A Linear Space Algorithm for Computing the Hermite Normal Form*

Daniele Micciancio
University of California, San Diego
9500 Gilman Drive, Mail Code 0114
La Jolla, California 92093
daniele@cs.ucsd.edu

Bogdan Warinschi
University of California, San Diego
9500 Gilman Drive, Mail Code 0114
La Jolla, California 92093
bogdan@cs.ucsd.edu

ABSTRACT

Computing the Hermite Normal Form of an $n \times n$ integer matrix using the best current algorithms typically requires $O(n^3 \log M)$ space, where M is a bound on the entries 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 algorithms for integer and matrix multiplication, our algorithm has the same time complexity of the asymptotically fastest (but space inefficient) algorithms. We also present a heuristic algorithm for HNF that achieves a substantial speedup when run on randomly generated input matrices.

1. INTRODUCTION

The *Hermite Normal Form* (HNF) is a standard form for integer matrices that is useful in many applications. For example, the HNF is used in finding the solution of systems of linear Diophantine equations [9], algorithmic problems in lattices [11], integer programming [14] and loop optimization techniques [19]. Recently, one more application of the HNF has been suggested [18]: the use of HNF to improve the security and efficiency of lattice based cryptosystems. Lattice problems have attracted considerable interest in the design of public key cryptosystems because of the surprising average-case/worst-case connection discovered by Ajtai in 1996 [1]. Despite their theoretical appeal, the applicability of lattice based cryptosystems has been greatly reduced by the large size of their public keys. For example, the cryptosystems proposed in [10, 7] require public keys of several megabytes in size in order to be practically secure. In [18] it is demonstrated that the HNF can be used to substantially reduce the public key size of lattice based cryptosystems while preserving the security level. For example the public keys suggested in [10] can be reduced from several megabytes to few hundred kilobytes. Unfortunately, the best current HNF algorithms have super-linear space complexity, so although the HNF can be used to easily reduce the size of the public keys, the key generation process (and the HNF computation in particular) can be extremely space consuming, requiring hundreds of megabytes for typical values of the security parameter.

In order to make key generation feasible on typical personal computers, a better, space efficient algorithm to compute HNF is required. A standard approach to save space in number theoretic computations, is to compute the result modulo many small prime numbers, and combine the results using the Chinese Reminder Theorem. Unfortunately, this trick does not work for HNF computations: if p is a prime that does not divide the determinant of the matrix A, then one can always transform A into a diagonal matrix performing elementary column operations modulo p, so trying to compute the HNF of A modulo p does not give any useful information.

Interestingly, the related problem of computing the HNF of a polynomial matrix (i.e. a matrix whose entries are polynomials with coefficients in a field), can be solved in NC and therefore in polylogarithmic space [16, 23]. However, these techniques do not seem to apply to integer matrices and the problem of finding an NC algorithm for integer HNF is still open.

The main contribution of this paper is a new practical algorithm to compute the Hermite Normal Form of integer matrices. The main advantage of the new algorithm is space efficiency: when applied to an $n \times n$ matrix with entries bounded by M, the space used for intermediate computation is provably $O(n^2 \log M)$ (i.e., essentially the same as the input size). When used on matrices that arise from cryptographic problems, this results in a potential saving factor of $n = 400$ or more for typical values of the security parameter, reducing the memory requirements from several hundred megabytes, to just a megabyte, or little more.

Our algorithm is also practical. The current fastest algorithm ([20]), combines the triangularization procedure of [12] with a very fast algorithm to compute the Hermite Nor-

*Research supported in part by NSF Career Award CCR-0093029.

