Abstract

Extended gcd calculation has a long history and plays an important role in computational number theory and linear algebra. Recent results have shown that finding optimal multipliers in extended gcd calculations is difficult. We study algorithms for finding good multipliers and present new algorithms with improved performance. We present a well-performing algorithm which is based on lattice basis reduction methods and may be formally analyzed. We also give a relatively fast algorithm with moderate performance.

1 Introduction

Euclid’s algorithm for computing the greatest common divisor of 2 numbers is considered to be the oldest proper algorithm known ([12]). This algorithm can be amplified naturally in various ways. Thus, we can use it recursively to compute the gcd of more than two numbers. Also, we can do a constructive computation, the so-called extended gcd, which expresses the gcd as a linear combination of the input numbers.

Extended gcd computation is of particular interest in number theory (see [4, chapters 2 and 3]) and in computational linear algebra ([8, 9, 11]), in
both of which it takes a basic role in fundamental algorithms. An overview of some of the earlier history of the extended gcd is given in [4], showing that it dates back to at least Euler, while additional references and more recent results appear in [15].

In view of many efforts to find good algorithms for extended gcd computation, Havas and Majewski ([15]) showed that this is genuinely difficult. They prove that there are a number of problems for which efficient solutions are not readily available, including: “Given a multiset of $n$ numbers, how many of them do we need to take to obtain the gcd of all of them?”; and “How can we efficiently find ‘good’ multipliers in an extended gcd computation?”.

They show that associated problems are NP-complete and it follows that algorithms which guarantee optimal solutions are likely to require exponential time.

We present new polynomial-time algorithms for solving the extended gcd problem. These outperform previous algorithms. We include some practical results and an indication of the theoretical analyses of the algorithms.

2 The Hermite normal form

As indicated in the introduction, extended gcd calculation plays an important role in computing canonical forms of matrices. On the other hand, we can use the Hermite normal form of a matrix as a tool to help us develop extended gcd algorithms. This relationship can be viewed along the following lines.

An $m \times n$ integer matrix $B$ is said to be in Hermite normal form if

(i) the first $r$ rows of $B$ are nonzero;

(ii) for $1 \leq i \leq r$, if $b_{ij_i}$ is the first nonzero entry in row $i$ of $B$, then $j_1 < j_2 < \cdots < j_r$;

(iii) $b_{ij_i} > 0$ for $1 \leq i \leq r$;

(iv) if $1 \leq k < i \leq r$, then $0 \leq b_{kj_i} < b_{ij_i}$.

Let $A$ be an $m \times n$ integer matrix. Then there are various algorithms for finding a unimodular matrix $P$ such that $PA = B$ is in row Hermite normal form. These include those of Kannan–Bachem [19, pages 349–357]
and Havas–Majewski [9], which attempt to reduce coefficient explosion during their execution.

Throughout this paper we use \( \lfloor x \rfloor \) to mean the nearest integer to \( x \).

Let \( C \) denote the submatrix of \( B \) formed by the \( r \) nonzero rows and write \( P = \begin{bmatrix} Q & R \end{bmatrix} \), where \( Q \) and \( R \) have \( r \) and \( m - r \) rows, respectively. Then \( QA = C \) and \( RA = 0 \) and the rows of \( R \) form a \( \mathbb{Z} \) basis for the sublattice \( N(A) \) of \( \mathbb{Z}^m \) formed by the vectors \( X \) satisfying \(XA = 0 \).

Sims points out in his book [19, page 381] that the LLL lattice basis reduction algorithm ([13]) can be applied first to the rows of \( R \) to find a short basis for \( N(A) \); then by applying step 1 of the LLL algorithm with no interchanges of rows, suitable multiples of the rows of \( R \) can be added to those of \( Q \), thereby reducing the entries in \( Q \) to manageable size, without changing the matrix \( B \). (Also see [6, page 144].)

Many variants and applications of the LLL exist. Here we simply observe that a further improvement to \( P \) is usually obtained by the following version of Gaussian reduction (see [20, page 100] for the classical algorithm): For each row \( Q_i \) of \( Q \) and for each row \( R_j \) of \( R \), compute \( t = \|Q_i - rR_j\|^2 \), where

\[
r = \left\lfloor \frac{(Q_i, R_j)}{(R_j, R_j)} \right\rfloor.\]

If \( t < \|Q_i\|^2 \), replace \( Q_i \) by \( Q_i - rR_j \).

The process is repeated until no further shortening of row lengths occurs.

Before this reduction is done, a similar improvement is made to the rows of \( R \), with the extra condition that the rows are presorted according to increasing length before each iteration.

