Finding column dependencies in sparse matrices over $mathbb{F}_2$ by block Wiedemann

O. Penninga

Modelling, Analysis and Simulation (MAS)

MAS-R9819 September 30, 1998

Report MAS-R9819
ISSN 1386-3703

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Finding Column Dependencies in Sparse Matrices over $\mathbb{F}_2$ by Block Wiedemann

Olaf Penninga

Department of Mathematics, Leiden University

P.O. Box 9512, 2300 RA Leiden, The Netherlands

Address as of September 1, 1998:

Robeco Research, afdeling Kwantitatief Onderzoek

P.O. Box 973, 3000 AZ Rotterdam, The Netherlands

ABSTRACT

Large systems of linear equations over $\mathbb{F}_2$ with sparse coefficient matrices have to be solved as a part of integer factorization with sieve-based methods such as in the Number Field Sieve algorithm. In this report, we first discuss the Wiedemann algorithm to solve these systems and investigate the relation between the sparsity of the matrix and the performance of (a slightly adapted version of) the algorithm. Then we turn to the more efficient block algorithms and discuss a new version of the block Wiedemann algorithm, proposed by Villard, based on the FPHPS algorithm by Beckermann and Labahn. Finally we compare the performance of our implementation of this version of the algorithm with that of Lobo's implementation of the classical block Wiedemann algorithm and that of Montgomery's implementation of his block Lanczos algorithm. The latter is shown to be much faster, even more than expected theoretically.

1991 Mathematics Subject Classification: Primary 15A06. Secondary 65F10, 12Y05.

1991 Computing Reviews Classification System: F.2.1.

Keywords and Phrases: Sparse linear systems, finite fields, exact arithmetic, Krylov subspace methods, Wiedemann methods, block iterative methods, block Wiedemann methods, block Lanczos methods power Hermite Padé approximation.

Note: This report is the author's Master's Thesis written in partial fulfillment of the requirements for the Master's Degree in Mathematics at Leiden University, awarded on August 21, 1998.

Acknowledgements. The author would like to thank H.J.J. te Riele and W.M. Lioen for their help during the writing and testing of the computer programs and the experiments performed with them, S. Cavallar for the matrix used as test case and her advice on Montgomery's program `gauss` and prof. Tijdeman for his help and supervision in the writing of this thesis.

Table of Contents

Introduction

Systems of linear equations can be solved by Gaussian elimination. This is a simple procedure that yields all solutions when applied with exact arithmetic. But in the case of sparse matrices, i.e. matrices containing mainly zeroes, methods which take advantage of this sparsity use much less memory and time to solve the system than Gaussian elimination. As a part of integer factorization with the Quadratic Sieve [19, 20] or the Number Field Sieve [14], one has to find dependencies among the columns of a matrix over $\mathbb{F}_2$, the field with two elements. These matrices are extremely large and sparse for big numbers. Consider for instance the world record factorization of RSA-130 [6], a 130-digit number: The matrix in which column dependencies had to found, had 3,516,502 rows and 3,504,823 columns, with on average only 39.4 non-zero entries per row. To store this matrix in a computer's memory one needs to store "only" the positions of 138,690,744 ones, which takes considerably less memory than storing for all positions either a 1 or a 0; the first option takes about 600 megabyte[1], the latter over 1500 gigabyte. In fact, for such huge matrices only the former solution is practically possible. During Gaussian elimination, however, the matrix loses its sparsity due to *fill-in*: One adds rows to another several times. Since the matrix is sparse, most non-zeroes will be added to zeroes that therefore become non-zeroes too. The initially almost empty (i.e. all-zero) matrix gets filled in with non-zeroes. Therefore the amount of memory required to store the matrix increases strongly. This makes Gaussian elimination impractical for solving large systems of equations with sparse coefficient matrices.

To avoid problems with fill-in, mathematicians use algorithms that do not change the matrix, but only manipulate a few vectors to solve such systems of equations. These methods only need storage for the sparse matrix and for a few (dense) vectors. Besides, they take advantage of the fact that sparse matrices can be multiplied by vectors much faster than normal, i.e. dense matrices. The best known examples are the Conjugate Gradient method [9], the Lanczos method [13, 12] and the Wiedemann method [22]. These methods are still quite inefficient when used over $\mathbb{F}_2$, because they involve mainly manipulations with single bits, whereas most computers handle 32 or 64 bits at the same time. Moreover, single bits have to be unpacked first if they are stored efficiently. To improve their efficiency, algorithms have been modified to work with blocks of vectors instead of single vectors. Coppersmith first made block versions of the Lanczos [4] and Wiedemann [5] algorithms. Inspired

[1]using four bytes to save a row number, storing the row numbers of the ones per column

by his work, Montgomery [18] implemented his own version of the block Lanczos algorithm, which is nowadays widely used for integer factorization. Kaltofen [10] suggested many different versions of block Wiedemann, but Lobo [15] and he implemented Coppersmith's block Wiedemann because the methods Kaltofen suggested, work only over fields with sufficiently many elements. For use over $\mathbb{F}_2$ Kaltofen recommended to me a version suggested by Villard [21], which is based on an algorithm by Beckermann and Labahn [2]. I have implemented this version of block Wiedemann, and compared it with Montgomery's block Lanczos and Lobo's implementation of Coppersmith's block Wiedemann.

In the first chapter we describe the scalar Wiedemann algorithm. In chapter 2 we discuss block vectors and several versions of the block Wiedemann algorithm. In the third chapter we take a look at the implementations of these algorithms and compare their performance in practice.

Chapter 1
Wiedemann's algorithm

The Wiedemann algorithm solves $Ax = b$ using the minimum polynomial of a Krylov sequence. The minimum polynomial is computed by the Berlekamp-Massey algorithm, but this requires some projection of the Krylov sequence. This projection randomizes the algorithm. First we explain how Wiedemann solves $Ax = b$ with this minimum polynomial. Then we describe the Berlekamp-Massey algorithm. Finally, we discuss the randomization and its impact upon the performance of the algorithm.

1. The Krylov sequence

Consider the equation $Ax = b$ over a finite field $\mathbb{F}_q$, with A a non-singular, sparse $n \times n$ matrix. In order to preserve its sparsity, we do not change anything at all to the matrix. We use it only as a "black box" that converts a vector b into Ab. Now we can apply the black box to Ab and so on. What we get is the Krylov sequence

$$b, Ab, A^2b, A^3b, \ldots \tag{1.1.1}$$

The set of all these vectors has dimension no more than n, so there exists a non-trivial linear dependency relation between the first $n+1$ vectors of this sequence. There are $c_0, \ldots, c_d \in \mathbb{F}_q$ satisfying

$$c_0 b + c_1 Ab + c_2 A^2 b + \cdots + c_d A^d b = 0 \tag{1.1.2}$$

with $d \leq n$ and $c_0 \neq 0$ [1]. We may even suppose the coefficients to be normalized in such a way that $c_0 = 1$. With this normalization (1.1.2) gives:

$$b = -A(c_1 b + c_2 Ab + c_3 A^2 b + \cdots + c_d A^{d-1} b).$$

If we define

$$x = -(c_1 b + c_2 Ab + c_3 A^2 b + \cdots + c_d A^{d-1} b)$$

then x is a solution of $Ax = b$. Wiedemann uses the following definitions

$$\begin{aligned} f(x) &= c_0 + c_1 x + c_2 x^2 + \cdots + c_d x^d \\ f^-(x) &= \frac{f(x) - f(0)}{x} \end{aligned}$$

[1] If $c_0 = 0, A(c_1 b + \cdots + c_d A^{d-1} b) = 0$; because A is non-singular, this means that $c_1 b + \cdots + c_d A^{d-1} b = 0$. Remove zero coefficients until a non-zero trailing coefficient is found. This must happen, because there exists some non-trivial linear dependency relation between these vectors.

and describes our solution to $Ax = b$ as

$$x = -f^{-}(A)b.$$

So if we are able, given the Krylov sequence (1.1.1), to determine a polynomial $f(x)$ that fulfills $f(A)b = 0$, we can find a solution to $Ax = b$. But how can we find such a polynomial? The following, almost trivial observation shows us the way:

$$A^j \left(c_0 b + c_1 Ab + c_2 A^2 b + \cdots + c_d A^d b\right) = A^j 0 = 0 \qquad \text{for } j = 0, 1, \ldots$$

or, equivalently,

$$c_0 A^j b + c_1 A^{j+1} b + \cdots + c_d A^{j+d} b = 0 \qquad \text{for } j = 0, 1, \ldots.$$

This means that the Krylov-sequence is a linear recurring sequence. Usually, this is denoted the other way round: A sequence $S = \{s_0, s_1, \ldots\}$ of elements of $\mathbb{F}_q$ is a linear recurring sequence generated by $p(x) = p_0 + p_1 x + \cdots + p_l x^l \in \mathbb{F}_q[x]$ iff

$$p_l s_i + p_{l-1} s_{i+1} + \cdots + p_0 s_{i+l} = 0 \qquad \text{for } i = 0, 1, \ldots \tag{1.1.3}$$

or, equivalently,

$$\deg\left(p(x)s(x)\right) < l \qquad \text{where } s(x) = \sum_{i=0}^{\infty} s_i x^i.$$

Lemma: The set of all $p(x) \in \mathbb{F}_q[x]$ satisfying (1.1.3) is an ideal in $\mathbb{F}_q[x]$.

Proof: Consider $p(x), q(x)$ and $r(x) \in \mathbb{F}_q[x]$ satisfying

$$\begin{aligned} p_l s_i + p_{l-1} s_{i+1} + \cdots + p_0 s_{i+l} &= 0 \qquad \text{for } i = 0, 1, \ldots \\ q_m s_i + q_{m-1} s_{i+1} + \cdots + q_0 s_{i+m} &= 0 \qquad \text{for } i = 0, 1, \ldots \\ r(x) = r_0 + r_1 x + \cdots + r_k x^k & \end{aligned}$$

- $p(x) + q(x) = (p_0 + q_0) + \cdots + (p_l + q_l)x^l + q_{l+1}x^{l+1} + \cdots + q_m x^m$ (we assume without loss of generality that $m \geq l$) and this satisfies (1.1.3): $q_m s_i + \cdots + q_{l+1} s_{i+m-l-1} + (p_l + q_l)s_{i+m-l} + \cdots + (p_0 + q_0)s_{i+m} =$ $p_l s_i + p_{l-1} s_{i+1} + \cdots + p_0 s_{i+l} + q_m s_i + q_{m-1} s_{i+1} + \cdots + q_0 s_{i+m} = 0 + 0 = 0.$ So $p(x) + q(x)$ generates the sequence.
- $r(x)p(x) = r_0 p_0 + (r_0 p_1 + r_1 p_0)x + \cdots + r_k p_l x^{k+l}$; this satisfies (1.1.3) too: $(r_k p_l)s_0 + (r_{k-1} p_l + r_k p_{l-1})s_1 + \cdots + (r_0 p_0)s_{k+l} =$ $r_0(p_l s_k + p_{l-1} s_{k+1} + \cdots + p_0 s_{k+l}) + \cdots + r_k(p_l s_0 + p_{l-1} s_1 + \cdots + p_0 s_l) =$ $r_0 \times 0 + \cdots + r_k \times 0 = 0.$
  So $r(x)p(x)$ generates the sequence.

Because $\mathbb{F}_q[x]$ is a principal ideal ring, there is a unique monic polynomial $p(x)$ generating this ideal. This polynomial is the minimum polynomial of the sequence S. It is the lowest-degree solution p to

$$p_l s_i + p_{l-1} s_{i+1} + \cdots + p_0 s_{i+l} = 0 \qquad \text{for } i = n - l, n - l + 1, \ldots, 2n - l - 1$$

which means that p is the lowest-degree polynomial for which the coefficients of x^i in $p(x)s(x)$ are zero for $n \leq i \leq 2n - 1$. This can be rewritten as

$$p(x) \sum_{i=0}^{\infty} s_i x^i \equiv r(x) \mod x^{2n}$$

with $\deg(r) < n$. Such a polynomial $p(x)$ can be found by the Berlekamp-Massey algorithm.

2. The Berlekamp-Massey algorithm

The Berlekamp-Massey algorithm [3, 17] from Coding Theory computes a solution to the key equation in decoding BCH codes:

$$\sigma(x)\sum_{i=0}^{\infty} s_i x^i \equiv \omega(x) \mod x^{2n}. \tag{2.1.1}$$

Here the s_i are given; both $\sigma(x)$ and $\omega(x)$ have to be found, subject to the condition that $\deg(\sigma)$ and $\deg(\omega)$ must be minimal. Zierler [23] gives a somewhat simplified version of the algorithm which computes only the minimum polynomial $\sigma(x)$ and not $\omega(x)$. This is all we need in the Wiedemann algorithm, but to understand Zierler's version we take a look at the complete algorithm first. We break the problem of solving (2.1.1) up into a series of much easier problems

$$\sigma^{(k)}(x)\sum_{i=0}^{\infty} s_i x^i \equiv \omega^{(k)}(x) \mod x^{k+1} \qquad \text{for } k = 0, 1, \ldots, 2n-1 \tag{2.1.2}$$

and use $\sigma^{(k)}(x)$ and $\omega^{(k)}(x)$ to find $\sigma^{(k+1)}(x)$ and $\omega^{(k+1)}(x)$. It might happen that

$$\sigma^{(k)}(x)\sum_{i=0}^{\infty} s_i x^i \equiv \omega^{(k)}(x) \mod x^{k+2}$$

in which case we can simply take $\sigma^{(k+1)}(x) = \sigma^{(k)}(x)$ and $\omega^{(k+1)}(x) = \omega^{(k)}(x)$, but in general we find

$$\sigma^{(k)}(x)\sum_{i=0}^{\infty} s_i x^i \equiv \omega^{(k)}(x) + \Delta_k x^{k+1} \mod x^{k+2}$$

where Δ_k is the coefficient of x^{k+1} in $\sigma^{(k)}(x)\sum_{i=0}^{\infty} s_i x^i$. If $\Delta_k \neq 0$ we construct $\sigma^{(k+1)}(x)$ and $\omega^{(k+1)}(x)$ by solving $\tau^{(k)}(x)$ and $\gamma^{(k)}(x)$ from the auxiliary equation

$$\tau^{(k)}(x)\sum_{i=0}^{\infty} s_i x^i \equiv \gamma^{(k)}(x) + x^k \mod x^{k+1}. \tag{2.1.3}$$

Having solved this equation, $\sigma^{(k+1)}(x)$ and $\omega^{(k+1)}(x)$ can be constructed by:

$$\begin{aligned}\sigma^{(k+1)} &= \sigma^{(k)} - \Delta_k x \tau^{(k)}\\ \omega^{(k+1)} &= \omega^{(k)} - \Delta_k x \gamma^{(k)}.\end{aligned}$$

