MATHEMATICS OF COMPUTATION
Volume 67, Number 223, July 1998, Pages 1149–1161
S 0025-5718(98)00936-3

A NATURAL LATTICE BASIS PROBLEM WITH APPLICATIONS

JOHN D. HOBBY

ABSTRACT. Integer lattices have numerous important applications, but some of them may have been overlooked because of the common assumption that a lattice basis is part of the problem instance. This paper gives an application that requires finding a basis for a lattice defined in terms of linear constraints. We show how to find such a basis efficiently.

1. INTRODUCTION

Numerous application problems involve trying to minimize a quadratic function with integer variables. The problem of finding an integer vector x that minimizes

$$\|A(x - x_0)\|_2^2 \tag{1.1}$$

for a matrix A and a vector x_0 is known as the *nearest lattice point problem.* The possible values of Ax form a lattice and the columns of A are a basis for the lattice. The nearest lattice point problem is NP-complete, but approximate solutions can be found with Lenstra, Lenstra, Lovász (LLL) basis reduction [12] and Babai's nearest plane algorithm [1].

Lenstra [13] uses the algorithms from [1, 12] for integer programming; Shamir [17], Odlyzko [14], Hastad [9] use them for cryptographic applications; Sugihara [18] uses them for representing geometrical objects; Hobby [10] uses them for font generation. Other applications appear in Frieze et al. [5], Grosse and Hobby [7], and Hastad et al. [8].

We shall see that at least one of these applications benefits from the generalization where we allow linear constraints on the integer vector x. It is safe to assume that the equations have rational coefficients and that we can find a basis for the solution space over the rationals. If x has dimension n, this allows x to be expressed as By where B is an $n \times m$ matrix over the rationals with dimensions $m < n$. Unfortunately, this cannot be substituted into (1.1) because an integer vector y might generate a non-integer x vector and a non-integer y might give an integer x. If $\mathbb{Z}$ is the set of integers, we need a matrix B such that

Received by the editor January 13, 1995 and, in revised form, May 1, 1996 and January 10, 1997.

1991 *Mathematics Subject Classification.* Primary 11H55; Secondary 52C07, 68U15.

Key words and phrases. Integer lattices; lattice basis; grid-fitting; outline fonts.

$$\{ By \mid y \in \mathbb{Z}^m \} \tag{1.2}$$

is exactly the subset of $\mathbb{Z}^n$ that satisfies the constraints on x.

Since the set of $\mathbb{Z}^n$ that obeys the constraints on x is closed under addition and subtraction, it is a lattice. Requiring (1.2) to match the lattice just says that the columns of B are a basis for the lattice. Thus the problem can be restated as follows: given a basis for a vector space in $\mathbb{Q}^n$, find a basis for the lattice formed by intersecting that vector space with $\mathbb{Z}^n$. An alternative formulation of this problem appears in Cohen's book [4, Algorithm 2.7.2]. The algorithm presented there is based on Pohst's modification of LLL basis reduction, so we refer to it as the *MLLL* algorithm [15]. Similar algorithms have appeared elsewhere [3]. The MLLL algorithm has some advantages, but experiments show that it is very ill-suited to the application given below.

The rest of this paper gives a practical application for this problem and explains how to find the desired basis. Section 2 explains the application, and Section 3 gives the algorithm. Section 4 gives some performance statistics and compares the algorithm to the MLLL algorithm, and Section 5 gives some concluding remarks. The time for an m-dimensional subspace of $\mathbb{Q}^n$ is at most $O(m^3 n)$ times a term that depends on the size of the integers needed to represent the input basis.

2. Applications

We have already referred to applications where it is useful to find a vector $x \in \mathbb{Z}^n$ that approximately minimizes the quadratic function (1.1). This section shows how one of them benefits from the ability to impose linear constraints on the vector x.

The application described in [10] deals with adjusting outline fonts so that they can be converted to bitmap form without introducing undesirable artifacts. This is important because printers and computer terminals require bitmaps, while character shapes are most naturally described via outlines. Virtually every font supplier uses some "hinting" or "grid-fitting" strategy to cope with the fact that naive scan-conversion does not generate acceptable bitmap images.

The quadratic function minimized in [10] is a heuristic estimate of the distortion in important features when a given set of outlines is scan-converted. The main variables are the coordinates of control points that describe the outlines. Additional integer-valued variables are introduced so that the distortion function can depend on how key features relate to the pixel grid. It is these integer variables that form the vector x when the distortion function is reduced to the form of (1.1).

The paper [10] proceeds by finding certain features of the original outlines and deciding what relationships need to hold in order for the features to be preserved after scan-conversion. Each new relation to be preserved generates a new row of a matrix which we can call $\bar{A}$. The distortion function is the squared 2-norm of $\bar{A}(\bar{x} - \bar{x}_0)$, where $\bar{x}$ gives the variables over which to minimize and $\bar{x}_0$ gives their ideal values. The next step is to find the QR-factorization $\bar{A} = QR$ where Q is an orthogonal matrix and R is upper-triangular. This reduces the distortion function to the form

$$\|R(\bar{x} - \bar{x}_0)\|_2^2 .$$

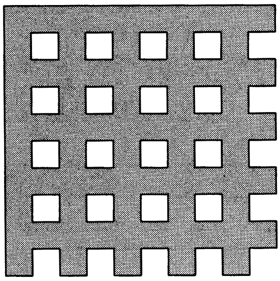

FIGURE 1. A character shape where the distortion function is much easier to minimize when linear constraints are used.