We have implemented algorithms along these lines and report that the results are excellent for Hermite normal form calculation in general. Details will appear elsewhere. Here we apply it to extended gcd calculation in the following way. Take \( A \) to be a column vector of positive integers \( d_1, \ldots, d_m \). Then \( B \) is a column vector with one nonzero entry \( d = \gcd(d_1, \ldots, d_m) \), \( r = 1 \), \( N(A) \) is an \((m - 1)\)-dimensional lattice and \( Q \) is a row vector of integers \( x_1, \ldots, x_m \) which satisfy

\[
d = x_1d_1 + \cdots + x_md_m.\]
Thus, any Hermite normal form algorithm which computes the transforming matrix incorporates an extended gcd algorithm. The first row of the transforming matrix is the multiplier vector. This immediately carries across the NP-completeness results for extended gcd computation to Hermite normal form computation in a natural way.

We expect to obtain a short multiplier vector for the following reasons. Using the notation of [6, page 142], we have

$$Q = Q^* + \sum_{l=1}^{m-1} \mu_{mj} R_j^*,$$

where $R_1^*, \ldots, R_{m-1}^*, Q^*$ form the Gram–Schmidt basis for $R_1, \ldots, R_{m-1}, Q$.

The vector $A^t$ is orthogonal to $R_1, \ldots, R_{m-1}$ and we see

$$Q^* = \frac{dA^t}{||A||^2}.$$

After the reduction of $Q$, we have

$$Q = \frac{dA^t}{||A||^2} + \sum_{i=1}^{m-1} \lambda_i R_i^*, \quad |\lambda_i| \leq \frac{1}{2}, \quad (i = 1, \ldots, m-1).$$

But $||R_i^*|| \leq ||R_i||$ and in practice the vectors $R_1, \ldots, R_{m-1}$ are very short. The triangle inequality now implies

$$||Q|| \leq 1 + \frac{1}{2} \sum_{i=1}^{m-1} ||R_i||.$$

While our method gives short multiplier vectors $Q$, it does not always produce the shortest vectors. These can be determined for small $m$ by using the Fincke–Pohst algorithm (see [16, page 191]) to find all solutions in integers $x_1, \ldots, x_{m-1}$ of the inequality

$$||Q - \sum_{i=1}^{m-1} x_i R_i||^2 \leq ||Q||^2.$$

Another approach to the extended gcd problem is to apply the LLL algorithm to the lattice $L$ spanned by the rows of the matrix.
\[
W = \begin{bmatrix}
1 & \cdots & 0 & \gamma a_1 \\
\vdots & \ddots & \vdots \\
0 & \cdots & 1 & \gamma a_n
\end{bmatrix}.
\]

If \( \gamma \) is sufficiently large, the reduced basis will have the form

\[
\begin{bmatrix}
P_1 & 0 \\
P_2 & \pm \gamma d
\end{bmatrix},
\]

where \( d = \gcd(a_1, \ldots, a_m) \), \( P = \begin{bmatrix} P_1 \\ P_2 \end{bmatrix} \) is a unimodular matrix and \( P_2 \) is a multiplier vector which is small in practice.

This is true for the following reason: \( L \) consists of the vectors

\[
(X, a) = (x_1, \ldots, x_m, \gamma(a_1 x_1 + \cdots + a_m x_m)),
\]

where \( x_1, \ldots, x_m \in \mathbb{Z} \). Hence \( X \in N(A) \iff (X, 0) \in L \). Also if \((X, a) \in L\) and \( X \) does not belong to \( N(A) \), then \( a \neq 0 \) and

\[
\| (X, a) \| \geq \gamma^2. \tag{1}
\]

Now Proposition 1.12 of [13] implies that if \( b_1, \ldots, b_m \) form a reduced basis for \( L \), then

\[
\| b_j \| \leq 2^{m-1} \max \{ \| X_1 \|, \ldots, \| X_{m-1} \| \} = M, \tag{2}
\]

if \( X_1, \ldots, X_{m-1} \) are linearly independent vectors in \( N(A) \).

Hence if \( \gamma > M \), it follows from inequalities 1 and 2 that the first \( m - 1 \) rows of a reduced basis for \( L \) have the form \((b_{j1}, \ldots, b_{jm}, 0)\).

The last vector of the reduced basis has the form \((b_{m1}, \ldots, b_{mm}, \gamma g)\) and the equations

\[
PA = \begin{bmatrix} 0 \\ g \end{bmatrix}, \quad A = P^{-1} \begin{bmatrix} 0 \\ g \end{bmatrix},
\]

imply \( d \| g \) and \( g \| d \), respectively.

We have implemented and tested all the methods described in this section based on the CALC platform developed by the third author.
3 A new LLL-based algorithm

The theoretical results of the previous section do indicate two specific LLL-based methods for computing small multipliers, but they have some drawbacks. In the first instance, we need to execute some algorithm that computes matrix $P$, apply the LLL algorithm to the null space and finally, carry out the size reduction step on $Q$. The second method requires executing the LLL algorithm on a matrix with numbers $\Omega(m)$ bits long, which incurs severe time penalties. In this section we show how to modify the algorithm of Lenstra, Lenstra and Lovász in such a way that it computes short multipliers without any preprocessing or rescaling. The modification is done along similar lines to those presented in [14], although the resulting algorithm is far simpler, due to the particular nature of our problem.

