Solving Large Sparse Linear Systems Over Finite Fields

B. A. LaMacchia *
A. M. Odlyzko

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Many of the fast methods for factoring integers and computing discrete logarithms require the solution of large sparse linear systems of equations over finite fields. This paper presents the results of implementations of several linear algebra algorithms. It shows that very large sparse systems can be solved efficiently by using combinations of structured Gaussian elimination and the conjugate gradient, Lanczos, and Wiedemann methods.

1. Introduction

Factoring integers and computing discrete logarithms often requires solving large systems of linear equations over finite fields. General surveys of these areas are presented in [14, 17, 19]. So far there have been few implementations of discrete logarithm algorithms, but many of integer factoring methods. Some of the published results have involved solving systems of over 6×10^4 equations in more than 6×10^4 variables [12]. In factoring, equations have had to be solved over the field $GF(2)$. In that situation, ordinary Gaussian elimination can be used effectively, since many coefficients (either 32 or 64 depending on machine word size) can be packed into a single machine word, and addition can be implemented as the exclusive-or operation. Even so, the large size of the systems to be solved has often caused storage problems (a 6×10^4 by 6×10^4 system requires approximately 110 million words of storage on a 32-bit machine), and it has often been difficult to obtain a correspondingly large amount of computer time. In many cases, the linear systems were purposefully kept

*Present address: MIT, Cambridge, MA 02139

A.J. Menezes and S.A. Vanstone (Eds.): Advances in Cryptology - CRYPTO '90, LNCS 537, pp. 109-133, 1991.

smaller than would have been optimal from the standpoint of other parts of the factoring algorithm, just to avoid the linear algebra difficulties.

Clearly we cannot hope to be able to solve future systems (even in $GF(2)$) using only ordinary Gaussian elimination. As the size of integers being factored increases, so does the size of the system of equations which must be solved. In addition, the recently discovered number field sieve currently requires (when applied to factoring general integers) the solution of equations over the integers, not just modulo 2. (The best way to obtain such a solution appears to be either to solve the system modulo many small or moderate primes and then apply the Chinese remainder theorem, or else to solve it modulo a particular prime, and then lift that solution to one modulo a power of that prime.) In the case of the number field sieve applied to general integers, the linear algebra problem is currently one of the critical bottlenecks that keep it impractical.

Even in cases where the equations have to be solved modulo 2, linear algebra difficulties are becoming a serious bottleneck. As an example, the very recent factorization of $F_9 = 2^{2^9} + 1 = 2^{512} + 1$ (using a special form of the number field sieve for integers of this form) by A. Lenstra and M. Manasse involved solving a system with dimension $n \approx 2 \times 10^5$. The attack on the RSA challenge cipher (which will require factoring a 129 decimal digit integer) that is currently planned using the quadratic sieve might require solving a system with $n \approx 4 \times 10^5$.

For discrete logarithm algorithms, the linear algebra problem has always seemed to be even more important than in factoring, since the equations have to be solved modulo large primes. The largest system that has been solved was of size about 1.6×10^4 by 1.6×10^4, modulo $2^{127} - 1$, which arose in connection with discrete logarithms in $GF(2^{127})$ [3, 17].

For an $n \times n$ system with $n \approx 10^5$, ordinary Gaussian elimination takes about $n^3 \approx 10^{15}$ operations. Modern supercomputers are capable of between 10^8 and 10^9 integer operations per second, so 10^{15} operations might take a few weeks to a few months on such a machine, if one matrix operation took one machine instruction. In practice, since up to 64 matrix operations are performed in a single supercomputer operation, the required time decreases substantially. However, for those without access to supercomputers, time can easily be a barrier. It is possible to obtain 10^{15} machine operations for free, as that is the approximate amount of work that the recent integer factorization achieved by A. Lenstra and M. Manasse [12] required. However, that effort involved a very decentralized and only loosely coordinated computation on hundreds of workstations. This is acceptable for the sieving stage of the quadratic sieve algorithm, but it causes problems when one tries to solve a system of linear equations; solving such systems requires very close coordination

and errors propagate quickly, destroying the validity of the final result. Thus the linear algebra phase requires the use of a single machine (although it can be a massively parallel computer), and it can often be hard to obtain access to one that is fast enough. Memory requirements also present serious difficulties. If the problem is to find a solution modulo 2, the full matrix has 10^{10} bits, or 1,250 megabytes. Only a few supercomputers have this much main memory. On all other machines, one has to work on the matrix in blocks, which slows down the operation considerably.

Both the time and space requirements become much more serious when solving equations modulo a "moderate size" prime p. If $p \approx 2^{100}$, the operation count goes up from 10^{15} to 10^{19}, which is prohibitive even for a supercomputer. The storage requirements increase to 10^{12} bits, which is 125 gigabytes, considerably more than any existing machine has.

Fast matrix multiplication algorithms do not offer much hope. The Strassen $n^{2.81}$ method [21] is practical for n on the order of several hundred, but does not save enough. Later methods, of which the Coppersmith-Winograd $n^{2.376}$ algorithm [8] is currently the fastest, are impractical.

If the equations that arise in factoring and discrete log algorithms were totally random, there would be little hope of solving large systems. However, these equations, while they appear fairly random in many respects, are extremely sparse, with usually no more than 50 non-zero coefficients per equation. Moreover, they are relatively dense in some parts, and very sparse in others. Previous Gaussian elimination implementations, as is mentioned below, already take advantage of some of these features. In addition, several special systematic methods have been developed to take advantage of this sparsity. They are:

1. structured Gaussian elimination (also called intelligent Gaussian elimination) [17],
2. the conjugate gradient and Lanczos algorithms [7, 17],
3. the Wiedemann algorithm [22].

Structured Gaussian elimination was designed to reduce a problem to a much smaller one that could then be solved by other methods. The other methods have running times that are expected to be of order not much bigger than n^2 for a sparse $n \times n$ system. Theoretically, the Wiedemann algorithm is the most attractive of all these techniques, since it can be rigorously proved to work with high probability (if one randomizes certain choices in the initial stages of the algorithm), while the other methods are only heuristic. Asymptotically, it was thought for a long time that the problem of solving large linear systems was the crucial bottleneck determining how algorithms such as the quadratic size performed [18].

This view then changed, as the Wiedemann algorithm showed that linear algebra was only about as important in determining the asymptotic complexity of algorithms such as the quadratic sieve as other steps.