Dividing $\bar{x}$ into a vector w of control point coordinates and a vector x of integer variables gives the following block structure:

$$\left\| \begin{pmatrix} C & B \\ 0 & A \end{pmatrix} \left(\begin{pmatrix} w \\ x \end{pmatrix} - \begin{pmatrix} w_0 \\ x_0 \end{pmatrix} \right) \right\|_2^2 \tag{2.1}$$

Since w can be chosen to make

$$C(w - w_0) + B(x - x_0) = 0,$$

minimizing (2) is equivalent to minimizing (1.1).

There are no provisions in [10] for linear constraints on x, but they easily fit into the framework outlined above. When a relation is very important, [10] multiplies the corresponding row of $\bar{A}$ by a large constant. Some of these relations involve only integer variables. Moving the corresponding rows of $\bar{A}$ to a separate constraint matrix effectively makes them "infinitely important." This reduces the danger of excessive numerical error and reduces the number of variables.

When separate constraints on the x vector were added to an implementation of [10], several instances were found where the number of variables was greatly reduced. This occurred when processing complicated, repetitive character shapes such as that shown in Figure 1. In this instance, the x vector originally had 452 components, but the linear constraints allowed x to be replaced by By where B is a 452×66 matrix. Thus $\|A(x - x_0)\|^2$ is replaced by $\|AB(y - y_0)\|^2$, where y has only 66 components. This greatly speeds up LLL basis reduction and Babai's nearest plane algorithm. It requires the columns of B to be a basis for the lattice formed by intersecting the column span of B with $\mathbb{Z}^{452}$, but that is the subject of Section 3.

The character shape shown in Figure 1 is rather unusual, but it did come up while processing an actual font. In fact, 216 of 1664 characters processed benefited from linear constraints. In one case, the number of integer variables was reduced from 7295 to 208. See Bigelow and Holmes [2] for information about the font.

3. THE ALGORITHM

Given a basis for a vector subspace of $\mathbb{Q}^n$, how do we find a basis for the lattice formed by intersecting with $\mathbb{Z}^n$? Begin by assuming that the given basis vectors

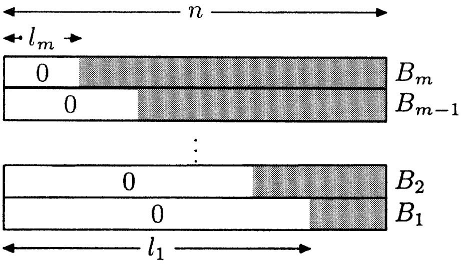

FIGURE 2. The pattern of leading zeros in the basis vectors B_1, $B_2, \ldots, B_m$.

$B_1, B_2, \ldots, B_m$ are scaled so that they are in $\mathbb{Z}^n$ and that the basis vectors are in a triangular form as shown in Figure 2. In other words, if l_i is the number of initial zeros at the beginning of the ith basis vector B_i, we assume l_i decreases with increasing i. (This can easily be obtained by Gaussian elimination.)

The goal is then to alter the basis vectors so that they span the same vector space, they still lie in $\mathbb{Z}^n$, and they have the *integer combination property.* A set of vectors in $\mathbb{Z}^n$ has this property if any linear combination of the vectors that involves a non-integer coefficient must yield a non-integer result.

Proceeding inductively, assume that basis vectors $B_1, B_2, \ldots, B_{\bar{m}-1}$ have the integer combination property and we want to find a vector $B'_{\bar{m}} \in \mathbb{Z}^n$ so that B_1, B_2, $\ldots, B_{\bar{m}-1}, B'_{\bar{m}}$ have the integer combination property and span the same subspace of $\mathbb{Q}^n$ as do $B_1, B_2, \ldots, B_{\bar{m}-1}, B_{\bar{m}}$.

This requires finding the largest integer q such that there is a vector

$$\beta_1 B_1 + \beta_2 B_2 + \cdots + \beta_{\bar{m}-1} B_{\bar{m}-1} + B_{\bar{m}}/q \tag{3.1}$$

in $\mathbb{Z}^n$, where $\beta_1, \beta_2 \ldots, \beta_{\bar{m}-1}$ are rational numbers. (The triangular form illustrated in Figure 2 guarantees that there is a maximal q. Indeed, q must divide the leading coefficient of $B_{\bar{m}}$.) The following lemma shows that we can use (3.1) for $B'_{\bar{m}}$.

Lemma 3.1. *Suppose basis vectors $B_1, B_2, \ldots, B_m$ are $\mathbb{Z}^n$ vectors in the triangular form of Figure 2, and the integer combination property holds for $B_1, B_2, \ldots,$ $B_{\bar{m}-1}$, where $\bar{m} \le m$ and q is the largest integer such that there is an integer vector (3.1). Let this vector be $B'_{\bar{m}}$. Then the integer combination property holds for B_1, $B_2, \ldots, B_{\bar{m}-1}, B'_{\bar{m}}$.*

Proof. Suppose the integer combination property fails. Then there is a vector $b_1 B_1 + b_2 B_2 + \cdots + b_{\bar{m}-1} B_{\bar{m}-1} + b_{\bar{m}} B'_{\bar{m}} \in \mathbb{Z}^n$, where some $b_i \notin \mathbb{Z}$. If $b_{\bar{m}} \in \mathbb{Z}$, then $b_1 B_1 + b_2 B_2 + \cdots + b_{\bar{m}-1} B_{\bar{m}-1} \in \mathbb{Z}^n$ and the property fails for $B_1, B_2, \ldots, B_{\bar{m}-1}$, contradicting the assumption.