It is clear that these $\sigma^{(k+1)}(x)$ and $\omega^{(k+1)}(x)$ satisfy

$$\sigma^{(k+1)}(x)\sum_{i=0}^{\infty} s_i x^i \equiv \omega^{(k+1)}(x) \mod x^{k+2}$$

if $\sigma^{(k)}$ and $\omega^{(k)}$ satisfy (2.1.2) and $\tau^{(k)}$ and $\gamma^{(k)}$ satisfy (2.1.3). Two obvious ways to define $\tau^{(k)}$ and $\gamma^{(k)}$ are

$$\tau^{(k+1)} = x\tau^{(k)} \quad , \quad \gamma^{(k+1)} = x\gamma^{(k)} \tag{2.1.4}$$

and

$$\tau^{(k+1)} = \frac{\sigma^{(k)}}{\Delta_k} \quad , \quad \gamma^{(k+1)} = \frac{\omega^{(k)}}{\Delta_k}, \tag{2.1.5}$$

each with

$$\tau^{(0)} = 1 \quad , \quad \gamma^{(0)} = 0.$$

Of course (2.1.5) can be used only when $\Delta_k \neq 0$. Both definitions give solutions to (2.1.3). Berlekamp [3] derives upper bounds to the degrees of $\sigma^{(k)}$ and $\omega^{(k)}$ using a function $D(k)$. Then he gives a criterion to choose between (2.1.4) and (2.1.5) using $D(k)$ and proves that one finds $\sigma^{(k)}$ and $\omega^{(k)}$ of minimal degree in every step by starting with the right initialization and applying his criterion. Now let $S = \{s_i\}_{i=0}^{\infty}$ be a linear recurring sequence with minimum polynomial $\sigma(x)$ of degree n. This σ is the lowest-degree polynomial satisfying (2.1.2) for $k = 2n-1$, but $\sigma^{(k)}(x)$ is the lowest-degree solution to this equation for each k. This shows that $\sigma^{(2n-1)} = \sigma$, i.e. $\sigma^{(2n-1)}(x)$ is the minimum polynomial of S. Doing more iterations than needed does not change the solution; if $\sigma^{(k)}$ generates S, then $\Delta_j = 0$ for all $j > k$. Moreover $D(k+1)$ can be computed from $D(k)$ using only Δ_k. In the Wiedemann algorithm we do not need $\omega^{(k)}$, and $\sigma^{(k+1)}$ can be computed using only S, $\tau^{(k)}$ and $\sigma^{(k)}$. This means there is no need for us to compute $\gamma^{(k)}$ and $\omega^{(k)}$ explicitly. In this case $D(k)$ can be replaced by a simpler function $\eta^{(k)}$. All we need to compute is the minimum polynomial of a given sequence of field elements $\{s_i\}_{i=0}^{\infty}$, which is known to be linearly generated by a polynomial of degree at most n. This minimum polynomial $\sigma(x)$ can be computed by Zierler's version of the algorithm:

1. $\sigma^{(0)}(x) \leftarrow 1$, $\tau^{(0)}(x) \leftarrow x$, $\eta^{(0)} \leftarrow 0$
2. Do for $j = 0, 1, \ldots, 2n-2$:

$$\begin{array}{rcl}
\Delta_j & \leftarrow & \text{the coefficient of } x^{j+1} \text{ in } \sigma^{(j)}(x) \sum_{i=0}^{\infty} s_i x^i \\
\sigma^{(j+1)} & \leftarrow & \sigma^{(j)} - \Delta_j \tau^{(j)} \\
\tau^{(j+1)} & \leftarrow & \left\{ \begin{array}{rl} x\Delta_j^{-1}\sigma^{(j+1)} & \text{if} \quad \Delta_j \neq 0 \text{ and } \eta^{(j)} \geq 0 \\ x\tau^{(j)} & \text{if} \quad \Delta_j = 0 \text{ or } \eta^{(j)} < 0 \end{array} \right. \\
\eta^{(j+1)} & \leftarrow & \left\{ \begin{array}{rl} -\eta^{(j)} & \text{if} \quad \Delta_j \neq 0 \text{ and } \eta^{(j)} \geq 0 \\ \eta^{(j)} + 1 & \text{if} \quad \Delta_j = 0 \text{ or } \eta^{(j)} < 0 \end{array} \right.
\end{array}$$

3. $\sigma(x) \leftarrow \sigma^{(2n-1)}(x)$.

Dornstetter [7] showed that the Berlekamp-Massey algorithm can be derived from the extended Euclidean algorithm. Consider the key equation

$$\sigma(x) \sum_{i=0}^{\infty} s_i x^i \equiv \omega(x) \mod x^{2n}$$

and remember that we only use the first $2n$ terms of S to determine $\sigma^{(2n-1)}(x) = \sigma(x)$, the minimum polynomial of S. Rewrite the key equation as

$$\sigma(x) \sum_{i=0}^{2n-1} s_i x^i = \omega(x) + x^{2n} r(x).$$

Now consider the so-called reciprocal equation:

$$\sigma^{rec}(x) \sum_{i=0}^{2n-1} s_{2n-1-i} x^i = x^{2n-1+d-t} \omega^{rec}(x) + x^{d-r-1} r^{rec}(x)$$

where d=deg(σ), t=deg(ω) $< d$ and r=deg(r) $< d$. Replace $\omega(x)^{rec}$ by $x^{d-t-1}\omega(x)^{rec}$ and $r(x)$ by $x^{d-r-1}r(x)$ to find

$$\sigma^{rec}(x)\sum_{i=0}^{2n-1} s_{2n-1-i}x^i = x^{2n}\omega^{rec}(x) + r^{rec}(x)$$

with $\deg(r^{rec}) < d$. This means that we have performed the Euclidean division $\sigma^{rec}S^{rec}/x^{2n}$. See [7] for a proof of the formal equivalence of the algorithms.

3. The randomization

We want to compute the minimum polynomial of the Krylov sequence $\{A^ib\}_{i=0}^{\infty}$ to solve $Ax = b$. This is a bit more complicated because the Krylov sequence does not consist of field elements, but of vectors with elements in $\mathbb{F}_q$. The lemma remains valid, so there still exists a well-defined minimum polynomial, but the Berlekamp-Massey algorithm does not work anymore because we cannot divide by Δ_j in the computation of $\tau^{(j+1)}$. We have to "convert" the Krylov vectors into field elements to be able to use the Berlekamp-Massey algorithm. Besides, storing $2n$ (dense) vectors takes more memory than storing a dense $n \times n$ matrix! Wiedemann proposed to replace the Krylov vectors by the inner products of all these vectors with a given vector u_1. These inner products are field elements. Then we can determine the minimum polynomial $p_1(x)$ of the sequence $\{\langle u_1, A^ib\rangle\}_{i=0}^{\infty}$ via the Berlekamp-Massey algorithm. Since we are computing the generating polynomials of sequences in reversed order, we have to reverse them back to find the solution $f(x)$ to $f(A)b = 0$ and hence the solution $x = -f^-A(b)$ to our system of linear equations $Ax = b$. We define $p^{rev}(x)$ to be the polynomial with the coefficients of $p(x)$ in reversed order. Let $p(x) = c_d + c_{d-1}x + \cdots + c_0x^d$ be the minimum polynomial of the Krylov sequence. Since

$$\langle u_1, A^i\left(c_0b + c_1Ab + \cdots + c_dA^db\right)\rangle = \langle u_1, A^i\mathbf{0}\rangle = \langle u_1, \mathbf{0}\rangle = 0$$

we see that

$$c_0\langle u_1, A^ib\rangle + c_1\langle u_1, A^{i+1}b\rangle + \cdots + c_d\langle u_1, A^{i+d}b\rangle = 0 \qquad i = 0, 1, \ldots$$

so this $p(x)$ generates the sequence of inner products, hence it belongs to the ideal of all generating polynomials for this sequence. This ideal is generated by $p_1(x)$, so $p_1(x)$ is a factor of $p(x)$. By repeating this procedure with other vectors $u_2, u_3, \ldots$ we will find factors $p_2(x), p_3(x), \ldots$ of $p(x)$. Now we hope to find the complete polynomial p in a few iterations, i.e. from a small number of factors p_i. This is a highly important point: the trick that enables us to use the Berlekamp-Massey algorithm, randomizes the Wiedemann algorithm and is therefore a key-factor in its performance. A priori, we do not know whether we will find the complete minimum polynomial $p(x)$ and if we find it, we do not know how many iterations this will take.

Luckily, we do not have to do all the work again in every iteration: Suppose we have found the factor $f_1(x) = p_1^{rev}(x)$ of $f(x) = p^{rev}(x)$. Put $b_0 = b$, $d_1 = \deg(p_1)$ and $b_1 = f_1(A)b_0$. If $b_1 = \mathbf{0}$ we are ready, else observe that the Krylov sequence of b_1 has as its minimum polynomial p/p_1, which has degree $d - d_1 \leq n - d_1$. Therefore we need only $2(n - d_1)$ terms of the sequence $\{\langle u_2, A^ib_1\rangle\}_{i=0}^{\infty}$ and $2(n - d_1)$ steps in the Berlekamp-Massey algorithm, to find its minimum polynomial $p_2(x)$. This is a factor of p/p_1; $f_2(x) = p_2^{rev}(x)$ is a factor of $f/f_1 = p^{rev}/p_1^{rev}$. Compute $b_2 = f_2(A)b_1 = f_2f_1(A)b_0$. If $f_2f_1(A)b_0 \neq \mathbf{0}$ then determine the minimum polynomial f_3 of $\{\langle u_3, A^ib_2\rangle\}_{i=0}^{\infty}$ and so on. Usually we find f_r with $f_r \ldots f_1(A)b = \mathbf{0}$ after a small number of iterations. Then we have found $f(x) = f_r \ldots f_1(x)$ and a solution $x = -f^-(A)b$ to $Ax = b$. This is the algorithm in the simplest case, namely when $A \in M(\mathbb{F}_q, n \times n)$ is a non-singular matrix:

```
1. $k \leftarrow 0, d_0 \leftarrow 0, y_0 \leftarrow \mathbf{0}, b_0 \leftarrow b$
2. if $b_k = \mathbf{0}$ then $x \leftarrow -y_k$ STOP
3. pick a random vector $u_{k+1} \in \mathbb{F}_q^n$
4. compute the first $2(n - d_k)$ terms of $\{< u_{k+1}, A^i b_k >\}_{i=0}^{\infty}$
5. determine $f_{k+1}(x)$, the reversed minimum polynomial of this sequence
6. $y_{k+1} \leftarrow y_k + f_{k+1}^{-}(A)b_k, b_{k+1} \leftarrow b_0 + Ay_{k+1}, d_{k+1} \leftarrow d_k + \deg(f_{k+1})$
7. $k \leftarrow k + 1$; continue with step 2.
```

In this description an extra vector y_k is introduced, and the b_k are computed using this vector, to avoid doing the same computations twice (or more) in determining the b_k. The following two lemmas show that these new computations give the desired results: the new b_k are the same as the previously defined ones and y_k gives the solution to $Ax = b$.

Lemma $b_0 + Ay_k = f_k \dots f_1(A)b \qquad \text{for } k = 1, 2, \dots$

Proof By induction on k

- $k = 1$: $b_0 + Ay_1 = b_0 + A(y_0 + f_1^{-}(A)b_0)$
  $= b + Ay_0 + (f_1(A) - 1)b = b + A\mathbf{0} + f_1(A)b - b = f_1(A)b$
- $k \to k + 1$: $b_0 + Ay_{k+1} = b_0 + A(y_k + f_{k+1}^{-}(A)b_k)$
  $= b_0 + Ay_k + f_{k+1}(A)b_k - b_k = f_{k+1}(A)b_k + b_0 + Ay_k - b_k$
  $= f_{k+1}(A)b_k = f_{k+1}(A)(b_0 + Ay_k) = f_{k+1}f_k \dots f_1(A)b$

Lemma $y_k = (f_k \dots f_1)^{-}(A)b \qquad \text{for } k = 1, 2, \dots$

Proof By induction on k

- $k = 1$: $y_1 = y_0 + f_1^{-}(A)b_0 = \mathbf{0} + f_1^{-}(A)b = f_1^{-}(A)b$
- $k \to k + 1$:
  $y_{k+1} = y_k + f_{k+1}^{-}(A)b_k = (f_k \cdots f_1)^{-}(A)b + f_{k+1}^{-}(A)(f_k \cdots f_1(A)b_0)$
  $= (f_k \cdots f_1)^{-}(A)b + \left(\frac{f_{k+1}(x) - f_{k+1}(0)}{x} f_k \cdots f_1(x)\right)_{x=A} b$
  $= (f_k \cdots f_1)^{-}(A)b + \left(\frac{f_{k+1} \cdots f_1 - f_{k+1} \cdots f_1(0)}{x}\right)_{x=A} b - f_{k+1}(0)\left(\frac{f_k \cdots f_1 - f_k \cdots f_1(0)}{x}\right)_{x=A} b$
  $= (f_k \cdots f_1)^{-}(A)b + (f_{k+1} \cdots f_1)^{-}(A)b - (f_k \cdots f_1)^{-}(A)b$
  $= (f_{k+1} \cdots f_1)^{-}(A)b$

The main question is, how many factors $f_i(x)$ of $f(x)$ do we have to find, before $f(x)$ fulfills $f(A)b = \mathbf{0}$, or how many iterations of the algorithm are needed to find $f(x)$? This is the only aspect in which the algorithm is randomized. Wiedemann [22] proves for $\Phi(k)$, the probability that $f(x)$ is found in at most k iterations, that

$$\Phi(k) > 1 - \log\left(\frac{q^{k-1}}{q^{k-1} - 1}\right) \qquad \text{for } k > 1$$

where q is the number of elements of our field and log denotes the natural logarithm. For sufficiently large values of q, the fraction $q/(q - 1)$ is close to 1 and hence $\Phi(2)$ is almost 1. This shows that the algorithm will usually need at most two iterations over large cardinality fields. In the case of small fields we might need more iterations. For $q = 2$ the following lower bounds are obtained:

$$\begin{aligned} \Phi(2) &> 0.307 \\ \Phi(3) &> 0.712 \\ \Phi(4) &> 0.866 \\ \Phi(5) &> 0.935 \\ \Phi(6) &> 0.968 \end{aligned}$$

This means that the algorithm stops after at most three passes with probability bigger than 0.7 for random matrices. Since we are primarily interested in sparse matrices, we have to investigate whether the number of iterations needed is affected by sparsity.

Furthermore, we are interested in finding column dependencies, i.e. in solving $Ax = \mathbf{0}$ for non-square or square but singular matrices A. The algorithm only needs to be slightly modified to solve these cases too. To solve an $n \times (n+1)$ matrix, take the first n columns as A and the last column as b. Now either we find a solution to this system, which gives us a dependency between the $n+1$ columns, or the algorithm fails. The only way the algorithm can fail to produce a solution is by producing a dependency relation with $c_0 = 0$ if A is singular:

$$c_1 Ab + c_2 A^2 b + \cdots + c_d A^d b = \mathbf{0}. \tag{3.1.1}$$

In this case we cannot conclude that $c_1 b + c_2 Ab + \cdots + c_d A^{d-1} b = \mathbf{0}$. If this equation does not hold, the algorithm fails, but in this case we have already found a vector from A's nullspace, i.e. a dependency between the first n columns:

$$A(c_1 b + c_2 Ab + \cdots + c_d A^{d-1} b) = \mathbf{0}.$$

In either case we find a column dependency. The only problem is that the reversal of polynomials that we introduced is not the reversal of a polynomial with respect to its degree, i.e. the reciprocal polynomial

$$f^{rec}(x) = x^{deg(f)} f(x^{-1}).$$

This problem occurs because p_l might be zero in (1.1.3). In that case we do not want to reverse the polynomial with respect to its degree $d < l$; we want to reverse it with respect to l, i.e. we want to take $x^l p(x^{-1})$ instead of $x^d p(x^{-1})$. The reciprocal polynomial is never divisible by x, so the relation (3.1.1) cannot be found by multiplying only the reciprocal polynomials from the Berlekamp-Massey algorithm. We have to multiply the reversed factor $f_i(x) = x^d p(x^{-1}) = c_0 + c_1 x + \ldots + c_d x^d$ by x^s for a suitable s in step 5 of our algorithm. Take as s the smallest non-negative integer satisfying

$$c_0 \langle A^i b, u\rangle + \cdots + c_d \langle A^{i+d} b, u\rangle = 0 \quad \text{for } i = s, s+1, \ldots, 2n-d-1 \tag{3.1.2}$$

Such an s exists since $s = l - d \geq 0$ satisfies (3.1.2). Replace $f_i(x)$ by $x^s f_i(x)$. This polynomial $\tilde{c}_0 + \tilde{c}_1 x + \ldots + \tilde{c}_{d+s} x^{d+s}$ satisfies

$$\tilde{c}_s \langle A^{i+s} b, u\rangle + \cdots + \tilde{c}_{s+d} \langle A^{i+d+s} b, u\rangle = 0 \quad \text{for } i = 0, 1, \ldots, 2n-s-d-1$$

because $\tilde{c}_{i+s} = c_i$ for $i = 0, \ldots, d$. Since $\tilde{c}_0 = \ldots = \tilde{c}_{s-1} = 0$ this is equivalent to

$$\tilde{c}_0 \langle A^i b, u\rangle + \cdots + \tilde{c}_{s+d} \langle A^{i+d+s} b, u\rangle = 0 \quad \text{for } i = 0, 1, \ldots, 2n-s-d-1$$

so $x^s f_i(x) = \tilde{c}_0 + \tilde{c}_1 x + \ldots + \tilde{c}_{d+s} x^{d+s}$ is a solution to

$$\langle \left(f_0 A^i b + f_1 A^{i+1} b + \cdots + f_{s+d} A^{s+i+d} b\right), u\rangle = 0 \quad \text{for } i = 0, 1, \ldots, 2n-s-d-1 \tag{3.1.3}$$

and can be used as $f_i(x)$ in step 5 of our algorithm. Note that a polynomial $f(x)$ that satisfies (3.1.3) for n independent vectors $u \in \mathbb{F}_q^n$ must generate $\{A^i b\}_{i=0}^{\infty}$. Wiedemann's lower bound for $\Phi(k)$ shows that usually polynomials solving (3.1.3) for three random vectors u already generate this sequence.