ISSAC 2001, UWO, Canada
©2001 ACM 1-58113-218-2/ 00/ 0008 $5.00

mal Form of triangular matrices. The dominant part of the running time of this algorithm is given by the triangularization procedure, which is $O(n^{\theta} B(n \log M))$, where n^{θ} is the number of arithmetic operations required to multiply two $n \times n$ matrices and $B(t)$ is an upper bound to the number of bit operations required to multiply two t-bit numbers. For comparison, the worst case running time of our algorithm is $O(n^5 \text{polylog}(n, M))$, i.e., essentially the same as the asymptotically fastest algorithm when implemented using standard integer and matrix multiplication.

Furthermore, we present a variant of our algorithm that achieves substantially smaller *heuristic* running time when executed on random input matrices, still keeping the space complexity linear. When certain conditions on the input matrix hold, we achieve a running time of $O(n^4 \operatorname{polylog}(M, n))$ (or even $O(n^3 \operatorname{polylog}(M, n))$, see discussion at the end of section 5), outperforming all currently known algorithms. We present a brief experimental analysis on the likelihood that these conditions are satisfied when the input matrix is chosen at random.

The structure of the paper is as follows. In Section 2 we give a synopsis of the previous algorithms and techniques used for computing HNF. Section 3 lists a series of facts used in the analysis of our algorithm. We present the algorithm in Section 4 and the heuristic version in Section 5. Finally, we show how to extend the algorithm to cope with non square matrices and make some final remarks in Section 6.

2. PREVIOUS ALGORITHMS

The simplest way to compute HNF is a variant of the well known Gaussian elimination algorithm (aka. integer column reduction), where divisions are replaced by greatest common divisor computation. Unfortunately, it has been observed (and recently proved for a particular elimination strategy [6]) that although this algorithm performs a polynomial number of arithmetic operations, its complexity is exponential, since the size of the numbers involved in the intermediate computations can grow exponentially during the execution of the algorithm. Numerous researchers have previously given algorithms for computing the Hermite normal form of integer matrices by using different techniques to cope with the coefficient explosion. The first polynomial time algorithm, due to Frumkin [8], uses an algorithm for solving linear Diophantine equations combined with modular reduction. In [17], 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 [2], who prove better bounds for both the time and space used, and Iliopoulos [15] who extends this procedure with modular techniques.

For the case of square matrices, Domich, Kannan and Trotter [5], 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 then recovers the Hermite normal form of the original matrix. A technique useful to improve the time and space efficiency of all these algorithms is introduced [4]. However, the technique only applies for the case when a factorization of the determinant of the input matrix is known.

Hafner and McCurley [12] extend the results of [5] to nonsquare matrices and also show how to use fast matrix multiplication in computing a triangular form of the input matrix. However, the result produced by this algorithm is not necessarily in Hermite normal form since off diagonal elements may be bigger than the corresponding diagonal element. Storjohann and Labahn [21] show how to obtain the HNF from of such a matrix using fast matrix multiplication. A even faster procedure to compute the HNF of a triangular matrix is given also by Storjohann in [20]. The running time of the HNF algorithm obtained this way is dominated by the Hafner-McCurley triangularization procedure and equals $O(n^{(2+\theta)} \log^2 M)$. Although the space efficiency is not explicitly analyzed, it follows from the triangularization procedure that the space is $O(n^3 \log M)$. The same space requirement seems to hold for all previous algorithms and this can be a serious bottleneck when operating on big matrices.

3. PRELIMINARIES

In this section we give a few definitions and facts that are needed to present and analyze our algorithm. First of all we define the *Hermite Normal Form* (HNF). Informally, a square nonsingular matrix 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 entry on the same row. The definition for the general case follows:

Definition 1. A matrix $\mathbf{H} \in \mathbb{Z}^{m \times n}$ is in Hermite Normal Form if

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

Definition 2. Given a matrix $\mathbf{A} \in \mathbb{Z}^{m \times n}$, with integer entries, the lattice generated by $\mathbf{A}$ is the set $\mathcal{L}(\mathbf{A})$ of all integer linear combinations of the columns of $\mathbf{A}$. Two matrices $\mathbf{A}, \mathbf{B} \in \mathbb{Z}^{m \times n}$ are column equivalent iff $\mathcal{L}(\mathbf{A}) = \mathcal{L}(\mathbf{B})$.

