Fast computation of linear generators for matrix sequences and application to the block Wiedemann algorithm

Emmanuel Thomé
Laboratoire d'Informatique (LIX)
École Polytechnique
91128 Palaiseau CEDEX
France
thome@lix.polytechnique.fr

ABSTRACT

In this paper we describe how the half-gcd algorithm can be adapted in order to speed up the sequential stage of Coppersmith's block Wiedemann algorithm for solving large sparse linear systems over any finite field. This very stage solves a sub-problem than can be seen as the computation of a linear generator for a matrix sequence. Our primary realm of interest is the field $\mathbb{F}_q$ for large prime power q. For the solution of a $N \times N$ system, the complexity of this sequential part drops from $O(N^2)$ to $O(\mathsf{M}(N)\log N)$ where $\mathsf{M}(d)$ is the cost for multiplying two polynomials of degree d. We discuss the implications of this improvement for the overall cost of the block Wiedemann algorithm and how its parameters should be chosen for best efficiency.

1. INTRODUCTION

Coppersmith's block Wiedemann algorithm [9] applies to the solving of large, sparse linear systems over finite fields. More precisely, a primary statement of the context of the algorithm would be as follows. We are given a large, sparse square matrix B of size $N \times N$ defined over $K = \mathbb{F}_q$, where q can be any prime power. We know that the matrix B is singular, and we are interested in computing one or several solutions of the equation:

$$Bw = 0. \qquad (1)$$

Solving such systems is a somewhat common task [16]. We originally encountered the problem in the course of solving discrete logarithm problems over $\mathbb{F}_{2^n}$ using Coppersmith's *index-calculus* algorithm from [7]. In its linear algebra stage, this algorithm requires finding a solution of a huge sparse linear system defined over $\mathbb{Z}/(2^n-1)\mathbb{Z}$ (in case this ring is not a field, we can infer the solution modulo 2^n-1 from the solution modulo each of its divisors via the Chinese remainder theorem if the factorization of 2^n-1 is known). Systems as in equation (1) appear more generally in any *index-calculus*-type algorithm for computing discrete logarithms in appropriate groups: see [19] and for instance [11, 12]. Huge matrices also appeared in the course of recent record-breaking factorizations of composite numbers [6, 4]. There, the linear system is defined over $\mathbb{F}_2$, which changes some of the issues.

The algorithm of Coppersmith is an extension of an algorithm proposed by Wiedemann in [26]. In Wiedemann's algorithm, we compute the sequence:

$$a_k = x^{\mathrm{T}} B^k y,\ 0 \le k \le 2N-1,$$

where x and y are fixed elements of the vector space K^N (again, K is the base field) acting as random inputs. If B has γ non-zero coefficients per line on average, we can compute all the a_k's using $O(\gamma N^2)$ scalar multiplications in K. This computation is faster if γ is small, that is, B is sparse. In counterpart, this evaluation is sequential by nature since it involves repeated applications of B. Doing this computation in a parallel or distributed setting is infeasible without a fairly huge amount of communication between the different processors or machines taking part to the computation. This first stage of the algorithm is followed by a second stage which computes the minimum generating polynomial for the sequence a_k. This polynomial yields a solution to equation (1). It can be obtained with the Berlekamp-Massey or Extended Euclidean algorithm. Both of these require $O(N^2)$ scalar multiplications, but subquadratic variants [1, 13] bring $O(\mathsf{M}(N)\log N)$ instead, where $\mathsf{M}(d)$ is the cost for multiplying polynomials of degree d. $\mathsf{M}(d)$ is $O(d\log d)$ with fast Fourier transform (FFT).

Coppersmith [9] brought the following interesting possibility: instead of vectors x and y, use *blocks* of vectors, of size $N \times m$ and $N \times n$, respectively, where m and n are chosen integers. One "sample" $x^{\mathrm{T}} B^k y$ therefore contains more information because it is made up of several scalars. This enables us to reduce the number of needed a_k's from $2N$ down to $L = \frac{N}{m} + \frac{N}{n} + \varepsilon$, where ε is a small additional term that shows up for technical reasons. For $m = n = 1$, this algorithm captures the original Wiedemann algorithm. This point of view makes it possible to distribute the computation among several machines, each of them computing for instance a given column of all the a_k's. This achieves coarse-grain parallelization of the computation of the a_k's. Coppersmith was primarily interested in the case of $\mathbb{F}_2$: an

n-bit machine can compute a whole line of $B^k y$ from $B^{k-1}y$ in one single operation, performing n binary multiplications (that is, bitwise ANDs) at a time.

The sequence of the a_k's is made up of $m \times n$ matrices. A linear generator for this sequence (that is, a $n \times 1$ column vector of polynomials as defined in 2.1) yields a solution to equation (1). The computation of such a linear generator can be done in a variety of ways. For this purpose, Coppersmith has designed an extended version of the Berlekamp-Massey algorithm, running with complexity $O((m+n)N^2)$. His algorithm is equivalent to the plain Berlekamp-Massey algorithm when $m = n = 1$.

We present a new algorithm for this task, adapted from the divide-and-conquer approach that yielded the HGCD (*half-gcd*) algorithm from Aho, Hopcroft and Ullman [1] and the PRSDC (*polynomial remainder sequences by divide-and-conquer*) algorithm from Gustavson and Yun [13]. Our algorithm has complexity $O(\frac{(m+n)^3}{n}\mathsf{M}(N)\log N)$, and if FFT is available over the base field, this reduces to $O(\frac{(m+n)^2}{n}N(m+n+\log N)\log N)$. Our algorithm is recursive, as HGCD or PRSDC. It can be seen as a plug-in replacement of Coppersmith's algorithm in the corresponding step of the block Wiedemann algorithm. Since both algorithms end up computing the same quantities, the analyses of the block Wiedemann algorithm by Kaltofen [14] and Villard [24, 23] apply indifferently. Therefore, the probability of success of the algorithm with respect to its random inputs x and y is not impaired by our enhancement: furthermore, this probability of success is comparable to the probabilities reached with the original Wiedemann algorithm or the Lanczos (standard or block version) algorithm with lookahead [8, 17, 22].

Section 2 addresses the problem of finding a linear generator for a given matrix sequence. Coppersmith's algorithm for this task is presented in 2.3. Existing subquadratic approaches and our new algorithm, are presented in section 3. Section 4 exposes the block Wiedemann algorithm, and shows how the computation of a linear generator for a matrix sequence yields a solution. In section 5, we discuss the overall cost of the block Wiedemann algorithm, along with the optimal value of its parameters m and n. Section 6 discusses practical concerns about the implementation of our approach.

2. LINEAR GENERATORS FOR MATRIX SEQUENCES

2.1 Definition

Throughout this section, we are given a $m \times n$ matrix of formal power series denoted $A(X)$. *Generators* for $A(X)$ are n-dimensional column vectors of polynomials. The *degree* of a column vector is defined as the maximum of the degrees in the n different entries of the vector. Let us introduce the following definition.