To the authors' knowledge, the Wiedemann algorithm has never been tested on a large system. The conjugate gradient and Lanczos methods have been tested [7, 17], but only on fairly small systems. Structured Gaussian elimination was tested on some very large systems in [17], but those tests used simulated data, while the runs on real data derived from a factoring project solved only fairly small systems. More recently, this method was implemented and used to solve some fairly large binary systems by Pomerance and Smith [20]. Even more recently, a version of this method was used by A. Lenstra and M. Manasse in their factorization of F_9.

This paper reports on the performance of some of these algorithms on several very large sparse systems derived from factoring and discrete logarithm computations. The largest of the systems that were solved had about 3×10^5 equations in about 10^5 unknown, modulo a prime of 191 bits; this system arose in the computation of discrete logarithms in a certain prime field [10]. The basic conclusion is that the conjugate gradient and Lanczos algorithms have essentially the same performance and are very effective in finite fields. One of their advantages is that they use relatively little space. However, even these algorithms are too slow to tackle very large problems. The Wiedemann algorithm (which was not implemented) has modest space requirements, almost exactly the same as those of the conjugate gradient and Lanczos methods. Its running time is likely to be comparable to those of the conjugate gradient and Lanczos algorithms, with the precise comparison depending on the architecture of the computer and implementation details.

We have also found that structured Gaussian elimination is very effective in reducing a large, sparse problem to a much smaller one. Structured Gaussian elimination takes very little time, and can be implemented so as to take very little space. When dealing with a large sparse system modulo a prime, it appears that the best procedure is to first apply structured Gaussian elimination to obtain a smaller system that is still sparse, and then to solve the smaller system with one of the conjugate gradient, Lanczos, or Wiedemann algorithms. When working with equations modulo 2, it is probably better to use ordinary Gaussian elimination for the final step, or else one can use conjugate gradient, Lanczos, or Wiedemann for the entire problem.

Section 2 describes the data sets that were used in the computations, as well as the machine on which the algorithms were run. We describe in Section 3 the Lanczos and conjugate gradient algorithms, and their performance. Section 4 briefly discusses the

Wiedemann algorithm. Structured Gaussian elimination is detailed in Section 5

2. Machines and data

All computations reported in this paper were carried out on a Silicon Graphics 4D-220 computer. It has 4 R3000 MIPS Computers, Inc., 25 MHz processors, each rated at about 18 mips, or 18 times the speed of a DEC VAX 11/780 computer (and about 1.3 times the speed of a DECstation 3100). The parallel processing capability of this system was not used; all times reported here are for a single processor. This machine has 128 Mbytes of main memory.

All programs were written in C or Fortran. They were not carefully optimized, since the aim of the project was only to obtain rough performance figures. Substantial performance improvements can be made fairly easily, even without using assembly language.

Table 1 describes the linear systems that were used in testing the algorithms. Data sets A through J were kindly provided by A. Lenstra and M. Manasse, and come from their work on factoring integers [12]. Sets A, B and C result from runs of the multiple polynomial quadratic sieve (*mpqs*) with the single large prime variation, but have had the large primes eliminated. Set D also comes from a run of *mpqs*, but this time the large primes were still in the data (except that equations with a large prime that does not occur elsewhere were dropped). Set E comes from a factorization that used the new number field sieve, and set $E1$ was derived from set E by deleting 5000 columns at the sparse end of the matrix. (Set $E1$ was created to simulate a system that has more extra equations than E does, but has comparable density.) Sets F and G derive from runs of *ppmpqs*, the two large prime variation of *mpqs* [13]. Both sets arose during the factorization of the same integer; set G was obtained by running the sieving operation longer. Sets H and I come from other runs of *ppmpqs* (set I was produced during the factorization of the 111 decimal digit composite cofactor of $7^{146}+1$). Set J was produced by the number field sieve factorization of F_9. All of these data sets (A-J) were tested modulo 2 only.

Data set K was obtained in the process of computing discrete logarithms modulo a prime p of 192 bits [10], and had to be solved modulo $\frac{p-1}{2}$ (a prime of 191 bits) and modulo 2. Set K was tested modulo both numbers. Sets $K0$ through $K6$ and L were derived from set K. Set $K2$ was derived by deleting the 144,000 equations from F that had the highest number of non-zero entries (*weight*). Set $K4$ was derived from K by deleting the 144,000 equations that had the lowest weights. Sets $K0$, $K1$, $K3$, $K5$, and $K6$ were obtained by deleting randomly chosen equations from K. (The reason the number of unknowns

Table 1: Large sparse sets of equations

Name	Number of Equations	Number of Unknowns	Average Number of Non-zeros per Equation
A	35,987	35,000	20.4
B	52,254	50,001	21.0
C	65,518	65,500	20.4
D	123,019	106,121	11.0
E	82,470	80,015	47.1
$E1$	82,470	75,015	46.6
F	25,201	25,001	46.7
G	30,268	25,001	47.9
H	61,343	30,001	49.3
I	102,815	80,001	43.2
J	226,688	199,203	48.8
K	288,017	96,321	15.5
$K0$	216,105	95,216	15.5
$K1$	165,245	93,540	15.5
$K2$	144,017	94,395	13.8
$K3$	144,432	92,344	15.5
$K4$	144,017	89,003	17.1
$K5$	115,659	90,019	15.5
$K6$	101,057	88,291	15.5
L	7,262	6,006	80.5
M	164,841	94,398	16.9

varies in these sets is that in sets $K, K0, \ldots, K6$, only the unknowns that actually appear in the equations are counted.) Set L was derived from set K by using structured Gaussian elimination. Finally, data set M was produced while computing discrete logarithms modulo a prime p of 224 bits [10].

3. Lanczos and conjugate gradient methods

We first describe the Lanczos algorithm. Suppose that we have to solve the system

$$Ax = w \tag{3.1}$$

for a column n-vector x, where A is a symmetric $n \times n$ matrix, and w is a given column n-vector. Let

$$w_0 = w, \tag{3.2}$$

$$v_1 = Aw_0, \tag{3.3}$$

$$w_1 = v_1 - \frac{(v_1, v_1)}{(w_0, v_1)} w_0, \tag{3.4}$$

and then, for $i \geq 1$, define

$$v_{i+1} = Aw_i, \tag{3.5}$$

$$w_{i+1} = v_{i+1} - \frac{(v_{i+1}, v_{i+1})}{(w_i, v_{i+1})} w_i - \frac{(v_{i+1}, v_i)}{(w_{i-1}, v_i)} w_{i-1}. \tag{3.6}$$

The algorithm stops when it finds a w_j that is conjugate to itself, i.e. such that $(w_j, Aw_j) = 0$. This happens for some $j \leq n$. If $w_j = 0$, then

$$x = \sum_{i=0}^{j-1} b_i w_i \tag{3.7}$$

is a solution to Equation 3.1, where

$$b_i = \frac{(w_i, w)}{(w_i, v_{i+1})}. \tag{3.8}$$

(If $w_j \neq 0$, the algorithm fails.)

