Compact incremental Gaussian elimination over $\mathbb{Z}/2\mathbb{Z}$

A.K. Lenstra and M.S. Manasse
The University of Chicago
Technical Report 88-16, October 1988

Address for copies of technical reports:

Department of Computer Science
University of Chicago
Ryerson Hall
1100 East 58th Street
Chicago IL 60637

Compact incremental Gaussian elimination over $\mathbb{Z}/2\mathbb{Z}$

Arjen K. Lenstra
Department of Computer Science
The University of Chicago
1100 E 58th Street
Chicago, IL 60637

Mark S. Manasse
DEC Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301

Abstract. This note describes a simple algorithm for Gaussian elimination over $\mathbb{Z}/2\mathbb{Z}$. To find dependencies among n rows of length m each, the algorithm needs storage for at most $n \cdot m + O(m \cdot \log n)$ bits. The algorithm is row-oriented, and the computation on the first n_0 rows, for any integer $n_0 \le n$, does not depend on rows n_0+1 through n. This makes it possible to start the elimination process given any number of rows of the matrix, and to proceed with the computation as more rows become available. The algorithm has been successfully applied to find dependencies among about 50000 rows of 50000 bits each. The method is well suited for implementation on vector computers or multi-processor machines.

Compact incremental Gaussian elimination over $\mathbb{Z}/2\mathbb{Z}$

Let $r_i = (r_{i,1}, r_{i,2}, \ldots, r_{i,m}) \in (\mathbb{Z}/2\mathbb{Z})^m$ for $1 \le i \le n$ be n rows of length m over $\mathbb{Z}/2\mathbb{Z}$. It is well known that the problem to decide whether there exist $c_1, c_2, \ldots, c_n \in \mathbb{Z}/2\mathbb{Z}$ such that $\sum_{i=1}^{n} c_i \cdot r_{i,j} \equiv 0 \bmod 2$ for $1 \le j \le m$, and to find the c_i if they exist, can be solved by means of Gaussian elimination over $\mathbb{Z}/2\mathbb{Z}$. For ordinary Gaussian elimination, however, one has to keep track of a so-called 'history matrix', requiring extra storage of $\Omega(n^2)$ elements of $\mathbb{Z}/2\mathbb{Z}$.

For the application we had in mind, reduction of matrices with tens of thousands of rows and columns resulting from Carl Pomerance's quadratic sieve algorithm [4, 5], this extra storage could easily become problematic. It is straightforward to alter the ordinary Gaussian elimination in such a way that the history matrix does not occupy any extra space. In this note we describe this compact Gaussian elimination algorithm.

Here we should note that, in our application, the matrices are very sparse. Gaussian elimination will rapidly cause fill-in of the matrix. As a result the matrix becomes dense, which will not only lead to storage problems, but also to a very slow performance of the algorithm. These two problems can be avoided by using sparse matrix techniques [2, 6]. Indeed, we are working on implementations of those sparse matrix algorithms. In the mean time, however, we had to reduce several huge matrices, so we needed an algorithm that was easy to implement and that would not cause storage problems. Slow performance of this algorithm was not a major consideration.

After the elimination algorithm had been implemented and successfully reduced the matrices we got [1], Carl Pomerance brought a paper by Parkinson and Wunderlich to our attention [3]. Our algorithm appears to be a slight variation of their Algorithm A. Algorithm A in [3] looks for the ith pivot in column i of the matrix for $i = 1, 2, \ldots, m$ in succession, our algorithm looks for the ith pivot in the ith row. As a consequence, our algorithm is completely row-oriented, and therefore incremental, by which we mean the following. If only the first $n_0 \le n$ rows of the matrix are known at a given moment, then we can begin the elimination on those

version 19881031.
Key words: Gaussian elimination, factorization.

first n_0 rows. The elimination process can then be continued without any loss of efficiency as soon as rows number n_0+1 through n_1 become available, for any $n_0 < n_1 \le n$, and so on, until sufficiently many dependencies have been found, or until the matrix is complete. For our application, where the rows become available over a period of several weeks, this has the advantage that the algorithm can be applied, say, once a day to process the newly found rows. Once the matrix is complete the elimination is completed almost immediately, and one does not have to wait a long time for the final result.