If $b_{\bar{m}} \notin \mathbb{Z}$, express it as q_1/q_2 where $\gcd(q_1, q_2) = 1$ and $q_2 > 1$. Then there exist $a, b \in \mathbb{Z}$ where $aq_1 + bq_2 = 1$. Multiplying

$$b_1 B_1 + b_2 B_2 + \cdots + b_{\bar{m}-1} B_{\bar{m}-1} + b_{\bar{m}} B'_{\bar{m}}$$

by a and adding $bB'_{\bar{m}}$ yields a vector in $\mathbb{Z}^n$. This vector is

$$\begin{aligned}\left(\sum_{i<\bar{m}} ab_i B_i\right) + \left(\frac{aq_1 + bq_2}{q_2}\right) B'_{\bar{m}} &= \left(\sum_{i<\bar{m}} ab_i B_i\right) + \frac{B'_{\bar{m}}}{q_2}\\ &= \left(\sum_{i<\bar{m}} \left(ab_i + \frac{\beta_i}{q_2}\right) B_i\right) + \frac{B_{\bar{m}}}{qq_2},\end{aligned}$$

contradicting the assumption that q is as large as possible. □

Section 3.1 shows how to perform the induction step so that Lemma 3.1 applies. Then Section 3.2 analyzes the resulting algorithm in a way designed to suggest how actual performance might be much better than the worst case bounds.

3.1. Updating the basis.

Given vectors $B_1, B_2, \ldots, B_{\bar{m}-1}$ in $\mathbb{Z}^n$ that satisfy the integer combination property, we need to preserve the property while appending a $\mathbb{Z}^n$ vector formed by taking a linear combination involving a new vector $B_{\bar{m}}$.

Let $B_{\bar{m}}[j]$ be the jth element of $B_{\bar{m}}$. Since l_i decreases with increasing i, $B_{\bar{m}}[l_{\bar{m}}+1]$ through $B_{\bar{m}}[l_{\bar{m}-1}]$ must be divisible by q in order for (3.1) to be integer-valued. Thus q must divide $g_{\bar{m}}$, where

$$g_{\bar{m}} = \gcd(B_{\bar{m}}[l_{\bar{m}} + 1], B_{\bar{m}}[l_{\bar{m}} + 2], \ldots, B_{\bar{m}}[l_{\bar{m}-1}]). \tag{3.2}$$

Our goal is to find the largest such q for which there is an integer vector of the form given by (3.1). Algorithm 1 does this by repeatedly replacing $B_{\bar{m}}$ by a vector of the form (3.1).

Algorithm 1. Given basis vectors $B_1, B_2, \ldots, B_{\bar{m}}$ in $\mathbb{Z}^n$ and in the triangular form of Figure 2, where $B_1, \ldots, B_{\bar{m}-1}$ satisfy the integer combination property, modify $B_{\bar{m}}$ so that the property holds for $B_1, \ldots, B_{\bar{m}}$.

1. Initialize $p = 2$ and set $g_{\bar{m}}$ according to (3.2).
2. If $g_{\bar{m}} = 1$, stop. Otherwise let p be the smallest remaining prime factor of $g_{\bar{m}}$.
3. Determine whether there exist integers $b_1, b_2, \ldots, b_{\bar{m}-1}$ such that
$$B_{\bar{m}} = b_1 B_1 + b_2 B_2 + \cdots + b_{\bar{m}-1} B_{\bar{m}-1} \pmod{p}.$$
4. If not, remove all factors of p from $g_{\bar{m}}$. Otherwise, replace $g_{\bar{m}}$ by $g_{\bar{m}}/p$ and $B_{\bar{m}}$ by
$$\frac{B_{\bar{m}} - b_1 B_1 - b_2 B_2 - \cdots - b_{\bar{m}-1} B_{\bar{m}-1}}{p}.$$
5. Go to Step 2.

Since the integers modulo p form a field, Step 3 of Algorithm 1 is a simple application of Gaussian elimination. Vectors $B_1, B_2, \ldots, B_{\bar{m}-1}$ are given in the triangular form of Figure 2, but reducing modulo p can create additional zeros. There could be some i, j where $i < j \leq \bar{m} - 1$ and B_j has more leading zeros modulo p than B_i does. Gaussian elimination gets rid of such cases by applying a suitable linear transformation T_p to $B_1, B_2, \ldots, B_{\bar{m}-1}$. If $B_{\bar{m}}$ can be expressed modulo p as a linear combination of the transformed $B_1, B_2, \ldots, B_{\bar{m}-1}$, applying T_p to the coefficients gives $b_1, b_2, \ldots, b_{\bar{m}-1}$.

Lemma 3.2. *Algorithm 1 replaces $B_{\bar{m}}$ with an integer vector of the form (3.1), where $q \in \mathbb{Z}$ is as large as possible.*

Proof. Let $S_{\bar{m}}$ be the set of integers q for which there is an integer vector of the form (3.1). For any $q \in S_{\bar{m}}$, any factor q' of q is in $S_{\bar{m}}$ since multiplying (3.1) by q/q' yields an integer vector where the $B_{\bar{m}}$ denominator is q'.