The Lanczos algorithm was invented to solve systems with real coefficients [11]. To solve systems over finite fields, we simply take Equations 3.3 to 3.8 and apply them to a finite field situation. This causes possible problems, since over a finite field one can have a non-zero vector that is conjugate to itself. However, this difficulty, as well as some other ones that arise, can be overcome in practice.

In addition to dealing with self-conjugacy, we have to cope with the fact that the systems we need to solve are in general not symmetric, but rather are of the form

$$Bx = u, \tag{3.9}$$

where B is $m \times n$, $m \geq n$, x is an unknown column n-vector, and u is a given column m-vector. (We assume that B has rank n, or at least that u is in the space generated by the columns of B.) We first embed the field over which we have to solve the equations in a possibly larger field F with $|F|$ considerably larger than n. We then let D be an $m \times m$ diagonal matrix with the diagonal elements selected at random from $F \backslash \{0\}$, and we let

$$\begin{aligned} A &= B^T D^2 B, \\ w &= B^T D^2 u. \end{aligned} \tag{3.10}$$

A solution to Equation 3.9 is then a solution to Equation 3.1, and we expect that with high probability a solution to Equation 3.1 will be a solution to Equation 3.9. The random choice of D ought to ensure that the rank of A will be the same as the rank of B (this is not always true), and that we will not run into a self-conjugate w_i in the Lanczos algorithm. Experience has shown that this is indeed what happens.

The Lanczos algorithm is not restricted to dealing with sparse matrices, but that is where it is most useful. At iteration i, we need to compute the vector $v_{i+1} = Aw_i$ ($v_i = Aw_{i-1}$ is already known from the previous iteration), the vector inner products (v_{i+1}, v_{i+1}), (w_i, v_{i+1}), and (v_{i+1}, v_i), and then form w_{i+1} using Equation 3.6. The matrix A will in general be dense, even when B is sparse. However, we do not need to form A, since to compute Aw_i we use Equation 3.10 and compute

$$B^T(D^2(Bw_i)), \tag{3.11}$$

Suppose that B has b non-zero entries. We will further assume, as is the case for the matrices arising in factorization and discrete logarithm computations, that almost all these b entries are ± 1. Let c_1 be the cost of an addition or subtraction in F, and c_2 the cost of a multiplication. Then computing Bw_i costs approximately

$$b\, c_1,$$

multiplying that by D^2 costs

$$n\, c_2,$$

and multiplying D^2Bw_i by B^T costs approximately

$$b\,c_1,$$

for a total cost of about

$$2\,b\,c_1 + n\,c_2$$

for each evaluation of Aw_i. The computation of each vector inner product costs about $nc_1 + nc_2$, and so the cost of each iteration is about

$$2\,b\,c_1 + 4\,n\,c_1 + 5\,n\,c_2, \tag{3.12}$$

so that the total cost of obtaining a solution is about

$$2\,n\,b\,c_1 + 4\,n^2\,c_1 + 5\,n^2\,c_2. \tag{3.13}$$

In this rough accounting we do not charge for the cost of accessing elements of arrays or lists, which will often be the dominant factor, especially when dealing with binary data. On the other hand, on some machines one can perform additions and multiplications in parallel. Therefore one should treat the estimate given by Equation 3.13 as a very rough approximation.

In practice, B will usually be stored with rows represented by lists of positions where that row has a non-zero entry, and (in the case of non-binary problems) by lists of non-zero coefficients. Thus we need about b pointers and (for non-binary data) about b bits to indicate whether that coefficient is $+1$ or -1. (Non-zero coefficients that are not ± 1 are relatively rare and can be treated separately.) The pointers normally have to be of at least $\lceil \log_2 n \rceil$ bits, but one can reduce that by taking advantage of the fact that most indices of positions where the coefficient is non-zero are very small. In any case, storage is not a problem even for very large systems. In the implementations described below, full 32-bit words were used for all pointers and coefficients. The largest of the systems in Table 1 had $b \lesssim 10^7$, so this did not cause any problems on the machine that was used. In fact, our algorithm had separate representations for both B and B^T, which is wasteful.

In a typical situation, we expect that the Lanczos method will be applied to a system that was processed first by structured Gaussian elimination, and so it will not be too sparse. As a result, the cost of the vector inner products ought to be dominated by the cost of the matrix-vector products, $2nbc_1$. As we will describe later, memory access times will often be the dominant factor in determining the efficiency of the matrix-vector product.

In discrete logarithm applications, the system given by Equation 3.9 is usually overdetermined, and so the aim is to find the unique x that solves it. In applications to factoring,

though, one is looking for linear dependencies in a set of vectors, and it is necessary to find several such dependencies. This can be stated as the problem of solving the system in Equation 3.9 with B fixed, but for several vectors u, so that we need $x_1, x_2, \ldots, x_r$ such that

$$Bx_j = u_j, \quad 1 \leq j \leq r. \tag{3.14}$$

It is possible to solve all these systems at once, without rerunning the Lanczos algorithm r times. Let $z_j = B^T D^2 u_j$. Apply the algorithm as presented above with $w = z_1$. This produces the vectors $w_0, w_1, \ldots, w_{n-1}$ which are conjugate to each other; i.e. $(w_i, Aw_j) = 0$ for $i \neq j$. Now if

$$x_k = \sum_{j=0}^{n-1} c_{k,j} w_j, \tag{3.15}$$

then

$$(w_i, Ax_k) = \sum_{j=0}^{n-1} c_{k,j}(w_i, Aw_j) = c_{k,i}(w_i, Aw_i). \tag{3.16}$$

On the other hand, since $Ax_k = z_k$, this gives

$$c_{k,i} = \frac{(w_i, z_k)}{(w_i, Aw_i)}. \tag{3.17}$$

Since the terms on the right side above can be computed during the i^{th} iteration of the algorithm, the only substantial extra space that is needed is that for storing the partial sums x_k, which at the end of the i^{th} iteration equal

$$x_k = \sum_{j=0}^{i} c_{k,j} w_j. \tag{3.18}$$

Although the above analysis was based on the Lanczos algorithm, the derived complexity bounds also serve as rough approximations for the conjugate gradient (CG) method [9]. The CG and Lanczos algorithms are very similar. The iterations for the two algorithms are slightly different, but the operation count is almost exactly the same.

