The modular extended gcd problem and space efficient algorithms for integer matrices

Arne Storjohann [1]

School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

Abstract

Given integers a, b and nonzero N, an output sensitive algorithm is given to compute the smallest c such that $\gcd(a+cb, N) = \gcd(a, b, N)$. The algorithm is extended to the case when b is a vector and then applied to get space efficient algorithms for triangularizing and diagonalizing integer matrices.

1 Introduction

Let $a_1 < a_2 < a_3 \cdots$ be the sequence of positive integers relatively prime to a nonzero N. Define the function $g(N) := \max_i(a_i - a_{i-1})$. The problem of bounding $g(N)$ in terms of the number $\omega(N)$ of distinct prime divisors of N was studied by Jacobsthal [1], Erdös [2] and Kanold [3]. Iwaniec [4] proves the asymptotic estimate $g(N) = O(\omega(N)^2(\log \omega(N))^2)$. Using $\omega(N) \leq \frac{\log N}{\log\log N} + 1.45743\frac{\log N}{(\log\log N)^2}$ [5] gives $g(N) = O((\log N)^2)$.

In §3 and §4 we study some computational problems related to g. The first is, given a and N with $0 \leq a < N$, find the smallest nonnegative c such that $a+c$ is relatively prime to N (Notation: $a + c \perp N$). Computing $\texttt{Gcd}(a+q, N)$ for $q = 0, 1, 2, \ldots, c$ in succession requires $O(\mathsf{B}(\log N)c)$ bit operations, where B is a cost function for certain operations with integers, in particular also gcd computation, see §2. Substituting $c \leq g(N) = O((\log N)^2)$ gives a worst case running time bound of $O(\mathsf{B}(\log N)(\log N)^2)$ bit operations. We give an

Email address: `astorjoh@uwaterloo.ca` (Arne Storjohann).
URL: `http://scg.uwaterloo.ca/personal/~astorjoh` (Arne Storjohann).

[1] Supported by the National Sciences and Engineering Research Council of Canada under Grant No. RGPIN 26082-03: Algorithms for the exact solution to problems in linear algebra.

algorithm that solves this problem with $O(\mathsf{B}(\log N)\log N)$ bit operations. The algorithm is designed to be sensitive to both $\omega(N)$ and the output c, both of which may be far from worst case (i.e., $\omega(N) \ll \log N/\log\log N$ and/or $c \ll g(N)$). We show that the running time of our algorithm is bounded by $O(c(\log c)^2 + \mathsf{B}(\log N)\min(c, \omega(N)\log\log N))$ bit operations.

In §4 we extend the algorithm for solving $a + c \perp N$ to solve a generalization: given $a, b_1, b_2, \ldots, b_k$ with $\gcd(a, b_1, b_2, \ldots, b_k, N) = 1$, the goal is to find a sequence of small nonnegative integers $c_1, c_2, \ldots, c_k$ such that $a + c_1 b_1 + c_2 b_2 + \cdots + c_k b_k \perp N$. We call this the modulo N extended gcd problem. Our algorithm has worst case complexity $O(\mathsf{B}(\log N)(n + \log N))$ bit operations and actually returns the unique sequence $c_1, c_2, \ldots, c_k$ of nonnegative integers that is lexicographically minimal among all solutions. This minimality condition allows us to obtain the bound $c_1 + c_2 + \ldots + c_k = O((\log N)^2)$ and the bound $\omega(N)$ for the number of nonzero c_i's. These bounds are useful for our application of the modulo N extended gcd algorithm to some integer matrix computations. We discuss this next.

The Hermite canonical form H of a full column rank matrix A over $\mathbb{Z}$ is upper triangular with off-diagonal entries in each column reduced modulo the positive diagonal entry in the same column. Computing the form is a classical problem with many applications; see the texts by Sims [6] or Cohen [7]. The form is also defined when A is rectangular with possibly non maximal rank, see §5.

The usual method to compute H is to initialize a matrix T to be a copy of A, and then apply a sequence of unimodular row transformations to place each column into correct form, from left to right. For example, in (1) the first $k-1$ columns of T are in Hermite form, and next we want to transform the column with entry N (assumed to be positive) into correct form.

$$\overset{C^{(k)}}{\left[\begin{array}{ccccc|ccc}
1 &&&&&&& \\
& \ddots &&&&&& \\
&& 1 &&&&& \\
&&& 1 &&&& \\ \hline
&&&& 1 & c_1 & \cdots & c_* \\
&&&&& 1 && \\
&&&&&& \ddots & \\
&&&&&&& 1
\end{array}\right]}
\overset{T}{\left[\begin{array}{cccc|ccc}
\bullet & \cdots & \bar{*} & * & * & \cdots & * \\
& \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
&& \bullet & * & * & \cdots & * \\
&&& N & * & \cdots & * \\ \hline
&&& a & * & \cdots & * \\
&&& b_1 & * & \cdots & * \\
&&& \vdots & \vdots & \ddots & \vdots \\
&&& b_* & * & \cdots & *
\end{array}\right]}
=
\overset{C^{(k)}T}{\left[\begin{array}{cccc|ccc}
\bullet & \cdots & \bar{*} & * & * & \cdots & * \\
& \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
&& \bullet & * & * & \cdots & * \\
&&& N & * & \cdots & * \\ \hline
&&& M & * & \cdots & * \\
&&& b_1 & * & \cdots & * \\
&&& \vdots & \vdots & \ddots & \vdots \\
&&& b_* & * & \cdots & *
\end{array}\right]} \tag{1}$$

Note that we put bars over the entries that are reduced modulo their column leaders. The correct form of the column containing N will have N replaced by $\gcd(N, a, b_1, \ldots, b_*)$. Our algorithm premultiplies T by a conditioning matrix $C^{(k)}$. Then $M = a + c_1 b_1 + \cdots + c_* b_*$ and $\gcd(N, M) = \gcd(N, a, b_1, \ldots, b_*)$. This means, in effect, that at this stage of the computation we can restrict our attention to the northwest block of $C^{(k)}T$ together with the first row of the southwest block. The entries in $C^{(k)}$ are computed using the modulo N extended gcd algorithm presented in §4 and will be small, with $\lg \|C^{(k)}\| = O(\lg \log N)$.

Based on the above approach, together with some additional ideas, we give in §6 an algorithm for computing the Hermite form. The algorithm works for an input matrix $A \in \mathbb{Z}^{n \times m}$ of arbitrary shape and rank profile and has cost $O(nmr^2(\log r + \log ||A||)^3)$ bit operations, r the rank of A and $\|A\| = \max_{ij} |A_{ij}|$. These cost estimates assume standard, quadratic integer arithmetic and compare very well with cost estimates of previously given algorithms (see below). Moreover, the main advantage of the algorithm here is that intermediate space is required for only $O(nr(\log r + \log \|A\|))$ bits. Previous algorithms which support the running time stated above may require a factor of r more space than this.

In §7 we give an $O(nmr^2(\log r + \log ||A||)^3)$ bit operations algorithm for computing the Smith canonical form of A. The algorithm is also space efficient, using only $O((n + m)r(\log r + \log \|A\|))$ bits of intermediate storage.

Previous and related work

Algorithm for solving the problems $a + c \perp N$ and $a + c_1 b_1 + \cdots + c_k b_k \perp N$ are given in [8]. The advantage of the algorithms in this paper is support for fast arithmetic, output sensitivity and the analysis in terms of $\omega(N)$. For the second problem we take a new approach which allows us to produce the lexicographically minimal solution. The modulo N extended gcd problem has been applied to computing matrix canonical forms in [8–10]. Actually, the polynomial analogue of the problem is considered in [9]. The study there gave rise to the following open question: given $a, N \in K[x]$, K a finite field of cardinality q, is there always a solution to $a+c \perp N$ with $\deg c \leq \lceil \log_q \deg N \rceil$?

The bit complexity of computing the Hermite and Smith form has been well studied. Consider the case of a square nonsingular input matrix $A \in \mathbb{Z}^{n \times n}$. The algorithms we give here compute both forms with $O(n^4(\log n\|A\|)^3)$ bit operations. Space is required for $O(n^2 \log n\|A\|)$ bits of intermediate storage (i.e., about the same number of bits as required to write down a dense input matrix).

The modulo determinant approach [11–13] has cost $O(n^3\, \mathsf{B}(n \log n\|A\|))$ bit operations for computing the Hermite form. The Smith form can be computed in the same time [14,15]. These algorithms don't make appealing candidates for implementation because intermediate space is required for $O(n^3 \log n\|A\|)$

bits. Motivated by this, the authors of [16] give an algorithm for the Hermite form that has worst cast running time $O(n^5(\log n\|A\|)^2)$ bit operations but space complexity only $O(n^2 \log n\|A\|)$ bits. Another approach to control intermediate space requirements is to use heuristics to guide the elimination [17–19].

