Computing the Hermite Normal Form with Applications in Post-Quantum Cryptography

Ruhr-University Bochum

Faculty of Computer Science

Chair for Cryptology & IT-Security

Bachelor Thesis

by

Leon Damer

Abstract

The Hermite Normal Form (HNF) of a matrix is an analogue of the echolon form over the integers. A matrix in HNF is lower triangular, the diagonal entries are strictly greater than zero and any other entry below the diagonal is non-negative and strictly less than the diagonal element of the same row. Any integer matrix can be transformed into its unique HNF.
A common obstacle in computing the HNF is the extensive blow up of intermediate values. As first approach to this problem, we discuss the Modulo Determinant Algorithm from [DKT87]. It keeps the entries bounded by d, the determinant of the lattice, and has a time complexity of $\mathcal{O}(n^3 \log^2 d)$, where n is the dimension of the matrix. Although this algorithm is very useful if the determinant is small, in the general case, the entries still become extremely large.
Secondly, we study the Linear Space Algorithm, taken from [MW01]. It has a time complexity of $\mathcal{O}(n^5 \text{polylog}(M, n))$, where M denotes the largest absolute value of the input matrix. This is as fast as the best previously known algorithms, but in contrast, it assures space complexity linear in the input size, i.e. $\mathcal{O}(n^2 \log M)$.
As last algorithm to compute the HNF we analyze the Heuristic Algorithm, which is based on the first two algorithms. It achieves a much faster runtime in practice, yielding a heuristic runtime of $\mathcal{O}(n^4 \text{polylog}(M, n))$, while keeping the linear space complexity. Python and C++ implementations for all of the above algorithms are provided.
To conclude with an application, we discuss an attack on the LWE problem, that includes perfect hints. A perfect hint is a tuple $(\mathbf{v}, \ell)$ such that $\langle \mathbf{v}, \mathbf{s} \rangle = \ell$, where $\mathbf{s}$ is the secret of the LWE-instance. Such hints would usually originate from side-channel analysis. The attack is based on the primal lattice reduction attack and further uses the HNF to integrate those hints into the lattice.

Contents

Nomenclature

Acronyms

CRT	Chinese Remainder Theorem
CVP	Closest Vector Problem
EA	Euclidean Algorithm
EEA	Extended Euclidean Algorithm
HNF	Hermite Normal Form
LWE	Learning With Errors
SIVP	Shortest Independent Vectors Problem
SVP	Shortest Vector Problem

List of Symbols

$\mathbb{N}$	the set of natural numbers {1, 2, 3,...}
$\mathbb{Z}$	the set of integers
$\mathbb{Q}$	the set of rational numbers
$\mathbb{R}$	the set of real numbers
$[n]$	the set $\{1, \ldots, n\}$ for $n \in \mathbb{N}$
$R^{m \times n}$	the set of all matrices with m rows, n columns and entries from a Ring R
$a \mid b$	a divides b

1 Introduction

Originating from the broad fields of number theory and algebra, the HNF of a matrix is a tool to solve problems across multiple different domains, reaching far beyond purely mathematical problems.

In the mid-19th century, Charles Hermite emerged as a prominent figure in the fields of number theory and algebra, dedicating his efforts to the exploration of concepts such as quadratic forms and invariant theory. Among his noteworthy contributions, in 1851 Hermite proved the existence and uniqueness of the HNF, which was later named in his honor. Subsequently, researchers from different disciplines found diverse applications for the HNF. In the realm of mathematics and computer science for example, the HNF has been generalized to principal ideal domains and is used to solve systems of diophantine equations. Besides its roots in pure mathematics, it finds further applications in integer programming, loop optimization and, most notably for this thesis, cryptography.

Although it is not quite difficult to find a polynomial time algorithm to compute the HNF, it seems to be much harder to discover an algorithm that is polynomial in both, time and space, which additionally runs fast in actual implementations. In naive algorithms, even for very small starting values, the intermediate values tend to become extremely large during the computation. Hence, demanding polynomial space complexity for an algorithm to compute the HNF is of key importance to make it feasible on computers.
Therefore, the first cornerstone of this thesis is to study three different algorithms, building on top of each other, to compute the HNF. This results in a fast algorithm, guaranteeing linear space complexity and time complexity of $\widetilde{\mathcal{O}}(n^4)$.

Quantum computers gained increasing computing power over the last decades. The question of when, or even if, we will witness quantum supremacy to solve authentic and genuine problems in modern computer science remains open. Nevertheless, the rise of quantum computers prompted the National Institute of Standards and Technology (NIST) to launch a competition for standardizing cryptographic schemes that withstand quantum attacks. This is due to the fact that all asymmetric schemes used in practice get broken by a sufficiently large quantum computer. Although it is not clear if such quantum computers will become reality, it is certain that if they do, they will able to decrypt the entire internet traffic. Due to this severe risk, the demand for post-quantum cryptography is substantial.

One of the most promising areas of post-quantum cryptography is based on lattices. There are many different computational problems, on which lattice based-cryptography builds on. One of them is the Learning with Errors (LWE) problem. The second cornerstone of this thesis is to discuss an attack on this LWE problem that requires a fast algorithm to compute the HNF.

1.1 Chapter Outline

Chapter 2. We provide a brief introduction to the mathematical background required for this thesis. We begin by stating fundamental properties and definitions of matrices. This is followed by important theorems regarding the theory of lattices as well as computational problems for lattice-based cryptography. We conclude this chapter with elementary number theory.

Chapter 3. This Chapter builds the core of this thesis by discussing different algorithms to compute the HNF.
Firstly, we study the Modulo Determinant Algorithm, which keeps the intermediate values bounded by the determinant of the lattice.
Secondly, we study the Linear Space Algorithm, decreasing the bound on the entries to be linear in the input size while maintaining an asymptotically fast runtime.
Thirdly, we discuss the Heuristic Algorithm, which is a combination of the two algorithms above. It still has linear space complexity but heuristically reduces the time complexity by a factor of n. Moreover, this algorithm runs much faster in actual implementations. All of the above algorithms are implemented in Python and C++. The the source code is available at

https://github.com/LDamer/HNF/.

Chapter 4. We conclude with an application of the previously discussed algorithms by studying an attack on the LWE problem that utilizes side-channel information. After digesting these information, the intersection of two lattices needs to computed. To accomplish this we make use of the HNF.

2 Mathematical Background

In this chapter we establish the basic mathematical definitions and theorems used throughout this thesis.

2.1 Matrices

Definition 2.1. *For $1 \leq i \leq n$, the **i-th principal submatrix** of a square matrix $\mathbf{A} = (a_{i,j})_{i,j\in[n]}$ is the i-dimensional square matrix obtained by removing the last $n - i$ rows and columns of $\mathbf{A}$. We denote the i-th principal submatrix of $\mathbf{A}$ by $\mathbf{A}(i)$.*

Example 2.2. *For an exemplary 3×3 matrix $\mathbf{A}$, the principal submatrices are*

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, \qquad \mathbf{A}(1) = \begin{pmatrix} 1 \end{pmatrix}, \qquad \mathbf{A}(2) = \begin{pmatrix} 1 & 2 \\ 4 & 5 \end{pmatrix}, \qquad \mathbf{A}(3) = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}. \tag{2.3}$$

Next, its worth to revoke what the determinant of a matrix is and how to calculate it.

Definition 2.4. ***(Laplace expansion)*** *The **determinant** of a square matrix $\mathbf{A} = (a_{i,j})_{i,j\in[n]}$ is computed as*

$$\det(\mathbf{A}) = \sum_{j=1}^{n} (-1)^{i+j} a_{ij} \det(\mathbf{A}^{i,j}) \tag{2.5}$$

where $\mathbf{A}^{i,j}$ is the submatrix of $\mathbf{A}$, resulting from removing row i and column j.

Remark 2.6. *Equivalently, the definition is given by the formula*

$$\det(\mathbf{A}) = \sum_{j=1}^{n} (-1)^{i+j} a_{ji} \det(\mathbf{A}^{j,i}), \tag{2.7}$$

where the roles of rows and columns are swapped. This implies $\det(\mathbf{A}) = \det(\mathbf{A}^T)$.

Remark 2.8. *Let $\mathbf{A} = (a_{ij})_{i\in[m],j\in[n]}$ be a matrix with column vectors $\mathbf{c}_j = (a_{ij})_{i\in[m]}$. To describe $\mathbf{A}$ in terms of its columns we write $\mathbf{A} = [\mathbf{c}_1, ..., \mathbf{c}_n]$.*

The determinant is a multilinear map, i.e. it is linear in each component. For example in the first component this means $\det([\lambda\mathbf{c}_1, \ldots, \mathbf{c}_n]) = \lambda \det([\mathbf{c}_1, \ldots, \mathbf{c}_n])$ and $\det([\mathbf{v} + \mathbf{w}, \ldots, \mathbf{c}_n]) = \det([\mathbf{v}, \ldots, \mathbf{c}_n]) + \det([\mathbf{w}, \ldots, \mathbf{c}_n])$.

Now that we have defined principal submatrices and the determinant, we continue with the next fundamental definition, which finds direct application in computing the HNF.

Definition 2.9. *The **i-th principal minor** of a matrix is the determinant of the i-th principal submatrix.*

In order to study lattices and corresponding algorithms, we introduce a special kind of matrix.

Definition 2.10. *A square integer matrix $\mathbf{A} \in \mathbb{Z}^{n\times n}$ is called* **unimodular** *if*

$$|\det(\mathbf{A})| = 1. \tag{2.11}$$

We describe the set of all unimodular matrices of dimension n with $\mathrm{GL}(n, \mathbb{Z})$

The set of unimodular matrices is exactly the set of invertible matrices over $\mathbb{Z}$. Therefore, a system of linear equations of the form $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is unimodular and $\mathbf{b}$ has integer entries, always has the integer solution $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$.

Lastly for this section, we introduce the Euclidean norm and the maximum norm for vectors and matrices, also known as ℓ_2- and ℓ_∞-norm respectively.

Definition 2.12. *For a vector $\mathbf{v} = (v_i)_{i\in[n]}$ and a matrix $\mathbf{A} = (a_{ij})_{i\in[m],j\in[n]}$, we define the **Euclidean norm** (ℓ_2-norm) as*

$$\|\mathbf{v}\|_2 = \sqrt{v_1^2 + \ldots + v_n^2} \tag{2.13}$$

*and the **maximum norm** (ℓ_∞-norm) as*

$$\|\mathbf{A}\|_\infty = \max_{i\in[m],j\in[n]} |a_{ij}|. \tag{2.14}$$

2.2 Lattices

In this section we introduce lattices. As the name suggests, they are the core building block for lattice-based cryptography, which is one of the most promising areas in post-quantum cryptography. Contents of this section are partly based on [LNP22; MR09]. To start, let us define what a lattice is.

Definition 2.15. *A **lattice** is a discrete additive subgroup of $\mathbb{R}^m$. Every lattice is spanned by linearly independent vectors $\mathbf{b}_1, \ldots, \mathbf{b}_n \in \mathbb{R}^m$ such that*

$$\mathcal{L}(\mathbf{B}) = \mathbf{B}\mathbb{Z}^n = \left\{ \sum_{i=1}^{n} x_i \mathbf{b}_i \;\middle|\; x_i \in \mathbb{Z} \right\} \tag{2.16}$$

where $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{R}^{m\times n}$. We call $\mathcal{L}(\mathbf{B})$ the lattice, generated by $\mathbf{B}$.

A lattice is just an additive group, but not a vectorspace, because it is not closed under multiplication with elements in $\mathbb{R}$. Algebraically, a lattice is a module over the integers. Similar to vector spaces, we can define a basis for a lattice.

Definition 2.17. *Let Λ be a lattice. A set of vectors $\mathbf{b}_1, \ldots, \mathbf{b}_n \in \Lambda$ is called a **basis** of Λ if they generate the whole lattice and are linearly independent. Formally, this means if*

$$\mathcal{L}(\mathbf{B}) = \left\{ \sum_{i=1}^{n} x_i \mathbf{b}_i \mid x_i \in \mathbb{Z} \right\} = \Lambda \tag{2.18}$$

and

$$\sum_{i=1}^{n} x_i \mathbf{b}_i = 0 \implies x_i = 0 \quad \forall i \in [n] \tag{2.19}$$

*where $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{R}^{m\times n}$. The amount of of vectors, n, is called the **dimension** of Λ.*

Integer lattices, meaning the basis vectors are from $\mathbb{Z}^m$, are very practical for real world implementations for several reasons. Firstly, most of the real numbers have infinitely many decimal places. For that reason, they can neither be stored on a real computer nor be implemented

precisely. Secondly, the lattices generated by rational matrices are just a scaled version of the ones generated by integers matrices. This is because if a rational matrix is given, one can simply multiply it with the smallest common multiple of all its denominators and get a corresponding matrix with only integer entries. Due to this scaling, lattice problems as described in Section 2.2.1, are just as hard on the rationals as on the integers.
And since the integer arithmetic is the most simple to implement, the lattices used in cryptography are mostly integer lattices. An example of an integer lattice is depicted in Figure 1.

It is important to point out, just as with vectorspaces, lattices do not have a unique basis. This means, there exist different matrices generating the same lattice. The following proposition describes what matrices generate the same lattices.

Proposition 2.20. *Let $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$ be two integer matrices. They generate the same lattice if and only if there exists a unimodular matrix $\mathbf{U} \in \mathrm{GL}(n, \mathbb{Z})$ such that $\mathbf{A} = \mathbf{BU}$.*

Another practical tool for comparing lattices is the fundamental parallelpiped as defined below.

Definition 2.21. *The **fundamental parallelpiped** $P(\mathbf{B})$ of a lattice $\mathcal{L}(\mathbf{B})$ is the set*

$$P(\mathbf{B}) = \{Bx \mid x \in [0, 1)\} \tag{2.22}$$

The next definition builds right on top, as we will see afterwards.

Definition 2.23. *The **determinant** of a lattice $\Lambda = \mathcal{L}(\mathbf{B})$ is defined as*

$$\det(\Lambda) = \sqrt{\det(\mathbf{B}^T\mathbf{B})} \tag{2.24}$$

If $\mathbf{B}$ is a square matrix, it holds that $\det(\mathbf{B}) = \det(\mathbf{B}^T)$, which immediately implies $\det(\mathcal{L}(\mathbf{B})) = |\det(\mathbf{B})|$. The determinant of a lattice is simply the volume of the fundamental parallelpiped.