Algorithm A as described in [3] does not have this advantage, although it could easily be changed. Parkinson and Wunderlich change Algorithm A along different lines into an algorithm that operates on the columns of the complete $n \times m$-matrix. They recommend using this later algorithm, but since it requires the whole matrix to be available before the elimination can be carried out, it is less useful for our purposes.

We now describe the algorithm. Suppose that rows r_1 through r_{n_b} have been processed already in b previous applications of the algorithm, for some integer $b \ge 0$ and $b \le n_b \le n$. So, initially we will have $b = 0$ and $n_0 = 0$. Let $r_{n_b+1}, r_{n_b+1}, \ldots, r_{n_{b+1}}$ with $n_{b+1} \le n$ be the rows that have to be processed next. Put $e_i = 0$ for $1 \le i \le m$. Perform steps (a) and (b).

(a) *Eliminate with the first n_b rows.*
Put d_j equal to r_j for $n_b+1 \le j \le n_{b+1}$. For $i = 1, 2, \ldots, n_b$ in succession perform step (a1) if $u_i > 0$.

(a1) Replace e_{u_i} by i. For $n_b+1 \le j \le n_{b+1}$ do the following: if $d_{j,u_i} = 1$, then replace d_j by $d_j + d_i$, where the addition is done coordinate-wise modulo 2. (Here d_{j,u_i} means the u_ith coordinate of d_j.)

(b) *Final elimination of the new rows.*
For $j = n_b+1, n_b+2, \ldots, n_{b+1}$ in succession perform step (b1).

(b1) If there is an i with $1 \le i \le m$ such that $d_{j,i} = 1$ and $e_i = 0$, then put u_j equal to such an i and perform step (b1a). Otherwise, perform step (b1b).

(b1a) Replace e_{u_j} by j and d_{j,u_j} by 0. For $j+1 \le k \le n_{b+1}$ do the following: if $d_{k,u_j} = 1$, then replace d_k by $d_k + d_j$, where the addition is done coordinate-wise modulo 2.

(b1b) Put c_i equal to 0 for $1 \le i \le n$ and u_j equal to 0. Replace c_j by 1, replace c_{e_k} by 1 for those k with $1 \le k \le m$ for which $d_{j,k} = 1$, and output the solution $c_1, c_2, \ldots, c_n$.

Some explanation might be helpful to understand the algorithm. If $e_i = j$ we know that the ith coordinate of d_j is used as a pivot to eliminate non-zero entries in the ith coordinates of $d_{j+1}, d_{j+2}, \ldots, d_n$. This ith coordinate has not been used before because in step (b1) a coordinate is selected for which e_i still equals zero (the initial situation, meaning that no row took care of the ith coordinate yet), and it is a valid pivot because, again in step (b1), we also have that $d_{j,i}$ equals 1. Before the elimination step is carried out in (b1a), however, the $d_{j,i}$, for the i that has been chosen in step (b1), is set to zero. In this way the non-zero entry in the ith coordinate in later rows will not be set to zero; instead, this 1 will from then on mean 'the ith coordinate in this row is eliminated using row number $e_i = j$'. So, the meaning of a one in coordinate i of a row depends on the value of e_i: if $e_i = 0$, then it is a 'real' one, otherwise it means that coordinate i has already been cleaned by d_{e_i}. Clearly, if no i can be found with a 1 in the ith coordinate and $e_i = 0$, it means that the row under consideration is completely cleaned, and the dependency can then be found as in step (b1b).

