Electronic Colloquium on Computational Complexity, Report No. 74 (2000)

A Linear Space Algorithm for Computing the Hermite Normal Form

Daniele Micciancio Bogdan Warinschi

Department of Computer Science and Engineering
University of California, San Diego
{daniele,bogdan}@cs.ucsd.edu

July 11, 2000

Abstract

Computing the Hermite Normal Form of an $n \times n$ matrix using the best current algorithms typically requires $O(n^3 \log M)$ space, where M is a bound on the length of the columns of the input matrix. Although polynomial in the input size (which is $O(n^2 \log M)$), this space blow-up can easily become a serious issue in practice when working on big integer matrices. In this paper we present a new algorithm for computing the Hermite Normal Form which uses only $O(n^2 \log M)$ space (i.e., essentially the same as the input size). When implemented using standard integer arithmetic, our algorithm has the same time complexity of the asymptotically fastest (but space inefficient) algorithms. We also suggest simple heuristics that when incorporated in our algorithm result in essentially the same asymptotic running time of the theoretically fastest solutions, still maintaining our algorithm extremely practical.

1 Introduction

The *Hermite Normal Form* is a standard form for matrices that is useful in many applications. For the special case of square matrices (see next section for general definition), a matrix $\mathbf{A}$ is in Hermite Normal Form if it is lower triangular, all of its nonzero entries are positive, and each off-diagonal entry is reduced modulo the corresponding diagonal entry on the same row. A classical result of Hermite [9] says that every integer matrix $\mathbf{A}$ is (column) equivalent to a matrix $\mathbf{H}$ which is in Hermite Normal Form. Here, by (column) equivalent we mean that $\mathbf{H}$ can be obtained from $\mathbf{A}$ by a sequence of elementary columns operations, or, equivalently, the two matrices generate the same set of vectors as integer linear combinations of their columns.

The Hermite Normal Form plays a fundamental role in many applications because of its triangular structure. In particular, it has been used in solving systems of linear Diophantine equations [6], algorithmic problems in lattices [7], cryptographic schemes [13], integer programming [10], loop optimizations techniques [14]. Not surprisingly, finding efficient algorithms for computing the Hermite Normal Form has been a very active research area, and lot of progress has been made since the discovery of the first HNF algorithm. The simplest way to compute HNF is a variant of the familiar Gaussian elimination algorithm (sometime called integer column reduction), where divisions are replaced by greatest common divisor computations. Unfortunately, it has been observed (and recently proved for a particular elimination strategy [4]) that although this algorithm performs a polynomial number of arithmetic operations, the complexity is exponential in general, because the

ISSN 1433-8092

size of the numbers involved in the intermediate computations can grow exponentially during the execution of the algorithm.

The first polynomial time algorithm for computing the Hermite Normal Form is due to Frumkin [5]. Since then, the time complexity of HNF computation has been widely investigated, finding better reduction strategies and incorporating ideas from fast matrix multiplication (see next section for a description of previous work). The currently fastest algorithm to compute HNF [8, 16], as well many other previous algorithms, are based on the idea of performing all arithmetic modulo the determinant of the matrix. Although this represents a tremendous improvement over the exponential time and space complexity of the simple Gaussian elimination algorithm, it should be noted that the determinant can be much bigger that the entries of the original matrix, so that even using modulo determinant arithmetic the space requirement for computing HNF can be quite big.

Consider for example an $n \times n$ matrix all of whose columns have norm at most M. It can be easily shown that both the input matrix and the corresponding Hermite Normal Form have $O(n^2 \log M)$ bit-size. Nevertheless, the determinant of the matrix can be as big as M^n and algorithms using modulo determinant arithmetic require $O(n^3 \log M)$ bits for the intermediate computations.

In this paper we show that it is possible to compute the HNF of an integer matrix using only $O(n^2 \log M)$ space also for the intermediate computations. When implemented using standard arithmetic and matrix multiplication, our algorithm has $O(n^5 \cdot \text{polylog}(n, M))$ time complexity, i.e., the same of the algorithms based on modular arithmetic. This time complexity translates to $O(mn^4 \cdot \text{polylog}(n, m, M))$ for nonsquare matrices. Moreover, fast matrix multiplication can be easily incorporated in our algorithm leading to time improvements analogous to those in [8, 16]. Only when implemented using asymptotically fast integer multiplication algorithms, previously known approaches are asymptotically faster than our new algorithm, but at the cost of a polynomially big space penalty. We also describe a simple heuristic based on the techniques developed in this paper that can be used to substantially improve the time complexity of our algorithm to $O(n^4 \cdot \text{polylog}(n))$, essentially matching the time bound of asymptotically fastest algorithms, but without the complications of using fast integer multiplication.

