Fast Computation of Hermite Normal Forms of Random Integer Matrices

Clément Pernet[1]

William Stein[2]

Abstract

This paper is about how to compute the Hermite normal form of a *random* integer matrix in practice. We propose significant improvements to the algorithm by Micciancio and Warinschi, and extend these techniques to the computation of the saturation of a matrix. Tables of timings confirm the efficiency of this approach. To our knowledge, our implementation is the fastest implementation for computing Hermite normal form for large matrices with large entries.

Key words: Hermite normal form, exact linear algebra

1. Introduction

This paper is about how to compute the Hermite normal form of a *random* integer matrix in practice. We describe the best known algorithm for random matrices, due to Micciancio and Warinschi [MW01] and explain some new ideas that make it practical. We also apply these techniques to give a new algorithm for computing the saturation of a module, and present timings.

In this paper we do not concern ourselves with nonrandom matrices, and instead refer the reader to [SL96, Sto98] for the state of the art for worse case complexity results. Our motivation for focusing on the random case is that it comes up frequently in algorithms for computing with modular forms.

Among the numerous notions of Hermite normal form, we use the following one, which is the closest to the familiar notion of reduced row echelon

[1]Supported by the National Science Foundation under Grant No. 0713225.
[2]Supported by the National Science Foundation under Grant No. 0653968.

form.

Definition 1.1 (Hermite Normal Form)**.** For any $n \times m$ integer matrix A the *Hermite normal form* (HNF) of A is the unique matrix $H = (h_{i,j})$ such that there is a unimodular $n \times n$ matrix U with $UA = H$, and such that H satisfies the following two conditions:

- there exist a sequence of integers $j_1 < \cdots < j_n$ such that for all $0 \leq i \leq n$ we have $h_{i,j} = 0$ for all $j < j_i$ (row echelon structure)
- for $0 \leq k < i \leq n$ we have $0 \leq h_{k,j_i} < h_{i,j_i}$ (the pivot element is the greatest along its column and the coefficient above are nonnegative).

Thus the Hermite normal form is a generalization over $\mathbb{Z}$ of the reduced row echelon form of a matrix over $\mathbb{Q}$. Just as computation of echelon forms is a building block for many algorithms for computing with vector spaces, Hermite normal form is a building block for algorithms for computing with modules over $\mathbb{Z}$ (see, e.g., [Coh93, Ch. 2]).

Example 1.2. The HNF of the matrix

$$A = \begin{pmatrix} -5 & 8 & -3 & -9 & 5 & 5 \\ -2 & 8 & -2 & -2 & 8 & 5 \\ 7 & -5 & -8 & 4 & 3 & -4 \\ 1 & -1 & 6 & 0 & 8 & -3 \end{pmatrix}$$

is

$$H = \begin{pmatrix} 1 & 0 & 3 & 237 & -299 & 90 \\ 0 & 1 & 1 & 103 & -130 & 40 \\ 0 & 0 & 4 & 352 & -450 & 135 \\ 0 & 0 & 0 & 486 & -627 & 188 \end{pmatrix}.$$

Notice how the entries in the answer are quite large compared to the input.

Heuristic observations: For a random $n \times m$ matrix A with $n \leq m$, the number of digits of each entry of the rightmost $m - n + 1$ columns of H are similar in size to the determinant of the left $n \times n$ submatrix of A. For example, a random 250×250 matrix with entries in $[-2^{32}, 2^{32}]$ has HNF with entries in the last column all having about 2590 digits and determinant with about 2590 digits, but all other entries are likely to be very small (e.g., a single digit).

There are numerous algorithms for the computing HNF's, including [KB79, DKLET87, Bra89, MW01]. We describe an algorithm that is based on the heuristically fast algorithm by Micciancio and Warinschi [MW01], updated with several practical improvements. Our implementation is currently asymptotically the fastest available (see Section 9).

In the rest of this paper, we mainly address computation of the HNF of a square nonsingular matrix A. We also briefly explain how to reduce the general case to the square case, discuss computation of saturation, and give timings. We give an outline of the algorithm in Section 2 and present more details in Sections 3, 5 and 6. The cases of more rows than columns and more columns than rows is discussed in the Section 7. In Section 9, we sketch the main features of our implementation in Sage, and compare the computation time for various class of matrices.