What follows is a modification of the method as described in [5, pages 83–87] and we follow the notation used there. Let $d_1, d_2, \ldots, d_m$ be the nonnegative (with zeros allowed) integers for which we want to solve the extended gcd problem. It is convenient to think of matrix $b$ as initially set to be

$$b = \begin{bmatrix} 1 & \cdots & 0 & d_1 \\ \vdots & & \vdots & \vdots \\ 0 & \cdots & 1 & d_m \end{bmatrix}.$$ 

However later we see that we do not actually include the last column in the matrix, but handle it separately.

Further, for any two (rational) vectors $u = [u_i]_1$ and $v = [v_i]_1$, we define their weighted inner product $W(u, v)$ to be:

$$u \cdot v = \sum_{i=1}^{l-1} u_i v_i + \gamma^2 u_l v_l,$$

with $\gamma$ tending to infinity. This definition ensures that if either $u_l$ or $v_l$ is 0, the inner product becomes the simple dot product of two vectors. However, if both $u_l$ and $v_l$ are nonzero, the preceding elements are insignificant and may be simply neglected. In the following text we refer to the norm defined by this inner product as the weighted norm.

**DEFINITION 3.1** [5, Definition 2.6.1.] *With the weighted norm, the basis*
\(b_1, b_2, \ldots, b_m\) is called LLL-reduced if

\[ |\mu_{i,j}| = \left| \frac{b_i \cdot b_j^*}{b_j^* \cdot b_j} \right| \leq \frac{1}{2}, \quad \text{for } 1 \leq j < i \leq m \]  

(C1)

and

\[ b_i^* \cdot b_i^* \geq (\alpha - \mu_{i,i-1}^2) b_{i-1}^* \cdot b_{i-1}^* \]  

(C2)

where \(\frac{1}{2} < \alpha < 1\). (Note the slight change in the lower bound for \(\alpha\) from the standard LLL. To ensure proper gcd computation, \(\alpha = 1/4\) is not allowed.)

Assume that vectors \(b_1, b_2, \ldots, b_{k-1}\) are already LLL-reduced. We state, for future use, the following lemma

**Lemma 3.1** If vectors \(b_1, b_2, \ldots, b_{k-1}\) are LLL-reduced with respect to the weighted norm then the following are true:

\[
\begin{align*}
& b_{j,m+1} = 0, \quad \text{for } 1 \leq j < k - 1; \\
& 1 \leq b_{k-1,m+1} \leq \gcd(d_1, d_2, \ldots, d_{k-1}); \\
& b_{k-1,m+1} = b_{k-1,m+1}.
\end{align*}
\]

**Proof.** The proof is a simple induction on conditions (C1) and (C2). The lemma is also implied by the argument from the end of Section 2. \(\square\)

The vector \(b_k\) needs to be reduced so that \(|\mu_{k,k-1}| \leq \frac{1}{2}\). (Here we use a partial size reduction, sufficient to test condition (C2).) This is done by replacing \(b_k\) by \(b_k - [\mu_{k,k-1}]b_{k-1}\). By the definition of \(\mu_{k,k-1}\) we have

\[
\mu_{k,k-1} = \begin{cases} 
\frac{b_{k,m+1}}{b_{k-1,m+1}} & \text{if } b_{k,m+1} \neq 0 \text{ and } b_{k-1,m+1} \neq 0; \\
0 & \text{if } b_{k,m+1} = 0 \text{ and } b_{k-1,m+1} \neq 0; \\
\mu'_{k,k-1} & \text{otherwise.}
\end{cases}
\]

Here \(\mu'_{i,j}\) is defined as \(\mu'_{i,j} = (\sum_{q=1}^{m} b_{i,q}b_{j,q}^*)/(\sum_{q=1}^{m} (b_{j,q}^*)^2)\), i.e., \(\mu'_{i,j}\) is a Gram-Schmidt coefficient for \(b\) with the last column ignored. In other words, the algorithm first tries to reduce \(b_{k,m+1}\). If \(|b_{k,m+1}| \leq |b_{k-1,m+1}|/2\) no reduction occurs, unless both tail entries are zero. (In this case we have a reduction in the traditional LLL sense.) Observe that the first and second case in the definition of \(\mu_{k,k-1}\) can be merged together. Thus we need only to test if
Take

Before the size reduction is done, we need to satisfy for that vector the so-called Lovász condition \((C2)\). Consider \((C2)\) when \(\gamma\) tends to infinity. As in the case of partial size reduction we need to distinguish three possibilities, depending if the tail entries are or are not equal to zero. (One option, where both tail entries in the orthogonal projections of \(b_k\) and \(b_{k-1}\) are nonzero, as proved below, is impossible.)

If \(b_{k-1,m+1}^*\) and \(b_{k,m+1}^*\) are zero, condition \((C2)\) becomes the standard Lovász condition, and the rows remain unchanged if the \(k\)th row is \((\alpha - \mu_{k,k-1}^2)\) times longer than the \((k-1)\)th row.

In the second instance, if \(b_{k-1,m+1}^* = 0\), while \(b_{k,m+1}^* \neq 0\), the \(k\)th vector, with respect to the weighted norm, is infinitely longer than the preceding one, hence no exchange will occur.