We present in Section 2 some of the previous algorithms and techniques used for computing HNF. Section 3 lists a series of facts we use in the analysis of our algorithm, which is presented in 4 for the particular case of square matrices. In Section 5 we show how the algorithm can be extended to nonsquare matrices and how to incorporate fast matrix multiplication algorithms in order to obtain better asymptotic running time.

2 Previous algorithms

Numerous researchers have previously given algorithms for computing the Hermite normal form of integer matrixes by using different techniques to cope with the coefficient explosion. The first polynomial time algorithm, due to Frumkin [5], uses an algorithm for solving linear Diophantine equations combined with modular reduction. In [12], Kannan and Bachem prove a polynomial bound on the size of the entries that arise during the execution of an algorithm that performs only basic column operations over the integers. Their procedure is successively improved by Chou and Collins [1], who prove better bounds for both the time and space used, and Iliopoulos [11] who extends this procedure with modular techniques.

For the case of square matrices, Domich, Kannan and Trotter [3], limit the size of the entries by using computation modulo suitably chosen numbers. Essentially, they describe an algorithm which in a first stage performs Gaussian elimination modulo the determinant, and later from the result that is obtained it recovers the Hermite normal form of the original matrix. A technique

useful to improve the space efficiency of all these algorithms is introduced [2] for the case when a factorization of the determinant of the input matrix is known.

Hafner and McCurley [8] extend the results of [3] to nonsquare matrix, but the main result of their paper is an algorithm for matrix triangularization based on fast matrix multiplication. However, it is not shown how to efficiently compute the Hermite normal form of such a matrix. Storjohann [16] gives a fast procedure which does this, and obtains the fastest running algorithm for HNF. The running time of this algorithm is $O(n^{(2+\theta)} \log^2 M)$, if a matrix multiplication algorithm that runs in $O(n^\theta)$ is used. Although the space efficiency is not explicitly analyzed, it follows from the triangularization procedure that is used that the space is of order $O(n^3 \log M)$. The same space requirement seems to hold for all previous algorithms and this can be a serious bottleneck for practical applications. Our algorithm improves on this limitation still maintaining a good running time.

3 Preliminaries

Let $\mathbf{A}$ be a $n \times n$ matrix with integer entries. We denote by $\mathbf{A}(i)$ the ith principal minor of $\mathbf{A}$, i.e. the $i \times i$ submatrix of $\mathbf{A}$ obtained by selecting its first i rows and columns. Let also $\mathbf{a}^T(i)$ be the $i+1$ row of $\mathbf{A}$ truncated to the first i elements, i.e., $\mathbf{a}^T(i) = (a_{i+1,1}, a_{i+1,2}, ..., a_{i+1,i})$. The lattice generated by $\mathbf{A}$ is the set $\mathcal{L}(\mathbf{A})$ of all integer linear combinations over the columns of $\mathbf{A}$.

Definition [Column equivalence] Let $\mathbf{A}, \mathbf{B} \in \mathbb{Z}^{n \times n}$. Then $\mathbf{A}$ and $\mathbf{B}$ are column equivalent iff $\mathcal{L}(\mathbf{A}) = \mathcal{L}(\mathbf{B})$.

Definition [Hermite Normal Form] We say that a matrix $\mathbf{B}$ is in Hermite normal form iff

- There exists $1 \leq i_1 < \ldots < i_n$ such that $b_{i,j} \neq 0 \Rightarrow i > i_j$ (strictly decreasing column height).
- For all $k < j, 0 \leq b_{i_j,k} < b_{i_j,j}$ (the top non-zero element of each column is the greatest element in the row).

Then, the Hermite normal form of square nonsingular matrices is a matrix in lower triangular form such that the offdiagonal entries are smaller than the diagonal entries of their respective row.