Acknowledgement: We thank Allan Steel for providing us with notes from a talk he gave on his implementation of [MW01]. Steel's implementation was much faster than any other system available (e.g., Pari, NTL, Mathematica, Maple, and GAP), which was the motivation for this paper. The extent to which our algorithm is similar to Steel's is unclear, because Steel's algorithm has not been published and Magma is closed source. We would also like to thank Burcin Erocal for implementing mod n computation of Hermite form in Sage, Robert Bradshaw for help benchmarking our implementation, Arne Storjohann for helpful conversations about the non-random case, and Andrew Crites and Michael Goff for their final student project on computing HNF's.

2. Outline of the algorithm when A is square

For the rest of this section, let $A = (a_{i,j})_{i,j=0,\ldots,n-1}$ be an $n \times n$ matrix with integer entries. There are two key ideas behind the algorithm of [MW01] for computing the HNF of A.

1. Every entry in the HNF H of a square matrix A is at most the absolute value of the determinant $\det(A)$, so one can compute H be working modulo the determinant of H. This idea was first introduced and developed in [DKLET87].
2. The determinant of A may of course still be extremely large. Micciancio and Warinschi's clever idea is to instead compute the Hermite form H' of a small-determinant matrix constructed from A using the Euclidean

algorithm and properties of determinants. Then we recover H from H' via three update steps.

We now explain the second key idea in more detail. Consider the following block decomposition of A:

$$A = \begin{bmatrix} B & b \\ c^T & a_{n-1,n} \\ d^T & a_{n,n} \end{bmatrix},$$

where B is the upper left $(n-1) \times (n-1)$ submatrix of A, and b, c, d are column vectors. Let $d_1 = \det\left(\begin{bmatrix} B \\ c^T \end{bmatrix}\right)$ and $d_2 = \det\left(\begin{bmatrix} B \\ d^T \end{bmatrix}\right)$. Use the extended Euclidean algorithm to find integers s, t such that

$$g = sd_1 + td_2,$$

where $g = \gcd(d_1, d_2)$.

Since the determinant is linear in row operations, we have

$$\det\left(\begin{bmatrix} B \\ sc^T + td^T \end{bmatrix}\right) = g \tag{2.1}$$

For random matrices, g is likely to be very small. Figure 2.1 illustrates the distribution of such gcd's, on a set of 500 random integer matrices of dimension 100 with 100-bit coefficients.

Algorithm 1 (on page 5) is essentially the algorithm of Micciancio and Warinschi. Our main improvement over their work is to greatly optimize Steps 3, 4 and 8. Step 8 is performed by a procedure they call `AddColumn` (see Algorithm 3 in Section 5 below), and steps 9 and 10 by a procedure they call `AddRow` (see Algorithm 4 in Section 6 below).

3. Double determinant computation

There are many algorithms for computing the determinant of an integer matrix A. One algorithm involves computing the Hadamard bound on $\det(A)$, then computing the determinant modulo p for sufficiently many p using an (asymptotically fast) Gaussian elimination algorithm, and finally using a Chinese remainder theorem reconstruction. This algorithm has bit complexity

$$\mathcal{O}\left(n^4(\log n + \log \|A\|) + n^3 \log^2 \|A\|\right),$$

Algorithm 1: Hermite Normal Form [MW01]

Data: A: an $n \times n$ nonsingular matrix over $\mathbb{Z}$

Result: H: the Hermite normal form of A

begin

Write $A = \begin{bmatrix} B & b \\ c^T & a_{n-1,n} \\ d^T & a_{n,n} \end{bmatrix}$

Compute $d_1 = \det\left(\begin{bmatrix} B \\ c^T \end{bmatrix}\right)$

Compute $d_2 = \det\left(\begin{bmatrix} B \\ d^T \end{bmatrix}\right)$

Compute the extended gcd of d_1 and d_2: $g = sd_1 + td_2$

Let $C = \begin{bmatrix} B \\ sc^T + td^T \end{bmatrix}$

Compute H_1, the Hermite normal form of C, by working modulo g as explained in Section 4 below. (NOTE: In the unlikely case that $g = 0$ or g is large, we compute H_1 using any HNF algorithm applied to C, e.g., by recursively applying the main algorithm of this paper to C.)