Definition 2.1. *Given two integers m and n, and a matrix $A(X) \in K[[X]]^{m\times n}$, $A(X)$ is linearly generated* up to degree L *by $u(X) \in K[X]^n$ if there exists $v(X) \in K[X]^m$ such that*

$$A(X)u(X) = v(X) + O(X^L), \text{ and } \deg v < \deg u.$$

Such a criterion can be checked using only the first L coefficients of $A(X)$. A generator up to any degree *has the same definition with $L = +\infty$.*

2.2 Usage in the block Wiedemann context

In the block Wiedemann algorithm, when trying to solve (1) for a $N \times N$ matrix, we will look for a linear generator for some $A(X) \in K[[X]]^{m\times n}$. As hinted at in the introduction, $A(X)$ will be of the form

$$A(X) = \sum_{k=0}^{\infty} a_k X^k \text{ where } a_k = x^{\mathrm{T}} B^k y,$$

x and y being respectively matrices of size $N \times m$ and $N \times n$. Villard [24, 23] bounds down (and away from zero) the probability of success of the algorithm with respect to the choice of x and y. The lower bound obtained is particularly close to 1 when the cardinality of the base field is large, or alternatively when the matrix is not too *pathological*: it is better not to have too many eigenvalues of high multiplicities (compared to m and n).

Since we want to focus on the linear generator computing task, which is the crux of the algorithm, we will briefly state the provided inputs and required outputs for this step. On input, it will be sufficient to have computed the first $L = \frac{N}{m} + \frac{N}{n} + \varepsilon$ terms of $A(X)$. ε can be increased to improve the probability of success. On output, we will need a generator $u(X)$ of lowest possible degree, which will be around $\frac{N}{n}$. Success will be almost certain as soon as $L - \deg u > \frac{N}{m}$, and the computed generator will also generate $A(X)$ up to any degree.

2.3 Coppersmith's algorithm

2.3.1 Framework

In this section, we focus on accomplishing the task defined in the paragraph above, that is, to find a linear generator for $A(X)$ of size $m \times n$, and degree $L - 1$, with $L = \frac{N}{m} + \frac{N}{n} + \varepsilon$. In [9], Coppersmith transposes the problem: he finds a left-hand generator for $A(X)^{\mathrm{T}}$. This is of course equivalent. We stick to the right-hand situation. Also, the algorithm we present here is valid only in the case where $m \geq n$. Validity of the algorithm depends on a non-degeneracy assumption on $A(X)$ that will appear in 2.3.2.

Rather than working with only one candidate $u(X)$ at a time for the linear generator of $A(X)$, we work with several of them at a time. Hence we have matrices: candidates for $u(X)$ are gathered in a matrix $f(X)$ in $K[X]^{n\times(m+n)}$, and candidates for $v(X)$ are gathered in $g(X)$ in $K[X]^{m\times(m+n)}$ (g is actually never needed in the computations, it only serves the presentation). Eventually, we will have $f(X)$ and $g(X)$ satisfy the following equation:

$$A(X)f(X) = g(X) + O(X^L), \qquad (2)$$

so that we hope that one or even several columns of f will bring us a generator. Aside from this one-liner equation, we introduce a quantity δ_j associated with each column j, $1 \leq j \leq m+n$. This quantity is called by Coppersmith the *nominal degree*, as this is actually an upper bound on the degree of the coefficients in the j-th column of the polynomial

$f(X)$. The algorithm proceeds step by step, increasing a counter t, and satisfying the following equation throughout:

$$A(X)f(X) = g(X) + X^t e(X), \qquad (\mathcal{E})$$

where $e(X) \in K[X]^{m\times(m+n)}$ is the current "error" that we aim at cancelling. Each particular column of this equation will satisfy:

$$\begin{gathered} A(X)f_j(X) = g_j(X) + X^t e_j(X), \qquad \text{(C1)} \\ \deg f_j \le \delta_j, \quad \deg g_j < \delta_j, \quad \deg e_j \le L + \delta_j - t, \end{gathered}$$

(throughout, the subscript $_j$ denotes a column). We will also enforce another condition, $[X^k]P$ denoting the coefficient in X^k of the polynomial P:

$$\operatorname{rank}([X^0]e) = m. \qquad \text{(C2)}$$

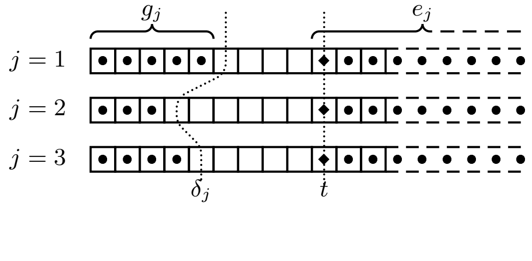
Throughout the description of this algorithm, the superscript $^{(t)}$ may be used to stress the fact that we are concerned with the value at round t. It will be omitted in most cases however, since it is generally obvious.

2.3.2 Initialization

We start the algorithm at $t_0 = \lceil \frac{m}{n} \rceil$. All the δ_j's are initially set to t_0. The first m columns of f are filled at random up to degree $t_0 - 1$. If we look at the first m columns of the product $[X^{t_0}](Af)$, that is, the $[X^{t_0}](Af_j)$'s for $1 \le j \le m$, we see that each of them is a random linear combination of the columns of $[X^1]A, \dots, [X^{t_0}]A$. Since each $[X^k]A$ has rank n at most, the maximal rank of the space spanned by the $[X^{t_0}](Af_j)$'s is nt_0. We repeat trials until we obtain that these m columns are independent, that is, the space they span is of dimension m. This is precisely made possible by setting $t_0 \ge \frac{m}{n}$. If this fails, we can try to increase t_0. Otherwise, the algorithm fails as we are unable to initialize the iterative procedure. $A(X)$ would have to be very degenerate for this to happen, so that this situation is considered unlikely. The remaining $n \times n$ submatrix of f is set to $X^{t_0}I$, where I is the $n \times n$ identity matrix. One easily checks that this initialization ensures that both conditions (C1) and (C2) hold.

2.3.3 Description of the iteration

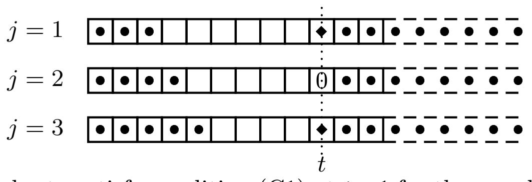
Our goal is to make the gap between the δ_j's and t bigger and bigger, until the condition demanded in 2.2 is satisfied for some j, that is $t - \delta_j > \frac{N}{m}$. In order to help the presentation, we depict the *degree pattern* of each of the columns of the product $A(X)f(X)$. Each column is represented by a *line* of boxes indexed from 0 to $+\infty$. Box number k contains no symbol (or "0") if we know for sure that all the entries in the column of $A(X)f(X)$ have their coefficient of degree k equal to zero. Otherwise, the box contains a black bullet, for instance. So let us imagine that we have the following degree patterns for the different columns of $A(X)f(X)$, at a given round t (the sketches are merely explanatory, they do not pretend to reflect a true situation).