Proposition 2.25. *The determinant of a lattice does not depend on the basis and is an invariant of the lattice.*

Proof. Let $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$ generate the same lattice. Due to Proposition 2.20, we know $\mathbf{A} = \mathbf{BU}$ for $\mathbf{U} \in \mathrm{GL}(n, \mathbb{Z})$. Hence,

$$\sqrt{\det(\mathbf{A}^T\mathbf{A})} = \sqrt{\det(\mathbf{U}^T\mathbf{B}^T\mathbf{BU})} = \sqrt{\det(\mathbf{U}^T)\det(\mathbf{B}^T\mathbf{B})\det(\mathbf{U})} = \sqrt{\det(\mathbf{B}^T\mathbf{B})}, \tag{2.26}$$

which implies $\det(\mathcal{L}(\mathbf{A})) = \det(\mathcal{L}(\mathbf{B}))$. □

The opposite direction does not always hold. Two matrices with equal determinant do not generally span the same lattice. Consider the Matrix $\mathbf{A} = \begin{bmatrix} \sqrt{2} & 1 \\ 1 & \sqrt{2} \end{bmatrix}$, which has determinant one, but clearly does not generate the lattice $\mathbb{Z}^2$, as the identity matrix with determinant one would do.

Remark 2.27. *This additionally shows, Definition 2.23 is indeed well defined.*

A very useful tool to bound the determinant of a lattice is given by the Hadamard bound.

Proposition 2.28. ***(Hadamard bound)*** *For* $\mathbf{A} = [\mathbf{c}_1, \ldots, \mathbf{c}_n] \in \mathbb{R}^{m \times n}$*, the determinant is bound by*

$$\det(\mathcal{L}(\mathbf{A})) \leq \prod_{i=1}^{n} \|\mathbf{c}_i\|_2. \tag{2.29}$$

Proposition 2.30. *Let* $\mathbf{A} \in \mathbb{R}^{n \times n}$ *be a square matrix with* $d = |\det(\mathbf{A})|$*. Then* $d\mathbb{Z}^n \subseteq \mathcal{L}(\mathbf{A})$*.*

Proof. We need to show the vector $\mathbf{v}_j = (0, \ldots, 0, d, 0, \ldots, 0)^T$, where d is in row j, is an integer linear combination of columns of $\mathbf{A}$ for all $j \in [n]$. Therefore, let $\mathbf{a}_i$ be the i-th column of $\mathbf{A}$. By applying Cramer's rule we get

$$\begin{aligned} x_i &= \frac{\det(\mathbf{a}_1, \ldots, \mathbf{a}_{i-1}, \mathbf{v}_j, \mathbf{a}_{i+1}, \ldots, \mathbf{a}_n)}{\det(\mathbf{A})} \\ &= \frac{d \cdot \det(\mathbf{a}_1, \ldots, \mathbf{a}_{i-1}, \mathbf{e}_j, \mathbf{a}_{i+1}, \ldots, \mathbf{a}_n)}{\det(\mathbf{A})} \\ &= \pm\det(\mathbf{a}_1, \ldots, \mathbf{a}_{i-1}, \mathbf{e}_j, \mathbf{a}_{i+1}, \ldots, \mathbf{a}_n) \in \mathbb{Z}, \end{aligned} \tag{2.31}$$

which shows all coordinates x_i are integers. □

This is the reason why adding multiples of d to any coordinate never leaves the lattice. Therefore, it is always possible to reduce an entry of a given vector modulo the determinant, which leads to the equivalence relation $\mathbf{v} \equiv \mathbf{w} \Leftrightarrow \mathbf{v} - \mathbf{w} \in d\mathbb{Z}^n$. In that sense $\Lambda / d\mathbb{Z}^n$ defines a quotient module of the lattice Λ with the quotient map $f(\mathbf{v}) = \mathbf{v} \mod d$.

The following definition is particularly relevant for lattice-based cryptography, because it is used to define many difficult problems, as we will see in Section 2.2.1.

Definition 2.32. *For* $i \in [n]$*, the* ***i-th successive minimum*** $\lambda_i(\Lambda)$ *of an* n*-dimensional lattice* Λ *is the minimal radius* r*, such that* i *linearly independent vectors of norm at most* r *are contained in the lattice. Formally, this can be defined as*

$$\lambda_i(\Lambda) = \min\{r \mid \dim(\operatorname{span}(\Lambda \cap B(\mathbf{0}, r)) \geq i)\}, \tag{2.33}$$

where $B(\mathbf{0}, r) = \{\mathbf{x} \in \mathbb{R}^n \mid \|\mathbf{x}\|_2 \leq r\}$ *is the closed ball of vectors around* $\mathbf{0}$ *with radius at most* r*.*

We continue by demonstrating some of the above definitions and propositions with a straightforward example of the most basic integer lattice. The two matrices

$$\mathbf{A} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \text{and} \quad \mathbf{B} = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix} \tag{2.34}$$

both generate the lattice $\mathbb{Z}^2$ as depicted in Figure 1. The matrices $\mathbf{A}$ and $\mathbf{B}$ are visualized by the blue and red basis vectors respectively, while the fundamental parallelpiped of $\mathcal{L}(\mathbf{B})$ is shown in red too.

According to Proposition 2.20, there exists a unimodular matrix $\mathbf{U} \in \mathrm{GL}(2, \mathbb{Z})$ such that $\mathbf{A} = \mathbf{B}\mathbf{U}$. This matrix is given by

$$\mathbf{U} = \begin{pmatrix} 1 & -1 \\ -1 & 2 \end{pmatrix} \tag{2.35}$$

Furthermore for this example, we can verify the determinant invariant, proven in Proposition 2.25, since $|\det(\mathbf{A})| = 1$ and also $|\det(\mathbf{B})| = 2 \cdot 1 - 1 \cdot 1 = 1$.

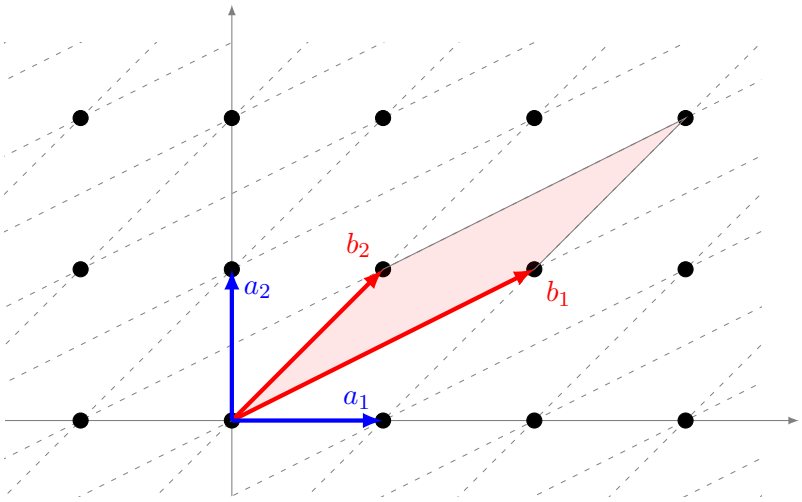


Figure 1: The integer lattice $\mathbb{Z}^2$, generated by two different bases

For this two-dimensional lattice, the successive minima are $\lambda_1(\mathbb{Z}^2) = \lambda_2(\mathbb{Z}^2) = 1$. This is because for example a_1 and a_2 are linearly independent and those are already as short as possible.

2.2.1 Difficult Problems for Cryptography

We now state some of the most relevant computational problems lattice-based cryptography is based on. There exist many worst-case and average-case reductions between the computational problems in the realm of lattices, while some of them are proven to be NP-hard. Since NP-hard problems are believed to be very difficult even for quantum computers, this is one reason why lattice-based cryptography belongs to the most promising candidates for post-quantum cryptography.

Definition 2.36. ***Shortest Vector Problem (SVP).*** *Given a lattice $\mathcal{L}(\mathbf{B})$, find a shortest non-zero vector $\mathbf{v} \in \mathcal{L}(\mathbf{B})$, i.e. such that $\|v\|_2 = \lambda_1(\mathcal{L}(\mathbf{B}))$.*

In [Ajt98], Miklós Ajtai proved the NP-hardness of SVP.

Definition 2.37. ***Shortest Independent Vectors Problem (SIVP).*** *Given a lattice $\mathcal{L}(\mathbf{B})$ of dimension n. Find n linearly independent vectors $\mathbf{v}_1, \ldots, \mathbf{v}_n \in \mathcal{L}(\mathbf{B})$ such that $\max_{i \in [n]} \|\mathbf{v}_i\|_2 = \lambda_n(\mathcal{L}(\mathbf{B}))$.*

Definition 2.38. ***Closest Vector Problem (CVP).*** *Given a lattice $\mathcal{L}(\mathbf{B})$ and a target vector $\mathbf{t} \in \mathbb{R}^m$ that is not contained in the lattice. Find a closest vector $\mathbf{v} \in \mathcal{L}(\mathbf{B})$ to $\mathbf{t}$, i.e. such that for all $w \in \mathcal{L}(\mathbf{B})$ it holds $\|\mathbf{v} - \mathbf{t}\|_2 \leq \|\mathbf{w} - \mathbf{t}\|_2$.*

CVP was proven to be NP-hard, even before SVP. More details are provided in [MG02]. For the above problems, an approximation variant can be formulated. For SVP it is stated below.

Definition 2.39. ***Shortest Vector Approximation Problem ($\mathbf{SVP}_\gamma$).*** *Given a lattice $\mathcal{L}(\mathbf{B})$, find a γ-approximation vector $\mathbf{v} \in \mathcal{L}(\mathbf{B})$, i.e. such that $\lambda_1(\mathcal{L}(\mathbf{B})) \leq \|v\|_2 \leq \gamma\lambda_1(\mathcal{L}(\mathbf{B}))$ for $\gamma \in \mathbb{R}$.*

Last but not least, we state the LWE problem. Although on first sight it does not involve lattices immediately, the LWE problem can be reduced to numerous lattice problems, including the decisional variant of SVP_γ as shown in [Pei09]. It further builds the foundation for most of modern lattice-based cryptographic schemes.

Definition 2.40. ***Learning With Errors (LWE).*** *Given a matrix $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$ and the vector $\mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e} \mod q$, where $\mathbf{s} \in \mathbb{Z}^n$ and $\mathbf{e} \in \mathbb{Z}^m$ are drawn from a known probability distribution χ, find $\mathbf{s}$.*

2.3 Number Theory

We carry on with some important results from elementary number theory.

Theorem 2.41. ***(Prime Number Theorem)*** *Let $\pi(x)$ denote the amount of prime numbers less or equal to x. $\pi(x)$ is bounded by*

$$\pi(x) = \mathcal{O}\left(\frac{x}{\log x}\right). \tag{2.42}$$

If p_n denotes the n-th prime number, it immediately follows that $p_n = \mathcal{O}(n \log n)$, since $\pi(p_n) = n$.

2.3.1 Greatest Common Divisor

An essential notion in number theory is the greatest common divisor, which refers to the largest number that divides two integers. We now provide a formal definition.

Definition 2.43. *The* ***greatest common divisor*** *of two integers $a, b \in \mathbb{Z} \setminus \{0\}$ is the natural number $g \in \mathbb{N}$ that satisfies*

$$g = \gcd(a, b) = \max\left\{d \in \mathbb{N} \ : \ d \mid a \ \wedge \ d \mid b\right\} \tag{2.44}$$

The next lemma shows another way of defining the greatest common divisor. We can write the $\gcd(a, b)$ as an integer linear combination of a and b.

Lemma 2.45. ***(Bezout's Lemma)*** *For $a, b \in \mathbb{Z} \setminus \{0\}$ we can express the $\gcd(a, b) \in \mathbb{N}$ as*

$$\gcd(a, b) = \min\left\{xa + yb \ \mid \ x, y \in \mathbb{Z} \text{ and } ax + by > 0\right\} \tag{2.46}$$

The integers x and y achieving this minimum are called Bezout coefficients.

By using the Euclidean Algorithm (e.g. see [Coh95]) we calculate the greatest common divisor of two integers and by reaching out to the Extended Euclidean Algorithm (EEA) we additionally compute the Bezout coefficients. The EEA is a key component to compute the HNF as explained in Section 3.3.3. Therefore we prove the following proposition in detail, where Algorithm 1 is taken from [Stu21].

Proposition 2.47. *On input $a, b \in \mathbb{N}$ with $a \geq b$, the Extended Euclidean Algorithm as shown in Algorithm 1 returns x, y such that $\gcd(a, b) = ax + by$ in time $\mathcal{O}(\log a)$.*

Algorithm 1: Extended Euclidean Algorithm

Data: $a, b \in \mathbb{N}$ and $a \geq b$
Result: x, y s.t. $ax + by = \gcd(a, b)$
$(r_0, r_1) \leftarrow (a, b)$
$(x_0, x_1) \leftarrow (1, 0)$
$(y_0, y_1) \leftarrow (0, 1)$
$i \leftarrow 1$
while $r_i \neq 0$ **do**
  $q_i \leftarrow r_{i-1} \operatorname{div} r_i$
  $r_{i+1} \leftarrow r_{i-1} - q_i r_i$
  $x_{i+1} \leftarrow x_{i-1} - q_i x_i$
  $y_{i+1} \leftarrow y_{i-1} - q_i y_i$
  $i \leftarrow i + 1$
return (x_{i-1}, y_{i-1})

Proof. Observe that $q_i \in \mathbb{Z}$ and the r_i are strictly decreasing for $i > 1$. This is true, because we can express r_{i-1} as $r_{i-1} = q_i \cdot r_i + c$, where $0 \leq c < r_i$. But then we get $r_{i+1} = r_{i-1} - q_i \cdot r_i = q_i \cdot r_i + c - q_i \cdot r_i = c < r_i$. Therefore, the while-loop is guaranteed to terminate and the algorithm must return.
Now, we first show by induction that the invariant $r_i = ax_i + by_i$ holds. Secondly, we show that $g = ax + by$ is actually the greatest common divisor of a and b, which completes the proof.
Let us prove the invariant. For $i = 0$ we have $ax_0 + by_0 = r_0 \cdot 1 + b \cdot 0 = r_0$ and for $i = 1$ we get $ax_1 + by_1 = a \cdot 0 + r_1 \cdot 1 = r_1$.
Assuming it holds for i and $i - 1$ we get

$$\begin{aligned} ax_{i+1} + by_{i+1} &= a(x_{i-1} - q_i x_i) + b(y_{i-1} - q_i y_i) \\ &= ax_{i-1} + by_{i-1} - q_i(ax_i + by_i) \\ &= r_{i-1} - q_i r_i \\ &= r_{i+1} \end{aligned} \tag{2.48}$$

Next, we show $ax + by \leq \gcd(a, b)$ and vice versa, which proves $\gcd(a, b) = ax + by$.
Let $g = \gcd(a, b)$. By definition, g divides both a and b. Therefore, we can write $a = sg$ and $b = tg$ for some $s, t \in \mathbb{Z}$. Then we have $ax + by = (sg)x + (tg)y = g \cdot (sx + ty)$, which shows $g \mid ax + by$. Because after termination $ax + by = r_{i-1} > 0$, it follows $0 < \gcd(a, b) \leq ax + by$.
For the opposite direction, observe after termination we have $r_i = 0$, but also $r_i = r_{i-2} - q_{i-1} r_{i-1}$. This means $r_{i-2} = q_{i-1} r_{i-1}$ and thus $r_{i-1} \mid r_{i-2}$. Looking at the iteration before, we have $r_{i-1} = r_{i-3} - q_{i-2} r_{i-2}$ leading to $r_{i-3} = r_{i-1} + q_{i-2} r_{i-2}$ and because obviously $r_{i-1} \mid r_{i-1}$ and, as seen before, $r_{i-1} \mid r_{i-2}$, we get $r_{i-1} \mid r_{i-3}$. By induction it follows, $r_{i-1} \mid b$ and subsequently $r_{i-1} \mid a$. Recalling $r_{i-1} = ax_{i-1} + by_{i-1} = ax + by$, it follows that $ax + by$ divides a and b and therefore, by Definition 2.43, we have $ax + by \leq \gcd(a, b)$, but also $\gcd(a, b) \leq ax + by$, proving $\gcd(a, b) = ax + by$.