A classical result of Hermite [13] states that for every integer matrix $\mathbf{A}$, there exists a unique matrix $\mathbf{H}$ that is in Hermite Normal Form and is column equivalent to $\mathbf{A}$.

The following number theoretic results are used in developing and analyzing the algorithm.

PROPOSITION 1 (HADAMARD BOUND). *Let* $\mathbf{A}$ *be a matrix in* $\mathbb{Z}^{n \times n}$ *and let* L *be an upper bound on the Euclidean length of its columns. Then* $\det \mathbf{A} \leq L^n$.

Another useful fact we use is that the determinant of an integer square matrix can be computed efficiently by calculating its residue modulo sufficiently many primes, and then recover the final result using the Chinese Remaindering Theorem (see Sections 5.4, 5.5 of [24].)

The following fact is proved in [17]:

PROPOSITION 2. *There exists an efficient algorithm that on input a nonsingular matrix, outputs a permutation of its columns such that all principal minors are nonsingular.*

PROPOSITION 3. *If p_n denotes the nth prime number, then $p_n = O(n \log n)$.*

4. A SPACE EFFICIENT ALGORITHM

In this section we present our algorithm for the particular case when the input is a square nonsingular matrix. In a later section we show how to extend the algorithm to arbitrary matrices.

The idea is the following. Decompose the input matrix $\mathbf{A}$ into

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

Then,

1. recursively compute the HNF $\mathbf{H}_B$ of $\mathbf{B}$;
2. extend $\mathbf{H}_B$ to the HNF $\mathbf{H}'$ of $\mathbf{B}' = \begin{bmatrix} \mathbf{B} \\ \mathbf{a}^T \end{bmatrix}$;
3. finally, compute and return the HNF of $[\mathbf{H}'|\mathbf{b}]$.

In order to execute steps (2) and (3), we need the two procedures below:

- AddRow($\mathbf{B}, \mathbf{H}_B, \mathbf{a}^T$): on input a square nonsingular matrix $\mathbf{B}$, its HNF $\mathbf{H}_B$, and a row vector $\mathbf{a}^T$, returns the (unique) row vector $\mathbf{x}^T$ such that $\mathbf{H}' = \begin{bmatrix} \mathbf{H}_B \\ \mathbf{x}^T \end{bmatrix}$ is the HNF of $\mathbf{B}' = \begin{bmatrix} \mathbf{B} \\ \mathbf{a}^T \end{bmatrix}$
- AddColumn($\mathbf{H}', \mathbf{b}$): on input a matrix $\mathbf{H}'$ in HNF, and column vector $\mathbf{b}$, returns the HNF of matrix $[\mathbf{H}'|\mathbf{b}]$.

We describe these two procedures in the next two subsection, but first let's see how AddRow and AddColumn can be used to devise an efficient HNF algorithm. We use the following notation. Let $\mathbf{A} \in \mathbb{Z}^{n \times n}$ be the input to the algorithm and let $\mathbf{A}(i)$ be the ith principal minor of $\mathbf{A}$. Without loss of generality we may assume that for all i, $\det(\mathbf{A}(i)) \neq 0$. (See Proposition 2.) Let $\mathbf{a}^T(i) = (a_{i+1,1}, a_{i+1,2}, ..., a_{i+1,i})$ be the ith dimensional vector obtained truncating the $(i+1)$th row of $\mathbf{A}$ to its first i elements. Our HNF algorithm iteratively computes a sequence of matrices $\{\mathbf{H}(i)\}$ and vectors $\{\mathbf{x}(i)\}$ as follows:

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

We now prove by induction that for all i, matrix $\mathbf{H}(i)$ is the HNF of $\mathbf{A}(i)$. Since $\mathbf{A} = \mathbf{A}(n)$, this implies that the algorithm returns the Hermite normal form of $\mathbf{A}$.

