# Direct Solution of the (11,9,8)-MinRank problem by the Block Wiedemann Algorithm in Magma with a Tesla GPU

Allan Steel
School of Mathematics & Statistics
University of Sydney
allan.steel@sydney.edu.au

## ABSTRACT

We show how some very large multivariate polynomial systems over finite fields can be solved by Gröbner basis techniques coupled with the Block Wiedemann algorithm, thus extending the Wiedemann-based 'Sparse FGLM' approach of Faugère and Mou. The main components of our approach are a dense variant of the Faugère $F_4$ Gröbner basis algorithm and the Block Wiedemann algorithm, which have been implemented within the Magma Computer Algebra System (released in version V2.20 in late 2014). A major feature of the algorithms is that they map much of the computation to dense matrix multiplication, and this allows dramatic speedups to be achieved for large examples when an Nvidia Tesla GPU is available. As a result, the Magma implementation can directly solve a 16-bit random instance of the Courtois (11,9,8)-MinRank Challenge C in about 15.1 hours with a single Intel Sandybridge CPU core coupled with an Nvidia Tesla K40 GPU.

## 1. INTRODUCTION

Let $K$ be a finite field and suppose that $I$ is a zero-dimensional ideal of the multivariate polynomial ring $R = K[x_1, \dots, x_k]$. Let *the degree of* $I$ denote the cardinality of the affine variety of $I$ over an algebraic closure of $K$, which equals the dimension of $R/I$ as a vector space over $K$.

One way to compute solutions to the multivariate polynomial system which corresponds to $I$, which we will call the **direct method**, involves two main steps: (1) one computes the Gröbner basis (GB) of $I$ w.r.t. a degree ordering; (2) using the GB of $I$, it is straightforward to work within the quotient algebra $R/I$ and to compute the matrix $A$ representing the multiplication action of the coset of $x_i$ in $R/I$ for some $i$; the roots of the minimal polynomial of $A$ yield the values that $x_i$ must take in each solution of the original system. However, as the degree $N$ of the ideal $I$ grows very large (into the thousands), the second step can become very difficult in practice, since it involves computing within an algebra of dimension $N$.

To improve the second step of the direct method, Faugère and Mou [6, 7] introduced the technique of applying the Wiedemann algorithm [18] to compute the minimal polynomial of a typical representation matrix $A$. They showed that $A$ is often moderately sparse (usually with density under 30%) and in this case the Wiedemann algorithm can be very effective both in time and memory usage for computing the minimal polynomial of $A$ or, more generally, for computing the lexicographical GB of the ideal (the 'Sparse FGLM' algorithm).

This paper shows that by using a Block Wiedemann algorithm [3] instead of the simple Wiedemann algorithm, one can now effectively compute solutions to the polynomial system described by $I$ in a moderate number of hours with an Nvidia Tesla GPU, even when the degree of $I$ is of the order of 250000. This algorithm and a dense variant of the Faugère $F_4$ Gröbner basis algorithm [5] have been implemented by the author in the Magma Computer Algebra System [2] (released in version V2.20 in late 2014). Since these algorithms map much of the computation to dense matrix multiplication, this allows a GPU to be exploited when available. We describe below how the Magma implementation can solve a 16-bit random instance of the Courtois (11,9,8)-MinRank Challenge C in about 15.1 hours on a single Intel Sandybridge CPU core coupled with an Nvidia Tesla K40 GPU.

## 2. THE MINRANK PROBLEM

The square $(n, k, r)$-MinRank problem over a field $K$ can be stated as follows: given a square linear matrix of size $n$ whose entries are multivariate polynomials in $K[x_1, \dots, x_k]$ of total degree 1, the goal is to find the set of points in $K^k$ such that the evaluation of the matrix at these points has rank less than or equal to $r$. In 2010, Faugère, Safey and Spaenlehauer [8] analyzed the use of GB techniques to solve this problem. They showed that one can compute the minors of the symbolic linear matrix and solve the corresponding multivariate polynomial system by GB techniques to solve the MinRank problem. See also [9] for further analysis.