To prove the runtime, note the while-loop starts with $r_0 = a$ and stops if $r_i = 0$. In each iteration we set r_{i+1} to $r_{i-1} - q_i r_i$. Hence, if q is as small as possible in each iteration, the while-loop takes the longest to finish. Because $r_0 \geq r_1$, the smallest possible is $q = 1$. Substituting this into the formula we get $r_{i+1} = r_{i-1} - r_i$. Since this is exactly the recurrence equation for the Fibonacci numbers, but converging towards zero, we found the worst case input to the EEA.

Let F_n be the n-th Fibonacci number closest to a. F_n is approximated by $F_n \approx \Phi^n$, where Φ denotes the golden ratio. We conclude there are $n = \log_\Phi F_n$ Fibonnaci numbers in between, thus we execute the while-loop $\mathcal{O}(n) = \mathcal{O}(\log a)$ times. □

Remark 2.49. *It is easy to see* $\gcd(a,b) = \gcd(b,a) = \gcd(-b,a)$. *This justifies the restrictions to* $a, b \in \mathbb{N}$ *and* $a \geq b$ *on the input to the EEA. Thus, by using Algorithm 1, the greatest common divisor is efficienlty computable for* $a, b \in \mathbb{Z}\backslash\{0\}$.

Remark 2.50. *Because the* $r_i = ax_i + by_i$ *are strictly decreasing and together with Remark 2.49, this yields a constructive proof for Bezout's Lemma.*

2.3.2 Chinese Remainder Theorem

The Chinese Remainder Theorem (CRT) provides a method to solve simultaneous modular congruences. Due to its importance for the HNF we are going to prove it in detail.

Theorem 2.51. ***(Chinese Remainder Theorem)*** *Let* $m_1, \ldots, m_k \in \mathbb{N}$ *and* $a_1, \ldots, a_k \in \mathbb{Z}$ *be (positive) integers. If* $m_1, \ldots, m_k$ *are coprime, meaning* $\gcd(m_i, m_j) = 1$ *for* $i \neq j$, *then there exist a unique* $0 \leq x < \prod_{i=1}^{k} m_i$ *fulfilling*

$$\begin{aligned} x &\equiv a_1 \mod m_1 \\ x &\equiv a_2 \mod m_2 \\ &\vdots \\ x &\equiv a_k \mod m_k \end{aligned} \tag{2.52}$$

Proof. We will revisit the proof presented in [Stu21]. The proof provides a direct method to construct the solution x.
Let $M = \prod_{i=1}^{k} m_i$. We first want to compute

$$x_i \equiv \begin{cases} 1 \mod m_i \\ 0 \mod m_j \quad (j \neq i) \end{cases} \tag{2.53}$$

for $i \in [k]$. To do so, we define $p_i = \frac{M}{m_i}$. Because the $m_1, \ldots, m_k$ are pairwise coprime we get $\gcd(m_i, p_i) = 1$. Therefore, by using Bezout's Lemma and get $1 = xm_i + yp_i$. Choosing $x_i = yp_i$ implies $x_i \equiv 0 \mod p_i$, but also $x_i \equiv 1 \mod m_i$ and therefore satisfying Equation 2.53.
Now we compute the solution as

$$x = \sum_{i=1}^{k} a_i x_i \tag{2.54}$$

because of the properties of the x_i. To fulfill the requirements of uniqueness and $x < M$, we reduce $x \mod M$ to get the final result. Since reducing modulo M is the same as subtracting multiples of M and because $M = \prod_{i=1}^{k} m_i$, we effectively subtract multiples of $m_1, \ldots, m_k$ respectively. Therefore, this does not alter x modulo m_i for all $i \in [k]$. □

A more abstract way of seeing the CRT, giving an insight of the underlying structure, is shown by the following ring isomorphism.

$$\mathbb{Z}/M\mathbb{Z} \cong (\mathbb{Z}/m_1\mathbb{Z}) \times \ldots \times (\mathbb{Z}/m_k\mathbb{Z}) \tag{2.55}$$

This motivates to perform calculations over the $\mathbb{Z}/m_i\mathbb{Z}$ individually and then restore the result over $\mathbb{Z}/M\mathbb{Z}$ by using the CRT. Although it increases the number of computations, this usually brings a notably speed up, since the involved numbers are much smaller. Because the space complexity decreases, using the CRT is a popular choice to reduce the memory space required by an algorithm [McC77; ST67].

Although the the CRT guarantees a unique solution $0 \leq x < M = \prod_{i=1}^{k} m_i$, it is possible to compute a solution x' centered around 0, i.e. $\lceil -\frac{M}{2} \rceil \leq x' < \lceil \frac{M}{2} \rceil$ as well. For that, simply compute x with the usual CRT and if $x \geq \lceil \frac{M}{2} \rceil$ set $x' = x - M$, otherwise set $x' = x$. Since subtracting multiples of M does not alter the result modulo M, x' is still a valid solution. Furthermore, observe that every integer in the interval $[\lceil \frac{M}{2} \rceil, M)$ has exactly one congruent integer in $[\lceil -\frac{M}{2} \rceil, 0)$, which implies x' is still unique.
This procedure has useful applications, e.g. if we we want to compute a value y with $-b < y < b$ for a known bound b. By using the CRT with a suitable $M > 2b$, we get a unique $y \geq 0$. And by subtracting M if necessary, we get a unique y' in the given bound.

3 Computing the Hermite Normal Form

In this chapter we discuss the main topic of this thesis, the HNF and how to efficiently compute it. We will start with the definition and proof of existence and uniqueness of the HNF. Afterwards, we analyze three different algorithms. A modular algorithm taken from [DKT87], a linear space algorithm and a heuristic algorithm, both taken from [MW01]. The heuristic algorithm builds on top of the first two algorithms.

3.1 Hermite Normal Form

This thesis and the later presented algorithms are based on the following definition of the HNF for arbitrary integer matrices.

Definition 3.56. *A matrix $\mathbf{A} \in \mathbb{Z}^{m \times n}$ is in **Hermite Normal Form** if*

1. *There exists a sequence of integers $1 \leq i_1 < \ldots < i_n$ such that $h_{ij} = 0$ for all $i < i_j$ (strictly decreasing column height)*
2. *$0 \leq h_{i_j,k} < h_{i_j,j}$ for all $0 \leq k < j \leq n$ (the top non-zero element of each column is the greatest element in its row)*

In the above definition, i_j indicates the row of the pivot element of column j. Meaning i_1 indicates the first non-zero entry of column 1, i_2 of column 2 and so on.
From the second point we observe, the linearly independent rows of a matrix in HNF do not contain negative entries and the pivot elements of are even strictly greater than zero.

Example 3.57. *Let us have a look at some matrices in HNF to get familiar with the definition.*

$$\mathbf{H}_1 = \begin{pmatrix} 12 & 0 & 0 \\ 0 & 1 & 0 \\ 15 & 3 & 644 \end{pmatrix}, \quad \mathbf{H}_2 = \begin{pmatrix} 2 & 0 \\ 76 & 89 \\ -5352 & -9 \end{pmatrix}, \quad \mathbf{H}_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 155 & 35654 & 543644 \end{pmatrix} \tag{3.58}$$

Note that $\mathbf{H}_2$ does not have full row rank. Consequently it is in HNF although the last row contains negative values.

Theorem 3.59. *For a matrix $\mathbf{A} \in \mathbb{Z}^{m \times n}$ with full row rank, there exists a unique matrix $\mathbf{H} \in \mathbb{Z}^{m \times n}$ in HNF that satisfies $\mathcal{L}(\mathbf{A}) = \mathcal{L}(\mathbf{H})$. We call $\mathbf{H}$ the Hermite Normal Form of $\mathbf{A}$*

Proof. The existence follows by the algorithm discussed in Section 3.3. To show the uniqueness we revisit the proof from [Sch86].
Suppose for $\mathbf{A} \in \mathbb{Z}^{m \times n}$ with full row rank, the matrices $\mathbf{H} = (h_{ij})$ and $\mathbf{H}' = (h'_{ij})$ are in HNF and they all generate the same lattice Λ. We assume $\mathbf{H} \neq \mathbf{H}'$.
Since removing zero columns in $\mathbf{H}$ and $\mathbf{H}'$ does not alter the lattice, we assume without loss of generality they are both $m \times m$ square nonsingular matrices in HNF.
Because $\mathbf{H} \neq \mathbf{H}'$, there are two elements $h_{ij} \neq h'_{ij}$ where i is as small as possible. Again without loss of generality, assume $h_{ii} \geq h'_{ii}$. Let h_j and h'_j denote the j-th column of $\mathbf{H}$ and $\mathbf{H}'$ respectively. As both matrices generate the same lattice, h_j and h'_j are contained in Λ and therefore $h_j - h'_j \in \Lambda$. This means, we can express $h_j - h'_j$ as integer linear combination of the columns of $\mathbf{H}$. Because we chose i as small as possible, the first $i-1$ elements of $h_j - h'_j$ are all zero. And since $\mathbf{H}$ is lower triangular, $h_j - h'_j$ can be expressed as integer linear combination of only the columns $h_i, \ldots, h_m$ of $\mathbf{H}$. But of these columns, only column h_i has a non-zero entry at row i. This leads to $h_{ij} - h'_{ij} = z h_{ii}$ for some $z \in \mathbb{Z} \backslash \{0\}$.

Due to the properties of the HNF and the assumption $h_{ii} \geq h'_{ii}$, we know $0 \leq h_{ij} < h_{ii}$ and $0 < h'_{ij} < h'_{ii} \leq h_{ii}$ if $j < i$. But then $0 \leq |h_{ij} - h'_{ij}| < h_{ii}$. This implies $z = 0$, which is a contradiction. In the case $i = j$ we have $h_{ij} - h'_{ij} = h_{ii} - h'_{ii}$. Since $h_{ii} \geq h'_{ii} > 0$ we have $h_{ii} > h_{ii} - h'_{ii} = h_{ij} - h'_{ij} > 0$, which also leads to the contradiction $z = 0$. It follows, $\mathbf{H} = \mathbf{H}'$. □

3.2 Modulo Determinant Algorithm

There exist multiple different algorithms to compute the HNF [Fru77; KB79a; Ili88]. The most basic one is a straight forward approach that utilizes just basic column operations. Let $\mathbf{A} = (a_{ij})_{i,j\in[n]}$ be the input matrix, a very basic algorithm consists of the following two parts.

1. Transform $\mathbf{A}$ into a lower triangular matrix (point 1 of Definition 3.56)
2. Perform appropriate column operations such that every off-diagonal element is non-negative and smaller than the diagonal element of its row (point 2 of Definition 3.56)

Regarding the first point, to produce zeros to the right of the diagonal elements, we define a unimodular transformation $\mathbf{U}_{rc} = (u_{ij})_{i,j\in[n]}$ that produces a zero at a_{rc} $(c > r)$ and sets a_{rr} to $\gcd(a_{rr}, a_{rc})$, after applying it from the right to the matrix $\mathbf{A}$.

Example 3.60. *The following example demonstrates how* $\mathbf{U}_{rc}$ *works for* $r = 1$ *and* $c = n$*.*

$$\mathbf{A}\mathbf{U}_{1n} = \begin{pmatrix} a_{11} & \dots & a_{1,n-1} & a_{1n} \\ \vdots & & & \vdots \\ a_{n1} & \dots & \dots & a_{nn} \end{pmatrix} \cdot \mathbf{U}_{1n} = \begin{pmatrix} \gcd(a_{11}, a_{1n}) & \dots & a_{1,n-1} & 0 \\ \vdots & & & \vdots \\ \widetilde{a}_{n1} & \dots & \dots & \widetilde{a}_{nn} \end{pmatrix} \tag{3.61}$$

Note, this matrix multiplication only changes column r and c. $\mathbf{U}_{rc}$ is defined as the identity matrix with the modifications

$$u_{rr} = k, \qquad u_{cr} = l, \qquad u_{rc} = -\frac{a_{rc}}{g}, \qquad u_{cc} = \frac{a_{rr}}{g}, \tag{3.62}$$

where k, l and g are computed with the EEA to fulfill $g = \gcd(a_{rr}, a_{rc}) = ka_{rr} + la_{rc}$. Thus, $\mathbf{U}_{rc}$ looks like this

$$\mathbf{U}_{rc} = \begin{pmatrix} 1 & & & \dots & & & 0 \\ & \ddots & & & & & \\ & & k & \dots & -\frac{a_{rc}}{g} & & \\ & & & 1 & & & \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ & & & & 1 & & \\ & & l & \dots & \frac{a_{rr}}{g} & & \\ & & & & & \ddots & \\ 0 & & & \dots & & & 1 \end{pmatrix} \begin{matrix} \\ \\ \longleftarrow \text{row } r \\ \\ \\ \\ \longleftarrow \text{row } c \\ \\ \\ \end{matrix} . \tag{3.63}$$

$$\begin{matrix} & \uparrow & & \uparrow \\ & \text{column } r & & \text{column } c \end{matrix}$$

$\mathbf{U}_{rc}$ is indeed a unimodular transform, not altering the lattice, as the following Laplace expansion along column r shows.

$$\det(\mathbf{U}_{rc}) = k\det(\mathbf{U}_{rc}^{r,r}) + l\det(\mathbf{U}_{rc}^{r,c}) = k\frac{a_{rr}}{g} + l\frac{a_{rc}}{g} = 1 \tag{3.64}$$

To see that $\det(\mathbf{U}_{rc}^{r,c}) = \frac{a_{rc}}{g}$, note that the diagonal element of $\mathbf{U}_{rc}^{r,c}$ in row r is zero and the only other nonzero element in the row is $-\frac{a_{rc}}{g}$. After changing those two columns, the diagonal element in row r is $-\frac{a_{rc}}{g}$ and the sign of the determinant flips. Lastly, rows $r+1, \dots, c-1$ have

a zero on their diagonal. By adding the neighboring column, the determinant does not change and we get a lower triangular matrix where every diagonal element is one, except in row r it is $-\frac{a_{rc}}{g}$. Hence, because of the previous sign flip, the determinant is $\frac{a_{rc}}{g}$.
After applying $\mathbf{U}_{rc}$ for all elements to the right of the diagonal element in row r, the diagonal element is $a_{rr} = \gcd(a_{rr}, a_{r,r+1}, \ldots, a_{rn})$. By repeating this procedure for all rows, we transform $\mathbf{A}$ into lower triangular form.