Anyway, we can solve $Ax = \mathbf{0}$ with A an $n \times (n+1)$ matrix. For matrices with more than $n+1$ columns, simply leave the excess columns away. Square matrices and matrices with fewer columns than rows can also be handled, but these cases are of no importance for sieve-based factorization methods; one can always produce matrices with excess columns by sieving long enough. For square matrices, one might take a random vector as b; with probability at least 0.5 this does not belong to A's nullspace. This forces the algorithm to produce a dependency with $c_0 = 0$ which gives a column dependency, i.e. a vector from A's nullspace. For the case of matrices with fewer columns than rows, see [22].

Chapter 2
Block Wiedemann

Sparse matrices can be applied very fast to vectors. Multiplying a matrix with a vector can be considered as taking a linear combination of the columns of the matrix. Adding a multiple of a sparse column to a vector amounts to adding the right multiples of only a few entries to the corresponding entries of the vector. Store the $M \times N$ matrix A with elements from $\mathbb{F}_q$ as follows:

- a_{ij} = row number of the i^{th} non-zero entry of column j of A
- b_{ij} = value of this entry
- n_j = the weight, i.e. the number of non-zeroes, of the j^{th} column of A

and multiply it by $v \in \mathbb{F}_q^N$ in the following way:

- start with w=$\mathbf{0}$, the null vector of length M, i.e. $w_1 = \cdots = w_M = 0$
- do for $j = 1, \ldots, N$:

```
    if $v_j \neq 0$ do for $i = 1, \ldots, n_j$:
        add $v_j \times b_{ij}$ to $w_{a_{ij}}$
```

then $w = Av$. When working over $\mathbb{F}_2$, this simplifies to

- a_{ij} = row number of the i^{th} non-zero entry of column j of A
- n_j = the weight of the j^{th} column of A

for the storage and

- start with $w = \mathbf{0} \in \mathbb{F}_2^M$
- do for $j = 1, \ldots, N$:

```
    if $v_j \neq 0$ do for $i = 1, \ldots, n_j$:
        add 1 to $w_{a_{ij}}$
```

for the computation. But this means we manipulate single bits, while computers can handle 32 or even 64 bits[1] just as fast! With XOR, the exclusive-or operator, a computer adds 32-bit words bitwise in one operation. This means that we can multiply a matrix with 32 vectors at once, if we store them as a vector of 32-bit words:

- a_{ij} = row number of the i^{th} non-zero entry of column j of A
- n_j = the weight of the j^{th} column of A
- $v_j = (v_{1,j} \dots v_{32,j})$ where v_{ij} is the j^{th} entry of the vector v_i

Now the product $w = Av$ is computed in the following way:

- start with $w=\mathbf{0}$, the $N \times 32$ zero block vector
- do for $j = 1, \dots, N$:

```
    if $v_j \neq 0$ do for $i = 1, \dots, n_j$:
        $w_{a_{ij}} \leftarrow w_{a_{ij}}$ XOR $v_j$.
```

This means that we can perform matrix-vector multiplications over $\mathbb{F}_2$ 16 times faster: v_j will be zero in about half of the cases when considering single random vectors over $\mathbb{F}_2$, but it will hardly ever be zero when considering a block of 32 random vectors. In our algorithm, only the matrices are sparse; the vectors can be considered as random vectors. Using blocks of vectors we can do 32 matrix-vector multiplications in twice the time of a single matrix-vector multiplication, which means a factor 16 speed up, if we can use those blocks of vectors in the rest of our algorithm just as efficiently as 32 single vectors in the normal algorithm. To store single bits efficiently, we have to pack eight bits in a byte and unpack these bits when we need them in a computation. We do not need such packing and unpacking when we work with block vectors. This is an extra speed-up.

In the normal algorithm we take a vector u and determine the first $2n$ terms of $\{a_i\}_{i=0}^{\infty} = \{\langle u, A^i b\rangle\}_{i=0}^{\infty}$. Then we try to find a linear combination of the Krylov vectors that solves $Ax = b$ by determining the minimum polynomial of the sequence $\{a_i\}_{i=0}^{\infty}$, i.e. a polynomial $f(x) = f_0 + \dots + f_n x^n \in \mathbb{F}_q[x]$ which fulfills

$$uA^i b f_n + \dots + uA^{i+n} b f_0 = 0 \qquad \text{for } i = 0, 1, \dots, n-1 \tag{0.2.1}$$

or, equivalently,

$$a(x)f(x) = p_1(x) + x^{2n}p_2(x) \qquad \text{with } a(x) = \sum_{i=0}^{2n-1} a_i x^i \text{ and } \deg(p_1) < n.$$

In the block case we denote the matrix size by $N \times N$ instead of $n \times n$ and instead of the vectors b and u we take block vectors (i.e. $N \times 32$ matrices) x and y and determine the first $2N/32$ terms of $\{A^i y\}_{i=0}^{2N/32}$. From now on we use λ instead of x as polynomial variable, because x already denotes a block vector. The question is: Can we determine a combination of these block vectors $A^i y$ with at least some of its columns in the nullspace of A? We have to reduce the amount of memory needed for the computation to something comparable with the storage for the sparse matrix; otherwise, the original Wiedemann method will be more practical (though slower). Taking the scalar Wiedemann algorithm as a guideline, we store $\{a^{(i)}\}_{i=0}^{2N/32} = \{x^T A^i y\}_{i=0}^{2N/32}$ and try to solve

$$x^T A^i y f_l + \dots + x^T A^{i+l} y f_0 = 0 \qquad \text{for } i = 0, 1, \dots, 2N/32 - l - 1 \tag{0.2.2}$$

[1] I consider from now on only 32 bit-computers

where the f_i are 32×32 matrices. This means we have either to generalize the Berlekamp-Massey algorithm to work with matrices instead of field elements or to find another way to solve (0.2.2).

1. Coppersmith's block Wiedemann algorithm

Coppersmith has generalized the Berlekamp-Massey algorithm to the case of matrix polynomials, i.e. polynomials with matrices as coefficients. These polynomials can also be regarded as "polynomial matrices", i.e. as matrices having polynomials as entries. This means one can speak of the degree of an element of the matrix and the degree of a row of the matrix (the maximum of the degrees of the elements of that row). There is no reason to restrict ourselves to the case of blocks of 32 vectors[2], so we take a block of m vectors for x and and a block of n vectors as y, with $m, n \in \mathbb{N}$. As a generalization of the scalar Wiedemann algorithm we try to solve

$$a(\lambda)f(\lambda) = p_1(\lambda) + \lambda^{N/m+N/n}p_2(\lambda) \quad \text{with } a(\lambda) = \sum_{i=0}^{N/m+N/n} a^{(i)}\lambda^i,\ \deg(p_1) < N/n.$$

The coefficients of $f(\lambda)$ should be vectors from $\mathbb{F}_2^m$, but we compute a polynomial with $m \times n$ matrices as coefficients, giving n solutions for f. We develop this matrix polynomial f step by step, starting with $f^{(t_0)}(\lambda)$ with $t_0 = \lceil \frac{m}{n} \rceil$ and computing $f^{(t+1)}(\lambda)$ from $f^{(t)}(\lambda)$. In the scalar Berlekamp-Massey algorithm we have to update both $\tau^{(k)}(x)$ and $\sigma^{(k)}(x)$ to find $\sigma^{(2n-1)}(x)$; in the block case we also have to do about twice the work expected and update what might be considered as the concatenation of two matrix polynomials: $f^{(t)}(\lambda)$ will be a $(m+n) \times n$ polynomial matrix. Define

$$\text{Coeff}(j; f_l^{(t)}a)_\mu = \sum_{k=0}^{j}\sum_{\nu=1}^{n} f_{l,\nu}^{t,k} a_{\nu,\mu}^{(j-k)}\ ,\ 0 \le j \le \frac{N}{m} + \frac{N}{n},\ 1 \le \mu \le m,\ 1 \le l \le m+n$$

where $a_{\nu,\mu}^{(j)}$ is the entry of $(x^T A^j y)^T$ in position (ν, μ). Let $\text{Coeff}(j; f^{(t)}a)$ be the matrix with $\text{Coeff}(j; f_l^{(t)}a)_\mu$ as entry (l, μ) with $1 \le l \le m+n, 1 \le \mu \le m$. As an upper bound to the degree of a row of $f^{(t)}$ we store for every row its so-called nominal degree. This is initialized with a suitable value, for instance the real degree of the row, and updated using the following rules:

$$\begin{aligned} \deg_{\text{nom}}(f+g) &= \max\{\deg_{\text{nom}}(f), \deg_{\text{nom}}(g)\} \\ \deg_{\text{nom}}(\lambda f) &= \deg_{\text{nom}}(f) + 1. \end{aligned}$$

In every step we maintain the following two conditions:

C1: $\text{Coeff}(j; f_l^{(t)}a) = \mathbf{0}^T \quad \text{for } 1 \le l \le m+n \text{ and } \deg_{\text{nom}}(f_l^{(t)}) \le j < t$

C2: $\text{Rank}(\text{Coeff}(t; f^{(t)}a)) = m.$

Condition C1 is the generalization of the condition

$$\sigma^{(k)}(x)\sum_{i=0}^{\infty} s_i x^i \equiv \omega^{(k)}(x) \mod x^{k+1}$$

in the scalar case. C2 is necessary to perform the inductive step. Coppersmith [5] assumes $m \ge n$ for his probabilistic analysis; Villard [21] shows that this condition can be weakened. Let A be an $N \times N$ matrix. This is Coppersmith's algorithm in the homogeneous case, i.e. for $Ax = \mathbf{0}$:

[2] Not even in the case of 32-bit computers!

1. Choose $x \in M(\mathbb{F}_2, N \times m), z \in M(\mathbb{F}_2, N \times n)$; let $y = Az$.
   Compute $a^{(i)} = (x^T A^i y)^T$ for all $0 \leq i < \frac{N}{m} + \frac{N}{n} + O(1)$.
   Define $a(\lambda) = \sum_i a^{(i)} \lambda^i$.

2. Let $t_0 = \lceil \frac{m}{n} \rceil$ and choose for $f^{(t_0)}$ an $(m+n) \times n$ matrix polynomial of nominal degree t_0: Let the first m rows of $f^{(t_0)}$ have degree $t_0 - 1$ and satisfy condition C1 and take $\lambda^{t_0} I_n$ for the last n rows.

3. Determine a non-singular $(m+n) \times (m+n)$ matrix $\tau^{(t)}$ such that the first n rows of $\tau^{(t)}\text{Coeff}(t; f^{(t)}a)$ are zero and such that the nominal degrees of the rows of $\tau^{(t)} f^{(t)}$ are the same as those of the rows of $f^{(t)}$. Note that the last m rows of $\tau^{(t)}\text{Coeff}(t; f^{(t)}a)$ must be linearly independent to maintain condition C2.

4. Compute $f^{(t+1)} = D\tau^{(t)} f^{(t)}$ where D is the diagonal $(m+n) \times (m+n)$ matrix with as diagonal elements first n times a 1 and then m times a λ.

5. Stop if $f^{(t+1)} = f^{(t)}$ for a few consecutive values of t; else increase t by one and continue with step 3.

We discuss the algorithm step by step. The "O(1)" term in step 1 is a safety measure; the actual number of terms needed to find a solution follows from the stopping criterion in step 5, but it will be shown to be very likely about $\frac{N}{m} + \frac{N}{n}$. The best choices for m and n are multiples of the computer word size, because matrix-block vector products can be computed most efficiently in these cases. By taking $n = m = 32k$ the number of matrix-vector products is reduced by an extra factor k. The amount of work per product increases, but these computations can be distributed over k computers. Each computer computes $x^T A^i y_j$ for $i = 0, 1, \ldots, \frac{2N}{32k} + O(1)$ where y_j is the block vector containing the columns $32j - 31, \ldots, 32j$ of y. Making m a multiple of n reduces the number of matrix-block vector products at the cost of an increase in work per block vector product $x^T \times A^i y$. Note that Coppersmith transposes the $a^{(i)}$.

In step 2, finding a matrix polynomial satisfying all restrictions might be difficult. Coppersmith suggests one may need to take another block vector as x or increase t_0 to be able to find such an $f^{(t_0)}$. In the case $m = n$, we have $t_0 = 1$ and the first m rows of $f^{(1)}$ have to be linearly independent rows with constant elements. The first m rows of Coeff$(1; f^{(1)}a)$ will be formed by the product of the first m rows of $f^{(1)}$ and $a^{(1)}$. So if $m = n$, and x and z are chosen subject to the condition that $x^T Ay$ is non-singular, then choosing m linearly independent rows with constant elements as the first m rows of $f^{(1)}$ is sufficient to satisfy condition C2.

Finding $\tau^{(t)}$ in step 3 can be achieved by sorting the rows of Coeff$(t; f^{(t)}a)$ in order of increasing nominal degree of the corresponding rows of $f^{(t)}$. Then determine n independent vectors in the nullspace of this sorted matrix by bringing it in row-echelon form using Gaussian elimination. Each of the n vectors is found by substituting 1 for one free variable and 0 for the others. These are the first n rows of $\tau^{(t)}$. Take the unit vectors corresponding to the positions of the pivots as the other m rows. This means that the last m rows of $\tau^{(t)} f^{(t)}$ are m rows of $f^{(t)}$, and the first n rows of $\tau^{(t)} f^{(t)}$ consist of one of $f^{(t)}$'s other rows plus some rows of lower or equal nominal degree. This assures the nominal degrees of the rows of $\tau^{(t)} f^{(t)}$ to be the same as those of the rows of $f^{(t)}$.

In step 4 we simply apply the linear transformation $\tau^{(t)}$ to $f^{(t)}$ and multiply the last m rows of $\tau^{(t)} f^{(t)}$ by λ. This means we have to increase the nominal degrees of these last m rows by 1.