Both the Lanczos and the CG algorithms were implemented, Lanczos for solving equations modulo a large prime, CG for solving equations modulo 2. Both worked effectively, as will be described below. Both usually required n iterations to solve a system of size n. One unexpected problem was encountered, though. In the standard Lanczos and CG algorithms, it is not necessary that the matrix A in Equation 3.1 be non-singular. As long as w is in the linear span of the columns of A, the algorithms will converge. In the case of equations over finite fields, however, we observed that a system of less than full rank often gave rise to a self-conjugate vector, and thus to an abort of the algorithm. This phenomenon has not been

explored carefully. It was not very important in the cases where these algorithms were tried (computing discrete logarithms), since there the systems of linear equations are overdetermined. This issue might be much more important in the case of factoring algorithms, since in that case one needs to find many linear dependencies modulo 2.

The CG implementation uses auxiliary storage to carry out field operations fast. The field F is chosen to be $GF(2^r)$, with $r = 19, 20$, or 21. F is defined as polynomials of degree $\leq r-1$ over $GF(2)$ modulo a fixed irreducible polynomial of degree r. Elements $\alpha \in F$ are represented by a full word, $\overline{\alpha}$, with the digits in the binary representation of $\overline{\alpha}$ corresponding to the coefficients of α. This means that $\alpha + \beta$ is represented by the exclusive-or of $\overline{\alpha}$ and $\overline{\beta}$, and this operation is fast. Multiplication is carried out with the aid of an auxiliary array. Some primitive element $\gamma \in F$ is chosen, and then an array $w(j)$, $0 \leq j \leq 2^r - 1$ is generated, with $w(\overline{\alpha})$ for $\alpha \neq 0$ equal to the discrete logarithm of α to base γ. Thus for $\alpha \neq 0$

$$\alpha = \gamma^{w(\overline{\alpha})}, \quad 0 \leq w(\overline{\alpha}) \leq 2^r - 2. \tag{3.19}$$

For $\alpha = 0$, $w(\overline{\alpha}) = w(0) = 2^{r+1} - 3$. Another auxiliary array $t(j)$, $0 \leq j \leq 2^{r+2} - 6$, is formed, with

$$t(j) = \overline{\gamma^j}, \quad 0 \leq j \leq 2^{r+1} - 4, \tag{3.20}$$

$$t(j) = 0, \quad 2^{r+1} - 3 \leq j \leq 2^{r+2} - 6. \tag{3.21}$$

As a result,

$$\overline{\alpha\beta} = t(w(\overline{\alpha}) + w(\overline{\beta})) \tag{3.22}$$

for all $\alpha, \beta \in F$, and thus each product in F can be computed using one integer addition, which on the machine that was used takes about as much time as an exclusive-or. The total cost of a multiplication in F is still higher than that of addition, though, because of the extra table lookups. Given the large size of the tables that are required which will not fit in the cache memory, on many machines it is likely to be faster to implement an algorithm that uses more operations but smaller tables. In practice, the time devoted to multiplication of elements of F is so small that it does not matter what algorithm one uses.

The implementation of the Lanczos algorithm is less efficient than that of CG. The very portable, and reasonably fast, multiprecision code that A. Lenstra and M. Manasse distribute with their factoring package [12] was used. When the non-zero entry in the matrix was ± 1 or ± 2, addition or subtraction routines were invoked; approximately 90% of the non-zero entries in matrix B were $\pm 1, \pm 2$ in data set L. In the other rare cases,

multiprecision multiplication was invoked, even when it would have been more efficient to perform several additions.

The implementation of the Lanczos algorithm solved system L modulo a prime of 191 bits in 44 hours. The CG algorithm solved that system modulo 2 in 1.9 hours. Timings of about 100 iterations of the CG algorithm on system E indicate that this system would require about 190 hours to solve.

While quite large systems can be solved with the CG algorithm, the timings reported above are rather slow. For system L, Equation 3.13 indicates that the total cost ought to be around

$$7 \times 10^9 c_1 + 1.8 \times 10^8 c_2$$

If each of c_1 and c_2 were around 1 machine instruction, the total run time would be about 6 minutes. What causes the 1.9 hour run time is the fact that on the computer that was used c_1 and c_2 are much more expensive. This is due to problems of memory access, with the cache size too small to contain the full arrays. On different machines the performance would be quite different. Even on the SGI computer that was used, it is quite likely that one could obtain substantial speedups by arranging the data flow so that the caches are utilized more efficiently.

4. The Wiedemann algorithm

This algorithm is described carefully in [22]. As was pointed out by E. Kaltofen, the basic idea of this algorithm has been known in numerical analysis for a long time in Krylov subspace methods [23]. (It is rather interesting to note that the Lanczos algorithm is also a Krylov subspace method.) The main innovation in the Wiedemann algorithm is the use of the Berlekamp-Massey algorithm [15, 16], which in a finite field setting allows one to determine linear recurrence coefficients very fast, even for huge systems. Here we present only an outline of the method which will enable us to estimate its efficiency. Let us assume that we need to solve

$$Bx = u, \tag{4.1}$$

where B is a sparse non-singular $n \times n$ matrix. (See [22] for methods of dealing with non-square and singular matrices.) B is not required to be symmetric in this algorithm. Suppose that the minimal polynomial of B is given by

$$\sum_{j=0}^{N} c_j B^j = 0, \quad N \leq n, \quad c_0 \neq 0. \tag{4.2}$$

Then for any n-vector v, we have

$$\sum_{j=0}^{N} c_j B^{j+k} v = 0, \quad \forall k \geq 0. \tag{4.3}$$

If we let $v_k = B^k v$, then Equation 4.3 says that any one of the coordinates of $v_0, v_1, \ldots, v_{2n}$ satisfies the linear recurrence with coefficients $c_0, c_1, \ldots, c_N$. Given any $2n$ terms of a linear recurrent sequence of order $\leq n$, the Berlekamp-Massey algorithm will find the minimal polynomial of that sequence in $O(n^2)$ operations. (There are even faster variants of the algorithm [1, 2, 4] which are likely to be practical for very large systems.) Hence if we apply the Berlekamp-Massey algorithm to each of the first K coordinates of the vectors $v_0, \ldots, v_{2n}$, in $O(Kn^2)$ steps we will obtain the K polynomials whose least common multiple is likely to be the minimal polynomial of B. Once we find $c_0, \ldots, c_N$, we can obtain the solution to Equation 4.1 from

$$\begin{aligned} u &= -c_0^{-1} \sum_{j=1}^{N} c_j B^j u \\ &= B \left[-c_0^{-1} \sum_{j=1}^{N} c_j B^{j-1} u \right]. \end{aligned} \tag{4.4}$$