Proposition 1 *Let* $\mathbf{A}$ *be a matrix of size* n *and* M *an upperbound on the length of its columns. Then* $\det(\mathbf{A}) \leq M^n$.

Proposition 2 (Chinese Reminder Theorem) *Let* $m_1, ..., m_k$ *be pairwise coprime integers, i.e. such that* $\gcd(m_i, m_j) = 1$ *when* $i \neq j$. *Then, for any integers* x_i *there exists an integer* x *unique modulo* $\prod m_i$, *such that* $x \equiv x_i \pmod{m_i}$ *for* $1 \leq i \leq k$.

The following propositions combined with the Chinese Reminder Theorem gives a procedure to compute the determinant using modular arithmetic:

Proposition 3 *Let* $\mathbf{A}$ *be a square matrix and let* $p_1, ..., p_s$ *be a sequence of distinct primes such that their product exceeds twice the Hadamard bound for* $\det(\mathbf{A})$. *Suppose that* $\det(\mathbf{A}) \equiv a_i \pmod{p_i}, 1 \leq i \leq s$. *Then,* $\det(\mathbf{A})$ *is the integer* d *with smallest absolute value satisfying* $d \equiv a_i \pmod{p_i}$ *for all* i.

Proof: Let $\mathbf{b}$ be the Hadamard bound for E. The numbers $e_i, 1 \leq i \leq s$ determine the determinant of E modulo $p_1, p_2, ..., p_s$. Since $2b < p_1 p_2 ... p_s$, no two integers between $-b$ and b are congruent modulo $p_1, p_2, ..., p_s$.

Proposition 4 *Let* $\mathbf{A}$ *be a nonsingular matrix. Then there exists a permutation of the columns such* $\det(\mathbf{A}(i)) \neq 0$ *for all* $1 \leq i \leq n$.

Proof: A constructive proof that gives an efficient algorithm to find such a permutation is given in [12].

Proposition 5 *Let* $A \in \mathbb{Z}^{n \times n}$ *be a nonsingular, lower triangular matrix with integer entries, and let* $d = \det(\mathbf{A})$. *Then there exists a matrix* $\mathbf{B}$ *with integer entries such that* $\mathbf{AB} = d\mathbf{I}_n$.

Proposition 6 (Prime Number Theorem) $p_n = O(n \log n)$, *where* p_n *represents the* n*th prime number.*

4 Our Algorithm

In this section we present our algorithm for the particular case when the input matrix $\mathbf{A}$ is a square nonsingular matrix. We will show how to extend the algorithm to arbitrary matrices in a later section. We also assume without loss of generality (see Proposition 4) that all principal minors of $\mathbf{A}$ are nonsingular. We first give an informal description of the algorithm. Let n be the dimension of the input matrix, $\mathbf{B}$ be the principal minor of rank $n-1$, $\mathbf{a}^T$ the row vector given by the first $(n-1)$ elements of the last row of $\mathbf{A}$, and $\mathbf{b}$ the nth column of $\mathbf{A}$, i.e., we decompose the input matrix as follows:

$$\mathbf{A} = \left[\begin{array}{c|c} \mathbf{B} & \\ \mathbf{a}^T & \mathbf{b} \end{array}\right].$$

Our algorithm works in three major steps:

1. First, we (inductively) compute the Hermite normal form $\mathbf{H}_B$ of $\mathbf{B}$. (Notice that by assumption matrix $\mathbf{B}$ and all of it principal minors are non-singular.)

2. Then, we extend $\mathbf{H}_B$ to the Hermite normal form of $\mathbf{B'} = \left[\begin{array}{c} \mathbf{B} \\ \mathbf{a}^T \end{array}\right]$. Notice that the Hermite normal form of $\mathbf{B'}$ is necessarily of the form $\mathbf{H'} = \left[\begin{array}{c} \mathbf{H}_B \\ \mathbf{x}^T \end{array}\right]$. This step is implemented by a procedure AddRow $(\mathbf{B}, \mathbf{H}_B, \mathbf{a}^T)$ that on input a square matrix $\mathbf{B}$, its Hermite normal form $\mathbf{H}_B$, and a row vector $\mathbf{a}^T$, computes the row vector $\mathbf{x}^T$ such that $\mathbf{H'}$ is the Hermite normal form of $\mathbf{B'}$.