For any q_1 and q_2 in $S_{\bar{m}}$ with $\gcd(q_1, q_2) = 1$, $B_{\bar{m}}$ can be expressed as $\sum_{i<\bar{m}} c_{1,i}\, B_i$ modulo q_1 and $\sum_{i<\bar{m}} c_{2,i}\, B_i$ modulo q_2, where $c_{j,i}$ are integer coefficients. There clearly exist integers $c_{3,i}$ congruent to $c_{1,i}$ modulo q_1 and $c_{2,i}$ modulo q_2. Thus

$$B_{\bar{m}} = \sum_{i<\bar{m}} c_{3,i} B_i \pmod{q_1 q_2}$$

and $q_1 q_2 \in S_{\bar{m}}$.

It follows that $S_{\bar{m}}$ consists of an integer and all of its factors. The algorithm initializes $g_{\bar{m}}$ so that this maximal q must divide $g_{\bar{m}}$. Suppose we add a "write-only" variable $\bar{q}$ that is initialized to 1 in Step 2 and multiplied by p whenever Step 4 updates $B_{\bar{m}}$. This clearly does not affect the correctness of the algorithm. After each iteration, the current $B_{\bar{m}}$ is $1/\bar{q}$ times the original $B_{\bar{m}}$ plus a rational linear combination of B_1, B_2, ... , $B_{\bar{m}-1}$. Another invariant is that any positive integer q' with $\bar{q}q' \in S_{\bar{m}}$ must be a factor of $g_{\bar{m}}$. This holds because we remove factors from $g_{\bar{m}}$ only if they are being multiplied into $\bar{q}$ or if we determine that $\bar{q}p \notin S_{\bar{m}}$. (The integer combination property guarantees that failure to find integers b_1, b_2, ... , $b_{\bar{m}-1}$ in Step 3 implies that no rational linear combination of B_1, B_2, ... , $B_{\bar{m}-1}$ can equal $B_{\bar{m}}$ modulo p.)

When $g_{\bar{m}}$ reaches $\bar{m}$, the invariants guarantee that an expression of the form (3.1) gives the current $B_{\bar{m}}$ as a function of the original $B_{\bar{m}}$ where $q = \bar{q}$ and $\bar{q}$ is the maximum element of $S_{\bar{m}}$. □

3.2. **Algorithm analysis.** Given basis vectors B_1, B_2, ... , B_m in $\mathbb{Z}^n$ where the number of leading zeros l_i in B_i decreases with increasing i, we need to find a basis that spans the same subspace of $\mathbb{Q}^n$ and has the integer combination property. Lemmas 3.1 and 3.2 show that it suffices to apply Algorithm 1 successively for $\bar{m} = 1, 2, \ldots, m$. Hence, the total run time is the time for Algorithm 1 summed over these values of $\bar{m}$.

If the input vectors have entries bounded by L, then Step 1 requires $O((l_{\bar{m}-1} - l_{\bar{m}}) \log L)$ arithmetic operations.

Step 2 is executed several times, but the net effect is to find the complete prime factorization of the original value of $g_{\bar{m}}$. Since $g_{\bar{m}} \leq L$, trial division can find the factors in $O(\sqrt{L})$ arithmetic operations. More sophisticated factoring algorithms have smaller time bounds, but we shall avoid discussing their complexities and just use F_L for the time to factor $g_{\bar{m}} \leq L$.

Gaussian elimination in Step 3 requires $O(n\bar{m}^2)$ operations for each new prime p, but this can be reduced by taking advantage of previous results and the structure of B_1, B_2, ... , $B_{\bar{m}}$. Before reduction mod p, each B_i has l_i leading zeros where l_i decreases with increasing i. The same holds mod p except that there may be more leading zeros due to entries divisible by p. Hence fewer than i vector operations may suffice for each B_i. In addition, we might already have some number $\bar{\bar{m}}_p$ of initial B_i in reduced form mod p; i.e., if $l_{p,i}$ is the number of leading zeros for B_i modulo p, $l_{p,i}$ decreases for increasing i as long as $i \leq \bar{\bar{m}}_p$. Hence the actual time for Gaussian elimination is

$$O\big(nD_{\bar{m},p}(\bar{m} - \bar{\bar{m}}_p)\big),$$

where $D_{\bar{m},p} \leq \bar{m}$ is the average number of row operations applied to any B_i during Gaussian elimination.

Gaussian elimination needs to be done for each distinct prime factor of $g_{\bar{m}}$, and $O(n\bar{m})$ additional operations are needed to find $b_1, b_2, \ldots, b_{\bar{m}}$ after the reduction is complete. If P_L is a bound on the number of distinct prime factors for $g_{\bar{m}}$ and $\bar{P}_L$ is a similar bound that allows multiplicity, Algorithm 1 does Gaussian elimination P_L times and computes $b_1, b_2, \ldots, b_{\bar{m}}$ a total of $\bar{P}_L$ times. Hence, the total time for Step 3 is

$$O\left(n\left(\bar{m}\bar{P}_L + \sum_{p\in\mathcal{F}(g_{\bar{m}})} D_{\bar{m},p}(\bar{m} - \bar{\bar{m}}_p)\right)\right), \tag{3.3}$$

where $\mathcal{F}(g_m)$ is the set of prime factors of g_m.