If B has b non-zero coefficients, most of them ± 1, and c_1 is again the cost of an addition or subtraction in the field we work with, then computation of $v_0, \ldots, v_{2n}$ costs about

$$a\ n\ b\ c_1. \tag{4.5}$$

This is just about the cost of the Lanczos and CG algorithms. Wiedemann's method saves a factor of 2 by not having to multiply by both B and B^T, but loses a factor of 2 by having to compute $B^j v$ for $0 \leq j \leq 2n$. (In the case of binary data, when all the non-zero coefficients are 1, the cost drops to nbc_1, a reduction similar to that in the Lanczos and CG algorithms.) The Berlekamp-Massey algorithm is expected to be very fast, with a cost of cn^2 for some small constant c, and this ought to be much less than Equation 4.5. Finally, obtaining u through Equation 4.4 will cost about

$$n\ b\ c_1\ +\ n^2\ c_2. \tag{4.6}$$

Thus if c_2 is not too large compared to c_1, or if the matrix is dense enough, the total cost of obtaining a solution using the Wiedemann algorithm is expected to be about 1.5 times as large as through the CG and Lanczos methods, even if $K = 1$ suffices to determine the minimal polynomial of B.

It is possible to cut down on the cost of the Wiedemann algorithm if one uses additional storage. If one uses $v = u$, and stores $v_0, \ldots, v_n$, then the computation of x in Equation 4.4 will only cost $O(n^2)$. In general, however, storing $v_0, \ldots, v_n$ in main memory is likely to be impossible, as that involves n^2 storage of n^2 field elements. On the other hand, since each of the v_j is only needed once during the computation of x, the v_j can be stored on a disk. If disk access is sufficiently rapid, this method could be used to avoid the additional cost, and thus make the Wiedemann method perform just about as fast as the CG and Lanczos algorithms.

Another way to eliminate the need for the extra n matrix-vector products in the Wiedemann algorithm (and thus reduce its cost so that it is no more than that of the CG and Lanczos methods) is to carry out the Berlekamp-Massey computation at the same time that the v_k are computed. At the cost of keeping around several additional vectors, this should allow one to construct the solution vector right away.

The assumption made above that taking $K = 1$ will suffice does not seem unreasonable for large fields. In the binary case, one can use an approach similar to that in the CG implementation described in Section 3 to generate as many sequences as the word size of the computer being used by taking the vector v to be over $GF(2^r)$. This approach would also make it possible to obtain several solutions at once (as in Equation 3.14), once the c_j are determined.

Most of the large systems that are likely to be solved in the near future are binary. In those cases, the discussion above implies that on a true random access machine, the Wiedemann algorithm is likely to be slower than CG or Lanczos by a factor of 3/2, and could approach their speed only by using substantial additional storage. However, on most machines data access is likely to be the main factor determining the efficiency of the algorithm. In the CG and Lanczos algorithms, the vectors w_i that are multiplied by A have to be on the order of 20 bits, and for all foreseeable problems longer than 16 bits. In the Wiedemann algorithm, it is conceivable that it would suffice to work with 3 or 4 bit vectors. (This is a point that needs testing.) Therefore it is possible that one could utilize the cache much more efficiently.

The general conclusion is that the Wiedemann algorithm is of about the same efficiency as the CG and Lanczos algorithms. However, it is quite a bit more complicated to program, and some crucial steps, such as the randomization procedures described in [22] for dealing with non-square and highly singular cases, have apparently never been tested. (Our analysis above assumes that they would not cause any problems.) It would be desirable to implement the Wiedemann algorithm and test it on some large systems.

5. Structured Gaussian elimination

This method is an adaptation and systematization of some of the standard techniques used in numerical analysis to minimize fill-in during Gaussian elimination, with some additional steps designed to take advantage of the special structure present in matrices arising from integer factorization and discrete logarithm algorithms. The part of the matrix corresponding to the very small primes is actually quite dense, while that corresponding to the large primes is extremely sparse. (In all cases that the authors have looked at, there are even variables corresponding to some large primes that do not appear in any equation.) This fact was taken advantage of in all previous solutions to large systems, in that Gaussian elimination was always performed starting at the sparse end. Had it been performed starting at the dense end, fill-in would have been immediately catastrophic.

By starting at the sparse end, substantial savings have been achieved. No precise measurements are available, but based on some data provided by R. Silverman (personal communication) it appears that about half the time was spent reducing n by n systems to about $n/3$ by $n/3$ systems, which were very dense. This indicates a factor of more than 10 savings over ordinary Gaussian elimination that starts at the dense end. A. Lenstra has indicated that similar results occurred in his work with M. Manasse.

The basic idea of structured Gaussian elimination is to declare some columns (those with the largest number of non-zero elements) as *heavy*, and to work only on preserving the sparsity of the remaining *light* columns. As was suggested by Pomerance and Smith [20], the set of heavy columns is allowed to grow as the algorithm progresses, instead of being chosen once, as was originally proposed [17]. In practice, the matrix would be stored in a sparse encoding, with rows represented by lists of positions where the coefficients are non-zero and with lists of the corresponding coefficients. To visualize the operation of the algorithm, it is easiest to think of the full matrix, though. The algorithm consists of a sequence of steps chosen from the following:

Step 1 Delete all columns that have a single non-zero coefficient and the rows in which those columns have non-zero coefficients.

Step 2 Declare some additional light columns to be heavy, choosing the heaviest ones.

Step 3 Delete some of the rows, selecting those which have the largest number of non-zero elements in the light columns.

Step 4 For any row which has only a single non-zero coefficient equal to ± 1 in the light

column, subtract appropriate multiples of that row from all other rows that have non-zero coefficients on that column so as to make those coefficients 0.

