Matrix Inversion (or LUP-Factorization) via the Method of Four Russians, in $\Theta(n^3/\log n)$ Time

Gregory V. Bard

Abstract

We present an algorithm for reducing dense matrices over the field of two elements to Row-Echelon Form or Reduced-Row-Echelon Form, based on the "Method of Four Russians" for matrix multiplication. It is logarithmically faster than Gaussian Elimination, and in practice leads to significant performance increase for reasonable sizes. We further describe how to interface it with Strassen's Algorithm for large matrices. The algorithm was known anecdotally in certain cryptographic circles, but we present complexity analysis, implementation details, experimental timings, and we reduce the probability of an abort to practically zero. The algorithm is the heart of the "M4RI" library which is available as an open-source tool and which is part of SAGE, and is also suitable for LUP-factorizations, determining the rank of a matrix, solving linear systems of equations and inverting matrices.

The running time is $\Theta(mn\min(m,n)/\log\max(m,n))$ for all applications listed except the RREF, and for that it is $\Theta(mn\min(m,n)/\log m)$. The speed-up comes from two tricks primarily. The first is using the Gray Code of length k to enumerate all the vectors in a k-dimensional subspace spanned by a set of k rows, where $k \approx \log_2 n$ is the parameter of the algorithm. The second is that a linear combination of rows need not be computed several times if it is used several times. Finally, we show experimentally that it is not necessary to be exact in choosing the best k.

AMS Classification: 15A09, 15A23, 15A33

1 Introduction

In this article, we present a parameterized algorithm for the reduction of a $\mathbb{GF}(2)$ matrix into Row-Echelon Form (REF), or Reduced-Row-Echelon Form (RREF). The primary intended applications are the LUP-factorization of a $\mathbb{GF}(2)$ matrix, solving a system of linear equations over $\mathbb{GF}(2)$, inverting a $\mathbb{GF}(2)$ matrix, or determining its rank. We also suggest a scheme for the selection of the algorithm's parameter, which yields $\Theta(n^3/\log n)$ time for square matrices, as compared to $\Theta(n^3)$ for Gaussian Elimination, and provide experiments to determine the optimal setting on a typical computer in practice. We further show that the consequences of being slightly off in the selection of the parameter is a penalty of around 1% in the running time.

Because the running time is $\Theta(n^3/\log n)$, the reader may be surprised to see so much time spent on this algorithm, since Strassen's Algorithm [23] runs in time $\Theta(n^{2.807\cdots})$. Our algorithm is faster than Strassen's Algorithm for medium-sized matrices. In particular, our implementation of our algorithm is faster than the implementation of Strassen's Algorithm by Magma for matrices of size up to 64,000 rows/columns or 16,000 rows/columns depending on machine architecture, and so is useful for medium-sized problems (See Section 6.1). Moreover, the algorithm can be used for the "small chunks" left at the end of the recursion in Strassen's Algorithm, resulting in a constant factor speed-up. Both of these issues will be addressed in Appendix E.

The algorithm is an agglomeration of the Method of Four Russians Matrix Multiplication Algorithm [6] [4, Ch. 6], and Gaussian Elimination. Due to this heritage, we call the algorithm M4RI (Method of Four Russians for Inversion) to acknowledge the work of [6] and to distinguish it from M4RM (Method of Four Russians for Multiplication), the matrix-multiplication algorithm which appears in [4, Ch. 6].

The M4RI (pronounced "Mary") algorithm has been known anecdotally for some time [13], and yet never published, so far as we have been able to detect. Here we include detailed complexity analysis; an optimization of the algorithm's parameter; a reduction of the probability of an early abort to zero in practice; performance analysis of the implementation of the algorithm in Sage [3]; and notes on how to integrate the algorithm with Strassen-like algorithms.

1.1 Probability of an Abort

The concept of an abort in $\mathbb{GF}(2)$ linear algebra is not new. Early aborts can occur in both M4RI and Strassen's Matrix Inversion Formula [23] (given later as Equation 1), when a submatrix is not of full-rank yet is expected to be. Unfortunately, in $\mathbb{GF}(2)$ this happens rather frequently. To be brief, it can be proven [7, Ch. A.3.3] that a real-valued matrix filled with independent and identically distributed random variables, with continuous probability distribution function, will be singular with probability zero, which is why the problem "never happens" over $\mathbb{R}$. In contrast, a $\mathbb{GF}(2)$ matrix filled by independent fair coins will be singular with probability $\approx 71.1\ldots\%$ (See [21], [25, Ch. 16], [9] or [7, Ch. A.3.3] for a proof). The proof parallels that of Lemma 1. This problem for Strassen's Matrix Inversion Formula was surmounted by [11] and later improved by [18].