The estimated number of iterations needed in step 1 and the stopping criterion in step 5 can be justified only heuristically: In every iteration we increase the average nominal degree by $m/(m+n)$. We start with all rows having nominal degree t_0, so when we have determined $f^{(t)}$ the average nominal degree is $t_0+(t-t_0)(m/(m+n))$. Now we see why we have to compute $\frac{N}{m}+\frac{N}{n}+O(1)$ terms of the sequence $(x^TA^iy)_{i=0}^{\infty}$; by the time $t > \frac{N}{m}+\frac{N}{n}+t_0$ the difference between t and the average nominal degree is

$$t - t_0 - (t-t_0)\frac{m}{m+n} = (t-t_0)\frac{n}{m+n} > (N/m+N/n)\frac{n}{m+n} = N/m.$$

This means that there are values of l such that $t - \deg_{\text{nom}}(f_l^{(t)}) > N/m$. For such values of l, condition C1 consists of more than $N/m \times m = N$ equations: We set $d_l = \deg_{\text{nom}}(f_l^{(t)})$. Now C1 gives

$$\mathbf{0}^T = \text{Coeff}(j; f_l^{(t)}a) = \sum_{0\le k\le j,\ 1\le\nu\le n} f_{l,\nu}^{t,k} a_{\nu,\mu}^{(j-k)} \quad \text{with } a_{\nu,\mu}^{(j-k)} = x_\mu^T A^{j-k} y_\nu$$

for $1 \le \mu \le m$ and $d_l \le j \le t$. This means that the vector $\sum_{\nu,k} f_{l,\nu}^{t,k} A^{d-k} y_\nu$ is orthogonal to the vectors $x_\mu^T A^{j-d}$ $(1 \le \mu \le m,\ d_l \le j \le t)$, so for l such that $t - d_l > N/m$, our candidate-generating polynomial fulfills more than $m \times N/m$ equations, which seems to be the appropriate generalization of the N equations (0.2.1) which f has to fulfill in the scalar case. Coppersmith proves that with high probability for such a $t = N/n + N/m + O(1)$ even the following holds:

$$\text{Coeff}(j; f_l^{(t)}a) = \mathbf{0}^T \quad \text{for } 1 \le l \le n,\ d_l \le j < \infty. \tag{1.2.1}$$

This equation may fail for a few values of l, but it holds (with high probability) for most of the $l \in \{1,\ldots,n\}$. In every iteration we compute $\text{Coeff}(t; f^{(t)}a)$. Usually this is non-zero, so we have to compute a transformation $\tau^{(t)}$ that makes $\text{Coeff}(t; f^{(t+1)}a)$ zero, but when we have performed enough iterations and the above equality holds, $\text{Coeff}(t; f^{(t)}a) = \mathbf{0}^T$. In this case $f_l^{(t+1)} = f_l^{(t)}$ for $1 \le l \le n$. If we detect this last condition on a few successsive iterations, we assume that we arrived at the point where (1.2.1) holds and stop the iterations. Now we know that

$$\langle x_\mu^T A^{j-d}, \sum_{\nu,k} f_{l,\nu}^{t,k} A^{d-k} y_\nu \rangle = 0 \quad \text{for } 1 \le \mu \le m,\ d_l \le j \le t.$$

Let d'_l be the actual degree of f_l; $d'_l \le d_l$. Define

$$g_{l,\nu}(\lambda) = \sum_{k=0}^{d'} f_{l,\nu}^{t,k} \lambda^{d'-k} \quad \text{for } 1 \le l \le n,\ 1 \le \nu \le n.$$

This is the reciprocal polynomial of $f_{l,\nu}^{(t)}$. Now define our solution vectors:

$$w_l := \sum_{\nu=1}^{n} g_{l,\nu}(A) z_\nu \quad \text{for } 1 \le l \le n.$$

We know that

$$x^T A^i \times A^{1+d_l-d'_l} w_l = \mathbf{0} \quad \text{for some } l \in \{1,\ldots,n\}$$

and Coppersmith [5] proves that this means that with high probability

$$A^{1+d_l-d'_l} w_l = \mathbf{0} \quad \text{for some } l \in \{1,\ldots,n\}. \tag{1.2.2}$$

If $w_l = \mathbf{0}$ we find the trivial solution to $Ax = \mathbf{0}$, but the probability that this happens is small, if we assume that (1.2.2) holds: In the computation of the matrix polynomial $f^{(t)}(\lambda)$ we do not use z; we only use $y = Az$. This means that any vector z' that differs from z only by an element from $\mathrm{Ker}(A)$ gives the same polynomial f. Since $Az' = \mathbf{0}$, values of w_l computed from z and z' will differ by the constant term of f times $z - z'$; this is an element of the kernel of A. Our vector z differs from the vector z' which gives $w_l' = \mathbf{0}$ by a random element of $\mathrm{Ker}(A)$, so the w_l that we get is the constant term of f times that random element, i.e. a random element of $\mathrm{Ker}(A)$. This means that the probability that $w_l = \mathbf{0}$ is $1/|\mathrm{Ker}(A)|$, which is in our case very small. So with high probability the block $w = (A^{d_1-d_1'}w_1 \dots A^{d_n-d_n'}w_n)$ contains almost n non-zero vectors from the nullspace of A.

2. Block Toeplitz algorithms

Wiedemann already suggested the possibility of computing a generator polynomial for the sequence of matrix-vector products by fast algorithms for Toeplitz systems instead of the Berlekamp-Massey algorithm, but remarked that this would only mean a speed-up of a part of the algorithm which already used only a negligible fraction of the total computing time. Kaltofen [10] uses this suggestion in the block case to propose several other block Wiedemann algorithms. Having computed

$$a^{(i)} = x^T A^i y \qquad \text{for all } 0 \le i < \frac{N}{m} + \frac{N}{n} + \frac{2n}{m} + 1$$

we want to find a polynomial

$$f(x) = \sum_{i=0}^{D} c^{(i)} x^i \qquad \text{with } D = \lceil \frac{N}{n} \rceil \text{ and } c^{(i)} \in M(\mathbb{F}_2, m \times n)$$

satisfying

$$\sum_{i=0}^{D} c^{(i)} a^{(i+j)} = 0 \qquad \text{for } j = 0, 1, \dots, E-1 \text{ with } S = n(D+1),\ E = \lceil \frac{S}{m} \rceil. \tag{2.2.1}$$

Note that Coppersmith shows that usually the solution to this problem which his algorithm finds, also solves it for $j \ge E$. This a priori stronger requirement is used in the further analysis of the algorithm. It is not clear that just any solution to the above equations will give a solution to $Ax = \mathbf{0}$. Kaltofen ignores this point: he proposes other ways to solve (2.2.1) without showing explicitly that these solutions can be used in Coppersmith's further analysis just as well. The system which Kaltofen solves is a linear system of block Toeplitz structure:

$$\begin{pmatrix} a^{(D)} & a^{(D-1)} & \dots & a^{(1)} & a^{(0)} \\ a^{(D+1)} & a^{(D)} & \dots & a^{(2)} & a^{(1)} \\ \vdots & \vdots & & \vdots & \vdots \\ a^{(D+E-1)} & a^{(D+E-2)} & \dots & a^{(E)} & a^{(E-1)} \end{pmatrix} \begin{pmatrix} c^{(D)} \\ c^{(D-1)} \\ \vdots \\ c^{(0)} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{2.2.2}$$

with $c^{(i)} \in \mathbb{F}_2^n$. We compute m solutions, so actually we find $c^{(i)} \in M(\mathbb{F}_2, n \times m)$. Kaltofen [10] suggests to use the generalized Levinson-Durbin algorithm to solve this system, or a speeded version of this algorithm using Fast Fourier Transform-based polynomial multiplication and a divide-and-conquer approach. Both solutions require that the matrix has generic rank profile, i.e. that the leading $i \times i$ principal submatrices[3] are non-singular for $0 \le i \le \mathrm{rank}\ A$. Kaltofen shows that $\tilde{A} = VAW$

[3]the $i \times i$ submatrices located in the left upper corner of A

has generic rank profile with probability at least $1 - \frac{1}{2}N(N-1)/q$ if V is an upper triangular Toeplitz matrix and W a lower triangular Toeplitz matrix with elements from the same field $\mathbb{F}_q$ as A. Unfortunately, in the case $q = 2$ this lower bound is negative for $N \geq 3$ and, even worse, with a reasonably high probability the matrix $\tilde{A}$ does not have generic rank profile: The probability that the first principal submatrix is singular, i.e. that the element in the left upper corner of A is 0, is already $\frac{1}{2}$. Kaltofen suggests some more approaches, but shows that these alternatives will be considerably slower than Coppersmith's algorithm.

3. Power Hermite Padé approximation

Villard [21] proposes yet another method to find a generating polynomial for $a(x)$. He considers (0.2.2) as a matrix Padé approximation problem: Given a $p \times q$ matrix A with power series over a field K as elements and given $M, N, r \in \mathbb{N}$ find $P \in K^{(p\times r)}[x]$, $Q \in K^{(q\times r)}[x]$ with $\deg(P) \leq M$, $\deg(Q) \leq N$ and the columns of Q being linearly independent over K such that

$$A(x)Q(x) + P(x) = x^{M+N+1}R(x) \quad \text{with } R \in K^{(p\times r)}[[x]].$$

Our case can be written as

$$\left(\sum_{i=0}^{\infty} a^{(i)}x^i\right) Q(x) + P(x) = x^{N/n+N/m+O(1)}R(x) \quad \text{with } R \in K^{(m\times 32)}[[x]]. \tag{3.2.1}$$

We want to find $P \in K^{(m\times 32)}[x]$, $Q \in K^{(n\times 32)}[x]$ with $\deg(P) \leq N/m$, $\deg(Q) \leq N/n$; a solution Q with linearly independent columns gives independent solutions to our system. Now we show (3.2.1) to be equivalent to a power Hermite Padé approximation problem, that is, to approximating a vector of power series by a vector of rational functions. The product **FG** of two vectors of power series (or polynomials) simply is the inner product of these two vectors.

Definition Let $\mathbf{F}=(F_1, \ldots, F_l)$ be an l-tuple of formal power series with coefficients from a field K and let $\sigma \geq 0, s > 0, n_1, \ldots, n_l \in \mathbb{N} \cup \{0, -1\}$ be integers; $\mathbf{n}=(n_1, \ldots, n_l)$. Then a power Hermite Padé approximant (PHPA) $\mathbf{P} = (P_1, \ldots, P_l)$ of type $(\mathbf{n}, \sigma, s)$ consists of polynomials P_i with scalar coefficients having degrees bounded by the n_i with

$$\mathbf{P}(x^s)\mathbf{F}(x) = P_1(x^s)f_1(x) + \cdots + P_l(x^s)f_l(x) = x^\sigma \mathbf{R}(x).$$

This concept was introduced by Beckermann and Labahn in [1]. It is important to realise that they use the convention that the zero polynomial has degree $-\infty$. We need a somewhat more general concept to prove the correctness of the algorithm that we give later on:

Definition For $\delta \in \mathbb{Z}$ and $\mathbf{n} = (n_1, \ldots, n_l)$, $n_i \in \mathbb{Z}, n_i \geq -1$, we define $\mathbf{n}(\delta)$ by $\mathbf{n}(\delta)$:=(max$\{-1, n_1 + \delta\}$, … ,max$\{-1, n_l + \delta\}$).

We use the notation L_δ^σ for the set of all PHPA's of type $(\mathbf{n}(\delta), \sigma, s)$. Beckermann and Labahn [2] show that this set L_δ^σ is a finite-dimensional space over K for every $\delta \in \mathbb{Z}$ and $\sigma \in \mathbb{Z}_{\geq 0}$. They also give an algorithm to compute a basis for all the PHPA's of a given **F**, and show that classical Hermite Padé approximations and matrix-Padé forms can be regarded as special cases of power Hermite Padé approximation. We use the following definition:

$$A(x) = \sum_{i=0}^{\infty} a^{(i)}x^i \quad \text{where} \quad a^{(i)} = x^T A^i y$$

and rewrite (3.2.1) as

$$A(x)Q(x) + P(x) = x^{N/n+N/m+O(1)}R(x) \quad \text{with } R \in K^{(m\times(m+n))}[[x]]. \tag{3.2.2}$$

Denote the j^{th} column of P, Q and R by P_j, Q_j and R_j, respectively, and take $B = N/n + N/m + O(1)$. The last equation is equivalent to

$$A(x)Q_j(x) + P_j(x) = x^B R_j(x) \qquad (1 \le j \le m+n)$$

that is

$$\sum_{i=1}^{n} (A_i(x)Q_{ij}(x) + e_i P_{ij}(x)) = x^B R_j(x) \qquad (1 \le j \le m+n)$$

where A_i is the i^{th} column of A, e_i the i^{th} unit vector and $P_{ij}(x)$ the element in row i, column j of P. Replace x by x^m and we find the equivalent equation

$$\sum_{i=1}^{n} (A_i(x^m)Q_{ij}(x^m) + e_i P_{ij}(x^m)) = x^{mB} R_j(x^m) \qquad (1 \le j \le m+n). \tag{3.2.3}$$

This vector equation consists of m scalar equations. It is equivalent to the following single scalar equation:

$$(1, x, \ldots, x^{m-1}) \sum_{i=1}^{n} (A_i(x^m)Q_{ij}(x^m) + e_i P_{ij}(x^m)) = (1, \ldots, x^{m-1}) x^{mB} R_j(x^m) \tag{3.2.4}$$

for $1 \le j \le m+n$. The condition that the i^{th} entry of the left vector in (3.2.3) equals the i^{th} entry of the right vector in (3.2.3) is equivalent to the condition that the coefficients of x^k with $k \equiv i-1 (\text{mod } m)$ in the left and right side of (3.2.4) are equal. This means that we can rewrite (3.2.2) as

$$\sum_{i=1}^{n} \big((1, x, \ldots, x^{m-1})A_i(x^m)Q_{ij}(x^m) + x^{i-1}P_{ij}(x^m)\big) = x^{mB}(1, \ldots, x^{m-1})R_j(x^m)$$

for $1 \le j \le m+n$. Our matrix Padé problem (3.2.2) can indeed be seen as a power Hermite Padé approximation problem

$$\mathbf{P}(x^s)\mathbf{F}(x) = P_1(x^s)f_1(x) + \cdots + P_l(x^s)f_l(x) = x^\sigma \mathbf{R}(x)$$

with $l = m+n$, $s = m$, $\sigma = mB = m(N/n + N/m + O(1))$ and

$$f_i(x) = \begin{cases} x^{i-1} & i = 1, \ldots, m \\ (1, x, \ldots, x^{m-1})A_i(x^m) & i = m+1, \ldots, m+n \end{cases}$$

where A_i is the i^{th} column of A. The elements of P and Q are

$$P_{ij}(x) = P_i(x) \text{ from the } j^{th} \text{ solution for } i = 1, \ldots, m\ ,\ j = 1, \ldots m+n$$

$$Q_{ij}(x) = P_{i+m}(x) \text{ from the } j^{th} \text{ solution for } i = 1, \ldots, n\ ,\ j = 1, \ldots m+n.$$

We introduce the defect and order of a power Hermite Padé approximant to give a better description of the solutions to our problem:

Definition The defect of a power Hermite Padé approximant $\mathbf{P} = (P_1, \ldots, P_l)$ (with respect to the fixed multiindex $\mathbf{n}$=$(n_1, \ldots, n_l)$) is

$$\operatorname{dct}(\mathbf{P}) = \min\{n_i + 1 - \deg(P_i) : \ 1 \leq i \leq l\}$$

where the zero polynomial has degree $-\infty$.

Definition The order of a power Hermite Padé approximant $\mathbf{P}$ (with respect to $s \in \mathbb{N}$ and $\mathbf{F}$) is

$$\operatorname{ord}(\mathbf{P}) = \sup\{\sigma \in N_0 : \mathbf{P}(x^s)\mathbf{F}(x) = x^\sigma R(x) \text{ with } R \in K[[x]]\}.$$

With these definitions we can describe our PHPA solution set L_δ^σ simply as

$$L_\delta^\sigma = \{\mathbf{P} \in K^l[x] : \ \operatorname{dct}(\mathbf{P}) > -\delta, \ \operatorname{ord}(\mathbf{P}) \geq \sigma\}$$

and we can define the so-called σ-bases of PHPA's:

Definition The system $\mathbf{P}_1, \ldots, \mathbf{P}_l$ with $\mathbf{P}_i \in K^l[x]$ is a σ-basis if and only if

1. $\mathbf{P}_1, \ldots, \mathbf{P}_l \in L_{+\infty}^\sigma$
2. For each $\delta \in \mathbb{Z} \cup \{+\infty\}$ and for each $\mathbf{Q} \in L_\delta^\sigma$ there exists a unique tuple of polynomials $(\alpha_1, \ldots, \alpha_l)$ with $\deg(\alpha_i) < \operatorname{dct}\mathbf{P}_i + \delta$ such that $\mathbf{Q}$=$\alpha_1\mathbf{P}_1 + \cdots + \alpha_l\mathbf{P}_l$.