Courtois [4] proposed a series of cryptographic challenges based on the MinRank problem in 2001; the corresponding multivariate ideals are zero-dimensional. Challenges A and B were easily solved by Faugère et al [8], while Challenge

*PASCO '15, July 10 - 12, 2015, Bath, United Kingdom*
© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-3599-7/15/07... $15.00
DOI: http://dx.doi.org/10.1145/2790282.2791392

C is very much harder since it involves solving a MinRank problem with parameters $(n, k, r) = (11, 9, 8)$ so that the degree of the ideal is 259545. Faugère et al considered this problem for the largest 16-bit prime field $K = \mathbb{F}_{65521}$ and concluded that the **direct method** (computing the *grevlex* GB of the original zero-dimensional ideal and then working in the quotient algebra $R/I$) was infeasible at the time, since the dimension of $R/I$ is 259545. They instead suggested that the best attack at the time would be to use the **evaluation method**: perform $\#K$ (65521) GB computations of the system with one variable evaluated at each value of $K$ (the GB of each evaluated ideal collapses to $\{1\}$ or a set of linear polynomials which give a full solution). They estimated that they could solve Challenge C with a runtime of 166.7 core years, or in about 238 days on a 256-core cluster with 3.2GHz Intel Xeon processors. As a comparison, with our dense variant of the $F_4$ algorithm, together with some storing of information for repeated computations as in the algorithm of Joux and Vitse [10], Magma can solve Challenge C with the evaluation method in an estimated 3.4 core years or in about 4.8 days on a 256-core cluster with 3.1GHz Intel Xeon processors (and without using a GPU): this is a speedup of about a factor of 50 over the estimate of Faugère et al for the evaluation method.

In contrast, we will show that the challenge is now very easy to solve with the direct method, when using the Block Wiedemann algorithm with a GPU. Even without a GPU, it is almost 150 times faster than the above evaluation method estimate for Magma and about 7000 times faster than the evaluation method estimate of Faugère et al. But it also has the major advantage that it is not particularly sensitive to the size of the field $K$. Changing the base field to $\mathbb{F}_p$, where $p$ has 23 bits, would make very little difference to the running time for the direct method in the Magma implementation, but would make a huge difference to the running time of the evaluation method (in any implementation), since very many more GB computations on the evaluation ideals would be needed, of course.

## 3. BLOCK WIEDEMANN OVERVIEW

This section gives a very brief overview of the Block Wiedemann algorithm, in the form in which it is applied in the Magma implementation; see the references below for further details on the algorithm.

Let $A \in \mathcal{M}_N(K)$ where $K$ is a finite field. In Wiedemann's original algorithm [18], one computes the sequence:

$$a_i = vA^i w^{tr}, \quad 0 \le i \le 2N - 1,$$

where $v, w$ are fixed random row vectors in $K^N$. Assuming that $K$ is large enough, then with high probability the minimal polynomial of $A$ can be computed by the Berlekamp-Massey algorithm as the lowest degree polynomial which generates the sequence $(a_0, \ldots, a_{2N-1})$.

Coppersmith [3] introduced a 'Block Wiedemann' variation of the algorithm, where one replaces the random vectors $v, w$ above with a random $m_1 \times N$ matrix $V$ and a random $m_2 \times N$ matrix $W$ respectively. We will fix on the simple symmetrical case where $m_1 = m_2 = m$ in this paper. Coppersmith showed that in this case, by letting $P = \lceil \frac{2N}{m} \rceil + \epsilon$, where $\epsilon$ is a small integer constant, one only need compute the following sequence with $P$ terms instead:

$$s_i = VA^i W^{tr}, \quad 0 \le i < P.$$

Here the $s_i$ are $m \times m$ matrices with entries in $K$, so each $s_i$ contains more information than the $a_i$ scalars above. One can also exploit fast matrix multiplication and/or parallelization when computing the $s_i$. Let

$$S(x) = \sum_{i=0}^{P-1} s_i x^i \in \mathcal{M}_m(K[x]).$$

Then with high probability, the minimal polynomial of $A$ can be recovered from $S(x)$, assuming the field $K$ is large enough. (Increasing $\epsilon$ increases the probability of correctness.) See [11, 16] for further analysis.