Using the methods in this article, the probability of an early abort for our algorithm can be made $1 - 1/(n \log_2 n)$ for an $n \times n$ matrix. In practice, aborts did not ever occur for M4RI except when extremely unwise choices for the algorithm's parameter were tested, in order to complete tables. See Table 2 for details. Furthermore, in Appendix A we give an alternative strategy for Stage One which never aborts, but which in worse case gives $\Theta(n^3)$ instead of $\Theta(n^3 \log n)$ running time. However, for all but the most pathological inputs, it will have running time $\Theta(n^3/\log n)$ and it cannot abort. See Appendix A.1 for details.

1.2 Density Assumptions

This algorithm was designed for dense random matrices, which in $\mathbb{GF}(2)$ means that the matrix was filled by independent fair coins. If the matrix is sparse then algorithms such as the Block Wiedemann Algorithm [12] or Lanczos's Algorithm [20] will likely be faster. This "fair coin assumption" is important only in determining the probability of an abort, but our experiments show aborts do not occur in practice even with sparse matrices. In any case, after a few iterations of normal Gaussian Elimination (but not Wiedemann or Lanczos), a sparse matrix becomes effectively dense, and then one can proceed with what is given here. In algebraic cryptanalysis, sometimes the system of equations being solved will be dense (e.g. QUAD [8]) or sparse, for low gate-count ciphers like SEA [22]. In many ciphers, such as HB [17], we have a strong reason to believe that the coefficients will behave like random coins, see [26].

If the distribution is not independent, the author conjectures that the probability of an abort will remain very low except for possibly particularly strange distributions. For the performance of the system on a matrix very far from having the fair coin distribution, see `http://m4ri.sagemath.org/` under "performance."

1.3 Software Availability

The author's implementation of the algorithm has been released under the GPL (GNU General Public License) and is available for download at `http://m4ri.sagemath.org/` . However, since the library was first launched, several programmers have contributed to the project, and the current library is highly optimized. Furthermore, it has been enlarged to cover virtually every aspect of dense linear algebra. Through the course of several clever optimizations, (similar to those done for matrix multiplication in [5]), the speed is currently 20 times faster than the author's original implementation, and only a factor of 2 or 3 can be accounted for by the progress in microprocessors since when most of the experiments in this paper were originally performed. Naturally, we provide some newer, up-to-date timings as well (see Table 5). The library is used for any dense matrix over $\mathbb{GF}(2)$ in SAGE [3], but also in the $\mathbb{GF}(2)$ Gröbner Bases tool PolyBori [10], and is in the process of becoming an optional package in the Debian distribution of Linux.

2 Gray Codes

The primary mechanism for the $\log_2 n$ speed-up is the Gray Code [16]. A Gray Code of n bits is a sequence of all 2^n-bit strings of length n, such that the following properties hold: each string appears exactly once; each string differs in exactly one spot (one bit) from the one before it (and therefore, likewise, the one after it); the initial string is the all-zero string. This sequencing enables one to enumerate all the vectors in an n-dimensional subspace of a $\mathbb{GF}(2)$ vector space in time equal to $2^n - 1$ vector additions, not the $\Theta(n2^n)$ vector additions that would otherwise be expected, as we will now show.

Consider an n-dimensional subspace S of $\mathbb{GF}(2)^m$ (obviously $m \geq n$), with vectors $b_1, b_2, \ldots, b_n$ forming a basis. There are 2^n vectors in S, and each can be written uniquely as a linear combination of the b_i vectors. The number of non-zero coefficients in that unique linear combination is the weight of that vector with respect to that basis. There are $\binom{n}{w}$ vectors of weight exactly w in S, for $0 \leq w \leq n$. Thus the average weight is $n/2$ and the number of vector additions required to compute it naïvely is $n/2 - 1$, or $2^n(n/2 - 1)m$ field operations in total.

