Computing the Hermite Normal Form: A Survey

Leon Damer

leon.damer@edu.rub.de

December 2024

Abstract

The Hermite Normal Form (HNF) of a matrix is an analogue of the echolon form over the integers. 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, also taken from [MW01], 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++ proof of concept implementations for all of the above algorithms are provided at

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

Besides some performance speed ups, the Linear Space Algorithm and Heuristic Algorithm are precisely the algorithms implemented by SageMath.

Contents

1 Preliminaries

1.1 Matrices

Definition 1.1. *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 1.2. *For an exemplary 3×3 matrix $\mathbf{A}$, the principal submatrices are*

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, \qquad \mathbf{A}(1) = \begin{pmatrix} 1 \end{pmatrix}, \qquad \mathbf{A}(2) = \begin{pmatrix} 1 & 2 \\ 4 & 5 \end{pmatrix}, \qquad \mathbf{A}(3) = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}. \tag{1.3}$$

Remark 1.4. *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 $\mathbf{A} = [\mathbf{c}_1, ..., \mathbf{c}_n]$.*

Definition 1.5. *The* ***i-th principal minor*** *of a matrix is the determinant of the i-th principal submatrix.*

Definition 1.6. *A square integer matrix $\mathbf{A} \in \mathbb{Z}^{n \times n}$ is called* **unimodular** *if*

$$|\det(\mathbf{A})| = 1. \tag{1.7}$$

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

Definition 1.8. *For a matrix $\mathbf{A} = (a_{ij})_{i \in [m], j \in [n]}$, we define the the* ***maximum norm*** *(ℓ_∞-norm) as*

$$\|\mathbf{A}\|_\infty = \max_{i \in [m], j \in [n]} |a_{ij}|. \tag{1.9}$$

1.2 Lattices

In this section we revisit some definitions and theorems on lattices. Contents of this section are partly based on [LNP22; MR09].

Proposition 1.10. *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 $\mathbf{A} = \mathbf{B}\mathbf{U}$.*

If $\mathbf{B}$ is a square matrix, it holds that $\det(\mathbf{B}) = \det(\mathbf{B}^T)$, which immediately implies $\det(\mathcal{L}(\mathbf{B})) = |\det(\mathbf{B})|$.

Proposition 1.11. *The determinant of a lattice does not depend on the basis.*

Proof. Let $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$ generate the same lattice. Due to Proposition 1.10, we know $\mathbf{A} = \mathbf{B}\mathbf{U}$ for $\mathbf{U} \in \mathrm{GL}(n, \mathbb{Z})$. Hence,

$$\sqrt{\det(\mathbf{A}^T\mathbf{A})} = \sqrt{\det(\mathbf{U}^T\mathbf{B}^T\mathbf{B}\mathbf{U})} = \sqrt{\det(\mathbf{U}^T)\det(\mathbf{B}^T\mathbf{B})\det(\mathbf{U})} = \sqrt{\det(\mathbf{B}^T\mathbf{B})}, \tag{1.12}$$

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.

Proposition 1.13. ***(Hadamard bound)*** *For* $\mathbf{A} = [\mathbf{c}_1, \dots, \mathbf{c}_n] \in \mathbb{R}^{m \times n}$*, the lattice determinant is bound by*

$$\det(\mathcal{L}(\mathbf{A})) \leq \prod_{i=1}^{n} \|\mathbf{c}_i\|_2. \tag{1.14}$$

Proposition 1.15. *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 $\mathbf{v}_j = (0, \dots, 0, d, 0, \dots, 0)^T$, 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

$$\begin{aligned} x_i &= \frac{\det(\mathbf{a}_1, \dots, \mathbf{a}_{i-1}, \mathbf{v}_j, \mathbf{a}_{i+1}, \dots, \mathbf{a}_n)}{\det(\mathbf{A})} \\ &= \frac{d \cdot \det(\mathbf{a}_1, \dots, \mathbf{a}_{i-1}, \mathbf{e}_j, \mathbf{a}_{i+1}, \dots, \mathbf{a}_n)}{\det(\mathbf{A})} \\ &= \pm\det(\mathbf{a}_1, \dots, \mathbf{a}_{i-1}, \mathbf{e}_j, \mathbf{a}_{i+1}, \dots, \mathbf{a}_n) \in \mathbb{Z}, \end{aligned} \tag{1.16}$$

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 and the map $f(\mathbf{v}) = \mathbf{v} \mod d$ is defined by the corresponding projection.