As a consequence of this definition, the elements of a σ-basis are linearly independent over $K[x]$. See [2] and the references therein for a proof of the existence of σ-bases. Note that

$$\operatorname{dct}(c\mathbf{P})=\operatorname{dct}(\mathbf{P}), \quad \operatorname{dct}(\mathbf{P}+\mathbf{Q}) \geq \min\{\operatorname{dct}(\mathbf{P}), \operatorname{dct}(\mathbf{Q})\}, \quad \operatorname{dct}(x\mathbf{P}) = \operatorname{dct}(\mathbf{P}) - 1,$$

$$\operatorname{ord}(c\mathbf{P})=\operatorname{ord}(\mathbf{P}), \quad \operatorname{ord}(\mathbf{P} + \mathbf{Q}) \geq \min\{\operatorname{ord}(\mathbf{P}), \operatorname{ord}(\mathbf{Q})\}, \quad \operatorname{ord}(x\mathbf{P}) = \operatorname{ord}(\mathbf{P}) + s.$$

Now it is easy to derive the following result:

Lemma $L_\delta^{\sigma+1} \subset L_\delta^\sigma$, dim $L_\delta^{\sigma+1} \geq$ dim $L_\delta^\sigma - 1$

Proof The first statement is obvious.
For the second assertion, consider $\mathbf{P} \in L_\delta^\sigma \backslash L_\delta^{\sigma+1}$, i.e. $\operatorname{ord}(\mathbf{P})$=$\sigma$, $\operatorname{dct}(\mathbf{P})$> $-\delta$. Then for $\mathbf{Q} \in L_\delta^\sigma$ there exists a $c \in K$ such that $\mathbf{Q} - c\mathbf{P} \in L_\delta^{\sigma+1}$:
Write $\mathbf{P}(x^s)\mathbf{F}(x) = x^\sigma R_P(x)$, $\mathbf{Q}(x^s)\mathbf{F}(x) = x^\sigma R_Q(x)$. The trailing coefficient of $R_P(x)$ is non-zero since $\operatorname{ord}(\mathbf{P})$=$\sigma$. Let c denote the quotient of the trailing coefficients of R_Q and R_P; then $R_Q(x) - cR_P(x) = xR(x)$. This means that $(\mathbf{Q} - c\mathbf{P})(x^s)\mathbf{F}(x) = x^\sigma(R_Q(x) - cR_P(x)) = x^{\sigma+1}R(x)$ so $\mathbf{Q} - c\mathbf{P} \in L_\delta^{\sigma+1}$.
So adjoining $\mathbf{P}$ to a basis of $L_\delta^{\sigma+1}$ gives a system of generators for L_δ^σ. This proves the second part of the lemma.

The proof of the lemma suggests the following way to construct σ-bases of PHPA's: it is the Fast Power Hermite Padé Solver by Beckermann and Labahn
Let $l \geq 2$, $s \in N$, $\mathbf{F} = (f_1, \ldots, f_l)^T$, $\mathbf{n} = (n_1, \ldots, n_l)$

```
1. $d_{i,0} \leftarrow n_i; \mathbf{P}_{i,0} \leftarrow e_i$, the ith unit vector, for $i = 1, \ldots, l$; $\sigma \leftarrow 1$
2. $c_{i,\sigma} \leftarrow x^{-\sigma}\mathbf{P}_{i,\sigma}(x^s)\mathbf{F}(x)|_{x=0}$ for $i = 1, \ldots, l$ and $\Lambda_\sigma \leftarrow \{i : c_{i,\sigma} \neq 0\}$
3. If $\Lambda_\sigma = \emptyset$ then $\mathbf{P}_{i,\sigma+1} \leftarrow \mathbf{P}_{i,\sigma}$, and $d_{i,\sigma+1} \leftarrow d_{i,\sigma}$ for $i = 1, \ldots, l$
```

4. If $\Lambda_\sigma \neq \emptyset$ then $\pi_\sigma \leftarrow \pi$ with $d_{\pi,\sigma}$ =max$\{d_{i,\sigma} : i \in \Lambda_\sigma\}$

$$\begin{array}{lllllll} \mathbf{P}_{i,\sigma+1} & \leftarrow & \mathbf{P}_{i,\sigma} - \frac{c_{i,\sigma}}{c_{\pi,\sigma}}\mathbf{P}_{\pi,\sigma}, & d_{i,\sigma+1} & \leftarrow & d_{i,\sigma} & \text{for } i \in \Lambda_\sigma, i \neq \pi \\ \mathbf{P}_{i,\sigma+1} & \leftarrow & \mathbf{P}_{i,\sigma}, & d_{i,\sigma+1} & \leftarrow & d_{i,\sigma} & \text{for } i \notin \Lambda_\sigma \\ \mathbf{P}_{\pi,\sigma+1} & \leftarrow & x\mathbf{P}_{\pi,\sigma}, & d_{\pi,\sigma+1} & \leftarrow & d_{\pi,\sigma} - 1 & \end{array}$$

5. $\sigma \leftarrow \sigma + 1$; continue with step 2

This FPHPS algorithm is well defined (in particular step 2) and the $\mathbf{P}_{1,\sigma}, \ldots, \mathbf{P}_{l,\sigma}$ satisfy

$$L_\delta^\sigma = \{\alpha_1\mathbf{P}_{1,\sigma} + \cdots + \alpha_l\mathbf{P}_{l,\sigma} : \ \deg(\alpha_i) \leq d_{i,\sigma} + \delta\} \text{ and } \mathrm{ord}(\mathbf{P}_{i,\sigma}) > \sigma.$$

This can be seen from the following lemma:

Lemma The $\mathbf{P}_{1,\sigma}, \ldots, \mathbf{P}_{l,\sigma}$ form a σ-basis with $\mathrm{dct}(\mathbf{P}_{i,\sigma}) = d_{i,\sigma} + 1$ for σ=0,1, ...

Proof By induction on σ for fixed δ.
For $\sigma = 0$ this follows directly from the definition of L_δ^0.
Suppose that the assertion is true for σ. We show that FPHPS gives the right output for $\sigma+1$. By assumption $\mathbf{P}_{i,\sigma}(x^s)\mathbf{F}(x) = x^\sigma R_i(x)$ with $R_i(x) \in K[[x]]$ so $c_{i,\sigma} = R_i(0)$ is well-defined. We have

$$\mathrm{ord}(\mathbf{P}_{i,\sigma+1}) \geq \sigma + 1 \quad \text{and} \quad \mathrm{dct}(\mathbf{P}_{i,\sigma+1}) \geq d_{i,\sigma+1} + 1$$

by construction. A linear dependency relation between the $\mathbf{P}_{1,\sigma+1}, \ldots, \mathbf{P}_{l,\sigma+1}$ would give a linear dependency relation between the $\mathbf{P}_{1,\sigma}, \ldots, \mathbf{P}_{l,\sigma}$, which we assume to be linearly independent with respect to polynomial coefficients, so $\mathbf{P}_{1,\sigma+1}, \ldots, \mathbf{P}_{l,\sigma+1}$ are linearly independent with respect to polynomial coefficients. Now we have to show that $\mathbf{P}_{1,\sigma+1}, \ldots, \mathbf{P}_{l,\sigma+1}$ generate L_δ^σ and $\mathrm{dct}(\mathbf{P}_{i,\sigma+1}) \geq d_{i,\sigma+1} + 1$.
First consider the case $L_\delta^{\sigma+1} = L_\delta^\sigma$. Now each $\mathbf{Q} \in L_\delta^{\sigma+1}$ has a representation

$$\mathbf{Q} = \alpha_1\mathbf{P}_{1,\sigma} + \cdots + \alpha_l\mathbf{P}_{l,\sigma}\ , \quad \deg(\alpha_i) < \mathrm{dct}(\mathbf{P}_{i,\sigma}) + \delta.$$

If $\alpha_i \neq 0$ we get $\mathrm{dct}(\mathbf{P}_{i,\sigma}) + \delta > \deg(\alpha_i)$; otherwise we have $\deg(\alpha_i) = -\infty < \mathrm{dct}(\mathbf{P}_{i,\sigma+1}) + \delta$. Since $\mathbf{P}_{i,\sigma} \in L_\delta^\sigma = L_\delta^{\sigma+1}$ we have $c_{i,\sigma} = 0$. This means that $\mathbf{P}_{i,\sigma+1} = \mathbf{P}_{i,\sigma}$ and $d_{i,\sigma} = d_{i,\sigma+1}$, so $\mathrm{dct}(\mathbf{P}_{i,\sigma+1}) = \mathrm{dct}(\mathbf{P}_{i,\sigma}) = d_{i,\sigma} + 1 = d_{i,\sigma+1} + 1$. This means that the above representation for $\mathbf{Q}$ already is a suitable representation in terms of a $\sigma + 1$-basis:

$$\mathbf{Q} = \alpha_1\mathbf{P}_{1,\sigma+1} + \cdots + \alpha_l\mathbf{P}_{l,\sigma+1}\ , \quad \deg(\alpha_i) < \mathrm{dct}(\mathbf{P}_{i,\sigma+1}) + \delta.$$

This shows that $\mathbf{P}_{1,\sigma+1}, \ldots, \mathbf{P}_{m,\sigma+1}$ generate $L_\delta^{\sigma+1}$ and $\mathrm{dct}(\mathbf{P}_{i,\sigma+1}) = d_{i,\sigma+1} + 1$ if $L_\delta^{\sigma+1} = L_\delta^\sigma$.
Now consider the case $L_\delta^{\sigma+1} \neq L_\delta^\sigma$. Put

$$L_\delta := \{\alpha_1\mathbf{P}_{1,\sigma+1} + \cdots + \alpha_l\mathbf{P}_{l,\sigma+1} : \ \deg(\alpha_i) < \mathrm{dct}(\mathbf{P}_{i,\sigma+1}) + \delta\}.$$

We have $L_\delta \subset L_\delta^{\sigma+1}$ and we can estimate the dimension of L_δ as follows:

$$\begin{array}{rcl} \dim(L_\delta) & = & \max\{\mathrm{dct}(\mathbf{P}_{1,\sigma+1}) + \delta, 0\} + \cdots + \max\{\mathrm{dct}(\mathbf{P}_{l,\sigma+1}) + \delta, 0\} \\ & \geq & \max\{d_{1,\sigma+1} + 1 + \delta, 0\} + \cdots + \max\{d_{l,\sigma+1} + 1 + \delta, 0\} \\ & \geq & \max\{d_{1,\sigma} + 1 + \delta, 0\} + \cdots + \max\{d_{l,\sigma} + 1 + \delta, 0\} - 1 \\ & = & \dim(L_\delta^\sigma) - 1 = \dim(L_\delta^{\sigma+1}) \end{array}$$

so $L_\delta = L_\delta^{\sigma+1}$ and we have equality in the above inequalities. We have $L_\Delta^{\sigma+1} \neq L_\Delta^\sigma$ $(\forall \Delta \geq \delta)$ since $\emptyset \neq L_\delta^\sigma \backslash L_\delta^{\sigma+1} \subset L_\Delta^\sigma \backslash L_\Delta^{\sigma+1}$, so the above remains valid if we replace δ by $\Delta \geq \delta$. We know that

$$\begin{aligned}\max\{\mathrm{dct}(\mathbf{P}_{1,\sigma+1}) + \delta, 0\} + \cdots + \max\{\mathrm{dct}(\mathbf{P}_{l,\sigma+1}) + \delta, 0\} &= \\ \max\{d_{1,\sigma+1} + 1 + \delta, 0\} + \cdots + \max\{d_{l,\sigma+1} + 1 + \delta, 0\}\end{aligned}$$

so if we choose Δ sufficiently large, i.e. make sure that $\mathrm{dct}(\mathbf{P}_{i,\sigma+1}) + \Delta > 0$ and $d_{i,\sigma+1} + 1 + \Delta > 0$ $(\forall l)$ we can conclude that

$$\mathrm{dct}(\mathbf{P}_{1,\sigma+1}) + \Delta + \cdots + \mathrm{dct}(\mathbf{P}_{l,\sigma+1}) + \Delta = d_{1,\sigma+1} + 1 + \Delta + \cdots + d_{l,\sigma+1} + 1 + \Delta.$$

Since $\mathrm{dct}(\mathbf{P}_{i,\sigma+1}) \geq d_{i,\sigma+1} + 1$ this gives $\mathrm{dct}(\mathbf{P}_{i,\sigma+1}) = d_{i,\sigma+1} + 1$ for $i = 1, \ldots, l$ which completes our proof.

This gives us a deterministic way to find all solutions $f(\lambda) = f_0 + f_1\lambda + \cdots + f_l\lambda^l$ to

$$x^T A^i y f_l + \cdots + x^T A^{i+l} y f_0 = 0 \qquad \text{for } i = 0, 1, \ldots, 2N/32 - l - 1$$

but we still have to rely on probabilistic results to see that these solutions are likely to solve

$$A(A^i z f_l + \cdots + A^{i+1} z f_0) = A^i y f_l + \cdots + A^{i+l} y f_0 = 0 \qquad \text{for } i = 0, 1, \ldots, 2N/32 - l - 1$$

so we have another probabilistic block Wiedemann algorithm to solve $Ax = \mathbf{0}$.

4. Probabilistic results

Coppersmith [5] proves that his algorithm works for most matrices with high probability. His lower bounds to the probabilities that some sufficient conditions are met impose a few conditions on the matrix A. Matrices which do not meet these requirements are called pathological. The formal definition of pathologicalness is quite unintuitive and it is infeasible to verify whether a specific matrix really is pathological. Coppersmith's block Wiedemann algorithm probably is the best test for pathologicalness: If the algorithm fails, the matrix is pathological with high probability. Coppersmith remarks that it has been his experience that matrices arising from integer factorization are not pathological ([5, p344]). There is a more intuitive criterion: Pathological matrices are matrices with several eigenvalues with high multiplicity (compared to m and n). This means that the algorithm will work correctly with high probability for all matrices having at most a few eigenvalues with high multiplicity.

Kaltofen [10] discusses several ways to randomize the block Wiedemann algorithm. These randomizations lead to probabilistic algorithms for which Kaltofen gives lower bounds to the probability that the desired results are obtained, but unfortunately all these lower bounds involve the number of elements in the field over which we solve $Ax = b$. We are interested in huge matrices with elements from $\mathbb{F}_2$. In this case all lower bounds are negative. Since we know already that the probability that the algorithm gives the desired results is at least zero, these lower bounds are not much of a help to us.

Villard [21] shows that the algorithm works with high probability for all matrices (including pathological ones) if $m \geq n$. This condition is not necessary in general; Villard shows that $m \geq \min\{\phi, n\}$ is sufficient, where ϕ is the number of blocks of the Frobenius normal form of the restriction of A to its range space. See appendix A of [21] for a description of the Frobenius normal form of a matrix. Usually this ϕ is smaller than the computer word size, of which m is a multiple; this makes

$m \geq n$ superfluous. Since we know that the algorithm by Beckermann and Labahn computes a basis for the solution space of our block Toeplitz system (2.2.2), which contains the generating polynomial f of $S_x = \{x^T A^i y\}_{i=0}^{\infty}$, the only remaining question is whether this polynomial generates $S = \{A^i y\}_{i=0}^{\infty}$, too. The number of terms of S_x used in the computation of f is $N/n + N/m + O(1)$. To be able to take a better look at the $O(1)$ term, we denote this number by $\lceil N/n \rceil + \lceil N/m \rceil + \Delta$. This Δ is called the shift parameter. Villard studies the relation between Δ and the probability that at least one column of f generates the corresponding column of S. In this case, we find at least one solution to $Ax = 0$. This is what Villard calls "success". The following lemma is Corollary 9.2 from [21]:

Lemma If $m \geq \min\{\phi, n\} + 2$ and $\Delta \geq 8$ then the probability of success is always greater than $\epsilon_0 = 0.03$.