Obtain from H_1 the Hermite form H_2 of $\begin{bmatrix} B & b \\ sc^T + td^T & sa_{n-1,n} + ta_{n,n} \end{bmatrix}$

Obtain from H_2 the hermite form H_3 of $\begin{bmatrix} B & b \\ c^T & a_{n-1,n} \end{bmatrix}$

Obtain from H_3 the Hermite form H of $\begin{bmatrix} B & b \\ c^T & a_{n-1,n} \\ d^T & a_{n,n} \end{bmatrix}$

end

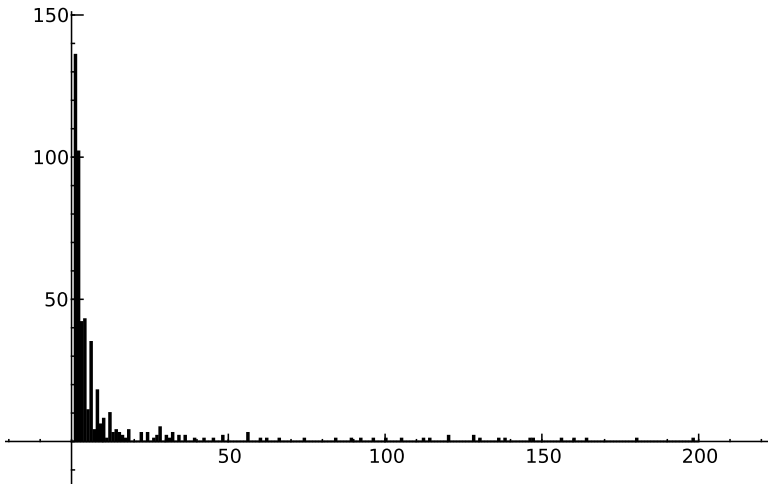

Figure 2.1: Distribution of the determinants in (2.1), for 500 random matrices with $n = 100$ and entries uniformly chosen to satisfy $\log_2 \|A\| = 100$. Only 9 elements had a determinant larger than 200, and the largest one was 6816.

or $\mathcal{O}\left(n^{\omega+1}(\log n + \log \|A\|)\right)$ with fast matrix arithmetic (see [GG99, Ch 5]).

Abbott, Bronstein and Mulders [ABM99] propose another determinant algorithm based on solving $Ax = v$ for a random integer vector v using an iterative p-adic solving algorithm (e.g., [Dix82, MC79]). In particular, by Cramer's rule the greatest common divisor of the denominators of the entries of x is a divisor d of $D = \det(A)$. The unknown integer D/d can be recovered by computing it modulo p for several primes and using the Chinese remainder theorem; usually D/d is very small, so this is fast. This approach has a similar worst case bit complexity: $\mathcal{O}\left(n^4 + n^3(\log n + \log \|A\|)^2\right)$ but a better average case complexity of $\mathcal{O}\left(n^3(\log^2 n + \log \|A\|)^2\right)$.

The computation time can also be improved by allowing early termination in the Chinese remainder algorithm: once a reconstruction stabilizes modulo several primes, the result is likely to remain the same with a certified probability, and one can avoid the remaining modular computations.

Further details on practical implementations for computing determinants of integer matrices can be found in [DU06].

Storjohann [Sto05] obtains the best known bit complexity for computing determinants using a Las Vegas algorithm. He obtains a complexity of $\mathcal{O}^{\sim}\left(n^{\omega} \log \|A\|\right)$, where ω is the exponent for matrix multiplication. However,

no implementation of this algorithm is known that is better in practice than the p-adic lifting based method for practical problem sizes. Consequently, we based our implementation on this latter algorithm by [ABM99].

The computation of the two determinants (Steps 3 and 4) therefore involves the solving of two systems, with very similar matrices. We reduce it to only one system solution *in the generic case* using the following lemma. Since this is a bottleneck in the algorithm, this factor of two savings is huge in practice.

Lemma 3.1. *Let A be an $n \times (n-1)$ matrix and c and d column vectors of degree n, and assume that the augmented matrices $[A|c]$ and $[A|d]$ are both invertible. Let $x = (x_i)$ be the solution of $[A|c]x = d$. If $x_n \neq 0$, then the solution $y = (y_i)$ to $[A|d]y = c$ is*

$$y = \left(-\frac{x_1}{x_n}, -\frac{x_2}{x_n}, \ldots, -\frac{x_{n-1}}{x_n}, \frac{1}{x_n} \right).$$

Proof. Write a_i for the ith column of A. The equation $[A|c]x = d$ is thus $\left(\sum_{i=1}^{n-1} a_i x_i\right) + cx_n = d$, so $\left(\sum_{i=1}^{n-1} a_i x_i\right) - d = -x_n c$. Dividing both sides by $-x_n$ yields $\left(\sum_{i=1}^{n-1} \left(-\frac{x_i}{x_n}\right) a_i\right) + \frac{1}{x_n} d = c$, which proves the lemma. □