Instead, starting with the m-dimensional all-zero vector and the first codeword, we will iterate through the Gray Code. When iterating from the ith codeword to the $i+1$th, only one bit changes, call it j. We will then add b_j to the current codeword. Thus, with one vector addition per codeword, we enumerate the entire subspace. This means only $(2^n - 1)m$ field operations are needed. This is essentially $n/2$ times faster than the naïve method.

Note: In the SAGE implementation of this algorithm, the Gray Codes are computed only once and stored permanently, for all sizes 1 to 16, and this computation takes much less than 1 second. (For the method of generating the codewords, see Knuth's *Art of Computer Programming* [19, Vol 4, Fascicle 2a], or Appendix D.1).

3 The Algorithm

3.1 Overview

At each iteration, we will process k columns, where k is the parameter of the algorithm. There will be $\lceil \min(m, n-1)/k \rceil$ of iterations, each consisting of the following three steps.

First, we will do a Gaussian Elimination on only $3k$ rows of the matrix. Then we will anticipate that the first k columns of the elimination did not fail to find a pivot in those $3k$ rows. If this assumption is false, we abort, but if it is true, then the first k rows form a basis for a k-dimensional subspace. The abort is an artifact of the complexity analysis, but essentially never occurs as shown below; furthermore, we present, in Appendix A an alternative for Gaussian Elimination which does not abort. Second, we will use the Gray Code to rapidly enumerate all vectors in that subspace, as explained above. Third, we will use these enumerated vectors as patterns as explained below, and add them to the remaining rows to create (for RREF) k columns of all zeros, except on the main diagonal. In the case for REF, we do not touch rows which are above the main diagonal in the left-most column of the columns being processed.

3.2 Pseudocode

Below, i represents the number of rows and columns processed so far, and k is the algorithm's parameter. The matrix A is to be reduced to REF, and has dimensions $m \times n$.

1. $i \leftarrow 0$
2. While $(i \leq m)$ AND $(i \leq n-1)$ do
   - (Stage 1) Perform a Gaussian Elimination on rows $i+1 \ldots i+3k$, to render those $3k$ rows in RREF.
   - With high probability, these $3k$ rows had pivots for the k active columns, and thus the first k rows among the $3k$ now form a basis for a k-dimensional subspace W of $\mathbb{GF}(2)^n$. If not, abort.
   - (Stage 2) Use a Gray Code of length k to enumerate the $2^k - 1$ non-zero vectors in the subspace W spanned by the first k rows.
   - (Stage 3) For each row $r \in \{i+3k+1, i+3k+2, \ldots, m\}$ do
     (a) Consider $A_{r,i+1}, A_{r,i+2}, \ldots, A_{r,i+k}$ as a k-bit integer z.
     (b) Take the vector, from W, that is "associated with z" (defined below), and add it to row r.
   - $i \leftarrow i + k$
3. If $i \neq \min(m, n-1)$ then finish with Gaussian Elimination. (Clean-up Stage).

Note to the Referee:*Note to the referee: The author could not choose between two formats for the pseudocode. Therefore this algorithm is formatted one way, and the one on Page 8, is formatted in the other. Tell me which you think is better, and I will format both that way, or feel free to suggest a hybrid of the formats, or a new format entirely.*

During Stage 1 (the Gaussian Elimination) if a column has no pivot, among the $3k$ rows, then this will result in all zeros in at least one of the k active columns. In terms of the theory of the algorithm, we abort. We will show that this happens very rarely. However, in practice, one does not abort, but one simply ignores the column and continues. The abort is only an artifact of modeling the complexity in theory. Furthermore, in Appendix A, we show an alternative Stage 1 which, in practice, will result in the algorithm never aborting if the input is non-singular.

Instead, if the Stage 1 Gaussian Elimination did find pivots for the first k columns, then there is a $k \times k$ identity matrix located in $A_{i+1,i+1}, \ldots, A_{i+k,i+k}$. This, in turn, guarantees that those k rows are linearly independent.

The condition required between Stages 1 and 2 can be detected by looking for a zero on the main diagonal. A zero will only appear there if the $3k$ rows used in Stage 1 failed to provide a pivot for the first k columns of the elimination. This will occur if and only if the intersection of the $3k$ rows and the k active columns, taken as a $3k \times k$ matrix, is not full-rank. We address the probability of that event in Lemma 1.