For the second part, to reduce the elements to be smaller than the diagonal element, we proceed as follows. To reduce an element a_{ij} $(j < i)$, we subtract a proper multiple of column i from column j such that $0 \leq a_{ij} < a_{ii}$. Since the first i rows are lower triangular, subtracting column i does not alter the entries of the rows $1, \ldots, i-1$. This way, we reduce the lower triangular part of the matrix row by row, starting with the first and ending with the last. This does not alter the lattice, because we only add multiples of the columns.

Unfortunately, the intermediate values during this algorithm become extremely large, which makes it infeasible on big matrices. The space complexity is tremendous and on actual computers huge numbers also imply slow practical runtime. In [FH97], the intermediate values were proven to grow exponentially for a specific procedure based on Gaussian elimination.

The algorithm presented in [DKT87] works on square nonsingular matrices and uses the same two steps as described above. The key enhancement is that it works modulo the determinant. To see why this is a reasonable idea, observe that the HNF $\mathbf{H}$ of a matrix $\mathbf{A}$ is lower triangular and has the same determinant as $\mathbf{A}$ in terms of the absolute value. Thus, the product of diagonal elements of $\mathbf{H}$ equals the determinant. And since $\mathbf{H}$ is in HNF, the diagonal elements are the largest in each row, which implies the determinant of $\mathbf{A}$ is a tight upper bound on the entries of $\mathbf{H}$. This upper bound reaches equality if and only if every diagonal element except one is equal to 1. The left over diagonal element is then equal to the determinant.
Furthermore, assume we know a diagonal element of $\mathbf{H}$, e.g. h_{11}, this implies that all other diagonal elements cannot be larger than $\frac{|\det(\mathbf{A})|}{h_{11}}$, because their product must be equal to the determinant.
In [DKT87], the runtime for this Modulo Determinant Algorithm was proven to be $\mathcal{O}(n^3 \log^2 d)$, where d is the determinant of the input matrix. If M is the largest absolute value of the input matrix, we bound d by $\mathcal{O}(M^n)$, which leads to an overall runtime of $\mathcal{O}(n^3 \log^2 M^n) = \mathcal{O}(n^5 \log^2 M)$.

If we used the basic algorithm and just inserted the modulo reductions in every operation, this would not yield the correct result. To understand how reducing modulo the determinant during the basic algorithm changes the result we have the following proposition.

Proposition 3.65. *Let $\mathbf{H} = (h_{ij})$ be the HNF of $\mathbf{A} \in \mathbb{Z}^{n \times n}$ and $d = \det(\mathbf{A})$. We define $d_1 = d$ and $d_{i+1} = \frac{d_i}{h_{ii}}$ for $1 \leq i < n$. If the matrix $\mathbf{B} = (b_{ij})$ results from the basic algorithm, but reducing every element modulo d during the process, then $h_{ii} = \gcd(d_i, b_{ii})$.*

Proof. A detailed proof is given in [DKT87]. □

Now, let us have a look on how the Modulo Determinant Algorithm works. Given an

input matrix $\mathbf{A}' = (a'_{ij})_{i,j\in[n]}$, we compute the HNF of the modified matrix

$$\mathbf{A} = (a_{ij})_{i\in[n],j\in[2n]} = \begin{pmatrix} a'_{11} & \dots & a'_{1n} & d_1 & \dots & 0 \\ \vdots & & \vdots & \vdots & \ddots & \vdots \\ a'_{n1} & \dots & a'_{nn} & 0 & \dots & d_n \end{pmatrix}, \tag{3.66}$$

where the d_i are defined as in Proposition 3.65. According to Proposition 2.30, $\mathcal{L}(\mathbf{A}) = \mathcal{L}(\mathbf{A}')$. To compute the HNF of $\mathbf{A}$, we use the basic algorithm with two slight modifications.

Firstly, we perform the two steps from the basic algorithm for every row, starting with the first and terminating with the last row. This means, for every row, we first want to produce zeros to the right of the diagonal element and then already reduce the elements to the left of the diagonal element to be smaller than the diagonal elements.

The second difference is, we reduce every element modulo the corresponding determinant, after transforming each row. This means, at the start we reduce everything modulo $d_1 = d$. But for the second row, since we already know h_{11}, we want to reduce modulo the smallest possible value, which is $d_2 = \frac{d_1}{h_{11}}$. Thus, with every iteration we learn a new diagonal element and reduce modulo the smallest possible principal minor. At the start of the algorithm we do not know the d_i for $i > 1$, which means the values are computed on the fly. After every row we learn h_{ii} and thus we compute $d_{i+1} = \frac{d_i}{h_{ii}}$ for the next iteration.

Lastly, as shown in Equation 3.66, the last nonzero element in row i is $a_{i,n+i} = d_i$. If we have set all $a_{ij} = 0$ for $i < j < n + i$ by using $\mathbf{U}_{ij}$ and computing modulo d_i, the elements a_{ii} and $a_{i,n+i} = d_i$ are the only nonzero elements in row i. At this point, a_{ii} is the result of the basic algorithm, but all steps were performed modulo d_i. Hence, if we now produce the last zero at $a_{i,n+i} = d_i$, according to the definition of $\mathbf{U}_{i,n+i}$, a_{ii} will become $\gcd(a_{ii}, d_i)$. Due to Proposition 3.65, this is the correct diagonal element of the HNF. This is the exact reason, why we operate on the modified matrix $\mathbf{A}$ instead of the input matrix $\mathbf{A}'$.

If we perform those procedures for every row, without altering the lattice, we will receive the HNF of the input matrix. This follows by induction over the principal submatrices, because after applying the above procedure to row j, the j-th principal submatrix is in HNF.

The above procedures for the Modulo Determinant Algorithm are shown in Algorithm 2. In line 5, the method *generateZero(r,c)* implements the matrix multiplication with $\mathbf{U}_{rc}$.

To conclude, computing the HNF modulo the determinant at least gives an upper bound on the size of the entries. Compared to the most basic algorithm, this is already a useful improvement. Since the Hadamard inequality implies super exponential growth of the determinant, this algorithm still has huge space complexity. Hence, the Linear Space Algorithm, discussed in Section 3.3, provides further improvements. But most importantly, if the determinant is known to be small, this algorithm accomplishes a tremendously fast runtime of $\mathcal{O}(n^3 \log^2 d)$. How to construct such matrices and use them to compute the HNF of arbitrary matrices will be discussed in Section 3.4.

Algorithm 2: Modulo Determinant Algorithm

Data: $\mathbf{A}' \in \mathbb{Z}^{n \times n}$, where $\det(\mathbf{A}') \neq 0$
Result: The HNF $\mathbf{H} \in \mathbb{Z}^{n \times n}$ of $\mathbf{A}'$

$d_1 \leftarrow |\det(\mathbf{A}')|$
$\mathbf{A} \leftarrow \begin{pmatrix} a'_{11} & \dots & a'_{1n} & d_1 & \dots & 0 \\ \vdots & & \vdots & \vdots & \ddots & \vdots \\ a'_{n1} & \dots & a'_{nn} & 0 & \dots & d_1 \end{pmatrix}$
for $r \leftarrow 1$ **to** n **do**
    for $c \leftarrow r+1$ **to** $n+r$ **do**
        $\mathbf{A}.generateZero(r, c)$
    for $c \leftarrow r$ **to** $n+r$ **do**
        for $t \leftarrow r$ **to** n **do**
            $a_{tc} \leftarrow a_{tc} \mod d_r$
    if $a_{rr} = 0$ **then**
        $a_{rr} = d_r$
    for $j \leftarrow 1$ **to** $r-1$ **do**
        for $t \leftarrow j+1$ **to** $r+1$ **do**
            $q \leftarrow \frac{a_{tj}}{a_{tt}}$
            for $k \leftarrow t$ **to** n **do**
                $a_{k,j+1} \leftarrow a_{k,j+1} - q \cdot a_{tt}$
                $a_{k,j+1} \leftarrow a_{k,j+1} \mod d_r$
    $d_{r+1} \leftarrow \frac{d_r}{a_{rr}}$
    $a_{r+1,n+r} \leftarrow d_{r+1}$
return H

3.3 Linear Space Algorithm

In this section we discuss the algorithm for computing the HNF of square matrices presented in [MW01]. Throughout this chapter we assume the input matrix to be of size $\mathcal{O}(n^2 \log M)$, where M is the maximal absolute value of the elements of the input matrix. The presented algorithm provably uses a linear amount of space in the input size, i.e. $\mathcal{O}(n^2 \log M)$, and further achieves a runtime of $\mathcal{O}(n^5 \text{polylog}(M, n))$.

3.3.1 Main Algorithm

The main part of the Linear Space Algorithm takes as input a nonsingular square matrix, where all principal minors are nonzero. In theory, this is not a restriction to the input, because for every nonsingular matrix, there exists a permutation of columns so that all principal minors are nonzero. In the appendix of [KB79b] an explicit method to compute this permutation is provided. Because during that method the principal minors need to be checked $\mathcal{O}(n^2)$ times, it has an overall runtime of $\mathcal{O}(n^5)$.
Permuting the columns does not alter the lattice and due to Theorem 3.59 the HNF of a lattice is unique. Therefore, the HNF is invariant under column permutation.

It further relies on the two procedures AddColumn and AddRow.

- AddColumn takes as input a matrix $\mathbf{A}$ in HNF and a column vector $\mathbf{b}$. It returns the HNF of the matrix $[\mathbf{A}|\mathbf{b}]$.
- AddRow works slightly different. It takes as input two square matrices $\mathbf{A}, \mathbf{H_A}$ and a row vector $\mathbf{a}^T$, where $\mathbf{H_A}$ is the HNF of $\mathbf{A}$ and $\mathbf{a}^T$ is the row we want to add. It returns the row vector $\mathbf{x}^T$ such that $\begin{bmatrix} \mathbf{H_A} \\ \mathbf{x}^T \end{bmatrix}$ is the HNF of $\begin{bmatrix} \mathbf{A} \\ \mathbf{a}^T \end{bmatrix}$.

Based on these two procedures, the idea is to call AddRow and AddColumn until we added all desired rows and columns so that the final call of AddColumn returns the correct HNF of the input matrix. In that sense, we consecutively compute the HNF of the principal submatrices.

Recall that $\mathbf{A}(i)$ denotes the i-th principal submatrix of $\mathbf{A} = (a_{ij})_{i,j\in[n]}$. Let $\mathbf{a}^T(i) = (\mathbf{a}_{i+1,1}, a_{i+1,2}, \ldots, a_{i+1,i})$ be the row vector, obtained by truncating the $(i+1)$-th row to its first i entries. Let $\mathbf{b}(i) = (a_{1,i}, a_{2,i}, \ldots, a_{i,i})$ denote the the first i elements of column i. Using this notation, the Linear Space Algorithm is shown in Algorithm 3.
Observe if $\mathbf{A}$ is of dimension $n = 1$, the for loop in line 5 never gets executed and $\mathbf{H} = \mathbf{A}(1)$ is returned. Since all diagonal elements must be strictly greater than zero in the HNF, we must ensure $\mathbf{A}(1) > 0$. This is the reason for lines 1 to 3, which just multiply the first column with -1 if necessary. Obviously, this does not alter the lattice.
Lines 1 to 3 are not included in [MW01], which makes their code only work for true matrices of dimension $n > 1$. Furthermore, their proof of correctness is technically wrong, because the base case for the induction does not hold if $\mathbf{A}(1) < 0$ and does not seem to be "obviously true" [1]. Even considering $n = 2$ the base case, AddRow would receive an incorrect input, because $\mathbf{H} = \mathbf{A}(1)$ would not be a correct HNF. Although $\mathbf{H}$ has the correct value after calling AddColumn, it is not obvious at all why this works.
Summing up, lines 1 to 3 ensure the formal and practical correctness for all matrices in $\mathbb{Z}^{n\times n}$ with $n \geq 1$.