3. Finally, given $\mathbf{H'}$ and $\mathbf{b}$, we compute the Hermite normal form of $[\mathbf{H'}|\mathbf{b}]$ using a procedure AddColumn . Notice that since $\mathbf{H'}$ is the Hermite normal form of $\mathbf{B'}$, the two matrices $\mathbf{H'}$ and $\mathbf{B'}$ generate the same lattice. Therefore also $[\mathbf{H'}|\mathbf{b}]$ and $\mathbf{A} = [\mathbf{A'}|\mathbf{b}]$ generate the same lattice and the Hermite normal form of $[\mathbf{H'}|\mathbf{b}]$ is also the Hermite normal form of the original input matrix $\mathbf{A}$.

We now give a precise description of the algorithm, using the two procedures AddRow and AddColumn , and prove its correctness. Procedures AddRow and AddColumn will be described in the next two subsections.

Algorithm: (Hermite Normal Form)
Input: A nonsingular matrix $\mathbf{A} \in \mathbb{Z}^{n \times n}$ such that $d_i = \det(\mathbf{A}(i)) \neq 0$ for all $1 \leq i \leq n$.
Output: $\mathbf{H}$, The Hermite Normal Form of $\mathbf{A}$

(1) $\mathbf{x}^T(1) \leftarrow (a_{2,1}); \mathbf{H}(1) \leftarrow \mathbf{A}(1);$
(2) **for** $i \leftarrow 2$ **to** n
(3) $\quad \mathbf{x}^T(i) \leftarrow \text{AddRow } (\mathbf{A}(i), \mathbf{H}(i), \mathbf{a}^T(i))$
(4) $\quad \mathbf{H}(i+1) \leftarrow \text{AddColumn } \left(\begin{bmatrix} \mathbf{H}(i) \\ \mathbf{x}^T(i) \end{bmatrix}, \begin{pmatrix} a_{1,i+1} \\ \cdots \\ a_{i+1,i+1} \end{pmatrix} \right)$
(5) **return** $\mathbf{H}(n)$

Correctness of the algorithm We use an inductive argument to prove that $\mathbf{H}(i)$ is the Hermite normal form of $\mathbf{A}(i)$. This obviously holds for $i = 1$. For the inductive step assume that $\mathbf{H}(i)$ is the Hermite normal form of $\mathbf{A}(i)$, and let $\mathbf{b}$ be the vector $(a_{1,i+1}, ..., a_{i+1,i+1})^T$ at iteration i. Then, after step (4) is executed, $\mathbf{H}(i+1)$ is the Hermite normal form of $\left[\begin{array}{c|c} \mathbf{H}(i) & \\ \mathbf{x}^T(i) & b \end{array}\right]$. Since this matrix generates the same lattice as $A(i+1) = \left[\begin{array}{c|c} \mathbf{A}(i) & \\ \mathbf{a}^T(i) & \mathbf{b} \end{array}\right]$, (from the correctness of the AddRow procedure), it follows from the uniqueness of Hermite normal form that $\mathbf{H}(i+1)$ is the Hermite normal form of $\mathbf{A}(i+1)$.

The main result of our paper is stated in the following

Theorem 7 *There exists a linear space algorithm that on input* $\mathbf{A}$*, returns its Hermite normal form* $\mathbf{H}$.

Proof: It should be clear from the description of the algorithm that the space used is the maximum between the space used in the AddColumn and AddRow . The running time is n times the running time of these procedures. To complete the proof, we further analyze the complexity of the two procedures.