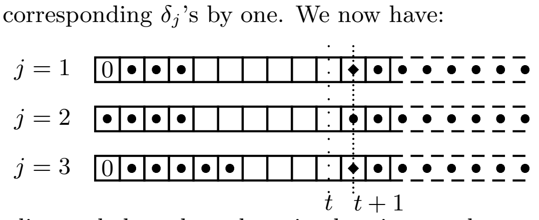
Trying to go from t to $t+1$, we aim at cancelling the coefficients denoted by diamonds, that is, $[X^t](Af)$, or $[X^0]e$. This is done through a kind of gaussian elimination, which has to obey the further requirement that the δ_j's must not be trashed. We are allowed the following operations on columns:

- Exchange columns.
- Add a multiple of column j_1 to column j_2 provided that $\delta_{j_1} \le \delta_{j_2}$.
- Multiply a column by X (this shifts the degree pattern one box to the right).

First, we sort the columns so that all the δ_j's are now in increasing order. Then we do gaussian elimination as usual on the columns, only adding to a column another column of lower index. This way, we zero out all but m columns of $[X^0]e$, the remaining ones being linearly independent because of condition (C2). Condition (C1) now holds at $t+1$ for all but these columns. On the degree pattern picture, we have changed all but m of the diamonds into zeroes, without changing the rest of the picture (save the reordering):



In order to satisfy condition (C1) at $t+1$ for the m columns for which it still doesn't hold, we multiply them by X. This shifts the degree pattern pictures to the right, and increases the corresponding δ_j's by one. We now have:



The diamonds have been kept in the picture, they are now part of $[X^{t+1}](Af)$. Since these are m independent columns, this ensures that condition (C2) holds at round $t+1$. All the work we have done can be expressed by the multiplication on the right of equation $\mathcal{E}$ by a $(m+n)\times(m+n)$ matrix that we denote $P^{(t)}$. Since the relevant input to this iteration consists of only $[X^t](Af)$ and $(\delta_1, \dots, \delta_{m+n})$, we have given a constructive proof to the theorem below. The C-like pseudocode 2.1 summarizes the procedure.

THEOREM 2.2. *If conditions* (C1) *and* (C2) *hold at round t, there is an algorithm* ALGO1 *that, knowing $[X^0]e^{(t)}$ and $(\delta_1^{(t)}, \dots, \delta_{m+n}^{(t)})$, computes a $(m+n)\times(m+n)$ matrix $P^{(t)}$ along with integers $(\delta_1^{(t+1)}, \dots, \delta_{m+n}^{(t+1)})$ such that:*

$$\begin{aligned} f^{(t+1)} &= f^{(t)}P^{(t)}, \\ g^{(t+1)} &= g^{(t)}P^{(t)}, \\ e^{(t+1)} &= e^{(t)}P^{(t)}\frac{1}{X}, \end{aligned}$$

and the $\delta_j^{(t+1)}$'s satisfy conditions (C1) *and* (C2) *at round $t+1$. Furthermore, we have $\sum_j \delta_j^{(t+1)} - \sum_j \delta_j^{(t)} = m$.*

```
Algorithm ALGO1
INPUT:  A $m \times (m+n)$ matrix $[X^0]e$,
        $m+n$ integers $\delta_1, \ldots, \delta_{m+n}$.
OUTPUT: $P$ as defined by theorem 2.2.
{
    $P$=$I_{m+n}$;                /* the identity matrix */
    Reorder columns so that $\delta_1 \leq \ldots \leq \delta_{m+n}$,
    updating $P$ accordingly;

    /* Gaussian elimination */
    for($j_0$=1;$j_0$<=$(m+n)$;$j_0$++) busy[$j_0$]=0;
    for($i$=1;$i$<=$m$;$i$++) {
        for($j_0$=1;$j_0$<=$(m+n)$;$j_0$++)
            if ($[X^0]e_{i,j_0}$!=0 && busy[$j_0$]==0) break;
        busy[$j_0$]=1;      /* $[X^0]e_{i,j_0}$ is the pivot */
        for($j$=$j_0+1$;$j$<=$(m+n)$;$j$++) {
            $\lambda$=$[X^0]e_{i,j}/[X^0]e_{i,j_0}$;
            /* subscripts denote columns */
            $[X^0]e_j$=$[X^0]e_j - \lambda[X^0]e_{j_0}$;
            $P_j$=$P_j - \lambda P_{j_0}$;
        }
    }

    /* Cancel the remaining columns */
    for($j$=1;$j$<=$(m+n)$;$j$++)
        if (busy[$j$]==1) $P_j$=X$P_j$;
    return $P$;
}
```

Program 2.1: Algorithm for computing $P^{(t)}$

2.3.4 Termination

Now let us consider the *average* value, say $\overline{\delta}$ of the δ_j's. It increases by $\frac{m}{m+n}$ each time t increases by 1. Therefore, the difference $t - \overline{\delta}$ obeys:

$$t - \overline{\delta} = t - (t_0 + (t - t_0)\frac{m}{m+n}) = (t - t_0)\frac{n}{m+n}.$$

We can see that for $t > \frac{N}{m} + \frac{N}{n} + t_0$, the difference exceeds $\frac{N}{m}$, so there exists at least one j for which $t - \delta_j > \frac{N}{m}$. The condition demanded in 2.2 is therefore satisfied by the column $f_j(X)$, and we expect it to yield a solution to our linear system.

Actual termination can be detected another way. We saw in 2.2 that the desired $f_j(X)$ is expected to be a generator up to any degree. If one of the columns of $f(X)$ becomes so at some round t, we will notice that the corresponding column of $[X^0]e$ is zero at the beginning of round $t+1$. If such a column is zero, then the corresponding f_j carries over untouched to the next round. If on several successive rounds, the column of $[X^0]e$ associated to a given column of f is zero, then f_j is likely to be a linear generator for $A(X)$ up to any degree, so this is probably the quantity we are looking for. In order to check for termination this way, we need to have L a bit above $\frac{N}{m} + \frac{N}{n}$. Coppersmith writes $L = \frac{N}{m} + \frac{N}{n} + O(1)$, gathering this margin in the $O(1)$ term that is not thoroughly investigated. He rather relies on the fact that the algorithm practically does produce a solution soon after t exceeds $\frac{N}{m} + \frac{N}{n}$. We stick to that approach, since the analysis of correctness of the block Wiedemann algorithm if far beyond our scope here, and has already been addressed in [14, 24, 23].

As a side note, as long as we have not reached the case where $(t - \delta_j) > \frac{N}{m}$ somewhere, all the δ_j's are expected to be (almost) equal to $\overline{\delta}$. Therefore, we might expect that when the average difference exceeds $\frac{N}{m}$, as many as n columns are candidates for producing a solution (but this can be less).