The total time for a single execution of Algorithm 1 is (3.3) plus the time for Steps 1, 2, and 4. The $O(n\bar{m}P_L)$ cost of Step 4 is subsumed by (3.3), as is the $O((l_{\bar{m}-1} - l_{\bar{m}})\log L)$ cost of Step 1. Thus the total cost is F_L more than (3.3).

Algorithm 1 is actually executed m times, once for each $\bar{m} \in [1, m]$. The most interesting term is

$$\sum_{\bar{m}=1}^{m} \sum_{p\in\mathcal{F}(g_{\bar{m}})} nD_{\bar{m},p}(\bar{m} - \bar{\bar{m}}_p).$$

Defining $D_{\bar{m},p} = 0$ when p does not divide $g_{\bar{m}}$ and reversing the order of summation gives

$$\sum_{p\in\mathcal{F}(g_1g_2\cdots g_m)} \sum_{\bar{m}=1}^{m} nD_{\bar{m},p}(\bar{m} - \bar{\bar{m}}_p) \leq \sum_{p\in\mathcal{F}(g_1g_2\cdots g_m)} nDm \leq nDmP_{L^m},$$

where D is a weighted average of all $D_{\bar{m},p}$ values and P_{L^m} is a bound on the number of distinct prime factors of $g_1g_2\cdots g_m$. Since the other two terms are $\sum_{\bar{m}} F_l$ and $\sum_{\bar{m}} n\bar{m}\bar{P}_L$, the total is

$$O\big(mF_L + nm(m\bar{P}_L + DP_{L^m})\big). \tag{3.4}$$

We can eliminate F_L, D, $\bar{P}_L$ and P_L from (3.4), but the result may be misleading because the required upper bounds can be rather pessimistic. We have $D \leq m$ and $\bar{P}_L \leq \log_2(L)$. To bound the number P_{L^m} of distinct prime factors less than L^m, let $f(k)$ be the sum of the logarithms of the first k primes:

$$P_{L^m} \leq \max_{f(k)\leq\log(L^m)} k.$$

Since $f(k) = O(k\log k)$, we have

$$P_{L^m} = O\left(\frac{m\log L}{\log m + \log\log L}\right).$$

The factorization time F_L is poorly understood, but using heuristic assumptions to analyze the best known algorithm gives an expected running time of

$$e^{(c+o(1))(\log L)^{1/3}(\log\log L)^{2/3}},$$

where the c is a constant and the limit implicit in the $o(1)$ is for $L \to \infty$ [4]. Substituting these values into (3.4) gives

$$O\left(me^{(c+o(1))(\log L)^{1/3}(\log\log L)^{2/3}} + \frac{nm^3\log L}{\log m}\right) \tag{3.5}$$

arithmetic operations on numbers of magnitude L.

For large L, it is more appropriate to measure the bit operation complexity. This would introduce an additional factor into (3.5) that is theoretically $o(\log^2 L)$.

4. Experimental results

Algorithm 1 was implemented in C++ using double precision floating point and tested on random data and on problem instances from the application of Section 2. Since finding a lattice basis requires m calls to Algorithm 1, we refer to the complete process as Algorithm 1*. Because these problem instances involved sparse matrices, the implementation used sparse matrix techniques. There are 500 lines of code in the main routines and 2000 lines including the test program and the sparse matrix package.

Even though double precision suffices for the font application, the algorithm was also implemented using an infinite-precision integer package due to Gansner [6]. This version used Pollard's factorization algorithm [16], [11] to factor $g_{\bar{m}}$ in those cases where trial division was inappropriate.

For comparison purposes, the MLLL algorithm [4, Algorithm 2.7.2] was also implemented using Gansner's infinite-precision package. Bugs in the published version of the algorithm were corrected by retrieving Cohen's Pari system and examining a routine named `lllall0`. An alternative implementation of the MLLL algorithm used sparse matrix techniques, thereby achieving approximately a factor of 2 speed-up for the results in Table 1.

The font application tended to generate large sparse problem instances. Table 1 summarizes the 9 problems encountered while processing the font from [2]. All the timings were done on a 150 Megaherz MIPS R4400 processor. The timings for the MLLL algorithm do not include the time to convert the input matrix into a matrix that gives a basis for the null space. The input is a matrix A whose rows span the desired subspace of $\mathbb{Q}^n$, and the MLLL algorithm starts with a matrix B whose rows form a basis for the null space of A. (Thus $AB^T = 0$.) The new input matrix B can have many more rows than A does. For instance, the 43×457 matrix for the fifth row of Table 1 results in a 423×457 matrix as the argument to the MLLL algorithm. (The 43×457 matrix had rank 34.)

In general, if the input matrix A is $m \times n$, the null space basis B will be $m' \times n$ where $m' \geq n - m$. If m' were shown in Table 1, it would range from 6 to 20 for the first four rows and it would be close to $n - m$ for the remaining rows. The largest entry in B turns out to be the same as the L value whose logarithm is listed in the table, and the number of nonzeros in B stays within a factor of 2.2 of the number of nonzeros in A. Thus the percentage of nonzeros in B ranges from 7.7% for row 1 to 0.23% for row 6.