1.3 Number Theory

Theorem 1.17. ***(Prime Number Theorem)*** *Let* $\pi(x)$ *denote the amount of prime numbers less or equal to* x*.* $\pi(x)$ *is bounded by*

$$\pi(x) = \mathcal{O}\left(\frac{x}{\log x}\right). \tag{1.18}$$

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 Computing the Hermite Normal Form

Let us have a look at 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 the three different algorithms.

2.1 Hermite Normal Form

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

Definition 2.19. *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 2.20. *Let us have a look at some matrices in HNF to get familiar with the definition.*

$$\mathbf{H}_1 = \begin{pmatrix} 12 & 0 & 0 \\ 0 & 1 & 0 \\ 15 & 3 & 644 \end{pmatrix}, \quad \mathbf{H}_2 = \begin{pmatrix} 2 & 0 \\ 76 & 89 \\ -5352 & -9 \end{pmatrix}, \quad \mathbf{H}_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 155 & 35654 & 543644 \end{pmatrix} \tag{2.21}$$

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

Theorem 2.22. *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 2.4. 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 $h_{ij} - h'_{ij} = zh_{ii}$ 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, $\mathbf{H} = \mathbf{H}'$. $\square$

2.2 Basic 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 2.19)

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 2.19)

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

$$\mathbf{A}\mathbf{U}_{1n} = \begin{pmatrix} a_{11} & \dots & a_{1,n-1} & a_{1n} \\ \vdots & & & \vdots \\ a_{n1} & \dots & \dots & a_{nn} \end{pmatrix} \cdot \mathbf{U}_{1n} = \begin{pmatrix} \gcd(a_{11}, a_{1n}) & \dots & a_{1,n-1} & 0 \\ \vdots & & & \vdots \\ \widetilde{a}_{n1} & \dots & \dots & \widetilde{a}_{nn} \end{pmatrix} \tag{2.24}$$

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

$$u_{rr} = k, \qquad u_{cr} = l, \qquad u_{rc} = -\frac{a_{rc}}{g}, \qquad u_{cc} = \frac{a_{rr}}{g}, \tag{2.25}$$

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

$$\mathbf{U}_{rc} = \begin{pmatrix} 1 & & & \dots & & & 0 \\ & \ddots & & & & & \\ & & k & \dots & -\frac{a_{rc}}{g} & & \\ & & & 1 & & & \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ & & & & 1 & & \\ & & l & \dots & \frac{a_{rr}}{g} & & \\ & & & & & \ddots & \\ 0 & & & \dots & & & 1 \end{pmatrix} \begin{array}{l} \\ \\ \longleftarrow \text{row } r \\ \\ \\ \\ \longleftarrow \text{row } c \\ \\ \\ \end{array} . \tag{2.26}$$
$$\qquad\qquad\qquad\qquad \uparrow \text{ column } r \qquad \uparrow \text{ column } c$$

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

$$\det(\mathbf{U}_{rc}) = k\det(\mathbf{U}_{rc}^{r,r}) + l\det(\mathbf{U}_{rc}^{r,c}) = k\frac{a_{rr}}{g} + l\frac{a_{rc}}{g} = 1 \tag{2.27}$$

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 $r+1, \ldots, c-1$ 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.

2.3 Modulo Determinant Algorithm

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 2.28. *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

$$\mathbf{A} = (a_{ij})_{i\in[n],j\in[2n]} = \begin{pmatrix} a'_{11} & \dots & a'_{1n} & d_1 & \dots & 0 \\ \vdots & & \vdots & \vdots & \ddots & \vdots \\ a'_{n1} & \dots & a'_{nn} & 0 & \dots & d_n \end{pmatrix}, \tag{2.29}$$

where the d_i are defined as in Proposition 2.28. According to Proposition 1.15, $\mathcal{L}(\mathbf{A}) = \mathcal{L}(\mathbf{A}')$. 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 2.29, 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 2.28, 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 1.