Algorithms for propitious input matrices (e.g., with the property that most of the leading diagonal entries in the Hermite form are one) are considered in [10,20]. Smith form computation of sparse matrices is considered in [21,22].

In the case of the Smith form some complexity improvements have been made using randomization. In [23] a space efficient, Monte Carlo probabilistic algorithm is given with running time $O(n^{3.5}(\log n\|A\|)^{2.5}(\log n)^2)$ bit operations (see [24] for a report on implementation of variations and extensions of the approach). The current record cost estimate for computing the Smith form (Monte Carlo) is given in [25]. The algorithms in [13–15,23,25] allow use of fast matrix multiplication techniques.

2 Cost model and preliminaries

In addition to the standard arithmetic operations $a + b$, $a - b$, $a \times b$ and a/b (exact division) we will make use of the following.

- `Quo`(a, b): return q such that $0 \leq a - bq < |b|$.
- `Mod`(a, b): return r such that that $a \equiv r \bmod b$ and $0 \leq r < |b|$.
- `Gcd`(a, b): return the greatest common divisor $g = \gcd(a, b)$.
- `Gcdex`(a, b): return (g, s, t, u, v) with $sv - tu = \pm 1$, $g = \gcd(a, b)$ and

$$\begin{bmatrix} s & t \\ u & v \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} g \\ 0 \end{bmatrix}.$$

- `Inv`(a, b): return c such that $ca \equiv 1 \bmod b$ and $0 \leq c < |b|$.
- `Split`(a, b): return the largest $t \in \mathbb{N}$ such that $t \mid a$ and $t \perp b$.

Cost estimates will be given in terms of a function B. We assume here that either $\mathsf{B}(t) = O(t^2)$ or $\mathsf{B}(t) = O(\mathsf{M}(t) \log t)$ where M is a multiplication time for $\mathbb{Z}$, see [26, Definition 8.26]. The best known methods allow $\mathsf{M}(t) = O(t \log t \log \log t)$.

The number of bits in the binary representation of an integer a is given by

$$\lg a = \begin{cases} 1, & \text{if } a = 0; \\ 1 + \lfloor \log_2 |a| \rfloor, & \text{if } a \neq 0. \end{cases}$$

Let $\lg a, \lg b < t$. Then each of $\times$, $/$, `Quo` and `Mod` costs $O(\mathsf{M}(t))$ bit operations. Each of `Gcd`, `Gcdex` and `Inv` costs $O(\mathsf{B}(t))$ bit operations. Operation `Split` also costs $O(\mathsf{B}(t))$ bit operations using the following implementation, see [27].

$$\texttt{Split}(a,b) := \left[\begin{array}{l} g := b; \\ \textbf{for } i \textbf{ from } 1 \textbf{ to } \lceil \log_2 \lg a \rceil \textbf{ do } g := \texttt{Mod}(g^2, a) \textbf{ od}; \\ \textbf{return } a/\texttt{Gcd}(g,a) \end{array}\right.$$

We will also need some compound operations. Let $(d_1, d_2, \ldots, d_k)$ be a sequence of distinct integers. If $\lg d_1 d_2 \cdots d_k < t$, then the sequences $(\texttt{Mod}(a, d_i))_{1 \leq i \leq k}$ and $(\texttt{Gcd}(a, d_i))_{1 \leq i \leq k}$ can be computed with $O(\mathsf{B}(t))$ bit operations using an algorithm based on a binary tree paradigm [26, Theorem 10.24].

Naive bit complexity

The algorithms we give for computing the Hermite and Smith form potentially perform many integer operations with operands of different length. For these algorithms it will be more realistic to assume standard, quadratic integer arithmetic (i.e., $\mathsf{B}(t) = t^2$). In this model, sometimes called naive bit complexity, two integers a and b can be multiplied with $O((\lg a)(\lg b))$ bit operations, operation $q := \texttt{Quo}(a, b)$ and $\texttt{Mod}(a, b)$ cost $O((\lg q)(\lg b))$ bit operations while operations $\texttt{Gcd}(a, b)$, $\texttt{Gcdex}(a, b)$ and $\texttt{Inv}(a, b)$ cost $O((\lg a)(\lg b))$ bit operations, see for example [28]. Operation `Split` costs $O((\lg a)(\lg a + \lg b))$ bit operations using the following implementation, see [27].

$$\texttt{Split}(a,b) := \left[\begin{array}{l} t,\ g := a,\ b; \\ \textbf{while } g \neq 1 \textbf{ do } g := \texttt{Gcd}(g,t);\ \ t := t/g \textbf{ od}; \\ \textbf{return } t \end{array}\right.$$

Bit complexity results that are not given in terms of B make the implicit assumption that $\mathsf{B}(t) = O(t^2)$.

Linear system solving

A key subroutine of our Hermite form algorithm is to solve a nonsingular linear system. For an integer matrix A we define $\|A\| = \max_{ij} |A_{ij}|$, similarly for a vector b. We begin with some classical size bounds. Part (a) follows from Hadamard's inequality and part (b) from Cramer's rule.

Fact 1 *Let $A \in \mathbb{Z}^{n\times n}$ and $b \in \mathbb{Z}^{n\times n}$. Then*

(a) $d := |\det A| \leq n^{n/2}\|A\|^n$.
(b) *If $d \neq 0$ then $v := A^{-1}db$ is over $\mathbb{Z}$ with $\|v\| \leq n^{n/2}\|A\|^{n-1}\|b\|$.*

An eminently practical method to recover v is to use p-adic lifting [29,30]. This requires knowing a small integer p such that $p \perp \det(A)$. If we are given $d = \det(A)$, then p can be chosen to be the smallest odd prime with $p \perp d$, found quickly using a standard sieving technique.

Lemma 2 (p-adic lifting) *Given A, d and b, the vector v in Fact 1(b) can be computed with $O(n^3(\log n + \log\beta)^2)$ bit operations, where $\beta = \max(\|A\|, \|b\|)$.*

Highly optimized implementations of p-adic lifting for the linear system solving problem have been developed [31–33]. A reduction of the problem to integer matrix multiplication is given in [34].

3 The modular gcd problem

This section gives an algorithm to solve the following problem: given $a \in \mathbb{Z}$ and $N \in \mathbb{N}$, find the smallest nonnegative c such that $a + c \perp N$. If $a \perp N$ then $c = 0$. Assume henceforth that $\gcd(a, N) > 1$ so that $c > 0$.

Our algorithm works with a set $\mathcal{S} \subset \mathbb{Z}_{>1}$ of pairwise relatively prime divisors of N. The set $\mathcal{S}$ may be modified during the algorithm but will always satisfy the following property: for all primes p, $p \perp N$ if and only if p is relatively prime to every element of $\mathcal{S}$ (Notation: $p \perp \mathcal{S}$).

Initially we can choose $\mathcal{S} = \{N\}$. For a positive integer K we will search for c in the range $[1 \ldots, K-1]$. Initially we can choose $K = 1$. The idea of our algorithm is to compute one of the following:

(1) The smallest nonnegative c such that $a + c \perp \mathcal{S}$.
(2) The information that $c > K - 1$.
(3) For some $d \in \mathcal{S}$, a set $\mathcal{R} \subset \mathbb{Z}_{>1}$ with the property that for all primes p,

$p \perp d$ if and only if $p \perp \mathcal{R}$. Furthermore,
(a) $\mathcal{R} = \{e\}$ with $e^2 \leq d$, or
(b) $\mathcal{R} = \{e, f\}$ with $e \perp f$ and $ef \leq d$.

In the first case we have solved our problem. In the second case we double the value of K and try again. In the third case we set $\mathcal{S} = (\mathcal{S} \setminus \{d\}) \cup \mathcal{R}$ and repeat the algorithm with the "refined" set $\mathcal{S}$. The next lemma bounds the number of times case 3 can happen.

Lemma 3 *Let $\mathcal{S} = \{N\}$ initially. Then the number of times $\mathcal{S}$ can be refined as $\mathcal{S} = (\mathcal{S} \setminus \{d\}) \cup \mathcal{R}$, with d and $\mathcal{R}$ as in case 3, is bounded by $\omega(N) \log_2 \lg N$.*

PROOF. Consider the refinement of $\mathcal{S}$ as a tree with root labelled N. Each internal node has one or two children, corresponding to case 3(a) or 3(b) respectively, and the leaves are labelled with the elements of the final $\mathcal{S}$. The number of internal nodes with two children is bounded by $\omega(N) - 1$. The number of leaves is bounded by $\omega(N)$. Consider the path from a given leaf node back to the root. The number of nodes on this path which correspond to an instance of case 3(b) is bounded by $\lfloor \log_2 \lg N \rfloor$, since $e^2 \leq d$.