Example 3.2. Let $A = \begin{bmatrix} 1 & 2 \\ -4 & 3 \\ 2 & -5 \end{bmatrix}$, $c = (-1, 3, 5)^T$, and $d = (2, -3, 4)^T$. The solution to $[A|c]x = d$ is

$$x = \left(\frac{111}{68}, \frac{35}{68}, \frac{45}{68} \right).$$

Thus

$$y = \left(-\frac{x_1}{x_3}, -\frac{x_2}{x_3}, \frac{1}{x_3} \right) = \left(-\frac{37}{15}, -\frac{7}{9}, \frac{68}{45} \right).$$

Algorithm 2 (on page 8) describes how the two determinants are computed using Lemma 3.1.

4. Hermite form modulo g

Recall that C is a square nonsingular matrix with "small" determinant g. Step 7 of Algorithm 1 (on page 5) is to compute the HNF of C as explained in

Algorithm 2: Double determinant computation

Data: B: an $(n-1) \times n$ matrix over $\mathbb{Z}$
Data: c, d: two vectors in $\mathbb{Z}^n$.
Result: $(d_1, d_2) = \left(\det\left(\begin{bmatrix} B^T & c \end{bmatrix}\right), \det\left(\begin{bmatrix} B^T & d \end{bmatrix}\right)\right)$
begin
  Solve the system $\begin{bmatrix} B^T & c \end{bmatrix} x = d$ using Dixon's p-adic lifting.
  Then $y_i = -x_i/x_n$, $y_n = 1/x_n$ solves $\begin{bmatrix} B^T & d \end{bmatrix} y = c$ by Lemma 3.1,
  unless $x_n = 0$, in which case we use the usual determinant
  algorithm to compute the determinants of the two matrices.
  $u_1 = \text{lcm}(\text{denominators}(x))$
  $u_2 = \text{lcm}(\text{denominators}(y))$
  Compute Hadamard's bounds h_1 and h_2 on the determinants of
  $\begin{bmatrix} B^T & c \end{bmatrix}$ and $\begin{bmatrix} B^T & d \end{bmatrix}$
  Select a set of primes (p_i) s.t. $\prod_i p_i > \max(h_1/u_1, h_2/u_2)$
  foreach p_i **do**
    compute $B^T = LUP$, the LUP decomposition of $B^T \mod p_i$
    $q = \prod_{i=1}^{n-1} U_{i,i} \mod p_i$
    $x = L^{-1}c \mod p_i$
    $y = L^{-1}d \mod p_i$
    $v_1^{(i)} = qx_n \mod p_i$
    $v_2^{(i)} = qy_n \mod p_i$
  reconstruct v_1 and v_2 from $(v_1^{(i)})$ and $(v_2^{(i)})$ using CRT
  return $(d_1, d_2) = (u_1 v_1, u_2 v_2)$
end

[DKLET87, §3]. There it is proved that since $g = \det(C)$, the Hermite normal form of $\begin{bmatrix} C \\ gI \end{bmatrix}$ is $\begin{bmatrix} H \\ 0 \end{bmatrix}$ where H is the Hermite normal form of C. Using this result, to compute H, we apply the standard row reduction Hermite normal form algorithm to C, always reducing all numbers modulo g. Conceptually, think of this as adding multiples of the rows of gI, which does not change the resulting Hermite form. At the end of this process we obtain a matrix $H = (h_{ij})$ with $0 \leq h_{ij} < g$ for all ij. There is one special case; since the product of the diagonal entries of the Hermite form of C is g, if the lower right entry of H is 0, then we replace it by g. Then the resulting matrix H is the Hermite normal form of C.

For additional discussion of the modular Hermite form algorithm, see [Coh93, §2.4, pg. 71] which describes the algorithm in detail, including a discussion of our above remark about replacing 0 by g.

Example 4.1. Let $C = \begin{bmatrix} 5 & 26 \\ 2 & 11 \end{bmatrix}$. Then $g = \det(C) = 3$, and the reduction mod g of C is $\begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$. Subtracting the second row from the first yields $\begin{bmatrix} 2 & 2 \\ 0 & 0 \end{bmatrix}$, which is already reduced modulo 3. Then multiplying through the first row by -1 and reducing modulo 3 again, we obtain $\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$. Then, as mentioned above, since the lower right entry is 0, we replace it by $g = 3$, obtaining the Hermite normal form $H = \begin{bmatrix} 1 & 1 \\ 0 & 3 \end{bmatrix}$.