Algorithm 1: 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 $\mathbf{A}'$

1 $d_1 \leftarrow |\det(\mathbf{A}')|$

2 $\mathbf{A} \leftarrow \begin{pmatrix} a'_{11} & \dots & a'_{1n} & d_1 & \dots & 0 \\ \vdots & & \vdots & \vdots & \ddots & \vdots \\ a'_{n1} & \dots & a'_{nn} & 0 & \dots & d_1 \end{pmatrix}$

3 **for** $r \leftarrow 1$ **to** n **do**
4   **for** $c \leftarrow r+1$ **to** $n+r$ **do**
5         $\mathbf{A}.generateZero(r, c)$
6   **for** $c \leftarrow r$ **to** $n+r$ **do**
7   **for** $t \leftarrow r$ **to** n **do**
8             $a_{tc} \leftarrow a_{tc} \mod d_r$
9   **if** $a_{rr} = 0$ **then**
10         $a_{rr} = d_r$
11   **for** $j \leftarrow 1$ **to** $r-1$ **do**
12   **for** $t \leftarrow j+1$ **to** $r+1$ **do**
13             $q \leftarrow \frac{a_{tj}}{a_{tt}}$
14   **for** $k \leftarrow t$ **to** n **do**
15                 $a_{k,j+1} \leftarrow a_{k,j+1} - q \cdot a_{tt}$
16                 $a_{k,j+1} \leftarrow a_{k,j+1} \mod d_r$
17     $d_{r+1} \leftarrow \frac{d_r}{a_{rr}}$
18     $a_{r+1,n+r} \leftarrow d_{r+1}$
19 **return H**

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 2.4, 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 2.5.

2.4 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 $\mathcal{O}(n^5 \text{polylog}(M, n))$.

2.4.1 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 2.22 the HNF of a lattice is unique. Therefore, the HNF is invariant under column permutation.

The algorithm 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.

Remark 2.30. *By utilizing AddRow and AddColumn we are able to compute the HNF of arbitrary (singular) matrices. We do so by first removing the linearly dependent columns and rows and then computing the HNF on this nonsingular square matrix. Afterwards we add the removed cloumns and rows back using the above methods.*

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 2.

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.

Algorithm 2: 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     $x^T \leftarrow$ ADDROW$(\mathbf{A}(i-1), H, \mathbf{a}^T(i-1))$
7     $H \leftarrow$ ADDCOLUMN$(\begin{bmatrix} \mathbf{H} \\ \mathbf{x}^T \end{bmatrix}, \mathbf{b}(i))$
8 **return H**

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 $\mathbf{A}(1) < 0$. 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 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 $n \geq 1$.

Theorem 2.31. *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 2. 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

$$\begin{pmatrix} \mathbf{H}_{i-1} \\ \mathbf{x}^T \end{pmatrix} \tag{2.32}$$

is the correct HNf of

$$\begin{pmatrix} \mathbf{A}(i-1) \\ \mathbf{a}^T(i-1) \end{pmatrix}. \tag{2.33}$$

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

$$\left(\begin{array}{c|c} \mathbf{A}(i-1) & \mathbf{b}(i-1) \\ \mathbf{a}^T(i-1) & \end{array} \right) = \mathbf{A}(i) \tag{2.34}$$

It follows, the return value $\mathbf{H}_n$ of Algorithm 2 is the correct HNF of $\mathbf{A}(n) = \mathbf{A}$. □

2.4.2 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 $\begin{bmatrix} \mathbf{H_A} \\ \mathbf{x}^T \end{bmatrix}$ is the HNF of $\begin{bmatrix} \mathbf{A} \\ \mathbf{a}^T \end{bmatrix}$.