Now we explain the algorithm. The idea is to arrive at case 1, 2 or 3 with only a single "vector gcd" computation, the total cost of which is comparable to one gcd with N. Initialize a filter array $B[1 \ldots K-1]$ so that every entry is zero by default. For each $d \in \mathcal{S}$, the following code sets $B[q] = 1$ for all $q \in [1 \ldots K-1]$ with $\gcd(a+q, d) = d$. Write $\mathcal{S} = \{d_1, d_2, \ldots, d_k\}$.

$(s_i)_{1 \leq i \leq k} := (d_i - \mathtt{Mod}(a, d_i))_{1 \leq i \leq k}$;
for i **from** 1 **to** k **do**
  while $s_i < K$ **do**
    $B[s_i] := 1;\ \ s_i := s_i + d_i$
  od
od;

Now find the smallest q such that $B[q] = 0$.

for q **from** 1 **to** $K-1$ **while** $B[q] = 1$ **do od**;

If $q = K$ then case 2 holds. If $q < K$ then continue with the following:

$(g_i) := (\mathtt{Gcd}(a+q, d_i))_{1 \leq i \leq k}$;
for i **from** 1 **to** k **while** $g_i = 1$ **do od**;

If $i > k$ then case 1 holds with $c = q$. Assume henceforth that $i \leq k$ and let $d = d_i$. Then $d = (g_i)(d/g_i)$ is a nontrivial factorization of d. From this factorization we will produce a set $\mathcal{R}$ as required by case 3. Let $(e, h) = (\min(g_i, d_i/g_i), \max(g_i, d_i/g_i))$. Then $e^2 \le d$ and $d = eh$ but it may be that e is not relatively prime to h. This is solved by replacing h with $f := \texttt{Split}(h, e)$, the largest factor of h which satisfies $f \perp e$. In particular, for all primes p we have $p \perp \{e, f\}$ if and only if $p \perp \{e, h\}$. Set $\mathcal{R} = \{e, f\} \setminus \{1\}$.

Algorithm `Rgcd` shown in Figure 1 implements the approach described above with the following modifications made for efficiency. In case 2 we keep the array B and continue filtering and searching where we left off. In case 3 we update the array B directly by filtering with the at most two new elements in $\mathcal{R}$. Noting that the algorithm tests $q = 1, 2, \ldots, c$ in succession, the next

Algorithm $\texttt{Rgcd}(\mathcal{S}, a)$
Input: A set $\mathcal{S} \subset \mathbb{Z}_{>1}$ and $a \in \mathbb{Z}$.
Output: The smallest nonnegative c with $a + c \perp \mathcal{S}$.
Comment Initialize k and d_* so that $\mathcal{S} = \{d_1, d_2, \ldots, d_k\}$.
if $a \perp \mathcal{S}$ **then return** 0 **fi**;
$q := 1$; $K := 1$; $(s_i)_{1\le i\le k} := (d_i - \texttt{Mod}(a, d_i))_{1\le i\le k}$;
do
  while $q < K$ **and** $B[q] = 1$ **do** $q := q + 1$ **od**;
  if $q = K$ **then**
    $K := 2K$;
    for i **from** 1 **to** k **do**
      while $s_i < K$ **do** $B[s_i] := 1$; $s_i := s_i + d_i$ **od**
    od;
    next
  fi;
  $(g_i)_{1\le i\le k} := (\texttt{Gcd}(a + q, d_i))_{1\le i\le k}$;
  for i **from** 1 **to** k **while** $g_i = 1$ **do od**;
  if $i > k$ **then return** q **fi**;
  $(e, h) := (\min(g_i, d_i/g_i), \max(g_i, d_i/g_i))$;
  $d_i := e$; $s_i := d_i - \texttt{Mod}(a, d_i)$;
  while $s_i < K$ **do** $B[s_i] := 1$; $s_i := s_i + d_i$ **od**;
  $f := \texttt{Split}(h, e)$;
  if $f = 1$ **then next fi**;
  $k := k + 1$; $d_k := f$; $s_k := d_k - \texttt{Mod}(a, d_k)$;
  while $s_k < K$ **do** $B[s_k] := 1$; $s_k := s_k + d_k$ **od**
od

Fig. 1. Algorithm `Rgcd`

result follows from the discussion above.

Theorem 4 *Algorithm* `Rgcd` *is correct.*

Before considering the cost analysis of algorithm `rgcd` we give an example.

Suppose we call `rgcd` with $\mathcal{S} = \{4071600\}$ and $a = 3448475$. Since $a \nmid 4071600$ the algorithm initializes $K = 1$ and $q = 1$. The following list gives the value of K and the state of $B[1.\ldots.K-1]$ at the start of each iteration of the main **do** loop and indicates what case occurs.

(1) $K = 1$. $q > K - 1$ and case 2 occurs: K is doubled.
(2) $K = 2$ and $B[1] = 0$. q remains 1 and case 3 (b) occurs: 4071600 is replaced with $\{36, 9425\}$; now $\mathcal{S} = \{36, 9425\}$.
(3) $K = 2$ and $B[1] = 1$. q is incremented to 2 and case 2 occurs: K is doubled.
(4) $K = 4$ and $B[1\ldots3] = [1, 0, 0]$. q remains 2 and case 3 (b) occurs: 9425 is replaced with $\{29, 325\}$; now $\mathcal{S} = \{36, 29, 325\}$.
(5) $K = 4$ and $B[1\ldots3] = [1, 1, 0]$. q is incremented to 3 and case 3 (b) occurs: 36 is replaced with $\{2, 9\}$; now $\mathcal{S} = \{2, 29, 325, 9\}$.
(6) $K = 4$ and $B[1\ldots3] = [1, 1, 1]$. q is incremented to 4 and case 2 occurs: K is doubled.
(7) $K = 8$ and $B[1\ldots7] = [1, 1, 1, 0, 1, 0, 1]$. q remains 4 and case 3 (a) occurs: 9 is replaced with 3; now $\mathcal{S} = \{2, 29, 325, 3\}$.
(8) $K = 8$ and $B[1\ldots7] = [1, 1, 1, 1, 1, 0, 1]$. q is incremented to 6 and case 1 occurs: the algorithm returns 6.

We now bound the cost of the algorithm. All the sieving of B is performed by three lines in the algorithm that have the following form:

while $s < K$ **do** $B[s] := 1;\ s := s + d$ **od**;

These can be written equivalently as:

(a)**if** $s < K$ **do**
(b) **while** $s < K$ **do** $B[s] := 1;\ s := s + d$ **od**;
(c)**fi**;

The upper bound K takes on the values $K = 1, 2, 4, \ldots, C$ where $C = 2^{\lceil \log_2(c+1) \rceil} \le 2(c + 1)$. Thus, to bound the total cost of all executions of line (b) it will suffice to bound the cost of

while $s < C$ **do** $B[s] := 1;\ s := s + d$ **od**;

over all $s = d - \mathtt{Mod}(a, d)$ such that $s < C$ and $d \in \mathcal{S}$ at some point during the algorithm. A single execution of the while loop costs $2 \lg C$ bit operations. The total cost is $2 \lg C \sum_s C/s \le 2C \lg C \sum_{1 \le i < C} 1/i = O(c(\log c)^2)$ bit operations.

Now consider all other computation in the main loop of the algorithm, excluding the filtering steps corresponding to line (b) above. Lines before the call to `Gcd` have cost $O(\lg N)$ bit operations and are repeated $\log_2 C$ times. The call to `Gcd` and all subsequent lines have cost $O(\mathsf{B}(\log N))$. The number of calls to $\texttt{Gcd}$ is bounded both by c and by the bound given by Lemma 3. This gives the total cost bound

$$O(c(\log c)^2 + (\log c)(\log N) + \mathsf{B}(\log N)\min(c, \omega(N)\log\log N))$$

bit operations. If $c \log c \geq \log N$ then the first term dominates the second. If $c \log c < \log N$ then the third term dominates the second. We get the following result.

Theorem 5 *With input $\mathcal{S} = \{N\}$ and $|a| \leq N$, the cost of algorithm* $\texttt{Rgcd}$ *is* $O(c(\log c)^2 + \mathsf{B}(\log N)\min(c, \omega(N)\log\log N))$ *bit operations.*

Theorem 6 ([4]) *The output of Algorithm* $\texttt{Rgcd}$ *satisfies* $c = O(\omega(N)^2(\log \omega(N))^2)$.

Corollary 7 *The cost estimate of Theorem 5 simplifies to* $O(\mathsf{B}(\log N)\log N)$ *bit operations.*

4 The modular extended gcd problem

This section gives an algorithm to solve the following problem: given $N \in \mathbb{N}$ and $a, b_1, b_2, \ldots, b_k \in \mathbb{Z}$ with $\gcd(N, a, b_1, b_2, \ldots, b_k) = 1$, find the lexicographically minimal sequence of nonnegative integers $c_1, c_2, \ldots, c_k$ such that $a + c_1b_1 + c_2b_2 + \ldots + c_kb_k \perp N$. An algorithm for solving the problem in the special case $k = 1$ is given by the following lemma.