[1] [MW01], p. 233

Algorithm 3: Linear Space Algorithm

Data: $\mathbf{A} \in \mathbb{Z}^{n\times n}$, where $\det(\mathbf{A}(i)) \neq 0 \ \ \forall i \in [n]$
Result: The HNF $\mathbf{H} \in \mathbb{Z}^{n\times n}$ of $\mathbf{A}$

1 **if** $\mathbf{A}_{1,1} < 0$ **then**
2 **for** $i \leftarrow 1$ **to** n **do**
3 $\mathbf{A}_{i,1} \leftarrow -\mathbf{A}_{i,1}$
4 $\mathbf{H} \leftarrow \mathbf{A}(1)$
5 **for** $i \leftarrow 2$ **to** n **do**
6 $x^T \leftarrow \text{AddRow}(\mathbf{A}(i-1), H, \mathbf{a}^T(i-1))$
7 $H \leftarrow \text{AddColumn}(\left[\begin{smallmatrix}\mathbf{H}\\ \mathbf{x}^T\end{smallmatrix}\right], \mathbf{b}(i))$
8 **return H**

Theorem 3.67. *Let $\mathbf{A} \in \mathbb{Z}^{n\times n}$ be a matrix where all principal minors are nonzero. Assuming AddRow and AddColumn work correctly, the Linear Space Algorithm returns the correct HNF of $\mathbf{A}$.*

Proof. For $i \in [n]$, let $\mathbf{H}_i$ denote the value of the variable $\mathbf{H}$ after $i-1$ iterations of the for loop in line 5 of Algorithm 3. This implies $\mathbf{H}_1 = \mathbf{A}(1)$. We now prove by induction that $\mathbf{H}_i$ is the HNF of $\mathbf{A}(i)$ for all i. Since $\mathbf{A}(n) = \mathbf{A}$, this includes $\mathbf{H}_n$ is the correct HNF of $\mathbf{A}$.
The base case $n = 1$ is true, because lines 1 to 3 ensure $\mathbf{A}(1) > 0$ and therefore all requirements to be in HNF are true and because multiplying the column with -1 does not change the lattice, $\mathbf{H}_1$ is the correct HNF of $\mathbf{A}(1)$.
For the inductive step we assume $\mathbf{H}_{i-1}$ is the correct HNF of $\mathbf{A}(i-1)$. In line 6, AddRow returns $\mathbf{x^T}$ such that

$$\begin{pmatrix}\mathbf{H}_{i-1}\\ \mathbf{x}^T\end{pmatrix} \tag{3.68}$$

is the correct HNf of

$$\begin{pmatrix}\mathbf{A}(i-1)\\ \mathbf{a}^T(i-1)\end{pmatrix}. \tag{3.69}$$

In line 7 AddColumn returns $\mathbf{H}_i$, which is by correctness of AddColumn the HNF of

$$\left(\begin{array}{c|c}\mathbf{A}(i-1) & \mathbf{b}(i-1)\\ \mathbf{a}^T(i-1) & \end{array}\right) = \mathbf{A}(i) \tag{3.70}$$

It follows, the return value $\mathbf{H}_n$ of Algorithm 3 is the correct HNF of $\mathbf{A}(n) = \mathbf{A}$. □

3.3.2 AddRow Procedure

The AddRow procedure takes as is input two nonsingular square matrices $\mathbf{A}, \mathbf{H_A} \in \mathbb{Z}^{n\times n}$ and a row vector $\mathbf{a}^T \in \mathbb{Z}^n$. $\mathbf{H_A}$ is the HNF of $\mathbf{A}$ and $\mathbf{a}^T$ is the row we want to add. The procedure returns a row vector $\mathbf{x}^T$ such that $\left[\begin{smallmatrix}\mathbf{H_A}\\ \mathbf{x}^T\end{smallmatrix}\right]$ is the HNF of $\left[\begin{smallmatrix}\mathbf{A}\\ \mathbf{a}^T\end{smallmatrix}\right]$.

$\mathbf{A}$ and $\mathbf{H_A}$ generate the same lattice, which implies there exists a unimodular matrix $\mathbf{U} \in \mathbb{Z}^{n\times n}$ satisfying $\mathbf{H_A} = \mathbf{AU}$. $\mathbf{U}$ is the solution of a system of linear equations defined by the entries of $\mathbf{A}$ and $\mathbf{H_A}$. As $\mathbf{A}$ and $\mathbf{H_A}$ are nonsingular, this system of linear equations has full

rank and exactly one solution. Therefore, $\mathbf{U}$ is unique.
Observe, the return value $\mathbf{x}^T$ has to fulfill $\mathbf{x}^T = \mathbf{a}^T\mathbf{U}$. By substituting $\mathbf{U}$ with $\mathbf{A}^{-1}\mathbf{H_A}$, we compute the result as $\mathbf{x}^T = \mathbf{a}^T\mathbf{A}^{-1}\mathbf{H_A}$. Unfortunately, the entries of $\mathbf{U} = \mathbf{A}^{-1}\mathbf{H_A}$ are potentially of the same magnitude as $\det(\mathbf{A})$, which would require an extensive amount of memory space to store all at the same time.
To tackle this problem, we compute $\mathbf{x}^T$ directly using the CRT as explained in the following four steps.

1. choose primes $p_1 \ldots, p_k$ such that $\prod_{i=1}^{k} p_i > 2 \cdot n^{n+1} M^{2n+1}$ and $|\det(\mathbf{A})| \neq 0 \mod p_i \ \forall i \in [k]$, where $M = \max(\|\mathbf{A}\|_\infty, \|\mathbf{a}^T\|_\infty)$.

Before we continue with the next step, let us briefly discuss why this bound is correct. We want to compute $\mathbf{x}^T$ with the CRT. Therefore, we need to bound the absolute values of the elements of $\mathbf{x}^T$. The starting point is $\mathbf{x}^T = \mathbf{a}^T\mathbf{A}^{-1}\mathbf{H_A}$. First, we want to bound the elements of the product $\mathbf{P} = \mathbf{A}^{-1}\mathbf{H_A} = (p_{ij})_{i,j\in[n]}$. For $M = \max(\|\mathbf{A}\|_\infty, \|\mathbf{a}^T\|_\infty)$, we bound the absolute value of the elements of $\mathbf{A}^{-1} = (a_{ij}^{(-1)})_{i,j\in[n]}$ by $L = (\sqrt{n}M)^n$. Additionally, without loss of generality we assume $\sum_{i=1}^{n} h_{ii} \leq \prod_{i=1}^{n} h_{ii}$. The only way for this equation not to hold, is when $h_{ii} = 1$. But in this case we just leave that column out and insert it later into our final result. We now bound the absolute value of p_{ij} by bounding the matrix product as

$$|p_{ij}| = |\sum_{l=1}^{n} a_{il}^{(-1)} h_{lj}| \leq \sum_{l=1}^{n} |a_{il}^{(-1)}| \cdot |h_{lj}| \leq \sum_{l=1}^{n} L h_{lj} = L \sum_{l=1}^{n} h_{ll} \leq L \prod_{i=1}^{n} h_{ll} \leq L^2. \tag{3.71}$$

We use the above result to bound the absolute entries of $\mathbf{x}^T = (x_i^T)_{i\in[n]} = \mathbf{a}^T\mathbf{P}$ by

$$|x_i^T| = |\sum_{l=1}^{n} a_l^T p_{il}| \leq \sum_{l=1}^{n} |a_l^T| \cdot |p_{il}| \leq \sum_{l=1}^{n} M \cdot L^2 = nML^2 = n^{n+1}M^{2n+1}. \tag{3.72}$$

Because we want to compute $\mathbf{x}^T$ in the range $-n^{n+1}M^{2n+1} \leq x_i^T \leq n^{n+1}M^{2n+1}$, we need to choose the primes to exceed $2 \cdot n^{n+1}M^{2n+1}$ as explained in Section 2.3.2. Lastly, we require $|\det(\mathbf{A})| \neq 0 \mod p_i$ to ensure that $\mathbf{A}$ stays nonsingular (e.g. $\left[\begin{smallmatrix} 2 & 0 \\ 0 & 2 \end{smallmatrix}\right]$ has full rank over $\mathbb{Q}$ but is singular over $\mathbb{F}_2$).

2. for every p_i compute the solution $\mathbf{y}_i$ to $\mathbf{A}^T\mathbf{y}_i = \mathbf{a} \mod p_i$.

3. for every prime p_i and solutions $\mathbf{y}_i$ compute $\mathbf{x}_i = \mathbf{H}_\mathbf{A}^T\mathbf{y}_i \mod p_i$.

To see this method yields the correct result, rewrite the equation from point 2 as $\mathbf{y}_i^T = \mathbf{a}^T\mathbf{A}^{-1} \mod p_i$. By rewriting the equation from point 3 we get $\mathbf{x}_i^T = \mathbf{y}_i^T\mathbf{H_A} = \mathbf{a}^T\mathbf{A}^{-1}\mathbf{H_A} \mod p_i$ as desired.

4. use the CRT on the $\mathbf{x}_i$ to compute $\mathbf{x}$ and return $\mathbf{x}^T$.

An upper bound on the entries of $\mathbf{x}^T$ was already computed in Equation 3.72 to be $n^{n+1}M^{2n+1}$. Thus, the bit size of the entire vector $\mathbf{x}^T$ and the overall memory space required by AddRow is

$$n((n+1)\log n + (2n+1)\log M) = \mathcal{O}(n^2 \log M), \tag{3.73}$$

which is linear in the input size, if we assume $M > n$.

For $V = \mathcal{O}(n^{n+1}M^{2n+1})$, we choose our primes such that $\prod_{i=1}^{k} p_i > 2V$. If p_{max} denotes the biggest chosen prime number, we get $(p_{max})^k > 2V$. This implies a rough estimate for the number of primes needed as $k = \mathcal{O}(\log V)$. Applying the Prime Number Theorem, the largest prime is of order $\mathcal{O}(\log V \log\log V)$. The runtime time mostly relies on solving the system of linear equations, which is done is $\mathcal{O}(n^3 \log^2 p_i)$ by using Gaussian elimination modulo p_i. Including the above results of the Prime Number Theorem, we get a time complexity of $\mathcal{O}(\log V)\mathcal{O}(n^3 \log^2(\log V \log\log V)) = \mathcal{O}(n^4 \text{polylog}(M, n))$ after expanding V.

For practical implementations it is useful to consider primes, which bit size is close to the size of the registers of the underlying computer architecture. The reason for that is, it makes almost no difference in runtime whether a single operand is 5-bit or 32-bit as long as it is less than the compute architecture. By doing so, we accomplish maximal efficiency, while keeping the amount of primes fairly small. For example, if we assume a 64-bit architecture, we choose primes between 2^{62} and $2^{64} - 1$.

The AddRow procedure is shown in Algorithm 4. In line 14, the method *linearSolveMod* returns the solution $\mathbf{y}$ of $\mathbf{A}^T\mathbf{y} = \mathbf{a} \mod p_i$. We continue with an example computation of the AddRow procedure.

Example 3.74. *The procedure is called with the parameters*

$$\mathbf{A} = \begin{pmatrix} 512 & 142 \\ 12 & 420 \end{pmatrix}, \qquad \mathbf{H_A} = \begin{pmatrix} 2 & 0 \\ 49584 & 106668 \end{pmatrix} \quad \textit{and} \quad \mathbf{a}^T = (983, 45). \tag{3.75}$$

We compute $d = |det(\mathbf{A})| = 213336$ *and* $M = \max(\|\mathbf{A}\|_\infty, \|\mathbf{a}^T\|_\infty) = 983$. *Therefore we need to find primes* $p_1, \ldots, p_k$ *such that their product exceeds* $2n^{n+1}M^{2n+1} = 2 \cdot 2^3 \cdot 983^5 = 7342730289481144$ *and every prime must not divide* d. *Hence, suitable primes for example are*

$$1031, 1033, 1039, 1049, 1051, 1061 \tag{3.76}$$

Continuing with the next step, we start with the prime $p_1 = 1031$ *and solve the system of equations*

$$\begin{pmatrix} 512 & 12 \\ 142 & 420 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \equiv \begin{pmatrix} 983 \\ 45 \end{pmatrix} \mod 1031 \tag{3.77}$$

We compute the solution $y_1 = 141$ *and* $y_2 = 166 \mod 1031$ *and set* $\mathbf{y}_1 = (141, 166)^T$. *With that solution we compute* $\mathbf{x}_1$ *as*

$$\mathbf{x}_1 = \begin{pmatrix} 2 & 49584 \\ 0 & 106668 \end{pmatrix} \begin{pmatrix} 141 \\ 166 \end{pmatrix} \equiv \begin{pmatrix} 2 & 96 \\ 0 & 475 \end{pmatrix} \begin{pmatrix} 141 \\ 166 \end{pmatrix} \equiv \begin{pmatrix} 753 \\ 494 \end{pmatrix} \mod 1031. \tag{3.78}$$

After repeating this procedure for all other primes, we get the results depicted in Table 1.

p_i	1031	1033	1039	1049	1051	1061
$\mathbf{y}_i^T$	$(141, 166)$	$(920, 513)$	$(615, 423)$	$(615, 504)$	$(228, 426)$	$(617, 370)$
$\mathbf{x}_i^T$	$(753, 494)$	$(807, 608)$	$(969, 950)$	$(190, 471)$	$(242, 583)$	$(502, 82)$

Table 1: Values for $\mathbf{y}_i$ and $\mathbf{x}_i$ for $1 \leq i \leq k = 6$

To conclude, we use the CRT to compute the entries of $\mathbf{x}$. *For the first entry we supply the*

Algorithm 4: AddRow

Data: $\mathbf{A}, \mathbf{H_A} \in \mathbb{Z}^{n\times n}$ and a row vector $\mathbf{a}^T \in \mathbb{Z}^n$, where $\mathbf{H_A}$ is the HNF of $\mathbf{A}$
Result: $\mathbf{x}^T$ s.t. $\begin{bmatrix} \mathbf{H_A} \\ \mathbf{x}^T \end{bmatrix}$ is the HNF of $\begin{bmatrix} \mathbf{A} \\ \mathbf{a}^T \end{bmatrix}$

$d \leftarrow |\det(\mathbf{A})|$
$M \leftarrow \max(\|\mathbf{A}\|_\infty, \|\mathbf{a}^T\|_\infty)$
$b \leftarrow n^{n+1} M^{2n+1}$
$product \leftarrow 1$
$p \leftarrow 2^{30}$
$k \leftarrow 1$
while $product \leq 2b$ **do**
  $p \leftarrow$nextPrime(p)
  if $d \not\equiv 0 \mod p$ **then**
    $p_i \leftarrow p$
    $product \leftarrow product \cdot p$
  $k \leftarrow k + 1$
$x\_vectors \leftarrow []$
for $i \leftarrow 1$ **to** k **do**
  $\mathbf{y} \leftarrow$linearSolveMod$(\mathbf{A}^T, \mathbf{a}, p_i)$
  $\mathbf{x} \leftarrow \mathbf{H}_\mathbf{A}^T \cdot \mathbf{y} \mod p_i$
  $x\_vectors$.append(x)
$\mathbf{x} \leftarrow []$
for $i \leftarrow 0$ **to** $n - 1$ **do**
  $values \leftarrow []$
  for $j \leftarrow 0$ **to** $k - 1$ **do**
    $values$.append$(x_vectors[j][i])$
  $result \leftarrow$ CRT$(values, p_1, \ldots p_k)$
  if $result > \lfloor \frac{prodcut}{2} \rfloor$ **then**
    $result \leftarrow result - product$
  $\mathbf{x}$.append$(result)$
return $\mathbf{x}^T$

values $(753, 807, 969, 190, 242, 502)$ *and the corresponding primes. The CRT yields* 1294398862103975699. *Because we expect a value centered around zero and the returned value is greater than* $\left\lfloor \frac{1}{2} \prod_{i=1}^{n} p_i \right\rfloor = 647199431052001391$, *we subtract the product and get* $\mathbf{x}_1 = -27084$.
For the second entry we supply the values $(494, 608, 950, 471, 583, 82)$ *together with the corresponding primes. The CRT yields* 1294398862103944510 *and after subtracting the product we get* $\mathbf{x}_2 = -58273$.
To sum up, the AddRow *procedure returns the value* $\mathbf{x}^T = (-27084, -58273)$, *such that*

$$\begin{pmatrix} 2 & 0 \\ 49584 & 106668 \\ -27084 & -58273 \end{pmatrix} \tag{3.79}$$

is the HNF of

$$\begin{pmatrix} 512 & 142 \\ 12 & 420 \\ 983 & 45 \end{pmatrix}. \tag{3.80}$$

We verify the result by computing $\mathbf{x}^T = \mathbf{a}^T\mathbf{A}^{-1}\mathbf{H}_\mathbf{A}$ *directly. This yields*

$$\mathbf{x}^T = \begin{pmatrix} 983 & 45 \end{pmatrix} \begin{pmatrix} \frac{420}{213336} & -\frac{12}{213336} \\ -\frac{142}{213336} & \frac{512}{213336} \end{pmatrix} \begin{pmatrix} 2 & 0 \\ 49584 & 106668 \end{pmatrix} = \begin{pmatrix} -27084 & -58273 \end{pmatrix} \tag{3.81}$$

as expected.