5. Add a column

Step 8 of Algorithm 1 is to find a column vector e such that

$$\begin{bmatrix} H_1 & e \end{bmatrix} = U \begin{bmatrix} B & b \\ sc^T + td^T & a_{n-1,n} \end{bmatrix} \tag{5.1}$$

is in Hermite form, for a unimodular matrix U.

By Hypothesis $C = \begin{bmatrix} B \\ sc^T + td^T \end{bmatrix}$ is invertible, so from (5.1), one gets

$$\begin{aligned} e &= U \begin{bmatrix} b \\ a_{n-1,n-1} \end{bmatrix} \\ &= H_1 \begin{bmatrix} B \\ sc^T + td^T \end{bmatrix}^{-1} \begin{bmatrix} b \\ a_{n-1,n-1} \end{bmatrix} \end{aligned}$$

In [MW01], the column e is computed using multi-modular computations and a tight bound on the size of the entries of e. We instead use the p-adic lifting algorithm of [Dix82, MC79] to solve the system

$$\begin{bmatrix} B \\ sc^T + td^T \end{bmatrix} x = \begin{bmatrix} b \\ a_{n-1,n-1} \end{bmatrix}$$

However, the last row $sc^T + td^T$ typically has much larger coefficients than the rest of the matrix, thus unduly penalizing the complexity of finding a solution. Our key idea is to replace the row $sc^T + td^T$ by a random row u that has small entries such that the resulting matrix is still invertible, find the solution y of this modified system, then recover x as follows. Let $\{k\}$ be a basis of the 1-dimensional kernel of B. Then the sought for solution of the original system is

$$x = y + \alpha k,$$

where α satisfies

$$(sc^T + td^T) \cdot (y + \alpha k) = a_{n-1,n-1}.$$

By linearity of the dot product, we have

$$\alpha = \frac{a_{n-1,n-1} - (sc^T + td^T) \cdot y}{(sc^T + td^T) \cdot k}$$

Note that if $(sc^T + td^T) \cdot k = 0$, then $Ck = 0$, which would contradict our assumption that $C = \begin{bmatrix} B \\ sc^T + td^T \end{bmatrix}$ is invertible.

6. Add a row

Steps 9 and 10 of Algorithm 1 consist of adding a new row to the current Hermite form and updating it to obtain a new matrix in Hermite form.

The principle is to eliminate the new row with all existing pivots and update the already computed parts when necessary. Algorithm 4 (on page 12) describes this in more detail.

Algorithm 3: AddColumn

Data: $B = \begin{bmatrix} B_1 & b_2 \\ b_3^T & b_4 \end{bmatrix}$: a $n \times n$ matrix over $\mathbb{Z}$, where B_1 is $(n-1) \times (n-1)$ and b_2, b_3 are vectors.

Data: H_1: the Hermite normal form of $\begin{bmatrix} B_1 \\ b_3^T \end{bmatrix}$

Result: H: the Hermite normal form of B

begin

Pick a random vector u such that $|u_i| \leq \|B\|\ \forall i$

Solve $\begin{bmatrix} B_1 \\ u \end{bmatrix} y = \begin{bmatrix} b_2 \\ b_4 \end{bmatrix}$

Compute a kernel basis vector k of B_1

$\alpha = b_4 - \frac{b_3^T \cdot y}{b_3^T \cdot k}$

$x = y + \alpha k$

$e = H_1 x$

return $\begin{bmatrix} H_1 & e \end{bmatrix}$

end

7. The Nonsquare Case

In the case where the matrix is rectangular, with dimensions $m \times n$, we reduce to the case of a square nonsingular matrix as follows: first compute the column and row rank profile (pivot columns and subset of independent rows) of A modulo a random word-size prime. With high probability, the matrix A has the same column and row rank profile over $\mathbb{Q}$, so we can now apply Algorithm 1 to the square nonsingular $r \times r$ matrix obtained by picking the row and column rank profile submatrix of A over $\mathbb{Z}$.

The additional rows and columns are then incorporated as follows:

additional columns: use Algorithm 3 (`AddColumn`) with a block of column vectors instead of just one column. If this fails, then we computed the rank profile incorrectly, in which case we start over with a different random prime.

additional rows: use Algorithm 4 (`AddRow`) for each additional row.