We use lattice basis reduction for vectors with entries in $K[x]$ to recover the minimal polynomial of $A$ from $S(x)$. This is achieved by a $K[x]$-analogue of the LLL [12] lattice basis reduction algorithm; see [13], for example. Assuming the precision of the entries of $S(x)$ is at least $P = \lceil \frac{2N}{m} \rceil + \epsilon$, we can apply the $K[x]$-analogue of LLL to the rows of the following $(2m) \times (2m)$ matrix:

$$\begin{pmatrix} I_m & S(x) \\ 0 & x^P I_m \end{pmatrix}.$$

The resulting LLL-reduced matrix, with rows ordered by increasing degree, is of the form:

$$\begin{pmatrix} L(x) & R(x) \\ U_1(x) & U_2(x) \end{pmatrix},$$

where $L(x), R(x) \in \mathcal{M}_m(K[x])$, and this gives the right rational form $L(x)^{-1}R(x) \equiv S(x) \bmod x^P$. Assuming that $K$ and $\epsilon$ are large enough, then with high probability the minimal polynomial of $A$ equals the LCM $d(x)$ of the denominators of $L(x)^{-1}$, considered as a matrix with entries in the rational function field $K(x)$. Now since in our application we only seek the roots of the minimal polynomial (and multiplicities are not relevant), we can compute the determinant of $L(x)$ which will have the same roots as $d(x)$. The roots of this determinant give values for the appropriate variable of the polynomial ring which we are solving for, and for each such root we can compute the full solutions (values for all coordinates) by another GB computation with that variable evaluated at the root (as in the evaluation method above).

## 4. THE MAGMA IMPLEMENTATION

We note some basic points about our implementation of the algorithms in Magma.

To compute the *grevlex* GB of the input ideal $I$, we use a dense variant of the Faugère $F_4$ Gröbner basis algorithm [5]. This variant was developed by the author in 2013 and will be described in detail in a forthcoming paper. Its most important feature is that the linear algebra phase of each main step uses only dense matrix multiplications, so the algorithm is greatly boosted when large matrices are involved and a GPU is present for fast matrix multiplication. The algorithm is only applicable to input systems which are dense; i.e., systems such that if the input polynomials are written as a matrix with columns labelled by the monomials, then this matrix has close to 100% density. Now the zero-dimensional systems coming from the MinRank problems are certainly extremely dense so this algorithm is highly applicable to them.

For the data collection phase for the Block Wiedemann algorithm (computing the $s_i$ which make up $S(x)$), let $A$ be the matrix representing the multiplication action of the chosen element of the quotient algebra $R/I$ (typically the coset of the last variable of $R$ is chosen). Then $A$ is an $N \times N$ matrix with entries in $K$, where $N$ is the degree of $I$. Now $A$ very often has many rows which are unit vectors and only $b << N$ rows which are dense, so as in [7, Sec. 6.1], a special compact representation of $A$ can be set up which includes a $b \times N$ dense matrix $B$ derived from the dense rows of $A$ plus information on the unit vectors. Then multiplication of an $m \times N$ matrix $V$ by $A$ can effectively be computed by one dense multiplication of an $m \times b$ submatrix of $V$ by $B$ with some other easy house keeping. Thus $A$ is never explicitly constructed and this saves a lot of time and memory. Once this compact representation of $A$ is set up, one simply computes a series of dense matrix multiplications and collects the data to form the matrix $S(x) \in \mathcal{M}_m(K[x])$.

For the $K[x]$-analogue of the LLL algorithm, we use an asymptotically-fast recursive algorithm which is in the spirit of Thomé's 'Half-GCD'-like version of Coppersmith's algorithm [14, 15]. Our implementation of the LLL algorithm recurses by reducing the degrees of the polynomial entries at each level (not by reducing the size of the matrix; that remains the same at every level). After recursing on a matrix whose entries are the upper halves of the original polynomial entries, the main matrix is updated by multiplying the relevant transformation matrix by the matrix whose entries are the lower halves of the original entries; the algorithm then recurses on the updated main matrix, etc., until the base case is reached. The algorithm thus asymptotically reduces the computation to matrix multiplication over $K[x]$, and such a multiplication is performed in Magma by a standard evaluation/interpolation algorithm: the input matrices are evaluated at sufficiently many points and the relevant scalar matrices are multiplied over $K$, and the result product over $K[x]$ is obtained by an univariate polynomial interpolation for each entry. For the base case of the LLL algorithm (when the maximum degree of the entries is at most 20) we use the $F_4$ algorithm to compute the module GB of a submodule of $(K[x])^c$ which corresponds to the lattice generated by the rows of the input matrix with $c$ columns; it is easy to see that computing such a GB with the term-over-position (TOP) module monomial order [1, Sec. 3.5] is equivalent to the LLL-reduction algorithm described by Paulus [13]. Using the $F_4$ algorithm means that highly optimized linear algebra is employed for the base scalar operations in $K$, so this is a very efficient LLL algorithm for the base case.