3. SUBQUADRATIC APPROACHES

3.1 Existing algorithms

Many algorithms exist for computing linear generators for matrix sequences. Coppersmith's, like most of these, has a quadratic complexity. Recall that we are looking for a generator up to degree $L = \frac{N}{m} + \frac{N}{n} + \varepsilon$. If we detail the complexity in terms of N, m and n, we obtain $O((m+n)N^2)$. Other, older, algorithms have sometimes a subquadratic version. These are the "power Hermite Padé solver" by Beckermann and Labahn [2] in $O(\frac{(m+n)^2 m}{n}\mathsf{M}(N)\log N)$, and the algorithm of Bitmead, Anderson and Morf [3, 18] for Toeplitz-like matrices, in $O((m+n)^2\mathsf{M}(N)\log N)$. While these algorithms are asymptotically faster than Coppersmith's, they do not seem to have been tried for large experiments with the block Wiedemann algorithm, like [15]. Apart from the relative simplicity of Coppersmith's approach compared to other algorithms, two points might explain why this quadratic algorithm can be seen as having the edge on subquadratic alternatives. First, it requires no randomization, whereas this is needed for the method of Bitmead, Anderson and Morf. Second, the "big O" in the complexity $O((m+n)N^2)$ hides no big constant. Counting the exact number of operations required, Coppersmith's algorithm requires no more than $\frac{m+n}{2}N^2$ scalar multiplications (see [9]). On the other hand, subquadratic algorithms always require the use of fast polynomial arithmetic using FFT. Added to the increased complexity in terms of m and n, it turns up that the speedup obtained from this algorithms is probably too small for realistic examples.

In the following subsection, we give another try, providing a subquadratic version of Coppersmith's algorithm. It shares with the original algorithm the absence of need for randomization, as well as the lower complexity in terms of m and n compared to other existing algorithms, provided that m and n remain small, say $O(\log N)$: the complexity of our algorithm is $O(\frac{(m+n)^3}{n}\mathsf{M}(N)\log N)$, and this lowers down to $\frac{4(m+n)^2}{n}N\log^2 N + \frac{3(m+n)^3}{n}N\log N$ scalar multiplications in a ring that supports FFT arithmetic (log denotes the logarithm in base 2). For small m and n (compared to $\log N$), this is asymptotically better. We will show the importance of this in section 5.

3.2 An accelerated version of Coppersmith's algorithm

In Coppersmith's algorithm, the quadratic cost comes from the evaluation of $[X^t](Af)$ at each round t, for $t_0 \leq t \leq L$. Our divide-and-conquer approach aims at replacing these numerous coefficient computations by a few big polynomial multiplications, in order to take advantage of fast multiplication algorithms, like FFT. In order to do this, we make an extensive use of theorem 2.2. Specifically, the fact that the only knowledge of $[X^t](Af)$ — that is, $[X^0]e$ — is necessary will prove to be crucial. In fact, if one knows the first k coefficients of $e^{(t)}(X)$, this information is enough to compute $P^{(t)}$ up to $P^{(t+k-1)}$, without updating $f(X)$. Let us formalize these considerations.

Definition 3.1. *A* k-context *is a pair of the form* $E = (e(X), \Delta)$ *corresponding to some stage of the iterative algorithm explained in section 2.3 where* $\Delta = (\delta_j^{(t)})_j$ *and* $e(X)$ *are known up to degree* $k-1$*. A* context *without precision is the same pair with full knowledge of* $e(X)$*.*

Definition 3.2. *Generalizing the definition from 2.3.3, if* E *is a context corresponding to round number* t *of the algorithm in 2.3, we call* $\pi_E^{(a,b)}$ *the following* $(m+n)\times(m+n)$ *matrix:*

$$\pi_E^{(a,b)} = P^{(t+a)} \dots P^{(t+b-1)},$$

where the $P^{(t+s)}$ *are the matrices computed from step 2.3.3 at the corresponding rounds after* t*.*

Theorem 3.3. *A given* k*-context* E *determines completely any* $\pi_E^{(a,b)}$ *as long as* $a \le b \le k$*. If* E *corresponds to round* t *of the algorithm, say* $E = E^{(t)}$*, then a* $(k-b)$*-context* $E^{(t+b)}$ *follows from the computation of* $\pi_E^{(0,b)}$*.*

Proof The proof is easy by induction. It is enough to observe that at round t, the computed matrix $P^{(t)}$ is such that $e^{(t)}P^{(t)} \equiv 0\ [X]$. Then, $e^{(t+1)}$ is $e^{(t)}P^{(t)}\frac{1}{X}$. $\Delta^{(t+1)}$ follows also from $P^{(t)}$ since one checks easily that:

$$\delta_j^{(t+1)} = \max_i \{\delta_i^{(t)} + \deg P_{i,j}^{(t)}\}.$$

By an abuse of notation, we denote the latter $\Delta^{(t)}P^{(t)}$. Together, $e^{(t)}P^{(t)}\frac{1}{X}$ and $\Delta^{(t)}P^{(t)}$ form a $(k-1)$-context. Generalization of this step from t to $t+1$ to the result of the theorem is trivial. □

With this formalism, it becomes clear that our main point of interest is the quantity $\pi_{E^{(t_0)}}^{(0,L-t_0)}$ where $E^{(t_0)} = (e^{(t_0)}, \Delta^{(t_0)})$ is the initial context. Once $\pi_{E^{(t_0)}}^{(0,L-t_0)}$ is known, then all the columns of $f^{(t_0)}(X)\pi_{E^{(t_0)}}^{(0,L-t_0)}$ satisfy the equation $\mathcal{E}$ with $t = L$, and since we know the δ_j's, we can pick a column that suits the requirements of subsection 2.2.

From theorem 3.3, we design an algorithm whose task is the computation of $\pi_E^{(0,b)}$ from a given b-context E. It is described in figure 3.1. In that piece of pseudo-code, ALGO1 is the algorithm in 2.1. The recursive algorithm is named MSLGDC from "matrix sequences linear generator by divide-and-conquer". It will be applied to the $(L-t_0)$-context $E^{(t_0)} = (e^{(t_0)}, \Delta^{(t_0)})$.