4.1 The AddRow procedure

The AddRow procedure takes as input a non-singular square matrix $\mathbf{A}$, its Hermite normal form $\mathbf{H}$ and a row vector $\mathbf{a}^T$. The output is a vector $\mathbf{x}^T$ such that the $H' = \begin{bmatrix} \mathbf{H} \\ \mathbf{x}^T \end{bmatrix}$ is the Hermite normal form of $A' = \begin{bmatrix} \mathbf{A} \\ \mathbf{a}^T \end{bmatrix}$. To compute $\mathbf{x}^T$, observe that vector $\mathbf{x}^T$ must satisfies $\mathbf{a}^T\mathbf{U} = \mathbf{x}^T$, where $\mathbf{U}$ is a unimodular transformation such that $\mathbf{AU} = \mathbf{B}$. It is important to notice that explicitly computing the transformation matrix U would be extremely space consuming. However, we notice that since $\mathbf{A}$ is non-singular, the transformation matrix $\mathbf{U} = \mathbf{A}^{-1}\mathbf{H}$ is uniquely defined and we can express the vector $\mathbf{x}^T = \mathbf{a}^T\mathbf{A}^{-1}\mathbf{H}$ directly in terms of $\mathbf{A}, \mathbf{H}$ and $\mathbf{a}^T$. Notice also that although $\mathbf{a}^T\mathbf{A}^{-1}$ is not generally an integer vector, the final result $\mathbf{x}^T = (\mathbf{a}^T\mathbf{A}^{-1})\mathbf{H}$ is always integral. So, we can compute $\mathbf{x}^T \pmod{p_i}$ modulo small primes p_i that do not divide $\det(\mathbf{A})$, and then combine the results using the Chinese Reminder Theorem. For each prime p_i this can be done as follows:

- First compute a solution $\mathbf{y}_i$ to the system of equations $\mathbf{A}^T\mathbf{y}_i = \mathbf{a} \pmod{p_i}$
- Then compute $\mathbf{x}_i = \mathbf{H}^\mathbf{T}\mathbf{y}_i \pmod{p_i}$.

Notice that the absolute value of the entries in $\mathbf{x}^T = (\mathbf{a}^T\mathbf{A}^{-1})\mathbf{H}$ is at most $V = n^2M^{2n+1}$, where M is an upperbound on the length of the columns of A, and n is its dimension. Then, if we compute $\mathbf{x}_i = \mathbf{x} \pmod{p_i}$ for $\log(2n^2M^{2n+1}) = O(n \log M)$ distinct primes, we can recover x using the Chinese Reminder Theorem. It follows that the space used to compute $\mathbf{x}^T$ is of order $O(n^2 \log M)$. For the time analysis notice that the dominant part of the procedure is solving the system of equations, and this can be done in $O(n^3 \log^2 p_i)$ by using standard Gaussian elimination procedure over $GF(p_i)$. Using the Prime Number Theorem(6), we get that the time complexity of AddRow is $O(n^3 \log V \log^2 \log V) = O(n^4 \cdot \text{polylog}(n, M))$.

4.2 The AddColumn Procedure

The AddColumn procedure takes as inputs a matrix $\mathbf{A} \in \mathbb{Z}^{n\times(n-1)}$ in Hermite normal form and a vector $\mathbf{b} \in \mathbb{Z}^n$. The output should be the Hermite normal form $\mathbf{H} \in \mathbb{Z}^{n\times n}$ of $[\mathbf{A}|\mathbf{b}]$. This is done as follows. First we extends $\mathbf{A}$ to a square matrix $\mathbf{H}_0 = [\mathbf{A}|\mathbf{c}]$ in Hermite normal form such that $[\mathbf{H}_0|\mathbf{b}]$ generates the same lattice as $[\mathbf{A}|\mathbf{b}]$. This is simply done setting $\mathbf{c} = (0, \ldots, d)^T$ where d is the determinant of matrix $[\mathbf{A}|\mathbf{b}]$. We then compute a sequence of matrix-vector pairs $\mathbf{H}_j, \mathbf{b}_j$ (for $j = 0, \ldots, n$), such that

- $\mathbf{b}_0 = \mathbf{b}$
- $\mathbf{H}_j$ is in Hermite normal form
- $\mathcal{L}([\mathbf{H}_j|\mathbf{b}_j]) = \mathcal{L}([\mathbf{H}_{j+1}|\mathbf{b}_{j+1}])$
- the first j elements of vector $\mathbf{b}$ are 0