Lemma 8 *Let $N \in \mathbb{N}$ and $a, b \in \mathbb{Z}$ satisfy $\gcd(N, a, b) = 1$. Then*

$$c := \texttt{Rgcd}(\{\texttt{Split}(N, b)\}, a\,\texttt{Inv}(b, \texttt{Split}(N, b)))$$

is the smallest nonnegative integer such that $a + cb \perp N$.

PROOF. Let $t = \texttt{Split}(N, b)$. Then

$$\begin{aligned} N \perp a + cb &\Leftrightarrow t \perp a + cb \\ &\Leftrightarrow t \perp \texttt{Inv}(b, t)(a + cb) \\ &\Leftrightarrow t \perp a\,\texttt{Inv}(b, t) + c \end{aligned}$$

Correctness now follows from Lemma 4. $\square$

The idea for extending the approach of Lemma 8 to the case when $k > 1$ is motivated by the following lemma, the proof of which is obvious.

Lemma 9 *A necessary condition for $a + c_1b_1 + \cdots + c_kb_k \perp N$ is that*

$$a + c_1b_1 + \ldots + c_ib_i \perp \gcd(N, b_{i+1}, b_{i+2}, \ldots, b_k)$$

for $0 \le i \le k$.

For $i = 1, 2, \ldots, k$ in succession, let c_i be the minimal nonnegative c_i such that

$$\overbrace{a + c_1b_1 + \ldots + c_{i-1}b_{i-1}}^{\bar{a}} + c_ib_i \perp \gcd(N, b_{i+1}, \ldots, b_k).$$

It then follows from Lemma 9 that $c_1, c_2, \ldots, c_k$ will be lexicographically minimal among all solutions to our problem. Algorithm `Mgcdex`, shown in Figure 2, implements efficiently the approach just described.

Algorithm `Mgcdex`$(N, a, b_1, b_2, \ldots, b_k)$
Input: $N \in \mathbb{N}$ and $a, b_1, b_2, \ldots, b_k \in \mathbb{Z}$ such that $\gcd(N, a, b_1, \ldots, b_k) = 1$.
Output: $c_1, c_2, \ldots, c_k \in \mathbb{Z}_{\ge 0}$ with $\gcd(N, a + c_1b_1 + \cdots c_lb_k) = 1$.
$h := N$;
for i **from** k **by** -1**to** 1 **do**
  $N_i :=$ `Split`(h, b_i);
  $h :=$ `Gcd`(h, b_i)
od;
$\bar{a} := a$;
for i **from** 1 **to** k **do**
  $c_i :=$ `Rgcd`$(\{N_i\}, \bar{a}$ `Inv`$(b_i, N_i))$;
  $\bar{a} := \bar{a} + c_ib_i$
od;
return $c_1, c_2, \ldots, c_k$

Fig. 2. Algorithm `Mgcdex`

Theorem 10 *Algorithm* `Mgcdex` *is correct. If $|a|, |b_1|, \ldots, |b_k| < N$ then the cost of algorithm is $O(\mathsf{B}(\log N)(k + \log N))$ bit operations.*

PROOF. After the first loop completes, $N_i =$ `Split`$(\gcd(N, b_{i+1}, \ldots, b_k), b_i)$ for $i = 1, 2, \ldots, k$. Lemma 8 together with induction on i (base case $i = 1$) shows that at each check for the termination condition of the second loop we have $\bar{a} = a + c_1b_1 + \cdots + c_{i-1}b_{i-1}$ and `Gcd`$(N, \bar{a}, b_i, \ldots, b_k) = 1$. The final case $i = k + 1$ shows correctness.

Now for the cost analysis. Since $N_1N_2 \cdots N_l | N$, the total cost of all calls to `Rgcd` is bounded by $O(\mathsf{B}(\lg N_1) \lg N_1 + \cdots + \mathsf{B}(\lg N_k) \lg N_k) = O(k + \mathsf{B}(\log N) \log N)$ bit operations. The cost of the remaining computation is bounded by $O(\mathsf{B}(\log N)k)$. $\square$

Corollary 11 *The output of Algorithm* `Mgcdex` *satisfies:*

(a) *The number of nonzero c_i's is bounded by $\omega(N) = O(\log N / \log\log N)$.*
(b) *There is an absolute constant γ such that $1 + c_1 + \ldots + c_k < \gamma(\log N)^2$.*
(c) $\sum_{1 \le i \le k,\, c_i \ne 0} \lg c_i < \omega(N) \lg(\gamma(\log N)^2) = O(\log N)$.

PROOF. Each $N_i \mid N$ and $N_i \perp N_j$ for $i \ne j$. If $N_i = 1$ then $c_i = 0$. Part (a) follows since N can have at most r pairwise relatively prime divisors.

If $c_i \ne 0$ then $N_i > 1$. Now use Theorem 6 to bound each c_i and note that $\sum_{1 \le i \le k,\, N_i > 1} (\log N_i)^2 \le (\sum_{1 \le i \le k,\, N_i > 1} \log N_i)^2$. This shows part (b).

Part (c) follows from parts (a) and (b). $\square$

5 The Hermite form

A matrix $H \in \mathbb{Z}^{n \times m}$ is said to be in Hermite form if the following conditions are satisfied.

(1) [Echelon shape] Let r be the number of nonzero rows of H. Then the first r rows of H are nonzero. For $1 \le i \le r$ let $H[i, j_i]$ be the first nonzero entry in row i. Then $j_1 < j_2 < \ldots < j_r$.
(2) [Uniqueness] $0 \le H[k, j_i] < H[i, j_i]$ for $1 \le k < i \le r$.

Recall that a square matrix U over $\mathbb{Z}$ is said to be unimodular if it has an inverse over $\mathbb{Z}$ (i.e., if $\det U = \pm 1$). Corresponding to every $A \in \mathbb{Z}^{n \times m}$ is a unimodular matrix $U \in \mathbb{Z}^{n \times n}$ such that $UA = H$ with H in Hermite form. The matrix H is called the Hermite canonical form of A. The submatrix comprised of the nonzero rows of H is called the Hermite basis of A: it's rows give a canonical basis for the set of all $\mathbb{Z}$-linear combinations of rows of A. If A is square and nonsingular then the unimodular transformation matrix U is also unique and is given by HA^{-1}.

The usual method to obtain the form is to apply a sequence of row transformations to place column k in correct form, $k = 1, 2, \ldots, m$. Suppose the first $k - 1$ columns of a matrix A have already been transformed, say

$$A = \left[\begin{array}{c|c} H_1 & * \\ \hline & * \end{array}\right] \in \mathbb{Z}^{n \times m} \tag{2}$$

with $H_1 \in \mathbb{Z}^{(r-1)\times(k-1)}$ the Hermite basis of the first $k - 1$ columns of A. If all entries in the first column of the southeast block are zero, then the k'th column of A already has the correct shape and we can proceed directly to column $k+1$. Otherwise, the following code will transform the k'th column to have the correct shape.

> **for** i **from** $r+1$ **to** n **do**
>
> $\quad (g, s, t, u, v) := \texttt{Gcdex}(A[r,k], A[i,k]);$
>
> $$\begin{bmatrix} A[r,*] \\ A[i,*] \end{bmatrix} := \begin{bmatrix} s & t \\ u & v \end{bmatrix} \begin{bmatrix} A[r,*] \\ A[i,*] \end{bmatrix};$$
>
> **od**;
>
> **if** $A[r,k] < 0$ **then** $A[r,*] := -A[r,*]$ **fi**;
>
> **for** i **from** 1 **to** $r-1$ **do**
>
> $\quad q := \texttt{Quo}(A[i,k], A[r,k]);$
>
> $\quad A[i,*] := A[i,*] - qA[r,*]$
>
> **od**

The above approach can be applied for $k = 1, 2, \ldots, m$ to get an algorithm for transforming any $A \in \mathbb{Z}^{n\times m}$ to Hermite form. Although this requires only $O(nmr)$ integer operations, where r is the rank of A, without care the bit-lengths of intermediate numbers in the blocks labelled $*$ in (2) may grow exponentially large as a function of r and thus lead to an unsatisfactory bit complexity [35]. However, if the the input matrix is mostly transformed already, the approach can be used to complete the transformation.

Lemma 12 *Let $A \in \mathbb{Z}^{n\times m}$ have first $m-2$ columns in Hermite form. Then the Hermite form of A can be computed with $O(n(\log \|A\|)^2)$ bit operations.*

The next facts will be used in the next section. All follow from the definition of the Hermite form.