3.3 Complexity of MSLGDC

Our algorithm requires two non-trivial (more than linear) operations at each recursion level. These are:

$$e_M \leftarrow (e\pi_L \mod X^b) \text{ div } X^{\lfloor \frac{b}{2} \rfloor}, \text{ and } \pi \leftarrow \pi_L\pi_R.$$

```
Algorithm MSLGDC
INPUT: A $b$-context $E = (e, \Delta)$.
OUTPUT: $\pi_E^{(0,b)}$.
{
    if $b$==0 return $I_{m+n}$;

    $(e_L, \Delta_L)$=$(e \mod X^{\lfloor \frac{b}{2} \rfloor}, \Delta)$;
    $\pi_L$=MSLGDC($(e_L, \Delta_L)$);
    $(e_M, \Delta_M)$=$(e\pi_L$ mod $X^b$ div $X^{\lfloor \frac{b}{2} \rfloor}, \Delta\pi_L)$;

    $P$=ALGO1($(e_M, \Delta_M)$);
    $\pi_L$=$\pi_L P$;

    if ($b > (\lfloor \frac{b}{2} \rfloor + 1)$) /* i.e.  $b > 2$ */ {
        $(e_R, \Delta_R)$=$(e_M P$ mod $X^{\lceil \frac{b}{2} \rceil}$ div $X, \Delta_M P)$;
        $\pi_R$ = MSLGDC($(e_R, \Delta_R)$);
        $\pi$ = $\pi_L \times \pi_R$;
        return $\pi$;
    } else return $\pi_L$;
}
```

Program 3.1: Recursive algorithm for computing $\pi_E^{(0,b)}$

Of these polynomials, e has degree b, and π_L and π_R have degree $\frac{m}{m+n}\frac{b}{2}$. Using a generic multiplication algorithm requiring $\mathsf{M}(d)$ operations to multiply two polynomials of degree d, these operations would cost $m(m+n)^2\mathsf{M}(b)$ and $(m+n)^3\mathsf{M}(\frac{m}{m+n}\frac{b}{2})$, that is, at most $\frac{3}{2}m(m+n)^2\mathsf{M}(b)$. Now, if we use fast Fourier transform (FFT), we can do much better:

Theorem 3.4. *If* K *supports FFT (see* [10, chapter 8]*), the two operations above can be achieved in time* $4\mathsf{M}_1 m(m+n)b\log b + 3\mathsf{M}_1 m(m+n)^2 b + O((m+n)^2 b)$*, where* M_1 *is the cost of a multiplication in* K*. This yields a complexity bound for algorithm* MSLGDC *with a* b*-context of* $4\mathsf{M}_1 m(m+n)b\log^2 b + 3\mathsf{M}_1 m(m+n)^2 b\log b + O((m+n)^2 b\log b))$*.*

Proof What this theorem says is that the generic result is not only specified using the complexity of FFT for $\mathsf{M}(d)$, but that we also improve the complexity with respect to m and n. Let us show how this is obtained.

We refer to [10, chapter 8] for an introduction to fast Fourier transform. In a few words, FFT relies on the one hand on the ability to efficiently compute the evaluation (discrete Fourier transform, DFT) of a polynomial at a bunch of points — the 2^k-th roots of unity for some k — and on the other hand on the ability to interpolate equally fast a polynomial given its values at those same points (inverse DFT, or IDFT, operation). The DFTs of two polynomials can be multiplied pointwise (at a linear cost in the number of points) to obtain the DFT of the product polynomial. The latter can then be recovered by an IDFT operation.

Here, our goal is to compute products of polynomial matrices. This is a simple generalization of the principle above: one computes the DFT of each entry in the matrices involved (e, π_L, and π_R). These form the matrix DFTs $\widehat{e}$, $\widehat{\pi_L}$, and $\widehat{\pi_R}$ (these are matrices of scalar sequences, or also sequences of scalar matrices). These DFTs can be multiplied pointwise, involving one scalar matrix multiplication per point, to obtain the (matrix) DFTs of the products: $\widehat{e_M}$, and $\widehat{\pi}$.

The number of points at which the DFTs are computed is actually driven by the number d of unknown coefficients in the product: we take the smallest power of 2 above d, so in any case less than $2d$. It follows from [10, thm 8.15] that the cost of the computation of the DFTs is below $d \log d$ multiplications in K (same for IDFTs). The degrees considered imply that we need transforms using b points at most for the computation $e_M \leftarrow (e\pi_L \mod X^b) \mathrm{div} X^{\lfloor \frac{b}{2} \rfloor}$, and transforms using $\frac{2m}{m+n}b$ points at most for the computation of $\pi \leftarrow \pi_L \pi_R$. Upper bounds on the time required to compute all the transforms are summarized hereafter (of course, the transform of π_L needs not to be computed twice, so we keep the largest figure).

$$\begin{array}{lll}
\text{DFT :} & e \to \widehat{e} & \mathsf{M}_1 m(m+n)\frac{b}{2}\log\frac{b}{2}, \\
\text{DFT :} & \pi_L \to \widehat{\pi_L} & \mathsf{M}_1 (m+n)^2 \frac{m}{m+n} b \log(\frac{m}{m+n}b), \\
\text{IDFT :} & \widehat{e_M} \to e_M & \mathsf{M}_1 m(m+n)\frac{b}{2}\log\frac{b}{2}, \\
\text{DFT :} & \pi_R \to \widehat{\pi_R} & \mathsf{M}_1 (m+n)^2 \frac{m}{m+n} b \log(\frac{m}{m+n}b), \\
\text{IDFT :} & \widehat{\pi} \to \pi & \mathsf{M}_1 (m+n)^2 \frac{m}{m+n} b \log(\frac{m}{m+n}b).
\end{array}$$

Additionally, the matrix products involved by the pointwise multiplication of the DFTs yield a complexity of $m(m+n)^2 b\mathsf{M}_1$ and $2m(m+n)^2 b\mathsf{M}_1$ operations, respectively. Summing these up, we obtain the announced results. The cost equation for the algorithm MSLGDC for order b is:

$$\begin{aligned} C(b) = & 2C\left(\frac{b}{2}\right) + 4\mathsf{M}_1 m(m+n) b \log b + \\ & 3\mathsf{M}_1 m(m+n)^2 b + O((m+n)^2 b), \end{aligned}$$

hence $C(b) = 4\mathsf{M}_1 m(m+n) b \log^2 b + 3\mathsf{M}_1 m(m+n)^2 b \log b + O((m+n)^2 b \log b)$, as claimed. □

Plugging into this the parameters used for the block Wiedemann algorithm, namely $L = \frac{m+n}{mn}N + O(1)$, we obtain:

$$C(L) = 4\mathsf{M}_1 \frac{(m+n)^2}{n} N \log^2 N + O((m+n)^3 N \log N).$$

So the actual* speedup obtained when we compare to Coppersmith's version is $\dfrac{nN}{8(m+n)\log^2 N}$, as long as m and n stay relatively small.

4. BLOCK WIEDEMANN ALGORITHM

Now that we have extensively discussed the available tools for finding linear generators of matrix sequences, we will see how they plug into the block Wiedemann algorithm. As told before, we are now interested in the polynomial matrix:

$$A(X) = \sum_{k=0}^{L-1} a_k X^k, \text{ with } a_k = x^{\mathrm{T}} B^k y,$$

where x and y are respectively $N \times m$ and $N \times n$ matrices. The coefficients a_k of $A(X)$ are $m \times n$ matrices. Not surprisingly, we look for a linear generator of these in order to obtain a solution. The number L of computed coefficients of $A(X)$ is here $\frac{N}{m} + \frac{N}{n} + \varepsilon$. In other words, we stick to the pragmatic approach that works well in practice: compute $A(X)$ up to far enough so that heuristics let us believe that we will obtain a solution. In order to ensure that a solution will be produced, it is necessary to actually set y to Bz, where z is a random vector block. From now on, our context will be the one described in this paragraph. The quantities B, N, L, m, n, x, z, and y will be fixed, corresponding to the aforementioned.