Looking ahead on potential enhancements, there are more efficient approaches to solve systems of linear equations based on fast matrix multiplication or p-adic expansions as proposed in [Dix82; MS99]. It is reasonable that these techniques are applicable in the context of AddRow as well, which would reduce the runtime by a factor of n to $\mathcal{O}(n^3\text{polylog}(M,n))$.

3.3.3 AddColumn Procedure

Throughout this section, we will revisit techniques applied in Section 3.2. The AddColumn procedure takes as input a matrix $\mathbf{A} \in \mathbb{Z}^{n\times n-1}$ in HNF and a column vector $\mathbf{b} \in \mathbb{Z}^n$. It returns the HNF $\mathbf{H} \in \mathbb{Z}^{n\times n}$ of $[\mathbf{A}|\mathbf{b}]$.

Proposition 2.30 implies $\mathbf{c} = (0,\ldots,0,d)^T \in \mathcal{L}([\mathbf{A}|\mathbf{b}])$, where $d = |\det([\mathbf{A}|\mathbf{b}])|$. First, we extend $\mathbf{A}$ to $\mathbf{H}_0 = [\mathbf{A}|\mathbf{c}]$, which is a lower triangular matrix in HNF. Based on this fact, AddColumn operates on the matrix $[\mathbf{H}_0|\mathbf{b}] \in \mathbb{Z}^{n\times n+1}$, which still generates the same lattice as $[\mathbf{A}|\mathbf{b}]$. It continues to generate pairs $\mathbf{H}_j, \mathbf{b}_j$ for $j \in \{0,\ldots,n\}$ satisfying

- $\mathbf{H}_0 = [\mathbf{A}|\mathbf{c}]$ and $\mathbf{b}_0 = \mathbf{b}$
- $\mathbf{H}_j$ is in HNF
- $\mathcal{L}([\mathbf{H}_j|\mathbf{b}_j]) = \mathcal{L}([\mathbf{H}_{j+1}|\mathbf{b}_{j+1}])$
- the first j entries of $\mathbf{b}_j$ are zero.

By induction it follows, the matrix $[\mathbf{H}_n|\mathbf{b}_n]$ generates the same lattice as $[\mathbf{A}|\mathbf{b}]$. Further, because $\mathbf{b}_n = \mathbf{0}$ and $\mathbf{H}_n$ is in HNF, $\mathbf{H}_n$ is the HNF of $[\mathbf{A}|\mathbf{b}]$.

The procedure starts with the matrix $[\mathbf{H}_0|\mathbf{b}_0]$. Let us have a look on how to get from $\mathbf{H}_{j-1}$ to $\mathbf{H}_j$, for every $j \in [n]$. We first apply the unimodular transform $\mathbf{U}_{j,n+1} \in \mathbb{Z}^{n+1\times n+1}$ (see Section 3.63) from the right to $[\mathbf{H}_{j-1}|\mathbf{b}_{j-1}]$ that modifies column j and the last column $\mathbf{b}_{j-1}$. Since during this procedure we only generate zeros in the last column, we denote $\mathbf{U}_{j,n+1}$ as $\mathbf{U}_j$ for a better readability. After this transformation, the j-th entry of $\mathbf{b}_{j-1}$ is zero. If the j-th entry of $\mathbf{b}_{j-1}$ was already zero beforehand, we would not apply $\mathbf{U}_j$, and continue with the reduction phase explained below.

Similar to Section 3.2, in this case the unimodular transformation is defined as

$$\mathbf{U}_{j,n+1} = \mathbf{U}_j = \begin{pmatrix} 1 & 0 & & & \dots & & 0 \\ 0 & \ddots & & & & & \vdots \\ & \ddots & 1 & & & & 0 \\ & & 0 & k & 0 & \dots & 0 & -\frac{\mathbf{b}_{j-1}^{(j)}}{g} \\ \vdots & & & 0 & 1 & 0 & \dots & 0 \\ & & & \vdots & \ddots & \ddots & & \vdots \\ & & & 0 & & \ddots & 1 & 0 \\ 0 & \dots & 0 & l & 0 & \dots & 0 & \frac{h_{jj}}{g} \end{pmatrix} \begin{array}{l} \\ \\ \\ \longleftarrow \text{row } j \\ \\ \\ \\ \\ \end{array} , \qquad (3.82)$$

$$\uparrow$$
$$\text{column } j$$

where $\mathbf{b}_{j-1}^{(j)}$ denotes the j-th element of $\mathbf{b}_{j-1}$ and h_{jj} is the diagonal element of $\mathbf{H}_{j-1}$ in row j. Moreover, k, l, g are computed with the EEA to satisfy

$$g = \gcd(h_{jj}, \mathbf{b}_{j-1}^{(j)}) = kh_{jj} + l\mathbf{b}_{j-1}^{(j)}. \qquad (3.83)$$

To see this transformation indeed vanishes the j-th entry of $\mathbf{b}_{j-1}$ consider the equation from the matrix multiplication for the resulting element $\mathbf{b}_j^{(j)}$.

$$\mathbf{b}_j^{(j)} = -\frac{\mathbf{b}_{j-1}^{(j)}}{g} h_{jj} + \mathbf{b}_{j-1}^{(j)} \frac{h_{jj}}{g} = 0. \qquad (3.84)$$

Moreover, in the actual implementation we do not need to store the whole matrix $\mathbf{U}_j$ and neither do we need to compute the whole product $[\mathbf{H}_{j-1}|\mathbf{b}_{j-1}] \cdot \mathbf{U}_j$. Because the unimodular transformation only alters column j and the last column $\mathbf{b}_{j-1}$, it suffices to implement the formulae for those two columns, given the four values from $\mathbf{U}_j$ that differ from the identity matrix.

Let m_k denote the determinant of the submatrix defined by the last $n - k + 1$ rows and columns of $\mathbf{H}_j$ for $k > j$. This means for example

$$m_n = \det(h_{n,n}) = h_{n,n} = d \quad \text{and} \quad m_{n-1} = \det \begin{pmatrix} h_{n-1,n-1} & 0 \\ h_{n,n-1} & h_{n,n} \end{pmatrix}. \qquad (3.85)$$

Because $\mathbf{H}_{j-1}$ is lower triangular, $|m_k|$ is the determinant of the lattice, generated by the last $n - k + 1$ columns. Due to Proposition 2.30, we reduce the entries of $\mathbf{H}_{j-1}$ in row k modulo $|m_k|$. This corresponds to subtracting a suitable multiple of the last $n-k+1$ columns. We use this fact during the matrix multiplication with $\mathbf{U}_j$ by computing every element of row k in the result modulo $|m_k|$.

The last part of this step from $\mathbf{H}_{j-1}$ to $\mathbf{H}_j$ is the reduction phase. Since we modified column j, we probably violated the condition of the HNF that every diagonal element is the largest in its row. To restore the HNF of $\mathbf{H}_{j-1}$ we reduce certain elements modulo the diagonal element of the corresponding row without altering the lattice in the following way. Since we only altered

column j there are two cases. First, the diagonal element h_{jj} of column j itself became smaller than the elements h_{ji} in row j ($i < j$). Thus, for every column i we subtract an appropriate multiple of column j to get $0 \leq h_{ji} < h_{jj}$. Because $\mathbf{H}_{j-1}$ is lower triangular, this does not alter the entries of the rows $1, \ldots, j-1$. Secondly, the entries h_{ij} ($i > j$) became larger than the diagonal elements h_{ii} to the right. We carry on like in the first case and subtract an appropriate multiple of column i from column j such that $0 \leq h_{ij} < h_{ii}$. Again, because $\mathbf{H}_{j-1}$ is lower triangular, subtracting column i does not alter the previously calculated values in the rows less than i. During this procedure, we additionally reduce the elements modulo the corresponding m_k to bound the memory space.
Therefore, we have restored the HNF condition of $\mathbf{H}_{j-1}$ and successfully computed $\mathbf{H}_j \leftarrow \mathbf{H}_{j-1}$ for the next iteration.

Unfortunately, this reduction phase is incorrectly stated in [MW01], where the Linear Space Algorithm was originally proposed. In their paper, the reduction phase only works on the columns $j, \ldots, n$. It is only considered, that after modifying column j, the entries of column j might be bigger than the diagonal element to the right of column j. But it also happens that the diagonal element in column j itself became smaller than the elements to the left of column j. Hence, we also need to reduce the columns $1, \ldots, j-1$.
The procedure for the reduction phase in [MW01] was taken from [Sto98], where, the diagonal elements never change. But in our case, the diagonal elements do change during the process, such that the reduction phase from [Sto98] is not entirely applicable and needs some further enhancement as explained in the paragraph above.

The AddColumn procedure is shown in Algorithm 5.
Unfortunately, the code of the AddColumn method presented in [MW01] is entirely wrong. Consequently, a corrected, but more complex version is depicted in Algorithm 5.
As first difference to the code in [MW01], observe in line 1 we define $\mathbf{c}$ with the absolute value of the determinant. If we did not use the absolute value, the computations of the m_k in line 5 would be wrong.
The second difference to the original code is line 8 to 13, because we compute x and y outside the for-loop. If we computed them inside the loop, as stated in [MW01], in the case $i = j$, we override $b_i = b_j$ in line 12, but this b_j is encoded in x. This would cause wrong results in the following iterations. Therefore, we must compute x and y outside the loop with the original value of b_j. Furthermore, in line 11 we store the value of b_i, because we override it in line 12, but must use it in line 13.
Moreover, in line 14, the if statement is added for the following reasons. If $b_j = 0$, the EEA returns $g = h_{jj}$, $r = 1$ and $s = 0$, which leads to $x = 0$ and $y = 1$. Therefore, if $j = n$ we reduce $h_{nn} \mod (m_n = h_{nn})$ in line 13, which sets $h_{nn} = 0$. This would cause an error in line 17, because we try to divide by h_{nn}.
Secondly, if $[\mathbf{A}|\mathbf{b}]$ is unimodular, we set $m_n = 1$ in line 3. Hence, in line 13 if $i = j = n$ we compute $h_{nn} \mod (m_n = 1) = 0$, which violates the HNF. This generally happens if $h_{jj} \leftarrow \gcd(h_{jj}, b_j)$ is a multiple of m_j. In that case we do not want to reduce $h_{jj} \mod m_j = 0$, but reduce to the smallest nonzero representative, which is m_j.
As last difference, lines 16 to 19 are added to reduce the columns left to column j, which is is missing in the reduction phase in [MW01] as explained in the paragraph above.

We continue with the analysis of the AddColumn procedure. First, let us have look at the space complexity. During each iteration of the for-loop in line 6, only column j of $\mathbf{H}$ and the

Algorithm 5: AddColumn

Data: $\mathbf{A} \in \mathbb{Z}^{n \times n-1}$ in HNF, $\mathbf{b} \in \mathbb{Z}^n$
Result: The HNF $\mathbf{H} \in \mathbb{Z}^{n \times n}$ of $[\mathbf{A}|\mathbf{b}]$

1 $\mathbf{c} \leftarrow (0, \ldots, 0, |\det([\mathbf{A}|\mathbf{b}]|)^T$
2 $\mathbf{H} = [\mathbf{A}|\mathbf{c}]$
3 $m_n \leftarrow h_{nn}$
4 **for** $i \leftarrow n-1$ **to** 1 **do**
5 $m_i \leftarrow m_{i+1} \cdot h_{i,i}$
6 **for** $j \leftarrow 1$ **to** n **do**
7 $g, r, s = \mathrm{EEA}(h_{j,j}, b_j)$
8 $x \leftarrow -\frac{b_j}{g}$
9 $y = \frac{h_{jj}}{g}$
10 **for** $i \leftarrow j$ **to** n **do**
11 $t \leftarrow b_i$
12 $b_i \leftarrow xh_{ij} + yb_i \mod m_i$
13 $h_{ij} \leftarrow rh_{ij} + st \mod m_i$
14 **if** $h_{jj} = 0$ **then**
15 $h_{jj} \leftarrow m_j$
16 **for** $c \leftarrow 1$ **to** j **do**
17 $q \leftarrow h_{jc} \operatorname{div} h_{jj}$
18 **for** $r \leftarrow j$ **to** n **do**
19 $h_{rc} \leftarrow h_{rc} - qh_{rj} \mod m_r$
20 **for** $k \leftarrow j+1$ **to** n **do**
21 $q \leftarrow h_{kj} \operatorname{div} h_{kk}$
22 **for** $l \leftarrow k$ **to** n **do**
23 $h_{lj} \leftarrow h_{lj} - qh_{lk} \mod m_l$
24 **return H**

vector $\mathbf{b}$ are modified. Both are reduced modulo the m_k. Hence, the required space is

$$\mathcal{O}\left(\sum_{k=1}^{n} \log m_k\right) = \mathcal{O}(n \log m_1). \tag{3.86}$$

since $m_1 = \max_k m_k$. Because $m_1 = \det(\mathbf{H})$ we have

$$\mathcal{O}(n \log m_1) = \mathcal{O}(n \log \det(\mathbf{H})) = \mathcal{O}(n \ \log M^n) = \mathcal{O}(n^2 \log M). \tag{3.87}$$

Because in the reduction phase in lines 16 to 23, we reduce the newly computed value modulo the diagonal elements from the input matrix, it follows the matrix needs overall memory space of $\mathcal{O}(n^2 \log M)$ again. Therefore, the space complexity of AddColumn is $\mathcal{O}(n^2 \log M)$.

Since the Linear Space Algorithm only uses AddRow and AddColumn as subprocedures, we already state at this point the whole algorithm uses $\mathcal{O}(n^2 \log M)$ space. This is the

same as the input size, which makes it a linear space algorithm.