$\mathbf{A}$ 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 $p_1 \dots, p_k$ 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

$$|p_{ij}| = |\sum_{l=1}^n a_{il}^{(-1)} h_{lj}| \leq \sum_{l=1}^n |a_{il}^{(-1)}| \cdot |h_{lj}| \leq \sum_{l=1}^n L h_{lj} = L \sum_{l=1}^n h_{ll} \leq L \prod_{i=1}^n h_{ll} \leq L^2. \tag{2.35}$$

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

$$|x_i^T| = |\sum_{l=1}^n a_l^T p_{il}| \leq \sum_{l=1}^n |a_l^T| \cdot |p_{il}| \leq \sum_{l=1}^n M \cdot L^2 = nML^2 = n^{n+1}M^{2n+1}. \tag{2.36}$$

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 **??**. Lastly, we require $|\det(\mathbf{A})| \neq 0 \mod p_i$ to ensure that $\mathbf{A}$ stays nonsingular (e.g. $\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$ 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 $\mathbf{x}_i^T = \mathbf{y}_i^T\mathbf{H}_\mathbf{A} = \mathbf{a}^T\mathbf{A}^{-1}\mathbf{H}_\mathbf{A} \mod p_i$ 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 2.36 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

$$n((n+1)\log n + (2n+1)\log M) = \mathcal{O}(n^2 \log M), \tag{2.37}$$

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.

The AddRow procedure is shown in Algorithm 3. 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 2.38. *The procedure is called with the parameters*

$$\mathbf{A} = \begin{pmatrix} 512 & 142 \\ 12 & 420 \end{pmatrix}, \qquad \mathbf{H}_\mathbf{A} = \begin{pmatrix} 2 & 0 \\ 49584 & 106668 \end{pmatrix} \quad \textit{and} \quad \mathbf{a}^T = (983, 45). \tag{2.39}$$

We compute $d = |det(\mathbf{A})| = 213336$ *and* $M = \max(\|\mathbf{A}\|_\infty, \|\mathbf{a}^T\|_\infty) = 983$. *Therefore we need to find primes* $p_1, \dots, p_k$ *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*

$$1031, 1033, 1039, 1049, 1051, 1061 \tag{2.40}$$

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

$$\begin{pmatrix} 512 & 12 \\ 142 & 420 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \equiv \begin{pmatrix} 983 \\ 45 \end{pmatrix} \mod 1031 \tag{2.41}$$