Assume now that \(b_{k-1,m+1}^* \neq 0\) and \(b_{k,m+1}^* \neq 0\). The orthogonal projection of \(b_k\) is computed as

\[
\hat{b}_k^* = b_k - \sum_{i=1}^{k-2} \mu_{k,i} b_i^* - \mu_{k,k-1} b_{k-1}^*.
\]

By Lemma 3.1, we know that if \(b_{k-1,m+1}^* \neq 0\) then it is the only such vector, i.e., \(b_{i,m+1}^* = 0\), for \(i = 1, \ldots, k-2\). By the above presented argument \(\mu_{k,k-1} = b_{k,m+1}/b_{k-1,m+1}^*\) and hence \(b_{k,m+1}^* = 0\). Consequently, the length of \(b_k^*\) is infinitely shorter than the length of \(b_{k-1}^*\), condition \((C2)\) is not satisfied, and rows \(k\) and \(k-1\) are exchanged.

Once condition \((C2)\) is satisfied for some \(k > 1\), we need to complete the size reduction phase for the \(k\)th vector, so that \(|\mu_{k,j}| \leq \frac{1}{2}\), for all \(j < k\). This is done along analogous lines as described for the partial size reduction.

To speed up the algorithm, we may use the observation that it always starts by computing the greatest common divisor of \(d_1\) and \(d_2\) and a set of minimal multipliers. Thus we may initially compute \(g_2 = \gcd(d_1,d_2)\) and \(x_1d_1 + x_2d_2 = g_2\), with \(x_1\) and \(x_2\) being a definitely least solution (cf. [15]). Also, because of the above discussed nature of the weighted norm, we need to compute Gram-Schmidt coefficients only for the \(n \times n\) submatrix \(b\), with the last column ignored. The complete algorithm is specified in Fig. 1, with some auxiliary procedures given in Fig. 2. We use a natural pseudocode which is consistent with the \texttt{GAP} language ([18]), which provided our first develop-
if $m = 1$ then
  return $[1]$;
fi;

$(g_2, x_1, x_2) := \text{ExtendedGcd}(d_1, d_2);$  
if $m = 2$ then
  return $[x_1, x_2]$;
fi;

$b := \begin{bmatrix}
  d_2/g_2 & -d_1/g_2 & 0 & \cdots & 0 \\
  x_1 & x_2 & 0 & 0 & \cdots \\
  0 & 0 & 1 & 0 & \cdots \\
  \vdots & \vdots & \ddots & \vdots & \vdots \\
  0 & 0 & 0 & \cdots & 1
\end{bmatrix};$

$k_{\text{max}} := 2; \quad k := k_{\text{max}} + 1;$

$d_2 := g_2; \quad d_1 := 0;$

$\mu_{2,1} := (b_1 \cdot b_2) / (b_1 \cdot b_1);$  
$b_1^* := b_1; \quad b_2^* := b_2 - \mu_{2,1} b_1^*;$

repeat
  if $k > k_{\text{max}}$ then
    $k_{\text{max}} := k;$
    $b_k^* := b_{k-1} \cdot b_{k-1}$,
    $j = 1, 2, \ldots, k - 1;$
    $b_k^* := b_k - \sum_{j=1}^{k-1} \mu_{k,j} b_j^*;$
  fi;
  do
    $\text{Reduce}(k, k - 1);$  
    exit if $|d_{k-1}| < |d_k|$ or $|d_{k-1}| = |d_k|$ and $b_k^* \cdot b_k^* \geq (\alpha - \mu_{k,k-1})b_{k-1}^* \cdot b_{k-1}^*$
    $\text{Swap}(k);$  
    $k := \max(2, k - 1);$  
  od;
  for $i \in \{k - 2, k - 3, \ldots, 1\}$ do
    $\text{Reduce}(k, i);$  
  od;
  $k := k + 1;$
until $k > m;$
if $d_m > 0$
  then return $b_m$;
else return $-b_m$;
fi;

Figure 1: The LLL based gcd algorithm
ment environment. Analogous algorithms have also been implemented in Magma ([2]).

\begin{verbatim}
procedure Reduce(k, i)
    if d_i \neq 0 then
        q := \lfloor \frac{d_k}{d_i} \rfloor;
    else
        q := \lfloor \mu_{k,i} \rfloor;
    fi;
    if q \neq 0 then
        b_k := b_k - qb_i;
        \mu_{k,i} := \mu_{k,i} - q;
    for j \in \{1..i-1\} do
        \mu_{k,j} := \mu_{k,j} - q\mu_{i,j};
    od;
fi;
end;

procedure Swap(k)
    \mu_{k-1} \leftarrow \mu_{k,k-1};
    b_k \leftarrow b_{k-1};
    for j \in \{1..k-2\} do
        \mu_{k,j} \leftarrow \mu_{k-1,j};
    od;
    \mu := \mu_{k,k-1};
    B := b_k^* \cdot b_k^* + \mu^2 b_{k-1}^* \cdot b_{k-1}^*;
    b^* := b_k^*;
    b_k^* := \frac{b_k^*}{B} b_{k-1}^* - \mu \frac{b_{k-1}^*}{B} b_k^*;
    b_{k-1}^* := b^* + \mu b_{k-1}^*;
    \mu_{k,k-1} := \frac{\mu}{B} b_{k-1}^*;
    for i \in \{k+1..k_{\text{max}}\} do
        \nu := \mu_{i,k};
        \mu_{i,k} := \mu_{i,k-1} - \nu\mu;
        \mu_{i,k-1} := \nu + \mu_{k-1,k} \mu_{i,k};
    od;
end;
\end{verbatim}