Algorithm 4: AddRow

Data: A: an $m \times n$ matrix in Hermite normal form
Data: b: a vector of degree n
Result: H: the Hermite normal form of $\begin{bmatrix} A \\ b \end{bmatrix}$

begin
  forall *pivots* a_{i,j_i} *of* A **do**
    if $b_{j_i} = 0$ **then**
      continue
    if $A_{i,j_i} | b_{j_i}$ **then**
      $b := b - b_{j_i}/A_{i,j_i} A_{i,1\ldots n}$
    else
      `/* Extended gcd based elimination */`
      $(g, s, t) = \mathrm{XGCD}(a_{i,j_i}, b_{j_i})$; `/* so` $g = sa_{i,j_i} + tb_{j_i}$ `*/`
      $A_{i,1\ldots n} := sA_{i,1\ldots n} + tb_{j_i}$
      $b := b_{j_i}/gA_{i,1\ldots n} - A_{i,j_i}/gb$
      for $k = 1$ *to* $i - 1$ **do**
        `/* Reduces row` k `with row` i `*/`
        $A_{k,1\ldots n} := A_{k,1\ldots n} - \lfloor A_{k,j_i}/A_{i,j_i} \rfloor A_{i,1\ldots n}$
  if $b \neq 0$ **then**
    let j be the index of the first nonzero element of b
    insert b^T between rows i and $i+1$ such that $j_i < j < j_{i+1}$
  Return $H = \begin{bmatrix} A \\ b \end{bmatrix}$
end

8. Saturation

If M is a submodule of $\mathbb{Z}^n$ for some n, then the saturation of M is $\mathbb{Z}^n \cap (\mathbb{Q}M)$, i.e., the intersection with $\mathbb{Z}^n$ of the $\mathbb{Q}$-span of any basis of M. For example, if M has rank n, then the saturation of M just equals $\mathbb{Z}^n$. Also, kernels of homomorphisms of free $\mathbb{Z}$-modules are saturated. Saturation comes up in many number theoretic algorithms, e.g., saturation is an important step in computing a basis over $\mathbb{Z}$ for the space of q-expansions of cuspidal modular forms of given weight and level, and comes up in explicit computation with homology of modular curves using modular symbols.

There is a well-known connection between saturation and Hermite form. If A is a basis matrix for M, and H is the Hermite form of the transpose of A with any 0 rows at the bottom deleted (so H is square), then $H^{-1}A$ is a matrix whose rows are a basis for the saturation of M. Thus computation of a saturation of a matrix reduces to computation of one Hermite form and solving a system $HX = A$.

If A is sufficiently random, then the Hermite form matrix H has a very large last column and all other entries are small, so we exploit the trick in Section 6 and instead solve a much easier system. We give some timings below in Table 9.3 of our implementation of this algorithm in Sage.

9. Sage Implementation and Timings

We implemented the algorithms described in this paper as part of Sage [Ste]. Note that our implementation relies on `IML` [SC] for the solution of integer systems using p-adic lifting, and on `LinBox` [Lin] for the computation of determinants modulo p (the `IML` and `LinBox` libraries are both part of Sage). Our Sage implementation is currently primarily optimized for the square case, and all our timings below only involve Hermite forms of square matrices.

We illustrate computing a Hermite normal form and saturation in Sage.

```
sage: A = matrix(ZZ,3,5,[-1,2,5,65,2,4,-1,-3,1,-2,-1,-2,1,-1,1])
sage: A
[-1  2  5 65  2]
[ 4 -1 -3  1 -2]
[-1 -2  1 -1  1]
sage: A.hermite_form()
[  1   0  17 259   7]
```

```
[  0   1  31 453  13]
[  0   0  40 582  17]
sage: A.saturation()
[-1  2  5 65  2]
[ 4 -1 -3  1 -2]
[-1 -2  1 -1  1]
```

There are open source implementations of Hermite normal form algorithms available in NTL [Sho], Pari [PAR], and GAP [GAP]. The timings in Table 9.1 illustrate that the algorithm in this paper is asymptotically better than these standard implementations. For example, reducing a 500x500 random matrix with 32-bit entries takes less than a minute in Sage (see Table 9.2), but over an hour in NTL, Pari, and GAP.