Fact 13 *Let $A \in \mathbb{Z}^{n\times n}$ be nonsingular with Hermite form H. Then*

(a) $\det H = d$, *where* $d = |\det A|$.
(b) $\max_i\{\sum_j \lg H_{ij}\} = O(n + \lg d)$.
(c) *For any $b \in \mathbb{Z}^{n\times 1}$, the Hermite form of $[A|b] \in \mathbb{Z}^{n\times(n+1)}$ is $[H|HA^{-1}b]$.*

6 Computing the Hermite Form

For now, let $A \in \mathbb{Z}^{n\times m}$ have full column rank (i.e., rank m). Our algorithm proceeds in stages for $k = 0, 1, 2, \ldots, m$, exploiting the fact that the first k columns of the Hermite form of A depend only on the first k columns of A. We begin with an overview of the approach by considering the situation at the end of stage k.

In general, the Hermite basis of a matrix depends on all the rows (e.g., the Hermite basis of a column vector is given by the gcd of all entries). Write A using a block decomposition as

$$A = \left[\begin{array}{c|c} B_1 & * \\ \hline B_2 & * \end{array}\right] \in \mathbb{Z}^{n \times m}$$

where B_1 is $k \times k$. At this point we have computed a unit upper triangular matrix $C \in \mathbb{Z}^{n \times n}$ such that

$$\begin{array}{cccc} C & A & & \\ \left[\begin{array}{c|c} * & * \\ \hline & I_{n-k} \end{array}\right] & \left[\begin{array}{c|c} B_1 & * \\ \hline B_2 & * \end{array}\right] & = & \left[\begin{array}{c|c} E & * \\ \hline B_2 & * \end{array}\right] \end{array}$$

where the $k \times k$ matrix E is nonsingular and, moreover, the Hermite basis of the first $k-1$ columns of E is equal to the Hermite basis of the first $k-1$ columns of A (e.g., if $k=2$ then the gcd of the two entries in first column of E is equal to the gcd of all entries in the first column of A). It follows that the square and nonsingular matrix

$$\left[\begin{array}{c|c} I_{n-k} & B_2 \\ \hline & E \end{array}\right] \tag{3}$$

has Hermite form

$$\left[\begin{array}{c|c} I_{n-k} & H_2 \\ \hline & H_1 \end{array}\right] \tag{4}$$

where H_1 is the Hermite form of E and H_2 has all entries in the first $k-1$ columns zero. All the matrices E, H_1 and H_2 have been explicitly computed at this point. Note that $H_1 E^{-1}$ is the unique unimodular matrix that transforms E to Hermite form. It follows that the product of the first three matrices in the following identity (which may easily be verified) is also unimodular.

$$\begin{array}{ccccccc} & & & C & A & & T \\ \left[\begin{array}{c|c} H_1 & \\ \hline H_2 & I_{n-k} \end{array}\right] & \left[\begin{array}{c|c} I_k & \\ \hline -B_2 & I_{n-k} \end{array}\right] & \left[\begin{array}{c|c} E^{-1} & \\ \hline & I_{n-k} \end{array}\right] & \left[\begin{array}{c|c} * & * \\ \hline & I_{n-k} \end{array}\right] & \left[\begin{array}{c|c} B_1 & * \\ \hline B_2 & * \end{array}\right] & = & \left[\begin{array}{c|c} H_1 & * \\ \hline H_2 & * \end{array}\right] \end{array}. \tag{5}$$

The matrix T is a partial, unimodular triangularization of A: the first $k-1$ columns are in Hermite form, see (1). The current stage applies unimodular row operations to place into correct form column k: the last column of H_1 and H_2 in (5), which is the column containing N in (1). The unknown blocks labelled $*$ in T can be computed as in (5). In fact, at this stage our algorithm will compute only the first column of these blocks and then update H_1, H_2,

B_1, B_2, E and C so that (5) holds with k increased by one. This motivates the following definition. In the definition, the matrix B can be considered to be the first k columns of the matrix A from the example above.

Definition 14 *Let $B \in \mathbb{Z}^{n\times k}$ have full column rank. A* conditioning tuple *for B is a tuple of matrices (C, E, H_1, H_2) that satisfies:*

- *$C \in \mathbb{Z}^{n\times n}$ is unit upper triangular and $E \in \mathbb{Z}^{k\times k}$ is nonsingular. C and E can be written as and satisfy*

$$\left[\begin{array}{c} E \\ \hline B_2 \end{array}\right] = \begin{array}{c} C \\ \left[\begin{array}{c|c} * & * \\ \hline & I_{n-k} \end{array}\right] \end{array} \begin{array}{c} B \\ \left[\begin{array}{c} B_1 \\ \hline B_2 \end{array}\right] \end{array} \tag{6}$$

  where B_1 is the first k and B_2 the last $n-k$ rows of B.
- *$H_1 \in \mathbb{Z}^{k\times k}$ is the Hermite form of E and*

$$\left[\begin{array}{c|c} I_{n-k} & B_2 \\ \hline & E \end{array}\right]$$

  has Hermite form

$$\left[\begin{array}{c|c} I_{n-k} & H_2 \\ \hline & H_1 \end{array}\right].$$

  where $H_2 \in \mathbb{Z}^{(n-k)\times k}$ has first $k-1$ columns zero.

Example 15 *$(I_n, I_0, I_0, 0^{n\times 0})$ is a conditioning tuple for $0^{n\times 0}$.*

Example 16 *Let $B \in \mathbb{Z}^{n\times 1}$ be nonzero. Let $C = I_n$ except that $C[1, j] = 1$ where j is minimal such that $B[j, 1] \neq 0$. Let E be the 1×1 matrix $[(CB)[1, 1]]$. Let H_1 be the 1×1 matrix $[|E[1, 1]|]$. Let H_2 be comprised of the last $n-1$ rows of B except that all entries are reduced modulo $H_1[1, 1]$. Then (C, E, H_1, H_2) is a conditioning tuple for B.*

Let (C, E, H_1, H_2) be a conditioning tuple for a full column rank $B \in \mathbb{Z}^{n\times k}$. Suppose we are given $b \in \mathbb{Z}^{n\times 1}$. We now describe subroutine `LastColumn` that computes the the Hermite form of $[B|b] \in \mathbb{Z}^{n\times(k+1)}$ and returns the last column thereof. The subsequent lemma and proof will fill in the details and provide a cost analysis.

If $k = 0$ then $[B|b] = b$ and we can directly compute the Hermite form of $b \in \mathbb{Z}^{n\times 1}$. Assume henceforth that $k > 0$, and let b_1 be the first k and b_2 be the last $n-k$ rows of b. Compute $e \in \mathbb{Z}^{k\times 1}$ as

$$\begin{array}{cccc} & & C & b \\ \left[\begin{array}{c} e \\ \hline b_2 \end{array}\right] & := & \left[\begin{array}{c|c} * & * \\ \hline & I_{n-k} \end{array}\right] & \left[\begin{array}{c} b_1 \\ \hline b_2 \end{array}\right] \end{array}. \tag{7}$$

Next, construct vectors h_1 and h_2 such that

$$\left[\begin{array}{c|c|c} I_{n-k} & B_2 & b_2 \\ \hline & B_1 & b_1 \end{array}\right]$$

has Hermite form

$$\left[\begin{array}{c|c|c} I_{n-k} & H_2 & h_2 \\ \hline & H_1 & h_1 \end{array}\right]$$

using Fact 13(c):

$$\left[\begin{array}{c} h_2 \\ \hline h_1 \end{array}\right] := \frac{1}{d}\left[\begin{array}{c|c} I_{n-k} & H_2 \\ \hline & H_1 \end{array}\right]\left[\begin{array}{c|c} I_{n-k} & -B_2 \\ \hline & I_k \end{array}\right]\left[\begin{array}{c|c} I_{n-k} & \\ \hline & E^{-1} \end{array}\right] d \left[\begin{array}{c} b_2 \\ \hline e \end{array}\right] \tag{8}$$

where $d = |\det E|$. Once h_1 and h_2 have been computed we have the following matrix

$$\left[\begin{array}{c|c} H_1 & h_1 \\ \hline H_2 & h_2 \end{array}\right] = \left[\begin{array}{cccc|c} \bullet & \bar{*} & \cdots & \bar{*} & * \\ & \bullet & \cdots & \bar{*} & * \\ & & \ddots & \vdots & \vdots \\ & & & N & \bar{N} \\ \hline & & & a & a \\ & & & b_1 & \bar{b}_1 \\ & & & \vdots & \vdots \\ & & & b_l & \bar{b}_l \end{array}\right] \in \mathbb{Z}^{n\times(k+1)} \tag{9}$$

which is left equivalent to $[B|b]$ and has first $k-1$ columns already in Hermite form. Complete the transformation to Hermite form and return the last column.