Algorithm 3: 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 ← |det(A)|
M ← max(‖A‖_∞, ‖a^T‖_∞)
b ← n^(n+1) M^(2n+1)
product ← 1
p ← 2^30
k ← 1
while product ≤ 2b do
    p ←nextPrime(p)
    if d ≢ 0  mod p then
        p_i ← p
        product ← product · p
    k ← k + 1
x_vectors ← []
for i ← 1 to k do
    y ←linearSolveMod(A^T, a, p_i)
    x ← H_A^T · y  mod p_i
    x_vectors.append(x)
x ← []
for i ← 0 to n − 1 do
    values ← []
    for j ← 0 to k − 1 do
        values.append(x_vectors[j][i])
    result ← CRT(values, p_1, . . . p_k)
    if result > ⌊prodcut/2⌋ then
        result ← result − product
    x.append(result)
return x^T
```

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*

$$\mathbf{x}_1 = \begin{pmatrix} 2 & 49584 \\ 0 & 106668 \end{pmatrix} \begin{pmatrix} 141 \\ 166 \end{pmatrix} \equiv \begin{pmatrix} 2 & 96 \\ 0 & 475 \end{pmatrix} \begin{pmatrix} 141 \\ 166 \end{pmatrix} \equiv \begin{pmatrix} 753 \\ 494 \end{pmatrix} \mod 1031. \tag{2.42}$$

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 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*

$$\begin{pmatrix} 2 & 0 \\ 49584 & 106668 \\ -27084 & -58273 \end{pmatrix} \tag{2.43}$$

is the HNF of

$$\begin{pmatrix} 512 & 142 \\ 12 & 420 \\ 983 & 45 \end{pmatrix}. \tag{2.44}$$

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

$$\mathbf{x}^T = \begin{pmatrix} 983 & 45 \end{pmatrix} \begin{pmatrix} \frac{420}{213336} & -\frac{12}{213336} \\ -\frac{142}{213336} & \frac{512}{213336} \end{pmatrix} \begin{pmatrix} 2 & 0 \\ 49584 & 106668 \end{pmatrix} = \begin{pmatrix} -27084 & -58273 \end{pmatrix} \tag{2.45}$$

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 $\mathcal{O}(n^3 \text{polylog}(M, n))$.

2.4.3 AddColumn Procedure

Throughout this section, we will revisit techniques applied in Section 2.3. 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 1.15 implies $\mathbf{c} = (0, \dots, 0, d)^T \in \mathcal{L}([\mathbf{A}|\mathbf{b}])$, 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 $j \in \{0, \dots, n\}$ 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 2.26) 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 2.3, in this case the unimodular transformation is defined as

$$\mathbf{U}_{j,n+1} = \mathbf{U}_j = \begin{pmatrix} 1 & 0 & & & \cdots & & & 0 \\ 0 & \ddots & & & & & & \vdots \\ & \ddots & 1 & & & & & 0 \\ & & 0 & k & 0 & \cdots & 0 & -\frac{\mathbf{b}_{j-1}^{(j)}}{g} \\ \vdots & & & 0 & 1 & 0 & \cdots & 0 \\ & & & \vdots & \ddots & \ddots & & \vdots \\ & & & 0 & & \ddots & 1 & 0 \\ 0 & \cdots & 0 & l & 0 & \cdots & 0 & \frac{h_{jj}}{g} \end{pmatrix} \begin{matrix} \\ \\ \\ \longleftarrow \text{row } j \\ \\ \\ \\ \\ \end{matrix}, \qquad \begin{matrix} \uparrow \\ \text{column } j \end{matrix} \tag{2.46}$$

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

$$g = \gcd(h_{jj}, \mathbf{b}_{j-1}^{(j)}) = k h_{jj} + l \mathbf{b}_{j-1}^{(j)}. \tag{2.47}$$

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 $\mathbf{b}_j^{(j)}$.

$$\mathbf{b}_j^{(j)} = -\frac{\mathbf{b}_{j-1}^{(j)}}{g} h_{jj} + \mathbf{b}_{j-1}^{(j)} \frac{h_{jj}}{g} = 0. \tag{2.48}$$

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

$$m_n = \det(h_{n,n}) = h_{n,n} = d \;\text{ and }\; m_{n-1} = \det\begin{pmatrix} h_{n-1,n-1} & 0 \\ h_{n,n-1} & h_{n,n} \end{pmatrix}. \tag{2.49}$$

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 1.15, 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 stated incorrectly 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 4.
Unfortunately, the code of the AddColumn method presented in [MW01] is not entirely correct, although the ideas are. Consequently, a corrected, but more complex version is depicted in Algorithm 4.

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

$$\mathcal{O}\left(\sum_{k=1}^{n} \log m_k\right) = \mathcal{O}(n \log m_1) \tag{2.50}$$

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

$$\mathcal{O}(n \log m_1) = \mathcal{O}(n \log \det(\mathbf{H})) = \mathcal{O}(n\ \log M^n) = \mathcal{O}(n^2 \log M). \tag{2.51}$$

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

$$n(2 \cdot \mathcal{O}(n \log^2 M^n) + \mathcal{O}(n^2 \log^2 M)) = \mathcal{O}(n^4 \log^2 M). \tag{2.52}$$

Algorithm 4: 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 $[\mathbf{A}|\mathbf{b}]$