Regarding the time complexity, the for-loop in line 20 is the triangulation procedure from [Sto98], which was proven to run in $\mathcal{O}(n \log^2 d)$, where $d = |\det(\mathbf{A})|$. In our case, we bound $\det(\mathbf{A}) = \mathcal{O}(M^n)$. Thus, lines 20 to 23 are in $\mathcal{O}(n \log^2 M^n)$. Lines 16 to 19 are essentially the same modulo operation, hence they are also in $\mathcal{O}(n \log^2 M^n)$.
The for-loop in line 10 adds a runtime of $\mathcal{O}(n^2 \log^2 M)$. Hence, we get a total runtime of

$$n(2 \cdot \mathcal{O}(n \log^2 M^n) + \mathcal{O}(n^2 \log^2 M)) = \mathcal{O}(n^4 \log^2 M). \tag{3.88}$$

We conclude this chapter with an example computation of the AddColumn procedure.

Example 3.89. *We compute the starting parameters for Example 3.74. Therefore, we call AddColumn with*

$$\mathbf{A} = \begin{pmatrix} 512 \\ 12 \end{pmatrix} \quad \text{and} \quad \mathbf{b} = \begin{pmatrix} 142 \\ 420 \end{pmatrix}. \tag{3.90}$$

We first compute $|\det([\mathbf{A}|\mathbf{b}])| = 512 \cdot 420 - 142 \cdot 12 = 213336$. *Therefore, we set*

$$\mathbf{c} = \begin{pmatrix} 0 \\ 213336 \end{pmatrix} \quad \text{and} \quad \mathbf{H}_0 = [\mathbf{A}|\mathbf{c}] = \begin{pmatrix} 512 & 0 \\ 12 & 213336 \end{pmatrix}. \tag{3.91}$$

We continue by computing the m_k *as* $m_2 = 213336$ *and* $m_1 = 213336 \cdot 512 = 109228032$.
We now start our iterations with $j = 1$ *and operate on the matrix*

$$[\mathbf{H}_0|\mathbf{b_0}] = \begin{pmatrix} 512 & 0 & 142 \\ 12 & 213336 & 420 \end{pmatrix}. \tag{3.92}$$

Because $\mathbf{b}_0^{(1)} = 142 \neq 0$ *we do not skip this step and compute*

$$g, k, l = EEA(h_{jj}, \mathbf{b}_{j-1}^{(j)}) = EEA(512, 142). \tag{3.93}$$

This returns $g = 2$, $k = -33$, $l = 119$. *With those values we define* $\mathbf{U}_1$ *as*

$$\mathbf{U}_1 = \begin{pmatrix} k & 0 & -\frac{\mathbf{b}_{j-1}^{(j)}}{g} \\ 0 & 1 & 0 \\ l & 0 & \frac{h_{jj}}{g} \end{pmatrix} = \begin{pmatrix} -33 & 0 & -71 \\ 0 & 1 & 0 \\ 119 & 0 & 256 \end{pmatrix}. \tag{3.94}$$

Next, we compute the product $[\mathbf{H}_0|\mathbf{b}_0] \cdot \mathbf{U}_1$ *and reduce row* k *modulo* m_k *in the result. This yields*

$$[\mathbf{H}_0|\mathbf{b}_0] \cdot \mathbf{U}_1 = \begin{pmatrix} 512 & 0 & 142 \\ 12 & 213336 & 420 \end{pmatrix} \begin{pmatrix} -33 & 0 & -71 \\ 0 & 1 & 0 \\ 119 & 0 & 256 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 49584 & 213336 & 106668 \end{pmatrix} \tag{3.95}$$

Since the result is already in HNF, we do not need to reduce any entries and set

$$[\mathbf{H}_1|\mathbf{b}_1] = \begin{pmatrix} 2 & 0 & 0 \\ 49584 & 213336 & 106668 \end{pmatrix}. \tag{3.96}$$

We continue with the last iteration $j = 2$. *Again, because* $\mathbf{b}_1^{(2)} = 106668 \neq 0$, *we do not*

skip this iteration and continue with computing $g = 106668$, $k = 0$ *and* $l = 1$ *by executing* $EEA(213336, 106668)$. *Therefore,* $\mathbf{U}_2$ *is defined as*

$$\mathbf{U}_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 2 \end{pmatrix}. \tag{3.97}$$

Computing the product $[\mathbf{H}_1|\mathbf{b}_1] \cdot \mathbf{U}_2$ *and reducing modulo* m_k *yields*

$$\begin{pmatrix} 2 & 0 & 0 \\ 49584 & 213336 & 106668 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 2 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 49584 & 106668 & 0 \end{pmatrix}. \tag{3.98}$$

Since this is already in HNF, there is no further modulo reduction necessary. Hence, we conclude that

$$\begin{pmatrix} 2 & 0 \\ 49584 & 106668 \end{pmatrix} \tag{3.99}$$

is the HNF of

$$\begin{pmatrix} 512 & 142 \\ 12 & 420 \end{pmatrix}. \tag{3.100}$$

3.4 Heuristic Algorithm

In the Linear Space Algorithm, the hidden constants become quite large, which gets noticeable in practical implementations. Therefore, we finish this chapter on computing the HNF by heuristically enhancing the Linear Space Algorithm. As result, we keep the linear space complexity and achieve a speed up by a factor of n or potentially even n^2, outperforming all previously discussed algorithms.
The main idea is to execute the Modulo Determinant Algorithm on a matrix with a heuristically small determinant and then use the algorithms AddRow and AddColumn to compute the HNF. The heuristic algorithm is depicted in Algorithm 6.

Algorithm 6: Heuristic Algorithm

Data: $\mathbf{A} \in \mathbb{Z}^{n \times n}$
Result: The HNF $\mathbf{H} \in \mathbb{Z}^{n \times n}$ of $\mathbf{A}$

1 decompose $\mathbf{A}$ into $\left[\begin{array}{c|c|c} \mathbf{B} & \mathbf{c} & \mathbf{d} \\ \mathbf{b}^T & a_{n,n-1} & a_{nn} \end{array}\right]$
2 $d_1 \leftarrow \det([\mathbf{B}|\mathbf{c}])$
3 $d_2 \leftarrow \det([\mathbf{B}|\mathbf{d}])$
4 $g, k, l \leftarrow \mathrm{EEA}(d_1, d_2)$
5 $\mathbf{H}' \leftarrow \mathrm{HNFModD}([\mathbf{B}|k\mathbf{c} + l\mathbf{d}])$
6 $\mathbf{x}^T \leftarrow \mathrm{AddRow}([\mathbf{B}|k\mathbf{c} + l\mathbf{d}], \mathbf{H}', [\mathbf{b}^T|ka_{n,n-1} + la_{nn}])$
7 $\mathbf{H} \leftarrow \begin{bmatrix} \mathbf{H}' \\ \mathbf{x}^T \end{bmatrix}$
8 $\mathbf{H} \leftarrow \mathrm{AddColumn}(\mathbf{H}, \begin{bmatrix} \mathbf{c} \\ a_{n,n-1} \end{bmatrix})$
9 $\mathbf{H} \leftarrow \mathrm{AddColumn}(\mathbf{H}, \begin{bmatrix} \mathbf{d} \\ a_{nn} \end{bmatrix})$
10 **return H**

We start by decomposing the input matrix and computing d_1 and d_2. It is important to note that in this case, d_1 and d_2 must be the matrix determinants and not the lattice determinants, i.e. keeping the sign and not taking the absolute value. Because the determinant of a matrix is multilinear, in line 4 we get

$$\det([\mathbf{B}|k\mathbf{c} + l\mathbf{d}]) = \det([\mathbf{B}|k\mathbf{c}]) + \det([\mathbf{B}|l\mathbf{d}]) = kd_1 + ld_2 = g = \gcd(d_1, d_2), \tag{3.101}$$

which is usually very small as depicted in Figure 2. That is why in line 5 we compute the HNF $\mathbf{H}'$ of that matrix with the Modulo Determinant Algorithm. We continue with adding the similarly modified last row of the input matrix and terminate the algorithm after adding the last two columns.

This procedure yields the correct result since we computed the HNF of

$$\left[\begin{array}{c|c|c|c} \mathbf{B} & k\mathbf{c}+l\mathbf{d} & \mathbf{c} & \mathbf{d} \\ \mathbf{b}^T & ka_{n,n-1}+la_{nn} & a_{n,n-1} & a_{nn} \end{array}\right], \tag{3.102}$$

which generates the same lattice as the original matrix.

To see how large $g = \gcd(d_1, d_2)$ heuristically grows, we perform the following experiment. We generate a random integer matrix $\mathbf{B} \in \mathbb{Z}^{n \times n-1}$, two random integer vectors $\mathbf{c}, \mathbf{d} \in \mathbb{Z}^n$ and compute $\gcd(d_1, d_2)$. The result is shown in Figure 2, where $N(g)$ denotes the observed relative frequency of g. A more detailed experimental analysis is discussed in [MW01].

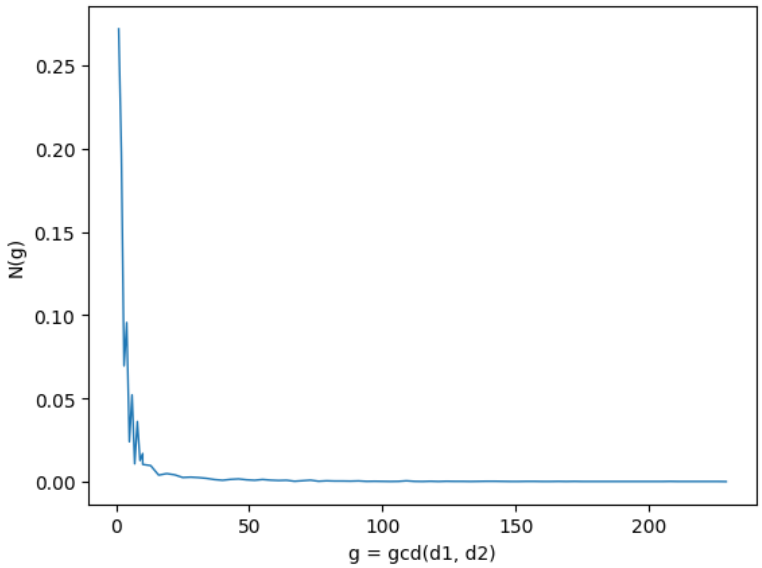


Figure 2: Frequency distribution of 10^4 samples of g for $n = 50$ and matrix entries bounded by 2^{30}.

86% of all samples yield a value for g less than 20. The largest entry observed for g was 247. Therefore, one may reasonably assume that $g = \det([\mathbf{B}|k\mathbf{c}+l\mathbf{d}])$ only needs one computer word of storage. In that case, by using the Modulo Determinant Algorithm in line 5, the space complexity has a tiny upper bound and is by far still linear. The runtime of $\mathcal{O}(n^3 \log^2 g)$ also gets really close to $\mathcal{O}(n^3)$. Even if g cannot be stored in one computer word, one simply applies the Heuristic Algorithm recursively. Hence, using this modified matrix as input to the Modulo Determinant Algorithm is the key component of this Heuristic Algorithm.

The main computational part consists of calling AddRow and AddColumn. But in the case where g is very small, AddColumn usually performs much better than its worst case runtime of $\mathcal{O}(n^4 \log^2 M)$. Since g is small, the corresponding HNF in line 5 also has a small determinant. Thus, the diagonal elements are small and the vast majority must be equal to 1. After calling AddRow this still holds for all rows, but the last one. There, the entries might be as big as the determinant of the original input matrix $\mathbf{A}$, while the $(n-1)$-th principal submatrix remains just slightly different from the identity matrix. In this case, the reduction phase in lines 16 to 23 of AddColumn is much faster. Reducing the first $n-1$ rows of one column only takes $\mathcal{O}(n)$ additions of small numbers that fit in one computer word, i.e. it takes constant time for

one addition. For the last row, which was appended with AddRow, it takes $\mathcal{O}(n)$ additions of values with a bit size up to $\mathcal{O}(\log \det(\mathbf{A})) = \mathcal{O}(n \log M)$. All in all reducing n columns, and thus the entire AddColumn procedure, takes $n(\mathcal{O}(n) + \mathcal{O}(n^2 \text{polylog } M)) = \mathcal{O}(n^3 \text{polylog } M)$.

Combining this with the runtime of $\mathcal{O}(n^4 \text{polylog}(M, n))$ for AddRow, the entire Heuristic Algorithm has a time complexity of $\mathcal{O}(n^4 \text{polylog}(M, n))$. Interestingly, this demonstrates the bottleneck of the Heuristic Algorithm to be the AddRow procedure and not the HNF computation of the $n - 1 \times n - 1$ submatrix.

If the methods to reduce the runtime of AddRow as proposed in Section 3.3.2 are applicable, the entire Heuristic Algorithm will have a linear space complexity of $\mathcal{O}(n^2 \log M)$ and a heuristic runtime of $\mathcal{O}(n^3 \text{polylog}(M, n))$. This would be the best known result till today.

4 Applications

After discussing how to efficiently compute the HNF, we proceed with an attack on the LWE problem. It is based on the primal lattice reduction attack and utilizes information leakage as well as the HNF to compute the intersection of two lattices. In [Dac+20], a similar attack was proposed, but the applied techniques lead to a tremendous runtime. By using a different method to compute the result, a much faster, but only heuristic runtime was presented in [MN23]. Building on top, we study a third technique, based on computing the HNF.

4.1 Attacking LWE with Perfect Hints

The main idea of this attack is to use information leakage, also called hints, about the secret $\mathbf{s}$ from the LWE instance. Such hints would usually be gathered via side-channel analysis. If we are given a set of hints, we want to encode them into the primal lattice reduction attack, in order to further reduce the dimension of the problem.