For now we will give cost estimates in terms of an α such that any $k\times k$ minor of $[B|b]$ is bounded in magnitude by α^k. By Fact 1(a) we can choose

$$\alpha = k^{1/2}\max(\|B\|, \|b\|)\max_{1\le i\le k}\{||C[i,*]||_1\} \tag{10}$$

so that $k^{1/2}\|E\| \le \alpha$.

Lemma 17 *The subroutine* `LastColumn` *described above takes as input* $((C, E, H_1, H_2), B, b)$ *and returns as output the last column of the Hermite form of* $[B|b] \in \mathbb{Z}^{n\times(k+1)}$. *The cost of the subroutine is bounded by* $O(nk^2(\log\alpha)^2)$ *bit operations.*

PROOF. If $k = 0$ then we need only compute the gcd of all entries in b. Assume henceforth that $k > 0$. The computation of e in (7) can be accomplished with $O(nk(\log \alpha)^2))$ bit operations. By Fact 13(a) we can compute $d = |\det E|$ as $d := \det H_1$, the product of the diagonal entries in H. Then $d \leq \alpha^k$ (Fact 1(a)). We now detail how to recover h_1 and h_2 according to (8). First, compute $v = E^{-1}de$ in $O(k^3(\log \alpha)^2)$ bit operations (Lemma 2). Then $\|v\| \leq \alpha^k$ (Fact 1(b)). Set $h_1 := (1/d)(H_1 v)$ and $h_2 := (1/d)(db_2 - B_2 v + H_2 v)$. To see that h_1 may be computed in the allotted time we use the fact that the sum of the bit-lengths of entries in any row of H_1 is bounded by $O(k + \log d)$ (Fact 13(b)). The matrix in (9), which has entries bounded in length by $O(k \log \alpha)$ bits, can be transformed to Hermite form using Lemma 12. $\square$

Like before, let (C, E, H_1, H_2) be a conditioning tuple for a full column rank $B \in \mathbb{Z}^{n \times k}$ and let $b \in \mathbb{Z}^{n \times 1}$. Suppose that $[B|b] \in \mathbb{Z}^{n \times (k+1)}$ has rank $k+1$. We describe subroutine `Update` that computes a conditioning tuple $(\bar{C}, \bar{E}, \bar{H}_1, \bar{H}_2)$ for $[B|b]$. If $k = 0$ then construct the conditioning tuple as in Example 16. Assume henceforth that $k > 0$.

Construct the matrix T as in (9). By assumption $[B|b]$ has full column rank, so the trailing $(l+2) \times 2$ submatrix of the matrix shown in (9) also has full column rank; we will precondition this submatrix as shown in (11). Using Algorithm `Mgcdex` find $c_1, c_2, \ldots, c_l \in \mathbb{Z}_{\geq 0}$ such that $N/g \perp M/g$, where $M = a + c_1 b_1 + \cdots + c_l b_l$ and $g = \gcd(N, a, b_1, \ldots, b_l)$.

$$\begin{bmatrix} 1 & & & & \\ & 1 & c_1 & \cdots & c_l \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{bmatrix} \begin{bmatrix} N & \bar{N} \\ a & \bar{a} \\ b_1 & \bar{b}_1 \\ \vdots & \vdots \\ b_l & \bar{b}_l \end{bmatrix} = \begin{bmatrix} N & \bar{N} \\ M & \bar{M} \\ b_1 & \bar{b}_1 \\ \vdots & \vdots \\ b_l & \bar{b}_l \end{bmatrix} \tag{11}$$

We want to ensure also that the principal 2×2 submatrix of the preconditioned matrix is nonsingular. If this is not the case, that is if $N\bar{M} - \bar{N}M = 0$, find the minimal l such that $N\bar{b}_l - \bar{N}b_l \neq 0$ and replace c_l with $c_l - \texttt{Mgcdex}(N/g, M/g - b_l/g, -b_l/g) - 1$, subsequently updating M and $\bar{M}$ to satisfy (11). Then $N\bar{M} - \bar{N}M \neq 0$ as well as $N/g \perp M/g$. Construct $\bar{C}$ as follows: first let $\bar{C}$ be a copy of C; then set $\bar{C}[k+1, j] := c_{j-k+1}$ for $k+1 \leq j \leq n$. Construct $\bar{E}$ as follows: first let $\bar{E}$ be a copy of B_1; then augment $\bar{E}$ with column vector e; then augment $\bar{E}$ with row $k+1$ of $\bar{C}[B|b]$. Note that $\bar{E}$ is $(k+1) \times (k+1)$ whereas E is $k \times k$.

For the construction of $\bar{H}_1$ and $\bar{H}_2$ consider that the matrix

$$\left[\begin{array}{ccc|ccccc} 1 & & & & & & b_1 & \bar{b}_1 \\ & \ddots & & & & & \vdots & \vdots \\ & & 1 & & & & b_l & \bar{b}_l \\ \hline & & & \bullet & * & \cdots & * & * \\ & & & & \bullet & \cdots & * & * \\ & & & & & \ddots & \vdots & \vdots \\ & & & & & & N & \bar{N} \\ & & & & & & M & \bar{M} \end{array}\right] \in \mathbb{Z}^{n \times n} \tag{12}$$

has Hermite form

$$\left[\begin{array}{c|c} I_{n-k-1} & \bar{H}_2 \\ \hline & \bar{H}_1 \end{array}\right] = \left[\begin{array}{ccc|ccccc} 1 & & & & & & & \bar{*} \\ & \ddots & & & & & & \vdots \\ & & 1 & & & & & \bar{*} \\ \hline & & & \bullet & * & \cdots & * & * \\ & & & & \bullet & \cdots & \bar{*} & \bar{*} \\ & & & & & \ddots & \vdots & \vdots \\ & & & & & & \bullet & \bar{*} \\ & & & & & & & \bullet \end{array}\right]. \tag{13}$$

Construct the matrix in (12) from the matrix in (9) by permuting rows and replacing a and $\bar{a}$ with M and $\bar{M}$ respectively. The Hermite form in (13) can be computed using Lemma 12 and will have entries in the second last column of $\bar{H}_2$ all zero because $\gcd(N, M)$ divides b_i, $1 \le i \le l$. Then $(\bar{C}, \bar{E}, \bar{H}_1, \bar{H}_2)$ is a conditioning tuple for $[B|b]$. We get the following.

Lemma 18 *Assume $[B|b]$ has rank $k+1$. The subroutine* `Update` *described above takes as input $((C, E, H_1, H_2), B, b)$ and returns as output a conditioning tuple $(\bar{C}, \bar{E}, \bar{H}_1, \bar{H}_2)$ for $[B|b]$. The cost of the subroutine is bounded by $O(nk^2(\log\alpha)^2 + k^3(\log\alpha)^3)$ bit operations.*

Algorithm `Hermite` is shown in Figure 3. To begin, (C, E, H_1, H_2) is initialized to be a conditioning tuple for the $n \times 0$ matrix B. Then the columns of the Hermite form are computed, one by one from left to right. If B has rank r and $[B|b]$ has rank $r+1$, then the last column h of the Hermite form of $[B|b]$ will have $h[r+1] \neq 0$. In this case, the conditioning tuple is updated. Correctness of the algorithm follows from the definitions (and correctness) of subroutines `LastColumn` and `Update`. Let C be the conditioning matrix that is produced by the algorithm. The loop computes row k of C for $k = 1, 2, \ldots, r$ in succession. The cost estimates for subroutines `LastColumn` and `Update` were given in terms of

$$\alpha(k) = k^{1/2}\|A\| \max_{1 \le i \le k}\{||C[i,*]||_1\}, \tag{14}$$

c.f. (10). Since $\alpha(1) < \ldots < \alpha(r)$, to bound the cost of algorithm `Hermite` it will be sufficient to bound $\lg\alpha(r)$.

Theorem 19 $\lg\alpha(r) = O(\log r + \log\|A\|)$.

Algorithm `Hermite`(A)
Input: The columns, from left to right, of the input matrix A.
Output: The columns, from left to right, of the Hermite form of A.
$r := 0$;
$B := 0^{n\times 0}$;
$(C, E, H_1, H_2) := (I_n, I_0, I_0, 0^{n\times 0})$;
for j **from** 1 **to** m **do**
  $b :=$ column j of A;
  $h :=$ `LastColumn`$((C, E, H_1, H_2), B, b)$;
  Write to output h;
  if $r < n$ **and** $h[r+1] \neq 0$ **then**
    $(C, E, H_1, H_2) :=$ `Update`$((C, E, H_1, H_2), B, b)$;
    $B := [B|b]$;
    $r := r + 1$
  fi
od

Fig. 3. Algorithm `Hermite`

PROOF. Let

$$d = \log \|A\| \quad \text{and} \quad x(k) = \gamma k^2 d^2$$

where γ is the absolute constant of Corollary 11(b). We will prove by induction on k that

$$\max_{1\le i\le k}\{||C[i,*]||_1\} \le 100x(k)(\log x(k))^2 \tag{15}$$

for $1 \le k \le r$. Since $||C[1,*]||_1 \le 2$ the claim (15) holds for $k = 1$. To complete the inductive proof it will be sufficient to show that (15) implies that $||C[k+1,*]||_1 \le 100x(k)(\log x(k))^2$ since $x(k) < x(k+1)$.