Usually, $\phi << m$ and this gives a much stronger result ([21], Corollary 9.3):

Lemma If $m \geq 4\phi$, $\Delta \geq 8$ and $|\mathrm{Ker}(A)| \geq 4$ then the probability of success is greater than 0.6.

Compare this to the lower bound $\Phi(k)$ to the probability of success within k iterations in the scalar case; this is still the strongest lower bound known. It only gives $\Phi(2) > 0.3$ over $\mathbb{F}_2$. The above Lemma shows that in the block algorithm one iteration usually suffices. In practice, the block Wiedemann algorithm performs even better. This is why the only iteration in the block algorithm is the iteration in the generalized Berlekamp-Massey (or the FPHPS) algorithm, which is the generalization of the inner iteration in the scalar algorithm, i.e. the iteration in the Berlekamp-Massey algorithm. In the block algorithm there is no generalization of the outer iteration; we do not have to repeat the whole procedure several times with other random block vectors x. Moreover, it follows from Villard's analysis that the probability of success is an increasing function of Δ. Villard also gives a lower bound to the probability that f generates S, i.e. that we find n solutions, from which similar corollaries (with much lower probabilities) can be derived.

Chapter 3
Experiments with the Wiedemann algorithms

In this chapter we describe the implementations of the scalar and block versions of the Wiedemann algorithm which we used and discuss the results of computations using these computer programs.

1. The scalar Wiedemann algorithm

In the scalar case we only used random matrices as test material, because the smallest matrices available from integer factorization already were too big to solve with scalar Wiedemann within a reasonable time. These matrices are not very carefully generated in my program `wiedeman`: It simply multiplies numbers from a standard random number generator by a suitable sparsity factor to compute the number of zeroes which separate two ones. The program starts in the left upper corner and fills the matrix column by column with such a random number of zeroes, a one, another random number of zeroes and so on. The sparsity factor determines the sparsity of the matrix, hence the name. Columns containing only zeroes are deleted and generated again. Note that all-zero columns can occur only if the product of the sparsity factor and `rand_max` (the maximum output of the random number generator) exceeds the number of rows. The big advantage of this sloppy procedure is its speed. A disadvantage is that it does not generate matrices that closely resemble factorization matrices. In matrices from sieve-based factorization, the first few rows are much more dense, i.e. contain many more ones, than the other rows. The first 100 rows of a factorization matrix with 100,000 rows and columns may already contain 25% of all the ones. Later we show that in the block algorithm these quasi-random matrices behave just like matrices arising from factorization.

The matrices must have more columns than rows; the number of solutions requested must not exceed the column surplus to grant the existence of these solutions. First a dependency relation between the first $n+1$ columns is found. Then the program replaces the first column of the matrix involved in the dependency relation by one of the excess columns that were ignored when finding the previous solutions, and repeats the whole algorithm to find another column dependency.

Instead of random vectors the program uses unit vectors. These are chosen using the following procedure: Start with $u_1 = e_1$, the first unit vector; if $b_k = \mathbf{0}$ after the k^{th} iteration, we are ready; else $u_{k+1} = e_l$ where l is the smallest index of a non-zero entry of b_k. The first reason to use this procedure is that computing the inner product of a vector v with the unit vector e_l is easy; it simply is v_l, the l^{th} entry of v. The second reason for this choice is that we avoid the risk that the sequence $\{\langle u_{k+1}, A^i b_k \rangle\}_{i=0}^{2n-d_k}$ is the zero sequence, which gives $f_{k+1}(x) = 1$ as the trivial factor of $f(x)$, so we avoid the risk of doing a complete iteration in vain. Since the sequence is not all-zero, the Berlekamp-Massey algorithm will find a polynomial of degree at least one that is a factor of $f/(f_1 \dots f_k)$, i.e. one of the non-trivial factors of the minimum polynomial that we still had to find. This means that the algorithm always finds a solution to $Ax = 0$ in at most $n+1$ iterations, where A is an $n \times (n+1)$ matrix. Apart from this modification, the computer program is a straightforward implementation of the algorithm as described in the second chapter.

2. Experiments with the scalar algorithm

This version of the scalar Wiedemann algorithm is randomized only in the sense that we do not know precisely in how many iterations the solution will be found. We do know that the algorithm will succeed in at most $n+1$ iterations, and that we avoid iterations without result, so we may hope that the performance of the algorithm with this special choice of u_k is at least as good as that of the algorithm with random u_k. If this is true, we can use Wiedemann's lower bound for $\Phi(k)$ as a lower bound to the probability that a solution is found by our algorithm in at most k iterations. In most of the experiments, the algorithm found more than 85% of the solutions in at most two iterations. This is much better than the Wiedemann lower bound $\Phi(2) \geq 0.31$. Matrices for which $Ax = 0$ can easily be solved by hand, may sometimes be quite difficult to solve for the program. In one case, the third and fourth column of the matrix were identical; the algorithm found a dependency relation involving almost all other columns of the matrix, after four iterations.

We have computed five solutions to $Ax = 0$ for 200 random matrices A of approximately the same average sparsity (i.e., matrices generated with the same sparsity factor) with 100 rows and 105 columns. In the following table we summarize the results for groups of matrices with average column weight 42, 21 and 5. The approximate average column weight of the matrices is denoted by α. Usually, the algorithm performs much better than suggested by the lower bound for $\Phi(k)$:

	$\alpha = 42$		$\alpha = 21$		$\alpha = 5$		
k	freq.	cum.	freq.	cum.	freq.	cum.	$\Phi(k)$
1	542	542	522	522	484	484	-
2	353	895	365	887	381	865	0.307
3	98	993	100	987	117	982	0.712
4	6	999	13	1000	16	998	0.866
5	1	1000	0	1000	2	1000	0.935

The number of solutions found for matrices of this weight in exactly k steps is listed under frequency ("freq."); "cum." is the cumulative frequency, i.e. the number of solutions found in at most k steps. Wiedemann's lower bound for $\Phi(k)$ is listed in the last column. The algorithm seems to need more iterations for matrices with lower weight. To see this effect more clearly, I have repeated this experiment with matrices of much lower weight. To generate matrices with such a low column weight we have to choose such a large sparsity factor that the product of this factor with the maximum random number exceeds the number of rows. In this case all-zero columns may be generated. These are deleted and re-generated. This means that we refuse certain "random" matrices. To see whether this selection of the matrices influences the performance of the algorithm we repeated the experiment, this time allowing all-zero columns.

	no zero columns	$\alpha = 1.7$	zero columns	$\alpha = 2.1$	
k	frequency	cumulative	frequency	cumulative	$\Phi(k)$
1	50	50	38	38	-
2	137	187	139	177	0.307
3	255	442	261	438	0.712
4	246	688	274	712	0.866
5	180	868	157	869	0.935
6	91	959	81	950	0.968
7	33	992	35	985	0.984
8	6	998	13	998	0.992
9	2	1000	2	1000	0.996

Comparing these results to those in the previous table, we see that this scalar Wiedemann algorithm needs more iterations to solve $Ax = 0$ for sparse matrices than for dense matrices, regardless of the appearance of all-zero columns. We cannot rely on Wiedemann's lower bound to predict the number of iterations needed for our program, since we do not take random vectors as u_k. For (extremely) sparse matrices the algorithm needs more iterations and hence it takes longer than expected to solve the system for such matrices. A similar experiment with matrices of size 500×505 shows the same results: The algorithm finds the solutions in fewer steps if the matrices are more dense, see the next table.

	$\alpha = 1.6$		$\alpha = 3.7$		$\alpha = 9.7$		
k	freq.	cum.	freq.	cum.	freq.	cum.	$\Phi(k)$
1	11	11	341	341	476	476	-
2	47	58	438	779	402	878	0.307
3	111	169	174	953	110	988	0.712
4	195	364	43	996	12	1000	0.866
5	236	600	2	998	0	1000	0.935
6	197	797	2	1000	0	1000	0.968
7	108	905	0	1000	0	1000	0.984
8	59	964	0	1000	0	1000	0.992
9	28	992	0	1000	0	1000	0.996
> 9	8	1000	0	1000	0	1000	-

The running time of the algorithm is dominated by the matrix-vector products. These products each take αN bit-operations, so although the number of these products is a bit lower for dense matrices, the total number of bit-operations for dense matrices is still higher than for sparse matrices.

3. Coppersmith's block Wiedemann algorithm

For the original block Wiedemann algorithm by Coppersmith I used the program `WLSS2` by Austin Lobo [16]. It consists of four sub-programs which write their results to files and an auto-scheduler to execute these sub-programs consecutively. Together they solve $Ax = 0$ over $\mathbb{F}_2$. These are the four parts:

select selects x and z randomly such that $a^{(1)} = (x^T Ay)^T$ is nonsingular;

sequence computes the sequence $\{a^{(i)}\}_{i=0}^{2N/32n+2} = \{(x^T A^i y)^T\}_{i=1}^{2N/32n+3}$;

minpoly computes a matrix-polynomial f generating this sequence;

evaluate evaluates the solution by computing $f(A)y$.

In `select` random block vectors x and z are chosen; if $x^T Ay = x^T A^2 z$ is singular then other vectors x and z are chosen. The number of vectors in the block vectors x and z, m and n respectively, have to be multiples of 32 in this program. By taking $n = 32k$ the matrix-vectorproducts can be distributed over k computers; each computer runs `sequence` and `evaluate` with 32 of the columns of $y = Az$.

The matrix-block vector products in `sequence` are performed as described before. Having computed the matrix-block vector product $A^i y$, the computation of $x^T A^i y$ boils down to taking the "inner-product" of block vectors $x^T \times A^i y$. Computing this as a standard matrix product, i.e. using

$$w_{ij} = \sum_{k=1}^{N} u_{ki} v_{kj} \quad 1 \leq i \leq 32,\ 1 \leq j \leq 32,$$

requires $2 \times 32^2 \times N$ bit-operations. Of course this can be improved with a factor 32 by using block-arithmetic:

$$w_i = \sum_{k=1}^{N} u_{ki} v_k \quad 1 \leq i \leq 32.$$

This w_i is the i^{th} row of w. In this way the computation of $w = u^T v$ takes $2 \times 32 \times N = 64N$ bit-operations. A matrix-block vector product Av takes αN bit-operations for a sparse $N \times N$ matrix A with on average α ones per column. This means that the standard way to compute an inner-product of block vectors takes more time than a matrix-block vector product (usually $20 \leq \alpha \leq 40$). Coppersmith ([5, p342]) describes a better way to compute such an inner-product $u^T v$, using a look-up table C: Let u and v be $N \times 32$ block vectors, stored as vectors of N 32-bit words u_i and v_i, and let C be a 4×256 array of 32-bit words $C(k,j)$.

- Do for i=1, ... ,N:
  - Express u_i as the concatenation of four bytes: $u_i = (j_1|j_2|j_3|j_4)$;
  - $C(k, j_k) \leftarrow C(k, j_k)$ `XOR` v_i for $k = 1, \ldots, 4$.
- Write for i=1, ... ,32: $i = 8(k-1) + l$ with $1 \leq k \leq 4$ and $1 \leq l \leq 8$;

$$w_i = \sum_{j \in J_l} C(k,j) \quad \text{where } J_l = \{j : 0 \leq j \leq 255, \text{ the } l^{th} \text{ bit of } j \text{ is } 1\}.$$

In the first step the look-up table $C(k,j)$ is initialized; `XOR` amounts to bit-wise addition modulo 2. At the end of this step, the word $C(k,j)$ is the sum of the words in v corresponding to the words in u whose k^{th} byte has value j. The sum in the next step is again a bit-wise sum modulo 2. The i^{th} word of the product $w = u^T v$ is the sum of all v_i corresponding to u_i with a 1 as their i^{th} bit, i.e. the sum of all $C(k,j)$ where the l^{th} bit of j is 1 if $i = 8(k-1) + l$ with $1 \leq k \leq 4$ and $1 \leq l \leq 8$. So the product $w = u^T v$ is the 32×32 matrix w with w_i as its i^{th} row for $1 \leq i \leq 32$. Hence this computation gives the desired result. The number of bit-operations needed to compute w in this way is only $O(8N)$.

This technique is needed not only in `sequence`, but also in `minpoly`. We need to compute

$$\mathrm{Coeff}(t; f^{(t)}a) = \left(f^{(t)}_{\deg(f^{(t)})} | \cdots | f^{(t)}_0\right) \begin{pmatrix} a^{(t-\deg(f^{(t)}))} \\ \hline \vdots \\ \hline a^{(t)} \end{pmatrix} \quad \text{for } t_0 \leq t \leq \frac{N}{n} + \frac{N}{m}$$

where $f^{(t)}_l$ is the coefficient of x^l in $f^{(t)}(x)$. The degree of f increases from t_0 to N/n, so it will be on average about $N/2n$. The $a^{(i)}$ are $n \times m$ matrices, so the vectors have average length N/2. Compare the number of bit-operations needed for the computation of Coeff to that for the matrix-block vectorproducts: The latter ones require $(N/n + N/m + O(1)) \times \alpha N$ operations. Consider the case $m = n = 32, \alpha = 24$: Without look-up table, the computation of Coeff$(t; f^{(t)}a)$ takes $O(64 \times N/2 \times (N/n + N/m)^2) = O(2N^2)$ bit-operations. Using the look-up table it only requires $O(8 \times N/2 \times (N/n + N/m)) = O(N^2/4)$ bit-operations. Compare this to the $O((N/n + N/m) \times \alpha N) = O(\frac{3}{2}N^2)$ bit-operations of the matrix-block vectorproducts: without this look-up table, computing the minimum polynomial of our sequence would take more time than the computation of this sequence itself! Block Wiedemann would be 16 times faster than scalar Wiedemann, if the running time of the complete algorithm was almost completely spent on matrix-block vector multiplications. Without the look-up table, both the inner-products of block vectors and the computation of Coeff would take more time than the matrix multiplications. In this case, block Wiedemann would not be much faster than the scalar algorithm. This subprogram stops if Coeff$(t; f^{(t)}a) = \mathbf{0}^T$ for 20 consecutive values of t.

In `evaluate` a similar technique is used to compute the product of an $N \times 32$ block vector u with a 32×32 matrix w: again u and $v = uw$ are stored as vectors of length N with 32-bit words as elements; w is stored as 32 of these words, each containing a row of w.

- Do for j=1, ... ,255:

$$\begin{aligned} C(1,j) &= (j|0|0|0) \times w, & C(2,j) &= (0|j|0|0) \times w, \\ C(3,j) &= (0|0|j|0) \times w, & C(4,j) &= (0|0|0|j) \times w. \end{aligned}$$

- Do for i=1, ... ,N:
  - $u_i = (j_1|j_2|j_3|j_3)$,
  - $v_i = C(1,j_1)$ `XOR` $C(2,j_2)$ `XOR` $C(3,j_3)$ `XOR` $C(4,j_4)$.

It is clear that this gives the desired results:

$$\begin{aligned} v_i &= C(1,j_1) \texttt{ XOR } C(2,j_2) \texttt{ XOR } C(3,j_3) \texttt{ XOR } C(4,j_4) \\ &= (j_1|0|0|0) \times w + (0|j_2|0|0) \times w + (0|0|j_3|0) \times w + (0|0|0|j_4) \times w \\ &= ((j_1|0|0|0) + (0|j_2|0|0) + (0|0|j_3|0) + (0|0|0|j_4)) \times w \\ &= (j_1|j_2|j_3|j_3) \times w = u_i \times w. \end{aligned}$$

This computation takes about $7N$ bit-operations for large N. The standard way would take some $64N$ bit-operations, so again without this technique this product would take more bit-operations than a matrix-block vector product; with the technique, it takes less.