Table 1 is incomplete because the MLLL algorithm did not terminate in a reasonable time for the larger problem instances. This is probably due to the large size of the intermediate results. The largest intermediate result $I_{\max}$ is simply the largest integer result encountered for all the integer arithmetic operations during a run of the algorithm. For the last problem where the MLLL algorithm ran to completion, $I_{\max}$ was 1688 bits long. For Algorithm 1* in contrast, the numbers remain very small. They are easily small enough to allow doing all computations in fixed precision using 64-bit floating point.

TABLE 1. Comparitive results of Algorithm 1* and the MLLL algorithm on problems from the font application. The dimensions $m \times n$, the largest entry L and the percentage of nonzeros in the matrix refer to the input for Algorithm 1*. Corresponding input for the MLLL algorithm was derived as explained in the text. The T column gives time in seconds and the T_s and T_d columns give time in seconds for the sparse and dense implementations of the MLLL algorithm. The maximum intermediate result in the MLLL algorithm is $I_{\max}$, and the mF_L and $m\bar{P}_L$ columns give the total trial divisions for factoring all the $g_{\bar{m}}$'s and the total number of divisors found.

				Algorithm 1*			MLLL algorithm		
m	n	$\log_2(L)$	$\% \neq 0$	mF_L	$m\bar{P}_L$	T	$\log_2(I_{\max})$	T_s	T_d
45	26	1	2.2%	19	19	0.0017	16	0.130	0.1
45	26	1	2.2%	19	19	0.0018	16	0.127	0.1
58	37	1	1.7%	17	17	0.0022	72	0.230	0.4
58	43	1	1.7%	24	24	0.0023	64	0.331	0.6
43	457	3.8	3.6%	33	33	0.0037	1688	2197	4450
8	1087	4.2	19%	6	6	0.0169		> 7800	
57	3162	5.1	2.8%	47	47	0.0218			
227	3162	5	0.68%	67	67	0.0230			
7	7315	5.2	22%	5	5	0.1230			

Is Algorithm 1* always so much faster than the MLLL algorithm? To check this, we derive additional test problems from rectangular matrices with random integer entries. The procedure is to take a rectangular matrix A_0, and use Gaussian elimination followed by row scaling to generate an integer matrix A_1 in the triangular form required by Algorithm 1*. (See Figure 2.) Applying Algorithm 1* to the rows of A_1 produces a lattice basis that we can write in the form of a matrix A_2. For instance, if

$$A_0 = \begin{pmatrix} -7 & -2 & 2 & 0 & -10 & 4 \\ 4 & -3 & 8 & -7 & -6 & -6 \\ -3 & -6 & -1 & -11 & 3 & 5 \\ 3 & 10 & 9 & 0 & 6 & -5 \\ 8 & -10 & -1 & 4 & 7 & 0 \end{pmatrix},$$

Gaussian elimination followed by row scaling and swapping rows yields

$$A_1 = \begin{pmatrix} 8 & -10 & -1 & 4 & 7 & 0 \\ 0 & 110 & 75 & -12 & 27 & -40 \\ 0 & 0 & 465 & 128 & -68 & 5 \\ 0 & 0 & 0 & -5587 & 4087 & 650 \\ 0 & 0 & 0 & 0 & -3833 & -1505 \end{pmatrix}$$

and the lattice basis is the rows of

$$A_2 = \begin{pmatrix} 1 & -1 & 0 & 0 & -1188 & -467 \\ 0 & 1 & 0 & 0 & 373 & 146 \\ 0 & 0 & 1 & 0 & -568 & -223 \\ 0 & 0 & 0 & -1 & 2369 & 930 \\ 0 & 0 & 0 & 0 & -3833 & -1505 \end{pmatrix}.$$

TABLE 2. Parameter values for runs of the indicated algorithms on random data. Ten runs were made for each m, n, ℓ value starting with random $m \times n$ integer matrices with entries between $-\ell$ and ℓ. Entries of the form min...max list the range of values so obtained. As explained in the text, each column is labeled with the theoretical upper bound on the quantity being measured.

				Algorithm 1* (fixed prec.)			MLLL algorithm	
m	n	ℓ	$\log_2(L)$	mF_L	mP_L	T(ms)	$\log_2(I_{\max})$	T(ms)
3	7	1	0...2	0...1	0...1	0.14...0.4	3...21	11.3...27.1
5	10	1	0...2	0...1	0...1	0.26...0.42	10...34	50.3...74.1
6	13	1	1...4	0...4	0...4	0.3...1.44	14...113	88...167
9	19	1	4...7	2...12	2...10	1.58...4.02	94...316	462...549
6	7	2	1...8	2...12	2...12	0.82...3.08	10...32	10.4...13.8
6	7	4	6...13	9...25	7...14	1.92...3.38	35...56	13.5...15.8
6	7	8	8...18	22...99	11...21	2.7...4.9	55...81	19.7...24.6
6	7	16	11...25	47...314	8...21	3.1...5.3	83...104	16.4...28.6
6	7	1	0...5	1...7	1...5	0.4...1.46	0...14	5.18...12.5
9	10	1	3...7	5...13	4...10	1.36...2.96	9...30	16...20.5
12	13	1	2...11	15...28	10...23	3.66...6.92	6...45	22.3...29.7
18	19	1	9...21	81...472	25...44	10.1...16.8	58...85	46.8...54.2