Table 9.1: Time in seconds for NTL, Pari, GAP, and Sage to compute the HNF of a random $n \times n$ matrix whose entries are uniformly distributed between -2^b and 2^b, where each column is labeled "library (b)".

n	ntl (8)	ntl (32)	pari (8)	pari (32)	gap (8)	gap (32)	sage (32)
50	0.09	0.31	0.09	0.26	0.12	0.19	0.24
250	74.58	494.46	163.69	776.91	36.08	165.49	7.3
500	1975	12199	2925	15263	712	3982	47

Tables 9.2-9.3 give timings using Sage-3.2.3. All timings in Tables 9.2–9.3 were done using a single processor on a Sun Fire X4450 server equipped with Intel 2.66Ghz X7460 Xeon processors[3]. For comparison, we also give timings for Magma [BCP97] V2.14-9, which is the only other software we know of that implements the algorithm of [MW01]. We give some timings using Pari [PAR] 2.3.3, NTL 5.4.2, and GAP 4.4.10 (these are the versions included in Sage-3.2.3). We do not include timings for either Maple or Mathematica, since they are both much slower at computing Hermite forms than any other system mentioned above, partly because the only function for computing Hermite forms in Maple and Mathematica also computes the transformation matrix.

[3]Purchased using National Science Foundation Grant No. DMS-0821725

Table 9.2: Time in seconds for Sage and Magma to compute the **HNF** of a random $n \times n$ matrix whose entries are uniformly distributed in the interval $[-2^b, 2^b]$, where b =bits. For $b < 512$, we time five runs and give the range of values obtained.

Sage

n	8 bits	32 bits	128 bits	256 bits	512 bits
50	0.1–0.1	0.2–0.2	0.7–0.8	2.7–2.9	12.45
250	4.1–5.4	6.6–7.1	33.6–37.0	102.8–110.7	470.22
500	24.8–26.6	38.8–43.5	179.4–187.2	534.7–566.1	2169.41
1000	164.9–191.1	266.3–282.8	1081.4–1143.0	2804.9–3149.9	10506
2000	1500.4	2837.5			
4000	11537.3	17105.9			

Magma

n	8 bits	32 bits	128 bits	256 bits	512 bits
50	0.0–0.0	0.0–0.1	0.3–0.3	0.9–1.0	2.73
250	0.9–1.1	11.5–14.5	60.1–87.6	196.4–249.4	764.6
500	6.8–8.6	183.5–226.1	977.5–1004.5	2611.9–2649.4	7100.91
1000	70.8–74.6	1580.4–2017.3	7876.0–8582.9	21370.2	58339
2000	886.1	14917.8			
4000	6096.5				

Table 9.3: Time in seconds for Sage and Magma to compute the **saturation** of a random $n \times m$ matrix whose entries are uniformly distributed in the interval $[-2^b, 2^b]$, where b =bits. When a range is given, we time ten runs and give the range of timings obtained.

Sage

$n \times m$	8 bits	32 bits	128 bits
100×101	0.2–3.2	0.6–3.2	1.8–9.2
100×150	0.3–0.5	0.4–4.7	1.8–9.7
100×300	0.3–4.8	0.4–10.5	1.9–14.8
200×201	1.3–12.0	2.2–12.8	10.3–61.1
200×300	1.3–3.4	2.3–18.2	11.0–75.5
200×600	1.5–44.8	2.3–38.8	20.6–83.6
250×251	2.3–26.1	3.6–33.6	18.9–99.6
250×375	2.3–4.9	3.7–58.1	18.6–102.8
250×750	2.5–79.3	3.9–68.8	47.8–55.1
300×301	3.7–24.0	5.8–41.6	29.8–159.2
300×450	3.8–42.5	6.3–59.4	26.7–69.3
300×900	4.1–109.4	6.2–172.2	25.7–198.5
500×501	14.6–92.1	22.5–145.7	91.4–460.2
500×750	15.4–246.7	22.3–195.8	94.4–237.6
500×1500	15.8–30.8	21.3–441.2	95.9–873.3

