j$ are the vectors of the Gram-Schmidt orthogonalization $\widetilde{B}$. A usual choice for $\omega(\sqrt{\log n})$ would be $\log(n)$ and the maximal length of the powerful basis is $\sqrt{n}$, so the algorithm works for example for $s = \sqrt{n} \cdot \log(n)$. In Remark 2.2.19 we already described the algorithm from [GPV08]. There, we used some unspecified subroutine for the sampling of discrete Gaussians over $\mathbb{Z}$. Gentry, Peikert and Vaikuntanathan developed an algorithm for this purpose in [GPV08]. We implemented this algorithm in `sampleDiscreteGaussiansInZZ`, which takes a standard deviation s, a center c and some function value t. The purpose of t is explained later. The algorithm uses rejection samling and works as follows. First, we choose an integer $x \in \mathbb{Z} \cap [c - s \cdot t, c + s \cdot t]$ uniformly at random and then return it with probability $\rho_s(x - c) = \exp(-\pi \|x - c\|^2 / s^2)$ or repeat otherwise. The value t makes sure, that the sample x is actually a relevant sample and not some integer far away from the center, whose returning probability $\rho_s(x-c)$ would nearly be zero. While Gentry, Peikert and Vaikuntanathan suggest to use $t = \log(n) \geq \omega(\sqrt{\log(n)})$, we set the default value of t to 3, since $99,73\%$ of the data of the Gaussian distribution are within 3 times the standard deviation. In our implementation we use the uniform and Bernoulli distribution from the STL. For further analysis and a correctness proof we refer to [GPV08].

Now we can implement the algorithm from Remark 2.2.19. To do so, we have to loop from $n-1$ to 0 and first compute the values $c'_i := \left\langle c_i, \tilde{b}_i \right\rangle / \left\langle \tilde{b}_i, \tilde{b}_i \right\rangle$ and $s_i := s / \|\tilde{b}_i\| > 0$, where s and $c_{n-1} = c$ are given as input parameters. Recall from Section 2.2.5 that c'_i can be computed as the inner product of c_i with the i-th row of the matrix U from the Gram-Schmidt orthogonalization of $\mathrm{CRT}_m = QDU$ and the value $\|\tilde{b}_i\|$ equals the i-th diagonal entry of D. Further, in Section 4.5.2 we implemented the application of U such that U^T is applied instead. Thus we can access the i-th row of U via the i-th column of U^T, i.e., applying U^T to the i-th unit vector. We use the same method to get the i-th column of D and then extract $\|\tilde{b}_i\|$ as the i-th entry of this column. For the computation of the inner product we use the `inner_product` algorithm from the STL. Finally, we can update c_i using a sample z_i from the `sampleDiscreteGaussiansInZZ` routine. After the loop we can equip the output element x with the desired coordinates c_0 with respect to the powerful basis $\vec{p}$.

`sampleUniformlyCoordVec`. Finally, the function `sampleUniformlyCoordVec` is defined for convenience. It simply samples a new coordinate vector for the given element x, whose entries are uniformly distributed over $\mathbb{Z} \cap [lb, ub]$, where the integral bounds lb and ub are given as input parameters. The implementation is straight forward and uses again tools from the STL to produce randomness.

Index

A

AbstractVectorTransformation, 80
add, 105
applyComplexTransformation, 98
applyIntTransformation, 98
applyKroneckerDecomposition, 72
applyRealTransformation, 98
applySingleKroneckerDecomposedMatrix, 72

C

canonical embedding, 17
changeBasisTo, 105
Chinese remainder basis
    of R_q, 34
    of $R_q^\vee$, 46
Chinese remainder transform
    over $\mathbb{C}$, 36
    over $\mathbb{Z}_q^*$, 35
computeRootOfUnity, 79
computeUniqueRepresentative, 106
cyclotomic number field, 14

D

decode, 106
decoding basis, 47
decoding function, 55
discrete Fourier transform
    over $\mathbb{C}$, 36
    over $\mathbb{Z}_q^*$, 34
discretization
    in lattices, 13
    valid, 13
discretize, 106
discriminant, 20

E

eulerTotient, 78

F

fastModPow, 78
findElementOfOrder, 79
findGeneratorOfZZpUnits, 77

G

getNextPowerOfTwo, 77
getPrimeMod1, 78

I

ideal, 19
    coprime, 22
    dual, 22
    fractional, 20
    generated, 19
    integral, 20
    maximal, 22
    prime, 22
    principle, 19
isPowerOfTwo, 77