The base case is obviously true. For the inductive step assume that $\mathbf{H}(i-1)$ is the HNF of $\mathbf{A}(i-1)$. Then, step (3) computes a vector $\mathbf{x}(i-1)$ such that

$$\begin{bmatrix} \mathbf{H}(i-1) \\ \mathbf{x}^T(i-1) \end{bmatrix}$$

is the HNF of

$$\begin{bmatrix} \mathbf{A}(i-1) \\ \mathbf{a}^T(i-1) \end{bmatrix}$$

and in particular they generate the same lattice. Let $\mathbf{b}$ be the vector $(a_{1,i}, ..., a_{i,i})^T$. Step (4) sets $\mathbf{H}(i)$ to the HNF of

$$\left[\begin{array}{c|c} \mathbf{H}(i-1) & \mathbf{b} \\ \mathbf{x}^T(i-1) & \end{array} \right].$$

Since this last matrix is column equivalent to

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

it follows that $\mathbf{H}(i)$ is the Hermite Normal Form of $\mathbf{A}(i)$.

4.1 The AddRow procedure

The AddRow procedure takes as input a non-singular square matrix $\mathbf{B}$, its HNF $\mathbf{H}_B$ and a row vector $\mathbf{a}^T$. The output is a vector $\mathbf{x}^T$ such that the $\mathbf{H}' = \begin{bmatrix} \mathbf{H}_B \\ \mathbf{x}^T \end{bmatrix}$ is the HNF of $\mathbf{A} = \begin{bmatrix} \mathbf{B} \\ \mathbf{a}^T \end{bmatrix}$.

The following considerations lead to a time/space efficient procedure to compute $\mathbf{x}^T$: if $\mathbf{U}$ is the (unique) unimodular transformation such that $\mathbf{H}_B = \mathbf{BU}$ then $\mathbf{x}^T$ is simply $\mathbf{a}^T\mathbf{U} = \mathbf{a}^T\mathbf{B}^{-1}\mathbf{H}_B$. However we cannot compute $\mathbf{U} = \mathbf{B}^{-1}\mathbf{H}_B$ explicitly because the entries of $\mathbf{U}$ can be as big as the determinant of $\mathbf{A}$, too big to be stored all at the same time. Instead we try to compute $\mathbf{x}$ directly as follows. For a suitably chosen sequence of primes $p_1, p_2, \ldots$

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

Observe that even if $\mathbf{a}^T\mathbf{B}^{-1}$ is not generally an integral vector, the final result $\mathbf{x}^T$ is guaranteed to be integral, and therefore $\mathbf{x_i}$ is congruent to $\mathbf{x}$ modulo p_i. If we compute $\mathbf{x}_i$ for sufficiently many primes p_i, then $\mathbf{x}$ can be recovered using the Chinese Remainder Theorem.

In order to bound the number of primes necessary to correctly recover $\mathbf{x}$, we need to bound the entries of $\mathbf{x}^T$. This also gives bounds to both the time and the space complexity of AddRow. Let M be a bound to the absolute value of the elements of $\mathbf{B}$, and let $h_1, .., h_n$ be the diagonal elements of $\mathbf{B}$ (in particular $\prod_i h_i = D$, and we can safely assume that $\sum_{i=1}^{n} h_i \leq D$). Since the entries of $\mathbf{B}^{-1}$ are bounded by $D = \det(\mathbf{B})$ an element of $\mathbf{B}^{-1}\mathbf{H}_B$ is $O(\sum_{j=1}^{n} Dh_i) \leq O(D^2)$. Therefore, an entry of $\mathbf{x}$ has as upper bound $V = O(nMD^2)$=$O(nM^{2n+1})$ (for simplicity we assumed $D = O(M^n)$, a more accurate bound is the Hadamard bound). The bit size of $\mathbf{x}$ is thus $O(n^2 \log M)$.