To start, let us recall how the primal lattice reduction attack works. Assume we are given an instance $(\mathbf{A}, \mathbf{b}, q)$ of the LWE-problem for $\mathbf{A} \in \mathbb{Z}^{n\times n}$ and $\mathbf{b} \in \mathbb{Z}^n$. We construct a lattice $\mathcal{L}(\mathbf{B})$ for a basis of dimension $2n+1 \times 2n+1$ defined as

$$\mathbf{B} = \begin{pmatrix} q\mathbf{I}_n & \mathbf{A} & \mathbf{b} \\ \mathbf{0} & \mathbf{I}_n & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & 1 \end{pmatrix}. \tag{4.103}$$

The key observation for the attack is now that for any $\mathbf{v} \in \mathbb{Z}^n$ it holds

$$\mathbf{B}\begin{pmatrix} \mathbf{v} \\ \mathbf{s} \\ -1 \end{pmatrix} = \begin{pmatrix} q\mathbf{v} + \mathbf{A}\mathbf{s} - \mathbf{b} \\ \mathbf{s} \\ -1 \end{pmatrix} \equiv \begin{pmatrix} -\mathbf{e} \\ \mathbf{s} \\ -1 \end{pmatrix} \mod q. \tag{4.104}$$

This implies, there exist a unique $\widetilde{\mathbf{v}} \in \mathbb{Z}^n$ so that the congruence becomes an equality and we have

$$\mathbf{B}\begin{pmatrix} \widetilde{\mathbf{v}} \\ \mathbf{s} \\ -1 \end{pmatrix} = \begin{pmatrix} -\mathbf{e} \\ \mathbf{s} \\ -1 \end{pmatrix} \in \mathcal{L}(\mathbf{B}). \tag{4.105}$$

Since the probability distribution for $\mathbf{s}$ and $\mathbf{e}$ typically has the property that the vectors have a very small norm, the probability for the above vector to be the shortest vector in $\mathcal{L}(\mathbf{B})$ is overwhelmingly high. For example, in the recently standardized key encapsulation mechanism Kyber [Bos+18], the probability distribution yields vectors where the absolute value of the entries are at most three.
Thus, to find the secret $\mathbf{s}$ of the LWE instance, it suffices to solve the SVP instance on $\mathcal{L}(\mathbf{B})$. The best currently known algorithm to solve SVP is the BKZ Algorithm [Sch87]. Its performance depends on two factors, the lattice dimension and the lattice-gap.

Definition 4.106. *Let $\mathcal{L}(\mathbf{B})$ be a lattice of dimension d. The **gap** of the lattice is defined as*

$$\frac{\lambda_1(\mathcal{L}(\mathbf{B}))}{\sqrt{d}\det(\mathbf{B})^{\frac{1}{d}}} \tag{4.107}$$

The next step is to use the information leakage to reduce the dimension and gap of the

lattice. In [Dac+20], different types of hints were defined, including perfect hints, modular hints, approximate hints. For this thesis, we discuss only the perfect hints. Therefore, let us first define how to model information leakage for the LWE problem.

Definition 4.108. *Let $\mathbf{s} \in \mathbb{Z}^n$ be the secret to an LWE-instance. A tuple $\overline{\mathbf{v}} = (\mathbf{v}, \ell) \in \mathbb{Z}^n \times \mathbb{Z}$ satisfying $\langle \mathbf{v}, \mathbf{s} \rangle = \ell$ is called a* ***perfect hint****.*

Now assume we a are given k linearly independent perfect hints $\overline{\mathbf{v}}_1, \ldots, \overline{\mathbf{v}}_k$. To integrate those hints into our attack we define a new vectorspace $\mathbf{H}$ as

$$\mathbf{H} = \bigcap_{i=1}^{k} \begin{pmatrix} \mathbf{0}^n \\ \mathbf{v}_i \\ \ell_i \end{pmatrix}^{\perp} = \left\{ \mathbf{w} \in \mathbb{R}^{2n+1} \;\middle|\; \left\langle \mathbf{w}, \begin{pmatrix} \mathbf{0}^n \\ \mathbf{v}_i \\ \ell_i \end{pmatrix} \right\rangle = 0 \;\; \forall i \in [k] \right\}. \tag{4.109}$$

Thus, $\mathbf{H}$ is the orthogonal complement to the space, spanned by all the hint-vectors $(\mathbf{0}^n, \mathbf{v}_i, \ell_i)^T$. Interestingly, the shortest vector from Equation 4.105 is also contained in $\mathbf{H}$, because

$$\left\langle \begin{pmatrix} -\mathbf{e} \\ \mathbf{s} \\ -1 \end{pmatrix}, \begin{pmatrix} \mathbf{0}^n \\ \mathbf{v}_i \\ \ell_i \end{pmatrix} \right\rangle = \langle \mathbf{0}^n, -\mathbf{e} \rangle + \langle \mathbf{s}, \mathbf{v}_i \rangle - \ell_i = 0 \;\; \forall i \in [k]. \tag{4.110}$$

Hence, $(-\mathbf{e}, \mathbf{s}, -1)^T \in \mathcal{L}(\mathbf{B}) \cap \mathbf{H}$. This lattice is of dimension $2n+1-k$, which shows, that with every single hint, we decrease the dimension of the lattice. It further has a smaller gap than the matrix $\mathbf{B}$. All in all, it is faster to use the BKZ algorithm to solve the SVP on $\mathcal{L}(\mathbf{B}) \cap \mathbf{H}$ instead of $\mathcal{L}(\mathbf{B})$.

Last but not least, in order to be able to apply BKZ, we need to compute a basis of $\mathcal{L}(\mathbf{B}) \cap \mathbf{H}$. Because $\mathbf{H}$ is the orthogonal complement to the vectorspace spanned by all hint-vectors, it is clear that

$$\mathbf{M} = \begin{pmatrix} \mathbf{0}^n & \ldots & \mathbf{0}^n \\ | & & | \\ \mathbf{v}_1 & \ldots & \mathbf{v}_k \\ | & & | \\ \ell_1 & \ldots & \ell_k \end{pmatrix} \in \mathbb{Z}^{2n+1 \times k} \tag{4.111}$$

is a basis for $\mathbf{H}^{\perp}$. From basic linear algebra it is known that $\mathbf{H} = \ker(\mathbf{M}^T)$. Therefore, by computing a basis of the kernel, we find a basis $\mathbf{X} \in \mathbb{Q}^{2n+1 \times 2n+1-k}$ of $\mathbf{H}$.

Given a basis $\mathbf{X}$ of $\mathbf{H}$, we are finally able to compute a basis of $\mathcal{L}(\mathbf{B}) \cap \mathbf{H}$ as follows. We compute the HNF of the matrix

$$\begin{pmatrix} \mathbf{B} & \mathbf{X} \\ \mathbf{B} & \mathbf{0} \end{pmatrix}, \tag{4.112}$$

which yields a matrix in the form

$$\begin{pmatrix} * & \mathbf{0} \\ * & \mathbf{R} \end{pmatrix}, \tag{4.113}$$

where $\mathbf{R}$ is the desired basis for $\mathcal{L}(\mathbf{B}) \cap \mathbf{H}$. This is indeed correct, because the HNF generates the $\mathbf{0}$ in the top right corner, by integer linear combinations of the columns of $\mathbf{B}$ and $\mathbf{X}$. If we write this down as

$$\alpha \mathbf{B} + \beta \mathbf{X} = \mathbf{0}. \tag{4.114}$$

and bring one part to the other side of the equation, it becomes clear that those linear combinations result in $\mathcal{L}(\mathbf{B}) \cap \mathcal{L}(\mathbf{X})$. Since $\mathbf{R}$ is the byproduct of exactly those transformations, it forms a basis for $\mathcal{L}(\mathbf{B}) \cap \mathbf{H}$. Furthermore, $\mathbf{X}$ is a matrix over $\mathbb{Q}$, but this is not a problem, because it is always possible to scale the matrix up to an integer matrix. Thus, we multiply $\mathbf{X}$ with the lowest common multiple of all denominators, compute the HNF of the resulting integer matrix and finish by scaling the matrix back again.

Comparing the runtime with [Dac+20], the calculations presented above run faster. Nevertheless, compared to [MN23], the above procedure is still tremendously slower. This is simply due to the complexity of computing the HNF. Furthermore, the matrix in Equation 4.112 is not a square matrix but is of dimension $4n + 2 \times 4n + 2 - k$. Hence, to compute the HNF of the entire matrix we first need to compute the HNF of a square submatrix and then add the left over rows by calling AddRow. But since AddRow is the part with the worst time complexity in both of the presented algorithms, this is a very bad input to compute the HNF. Actually, the more hints we know, the more non-square the matrix becomes and hence the computation of the HNF gets harder with every hint.

List of Algorithms

List of Figures

List of Tables

References

[ST67] Nicholas S. Szabó and Richard I. Tanaka. “Residue arithmetic and its applications to computer technology”. In: 1967. URL: https://api.semanticscholar.org/CorpusID:60825754.

[Fru77] M. A. Frumkin. “Polynomial time algorithms in the theory of linear diophantine equations”. In: *Fundamentals of Computation Theory*. Ed. by Marek Karpiński. Berlin, Heidelberg: Springer Berlin Heidelberg, 1977, pp. 386–392. ISBN: 978-3-540-37084-0.

[McC77] Michael T. McClellan. “The Exact Solution of Linear Equations with Rational Function Coefficients”. In: *ACM Trans. Math. Softw.* 3.1 (Mar. 1977), pp. 1–25. ISSN: 0098-3500. DOI: 10.1145/355719.355720. URL: https://doi.org/10.1145/355719.355720.

[KB79a] Ravindran Kannan and Achim Bachem. “Polynomial Algorithms for Computing the Smith and Hermite Normal Forms of an Integer Matrix”. In: *SIAM Journal on Computing* 8.4 (1979), pp. 499–507. DOI: 10.1137/0208040. eprint: https://doi.org/10.1137/0208040. URL: https://doi.org/10.1137/0208040.

[KB79b] Ravindran Kannan and Achim Bachem. “Polynomial Algorithms for Computing the Smith and Hermite Normal Forms of an Integer Matrix”. In: *SIAM Journal on Computing* 8.4 (1979), pp. 499–507. DOI: 10.1137/0208040. eprint: https://doi.org/10.1137/0208040. URL: https://doi.org/10.1137/0208040.

[Dix82] J.D. Dixon. “Exact Solution of Linear Equations Using P-Adic Expansions”. In: *Numerische Mathematik* 40 (1982), pp. 137–142. URL: http://eudml.org/doc/132821.

[Sch86] A. Schrijver. *Theory of Linear and Integer programming*. Wiley-Interscience, 1986.

[DKT87] P. D. Domich, R. Kannan, and L. E. Trotter. “Hermite Normal Form Computation Using Modulo Determinant Arithmetic”. In: *Mathematics of Operations Research* 12.1 (1987), pp. 50–59. ISSN: 0364765X, 15265471. URL: http://www.jstor.org/stable/3689672 (visited on 08/15/2023).

[Sch87] Claus-Peter Schnorr. “A Hierarchy of Polynomial Time Lattice Basis Reduction Algorithms”. In: *Theor. Comput. Sci.* 53 (1987), pp. 201–224. DOI: 10.1016/0304-3975(87)90064-8. URL: https://doi.org/10.1016/0304-3975(87)90064-8.

[Ili88] Costas S. Iliopoulos. “Worst-case complexity bounds on algorithms for computing the structure of finite abelian groups and Hermite and Smith normal forms of an integer matrix”. In: *SIAM Journal Computing* 18 (1988), pp. 131–141.

[Coh95] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Berlin, Heidelberg: Springer-Verlag, 1995. ISBN: 0387556400.

[FH97] Xin Gui Fang and George Havas. “On the Worst-Case Complexity of Integer Gaussian Elimination”. In: *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation*. ISSAC ’97. Kihei, Maui, Hawaii, USA: Association for Computing Machinery, 1997, pp. 28–31. ISBN: 0897918754. DOI: 10.1145/258726.258740. URL: https://doi.org/10.1145/258726.258740.

[Ajt98] Miklós Ajtai. “The shortest vector problem in L2 is NP-hard for randomized reductions (extended abstract)”. In: *Symposium on the Theory of Computing*. 1998. URL: https://api.semanticscholar.org/CorpusID:4503998.

[Sto98] Arne Storjohann. “Computing Hermite and Smith normal forms of triangular integer matrices”. In: *Linear Algebra and its Applications* 282 (1998), pp. 25–45. ISSN: 0024–3795. URL: http://dx.doi.org/10.1016/S0024-3795(98)10012-5.

[MS99] Thom Mulders and Arne Storjohann. “Diophantine Linear System Solving”. In: *Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation*. ISSAC ’99. Vancouver, British Columbia, Canada: Association for Computing Machinery, 1999, pp. 181–188. ISBN: 1581130732. DOI: 10.1145/309831.309905. URL: https://doi.org/10.1145/309831.309905.

[MW01] Daniele Micciancio and Bogdan Warinschi. “A Linear Space Algorithm for Computing the Hermite Normal Form”. In: *Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation*. ISSAC ’01. London, Ontario, Canada: Association for Computing Machinery, 2001, pp. 231–236. ISBN: 1581134177. DOI: 10.1145/384101.384133. URL: https://doi.org/10.1145/384101.384133.

[MG02] Daniele Micciancio and Shafi Goldwasser. *Complexity of Lattice Problems: a cryptographic perspective*. Vol. 671. The Kluwer International Series in Engineering and Computer Science. Boston, Massachusetts: Kluwer Academic Publishers, Mar. 2002.

[MR09] Daniele Micciancio and Oded Regev. “Lattice-based cryptography”. English (US). In: *Post-quantum cryptography*. Ed. by D.J. Bernstein and J. Buchmann. Springer, 2009.

[Pei09] Chris Peikert. “Public-Key Cryptosystems from the Worst-Case Shortest Vector Problem: Extended Abstract”. In: *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*. STOC ’09. Bethesda, MD, USA: Association for Computing Machinery, 2009, pp. 333–342. ISBN: 9781605585062. DOI: 10.1145/1536414.1536461. URL: https://doi.org/10.1145/1536414.1536461.

[Bos+18] Joppe Bos et al. “CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM”. In: *2018 IEEE European Symposium on Security and Privacy*. 2018, pp. 353–367. DOI: 10.1109/EuroSP.2018.00032.

[Dac+20] Dana Dachman-Soled et al. “LWE with Side Information: Attacks and Concrete Security Estimation”. In: *Advances in Cryptology – CRYPTO 2020*. Ed. by Daniele Micciancio and Thomas Ristenpart. Cham: Springer International Publishing, 2020, pp. 329–358. ISBN: 978-3-030-56880-1.

[Stu21] Christian Stump. *Lecture Notes on Algorithmic Mathematics*. 2021.

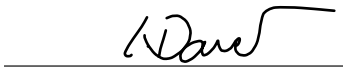
[LNP22] Yang Li, Kee Siong Ng, and Michael Purcell. *A Tutorial Introduction to Lattice-based Cryptography and Homomorphic Encryption*. 2022. arXiv: 2208.08125 [cs.CR].

[MN23] Alexander May and Julian Nowakowski. *Too Many Hints - When LLL Breaks LWE*. Cryptology ePrint Archive, Paper 2023/777. 2023. URL: https://eprint.iacr.org/2023/777.

Eidesstattliche Erklärung

Ich, erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule zur Erlangung eines akademischen Grades eingereicht habe.
Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.
Ich erkläre mich des Weiteren damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird

Bochum, 29. September 2023



Leon Damer