Some remarks about implementations of this algorithm are in order. Clearly, it is not at all necessary that the whole matrix (or the part that is known) resides in core. In step (a) it suffices to read the d_i and u_i for $i = 1, 2, \ldots, n_b$ in succession into core, one at a time. For rows that have been processed (r_1 through r_{n_b}), it suffices therefore to store the corresponding rows d_1 through d_{n_b} with their corresponding u_1 through u_{n_b} in some external file (where the d_i needs only be present if $u_i > 0$). After completion of step (b) for a certain j, the row d_j and its corresponding u_j can be appended at the end of this file. In that way the processed rows can successively be retrieved during later applications of step (a). Each row can be stored in sparse or dense representation, depending on its number of non-zero coordinates. The cross-over point between sparse and dense representation is implementation-dependent.

If m is huge it may be useful, as it was in our case, to determine some integer s such that s densely-represented rows will fit in core at the same time. If $n_{b+1}-n_b > s$, the new rows can then be processed in blocks of at most s rows at a time.

In our application the rows are much sparser at one end than at the other. In such cases it is advisable to look for pivots from the sparsest end (the i that will be selected in step (b1)), as it will lead to fewer eliminations and consequently slower fill-in. Although this strategy certainly delays fill-in of the matrix, it is by no means able to prevent it. While we have not considered it carefully, it is possible that the techniques described by A.M. Odlyzko [2] can be used as a strategy to select pivots and reduce fill-in.

In a dense representation the vectors can of course be represented by one bit per coordinate, and consecutive bits can be packed into one word. Notice that the actual elimination step, the vector additions modulo 2 in steps (a1) and (b1a), can be implemented by XOR-ing the consecutive words of both vectors if both vectors are densely represented. On vector machines these steps can be carried out very fast. Also notice that the algorithm can be parallelized in various ways (by assigning different coordinate ranges to different processors, or by assigning different blocks of new rows to different processors, or a combination of these two).

We conclude this note with the same example that was given in [3]. Let $n_1 = 4$ and

$$r_1 = (0,0,1,0,0,1,0),$$
$$r_2 = (0,1,0,0,0,1,1),$$
$$r_3 = (1,0,0,0,1,0,0),$$
$$r_4 = (0,0,1,1,0,0,1).$$

We immediately proceed to step (b), and find for $j = 1$

$$u_1 = 3,\ e_3 = 1,$$
$$d_1 = (0,0,0,0,0,1,0),$$
$$d_2 = (0,1,0,0,0,1,1),$$
$$d_3 = (1,0,0,0,1,0,0),$$
$$d_4 = (0,0,1,1,0,1,1),$$

for $j = 2$

$$u_2 = 2,\ e_2 = 2,$$
$$d_2 = (0,0,0,0,0,1,1),$$
$$d_3 = (1,0,0,0,1,0,0),$$
$$d_4 = (0,0,1,1,0,1,1),$$

for $j = 3$

$$u_3 = 1,\ e_1 = 3,$$
$$d_3 = (0,0,0,0,1,0,0),$$
$$d_4 = (0,0,1,1,0,1,1),$$

and for $j = 4$

$$u_4 = 4,\ e_4 = 4,$$
$$d_4 = (0,0,1,0,0,1,1).$$

No dependency has been found yet, and we can store u_1, d_1, u_2, d_2, u_3, d_3, u_4, and d_4 for later use.

Let $n_2 = 9$ and

$$r_5 = (1,1,0,0,1,0,0),$$
$$r_6 = (0,0,0,1,0,0,1),$$
$$r_7 = (0,0,0,1,1,0,0),$$
$$r_8 = (1,0,1,0,0,1,0),$$
$$r_9 = (0,1,0,1,0,0,1).$$