A rough estimate of the number of primes needed to recover $\mathbf{x}$, is $O(\log V)$, so from Proposition 3, the largest of this primes is $\log V \log\log V$. Each of the systems of equations modulo p_i can be solved in $O(n^3 \log^2 p_i)$ by Gaussian elimination. So AddRow is $O(\log V)O(n^3 \log^2(\log V \log\log V))$ which after expanding V becomes $O(n^4 \text{polylog}(n, M))$. Faster methods to solve systems of linear equations are described in [3, 22] based on p-adic expansion techniques. It is plausible that the same techniques can be applied to the implementation of our AddRow procedure, reducing the running time from $O(n^4 \text{polylog}(n, M))$ to $O(n^3 \text{polylog}(n, M))$. We will analyze these alternative techniques in the final version of this paper.

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 is the Hermite normal form $\mathbf{H} \in \mathbb{Z}^{n \times n}$ of $[\mathbf{A}|\mathbf{b}]$. The procedure works as follows. First we extend $\mathbf{A}$ 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.

```
(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)          $h_{i,j} \leftarrow kh_{i,j} + lb_i \pmod{m_i}$;
(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 sub-matrix 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 $\mathbf{H}_j$ and 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 $\mathbf{b}_n = 0$, therefore $\mathbf{H} = \mathbf{H}_n$ 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 $O(n^2 \log M)$. 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 the jth column of $\mathbf{H}$ and the vector $\mathbf{b}$, and we keep the entries bounded by performing computations modulo m_i. The space occupied by these two vectors is $O(\sum_{i=1}^{n} \log m_i) = O(n \log m_1)$. Since $m_1 = \det(H)$, this space is $O(n^2 \log M)$. 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 of the next iteration. All computations are done in place, so the total space needed by AddColumn is $O(n^2 \log M)$.

The main computational part of the the procedure consists of lines (8)-(11). This is essentially the "Triangular Reduction" procedure of [20] with running time $O(n \log^2 D)$, where D is the determinant of the matrix $\mathbf{A}$. In our case, D is of order $O(M^n)$, and since the execution of the **for** loop in line (6) takes $O(n^2 \log^2 M)$, the total execution time of AddColumn is $n(O(n \log^2 M^n) + O(n^2 \log^2 M)) = O(n^4 \log^2 M)$.

5. A FAST HEURISTIC ALGORITHM

In this section, we use the techniques presented in section 4 to give a heuristic algorithm which, in practice, achieves significantly better running time than the one described in the previous section. In fact, we show that the new algorithm is likely to reduce the running time by a factor n or even n^2, outperforming all previously known algorithms. We emphasize that our analysis, although heuristic, has been tested on matrix distributions that arise in practice (for example in the key generation process of the lattice based cryptosystem of [18]).

The idea is to use computation modulo the determinant, but do so for some matrix whose determinant is very small. The algorithm is given below:

1. decompose $\mathbf{A}$ into
$$\left[\begin{array}{c|c|c} \mathbf{B} & \mathbf{c} & \mathbf{d} \\ \mathbf{b}^T & a_{n,n-1} & a_{n,n} \end{array}\right]$$
where $\mathbf{B} \in \mathbb{Z}^{(n-2)\times(n-1)}$, $\mathbf{c}, \mathbf{d} \in \mathbb{Z}^{n-1}$, and $\mathbf{b} \in \mathbb{Z}^{n-2}$;
2. compute the determinants $d_1 = \det([\mathbf{B}|\mathbf{c}])$ and $d_2 = \det([\mathbf{B}|\mathbf{d}])$
3. execute the extended Euclidean algorithm to find integers k, l such that $d = kd_1 + ld_2 = \gcd(d_1, d_2)$;
4. compute the HNF $\mathbf{H}$ of matrix $[\mathbf{B}|k\mathbf{c} + l\mathbf{d}]$
5. compute the HNF $\mathbf{H}'$ of matrix
$$\left[\begin{array}{c|c} \mathbf{B} & k\mathbf{c} + l\mathbf{d} \\ \mathbf{b}^T & ka_{n,n-1} + la_{n,n} \end{array}\right],$$
running AddRow on input $[\mathbf{B}|k\mathbf{c} + l\mathbf{d}]$, $\mathbf{H}$, and $[\mathbf{b}^T \mid ka_{n,n-1} + la_{n,n}]$);