*However, so many parameters are involved that this estimate is not really sharp.

The computation of the coefficients of $A(X)$, which will be named "stage BW1", is done sequentially. A vector variable Y is repeatedly updated by $Y \leftarrow BY$, and dot products $x^{\mathrm{T}}Y$ are computed at each step. Once we have $A(X)$ at our disposal, we can infer a linear generator for this matrix sequence, using the tools we have already mentioned (for example, we can use the MSLGDC algorithm). This will be the step BW2 of the block Wiedemann algorithm. We quickly show that such a linear generator yields a solution to the system $Bw = 0$ with high probability. Let us first recall the condition (C1) that is satisfied at each round by the different columns of the relevant matrices:

$$A(X)f_j(X) = g_j(X) + X^t e_j(X), \qquad \text{(C1)}$$
$$\deg f_j \leq \delta_j, \quad \deg g_j < \delta_j, \quad \deg e_j \leq L + \delta_j - t.$$

We detail now why the context given in subsection 2.2 is appropriate in order to find a solution of the equation $Bw = 0$. At a given round t, considering the quantities in the equation above, the product $A(X)f_j(X)$ has zero coefficients for degrees δ_j up to $t-1$. In other words, we have:

$$\forall s,\ 0 \leq s < t - \delta_j, \quad \sum_{k=0}^{\delta_j}([X^{s+\delta_j-k}]A)([X^k]f_j) = 0,$$

$$\text{i.e. } \sum_{k=0}^{\delta_j} x^{\mathrm{T}} B^{s+\delta_j-k} y [X^k] f_j = x^{\mathrm{T}} B^s \sum_{k=0}^{\delta_j} B^{\delta_j-k} y [X^k] f_j = 0.$$

Then, the quantity $\sum_{k=0}^{\delta_j} B^{\delta_j-k} y [X^k] f_j$ is orthogonal to as many as $m(t-\delta_j)$ vectors: the vectors $(B^{\mathrm{T}})^s x_i$ where the x_i's are the columns of x. Eventually, namely when $(t-\delta_j) > \frac{N}{m}$, $m(t-\delta_j)$ exceeds N. If the space spanned by the $(B^{\mathrm{T}})^s x_i$ has maximal dimension (this is our heuristic argument that we hope will hold), our quantity is then zero. If we replace y by Bz in the previous equation, we obtain:

$$B \sum_{k=0}^{\delta_j} B^{\delta_j-k} z([X^k]f_j) = 0.$$

If this equation holds, $w = \sum_{k=0}^{\delta_j} B^{\delta_j-k} z([X^k]f_j)$ is a solution if it is non-zero. In case it is zero but as a "polynomial" in B, w has a non-zero valuation ν, then we have $B^{1+\nu}\hat{w} = 0$ for $\hat{w} = \sum_{k=0}^{\delta_j-\nu} B^{\delta_j-k} z([X^k]f)$, and some $B^s\hat{w}$ is guaranteed to be a solution if $\hat{w} \neq 0$. The computation of $\hat{w}$ and s such that $B^s\hat{w}$ is a solution is referred to as step BW3.

5. COMPLEXITY ANALYSIS AND OPTIMIZATION

Having a block version of the Wiedemann algorithm introduces a new flexibility: we can play with parameters m and n. Nevertheless, these parameters do have some optimal value that we'd better use: obviouly, the bigger m and n, the shorter the computation of the a_k's, but also the more tedious the computation of a solution from these. We will therefore detail the complexity of the different stages (BW1, BW2, BW3) of the algorithm with respect to m, n, and N. For step BW2, we will give complexities for both Coppersmith's and our algorithm.

The block approach allows coarse grain parallelization (see [9] or [15]). In a parallel or distributed setting, distributing the columns of a vector block Y across several machines allows to compute $Y \leftarrow BY$ in a real time that does not depend on n (if we have that many machines available). Steps BW1 and BW3 can take advantage of this, and therefore the real time is the appropriate measure for the algorithm. Now the question is, provided that the hardware we have access to allows us several values for m and n, how to choose them in order to achieve the lowest total real time ? This question is answered in theorems 5.3 and 5.4. Since m and n are typically limited by the available hardware, it is reasonable to assume that m and n are bounded by a constant. Therefore, at least for the complexity of step BW2 using our recursive algorithm, we will incorporate this in the complexity equation, and focus on the dominating term.

In order to obtain complexity measurements we introduce two important constants, M_0 and M_1. M_0 stands for the time needed for multiplying a coefficient of the matrix B (typically of size equal to one machine word) by an element of K, while M_1 is the time needed for multiplying together two elements of K. Also, we denote by γ the average number of non-zero entries of rows of B (B is expected to be sparse, so γ is small). We do not take additions into account in our analysis. This is an excessive simplification over small fields like $\mathbb{F}_2$, but reasonable over larger fields. We prove the following results:

Theorem 5.1. *The different steps of the block Wiedemann algorithm require the following real time:*

BW1 $(\gamma\mathsf{M}_0 + m\mathsf{M}_1)\frac{m+n}{mn}N^2$ *(see also remark 5.2 below).*

BW2 $\mathsf{M}_1\frac{m+n}{2}N^2 + O(N)$ *using Coppersmith's algorithm*
$4\mathsf{M}_1\frac{(m+n)^2}{n}N\log^2 N + O(N\log N)$ *using our algorithm (provided that m and n are bounded).*

BW3 $\gamma\mathsf{M}_0\frac{1}{n}N^2$.

Proof As said before, step BW1 is accomplished by repeating the operation $Y \leftarrow BY$, where $Y = y$ initially. This sums up as nL matrix times vector product, but since the n columns of Y are assumed to be treated separately, the real time is the time needed for L applications of B: $\gamma N\mathsf{M}_0 L$. Furthermore, we have to add the cost of the dot products $({x_i}^{\mathrm{T}}Y_j)$. These cost $m\mathsf{M}_1 N$ at each step, which brings the result (see also remark 5.2 below). The second result follows. As for step BW2, the result follows from [9] for Coppersmith's algorithm, and from theorem 3.4 for our algorithm, specialized to m and n bounded. The complexity of step BW3 follows from the fact that as a "polynomial" in B, $\hat{w}$ has degree $\frac{N}{n}$. □

Remark 5.2. In practice, the real time needed for step BW1 can be lowered down to $\gamma\mathsf{M}_0\frac{m+n}{mn}N^2$ by using vectors of the canonical basis for the x_i's. Indeed, the dot products which account for the $m\mathsf{M}_1\frac{m+n}{mn}N^2$ term become trivial (1 operation instead of $m\mathsf{M}_1 N$ with random x's). It should be noted however that when we do so, x is no longer truly random, and the correctness analyses of [14, 24, 23] do not necessarily apply.