The determinant of $L$ is simply read off the diagonal of the Hermite Normal Form (HNF) of $L$ (we can ignore scalar multiples since we only seek roots of the determinant polynomial). The HNF itself is computed recursively as follows: the above $K[x]$-analogue of the LLL algorithm is applied to the left half of $L$ and the result is reduced to HNF (which is usually trivial to do as the LLL result typically has only unit vectors since there are twice as many rows as columns). The left half of $L$ is replaced with this HNF matrix and the right half of $L$ is multiplied by the appropriate transformation matrix (the degrees of the entries in the transformation matrix are controlled by the LLL algorithm). The algorithm then recurses on the bottom right quarter of $L$ and proceeds until the size of the matrix to be reduced is at most 10, in which case a classical HNF algorithm is then used. This recursive algorithm is quite effective when both the input dimension and the degrees of the input polynomials are large (both in the hundreds).

Finally, we note that using the Block Wiedemann in our context does not take significantly more memory than the simple Wiedemann algorithm; the *grevlex* GB and the matrix $B$ for the compact representation of $A$ (common to both versions of the Wiedemann algorithm) take much more memory than anything else in any case.

## 5. RESULTS

We now describe the results of running our Magma implementation. All computations below are done with coefficients in the largest 16-bit prime finite field $K = \mathbb{F}_{65521}$. We first note the hardware used for our runs (and the labels used for the columns in Tables 2 and 3).

- *CPU*: a single Intel E5-2687W (3.1GHz) Sandybridge processor. *Only one core is used in all computations.* The ATLAS [17] (Automatically Tuned Linear Algebra Software) package is used for dense matrix multiplications over $K$. The CPU is on a server node with 384GB memory.

- *C2075*: a single Nvidia C2075 Tesla GPU with 448 threads and 6GB memory. The CUBLAS dense linear algebra package for GPUs is used for large dense matrix multiplications over $K$ (typically when all dimensions of the matrices are more than 1000). This is often 15 times faster than ATLAS with the above CPU for large multiplications.

- *K40*: a single Nvidia K40 Tesla GPU with 2880 threads and 12GB memory. The CUBLAS library on this GPU is typically 2 to 3 times faster than on the C2075 GPU and so is often 30 to 45 times faster than ATLAS on the CPU for very large multiplications (the speedups depend on the shape of the specific matrices).

Table 1 lists timings for solving $(r+3, 9, r)$-MinRank systems over $K$, for $r = 4, 5, 6, 7, 8$ (all times are in seconds). The *Degree* column gives the degree of the ideal, while the *Density* column gives the density of the representation matrix $A$ used (corresponding to the last variable). The values of $r = 4, 5, 6$ give a simple comparison with the simple (non-block) Wiedemann algorithm, as implemented by Faugère and Mou. The *F-M GB* and *F-M Sol* columns give the times of Faugère and Mou in [7] to compute the *grevlex* GB and to solve the system by 'Sparse FGLM' with the simple Wiedemann algorithm, respectively. (Their timings were for a 2.5GHz Intel E5420 CPU, so these are scaled here by a ratio of 2.5/3.1 to give a reasonable comparison.) The *Magma GB* and *Magma Sol* columns give the times for our Magma implementation run on the CPU with the K40 GPU to compute the *grevlex* GB and to find one solution of the system by Block Wiedemann with a final evaluation GB, respectively. We note also that the blocking parameter $m$ used for the Block Wiedemann algorithm is $40, 100, 200, 400, 800$ and the memory usage is 0.4, 1.3, 12.1, 39.2, 191.1 GB for $r = 4, 5, 6, 7, 8$ respectively.

The times for $r = 4$ are practically the same as for Faugère and Mou, but as the degree grows, the Block Wiedemann with GPU algorithm clearly becomes much more effective. The last line of the table corresponds to solving the Courtois Challenge C.