6. run AddColumn twice to add columns
$$\left[\begin{array}{c} \mathbf{c} \\ a_{n,n-1} \end{array}\right] \quad \text{and} \quad \left[\begin{array}{c} \mathbf{d} \\ a_{n,n} \end{array}\right]$$
back to $\mathbf{H}$.

Notice that step (4) can be execute in time $O(n^3 \log^2 d)$ (e.g., using the modulo determinant HNF algorithm from [5]). So, a substantial reduction of the running time is obtained whenever the quantity $d = \gcd(d_1, d_2)$ is small.

When d is small, the running time of the algorithm is dominated by a single execution of the AddRow and two executions of AddColumn. So, the running time is $O(n^4 \log^2 M)$.

To estimate the behavior of this algorithm we performed a few numerical experiments. The following procedure was iterated 500 times, and collecting the results:

- generate a random integral matrix $\mathbf{A}$
- generate two random integer vectors $\mathbf{X}$ and $\mathbf{Y}$
- compute $d_1 = \det[\mathbf{A}|\mathbf{X}]$ and $d_2 = \det[\mathbf{A}|\mathbf{Y}]$;
- compute $d = \gcd(d_1, d_2)$.

In our experiments, we used matrices with size between 30 and 300 and entries bounded by 100000. It turned out that even when the matrix is big the quantity d is typically very small: in 30% of the cases d was 1 and in 80% of the cases d was less than 10. Moreover, the largest value of d that was observed was around 4000. Thus, it seems reasonable to assume that when the input matrix is chosen at random, a typical number involved in the computations may be represented using a single computer word, and consequently, operations using these numbers take constant time.

The following estimate shows that the running time of the algorithm can be even much less than $O(n^4 \operatorname{polylog}(M, n))$. Usually, the HNF matrix $\mathbf{H}$ computed at step (4) has small determinant. Consequently, its diagonal elements are small, and most of them must be 1. A similar fact holds for $\mathbf{H}'$: the elements of the last row may be as big as D, the determinant of the matrix, but the principal minor of size $n-1$ is “almost” the identity matrix. It is not hard to see that on this kind of input AddColumn is much faster than the $O(n^4 \operatorname{polylog}(M, n))$ worst case running time: the triangular reduction (lines(8)-(11)), will consist of $O(n)$ additions of small numbers (corresponding to the reduction of the first $n-2$ elements of the column being reduced), plus $O(n)$ additions of numbers of size $O(\log D)$ (these are the operations involving the bottom elements of the column that is being reduced.) This procedure is repeated for each column of $\mathbf{H}$, so the overall running time of AddColumn is $n(O(n) + O(n^2 \operatorname{polylog} M)) = O(n^3 \operatorname{polylog} M)$. If we could reduce the running time of AddRow to $O(n^3 \operatorname{polylog} M)$ as well, (e.g., using methods from [3, 22], see remarks at the end of subsection 4.1) then the heuristic running time of the new HNF algorithm would be just $O(n^3 \operatorname{polylog} M)$, outperforming all previously known HNF algorithm also in terms of running time. We believe that such extension is indeed possible, but we still have to analyze most details.

6. DISCUSSION

We presented an algorithm to compute the Hermite Normal Form of a square non-singular 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 its entries. Notice that the bit-size of the input is $O(n^2 \log M)$, so our algorithm has linear space complexity.

The algorithm was given for square nonsingular matrices. We now show how AddRow and AddColumn of Section 4 can be used to give an algorithm that works with arbitrary matrices: if $\mathbf{A} \in \mathbb{Z}^{n\times m}$ is a full rank non-square matrix, one can first run the algorithm on the square matrix consisting of the first n linearly independent columns of $\mathbf{A}$. The matrix that is obtained is of the form $[\mathbf{H}|\mathbf{A}']$. It has the properties that it is column equivalent to $\mathbf{A}$, and that $\mathbf{H}$ is in Hermite normal form. Next, run AddColumn to add the columns of $\mathbf{A}'$. The space complexity does not change, and the running time becomes $O(mn^4 \log^2 M)$. For the general case when the input is not necessarily a full rank matrix, one can first find a maximal set of linearly independent rows, and then find the Hermite normal form of the corresponding full rank matrix. Finally, extend this 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.