The large numbers in A_1 in this example are a consequence of Gaussian elimination followed by row scaling to clear out the denominators. We have $A_1 = T_{01}A_0$, where T_{01} has determinant $\approx 4.7 \times 10^7$. This is interesting because the rows of A_0 are in the lattice generated by the rows of A_2 and therefore there is an integer matrix T_{20} for which $A_0 = T_{20}A_2$ and $|\det(T_{20})| \geq 1$. If T_{12} is the transformation performed by the algorithm while converting A_1 into A_2,

$$\det(T_{12}) = \det(T_{20}^{-1}T_{01}^{-1}) = \frac{1}{\det(T_{20})\det(T_{01})} \leq \frac{1}{\det(T_{01})} \approx \frac{1}{4.7 \times 10^7}.$$

Hence Step 4 of Algorithm 1 will have to be executed many times in order to make $\det(T_{12})$ so small. In fact, we often have $\det(T_{12}) < \det(T_{01})^{-1}$ since equality would imply that T_{20}^{-1} has integer entries and the rows of the random matrix A_0 happen to be a basis for the desired lattice.

Experiments were run on random $m \times n$ integer matrices with entries in $[-\ell, \ell]$ for various m, n and ℓ. When numbers in the input matrix got too large, the infinite-precision version of Algorithm 1* was used. Since the matrices were fairly dense, the use of sparse matrix techniques probably added unnecessary overhead to all the run times (including those for the MLLL algorithm).

Tables 2 and 3 summarize the test results that are relevant to the run time analysis. Each table gives MIPS R4400 run times for Algorithm 1* and the MLLL algorithm as well as the number of bits in $I_{\max}$, the maximum intermediate result for the MLLL algorithm. The tables also list three quantities that appear in Section 3.2: the logarithm of the upper bound L on the entries of the input matrix produced by Gaussian elimination and row scaling; the number of trial divisions during factoring (column labeled mF_L); and the total number of prime divisors tried for all $g_{\bar{m}}$ values (column labeled $m\bar{P}_L$). The values corresponding to P_{L^m} and DP_{L^m} are not listed due to space limitations. The omitted P_{L^m} values (number of distinct prime factors) are somewhat less than the values in the mP_L column, and the omitted DP_{L^m} values ($1/m$ times the number of row operations in Step 3) are all less than 1.

TABLE 3. Parameter values for runs of the algorithms on random data using infinite precision integers in Algorithm 1*. For each m, n, ℓ, there were ten runs starting with random $m \times n$ integer matrices with entries between $-\ell$ and ℓ.

				Algorithm 1* (infinite prec.)			MLLL algorithm	
m	n	ℓ	$\log_2(L)$	mF_L	mP_L	T(sec)	$\log_2(I_{\max})$	T(sec)
5	10	1	0...2	0...1	0...1	.001... .003	10...34	.049... .065
6	13	1	1...4	0...3	0...3	.002... .009	14...113	.089... .167
7	16	1	2...6	0...7	0...4	.003... .022	65...154	.218... .304
12	25	1	6...12	9...78	5...10	.034... .076	397...635	4.48...5.22
16	32	1	12...18	32...175	15...22	.13... .258	768...1064	2.55...2.83
20	40	1	18...25	124...474	23...37	.423... .639	1288...1945	6.54...9.23
5	10	16	13...20	13...142	5...13	.02... .036	333...386	.106... .142
6	13	16	16...26	36...177	8...16	.04... .073	560...714	.266... .332
7	16	16	23...31	100...717	10...28	.06... .16	774...1091	.624... .791
8	19	16	28...34	208...689	15...25	.12... .192	1173...1466	1.05...2.45
12	25	16	50...53	1622...10^6	26...48	.518...2.59	2609...2775	3.86...4.34
16	32	16	70...74	$10^5 \ldots 7 \cdot 10^6$	36...66	2.2...6.71	4418...4706	15.4...17.3
18	19	2	26...34	375...2669	29...51	.202... .343	108...139	.06... .066
18	19	4	41...50	$10^4 \ldots 5 \cdot 10^5$	40...62	.394... .693	176...205	.067... .076
18	19	8	51...66	$5 \cdot 10^4 \ldots 10^7$	54...72	.585...10.1	228...265	.077... .089
18	19	16	69...83	$10^5 \ldots 9 \cdot 10^6$	56...76	3.01...10.9	297...335	.089... .102
18	19	32	89...99	$10^5 \ldots 6 \cdot 10^6$	53...88	7.38...18.5	377...404	.103... .117

Table 2 gives results for problems where the fixed-precision version of Algorithm 1* suffices. While Algorithm 1* is faster than the MLLL algorithm in all cases, it performs best when $m \ll n$ and worst when $m \approx n$. This is reasonable since a full-rank $m \times n$ input matrix for Algorithm 1* leads to an $(n-m) \times n$ null space basis matrix as input to the MLLL algorithm.

Table 3 compares the infinite-precision version of Algorithm 1* with the MLLL algorithm. Algorithm 1* is considerably faster in many cases, but it is slower in the last few rows of the table where $m = n - 1$ and the bound L on the entries in the input matrix gets large. In these cases, the large numbers in the mF_L column indicate that the bottleneck is the need to factor $g_{\bar{m}}$.

The relatively limited range of m, n and ℓ in Tables 2 and 3 values make it somewhat hard to decide how the observed running time compares to (3.4), but the $O(nm^2\bar{P}_L)$ term appears to dominate in Table 2. In Table 3, there are some cases where the $O(mF_L)$ term clearly dominates.