**Table 1: MinRank $(r+3,9,r)$ Timings**

| r | Degree | Density | F-M GB | F-M Sol | Magma GB | Magma Sol |
|---|---|---|---|---|---|---|
| 4 | 4116 | 23.0 | 14 | 9 | 14 | 9 |
| 5 | 14112 | 19.0 | 240 | 272 | 55 | 73 |
| 6 | 41580 | 16.9 | 4041 | 5715 | 388 | 580 |
| 7 | 108900 | 14.8 | | | 3129 | 4781 |
| 8 | 259545 | 13.4 | | | 19745 | 34575 |

**Table 2: Challenge C Magma Timings**

| Stage | CPU | CPU + C2075 | CPU + K40 |
|---|---|---|---|
| GB (Dense $F_4$) | 302442 | 24358 | 19745 |
| BW Data | 406207 | 51550 | 19674 |
| BW LLL | 10928 | 9490 | 9295 |
| BW Det | 4659 | 4644 | 4627 |
| BW total | 421794 | 65693 | 33596 |
| Eval GB | 3743 | 1275 | 979 |
| Total time | 727979 (202.2h) | 91326 (25.4h) | 54320 (15.1h) |

We now describe in greater detail how a random instance of Challenge C is solved by our Magma implementation. Table 2 shows the timings for solving this system, with details on each stage; all times are in seconds, except for the total time in hours in the last line. Three runs were performed with respective times in the last 3 columns: (1) CPU only, (2) CPU with the C2075 GPU and (3) CPU with the K40 GPU (see above for details on the hardware). We were first able to solve a random instance with our Magma implementation on 23 August 2014, using the above CPU with the C2075 GPU in about 25 hours computation. Since then, we have also been able to use a K40 GPU, so that is why timings for both types of GPUs are provided.

A random (11,9,8)-MinRank problem is first constructed (known to have a solution) and the minors of the symbolic MinRank matrix are computed in about 1280 seconds on all versions (the GPUs make no difference). This yields a zero-dimensional ideal $I$ of $R = K[x_1, \dots x_9]$ generated by 3025 polynomials, each with total degree 9.[1]

The *GB (Dense $F_4$)* row in Table 2 gives timings for computing the *grevlex* GB of $I$, which is done by the dense variant of the $F_4$ algorithm. For the computation with the C2075 GPU, roughly half the time is spent on very large matrix multiplications, so the total time with the K40 GPU is about 25% less, while the CPU-only time is very much greater since those multiplications can be 20 to 30 times slower on the CPU. The *grevlex* GB has 51275 polynomials with maximum degree 25 and up to 48620 terms each, while the whole GB computation takes 137GB memory (in all versions).

Let $A$ be the matrix representing the multiplication action of $x_9 + I$ in the quotient algebra $R/I$. Then $A$ is an $N \times N$ matrix with entries in $K$, where $N = 259545$, and $A$ has density 13.42%. Now $A$ has 221690 rows which are unit vectors and $b = 37855$ rows which are dense, so the special compact representation (see the previous section) is set up with a $b \times N$ dense matrix $B$, which takes 1166 seconds and 73.2GB to store (in all versions).

[1]I thank J.-C. Faugère for originally providing me with Magma code to set up this kind of MinRank system.

**Table 3: Challenge C BW Data Step**

| | CPU | CPU + C2075 | CPU + K40 |
|---|---|---|---|
| Product 1 | 605.4 | 74.6 | 27.2 |
| Product 2 | 13.2 | 2.1 | 0.9 |
| Total | 618.2 | 76.7 | 28.1 |

We take the Block Wiedemann blocking parameter $m$ to be 800 and $\epsilon = 6$. Thus the precision $P = \lceil \frac{2N}{m} \rceil + \epsilon = 655$. The data collection phase (computing the $s_i$) for the Block Wiedemann algorithm thus involves 655 steps; the time for these steps plus the compact representation setup mentioned above is labelled by *BW Data* in the table. Each step involves two products of dense matrices over $K$ as follows:

1. $[800 \times 37855]$ by $[37855 \times 259545]$ (matrix $B$) to compute $V_i = V_{i-1}A$ for $i > 0$ via the compact representation of $A$ (where $V_0 = V$, so $V_i = VA^i$);

2. $[800 \times 259545]$ by $[259545 \times 800]$ to compute $s_i = V_i W^{tr}$.

Table 3 shows the number of seconds taken for these two products for the different hardware configurations in the runs; the total time thus indicates how many seconds are taken in each step.

The times for the LLL and determinant computations are labelled by *BW LLL* and *BW Det*, respectively. The scalar matrices over $K$ involved in these computations are not very large, so the timings with the GPUs are not much different. The total time taken for the whole Block Wiedemann algorithm is labelled by *BW total*.