Figure 2: Algorithms \textit{Reduce} and \textit{Swap}

Experimental evidence suggests that sorting the sequence \(d_i\) in increasing order results in the fastest algorithm and tends to give better multipliers than for unsorted or reverse sorted orderings.

The theoretical analysis of the LLL algorithm with \(\alpha = 3/4\) leads to an \(O(m^4 \log \max \{d_j\})\) complexity bound for the time taken by this algorithm to perform the extended gcd calculation. We can also obtain bounds on the quality of the multipliers, but these bounds seem unduly pessimistic in comparison with practical results, as is observed in other LLL applications (cf. [17]).
4 A sorting gcd approach

The presented LLL variation is very successful in obtaining small multipliers. However the quartic complexity (in terms of the number of numbers) of such an approach may be unjustified for those applications that can accept somewhat worse solutions. This is one of the reasons why we have developed a faster heuristic procedure, which we call a sorting gcd algorithm. (In fact it is quite similar to a method due to Brun [4, section 3H].)

The algorithm starts by setting up an array $b$, similar to the array $b$ created in the previous section.

$$b = \begin{bmatrix} d_1 & 1 & \cdots & 0 \\ \vdots & \ddots & \ddots \\ d_m & 0 & \cdots & 1 \end{bmatrix}$$

At each step the algorithm selects two rows, $i$ and $j$, and subtracts row $i$ from row $j$, some $q$ times. Naturally, in the pessimistic case

$$\max_k \{ |b_{j,k} - qb_{i,k}| \} = \max_k \{|b_{j,k}|\} + |q| \max_k \{|b_{i,k}|\}.$$ 

Once rows $i$ and $j$ are selected, we cannot change any individual element of either row $i$ or $j$. Thus in order to minimize the damage done by a single row operation we need to minimize $q$. In the standard Euclidean algorithm, $q$ is always selected as large as possible. Artificially minimizing $q$ by (say) selecting $q$ to be some predefined constant may cause unacceptably long running times for the algorithm. However, by clever choice of $i$ and $j$, which minimizes $q = |b_{j,1}/b_{i,1}|$, we obtain a polynomial time algorithm, which is quite successful in yielding “good” solutions to the extended gcd problem. Clearly, to minimize $|b_{j,1}/b_{i,1}|$, we need to select $i$ so that $b_{i,1}$ is the largest number not exceeding $b_{j,1}$. This is the central idea of the sorting gcd algorithm. In order to make the algorithm precise, we decide that the row with the largest leading element is always selected as the $j$th row. Throughout the algorithm, we operate on positive numbers only, i.e., we assume $b_{i,1} \geq 0$, for $i = 1, 2, \ldots, m$, and for two rows $i$ and $j$ we compute the largest $q$ such that $0 \leq b_{j,1} - qb_{i,1} \leq b_{i,1} - 1$. Furthermore, we observe that we can obtain additional benefits if we spread the multiplier $q$ over several rows. This may be done whenever there is more than one row with
the same leading entry as row \( i \). Also in such cases, the decision as to which of the identical rows to use first should be based on some norm of these rows. The distribution of the multiplier \( q \) should be then biased towards rows with smaller norm. (A discussion of relevant norms for this and related contexts appears in [7].)

An efficient implementation of the method may use a binary heap, with each row represented by a key comprising the leading element and the norm of that row with the leading element deleted, i.e., \( \langle b_{i,1}, ||b_{i,2}, b_{i,3}, \ldots, b_{i,n+1}|| \rangle \), plus an index to it. Retrieving rows \( i \) and \( j \) takes then at most \( O(\log m) \) time, as does reinserting them back to the heap (which occurs only if the leading element is not zero). Row \( j \) cannot be operated upon more than \( \log d_j \) times and hence the complexity of the retrieval and insertion operations for all \( m \) rows is \( O(m \log m \log \max_j \{d_j\}) \). On top of that we have the cost of subtracting two rows, repeated at most \( \log d_j \) times, for each of \( m \) rows. Thus the overall complexity of the algorithm is \( O(m^2 \log \max_j \{d_j\}) \). Subtle changes, like spreading the multiplier \( q \) across several rows with identical leading entry, do not incur any additional time penalties in the asymptotic sense.

The following lemma gives us a rough estimate of the values that \( q \) may take. Most of the time \( q = O(1) \), if \( m \geq 3 \).