During Stage 3 the term "associated with z" is used. Each vector in the subspace is generated by a codeword in the Gray Code. The codeword can be read as an integer, and that subspace vector is associated with that integer. Always it will be the case that the subspace vector associated with any particular integer is enumerated at the same spot in the list of enumerated vectors. Therefore, this entire mapping can be performed by a lookup in a pre-computed array. Thus the code using

$$000, 001, 011, 010, 110, 111, 101, 100$$

would result in

$$(0 \to w_0 = 0), (1 \to w_1 = b_3), (2 \to w_3 = b_2), (3 \to w_2 = b_2 + b_3),$$
$$(4 \to w_7 = b_1), (5 \to w_6 = b_1 + b_3), (6 \to w_4 = b_1 + b_2), (7 \to w_5 = b_1 + b_2 + b_3)$$

Adding the associated vector forces all the entries in the k columns currently being worked upon to become zero, since we operate in $\mathbb{GF}(2)$.

Note: All Gaussian Elimination is performed with row swaps permitted, but not column swaps (sometimes called "partial pivoting"), as is standard.

4 Complexity Calculation

Here we calculate the cost of the jth iteration of the algorithm, given an $m \times n$ matrix and parameter k.

A Gaussian Elimination on an $a \times b$ rectangular matrix takes $\Theta(ab \min(a, b))$ field operations [24]. Thus, in Stage 1, the $3k \times n$ Gaussian Elimination requires $\Theta(k^2 n)$ field operations, provided $3k < n$. Since we will later choose $k \approx \log_2 n$, this inequality will be true for any reasonable n.

In Stage 2, using the Gray Code subspace enumeration, requires $2^k - 1$ vector-additions. Each vector is of length n, so this is $(2^k - 1)n$ field operations, or $\Theta(2^k n)$.

Finally, in Stage 3, $\max(m - (j-1)k - 3k, 0)$ rows will be processed. Only a few memory reads are required to figure out which Gray Code Table entry is associated with the k bits in question, and so the vector-addition is the important step. The vector is of length n again, and so this is $n \max(m - (j-1)k - 3k, 0)$ field operations, or $\Theta(mn)$.

Of course, in all three Stages, we can ignore all columns to the left of those currently being worked upon. A more careful complexity analysis taking this into account, and retaining the coefficients rather than using big-Θ notation can be found in [7, Ch 5.6].

Thus, all three stages of the iteration together, have cost $\Theta(n(2^k + m))$. Allowing j to run from 1 to $\lceil \min(m/k, (n-1)/k) \rceil$, we get a total of $\Theta(n(2^k + m) \min(m, n)/k)$ operations.

The Clean-Up Stage will be a Gaussian Elimination that is very narrow, or very short, or both, if k does not divide $\min(m, n-1)$. The value $i - \min(m, n-1) = c$ is the number of rows/columns that need to be processed at the end, and $0 \leq c < k$. Thus $O(k^2 \max(m, n-1))$ field operations will be used for clean-up, which is rather negligible.

4.1 Choosing the Parameter

Furthermore, if $k = \log_2 m$ this comes to $\Theta(nm \min(m, n)/\log m)$, or a $\log_2 m$ speed-up over Gaussian Elimination. In practice, k will be determined experimentally. These experiments are easy to perform on any particular machine. However, experiment #3 shows that it is not too important to get k exactly right (See Section 6).

If $m \ll n$ then working with A^T would be better if it is possible. The running time is $\Theta(mn \min(m, n)/\log \max(m, n))$ in that case, see Appendix C.3 for details.

4.2 Probability of Abort

The only possibility of an abort is if the first stage fails to find a pivot for one or more of the first k columns.

This abort will occur if and only if the $3k \times k$ submatrix, defined by the intersection of the k active columns and the $3k$ rows operated upon in Stage 1, is singular. Using Lemma 1, we discover that a $3k \times k$ matrix of fair coins will be non-singular with probability $1 - 2^{-2k}$. With $k = \log_2 m$ this comes to $\approx 1 - m^{-2}$. Thus even if $m = 1000$, the probability is quite low that one has a singular matrix. Of course, if $2k$ were used in place of $3k$ then the probability comes to $\approx 1 - m$ which, we will show below, causes problems.

In theory, it makes sense to use $3k$ or even $4k$ because this Stage 1 is invisible in the final complexity analysis. In practice, since the rank of the $3k$ rows is likely to be between $3k$ and $3k - 8$ (See Table 5.3 in [7, Ch 5.4]), then the next two iterations will have very fast Stage 1s, since most of the work of that stage is already performed.