Step (a) gives for i =1

$$d_5 = (1,1,0,0,1,0,0),$$
$$d_6 = (0,0,0,1,0,0,1),$$
$$d_7 = (0,0,0,1,1,0,0),$$
$$d_8 = (1,0,1,0,0,0,0),$$
$$d_9 = (0,1,0,1,0,0,1),$$

for $i = 2$

$$d_5 = (1,1,0,0,1,1,1),$$
$$d_6 = (0,0,0,1,0,0,1),$$
$$d_7 = (0,0,0,1,1,0,0),$$
$$d_8 = (1,0,1,0,0,0,0),$$
$$d_9 = (0,1,0,1,0,1,0),$$

for $i = 3$

$$d_5 = (1,1,0,0,0,1,1),$$
$$d_6 = (0,0,0,1,0,0,1),$$
$$d_7 = (0,0,0,1,1,0,0),$$
$$d_8 = (1,0,1,0,1,0,0),$$
$$r_9 = (0,1,0,1,0,1,0),$$

and for $i = 4$

$$d_5 = (1,1,0,0,0,1,1),$$
$$d_6 = (0,0,1,1,0,1,0),$$
$$d_7 = (0,0,1,1,1,1,1),$$
$$d_8 = (1,0,1,0,1,0,0),$$
$$d_9 = (0,1,1,1,0,0,1).$$

We now have that $e_1 = 3$, $e_2 = 2$, $e_3 = 1$, $e_4 = 4$, and that e_5, e_6, and e_7 are equal to zero. Next, we proceed to step (b), and find for $j = 5$

$$u_5 = 6,\ e_6 = 5,$$
$$d_5 = (1,1,0,0,0,0,1),$$
$$d_6 = (1,1,1,1,0,1,1),$$
$$d_7 = (1,1,1,1,1,1,0),$$
$$d_8 = (1,0,1,0,1,0,0),$$
$$d_9 = (0,1,1,1,0,0,1),$$

for $j = 6$

$$u_6 = 7,\ e_7 = 6,$$
$$d_6 = (1,1,1,1,0,1,0),$$
$$d_7 = (1,1,1,1,1,1,0),$$
$$d_8 = (1,0,1,0,1,0,0),$$

$d_9 = (1,0,0,0,0,1,1)$,

for $j = 7$

$u_7 = 5$, $e_5 = 7$,
$d_7 = (1,1,1,1,0,1,0)$,
$d_8 = (0,1,0,1,1,1,0)$,
$d_9 = (1,0,0,0,0,1,1)$,

for $j = 8$ we find that $r_{e_2} = r_2$, $r_{e_4} = r_4$, $r_{e_5} = r_7$, $r_{e_6} = r_5$, and r_8 are linearly dependent over $\mathbb{Z}/2\mathbb{Z}$, and for $j = 9$ finally we find that $r_{e_1} = r_3$, $r_{e_6} = r_5$, $r_{e_7} = r_6$ and r_9 are linearly dependent over $\mathbb{Z}/2\mathbb{Z}$.

References

1. A.K. Lenstra, M.S. Manasse, Factoring by electronic mail, in preparation.
2. A.M. Odlyzko, "Discrete logarithms and their cryptographic significance," pp. 224-314; in: T.Beth, N. Cot, I. Ingemarsson (eds), *Advances in cryptology*, Springer Lecture Notes in Computer Science, vol. 209, 1985.
3. D. Parkinson, M. Wunderlich, "A compact algorithm for Gaussian elimination over GF(2) implemented on highly parallel computers," *Parallel Computing*, v. 1, 1984, pp. 65-73.
4. C. Pomerance, "Analysis and comparison of some integer factoring algorithms," pp. 89-139; in: H.W. Lenstra, Jr., R. Tijdeman (eds), *Computational methods in number theory*, Mathematical Centre Tracts 154, 155, Mathematisch Centrum, Amsterdam, 1982.
5. R.D. Silverman, "The multiple polynomial quadratic sieve," *Math. Comp.*, v. 48, 1987, pp. 329-339.
6. D.H. Wiedemann, "Solving sparse linear equations over finite fields," *IEEE Transactions on Information Theory*, v. 32, 1986, pp. 54-62.