Let N be as in (11). By Corollary 11(b) we have $||C[k+1,*]||_1 \le \gamma(\log N)^2$. Using the bound $N \le \alpha(k)^k$ and substituting (15) into (14) yields

$$\begin{aligned}
||C[k+1,*]||_1 &\le \gamma(\log \alpha(k)^k)^2 \\
&= \gamma k^2 \cdot (\log k^{1/2} + d + (\log(100) + \log x(k) + 2\log\log x(k)))^2 \\
&= \gamma k^2 d^2 \cdot ((\log k^{1/2})/d + 1 + (\log(100) + \log x(k) + \log\log x(k))/d)^2 \\
&\le \gamma k^2 d^2 \cdot (\log k^{1/2} + 1 + \log(100) + \log x(k) + \log\log x(k))^2 \\
&< x(k) \cdot (10\log x(k))^2.
\end{aligned}$$

Substituting (15) into (14) for $k = r$ and taking logarithms gives $\log \alpha(r) = O(\log r + d)$. $\square$

At any point in the algorithm, the total size of H_1 will be bounded by $O(r(r + \lg|\det E|))$ bits (Fact 13(b)). This simplifies to $O(r(r\log\alpha(r)))$ bits. The total size of H_2 will be bounded by $O(n(r\log\alpha(r)))$ bits, c.f. (13). The total size of C will be very small, see below. Using Theorem 19, the following theorem follows from Lemmas 17 and 18.

Theorem 20 *The Algorithm* `Hermite` *is correct. The cost of the algorithm is* $O(nmr^2(\log r + \log \|A\|)^2 + r^4(\log r + \log \|A\|)^3)$ *bit operations. The intermediate space requirement of the algorithm is bounded by* $O(nr(\log r + \log \|A\|))$ *bits.*

We now mention how to compute a unimodular transforming matrix. The idea is to embed an $(n-1) \times (m-1)$ input matrix $\bar{A}$ into a larger matrix

$$A = \left[\begin{array}{c|c} \bar{A} & \\ \hline & 1 \end{array}\right] \in \mathbb{Z}^{n \times m}.$$

Let (C, E, H_1, H_2), B and r be as at termination of Algorithm `Hermite` with input A. Let B_2 be the last $n-r$ row of B. Then QCA is in Hermite form where

$$Q = \left[\begin{array}{c|c} H_1 & \\ \hline (H_2 - B_2) & I_{n-r} \end{array}\right] \left[\begin{array}{c|c} E^{-1} & \\ \hline & I_{n-r} \end{array}\right].$$

The matrix Q can be computed using n applications of Lemma 2. Let $\beta = \alpha(r)^r$. Then $\log \beta = O(r(\log r + \log \|A\|))$ (Theorem 19) and the sum of the bit-lengths of all nonzero entries in the first r rows of C will be bounded by $O(r \log \beta)$ (Corollary 11(c)). Thus, the product QC can be computed with $O(nr(\log \beta)^2)$ bit operations. Because of the structure of A, the last row of U will equal $(0, 0, \ldots, 0, 1) \in \mathbb{Z}^{1 \times n}$, so the principal $(n-1) \times (n-1)$ submatrix $\bar{U}$ of U is unimodular and $\bar{U}\bar{A}$ is in Hermite form. This gives the following.

Corollary 21 *Given* $A \in \mathbb{Z}^{n \times m}$, *a unimodular* $U \in \mathbb{Z}^{n \times n}$ *such that* UA *is in Hermite form can be computed with* $O(nmr^2(\log r + \log \|A\|)^2 + r^4(\log r + \log \|A\|)^3)$ *bit operations,* r *the rank of* A.

7 Computing the Smith form

A matrix $S \in \mathbb{Z}^{n \times m}$ is said to be in Smith form if the following conditions are satisfied.

(1) [Diagonal shape] Let r be the number of nonzero entries of S. The $S = \mathrm{diag}(s_1, s_2, \ldots, s_r, 0, \ldots, 0)$.
(2) [Uniqueness] $s_1 \mid s_2 \mid \ldots \mid s_r$ and each $s_i > 0$.

Corresponding to every $A \in \mathbb{Z}^{n \times m}$ there exist unimodular matrices $U \in \mathbb{Z}^{n \times n}$ ($\det U = \pm 1$) and $V \in \mathbb{Z}^{m \times m}$ ($\det V = \pm 1$) such that $UAV = S$. By combining the algorithm `Hermite` from the previous section with some algorithms in [36], we get a deterministic and space efficient algorithm for computing the Smith form of an integer matrix. We first recall the two main subroutines we require.

Consider the following transformation of an $n \times k$ integer matrix.

$$\begin{array}{c} T \\ \left[\begin{array}{cccc|c} h_1 & \bar{*} & \cdots & \bar{*} & * \\ & h_2 & \cdots & \bar{*} & * \\ & & \ddots & \vdots & \vdots \\ & & & h_{k-1} & * \\ \hline & & & & * \\ & & & & \\ & & & & \end{array}\right] \end{array} \longrightarrow \begin{array}{c} \bar{T} \\ \left[\begin{array}{cccc|c} h_1 & \bar{*} & \cdots & \bar{*} & \bar{*} \\ & h_2 & \cdots & \bar{*} & \bar{*} \\ & & \ddots & \vdots & \vdots \\ & & & h_{k-1} & \bar{*} \\ \hline & & & & * \\ & & & & \\ & & & & \end{array}\right] \end{array} \tag{16}$$

The matrix T in (16) is upper triangular with each h_i positive and off-diagonal entries $\bar{*}$ in each *row* (as opposed to each column for the Hermite form) reduced modulo the diagonal entry in each row. The entries $*$ in the last column of T are possibly not reduced. Entry $T[k,k]$ may or may not be zero. The transformed matrix $\bar{T}$ is identical to T except that all off-diagonal entries are reduced. Algorithm `Reduce`, described in [36, Section 2], effects the transformation from T to $\bar{T}$ by adding integer multiples of the first $r-1$ columns of T to the last column.

The following is from [36, Lemmas 2.2 and 2.3].

Lemma 22 *There exists an algorithm* `Reduce` *that takes as input a $T \in \mathbb{Z}^{n\times k}$ as in (16) and returns as output $\bar{T}$. If $\beta \geq \max(h_1 h_2 \cdots h_{k-1}, \|T\|)$ then the cost of the algorithm is $O(k(\log \beta)^2)$ bit operations. The algorithm requires space for $O(k \log \beta)$ bits.*

Now consider transforming an already triangularized matrix

$$T = \left[\begin{array}{cccc|ccc} h_1 & \bar{*} & \cdots & \bar{*} & \bar{*} & \cdots & \bar{*} \\ & h_2 & \cdots & \bar{*} & \bar{*} & \cdots & \bar{*} \\ & & \ddots & \vdots & \vdots & \ddots & \vdots \\ & & & h_r & \bar{*} & \cdots & \bar{*} \\ \hline & & & & & & \\ & & & & & & \\ & & & & & & \end{array}\right] \tag{17}$$

to Smith canonical form. The matrix T in (17) has each diagonal entry h_i positive and all off-diagonal entries in the first r rows reduced modulo the diagonal entry in the same row. Algorithm `TriangularSmith`, described in [36, Section 3], transform T to Smith form by applying a sequence of unimodular row a and column operations.

The following is from [36, Theorem 3.4].

Lemma 23 *There exists an algorithm* `TriangularSmith` *that takes as input a* $T \in \mathbb{Z}^{n\times m}$ *as in (17) and returns as output the Smith form of* T*. If* $\beta \geq h_1 h_2 \cdots h_r$ *then the cost of the algorithm is* $O(mr(\log \beta)^2)$ *bit operations. The algorithm requires space for* $O(m \log \beta)$ *bits.*

Algorithm `Smith` is shown in Figure 4. Since algorithm `Reduce` performs only

Algorithm `Smith`(A)
Input: The columns, from left to right, of the Hermite form of $A \in \mathbb{Z}^{n\times m}$.
Output: The Smith form of A.
$r := 0$;
$T := 0^{n\times 0}$;
$C := 0^{n\times 0}$;
for j **from** 1 **to** m **do**
  $h :=$ column j of the Hermite form of A;
  $h :=$ the last column of `Reduce`$([T|h])$;
  if $r < n$ **and** $h[r+1] \neq 0$ **then**
    $T := [T|h]$;
    $r := r + 1$
  else
    $C := [C|h]$
  fi
od;
return `TriangularSmith`$([T|C])$

Fig. 4. Algorithm `Smith`

column operations, the matrix $[T|C]$ in the last line of the algorithm is equivalent to A. We get the following as a corollary of Lemmas 22 and 23.