Note, the variant method of "Stage 1" described in Appendix A and called "Sideways Gaussian Elimination" uses exactly as many rows as required.

Of course if $m \ll n$ then working with A^T rather than A would improve the lowering of the probability of an abort, resulting in $1 - \max(m, n)^{-2}$ for any particular iteration. See Appendix C.3 on the "transpose" operation.

Of course, there are $\lceil \min(m, n-1)/k \rceil$ iterations, so the net probability of an abort can be calculated using a probabilistic argument. For simplicity, we will assume $m = n$. In that case

$$(1 - n^{-2})^{(n/k)} \approx 1 - (n/k)(n^{-2}) = 1 - 1/nk = 1 - 1/(n \log_2 n)$$

and one can see why $2k$ is unacceptable compared to $3k$, because then the total probability of an abort would be $1 - 1/\log_2 n$, which is far too high.

5 Adaptations

5.1 Inverting Matrices

In Stage 3, instead of only doing rows r such that $i + 3k + 1 \leq r \leq m$ one can also include the rows $1 \leq r \leq i$ as well. This will result in a matrix in RREF rather than REF. Since the RREF of $[A|I]$, the matrix A adjoined with the identity matrix of the correct size, is $[I|A^{-1}]$, provided A is non-singular and square, this can be used to find A^{-1}.

5.2 LUP-Factorization of Matrices

To perform the LUP-factorization one simply performs the reduction to REF. There must be a matrix, initially the identity matrix of size $m \times m$, that will become L. The REF of A will be U. The permutation matrix P is also initially an identity matrix, but of size $n \times n$.

Whenever rows a and b are swapped in Stage 1, or in the clean-up stage, swap rows a and b in P. No other action is required to construct P.

Suppose the "active columns" are $i + 1, i + 2, \ldots, i + k$ and in Stage 3, while working on row z, we observe $A_{z,i+1}, A_{z,i+2}, \ldots, A_{z,i+k}$ in the active columns. In addition to adding the appropriate Table-Row to row z, we will do the following to L. For each $j \in \{i + 1, i + 2, \ldots, i + k\}$, we set $L_{jz} = -A_{zj}$. (The negation is redundant for $\mathbb{GF}(2)$ but we include it for use over other fields, see Appendix D).

Once the LUP-factorization is produced, then Strassen's Matrix Inversion Formula [23]

$$A = \begin{bmatrix} B & C \\ D & E \end{bmatrix} \Rightarrow A^{-1} = \begin{bmatrix} B^{-1} + B^{-1}CS^{-1}DB^{-1} & -B^{-1}CS^{-1} \\ -S^{-1}DB^{-1} & S^{-1} \end{bmatrix} \quad (1)$$

where $S = D^{-1} - E^{-1}CB^{-1}$, the Schurr Complement of A with respect to E. can be used to calculate the inverse as well, because L, U, and P will all be non-singular, along with all submatrices inverted by the formula being applied recursively. This follows from the fact that the main diagonal will be all ones for both L and U, and so they are both invertible, as are all permutation matrices. There are also faster methods for converting an LUP-factorization into a matrix inverse [2].

6 Experimental Results

In the first experiment, with results listed in Table 1, all choices of k from 1 to 16 were attempted. The optimal choice was recorded, with running times within 1% recorded as a tie. The running times are given, including the

running time of a highly optimized Gaussian Elimination written using the same low-level functions as the M4RI's library uses, including the use of 64-bit operands as described in Appendix C. One can see that the speed-up is appreciable at even these low sizes, but also that the ratio is given after dividing by $\log_2 n$, and is still increasing. This means that the speed-up, for these sizes, is slightly better than $\log_2 n$.

The next experiment was to select an optimal k for various larger matrices. Unfortunately, compilation under the highest setting was extremely slow at that time (several hours), because of the large size of the pre-computed Gray Code tables. We were therefore compelled to engage in the unusual practice of using settings lower than the best possible. This is no longer a problem, however. The optimization setting of "no optimization" in `gcc` was selected to allow for rapid recompilation between changes of k but also to exaggerate the differences between k values. The results are given in Table 2. But then, one can measure the effects of choosing a non-optimal k. This is given in Table 3, where for example +2 in the left column means selecting a k that is two higher than the optimal choice. One can see that the penalty of being off by 1 is very small, in either direction. This means it is not necessary to be too precise in developing a formula for the optimal k.