It immediately follows by induction that $\mathbf{H} = \mathbf{H}_n$ is the Hermite normal form of $[\mathbf{A}|\mathbf{b}]$. Each pair $\mathbf{H}_{j+1}, \mathbf{b}_{j+1}$ is obtained from the previous one $\mathbf{H}_j, \mathbf{b}_j$ as follows. If the $(j+1)$th element of $\mathbf{b}_j$ is zero, then we simply set $\mathbf{H}_{j+1} = \mathbf{H}_j$ and $\mathbf{b}_{j+1} = \mathbf{b}_j$. Otherwise, we replace the $(j+1)$th column of $\mathbf{H}_j$ and $\mathbf{b}_j$ with two other columns obtained applying a unimodular transformation that clears the $(j+1)$th element of $\mathbf{b}_j$. This is done executing the extended Euclidean algorithm to the top two elements of the two columns. Once this is done, the remaining elements of the two columns might be bigger than the diagonal elements of $\mathbf{H}_j$. So, we reduce the two columns modulo the diagonal elements of $\mathbf{H}_j$ using the last $n - j$ columns of $\mathbf{H}_j$. During this modular reduction stage, entries are kept bounded performing the arithmetic modulo the determinants m_k of the trailing minors of $\mathbf{H}_j$. Matrix $\mathbf{H}_j$ and vector $\mathbf{b}_j$ correspond to the values of $\mathbf{H}$ and $\mathbf{b}$ at the jth iteration of the following algorithm. **Procedure** (AddColumn)
Input: A matrix $\mathbf{A} \in \mathbb{Z}^{n\times n-1}$ in Hermite normal form and a vector $\mathbf{b}$.
Output: The Hermite Normal Form $\mathbf{H}$ of the matrix $[\mathbf{A}|\mathbf{b}]$.

(0) Set $\mathbf{H}$ to the matrix $[\mathbf{A}|\mathbf{c}]$ where $\mathbf{c} = [0, \ldots, \det([\mathbf{A}|\mathbf{b}])]^T$
(1) $m_n \leftarrow h_{n,n}$;
(2) **for** $i \leftarrow n-1$ **downto** 1 **do** $m_i \leftarrow m_{i+1} \cdot h_{i,i}$;
(3) **for** $j \leftarrow 1$ **to** n **do**
(4) find k, l, g such that $kh_{j,j} + lb_j = g = \gcd(h_{j,j}, b_j)$;
(5) **for** $i \leftarrow j$ **to** n **do**

(6) $h_{i,j} \leftarrow kh_{i,j} + lb_j \pmod{m_i}$;
(7) $b_i \leftarrow b_i h_{j,j}/g - h_{i,j} b_j/g \pmod{m_i}$
(8) **for** $k \leftarrow j+1$ **to** n **do**
(9) $q \leftarrow h_{k,j}$ div $h_{k,k}$;
(10) **for** $l \leftarrow k$ **to** n **do**
(11) $h_{l,j} \leftarrow h_{l,j} - qh_{l,k} \pmod{m_l}$;

Given the matrix $\mathbf{A}$ and the column vector $\mathbf{b}$ the procedure eliminates the entries of $\mathbf{b}$ by performing column operations and reducing elements at row k modulo m_k. In order to prove that these operations do not change the lattice we have to show that they correspond to sequences of elementary column operations. Regarding the modular reduction operations, notice that m_k is the determinant of the submatrix corresponding to the non-zero rows of the last $n-k+1$ columns of $\mathbf{H}_j$ (for all $k > j$). So, the vector $(0, \ldots, 0, m_k, 0, \ldots, 0)^T$ belongs to the lattice generated by the last $n-k+1$ columns of $\mathbf{H}_j$. So reducing the kth entry of a vector modulo m_k correspond to subtracting appropriate multiples of the last $n-k+1$ columns of $\mathbf{H}_j$. Finally, notice that the column operations in step $(4, 5, 6, 7)$ correspond to the linear transformation

$$\begin{bmatrix} k & -b_j/g \\ l & h_{j,j}/g \end{bmatrix}$$

which has determinant equal to 1 by definition of k, l, g. So, this transformation is unimodular and corresponds to a sequence of elementary column operations.

This proves that the lattice generated by $[\mathbf{H}_n|\mathbf{b}_n]$ at the end of the algorithm is the same as the original lattice $[\mathbf{A}|\mathbf{b}]$. Moreover, $\mathbf{H}_n$ is in Hermite normal form and $\mathbf{b}_n = 0$, so the $\mathbf{H} = \mathbf{H}_n$ is the Hermite normal form of $[\mathbf{A}|\mathbf{b}]$.