As long as only these steps are taken, the number of non-zero coefficients in the light part of the matrix will never increase. In the original description in [17], it was suggested that one might need to take further steps, involving subtracting multiples of rows that have ≥ 2 non-zero elements in the light part of the matrix. However, experiments (some already mentioned in [17]) suggest that this is not only unnecessary, but also leads to rapidly increasing storage and time requirements, and so it is better to avoid such steps.

Experiments mentioned in [17] used a pseudo-random number generator to create data sets that had the statistics of very large discrete logarithm problems over fields of characteristic 2. Those experiments indicated that structured Gaussian elimination ought to be very successful, and that to achieve big reductions in the size of the matrix that has to be solved, the original matrix ought to be kept very sparse, which has implications for the choices of parameters in factorization and discrete logarithm algorithms. Those experiments indicated that if the matrix was sparse enough (either because one started with a sparse data set, or else because enough columns were declared heavy), one could expect a very rapid collapse of the system to a much smaller one. The results of the experiments that we performed on real data confirm these findings. Furthermore, they show that excess equations are a very important factor in the performance of the algorithm. If there are many more equations than unknowns, one can obtain much smaller final systems.

Two versions of the program were implemented, one for solving equations modulo 2, the other for all other systems. (In the second version, coefficients were never reduced modulo anything.) Their performances on data set K were very similar, with the mod 2 version producing a slightly smaller final system. The general versions never produced coefficients larger than 40 in that case. This situation would probably change if the matrix were not so sparse.

The matrix is stored in a single linear array, with several smaller arrays being used to hold information about the status of rows and columns (whether a given column is heavy, for example). Each *active* row (i.e., row that has not been eliminated, was not dropped as unnecessary, and has some non-zero entries in the sparse part) is stored as two adjacent lists, one for the indices of columns in the sparse part of the matrix that have non-zero entries, and one for the indices of the rows of the original matrix that make up the current row. (In the case of the general version, there are also corresponding lists of the coefficients of the matrix entries and of the rows.) When row j is subtracted from row i, a new entry for the

modified row, still called i, is created at the end of the linear array, and the space previously occupied by rows i and j is freed up. When available space is exhausted, the large linear array is compacted by invoking a garbage collection routine. If this measure does not free up enough space, then Step 2 is invoked.

Many variations on the above approach are possible. Note that the number of non-zero entries in the light part of the matrix never increases. The only storage requirement that does grow is that for the lists of ancestor rows. Those, however, do not have to be kept in core. If one stores the history of what the algorithm does in a file, the ancestor lists can be reconstructed later. This is the approach used by Pomerance and Smith [20], for example, as well as by A. Lenstra and M. Manasse. Our implementation was motivated by the availability of substantial memory on our machine and the fact that when the ancestor lists get large, the heavy part of the matrix gets quite dense, which is undesirable, and so (as will be described later) it seems better to thin out the matrix by using Step 2.

One advantage of the algorithm as described here is that it can be implemented in very little space. Our implementation keeps all the arrays in core, and is very wasteful in that it uses full 32-bit words for all pointers and coefficients. Since only a modest number of passes through the data were performed, one could keep the rows stored on a disk, and use core storage only for the more frequently accessed arrays that store information about row and column sums.

In our implementation, Step 1 is applied repeatedly until no more columns of weight 1 (i.e., with a single non-zero coefficient) exist, then Step 2 is used. The number of columns that are declared heavy affects the performance of the algorithm to some extent. We usually applied this step to about $c/30$ columns, where c is the number of currently light columns that have weight > 0. For matrices that were expected to reduce to very small size, such as data set K and sets derived from it, values around $c/100$ were used. Generally speaking, the smaller the number of columns that are put into the heavy part at a single time, the better the final result, but also more work is required. (Pomerance and Smith [20] use values of about $c/1000$, for example.) The columns that are declared heavy are those of highest weight. Step 2 is followed by Step 4, which is applied repeatedly. When Step 4 cannot be applied any longer, Step 2 (followed by Step 4) is applied again, and so on. At a certain stage, when the number of heavy columns is a substantial fraction of the expected size of the final dense matrix, Step 3 (followed by Step 1) is invoked. The selection of the point at which to apply Step 3 is very important, and will be discussed later.

Very often, especially if the initial matrix is fairly dense, or there are not many more equations than unknowns, the final stages of structured Gaussian elimination produce rows

that have many ancestors, and so the heavy parts of those rows are quite dense. What was done to counteract this tendency was to first run the algorithm as described above, and then rerun it with two parameters c_1 and c_2 that were set based on the experience of the first run. When the number of heavy columns exceeded c_1, Step 2 was invoked so as to bring this number all the way up to c_2. After this application of Step 2, the sparse part of the matrix usually collapsed very quickly. The results of this step are described below and in Table 3.

The description above is not very precise. The reason is that the various elements of the algorithm can be, and often were, applied in different sequences and with different parameters. The output is fairly sensitive to the choices that are made. No clear guidelines exist as to what the optimal choices are, since it was hard to explore all the possible variations. However, even very suboptimal choices usually lead to small final systems.

The output of the structured Gaussian elimination program is a smaller matrix, which then has to be solved by another method. In our experience (primarily based on data set K), substitution of the results of solving the dense system into the original one gives values for almost all of the original variables in a small number of passes, each of which looks for equations in which only one variable is not yet determined.

Table 2: Structured Gaussian elimination performance – factoring data

Data Set	Number of Equations	Number of Unknowns	$\frac{\text{No. Equations}}{\text{No. Unknowns}}$	Size of Dense Matrix	Percent Reduction
A	35,987	35,000	1.03	9,222	73.7
B	52,254	50,001	1.05	12,003	76.0
C	65,518	65,500	1.00	17,251	73.7
D	123,019	106,121	1.16	12,700	88.0
E	82,470	80,015	1.03	36,810	54.0
$E1$	82,470	75,015	1.10	31,285	58.3
F	25,201	25,001	1.01	11,461	54.2
G	30,268	25,001	1.21	10,835	56.7
H	61,343	30,001	2.04	19,011	36.6
I	102,815	80,001	1.29	32,303	59.6
J	226,688	199,203	1.14	90,979	54.3

Structured Gaussian elimination was very successful on data set K, since it reduced it to set L very quickly (in about 20 minutes for reading the data, roughly the same amount

of time for the basic run, and then under an hour to produce the dense set of equations that form set L). It also worked well on the other systems. Table 2 summarizes the performance of structured Gaussian elimination on data sets A through J, and Table 4 does this for sets K,$K0,\ldots,K6$, and M. The size of the dense matrix is the number of unknowns in the reduced system. In each reduced data set, the number of equations exceeded the number of unknowns by 20.