**Lemma 4.1** Assume we are given a positive integer \( d_1 \). The average value of \( Q = \left\lfloor \frac{d_1}{\max_{i \geq 2} \{d_i\}} \right\rfloor \), where \( \langle d_2, \ldots, d_m \rangle \) is a sequence of \( m-1 \) randomly generated positive integers less than \( d_1 \), is

\[
Q = \begin{cases} 
\Psi(d_1 + 2) + \gamma - 1 & \text{for } m = 2 \\
\zeta(m - 1) + O\left(\frac{m-1}{(m-2)d_1^{m-2}}\right) & \text{for } m \geq 3
\end{cases}
\]

where \( \Psi(x) \sim \ln(x) - \frac{1}{2x} - O(x^{-2}) \) and \( \zeta(k) = \sum_{i=1}^{\infty} i^{-k} \).

**Outline of Proof.** For the expression \( \lfloor d_1/\max_i \{d_i\} \rfloor \) to be equal to some \( q \in \{1, \ldots, d_1\} \), we must have at least one \( d_i \) falling into the interval \( (d_1/(q+1), \ldots, d_1/q] \) and no \( d_i \) can be larger than \( d_1/q \). Hence the probability of obtaining such a quotient \( q \) is equal to \( (1/q)^{m-1} - (1/(q+1))^{m-1} \). The average value of \( q \) is obtained by calculating \( \sum_{q=1}^{d_1} q[(1/q)^{m-1} - (1/(q+1))^{m-1}] \). \( \square \)
5 Some examples

We have applied the methods described here to numerous examples, all with excellent performance. Note that there are many papers over the years which study explicit input sets and quite a number of these are listed in the references of [4] and [15]. We illustrate algorithm performance with a small selection of examples.

Note also that there are many parameters which can affect the performance of LLL lattice basis reduction algorithms (also observed by many others, including [17]). First and foremost is the value of $\alpha$. Smaller values of $\alpha$ tend to give faster execution times but worse multipliers, however this is by no means uniform. Also, the order of input may have an effect, as mentioned before.

(a) Take $d_1, d_2, d_3, d_4$ to be 116085838, 181081878, 314252913, 10346840.

Following the first method described in some detail, we see that the Kannan–Bachem algorithm gives a unimodular matrix $P$ satisfying $PA = [1, 0, 0, 0]^T$:

\[
P = \begin{bmatrix}
2251284449726056 & -1443226924799280 & 1 & 0 \\
4502568913779963 & -2886453858783690 & 2 & 0 \\
74123990920420 & -47518535244600 & 0 & 1 \\
-90540939 & 58042919 & 0 & 0
\end{bmatrix}.
\]

Applying LLL with $\alpha = 3/4$ to the last three rows of $P$ gives

\[
\begin{bmatrix}
2251284449726056 & -1443226924799280 & 1 & 0 \\
103 & -146 & 58 & -362 \\
-603 & 13 & 220 & -144 \\
-15 & 1208 & -678 & -381
\end{bmatrix}.
\]

Continuing the reduction with no row interchanges, shortens row 1:

\[
\begin{bmatrix}
-88 & 352 & -167 & -101 \\
103 & -146 & 58 & -362 \\
-603 & 13 & 220 & -144 \\
-15 & 1208 & -678 & -381
\end{bmatrix}.
\]

The multiplier vector $-88, 352, -167, -101$ is the unique multiplier vector of least length, as is shown by the Fincke-Pohst method. In fact LLL-based methods give this optimal multiplier vector for all $\alpha \in [1/4, 1]$. 

13
Earlier algorithms which aim to improve on the multipliers do not fare particularly well. Blankinship’s algorithm ([1]) gives the multiplier vector 0, 355043097104056, 1, −6213672077130712. The algorithm due to Bradley ([3]) gives 27237259, −17460943, 1, 0. This shows that Bradley’s definition of minimal is not useful.

The gcd tree algorithm of [15] meets its theoretical guarantees and gives 0, 0, 6177777, −187630660. (This algorithm was not designed for practical use.)

The sorting gcd algorithm using the Euclidean norm gives −5412, 3428, 881, −26032 while it gives 3381, −49813, 27569, −3469 with the max norm. Another sorting variant gives −485, −272, 279, 1728, all of which are reasonable results in view of the greater speed.

There are a number of interesting comparisons and conclusions to be drawn here. First, the multipliers revealed by the first row of $P$ are the same as those which are obtained if we use the natural recursive approach to extended gcd calculation. Viewed as an extended gcd algorithm, the Kannan-Bachem algorithm is a variant of the recursive gcd and the multipliers it produces are quite poor compared to the optimal multipliers. This indicates that there is substantial scope for improving on the Kannan-Bachem method not only for extended gcd calculation but also for Hermite normal form calculation in general. Better polynomial time algorithms for both of these computations are a consequence of these observations. Thus, the gcd-driven Hermite normal form algorithms described in [9] give similar multipliers to those produced by the sorting gcd algorithm on which they are based.