Since the theoretical bounds for LLL basis reduction also apply to the MLLL algorithm, the MLLL algorithm does $O(n^4 \log L)$ arithmetic operations on numbers $O(n \log L)$ bits long, but the $O(n^4 \log L)$ bound is known to be pessimistic in practice. This gives a pessimistic bound of $O(n^6 \log^3 L)$ for the overall running time, assuming that multiplication of two k-bit numbers takes time $O(k^2)$ as is appropriate for Gansner's arithmetic package [6]. The $\log_2(I_{\max})$ values in Table 3 are roughly proportional to $n \log_2(L)$, but the MLLL run times do not grow as rapidly as the $O(n^6 \log^3 L)$ bound.

5. REMARKS

The application described in Section 2 needs Algorithm 1* (or the MLLL algorithm) in order to work correctly, but it generates input bases that need little further processing. The implementation of Algorithm 1* was designed for this situation where a large sparse matrix describes the basis for the vector space that is to be intersected with $\mathbb{Z}^n$. Other types of test problems are readily generated from random data, but it is hard to say how well they reflect problems that might occur in actual applications.

The results in Section 4 clearly show that the algorithm presented here (Algorithm 1*) is dramatically superior for problems like those encountered in the font application, but there are other problems where the MLLL algorithm is better. These problems are the ones where the input matrix contains large numbers and Algorithm 1* gets bogged down in the factoring step.

It is tempting to argue that we could avoid this problem by having Algorithm 1 proceed with p values that are not prime but have no small prime factors. The only way this could interfere with Step 3 is if we stumble into a number that is not relatively prime to p. In that case p has been factored and we can restart Step 3 with a new p value. The problem is that Step 3 might discover that the condition on $b_1, b_2, \ldots, b_m$ cannot be satisfied modulo p when it could be satisfied modulo one of the unknown prime factors of p. The overall effect would be that the algorithm produces a sublattice of the desired lattice.

References

1. L. Babai, *On Lovász' lattice reduction and the nearest lattice point problem*, Combinatorica **6** (1986), no. 1, 1–13. MR **88a:**68049
2. Charles Bigelow and Kris Holmes, *The design of a Unicode font*, Elec. Publish. **6** (1993), no. 3, 289–305.
3. J. A. Buchmann and H. W. Lenstra, Jr., *Approximating rings of integers in number fields*, Journée de Théorie des Nombres de Bordeaux **6** (1994), 221–260. MR **96m:**11092
4. Henri Cohen, *A course in computational algebraic number theory*, Springer Verlag, Berlin, 1993. MR **94i:**11105
5. Alan M. Frieze, Johan Hastad, Ravi Kannan, Jeffery C. Lagarias, and Adi Shamir, *Reconstructing truncated integer variables satisfying linear congruences*, SIAM J. Comput. **17** (1988), no. 2, 262–280. MR **89d:**11115
6. E. R. Gansner, *An infinite precision integer package for c++*, AT&T Bell Laboratories technical memorandum, 1987.

→ Eric Grosse and John D. Hobby, *Improved rounding for spline coefficients and knots*, Math. of Comput. **63** (1994), no. 207, 175–194. MR **94j:**65019

8. J. Hastad, B. Just, J. C. Lagarias, and C. P. Schnorr, *Polynomial-time algorithms for finding integer relations among real numbers*, SIAM J. Comput. **18** (1989), no. 5, 859–881. MR **90g:**11171
9. Johan Hastad, *Solving simultaneous modular equations of low degree*, SIAM J. Comput. **17** (1988), no. 2, 336–341. MR **89e:**68049
10. John D. Hobby, *Generating automatically tuned bitmaps from outlines*, J. ACM **40** (1993), no. 1, 48–94.
11. D. E. Knuth, *The art of computer programming*, vol. 2, Addison Wesley, Reading, Massachusetts, 1981. MR **83i:**68003
12. A. K. Lenstra, H. W. Lenstra, Jr., and L. Lovász, *Factoring polynomials with rational coefficients*, Math. Annalen **261** (1982), 515–534. MR **84a:**12002

→ H. W. Lenstra, Jr., *Integer programming with a fixed number of variables*, Math. Oper. Res. **8** (1983), no. 4, 538–548. MR **86f:**90106

14. Andrew M. Odlyzko, *Cryptanalytic attacks on the multiplicative knapsack cryptosystem and on Shamir's fast signature scheme*, IEEE Trans. Inf. Theory **IT-30** (1984), no. 4, 594–601. MR **86i:**94041
15. M. Pohst, *A modification of the LLL reduction algorithm*, Journal of Symbolic Computation **4** (1987), 123–128. MR **89c:**11183

16. J. M. Pollard, *A Monte Carlo method for factorization*, BIT Nord. Tid. f. Inf. **15** (1975), no. 3, 331–334. MR **52:**13611
17. Adi Shamir, *A polynomial time algorithm for breaking the basic Merkle-Hellman cryptosystem*, IEEE Trans. Inf. Theory **IT-30** (1984), no. 5, 699–704. MR **86m:**94032
18. Kokichi Sugihara, *On finite-precision representations of geometric objects*, J. Comput. Syst. Sci. **39** (1989), no. 2, 236–247. MR **91c:**68060

Bell Laboratories, Lucent Technologies, 700 Mountain Ave., Murray Hill, New Jersey 07974

E-mail address: hobby@bell-labs.com