Table 3: Density of heavy matrix resulting from structured Gaussian elimination

Data Set	c_1	c_2	Average Weight of Dense Row
B	–	–	$\geq 2,000$
B	8,000	16,000	456
B	6,000	16,000	486
B	6,000	20,000	149
E	–	–	6,620
E	20,000	50,000	260
E	20,000	60,000	115
$E1$	–	–	6,172
$E1$	20,000	35,000	1,366
$E1$	25,000	40,000	499
K	–	–	1,393
K	3,000	3,400	883
K	3,000	4,000	346
K	2,500	4,000	230
K	2,000	4,500	140
M	–	–	2,602
M	6,750	9,750	295
M	6,750	10,500	212
M	6,750	11,000	177

Obtaining a small set of equations is not satisfactory by itself in many cases, since the new system might be so dense as to be hard to solve. Table 3 presents some data on this point. For example, while set B was reduced to about 12,000 equations, the resulting set was very dense, with each row having on average $\geq 2,000$ non-zero entries. When the number of heavy columns was increased to 20,000 as soon as it exceeded 6,000, the

resulting dense matrix had only 149 non-zero entries per row. Similarly, for set E, the standard algorithm produces 36,810 equations, with 6,620 non-zeros each on average. If we only reduce the system to 60,000 equations, the resulting matrix has average row weight of only 115.

Table 3 also shows how the density of final systems derived from discrete logarithm data could be improved. Data set K was reduced to a system in 3,215 unknowns, but that system had, on average, 1,393 non-zero entries per equation. By invoking Step 2 early, before the number of heavy columns reached 3,215, less dense final systems were obtained. Increasing the number of unknowns to 4,500 as soon as 2,000 heavy columns are obtained reduced the density of the smaller system to only 140 non-zero entries per equation. For data set M, a similar order of magnitude decrease in the density of the smaller system was obtained with less than a factor of two increase in the number of unknowns.

Table 4: Structured Gaussian elimination performance – discrete logarithm data

Data Set	Number of Equations	Number of Unknowns	$\frac{\text{No. Equations}}{\text{No. Unknowns}}$	Size of Dense Matrix	Percent Reduction
K	288,017	96,321	2.99	3,215	96.7
$K0$	216,105	95,216	2.27	3,850	96.0
$K1$	165,245	93,540	1.77	4,625	95.1
$K2$	144,017	94,395	1.53	9,158	90.3
$K3$	144,432	92,344	1.56	5,534	94.0
$K4$	144,017	89,003	1.62	3,544	96.0
$K5$	115,659	90,019	1.28	6,251	93.1
$K6$	101,057	88,291	1.14	7,298	91.7
M	164,841	94,398	1.75	9,508	90.0

Table 4 presents the results of some experiments that show the influence of extra equations and variations in matrix density. All the runs were performed with the same algorithm on data sets derived from data set K and solved the system modulo 2. The performance of the algorithm on set K that is reported in Table 4 is better than that mentioned before (which reduced it to set L, which has 6,006 unknowns). This is partially due to working modulo 2, but mostly it results from use of somewhat different parameters in the algorithm, and not caring about the density of the final dense matrix.

The results for sets $K2$, $K3$, and $K4$ might seem very counterintuitive, since the densest set ($K4$) was reduced the most, while the sparsest one ($K2$), was reduced the least. This appears to be due to the fact that heavy rows tend to have few entries in the sparse part of the matrix. (This has been checked to be the case in the data, and is to be expected, since in the discrete logarithm scheme that was used to derive set K [10], equations come from factoring integers of roughly equal size, so that if there are many prime factors, few of them can be large.) Thus, by selecting the heaviest rows, one makes it easier for structured Gaussian elimination to take advantage of the extreme sparsity of the sparse end of the matrix.

The results obtained with set K may not be entirely characteristic of what one might encounter in other situations because this data set is much larger in the number of unknowns (as well as in the number of equations) than would be optimal for solving the discrete logarithm problem of [10]. If one selected only equations out of K that had roughly the 25,000 unknowns corresponding to the smallest primes and prime elements of smallest norms, set K would have yielded a solution to it. The existence of all the extraneous variables and equations may enable structured Gaussian elimination to yield a better result than it would obtain in more realistic situations.

The results for systems A to J were derived in a non-systematic manner; it is quite likely that much better results can be obtained by different choices of parameters. In the case of system K, and to some extent also systems $K0$ through $K6$, more extensive tests were performed. They were all done with a linear array of length 1.6×10^7 for storage, and with variations only in the applications of Steps 2 and 3. The number of columns to which Step 2 was applied did not seem to have a major influence on the size of the final matrix. On the other hand, the decision of when to apply Step 3 was very important. In all the experiments that were carried out, essentially all the excess rows were dropped at the same time; the effect of spreading out this procedure was not studied. It appeared that the best time to apply Step 3, if one is simply trying to minimize the size of the final system, is when the number of heavy columns reaches the size of the final matrix, since in that case the system tends to collapse very quickly after the application of Step 3. In practice, what this means is that one has to experiment with several different thresholds for when to apply Step 3. Since the generation of the dense equations usually takes several times (and when the dense system is large, many times) longer than structured Gaussian elimination, this does not effect the total running time very much.

On the basis of the experiments that have been performed so far, it appears that the best results are achieved if the point at which Step 3 is applied is chosen so that the steps that

follow reduce the matrix very rapidly, without any additional applications of Step 2. For example, the entry for set K in Table 4 was obtained by specifying that Step 3 be applied as soon as the number of heavy columns exceeded 3,200. The resulting matrix collapsed right afterwards. On the other hand, when the excess rows were deleted as soon as the number of heavy columns exceeded 3,150, the algorithm took a lot longer to terminate, resulted in a final matrix with 3,425 columns (but with the density of the final matrix lower than in the other case). At an extreme, set $K2$ (which corresponds to dropping 144,000 rows right at the beginning of the algorithm) resulted in a final matrix of size 9,158. In the case of systems with fewer excess equations (such as set A), some results indicate that it is preferable to apply Step 3 somewhat earlier.

The results reported here are very specific to our implementation. One of the essential features of the program was that it kept the lists of ancestor rows in core memory. Thus the algorithm was constrained most severely by the size of the big linear array, which essentially limited the total number of ancestor rows of the active rows. This had the desirable indirect benefit of producing a relatively sparse final matrix, but it undoubtedly made the algorithm behave quite differently than it would have otherwise. In particular, it must have skewed the comparisons between different data sets (since initially smaller data sets in effect could have more ancestor rows). It might be worthwhile to experiment with various variations of this method.