(b) Take $d_1, \ldots, d_{10}$ to be 763836, 1066557, 113192, 1785102, 1470060, 3077752, 114793, 3126753, 1997137, 2603018.

The LLL-based method of section 3 gives the following multiplier vectors for various values of $\alpha$. We also give the length—squared for each vector.

| $\alpha$ | multiplier vector $x$ | $||x||^2$ |
|----------|-----------------------|-----------|
| 1/4      | 7, −1, −5, −1, −1, 0, −4, 0, 0, 0 | 93        |
| 1/3      | −1, 0, 6, −1, −1, 1, 0, 2, −3, 0 | 53        |
| 1/2      | −3, 0, 3, 0, −1, 1, 0, 1, −4, 2 | 41        |
| 2/3      | 1, −3, 2, −1, 5, 0, 1, 1, −2, −1 | 47        |
| 3/4      | 1, −3, 2, −1, 5, 0, 1, 1, −2, −1 | 47        |
| 1        | −1, 0, 1, −3, 1, 3, 3, −2, −2, 2 | 42        |
The Fincke-Pohst algorithm reveals that the unique shortest solution is

\[3, -1, 1, 2, -1, -2, -2, 2, 2\]

which has length-squared 36. The sorting gcd algorithm using the Euclidean norm gives

\[-2, -3, -6, 2, 9, 0, 2, 0, 2, -6\]

with length-squared 178 while with the max norm it gives

\[-9, 5, 2, -7, 2, 3, 0, -3, -1, 5\]

which has length-squared 207.

The recursive gcd gives multipliers

\[1936732230, -1387029291, -1, 0, 0, 0, 0, 0, 0, 0\].

The Kannan-Bachem algorithm gives

\[44537655090, -31896527153, 0, 0, 0, 0, 0, 0, -1\].

Blankinship’s algorithm gives the multiplier vector

\[3485238369, 1, -23518892995, 0, 0, 0, 0, 0, 0\]

while Bradley’s algorithm gives

\[-135282, 96885, -1, 0, 0, 0, 0, 0, 0, 0\].

(c) The following example has theoretical significance. Take \(d_1, \ldots, d_m\) to be the Fibonacci numbers

(i) \(F_n, F_{n+1}, \ldots, F_{2n}, \) \(n\) odd, \(n \geq 5\);

(ii) \(F_n, F_{n+1}, \ldots, F_{2n-1}, \) \(n\) even, \(n \geq 4\).

Using the identity \(F_mL_n = F_{m+n} + (-1)^nF_{m-n}\), it can be shown that the following are multipliers:

(i) \(-L_{n-3}, L_{n-4}, \ldots, -L_2, L_1, -1, 1, 0, 0, \) \(n\) odd;

(ii) \(L_{n-3}, -L_{n-4}, \ldots, -L_2, (L_1 + 1), -1, 0, 0, \) \(n\) even,

where \(L_1, L_2, \ldots\) denote the Lucas numbers 1, 3, 4, 7, \ldots.

In fact we have the identities:
\[
\begin{bmatrix}
1 & 1 & -1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & -1 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & -1 & \cdots & 0 & 0 & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 1 & 1 & -1 & 0 \\
L_{n-1} & L_{n-2} & \cdots & \cdots & L_4 & -L_3 & L_2 & -L_1 & 1 & -1 \\
-L_{n-3} & L_{n-4} & \cdots & \cdots & -L_2 & L_1 & -1 & 1 & 0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
F_n \\
F_{n+1} \\
\vdots \\
F_{n-3} \\
F_{n-2} \\
F_{n-1} \\
F_{2n} \\
\end{bmatrix}
= 
\begin{bmatrix}
0 \\
0 \\
\vdots \\
0 \\
0 \\
0 \\
1 \\
\end{bmatrix}
\tag{3}
\]

if \( n \) is odd and \( n \geq 5 \).

\[
\begin{bmatrix}
1 & 1 & -1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & -1 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & -1 & \cdots & 0 & 0 & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 1 & 1 & -1 & 0 \\
L_{n-1} & L_{n-2} & \cdots & \cdots & L_4 & -L_3 & L_2 & -L_1 & (L_1 + 1) & 1 \\
-L_{n-3} & L_{n-4} & \cdots & \cdots & -L_2 & (L_1 + 1) & -1 & 0 & 0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
F_n \\
F_{n+1} \\
\vdots \\
F_{n-4} \\
F_{n-3} \\
F_{n-2} \\
F_{2n-1} \\
\end{bmatrix}
= 
\begin{bmatrix}
0 \\
0 \\
\vdots \\
0 \\
0 \\
0 \\
1 \\
\end{bmatrix}
\tag{4}
\]

if \( n \) is even and \( n \geq 4 \).