To analyze the space complexity of AddColumn , assume that the size of the input matrix $\mathbf{A}$, and consequently the size of $\mathbf{H}$, is of order $O(n^2 \log M)$. It is easy to see that this assumption holds during the execution of our algorithm. During one iteration of the **for** in line (3) we only modify $\mathbf{H}_j$, the jth column of $\mathbf{H}$ and the vector $\mathbf{b}$. All computations on the elements of these vectors are done modulo m_i, so, the total space needed to store $\mathbf{b}$ and $\mathbf{H}_j$ is of order $\log m_1 + \ldots + \log m_n = \log \prod m_i = \log(\det(\mathbf{H}_n)) = n^2 \log M$. Because of the triangular reduction of lines (8)-(11), the matrix $[\mathbf{H}|\mathbf{b}]$ needs $O(n^2 \log M)$ storage space at the beginning at the next iteration. All computations are done inplace, so the total space needed by AddColumn is of order $O(n^2 \log M)$.

The main computational part of the the procedure consists of lines (8)-(11). This is essentially the "Triangular Reduction" procedure of [15], where the running time is proved to be $O(n^3 \log^2 M)$. Since the execution of the **for** loop in line (6) takes $O(n^2 \log^2 M)$, the total execution time of AddColumn is $O(n^4 \log^2 M)$.

5 Discussion

We presented an algorithm to compute the Hermite normal form of a square non-integer matrix with $O(n^2 \log M)$ space complexity and $O(n^5 \log^2 M)$ running time, where n is the dimension of the matrix and M is a bound on the length of the column vectors. Notice that the bit-size of the input is $O(n^2 \log M)$, so our algorithm has linear space complexity. In this section we first show how our algorithm can be easily adapted to work on any (possibly non-square) matrix. Then we also describe various ways to speed up our algorithm. In particular we describe both a theoretical improvement that can lower the running time to $O(n^{4.376} \log^2 M)$, and a simple heuristics that results in a running time approximately of the order of $O(n^4 \log^2 M)$, when input is chosen uniformly at random. . We

remark that while the first improvement gives a provably better worst case asymptotic upper bound on the running time, it is only of theoretical interest because of the big hidden constants. On the other hand, the second improvement, although only heuristics, can be quite effective in practice.

Our algorithm can be easily extended to non-square matrices using the AddRow and AddColumn procedures described in section 4. In particular, if $\mathbf{A} \in \mathbb{Z}^{n \times m}$ is a non-square full rank matrix, one can first run the algorithm on the square matrix consisting of the first n linearly independent columns of $\mathbf{A}$. (These columns can be easily found using a strightforward generalization of Proposition 3.) The matrix that is obtained is of the form $[\mathbf{H}|\mathbf{A}']$ where $\mathbf{H}$ is in Hermite normal form, and it is column equivalent to $\mathbf{A}$. Next, run the AddColumn procedure to add the columns of $\mathbf{A}'$ to $\mathbf{H}$. It is easy to see that the space complexity of the algorithm does not change, while the running time becomes $O(mn^4 \log^2 M)$. For the general case of not necessarily full rank matrices, one can first find a maximal set of linearly independent rows, then find the Hermite normal form of the corresponding full rank matrix, and finally extend it to the Hermite normal form of the input matrix using the AddColumn procedure. Notice that the entries of these last rows of the Hermite normal form can be quite big, and in order to keep the space complexity of the algorithm low, these rows should be computed one at a time and immediately output.

We now consider two ways to speed up our algorithm. A better running time can be obtained by incorporating fast matrix multiplication algorithm in the AddRow procedure. Since matrix multiplication and matrix inversion are problems of the same complexity, solving the linear system in the procedure can be done in $O(n^\theta \log^2 M)$, where $O(n^\theta)$ is the running time of a matrix multiplication algorithm. Thus, we obtain an algorithm with complexity $O(n^{2+\theta} \log^2 M)$. In practice one can obtain even better running time by using the following heuristic algorithm:

1. decompose $\mathbf{A}$ as $\left[\begin{array}{c|c|c} \mathbf{B} & \mathbf{c} & \mathbf{d} \\ \mathbf{b}^T & a_{n,n-1} & a_{n,n} \end{array}\right]$, with $\mathbf{B} \in \mathbb{Z}^{(n-2)\times(n-1)}, \mathbf{c}, \mathbf{d} \in \mathbb{Z}^{n-1}, \mathbf{b}^T \in \mathbb{Z}^{n-2}$;
2. compute $d_1 = \det([\mathbf{B}|\mathbf{c}])$ and $d_2 = \det([\mathbf{B}|\mathbf{d}])$ and find k, l such that $kd_1 + ld_2 = \gcd(d_1, d_2)$;
3. compute the Hermite normal form of $\left[\begin{array}{c|c} \mathbf{B} & k\mathbf{c} + l\mathbf{d} \\ \mathbf{b}^T & ka_{n,n-1} + la_{n,n} \end{array}\right]$;
4. run AddColumn twice to add the columns $\left[\begin{array}{c} \mathbf{c} \\ a_{n,n-1} \end{array}\right]$ and $\left[\begin{array}{c} \mathbf{d} \\ a_{n,n} \end{array}\right]$.