Theorem 24 *The algorithm* `Smith` *is correct. The cost of the algorithm is* $O(mr^3(\log r + \log \|A\|)^2)$ *bit operations.*

Now consider running algorithms `Hermite` and `Smith` simultaneously. As algorithm `Hermite` outputs a column of the Hermite form this is passed to algorithm `Smith` and reduced. The reduction reduces the total size of a non pivot column of H from about $r^2 \log \|A\|$ to only $r \log \|A\|$ bits. We get the following as a corollary to Theorems 20 and 24.

Theorem 25 *The algorithms* `Hermite` *and* `Smith` *give a deterministic algorithm for computing the Smith form of an* $A \in \mathbb{Z}^{n\times m}$ *with* $O(nmr^2(\log r + \log \|A\|)^2 + r^4(\log r + \log \|A\|)^3)$ *bit operations,* r *the rank of* A*. The intermediate space requirement is bounded by* $O((n+m)r(\log r + \log \|A\|))$ *bits.*

8 Conclusions

An algorithm for the modulo N extended gcd was developed and applied to the problem of computing the Hermite and Smith canonical form. Our goal in this paper was to develop algorithms that were deterministic, space efficient and handle the case of input matrices with arbitrary shape and rank profile. Within this model we have shown that the Hermite and Smith canonical form of an integer matrix A, with constant size entries and row and column dimension bounded by n, can be computed with $O(n^4(\log n)^3)$ bit operations.

We have implemented versions of the algorithms using GMP [37]. We mention some timing results on an Intel® Itanium® 2 @ 1.3 GHz. The Hermite and Smith form of a nonsingular 300×300 input matrix with single decimal digit entries will be computed in about forty seconds; when the size of the entries in the input matrix is increased to twenty decimal digits the time increases to about nine minutes. A nonsingular 1000×1000 input matrix with single decimal digit entries takes about ninety minutes.

References

[1] E. Jacobsthal, Über Sequenzen ganzer Zahlen, von denen keine zu n teilerfremd ist I–III, Norske Vid. Selsk. Forhdl. 33 (1960) 117–124, 125–131, 132–139.

[2] P. Erdös, On the integers relatively prime to n and on a number–theoretic function considered by Jacobsthal, Math. Scand. 11 (1962) 163–170.

[3] H.-J. Kanold, Über eine zahlentheoretische Funktion von E. Jacobsthal, Abh. Braunschweig. Wiss. Gesellsch. 25 (1975) 7–10.

[4] H. Iwaniec, On the problem of Jacobsthal, Demonstratio Mathematica 11 (1) (1978) 225—231.

[5] G. Robin, Estimation de la fonction de Tchebychef Θ sur le k-ième nombre premier et grandes valeurs de la fonction $\omega(n)$ nombre de diviseurs premiers de n, Acta Arith. 42 (1983) 367–389.

[6] C. C. Sims, Computation with finitely presented groups, Vol. 48 of Encyclopedia of mathematics and its applications, Cambridge University Press, 1984.

[7] H. Cohen, A Course in Computational Algebraic Number Theory, Springer-Verlag, 1996.

[8] A. Storjohann, A solution to the extended gcd problem with applications, in: W. W. Küchlin (Ed.), Proc. Int'l. Symp. on Symbolic and Algebraic Computation: ISSAC '97, ACM Press, New York, 1997, pp. 109–116.

[9] T. Mulders, A. Storjohann, The modulo N extended gcd problem for polynomials, in: O. Gloor (Ed.), Proc. Int'l. Symp. on Symbolic and Algebraic Computation: ISSAC '98, 1998, pp. 105–112.

[10] M. Giesbrecht, M. Jacobson, A. Storjohann, Algorithms for large integer matrix problems, in: Applied Algebra, Algebraic Algorithms and Error Correcting Codes (AAECC-14), LNCS 2227, Springer Verlag, 2001, pp. 297–397.

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

[12] 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 of Computing 18 (4) (1989) 658–669.

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

[14] A. Storjohann, Near optimal algorithms for computing Smith normal forms of integer matrices, in: Y. N. Lakshman (Ed.), Proc. Int'l. Symp. on Symbolic and Algebraic Computation: ISSAC '96, ACM Press, New York, 1996, pp. 267–274.

[15] A. Storjohann, Algorithms for matrix canonical forms, Ph.D. thesis, Swiss Federal Institute of Technology, ETH–Zurich (2000).

[16] D. Micciancio, B. Warinschi, A linear space algorithm for computing the Hermite normal form, in: B. Mourrain (Ed.), Proc. Int'l. Symp. on Symbolic and Algebraic Computation: ISSAC '00, ACM Press, New York, 2001, pp. 231—236.

[17] G. Havas, D. F. Holt, S. Rees, Recognizing badly presented Z-modules, Linear Algebra and its Applications 192 (1993) 137–163.

[18] G. Havas, B. S. Majewski, Hermite normal form computation for integer matrices, Congressus Numerantium 105 (1994) 87–96.

[19] G. Havas, B. S. Majewski, Integer matrix diagonalization, Journal of Symbolic Computation 24 (1997) 399–408.

[20] U. Vollmer, A note on the Hermite basis computation of large integer matrices, in: R. Sendra (Ed.), Proc. Int'l. Symp. on Symbolic and Algebraic Computation: ISSAC '03, ACM Press, New York, 2003, pp. 255–257.

[21] M. Giesbrecht, Fast computation of the Smith form of a sparse integer matrix, Computational Complexity 10 (1) (2001) 41–69.

[22] J. G. Dumas, B. D. Saunders, G. Villard, On efficient sparse integer matrix Smith normal form computation, JSC 32 (1/2) (2001) 71–99.

[23] W. Eberly, M. Giesbrecht, G. Villard, Computing the determinant and Smith form of an integer matrix, in: Proc. 31st Ann. IEEE Symp. Foundations of Computer Science, 2000, pp. 675–685.

[24] D. Saunders, Z. Wan, Smith normal form of dense integer matrices, fast algorithms into practice, in: J. Gutierrez (Ed.), Proc. Int'l. Symp. on Symbolic and Algebraic Computation: ISSAC '04, ACM Press, New York, 2004, pp. 274–281.

[25] E. Kaltofen, G. Villard, On the complexity of computing determinants, Computational Complexity 13 (3–4) (2004) 91–130.

[26] J. von zur Gathen, J. Gerhard, Modern Computer Algebra, 2nd Edition, Cambridge University Press, 2003.

[27] T. Mulders, A. Storjohann, Diophantine linear system solving, in: S. Dooley (Ed.), Proc. Int'l. Symp. on Symbolic and Algebraic Computation: ISSAC '99, ACM Press, New York, 1999, pp. 281–288.

[28] E. Bach, J. Shallit, Algorithmic Number Theory, Vol. 1 : Efficient Algorithms, MIT Press, 1996.

[29] R. T. Moenck, J. H. Carter, Approximate algorithms to derive exact solutions to systems of linear equations., in: Proc. EUROSAM '79, volume 72 of *Lecture Notes in Compute Science*, Springer-Verlag, Berlin-Heidelberg-New York, 1979, pp. 65–72.

[30] J. D. Dixon, Exact solution of linear equations using p-adic expansions, Numer. Math. 40 (1982) 137–141.

[31] P. Giorgi, Arithmetic and algorithmic in exact linear algebra for the LinBox library, Ph.D. thesis, Ecole normale superieure de Lyon, LIP, Lyon, France (December 2004).

[32] Z. Chen, A BLAS based C library for exact linear algebra on integer matrices, Master's thesis, School of Computer Science, University of Waterloo (2005).

[33] Z. Chen, A. Storjohann, A BLAS based C library for exact linear algebra on integer matrices, in: M. Kauers (Ed.), Proc. Int'l. Symp. on Symbolic and Algebraic Computation: ISSAC '05, ACM Press, New York, 2005.

[34] A. Storjohann, The shifted number system for fast linear algebra on integer matrices, Journal of Complexity. To appear.

[35] X. G. Fang, G. Havas, On the worst-case complexity of integer gaussian elimination, in: W. W. Küchlin (Ed.), Proc. Int'l. Symp. on Symbolic and Algebraic Computation: ISSAC '97, ACM Press, New York, 1997, pp. 28–31.

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

[37] T. Granlund, The GNU multiple precision arithmetic library, edition 4.1.4. `http://www.swox.com/gmp` (2004).