The square matrices are unimodular, as their inverses are the following matrices, respectively:

\[
\begin{bmatrix}
F_3 & F_2 & F_2 & F_3 & F_4 & \cdots & F_{n-3} & 0 & -F_{n-2} & F_{n-2} & F_n \\
F_4 & F_3 & F_3 & F_4 & F_5 & \cdots & F_{n-2} & 0 & -F_{n-1} & F_{n-1} & F_{n+1} \\
F_5 - F_3 & F_4 & F_4 & F_5 & F_6 & \cdots & F_{n-1} & 0 & -F_3 & F_3 & F_n + 2 \\
F_n - F_2 & F_3 - F_1 & F_4 & F_5 & \cdots & F_n & 0 & -F_{n-1} & F_{n+1} & F_{n+3} \\
F_2 - F_3 & F_5 - F_2 & F_6 & F_7 & \cdots & F_{n+1} & 0 & -F_{n-2} & F_{n+2} & F_{n+4} \\
F_2 - F_3 & F_5 - F_2 & F_6 - F_3 & F_7 & \cdots & F_{n+2} & 0 & -F_{n-3} & F_{n+3} & F_{n+5} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\
F_{n+1} - F_{n-3} & F_n - F_{n-4} & F_{n+3} - F_{n-5} & F_{n+2} - F_{n-6} & \cdots & F_{2n-5} - F_3 & 0 & -F_{2n-4} & F_{2n-4} & F_{2n-2} \\
F_{n+2} - F_{n-2} & F_{n+1} - F_{n-3} & F_{n+4} - F_{n-4} & F_{n+3} - F_{n-5} & \cdots & F_{2n-4} - F_2 & -1 & -F_{2n-3} & F_{2n-3} & F_{2n-1} \\
F_{n+3} - F_{n-1} & F_{n+2} - F_{n-2} & F_{n+5} - F_{n-3} & F_{n+4} - F_{n-4} & \cdots & F_{2n-3} - F_3 & -1 & -F_{2n-2} & F_{2n-2} & F_{2n} \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
F_3 & F_2 & F_2 & F_3 & F_4 & \cdots & F_{n-4} & -F_{n-2} & F_{n-2} & F_n \\
F_4 & F_3 & F_3 & F_4 & F_5 & \cdots & F_{n-3} & -F_{n-1} & F_{n-1} & F_{n+1} \\
F_5 - F_3 & F_4 & F_4 & F_5 & F_6 & \cdots & F_{n-2} & -F_3 & F_3 & F_n \ + 2 \\
F_n - F_2 & F_3 - F_1 & F_4 & F_5 & \cdots & F_n & -F_{n-1} & F_{n+1} & F_{n+3} \\
F_2 - F_3 & F_5 - F_2 & F_6 & F_7 & \cdots & F_{n+1} & -F_{n-2} & F_{n+2} & F_{n+4} \\
F_2 - F_3 & F_5 - F_2 & F_6 - F_3 & F_7 & \cdots & F_{n+2} & -F_{n-3} & F_{n+3} & F_{n+5} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\
F_{n+1} - F_{n-3} & F_n - F_{n-4} & F_{n+3} - F_{n-5} & F_{n+2} - F_{n-6} & \cdots & F_{2n-5} - F_3 & -1 & -F_{2n-4} & F_{2n-4} & F_{2n-2} \\
F_{n+2} - F_{n-2} & F_{n+1} - F_{n-3} & F_{n+4} - F_{n-4} & F_{n+3} - F_{n-5} & \cdots & F_{2n-4} - F_2 & -1 & -F_{2n-3} & F_{2n-3} & F_{2n-1} \\
F_{n+3} - F_{n-1} & F_{n+2} - F_{n-2} & F_{n+5} - F_{n-3} & F_{n+4} - F_{n-4} & \cdots & F_{2n-3} - F_3 & -1 & -F_{2n-2} & F_{2n-2} & F_{2n} \\
\end{bmatrix}
\]

It is not difficult to prove that the multipliers given here are the unique vectors of least length (by completing the appropriate squares and using various Fibonacci and Lucas number identities, see [10]). The length-squared of the multipliers is \( L_{2n-5} + 1 \) in both cases. (In practice, the LLL-based algorithms compute these minimal multipliers.)
These results give lower bounds for extended gcd multipliers in terms of Euclidean norms. Since, with $\phi = \frac{1+\sqrt{5}}{2}$,

$$L_{2n-5} + 1 \sim \phi^{2n-5} \sim \phi^{-5} \sqrt{5} F_{2n}$$

it follows that a general lower bound for the Euclidean norm of the multiplier vector in terms of the initial numbers $d_i$ must be at least $O(\sqrt{\max\{d_i\}})$. Also, the length of the vector

$$F_n, F_{n+1}, \ldots, F_{2n}$$

is of the same order of magnitude as $F_{2n}$, so a general lower bound for the length of the multipliers in terms of the Euclidean length of the input, $l$ say, is $O(\sqrt{l})$.

6 Conclusions

We have described new algorithms for extended gcd calculation which provide good multipliers. We have given examples of their performance and indicated how the algorithms may be fully analyzed. Related algorithms which compute canonical forms of matrices will be the subject of another paper.

Acknowledgments

The first two authors were supported by the Australian Research Council.

References