```
c ← (0, ..., 0, |det([A|b])|)^T
H = [A|c]
m_n ← h_nn
for i ← n − 1 to 1 do
    m_i ← m_{i+1} · h_{i,i}
for j ← 1 to n do
    g, r, s = EEA(h_{j,j}, b_j)
    x ← −b_j / g
    y = h_jj / g
    for i ← j to n do
        t ← b_i
        b_i ← x h_ij + y b_i  mod m_i
        h_ij ← r h_ij + s t  mod m_i
    if h_jj = 0 then
        h_jj ← m_j
    for c ← 1 to j do
        q ← h_jc div h_jj
        for r ← j to n do
            h_rc ← h_rc − q h_rj  mod m_r
    for k ← j + 1 to n do
        q ← h_kj div h_kk
        for l ← k to n do
            h_lj ← h_lj − q h_lk  mod m_l
return H
```

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

Example 2.53. *We compute the starting parameters for Example 2.38. Therefore, we call* ADDCOLUMN *with*

$$\mathbf{A} = \begin{pmatrix} 512 \\ 12 \end{pmatrix} \quad \text{and} \quad \mathbf{b} = \begin{pmatrix} 142 \\ 420 \end{pmatrix}. \tag{2.54}$$

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

$$\mathbf{c} = \begin{pmatrix} 0 \\ 213336 \end{pmatrix} \quad \text{and} \quad \mathbf{H}_0 = [\mathbf{A}|\mathbf{c}] = \begin{pmatrix} 512 & 0 \\ 12 & 213336 \end{pmatrix}. \tag{2.55}$$

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*

$$[\mathbf{H}_0|\mathbf{b_0}] = \begin{pmatrix} 512 & 0 & 142 \\ 12 & 213336 & 420 \end{pmatrix}. \tag{2.56}$$

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

$$g, k, l = EEA(h_{jj}, \mathbf{b}_{j-1}^{(j)}) = EEA(512, 142). \tag{2.57}$$

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

$$\mathbf{U}_1 = \begin{pmatrix} k & 0 & -\frac{\mathbf{b}_{j-1}^{(j)}}{g} \\ 0 & 1 & 0 \\ l & 0 & \frac{h_{jj}}{g} \end{pmatrix} = \begin{pmatrix} -33 & 0 & -71 \\ 0 & 1 & 0 \\ 119 & 0 & 256 \end{pmatrix}. \tag{2.58}$$

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*

$$[\mathbf{H}_0|\mathbf{b}_0] \cdot \mathbf{U}_1 = \begin{pmatrix} 512 & 0 & 142 \\ 12 & 213336 & 420 \end{pmatrix} \begin{pmatrix} -33 & 0 & -71 \\ 0 & 1 & 0 \\ 119 & 0 & 256 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 49584 & 213336 & 106668 \end{pmatrix} \tag{2.59}$$

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

$$[\mathbf{H}_1|\mathbf{b}_1] = \begin{pmatrix} 2 & 0 & 0 \\ 49584 & 213336 & 106668 \end{pmatrix}. \tag{2.60}$$

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*

$$\mathbf{U}_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 2 \end{pmatrix}. \tag{2.61}$$

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

$$\begin{pmatrix} 2 & 0 & 0 \\ 49584 & 213336 & 106668 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 2 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 49584 & 106668 & 0 \end{pmatrix}. \tag{2.62}$$

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

$$\begin{pmatrix} 2 & 0 \\ 49584 & 106668 \end{pmatrix} \tag{2.63}$$

is the HNF of

$$\begin{pmatrix} 512 & 142 \\ 12 & 420 \end{pmatrix}. \tag{2.64}$$

2.5 Heuristic Algorithm

In the Linear Space Algorithm, the hidden constants become quite large, which gets noticeable in practical implementations. Therefore, in [MW01] the Linear Space Algorithm and Modular Determinant Algorithm are combined, based on heurisitc measurements. As result, the authors 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 5.

Algorithm 5: 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

$$\det([\mathbf{B}|k\mathbf{c} + l\mathbf{d}]) = \det([\mathbf{B}|k\mathbf{c}]) + \det([\mathbf{B}|l\mathbf{d}]) = kd_1 + ld_2 = g = \gcd(d_1, d_2), \qquad (2.65)$$

which is usually very small as depicted in Figure **??**. 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

$$\left[\begin{array}{c|c|c|c} \mathbf{B} & k\mathbf{c} + l\mathbf{d} & \mathbf{c} & \mathbf{d} \\ \mathbf{b}^T & ka_{n,n-1} + la_{nn} & a_{n,n-1} & a_{nn} \end{array}\right], \tag{2.66}$$

which generates the same lattice as the original matrix.

Based on heuristics, 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 2.4.2 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.

References

[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.

[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).

[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.

[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.

[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.

[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.

[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].