K

Kronecker product, 10

L

lattice, 11
    ideal lattice, 20

M

MatrixCompFFT, 82
MatrixCompMult, 82
MatrixRealGS, 86
MatrixRealSampleGauss, 87
MatrixZZ, 89
MatrixZZq, 93
minimum distance, 12

mult, 105
multWithScalar, 105
multWithT, 105
multWithT_Inverse, 106

N

neg, 105
norm, 18

P

powerful basis
    of R, 33
    of $R^\vee$, 46
primeFactorization, 78

R

radical, 52
ring of integers, 19
RingLweCryptographyElement, 101
RingLweCryptographyField, 95

S

sampleDiscreteGaussiansInR, 108
sampleDiscreteGaussiansInZZ, 108
sampleDiscretizedGaussian, 107
sampleGaussiansInK_RR, 107
sampleUniformlyCoordVec, 108
sub, 105

T

tensor product of the fields, 15
trace, 18
TransformationCompCRT, 85
TransformationCompDft, 84
TransformationRealPrimePower, 88
TransformationZZ_PrimePower, 91
TransformationZZqCrt, 94

Bibliography

[Bab86] L. Babai. On Lovász lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986. Preliminary version in STAC 1985.

[BBD09] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. *Post-Quantum Cryptography*. Springer-Verlag Berlin Heidelberg, 2009.

[Buc08] Johannes Buchmann. *Introduction to Cryptography*. Springer, 2008.

[Con09] Keith Conrad. The different ideal, 2009. Available at `http://www.math.uconn.edu/~kconrad/blurbs/`.

[CP15] Eric Crockett and Chris Peikert. $\Lambda \circ \lambda$: A functional library for lattice cryptography. Cryptology ePrint Archive, Report 2015/1134, 2015. `http://eprint.iacr.org/`.

[DA15] Beman Dawes and David Abrahams. Boost C++ Libraries, 2015. The C++ Library and its documentation are available at `http://www.boost.org/`.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, pages 197–206, New York, NY, USA, 2008. ACM.

[HJ91] Roger A. Horn and Charles R. Johnsen. *Topics in Matrix Analysis*. Cambridge University Press, 1991.

[Lan94] Serge Lang. *Algebraic Number Theory*. Springer-Verlag New York, 1994.

[Lar12] Christian Stigen Larsen. An algorithm for euler's totient function, 2012. Available at `https://gist.github.com/cslarsen/1635288`.

[LPR13a] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *J. ACM*, 60(6):43:1–43:35, November 2013.

[LPR13b] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A toolkit for ring-LWE cryptography. In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, pages 35–54, 2013.

[MG02] Daniele Micciancio and Shafi Goldwasser. *Complexity of Lattice Problems*. Kluwer Academic Publishers, 2002.

[Mic14] Daniele Micciancio. Lecture in lattice algorithms and applications: Minkowski's theorem, 2014. Available at `https://cseweb.ucsd.edu/classes/sp14/cse206A-a/lec2.pdf`.

[Mil14] James S. Milne. Algebraic number theory (v3.06), 2014. Available at `www.jmilne.org/math/`.

[MvdGvdG15] Margaret E. Myers, Pierce M. van de Geijn, and Robert A. van de Geijn. *Linear Algebra: Foundations to Frontiers*. Self published, 2015. Book get updated regularly. The latest version is available at `www.ulaff.net`.

[Nus82] Henri J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms*. Springer, 1982.

[PM08] M. Püschel and J. M. F. Moura. Algebraic signal processing theory: Cooley-Tukey type algorithms for DCTs and DSTs. *IEEE Transactions on Signal Processing*, 56(4):1502–1521, 2008. Preprint available at `http://users.ece.cmu.edu/~moura/papers/t-sp-apr08--pueschelmoura-ieeexplore.pdf`.

[Reg04] Oded Regev. Lecture : Dual lattice, Fall 2004. Available at `www.cims.nyu.edu/~regev/teaching/lattices_fall_2004/ln/DualLattice.pdf`.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6):34:1–34:40, September 2009. Preliminary version in STOC 2005.

[Ros13] Ian Ross. Prime-length FFT with Rader's algorithm, 2013. Available at `http://www.skybluetrades.net/blog/posts/2013/12/31/data-analysis-fft-9.html`.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, October 1997.

[Sho15] Victor Shoup. NTL: A library for doing number theory, 2015. The C++ Library and its documentation are available at `http://www.shoup.net/ntl/`.

[Was82] Laurence C. Washington. *Introduction to Cyclotomic Fields*. Springer-Verlag New York Heidelberg Berlin, 1982.