We will now write down the overall cost of the block Wiedemann, in light of the theorem above. Our analysis is valid over fields other than $\mathbb{F}_2$, since numerous possible tricks in that case tend to shape the things differently.

Theorem 5.3. *Using Coppersmith's algorithm for step* BW2*, the real time for the block Wiedeman algorithm is lowest for $m_{opt} = n_{opt} = \sqrt{\frac{3\gamma\mathsf{M}_0}{\mathsf{M}_1}}$. The total time needed in this case is $\mathsf{W}_{opt} = 2\sqrt{3\gamma\mathsf{M}_0\mathsf{M}_1}N^2$.*

Proof If we write down the total time for the block Wiedemann, following theorem 5.1, we obtain:

$$\mathsf{W} = \gamma\mathsf{M}_0\frac{m+n}{mn}N^2 + \mathsf{M}_1\frac{m+n}{2}N^2 + \gamma\mathsf{M}_0\frac{1}{n}N^2,$$
$$\mathsf{W} = (\gamma\mathsf{M}_0\frac{2\lambda+1}{\lambda}\frac{1}{n} + \mathsf{M}_1\frac{\lambda+1}{2}n)N^2 \quad \text{for } m = \lambda n.$$

If we minimize W for a given λ, the optimal values W_{opt} and n_{opt} are:

$$n_{opt} = \sqrt{\frac{\gamma\mathsf{M}_0 2(2\lambda+1)}{\mathsf{M}_1\lambda(\lambda+1)}},$$
$$\mathsf{W}_{opt} = 2N^2\sqrt{\gamma\mathsf{M}_0\mathsf{M}_1\frac{(2\lambda+1)(\lambda+1)}{2\lambda}}.$$

The minimum value of the quantity $\frac{(2\lambda+1)(\lambda+1)}{2\lambda}$ is obtained for $\lambda = \frac{1}{\sqrt{2}}$, but since we are restricted to $\lambda \geq 1$, $\lambda = 1$ is the most appropriate choice. Specializing n_{opt} and W_{opt} to $\lambda = 1$ yields the announced values. □

Theorem 5.4. *Using algorithm* MSLGDC *for step* BW2*, the real time for the block Wiedeman algorithm is lowest for $m_{opt} = n_{opt} = \sqrt{\frac{3\gamma\mathsf{M}_0 N}{16\mathsf{M}_1\log^2 N}}$. The total time needed in this case is $W_{opt} = 8\sqrt{3\gamma\mathsf{M}_0\mathsf{M}_1}N\sqrt{N}\log N$.*

Proof As before, this follows from theorem 5.1. W writes down as (again using $m = \lambda n$):

$$\mathsf{W} = \gamma\mathsf{M}_0\frac{2\lambda+1}{\lambda}\frac{1}{n}N^2 + 4\mathsf{M}_1(\lambda+1)^2 nN\log^2 N,$$

Following the same reasoning as before, we obtain the best ratio between m and n for $\lambda = 1$ (since we are restricted to $\lambda \geq 1$), and hence the announced W_{opt} and n_{opt}. □

It should be noted that theorem 3.4 yields a low complexity for step BW2 with respect to m and n because of the introduction of the FFT which hides the cubic dependency on $m+n$. If we had used algorithm MSLGDC with a generic multiplication algorithm, W_{opt} would certainly be higher.

The optimal value n_{opt} is not necessarily acceptable, because we are limited by the available hardware. We will see in the following section that for realistic examples, n_{opt} is still reasonable.

6. IMPLEMENTATION CONCERNS

6.1 Interest of the block version

The consequences of our analysis depend on the base field. We excluded $\mathbb{F}_2$ from our scope because in that case, precise time estimates are extremely variable depending on the implementation (for example, additions can no longer be neglected versus multiplications, as we do here). The best choice for m and n is almost certainly the wordsize w of the machine (in fact, we can regard this as having M_0 and M_1 divided by w: as many as w multiplications in K can be done at a time). It is pretty hard to tell without actually trying on real experiments whether it is worthwhile to set m and n to something else (an integer multiple of w for instance). Since our code is focused on large fields, we did no such experiments.

Over large fields, like in the linear algebra problems encoutered in the course of discrete logarithm problems [19], the problem is quite different. First, M_1 is now typically much bigger than M_0: indeed, the coefficients of the input matrix are usually kept bounded to a size of one machine word (see below), so when a generic element of K has size about ten words, M_0 is a dozen machine cycles, whereas M_1 can reach several hundreds machine cycles. Therefore, the second part of the algorithm could end up dominating the overall cost. If we include these considerations in the computation of the optimal value n_{opt} for the parameters m and n, we see that if one uses Coppersmith's version of step BW2, n_{opt} is very small (sometimes hardly above 1). In other words, there is not much interest in using the block version of the Wiedemann algorithm. On the other hand, our algorithm MSLGDC yields a bigger optimal value. Since we must ensure that the cubic term in $(m+n)$ stays small, numerical values are hard to provide, and depend on a thorough implementation. As a rough estimate, we consider that today's biggest linear systems over large fields can take advantage of the method. For example, for a matrix of size $250,000 \times 250,000$ defined over a field of size 1000 bits with an average of 20 non-zero entries per row, the optimal value for n_{opt} with algorithm MSLGDC is 5, whereas we obtain 1 for the quadratic counterpart. The overall speedup obtained is between 3 and 4, but not counting the cubic complexity in $m+n$. It is not likely that more "reasonable" systems can benefit of this method for now.

6.2 Influences on input filtering

Perhaps more important, our computations have a very interesting consequence on the input given to the block Wiedemann algorithm, when it comes out of a structured gaussian program [20], or more generally any filtering stage like in [5]. Such algorithms aim at reducing the matrix size with minimal fill-in — we want the matrix to remain sparse —, as far as this is possible. Their output is then given to an algorithm like Wiedemann's, or alternatively a block version. Depending on the context, reduction rates going from one third to one tenth are achieved. When the base field is not simply $\mathbb{F}_2$, the matrices given on input to the filtering program have small coefficients. Therefore, coefficients of the matrix are stored in a single machine word and not allowed to go beyond this in order to reduce the memory storage. In the course of this filtering, one usually arranges for stopping it as soon as the estimated subsequent cost (of the Wiedemann algorithm for instance) starts to rise up again, after having been diminished. See [25] for an example of such an estimation. The estimated cost is generally something like γN^2. As the filtering proceeds, γ grows while N gets smaller.

Our point here is that when using the block version with optimal parameters m and n, we can focus on the quantity $\sqrt{\gamma}N^2$ instead. This means that we are able to continue the gaussian elimination a bit further. If we plan to use our subquadratic alternative, the relevant figure is $N\sqrt{\gamma N}\log N$, but this is only valid as long as m and n remain small. Experiments with matrices coming from discrete logarithm problems showed that the filtering can actually be brought substantially further.