In the case of data set J (which came from the factorization of F_9), our version of structured Gaussian elimination reduced 199,203 equations to 90,979. A. Lenstra and M. Manasse used their version of the program to reduce set J to about 72,000 equations. Their program did not maintain ancestor lists, but the final matrix was almost completely dense, with about 36,000 non-zero coefficients per equation. Our program produced equations that on average had 7,000 non-zero coefficients. Looking at the output of the program, it appears that as soon as the number of heavy columns exceeded about 70,000, catastrophic collapse of the sparse part of the matrix began. The increase in the size of the heavy part was caused by space restrictions which bounded the size of the ancestor lists. In the F_9 case, since the final matrix was solved using ordinary Gaussian elimination, the decrease in the density of the final matrix that our program gave was probably not worth the increase in size. In other applications, especially when dealing with solving equations modulo large primes, and with sparser initial systems, our strategy is likely to be preferable.

The main conclusion that can be drawn from the results of these experiments is that sparse systems produce much smaller final systems than do denser ones. What is perhaps even more important, however, is that excess equations substantially improve the perfor-

mance of the algorithm. When one has access to a distributed network of machines with a lot of available computing time, but where solving the matrix might be a bottleneck (due to a need to perform the calculations on a single processor) one can simplify the task substantially by choosing a larger factor base and obtaining more equations. In extreme cases, when using the quadratic sieve, for example, and when only small machines are available, it might even be worthwhile not to use the two large primes variation of A. Lenstra and M. Manasse [13] (which in any case only appears to be useful for integers $> 10^{100}$), or possibly not even the old single large prime variation.

It appears that structured Gaussian elimination should be used as a preliminary step in all linear systems arising from integer factorization and discrete logarithm algorithms. It takes very little time to run, and produces smaller systems, in many cases dramatically smaller. For this method to work best, linear systems ought to be sparse, and, perhaps most important, there should be considerably more equations than unknowns. Producing the extra equations requires more effort, but it does simplify the linear algebra steps.

6. Acknowledgements

The authors thank E. Kaltofen, L. Kaufman, A. Lenstra, M. Manasse, C. Pomerance, and R. Silverman for their comments.

References

[1] G. S. Ammar and W. G. Gragg, Superfast solution of real positive definite Toeplitz systems, *SIAM J. Matrix Anal. Appl.*, **9** (1988), 61-76.

[2] R. R. Bitmead and B. D. O. Anderson, Asymptotically fast solution of Toeplitz and related systems of linear equations, *Lin. Alg. Appl.* **34** (1980), 103-116.

[3] I. F. Blake, R. Fuji-Hara, R. C. Mullin, and S. A. Vanstone, Computing logarithms in fields of characteristic two, *SIAM J. Alg. Disc. Methods* **5** (1984), 276-285.

[4] R. P. Brent, F. G. Gustavson, and D. Y. Y. Yun, Fast solution of Toeplitz systems of equations and computation of Padé approximants, *J. Algorithms* **1** (1980), 259-295.

[5] D. Coppersmith, Fast evaluation of discrete logarithms in fields of characteristic two, *IEEE Trans. on Information Theory* **30** (1984), 587-594.

[6] D. Coppersmith and J. H. Davenport, An application of factoring, *J. Symbolic Computation* **1** (1985), 241-243.

[7] D. Coppersmith, A. Odlyzko, and R. Schroeppel, Discrete logarithms in $GF(p)$, *Algorithmica* **1** (1986), 1-15.

[8] D. Coppersmith and S. Winograd, Matrix multiplication via arithmetic progressions, *Proc. 19th ACM Symp. Theory Comp.* (1987), 1-6.

[9] M. R. Hestenes and E. Stiefel, Methods of conjugate gradients for solving linear systems, *J. Res. Nat. Bureau of Standards* **49** (1952), 409-436.

[10] B. A. LaMacchia and A. M. Odlyzko, Computation of discrete logarithms in prime fields, *Designs, Codes, and Cryptography* **1** (1991), to appear.

[11] C. Lanczos, Solution of systems of linear equations by minimized iterations, *J. Res. Nat. Bureau of Standards* **49** (1952), 33-53.

[12] A. K. Lenstra and M. S. Manasse, Factoring by electronic mail, *Advances in Cryptology: Proceedings of Eurocrypt '89*, J.-J. Quisquater, ed., to be published.

[13] A. K. Lenstra and M. S. Manasse, Factoring with two large primes, *Advances in Cryptology: Proceedings of Eurocrypt '90*, I. Damgard, ed., to be published.

[14] K. S. McCurley, The discrete logarithm problem, in *Cryptography and Computational Number Theory*, C. Pomerance, ed., *Proc. Symp. Appl. Math.*, Amer. Math. Soc., 1990, to appear.

[15] J. L. Massey, Shift-register synthesis and BCH decoding, *IEEE Trans. Information Theory* **IT-15** (1969), 122-127.

[16] W. H. Mills, Continued fractions and linear recurrences, *Math. Comp.* **29** (1975), 173-180.

[17] A. M. Odlyzko, Discrete logarithms in finite fields and their cryptographic significance, *Advances in Cryptology: Proceedings of Eurocrypt '84*, T. Beth, N. Cot, I. Ingemarsson, eds., *Lecture Notes in Computer Science* **209**, Springer-Verlag, NY (1985), 224-314.

[18] C. Pomerance, Analysis and comparison of some integer factoring algorithms, *Computational Methods in Number Theory: Part 1*, H. W. Lenstra, Jr., and R. Tijdeman, eds., *Math. Centre Tract* **154** (1982), Math. Centre Amsterdam, 89-139.

[19] C. Pomerance, Factoring, in *Cryptography and Computational Number Theory*, C. Pomerance, ed., *Proc. Symp. Appl. Math.*, Amer. Math. Soc., 1990, to appear.

[20] C. Pomerance and J. W. Smith, Reduction of large, sparse matrices over a finite field via created catastrophes, manuscript in preparation.

[21] V. Strassen, Gaussian elimination is not optimal, *Numerische Math.* **13** (1969), 354-356.

[22] D. H. Wiedemann, Solving sparse linear equations over finite fields, *IEEE Trans. Information Theory* **IT-32** (1986), 54-62.

[23] J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Oxford Univ. Press, 1965.