The computation of $f(A)b$ uses a Horner scheme:

$$A^n bf_n + \cdots + Abf_1 + bf_0 = bf_0 + A\left(bf_1 + A\left(bf_2 + A\left(\cdots + A\left(bf_n\right)\right)\ldots\right)\right).$$

This expression is developed backwards; we compute bf_n, $A(bf_n)$, $bf_{n-1} + A(bf_n)$, $A(bf_{n-1} + A(bf_n))$, $bf_{n-2} + A(bf_{n-1} + A(bf_n))$ and so on. Every column of $f(x)$ is a vector polynomial representing a possible solution to $Ax = 0$. The degrees of the columns of f are usually different. When a column has been developed completely, it is made inactive before we continue the Horner evaluation. If we would not do this, we would multiply vectors from A's nullspace by A and thus destroy the solutions found.

4. The FPHPS block Wiedemann algorithm

The program `FPHPSwie` is my implementation of the version of the block Wiedemann algorithm which uses the Fast Power Hermite Padé Solver by Beckermann and Labahn to compute a generating polynomial for the sequence of matrix-block vectorproducts instead of Coppersmith's generalization of the Berlekamp-Massey algorithm. This program only works with $m = n = 32$, i.e. x and y are $N \times 32$ matrices. The random block vector z is generated by a random number generator. The block vector x, however, is not chosen at random. I take the first 32 unity vectors as the columns of x, as Coppersmith suggests in his article ([5, p342]). By this choice of x we avoid the computation of the block vector inner products $x \times A^i y$; now $a^{(i)} = x \times A^i y$ simply is the matrix consisting of the first 32 rows of $A^i y$. The computation of $\{x^T A^i y\}_{i=0}^{N/16}$ simplifies to a repeated matrix-block vector product; we compute Ay, store its first 32 rows, multiply it by A to get $A^2 y$, store its first 32 rows etc.

The FPHPS-algorithm is implemented in the following way: The initialization in the first step is trivial. In the next step we compute the coefficient of x^σ in $\mathbf{P}_{i,\sigma}(x^{32})\mathbf{F}(x)$. This is the sum of the coefficients of x^σ in the 64 products $P_{i,\sigma,j}(x^{32})f_j(x)$ where $P_{i,\sigma,j}$ denotes the j^{th} polynomial from the tuple $\mathbf{P}_{i,\sigma}$. Remember the definition of $\mathbf{F} = (f_1, \ldots, f_{64})$:

$$f_i(x) = \begin{cases} x^{i-1} & i = 1, \ldots, 32 \\ (1, x, \ldots, x^{31})A_i(x^{32}) & i = 33, \ldots, 64 \end{cases}$$

where A_i is the i^{th} column of $A = \sum_i a^{(i)} x^i$. From the first 32 products, only the product $P_{i,\sigma,j}(x^{32})f_j(x)$ with $j \equiv \sigma \pmod{32}$ has a non-zero coefficient for x^σ; this coefficient is the coefficient of $x^{\lfloor \sigma/32 \rfloor}$ in $P_{i,\sigma}^{(j)}(x)$. In the last 32 products, the computation takes much more work. The coefficient of x^j in $f_i(x)$ is $a_{ij'}^{(\lfloor \sigma/32 \rfloor)}$, i.e. the element in position (i, j') of $a^{(\lfloor \sigma/32 \rfloor)}$ where $j' \equiv j \pmod{32}$, $0 \leq j' \leq 31$. This means that the coefficient of x^σ in the last 32 products $P_{i,\sigma,j}(x^{32})f_j(x)$ is

$$c_{i,\sigma} = \sum_{j=33}^{64} \sum_{k=0}^{\lfloor \sigma/32 \rfloor} \mathbf{P}_{i,\sigma,j}^{(k)} a_{j',j-33}^{(\lfloor \sigma/32 \rfloor - k)}$$

where $\mathbf{P}_{i,\sigma,j}^{(k)}$ is the coefficient of x^k in $P_{i,\sigma,j}$, the j^{th} polynomial from the tuple $\mathbf{P}_{i,\sigma}$. Computing $c_{i,\sigma}$ amounts to taking the sum of 32 inner products of vectors. Since these computations involve only vector manipulations we cannot use the block vector multiplication techniques with the look-up table described before. The computation of $c_{i,\sigma}$ is done for 32 values of i in parallel: We store $\mathbf{P}_{1,\sigma,j}^{(k)}, \ldots, \mathbf{P}_{32,\sigma,j}^{(k)}$ in a 32-bit word $\mathbf{P}_{\sigma,j}^{(k)}$ and compute $c = (c_1, \ldots, c_{32})$ simultaneously. For $j = 33, \ldots, 64$ we do the same. For $k = 0, 1, \ldots, \max\{\deg(\mathbf{P}_{i,\sigma})\}$ we take row j' of $a^{(\lfloor \sigma/32 \rfloor - k)}$ and consider it bit by bit; if $a_{j',j-33}^{(\lfloor \sigma/32 \rfloor - k)} = 1$ we add $\mathbf{P}_{\sigma,j}^{(k)}$ to our sum.

If all these coefficients are zero, this iteration is completed; else we determine for which index i with a non-zero coefficient the value of $d_{i,\sigma}$ is maximal. The tuple $\mathbf{P}_{i,\sigma}$ is added to the other tuples with non-zero coefficients, $\mathbf{P}_{i,\sigma}$ is multiplied by x (i.e. all coefficients of powers of x in polynomials in the tuple are moved one position up) and the $d_{l,\sigma}$ are adjusted accordingly. This completes this iteration.

The program stops the algorithm after $\lceil 2N/32 \rceil + \Delta$ iterations, where Δ is specified as a command line argument. If the evaluation shows that there are fewer solutions found than desired, the FPHPS algorithm continues and after a given number of iterations the solution is evaluated again. Both the extra number of iterations performed between two evaluations and the maximum number of iterations have to be specified as command line arguments. In the evaluation we use a Horner evaluation scheme as explained in the previous section and the look-up table technique to compute a block vector inner product.

5. Experiments with FPHPS block Wiedemann

The main goal of these experiments has been to investigate whether the FPHPS algorithm can obtain the right results with our particular choice of the random vector x, and if so, in how many iterations, i.e. with which shift parameter Δ. We used random matrices, generated exactly as in the scalar case. In the scalar case, sparsity seemed to be a crucial factor in the performance of the algorithm. To see whether this is true for the block algorithm too I compared $N \times N$ matrices with varying densities. In the following tables the number of solutions found for several values of the average column weight α and shift parameter Δ are listed in the case $N = 8200$ (left) and $N = 15{,}000$ (right):

$\alpha \setminus \Delta$	0	17	18	19	20	21
2.02	30	30	30	30	30	30
2.39	30	30	30	30	30	30
3.76	0	0	0	0	18	26
7.83	0	0	12	28	-	-
16.01	0	0	6	19	19	20
32.35	0	0	11	27	-	-
81.23	0	0	4	16	16	16
321.26	0	0	8	23	23	24

$\alpha \setminus \Delta$	0	10	30	40	50	60
1.72	16	16	16	16	16	16
2.65	0	17	17	17	17	17
5.64	0	0	0	0	0	13
14.63	0	0	0	24	24	24
24.63	0	0	0	25	25	25
74.44	0	0	0	22	23	23
148.91	0	0	0	23	24	24
296.80	0	0	0	25	25	25

In both tables we see a distinction between the two smaller values of α and the five bigger values of α: In the first case the algorithm finds solutions without or with only a few extra iterations; in the second case it takes about 20 ($N = 8200$) or 40 ($N = 15{,}000$) iterations to find some 20 solutions. Within each of these two "sparsity classes", the exact magnitude of α does not seem to influence the number of solutions found or the number of iterations needed. It is not clear yet whether $\alpha = 3.76$ ($N = 8200$) and $\alpha = 5.64$ ($N = 15{,}000$) belong to one of these classes, or to a third class. This gets clear in the next table, for $N = 25{,}000$:

$\alpha \setminus \Delta$	0	5	50	55	60	100	105	110	120
1.71	30	32	-	-	-	-	-	-	-
2.99	0	0	0	0	0	0	0	0	30
4.65	0	0	0	0	0	0	0	28	28
5.90	0	0	0	0	0	0	28	-	-
9.65	0	0	0	0	28	-	-	-	-
19.64	0	0	0	9	21	21	21	21	22
24.65	0	0	0	17	17	-	-	-	-
49.60	0	0	0	25	25	-	-	-	-
99.47	0	0	0	22	27	-	-	-	-
248.46	0	0	0	6	22	-	-	-	-

The program stops if the number of solutions or the number of iterations exceeds a given bound. This is why for most matrices the number of solutions is unknown for some of the larger values of Δ. Columns that give rise to non-zero solutions are not changed in the rest of the program, so we do know that the number of solutions for these values of Δ is at least as high as that for the last value of Δ for which it is known. For sufficiently dense matrices the algorithm needs about 55 iterations to solve the system $Ax = 0$. To solve this for the matrix A with $\alpha = 1.71$ extra iterations are not needed. The matrices that are not sufficiently dense to be solved in 55 iterations, but also not sufficiently sparse to make extra iterations superfluous, are in a class of their own; the algorithm needs more iterations to solve these matrices than to solve matrices from both other classes. We see globally the same picture for $N = 20{,}000$:

$\alpha \setminus \Delta$	0	5	40	50	60	70
2.16	30	30	30	30	30	30
2.72	18	21	21	21	21	21
4.65	0	0	0	0	0	27
7.54	0	0	0	28	-	-
9.64	0	0	0	0	0	29
19.64	0	0	0	25	26	-
39.60	0	0	0	0	0	0
79.50	0	0	0	21	21	21
159.06	0	0	0	20	20	20
264.63	0	0	0	26	-	-

Note that the algorithm did not find any solutions at all in one case; the random matrix may have been non-singular. Apart from this case, the algorithm needs 51 to 60 iterations and finds 20 to 26 solutions for the matrices with $\alpha > 10$, irrespective of the exact magnitude of α. For extremely small values of α ($\alpha < 3$) the algorithm finds solutions without extra iterations. Only the performance in the third class, i.e. for α neither extremely small nor sufficiently large, is quite unpredictable. This holds even more for $N = 30{,}000$:

$\alpha \setminus \Delta$	0	60	70	80	90	100	110	120	130	140	150	160
1.71	30	30	30	30	30	30	-	-	-	-	-	-
2.65	30	-	-	-	-	-	-	-	-	-	-	-
3.65	0	0	0	0	0	29	-	-	-	-	-	-
5.65	0	0	0	0	0	0	0	0	0	0	0	27
7.66	0	0	28	-	-	-	-	-	-	-	-	-
9.66	0	0	0	0	0	0	0	0	0	29	29	29
14.65	0	0	13	13	13	13	-	-	-	-	-	-
29.63	0	0	18	-	-	-	-	-	-	-	-	-
59.60	0	0	18	18	18	18	18	18	18	18	18	-
119.45	0	0	22	22	22	22	22	22	-	-	-	-
199.02	0	0	11	11	11	11	-	-	-	-	-	-

The performance in the third class can be quite bad for $N = 40{,}000$ too:

$\alpha \setminus \Delta$	0	10	80	90	95	100	110	120	150	160
1.72	30	30	30	30	30	30	30	30	30	30
2.32	30	32	-	-	-	-	-	-	-	-
3.66	0	0	0	0	0	0	0	0	0	28
7.66	0	0	0	28	28	30	-	-	-	-
9.66	0	0	0	28	-	-	-	-	-	-
15.65	0	0	0	0	6	16	-	-	-	-
19.67	0	0	0	9	10	10	10	10	10	10
39.65	0	0	0	24	24	24	25	25	25	25
132.81	0	0	0	21	21	21	21	22	-	-

The extremely sparse matrices are solved without extra iterations and the sufficiently dense matrices take about 70 ($N = 30{,}000$) or 90 ($N = 40{,}000$) extra iterations. As in the case $N = 20{,}000$, the number of extra iterations in the third class is sometimes the same and sometimes up to twice as much as in the class of the sufficiently dense matrices. The last table of this kind summarizes the results obtained for $N = 50{,}000$:

$\alpha \setminus \Delta$	0	10	100	110	120	150	160	270	280
2.16	30	32	-	-	-	-	-	-	-
2.17	1	1	1	1	1	1	1	-	-
4.66	0	0	0	0	0	0	0	0	27
6.80	0	0	0	0	0	0	30	-	-
7.99	0	0	0	29	-	-	-	-	-
9.66	0	0	0	22	29	-	-	-	-
24.65	0	0	0	20	20	20	-	-	-
49.64	0	0	0	17	17	17	-	-	-
99.59	0	0	0	17	17	-	-	-	-

These results confirm our observations: extremely sparse matrices are solved immediately, sufficiently dense matrices require a number of extra steps that does not depend on the exact sparsity and the matrices that are neither sufficiently dense nor extremely sparse take up to a small multiple of this number of extra steps, but compared to the total number of iterations (and the number of matrix-vector products needed) the differences are small. Theoretically, we need about $3N/32$ matrix-vectorproducts. For a 25,000 $\times$ 25,000 matrix the 120 extra terms that we needed in the worst case constitute about 5% of the expected number of terms. For $N = 50{,}000$ we needed 280 extra terms in the worst case; this is still less than 6% more than expected. Moreover, the algorithm performed even better if the density of the matrices is not extremely low, but more like that of actual factorization matrices. The number of extra iterations needed for sufficiently dense matrices does depend on the size of the matrix. We take a better look at the relation between N, the size of the matrix, and the number of extra terms needed, Δ, for the matrices that are sufficiently dense:

N	8200	15,000	20,000	25,000	30,000	40,000	50,000
Δ	18	35	45	53	65	90	105
N/Δ	456	429	444	472	462	444	476

Of course these values are estimated and interpolated. Nevertheless, they give a heuristic to estimate the number of extra terms needed beforehand. These numbers fit the relation $\Delta = N/450$ remarkably well. This means that usually we need only 2.4% more matrix-vectorproducts than expected theoretically if we use unit vectors instead of random vectors.

We also did experiments with 10,000$\times$10,000 matrices using each time 25 matrices of approximately the same sparsity instead of one, to improve the reliability of the results. In the next table we give the number of matrices with an average column weight of about α for which at least ten solutions were found after Δ extra iterations.

$\alpha \setminus \Delta$	0	5	10	15	20	25	30	35	40	45	50	55	60	200
2.2	23	24	24	24	24	24	24	24	24	24	24	-	-	-
3.0	4	4	6	6	9	11	12	13	15	17	17	-	-	-
4.7	0	0	1	2	2	4	5	9	9	15	15	17	23	25
6.3	0	0	0	0	9	9	13	16	16	22	22	-	-	-
9.7	0	0	0	0	0	21	21	22	22	24	24	24	25	25
19.7	0	0	0	0	0	18	18	18	18	18	18	18	18	18
39.5	0	0	0	0	0	20	20	20	20	20	20	20	20	20
79.3	0	0	0	0	0	19	19	19	19	19	19	-	-	-

This confirms our other observations quite well: For sufficiently dense matrices, finding dependencies always takes 21 to 25 extra iterations, regardless of the exact magnitude. This accords with our heuristic: $10{,}000/450 \approx 22$. The extremely sparse matrices ($\alpha \approx 2.2$) are almost all solved without extra iterations. The matrices with $\alpha \approx 3.0$, $\alpha \approx 4.7$ and $\alpha \approx 6.3$ belong to what we called the third class; here the number of extra iterations varies, but usually it is bigger than for the sufficiently dense matrices.

The bound for extreme sparsity is an average column weight of about 2.8 or 2.9. It is almost impossible to determine an exact bound for being sufficiently dense; usually an average column weight of about 7 or 7.5 seems to be sufficient, although sometimes matrices with $\alpha \approx 9.7$ or $\alpha \approx 15$ seem to belong to the third class. Moreover, these bounds seem to be independent of the matrix size; at least there is no indication that bigger matrices need a higher average column weight to be sufficiently dense, i.e. to be solved with $\Delta = N/450$ extra iterations.