After a root of the minimal polynomial is computed (in one second), an evaluation GB is computed to compute the full solution (all the other coordinates); the time for this is labelled by *Eval GB*.

Finally, the line labelled by *Total time* gives the total time for the whole computation (*GB + BW Total + Eval GB*). The maximum total memory usage for the whole computation is 191.1GB (for all versions).

## 6. CONCLUSION

Some very large zero-dimensional multivariate polynomial systems, where the degree of the relevant ideal is of the order of 250000, can clearly now be solved effectively by the direct GB method coupled with the Block Wiedemann algorithm and a GPU. With more memory, even larger systems could be solved in reasonable time.

Finally, we note that all the above computations can also be performed over the finite field $K_2 = \mathbb{F}_{2^{16}}$ instead of $K_1 = \mathbb{F}_p$ ($p = 65521$) in roughly similar time, at least on the CPU alone. A matrix multiplication in Magma over $K_2$ on the CPU takes roughly the same time as a matrix multiplication over $K_1$ using ATLAS, so Challenge C could be solved over $K_2$ in about a week too, using the CPU alone. The author has also implemented CUDA code for dense matrix multiplication in characteristic 2 on Tesla GPUs; currently this is about 2 to 4 times faster for the above GPUs than for the CPU code, so this gives speedups on GPUs for solving Challenge C in characteristic 2.

## 7. REFERENCES

[1] W. Adams and P. Loustaunau. *An introduction to Gröbner bases*, volume 3 of *Graduate studies in mathematics*. American Mathematical Society, Providence, R.I., 1994.

[2] W. Bosma, J. Cannon, and C. Playoust. The Magma Algebra System I: The User Language. *J. Symbolic Comp.*, 24(3):235–265, 1997.

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

[4] N. Courtois. Efficient zero-knowledge authentication based on a linear algebra problem minrank. In C. Boyd, editor, *Advances in Cryptology – ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 402–421. Springer Berlin Heidelberg, 2001.

[5] J.-C. Faugère. A new efficient algorithm for computing Gröbner bases (F4). *Journal of Pure and Applied Algebra*, 139(1–3):61–88, June 1999.

[6] J.-C. Faugère and C. Mou. Fast Algorithm for Change of Ordering of Zero-dimensional Gröbner Bases with Sparse Multiplication Matrices. In *Proceedings of the 36th international symposium on Symbolic and algebraic computation*, ISSAC '11, pages 115–122, New York, NY, USA, 2011. ACM.

[7] J.-C. Faugère and C. Mou. Sparse FGLM algorithms. preprint, 2013. http://hal.inria.fr/hal-00807540.

[8] J.-C. Faugère, M. Safey El Din, and P.-J. Spaenlehauer. Computing Loci of Rank Defects of Linear Matrices using Grobner Bases and Applications to Cryptology. In *ISSAC '10: Proceedings of the 2010 international symposium on Symbolic and algebraic computation*, ISSAC '10, pages 257–264, New York, NY, USA, 2010. ACM.

[9] J.-C. Faugère, M. Safey El Din, and P.-J. Spaenlehauer. On the Complexity of the Generalized MinRank Problem. *Journal of Symbolic Computation*, 55(0):30–58, March 2013.

[10] A. Joux and V. Vitse. A variant of the F4 algorithm. Cryptology ePrint Archive, Report 2010/158, 2010. http://eprint.iacr.org/2010/158.

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

[12] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

[13] S. Paulus. Lattice basis reduction in function fields. In *Algorithmic number theory (Portland, OR, 1998)*, volume 1423 of *Lecture Notes in Comput. Sci.*, pages 567–575. Springer, Berlin, 1998.

[14] E. Thomé. Fast computation of linear generators for matrix sequences and application to the block Wiedemann algorithm. In *Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation*, pages 323–331 (electronic). ACM, New York, 2001.

[15] E. Thomé. Subquadratic computation of vector generating polynomials and improvement of the block Wiedemann algorithm. *J. Symbolic Comput.*, 33(5):757–775, 2002. Computer algebra (London, ON, 2001).

[16] G. Villard. A study of Coppersmith's block Wiedemann algorithm using matrix polynomials. Technical report, LMC-IMAG, REPORT # 975 IM, 1997.

[17] R. C. Whaley and A. Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005. `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`.

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