In Section 5 we presented an alternative algorithm with much smaller heuristic running time. In particular, the algorithm is extremely fast when run on randomly generated input matrices, as those used in cryptographic application (see [18]). Of course, there are cases when the heuristics fails. One such example is when in the original matrix, elements of one row have a big common factor. Although the heuristic could be fixed to take into account this case (factor out the common factor, execute the algorithm, recover the result) there exist other cases for which this simple fix does not work. We leave as an open problem to find a linear space HNF algorithm with *worst case* $O(n^3 \operatorname{polylog}(M, n))$ running time.

7. ACKNOWLEDGMENTS

We would like to thank the referees for their careful reading and very helpful comments.

8. REFERENCES

[1] Ajtai, M. Generating hard instances of lattice problems (extended abstract). In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing* (Philadelphia, Pennsylvania, 22–24 May 1996), pp. 99–108.

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

[3] Dixon, J. Exact solution of linear equations using p-adic expansions. *Numerical Mathematics 40* (1982), 137–141.

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

[5] Domich, P. D., R.Kannan, and L.E.Trotter. Hermite normal form computation using modulo determinant arithmetic. *Mathematics of Operations Research 12*, 1 (Feb. 1987), 50–59.

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

[7] Fischlin, R., and Seifert, J.-P. Tensor-based trapdoors for CVP and their application to public key cryptography. In *7th IMA International Conference "Cryptography and Coding"* (1999), vol. 1746 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 244–257.

[8] Frumkin, M. A. Polynominal time algorithms in the theory of linear diophantine equations. In *Proceedings of the 1977 International Conference on Fundamentals of Computation Theory* (Poznań-Kórnik, Poland, Sept. 1977), M. Karpiński, Ed., vol. 56 of *LNCS*, Springer, pp. 386–392.

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

[10] Goldreich, O., Goldwasser, S., and Halevi, S. Public-key cryptosystems from lattice reduction problems. In *Advances in Cryptology—CRYPTO '97* (17–21 Aug. 1997), B. S. Kaliski Jr., Ed., vol. 1294 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 112–131.

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

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

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

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

[15] Iliopoulos, C. S. 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 (1989), 658–669.

[16] Kaltofen, E., Krishnamoorthy, M., and Saunders, B. Fast parallel computation of Hermite and Smith forms of polynomial matrices. *SIAM J. ALg. Disc. Math* (1987), 683–690.

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

[18] Micciancio, D. Improving lattice based cryptosystems using the Hermite normal form. In *Cryptography and Lattices Conference 2001* (2001), LNCS, Springer-Verlag.

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

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

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

[22] Storjohann, A., and Mulders, T. Diophantine linear system solving. In *ISSAC'99* (Zurich, Switzerland, 1999), ACM, pp. 181–188.

[23] Villard, G. Computing Popov and Hermite forms of polynomial matrices. In *Proceedings of the 1996 international symposium on Symbolic and algebraic computation* (1996), pp. 250–258.

[24] von zur Gathen, J., and Gerhard, J. *Modern Computer Algebra*. Cambridge University Press, Cambridge, 1999.