6. Comparing the block algorithms

To compare the two different versions of the block Wiedemann algorithm we solved the 61,249 $\times$ 85,095 matrix arising from the factorization of $r_{71} = (10^{71} - 1)/9$ with each of these algorithms. This matrix had already been solved by Montgomery's block Lanczos [18] using the program `gauss` by Montgomery which is a part of the implementation of the General Number Field Sieve developed at Oregon/CWI [8]. The matrix has quite a lot of excess columns. The program `FPHPSwie` solved the truncated matrix containing only the first 61,249 columns of the matrix. The algorithm needed 130 extra iterations, which accords well with our heuristic (61,249/450 $\approx$ 136). Moreover, we solved two random matrices of the same size and weight. One matrix took 140 extra iterations, the other 130. This indicates that the number of iterations needed for our sloppily generated random matrices is the same as that for factorization matrices. Lobo's program `WLSS2` was able to solve the system with only 100 excess columns. Block Lanczos needed 10,000 excess columns to solve the system. To compare these last two methods I also solved the system with the complete matrix with both of them. In the following table are the running times (in hours:minutes) for the four parts of the algorithms, the number of non-zero solutions and the column surplus, i.e. the number of excess columns. In the column for FPHPS we give the running time for the parts of the program corresponding to the subprograms of `WLSS2`. Note that there is no running time for `select` in this column; we simply take the first 32 unity vectors as our random block vector x and we do not have to test whether $x^T Ay$ is singular. Generating y is considered as a part of `sequence` in this table. We tried to use unity vectors instead of a random block vector in Lobo's program, too. The results from this experiment are in the column with as remark "e_i". As explained before, the computation of the sequence and the evaluation of the minimum polynomial can be distributed over several computers by taking for m and n suitable multiples of 32. The remarks "$m = n = 96$" and "$m = n = 192$" indicate that in these cases the matrix multiplications have been distributed over 3 and 6 computers, respectively. The times given are the running times for one computer. We used Silicon Graphics Indy workstations with a 100MHz IP22 processor.

program	FPHPS	WLSS2	WLSS2	WLSS2	WLSS2	WLSS2
surplus	0	100	100	23,254	23,254	23,254
remark	-	-	$m = n = 192$	-	$m = n = 96$	e_i
select	-	0:01	0:02	0:01	0:01	-
sequence	1:10	1:41	0:21	3:00	1:06	2:59
minpoly	3:43	0:42	2:01	0:58	1:29	1:52
evaluate	0:41	0:52	0:11	1:06	0:25	3:23
total	5:34	3:16	2:35	5:05	3:01	8:15
solutions	30	32	192	32	96	17

We start with comparing the two different versions of block Wiedemann. This means we have to compare the first and second column. The running time of

Lobo's program WLSS2, i.e. of Coppersmith's block Wiedemann, is roughly determined by the two parts which compute matrix-block vector products: `sequence` and `evaluate`. For the FPHPS algorithm this is not true: the computation of the minimum polynomial of the sequence of 32×32 matrices by Beckermann and Labahn's algorithm takes twice the time of the other parts. This makes the FPHPS algorithm considerably slower than Coppersmith's algorithm. The difference in running time for the two ways to find the minimum polynomial is mainly due to the fact that we cannot use a look-up table technique in the FPHPS algorithm. Without this technique, Coppersmith's generalization of the Berlekamp-Massey algorithm would also take more time than the other parts of the algorithm together. The `FPHPSwie` program performs the computation of the sequence $\{x^T A^i y\}_{i=0}^{2N/32}$ about 30% faster than the program `sequence`, mainly because the first program only computes $\{A^i y\}_{i=0}^{2N/32}$ and then takes the trivial inner products with unit vectors, whereas the latter program has to compute $x^T \times A^i y$ for a random vector x for $i = 0, 1, \ldots, 2N/32$. Moreover, the $a^{(i)}$ are transposed in the latter program. The difference in the running times for the evaluation of the solutions is comparatively small.

If we compare the running time of `WLSS2` using one computer with that using several computers, it is immediately clear that the overall improvement of this parallelization is limited; the total running time using six computers is only 20% less than with one computer. The running times for the parallelized parts decrease almost by a factor six indeed, but the running time for `minpoly` (which is performed on a single computer) increases strongly. It is clear that using even more computers will not reduce the running time much further; it will reduce the running time of parts of the program which are not time-critical, but increase the running time of the part of the program which already takes most of the total running time, `minpoly`. As a matter of fact, comparing the fourth and fifth column of the table we see that for the complete matrix `minpoly` already uses half of the total time if we use three computers. This means that using more than three computers will not decrease the running time much. Although the increase in speed is quite small, this parallelization has another possible advantage: the coefficients of the polynomial found using k computers has $32k$ columns, so we find up to $32k$ solutions instead of up to 32. If we want to find more solutions with the `FPHPSwie` program, we can take advantage of the fact that the FPHPS algorithm produces a σ-basis containing 64 elements and evaluate the other 32 PHPA's from the σ-basis too. This doubles the time for the evaluation, without affecting the other parts of the program. The number of solutions found may increase, but the experiments with random matrices showed that usually almost all solutions are found in the last 32 elements of the σ-bases. Only in the case of the extremely sparse matrices most solutions are found in the first 32 PHPA's.

Now we take a look at the experiment where we tried to combine the advantages of the two versions of block Wiedemann by using unit vectors in Coppersmith's algorithm. This means we have to compare the sixth column of the table with the fourth. The running time for the computation of the sequence is the same in both cases, because the program computes the inner products with the unit vectors in the standard way. This can be improved, in which case we expect an improvement similar to the difference between the running times for `sequence` in the second and the first column. The time needed for the computation of the minimum polynomial doubles if we use unit vectors instead of a random block vector, but finally the program succeeds to find a suitable polynomial, although it produces thousands of warnings that matrices that should be non-singular are singular. The degree of the polynomial is higher than expected so we need more matrix-block vector products in the evaluation of the solution than with the random vectors, but finally, we find 17 non-trivial dependency relations in our matrix. Unfortunately, the possible gain in computing time from the easier computations of the block vector inner products is superseeded by the extra time needed to find and evaluate the minimum polynomial.

Finally, we try to compare these running times with those of block Lanczos. In the look-up table computations in `WLSS2` a 32-bit word is split into four words of eight bits; in the block Lanczos program, a 32-bit word is split into three words of length 11 (where the last bit of the last word is not used). This makes these techniques even more effective. In theory, block Wiedemann needs about $3N/32$ matrix-block vector products and block Lanczos only $2N/32$. Moreover, the number of bit-operations needed for block Lanczos is even more dominated by the matrix-block vector products than that for block Wiedemann. So theoretically we expect block Wiedemann to use more than 50% more time than block Lanczos. In fact, it only took 64 minutes to solve the complete matrix with block Lanczos. This is almost five times faster than the 5h05 for Lobo's block Wiedemann. In this case block Lanczos had an extra handicap: It simulated a 64-bit computer (splitting a 64 bit word into five words of length 13). However, Coppersmith's block Wiedemann found 32 solutions in 3h16, using only 100 excess columns. Block Lanczos needed 10,000 surplus columns to find 8 solutions, but this took only 35 minutes (without simulating a 64-bit computer). Without the handicap, the block Lanczos program is even more than five times faster than `WLSS2` and more than nine times faster than `FPHPSwie`.

Conclusions

We have implemented a new version of the block Wiedemann algorithm (using the FPHPS algorithm by Beckermann and Labahn) to solve large systems of linear equations with sparse coefficient matrices over $\mathbb{F}_2$ and compared the performance of this algorithm with that of Coppersmith's block Wiedemann and that of Montgomery's block Lanczos algorithm. First we experimented with the scalar (i.e. standard) Wiedemann algorithm. This showed better practical performance than expected when considering Wiedemann's lower bound for the probability of success. We used a particular way to choose unit vectors instead of random vectors (see section 3.1). This may have improved the performance of the algorithm. The most important observation, however, is the fact that the sparsity of the matrix influences the number of (outer) iterations, i.e. the number of factors of the minimum polynomial of the Krylov sequence which we have to find to determine this polynomial and hence to solve our system. For extremely sparse matrices the performance of our version of the Wiedemann algorithm is worse than the lower bound. This must have been caused by our way of choosing "random" vectors.

Using block algorithms (i.e. working with blocks of 32 bits instead of single bits to perform computations more efficiently) we do not have to choose random block vectors several times; we find solutions with our first choice. This means that there is no equivalent in the block case of the outer iterations of the scalar algorithm. Thus, using blocks instead of single vectors improves the performance of the Wiedemann algorithm. This is already observed in [15], [11]. In Coppersmith's block Wiedemann algorithm we have to choose the random vectors obeying the condition that $x^T Ay$ is non-singular. In the FPHPS version of the block Wiedemann algorithm there is no such condition and we can use the block vector consisting of the first 32 unit vectors. This makes the computation of the block vector product $x^T \times A^i y$ trivial, but we have to perform a few percent extra iterations. Villard's theoretical analysis shows that this would not be necessary with high probability if either our matrix or our blocking factor m would fulfill a condition which is hard to verify. The number of extra iterations needed is about $N/450$ for "sufficiently dense" matrices. In our experiments (matrix dimensions from 8200 to 50,000) sufficiently dense meant an average column weight of at least 7. For extremely sparse matrices (an average column weight of at most 2.9) the FPHPS program solved the system (almost) without extra iterations. Between these two classes of matrices, we find a third class where the performance of the algorithm varies. We needed up to two times as many extra iterations as usual to solve systems with matrices from this third class. We see no clues that the bounds between the sparsity classes vary with the matrix size.

The FPHPS program computes the sequence $\{x^T A^i y\}_{i=0}^{2N/32+\Delta}$ 30% faster than Lobo's implementation of Coppersmith's block Wiedemann algorithm, the program `WLSS2`, because the multiplication with x^T is trivial (in this case, x^T is a unit vector) and because the $a^{(i)}$ do not have to be inverted in `FPHPSwie`. The number of extra iterations needed because our x is not a "good blocking vector" is very small for most matrices. Yet, `FPHPSwie` is much slower than Lobo's program since the computation of the minimum polynomial with the FPHPS algorithm takes much more time than Coppersmith's algorithm. In the FPHPS algorithm we manipulate polynomials with scalar coefficients. Although we can perform 32 of these operations in parallel by storing the polynomials in blocks, we cannot use the techniques with the look-up table for the computation of matrix products explained in section 3.3. These techniques are essential for the speed of Coppersmith's block Wiedemann algorithm.

Compared with Montgomery's block Lanczos, the block Wiedemann algorithms are much slower. Theoretically we expected block Wiedemann to use at least about twice the running time of block Lanczos, but in practice Montgomery's block Lanczos program is five times as fast as Lobo's program for Coppersmith's block Wiedemann. This is partly caused by the very efficient implementation Montgomery made of his block Lanczos algorithm. Both versions of block Wiedemann seem to be better in solving systems with few excess columns than block Lanczos. However, factorization matrices normally have quite a lot of excess columns, since it is practically infeasible to stop sieving exactly at the moment when enough columns are found. Moreover, these matrices are "filtered" by combining relations [8], which results in smaller but denser and almost square matrices. Usually, block Lanczos is able to solve these filtered matrices. This means that in practice, the cases in which block Wiedemann beats block Lanczos do not occur.

References

1. B. Beckermann and G. Labahn. A uniform approach for Hermite Padé and simultaneous Padé approximants and their matrix generalizations. Numerical Algorithms 3 (1992), pages 45-54.
2. B. Beckermann and G. Labahn. A uniform approach for the fast computation of matrix-type Padé approximants. SIAM Journal of Matrix Analysis and Applications 15, 3 (1994), pages 804-823.
3. E.R. Berlekamp. Algebraic Coding Theory. McGraw-Hill, New York, 1968.
4. D. Coppersmith. Solving linear equations over GF(2): block Lanczos algorithm. Linear Algebra and its Applications 192 (1993), pages 33-60.
5. D. Coppersmith. Solving linear equations over GF(2) via block Wiedemann algorithm. Mathematics of Computation 62, 205 (1994), pages 333-350.
6. J. Cowie, B. Dodson, R.-M. Elkenbracht-Huizing, A.K. Lenstra, P.L. Montgomery and J. Zayer. A world wide number field sieve factoring record: on to 512 bits. In K. Kim and T. Matsumoto, editors, Advances in Cryptology - Asiacrypt'96, Lecture Notes in Computer Science 1163, pages 183-215. Springer-Verlag, Berlin, 1995.
7. J.L. Dornstetter. On the equivalence between Berlekamp's and Euclid's algorithms. IEEE Transactions on Information Theory 33 (1987), pages 428-431.
8. R.-M. Elkenbracht-Huizing. An implementation of the number field sieve. Experimental Mathematics, 5, 3 (1996), pages 231-253.
9. M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. Journal of Research of the National Bureau of Standards 49 (1952), pages 409-436.
10. E. Kaltofen. Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. Mathematics of Computation 64, 210 (1995), pages 777-806.
11. E. Kaltofen and A. Lobo. Distributed matrix-free solution of large sparse linear systems over finite fields. In A.M. Tentner, editor, High Performance Computing 1996, San Diego, pages 244-247. Society for Computer Simulation, Simulation Councils, Inc, 1996.
12. B.M. LaMacchia and A.M. Odlyzko. Solving large sparse linear systems over

finite fields. In A.J. Meneskes and S.A. Vanstone, editors, Advances in Cryptology - Proceedings of Crypto'90, Lecture Notes in Computer Science 537, pages 109-133. Springer-Verlag, Berlin, 1991.

13. C. Lanczos. Solution of systems of linear equations by minimized iterations. Journal of Research of the National Bureau of Standards 49 (1952), pages 33-53.
14. A.K. Lenstra, H.W. Lenstra Jr., M.S. Manasse and J.M. Pollard. The number field sieve. In A.K. Lenstra and H.W. Lenstra Jr., editors, The development of the number field sieve, Lecture Notes in Mathematics 1554, pages 11-42. Springer-Verlag, Berlin, 1993.
15. A. Lobo. Matrix-free linear system solving and applications to symbolic computation. PhD thesis, Dept. Comp. Sc., Rensselaer Polytech. Instit., Troy, New York, Dec. 1995.
16. A. Lobo. WLSS2 computer package and manual. Available by anonymous ftp://`ftp.cs.rpi.edu/wlss2.sh.Z.uu`.
17. J.L. Massey. Shift-register synthesis and BCH decoding. IEEE Transactions on Information Theory 15 (1969), pages 122-127.
18. P.L. Montgomery. A block Lanczos algorithm for finding dependencies over GF(2). In L.C. Guillou and J.-J. Quisquater, editors, Advances in Cryptology - Eurocrypt'95, Lecture Notes in Computer Science 921, pages 106-120. Springer-Verlag, Berlin, 1995.
19. C. Pomerance. Analysis and comparison of some integer factoring algorithms. In H.W. Lenstra, Jr. and R. Tijdeman, editors, Computational methods in number theory, Mathematical Centre Tracts 154, pages 89-139. Mathematisch Centrum, Amsterdam, 1982.
20. C. Pomerance. The Quadratic Sieve Factoring Algorithm. In T. Beth, N. Cot and I. Ingemarsson, editors, Advances in Cryptology - Proceedings of Eurocrypt'84, Lecture Notes in Computer Science 209, pages 169-182. Springer-Verlag, Berlin, 1985.
21. G. Villard. A study of Coppersmith's block Wiedemann algorithm using matrix polynomials, Feb. 1997. Rapport de Recherche 975 IMAG Grenoble, France. Available by anonymous ftp at `ftp.imag.fr` in the directory `/pub/CALCUL_FORMEL`.
22. D. Wiedemann. Solving sparse linear equations over finite fields. IEEE Transactions on Information Theory 32 (1986), pages 54-62.
23. N. Zierler. Linear recurring sequences and error-correcting codes. In H.B. Mann, editor, Error correcting codes, pages 47-59. Wiley, New York, 1968.