Magma

$n \times m$	8 bits	32 bits	128 bits
100×101	0.2–0.2	1.2–1.3	6.3–9.6
100×150	0.3–0.3	1.9–2.3	9.6–11.9
100×300	0.7–0.8	3.9–5.0	22.0–26.8
200×201	2.0–2.3	17.3–23.3	86.0–113.5
200×300	3.1–3.6	21.2–32.1	149.2–185.6
200×600	7.1–7.8	43.3–51.0	232.5–288.5
250×251	4.2–4.5	43.5–55.5	177.1–216.3
250×375	6.5–7.6	55.1–64.0	272.8–294.8
250×750	14.4–16.4	99.5–107.7	426.1
300×301	7.8–8.4	73.8–91.5	378.5–434.2
300×450	11.9–13.7	100.4–128.9	369.9–455.3
300×900	27.4–31.2	176.0–203.0	822.9–923.9
500×501	46.5–48.7	323.5–385.7	1576.8–1670.1
500×750	74.3–88.4	421.9–554.9	2427.1–2731.3
500×1500	164.7–187.4	855.2–935.8	4758.4–5593.7

References

[ABM99] Abbott, Bronstein, and Mulders, *Fast deterministic computation of determinants of dense matrices*, International Symposium on Symbolic and Algebraic Computation, ACM press, 1999.

[BCP97] W. Bosma, J. Cannon, and C. Playoust, *The Magma algebra system. I. The user language*, J. Symbolic Comput. **24** (1997), no. 3–4, 235–265, Computational algebra and number theory (London, 1993). MR 1 484 478

[Bra89] R. J. Bradford, *Hermite normal forms for integer matrices*, European Conference on Computer Algebra (EUROCAL '87) (Berlin - Heidelberg - New York), Springer, June 1989, pp. 315–316.

[Coh93] H. Cohen, *A course in computational algebraic number theory*, Springer-Verlag, Berlin, 1993. MR 94i:11105

[Dix82] John D. Dixon, *Exact solution of linear equations using p-adic expansions*, Numerische Mathematik **40** (1982), 137–141.

[DKLET87] P. D. Domich, R. Kannan, and Jr L. E. Trotter, *Hermite normal form computation using modulo determinant arithmetic*, Mathematics of Operations Research **12** (1987), no. 1, 50–59.

[DU06] Jean-Guillaume Dumas and Anna Urbanska, *An introspective algorithm for the integer determinant*, Proc. Transgressive Computing, Grenade Spain, april 2006, pp. 185–202.

[GAP] GAP, *Groups, algorithms, programming—a system for computational discrete algebra*, `http://www-gap.mcs.st-and.ac.uk/`.

[GG99] Joachim von zur Gathen and Jürgen Gerhard, *Modern computer algebra*, Cambridge University Press, New York, NY, USA, 1999.

[KB79] Ravindran Kannan and Achim Bachem, *Polynomial algorithms for computing the smith and hermite normal forms of an integer matrix*, SIAM J. Comput. **8** (1979), no. 4, 499–507.

[Lin] The LinBox Group, *LinBox: Exact linear algebra with dense and blackbox matrices, (Version 1.1.5)*, `http://www.linalg.org`.

[MC79] R. Moenck and J. Carter, *Approximate algorithms to derive exact solutions to systems of linear equations*, Proceedings of the International Symposium on Symbolic and Algebraic Manipulation (EUROSAM '79) (Marseille, France) (Edward W. Ng, ed.), LNCS, vol. 72, Springer, June 1979, pp. 65–73.

[MW01] Daniele Micciancio and Bogdan Warinschi, *A linear space algorithm for computing the Hermite Normal Form*, International Symposium on Symbolic and Algebraic Computation, ACM press, 2001, pp. 231–236.

[PAR] PARI, *A computer algebra system designed for fast computations in number theory*, `http://pari.math.u-bordeaux.fr/`.

[SC] Arne Storjohann and Zhuliang Chen, *Integer Matrix Library, (Version 1.0.2)*, `http://www.cs.uwaterloo.ca/~z4chen/iml.html`.

[Sho] V. Shoup, *NTL: Number theory library*, `www.shoup.net/ntl/`.

[SL96] Arne Storjohann and George Labahn, *Asymptotically fast computation of Hermite normal forms of integer matrices*, Proc. Int'l. Symp. on Symbolic and Algebraic Computation: ISSAC '96, ACM Press, 1996, pp. 259–266.

[Ste] William Stein, *Sage: Open Source Mathematical Software (Version 3.2.3)*, The Sage Group, `http://www.sagemath.org`.

[Sto98] Arne Storjohann, *Computing Hermite and Smith normal forms of triangular integer matrices*, Linear Algebra Appl **282** (1998), 25–45.

[Sto05] ———, *The shifted number system for fast linear algebra on integer matrices*, Journal of Complexity **21** (2005), no. 4, 609–650.