Notice that the determinant of the matrix at step (3) is $d = \gcd(d_1, d_2)$ which is tipically very small for randomly chosen matrices. So, the Hermite normal form at step (3) can be computed directly using for example the modular HNF algorithm from [3] with $O(n^3 \log^2 d)$ running time. The dominant part of the running time will become a single execution of the AddRow procedure resulting in a (heuristic) $O(n^4 \log^2 M)$ bound on the running time.

References

[1] T. W. J. Chou and G. E. Collins. Algorithms for the solution of systems of linear Diophantine equations. *SIAM Journal on Computing*, 11(4):687–708, 1982.

[2] P. D. Domich. Residual Hermite normal form computations. *ACM Trans. Math. Software*, 15(3):275–286, 1989.

[3] P. D. Domick, R. Kannan, and L. T. Jr. Hermite normal form computation using modulo determinant arithmetic. *Mathematics of Operations Research*, 12(1):50–59, Feb. 1987.

[4] X. G. Fang and G. Havas. On the worst-case complexity of integer gaussian elimination. In *Proceedings of the 1997 22nd International Symposium on Symbolic and Algebraic Computation, ISSAC*, pages 28–31, Maui, HI, USA, 1997. ACM.

[5] M. A. Frumkin. Polynominal time algorithms in the theory of linear diophantine equations. In M. Karpiński, editor, *Proceedings of the 1977 International Conference on Fundamentals of Computation Theory*, volume 56 of *LNCS*, pages 386–392, Poznań-Kórnik, Poland, Sept. 1977. Springer.

[6] M. A. Frumkin. Complexity question in number theory. *J. Soviet Math., 29*, 29:1502–1517, 1985.

[7] J. L. Hafner and K. S. McCurley. A rigurous subexponential algorithm for computation of class groups. *J. Amer. Math. Soc.*, 2:837–850, 1989.

[8] J. L. Hafner and K. S. McCurley. Asymptotically fast triangularization of matrices over rings. *SIAM Journal on Computing*, 20(6):1068–1083, 1991.

[9] C. Hermite. Sur l'introduction des variables continues dans la théorie des nombres. *J. Reine Angew. Math.*, 41:191–216, 1851.

[10] M. S. Hung and W. O. Rom. An application of the Hermite normal form in integer programming. *Linear Algebra and its Applications*, 140:163–179, 1990.

[11] C. S. Iliopoulos. Worst-case complexity bounds on algorithms for computing the canonical structure of finite abelian groups and the Hermite and Smith normal forms of an integer matrix. *SIAM Journal on Computing*, 18(4):658–669, 1989.

[12] R. Kannan and A. Bachem. Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix. *SIAM Journal on Computing*, 8(4):499–507, Nov. 1979.

[13] D. Micciancio. Lattice based cryptography: A global improvement. (Theory of Cryuptography Library Report 99-05). Available at http://philby.ucsd.edu/cryptolib.

[14] J. Ramanujam. Beyond unimodular transformation. *The Journal of Supercomputing*, 9(4):365–389, 1995.

[15] A. Storjohann. Computing Hermite and Smith normal forms of triangular integer matrices. *Linear Algebra and its Applications*, 282(1-3):25–45, 1998.

[16] A. Storjohann and G. Labahn. Asymptotically fast computation of Hermite normal forms of integer matrices. In *ISSAC'96*, pages 259–266, Zurich, Switzerland, 1996. ACM.

ECCC ISSN 1433-8092
http://www.eccc.uni-trier.de/eccc
ftp://ftp.eccc.uni-trier.de/pub/eccc
ftpmail@ftp.eccc.uni-trier.de, subject 'help eccc'