6.3 Memory requirements

We hardly addressed the memory concerns for the block Wiedemann algorithm. However, these are really important because the memory storage needed for the matrix B is usually huge. For this very reason, parallelization or distribution is hampered by the relative scarcity of computing resources available that can handle such a big object: if we plan to distribute step BW1 among several machines, these have to work on a local copy of the matrix B. This is why having n_{opt} reasonable was crucial. As a side note, we would like to stress the fact step BW2 does not need the matrix B, so the increased memory requirements of algorithm MSLGDC compared to Coppersmith's algorithm are not very important.

An important point in the ability explained in the previous subsection to carry out the filtering or structured gaussian elimination further than what we used to is that this helps in reducing the storage needed for B: the memory requirements for step BW1 are driven by two quantities: γN for the matrix size, and N for the size of all the linear storage data and such. Continuing the filtering further than before makes these quantities lower, and therefore the algorithm could become more usable if its memory requirements are reduced.

7. CONCLUSION

We believe that our study of the block Wiedemann algorithm helps greatly in making it more practical over large fields, and also gives some insights on its characteristics in terms of running time and such. In certain settings, we can expect the block Wiedemann algorithm to become very efficient and competitive, but still at sizes that may seem huge for now. Although our original interest was dealing with matrices coming from discrete log problems, this approach could help to advocate for the use of the block Wiedemann algorithm in other contexts where a (very) large sparse linear system over a large finite field shows up.

I would like to thank François Morain who helped me in preparing this paper. I am also grateful to Gilles Villard and the anonymous referees for their valuable questions and comments.

8. REFERENCES

[1] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *The design and analysis of computer algorithms.* Addison-Wesley, Reading, MA, 1974.

[2] Beckerman, B., and Labahn, G. A uniform approach for the fast computation of matrix-type Padé approximants. *SIAM J. Matrix Anal. Appl. 15*, 3 (July 1994), 804–823.

[3] Bitmead, R. R., and Anderson, B. D. O. Asymptotically fast solution of Toeplitz and related systems of linear equations. *Linear Algebra Appl. 34* (1980), 103–116.

[4] CABAL. 233-digit SNFS factorization. Available at `ftp://ftp.cwi.nl/pub/herman/SNFSrecords/SNFS-233`, Nov. 2000.

[5] Cavallar, S. Strategies in filtering in the number field sieve. In *ANTS-IV* (2000), W. Bosma, Ed., vol. 1838 of *Lecture Notes in Comput. Sci.*, Springer–Verlag, pp. 209–231. Proceedings.

[6] Cavallar, S., Dodson, B., Lenstra, A. K., Lioen, W., Montgomery, P. L., Murphy, B., te Riele, H. J. J., Aardal, K., Gilchrist, J., Guillerm, G., Leyland, P., Marchand, J., Morain, F., Muffett, A., Putnam, C., Putnam, C., and Zimmermann, P. Factorization of a 512-bit RSA modulus. In Preneel [21], pp. 1–18. Proceedings.

[7] Coppersmith, D. Fast evaluation of logarithms in fields of characteristic two. *IEEE Trans. Inform. Theory IT–30*, 4 (July 1984), 587–594.

[8] Coppersmith, D. Solving linear equations over GF(2): Block Lanczos algorithm. *Linear Algebra Appl. 192* (Jan. 1993), 33–60.

[9] Coppersmith, D. Solving linear equations over GF(2) via block Wiedemann algorithm. *Math. Comp. 62*, 205 (Jan. 1994), 333–350.

[10] von zur Gathen, J., and Gerhard, J. *Modern Computer Algebra.* Cambridge University Press, Cambridge, England, 1999.

[11] Gaudry, P. An algorithm for solving the discrete log problem on hyperelliptic curves. In Preneel [21], pp. 19–34. Proceedings.

[12] Gaudry, P. *Algorithmique des courbes hyperelliptiques et applications à la cryptologie.* Thèse, École Polytechnique, Dec. 2000.

[13] Gustavson, F. G., and Yun, D. Y. Y. Fast algorithms for rational Hermite approximation and solution of Toeplitz systems. *IEEE Transactions on Circuits and Systems CAS–26*, 9 (Sept. 1979), 750–755.

[14] Kaltofen, E. Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. *Math. Comp. 64*, 210 (July 1995), 777–806.

[15] Kaltofen, E., and Lobo, A. Distributed matrix-free solution of large sparse linear systems over finite fields. *Algorithmica 24*, 4 (1999), 331–348.

[16] LaMacchia, B. A., and Odlyzko, A. M. Solving large sparse linear systems over finite fields. In *Advances in Cryptology – CRYPTO '90* (1990), A. J. Menezes and S. A. Vanstone, Eds., vol. 537 of *Lecture Notes in Comput. Sci.*, Springer–Verlag, pp. 109–133. Proceedings.

[17] Montgomery, P. L. A block Lanczos algorithm for finding dependencies over GF(2). In *Advances in Cryptology – EUROCRYPT '95* (1995), L. C. Guillou and J.-J. Quisquater, Eds., vol. 921 of *Lecture Notes in Comput. Sci.*, pp. 106–120. Proceedings.

[18] Morf, M. Doubling algorithms for Toeplitz and related equations. In *Proc. IEEE Internat. Conference Acoustics, Speech and Signal Processing* (New York, NY, 1980), IEEE, pp. 954–959.

[19] Odlyzko, A. M. Discrete logarithms in finite fields and their cryptographic significance. In *Advances in Cryptology – EUROCRYPT '84* (1985), T. Beth, N. Cot, and I. Ingemarsson, Eds., vol. 209 of *Lecture Notes in Comput. Sci.*, Springer–Verlag, pp. 224–314. Proceedings.

[20] Pomerance, C., and Smith, J. W. Reduction of huge, sparse matrices over finite fields via created catastrophes. *Experiment. Math. 1*, 2 (1992), 89–94.

[21] Preneel, B., Ed. *Advances in Cryptology – EUROCRYPT 2000* (2000), vol. 1807 of *Lecture Notes in Comput. Sci.*, Springer–Verlag. Proceedings.

[22] Teitelbaum, J. Euclid's algorithm and the Lanczos method over finite fields. *Math. Comp. 67*, 224 (Oct. 1998), 1665–1678.

[23] Villard, G. Further analysis of Coppersmith's block Wiedemann algorithm for the solution of sparse linear systems. In *ISSAC '97* (1997), W. W. Küchlin, Ed., ACM Press, pp. 32–39. Proceedings.

[24] Villard, G. A study of Coppersmith's block Wiedemann algorithm using matrix polynomials. Research report 975, LMC-IMAG, Grenoble, France, Apr. 1997.

[25] Weber, D., and Denny, T. The solution of McCurley's discrete log challenge. In *Advances in Cryptology – CRYPTO '98* (1998), H. Krawczyk, Ed., vol. 1462 of *Lecture Notes in Comput. Sci.*, Springer–Verlag, pp. 458–471. Proceedings.

[26] Wiedemann, D. H. Solving sparse linear equations over finite fields. *IEEE Trans. Inform. Theory IT–32*, 1 (Jan. 1986), 54–62.