Finally, the Experiment 4 is just the same as Experiment 3, but with all the compiler optimization settings turned on, and working with larger matrices. The data is given in Table 4. This also confirms that the phenomenon noted in Experiment 3 was not an artifact of turning the compiler optimizations off. Not all possible k were tried, but only those near to the optimal. The ratio of the running time of M4RI to Gaussian Elimination is given, and again that ratio appears to move as $\log_2 n$ times some constant.

Based on this, it appears that the optimal is $k = (\log_2 n) - 2.5$, for the computer used in the experiments, which was a 1 GHz PC, with 1 GB of RAM, running UNIX via Cygwin, through Microsoft Windows.

6.1 Recent Experiments: Comparison to Magma

These experiments represent all of the optimizations added to this algorithm by the several programmers listed in the Acknowledgements. Most of these are cache-related. As you can see, the algorithm is quite fast, now inverting a $32,000 \times 32,000$ matrix (with roughly 1 billion entries) with running time under 2 minutes.

Here Computer I signifies, a Macbook Pro 2nd Generation laptop, running 64-bit Debian/GNU Linux, with a 2.33Ghz Intel Core2Duo processor; Computer II signifies a 2.6Ghz Opteron (VMWare Virtualised) running 64-bit Debian/GNU Linux; Computer III signifies a 1.6Ghz Itanium Processor.

The cross-over with Magmaappears to be slightly below dimension $64,000$ for the Macbook laptop, and somewhere between $16,384$ and $20,000$ for the Opteron. But for the Itanium, the cross-over appears to be slightly above dimension $64,000$. This makes sense, as it is known that Magmahas custom assembly-language code for the Opteron architecture. Thus the performance is slightly better there, whereas in our library there are no custom assembly-language changes.

6.2 To Find k On a Particular Machine

For particular fixed sizes, try all k within 4 of $k = (\log_2 n) - 2.5$ and then choose the best. If one is concerned with matrices of any size within a broad interval, a least-squares fit of the form $(a \log_2 n) + b$ should be trivial to carry out, and sufficiently accurate. For example, on `sage.math.washington.edu` it appears that $0.75 \log_2 n$ was optimal.

7 Acknowledgements

The author would like to thank the following: Dr. Nicolas Courtois, for explaining the algorithm generally, at Eurocrypt 2005 in Aarhus; Prof. Lawrence Washington, for reading this work and providing useful suggestions; Prof. Bill Stein, for giving me access to `sage.math.washington.edu` and including my work in Sage, as well as encouragement; Martin Albrecht, for endless support and hard labour relating to the integration of this work into Sage, and testing the performance; for Dr. Clément Pernet and also Martin Albrecht for providing the commentary which lead to writing Appendix A.1; Prof. Robert Lewis, for providing commentary on drafts of this article; Prof. Antoine Joux, for explaining several additional details of Gray Codes, during my talk at Versailles; to Prof. Richard Brent, Dr. Paul Zimmermann, and Dr. Paul Leopardi, for reading this draft.

Since the original launch of the M4RI library, several programmers have contributed. They are Tim Abbot, Martin Albrecht, Michael Brickenstein, Dr. Alexander Dreyer, Maître de Conférences Jean-Guillaume Dumas, Dr. William Hart, Prof. David Harvey, and Dr. Clément Pernet.

References

[1] Magma. Software Package. Available at `http://magma.maths.usyd.edu.au/magma/`.

[2] Personal communications with Clement Pernet, draft in progress.

[3] Sage. Software Package. Available at `http://www.sagemath.org/`.

[4] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, second edition, 1974.

[5] Martin Albrecht, Gregory Bard, and William Hart. Efficient multiplication of dense matrices over gf(2). Submitted to a journal., 2008.

[6] V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev. On economical construction of the transitive closure of a directed graph. *Dokl. Akad. Nauk. SSSR*, 194(11), 1970. (in Russian), English Translation in Soviet Math Dokl.

[7] Gregory Bard. *Algorithms for the solution of linear and polynomial systems of equations over finite fields, with applications to cryptanalysis.* PhD thesis, Department of Applied Mathematics and Scientific Computation, University of Maryland at College Park, August 2007.

[8] Côme Berbain, Henri Gilbert, and Jacques Patarin. Quad: A practical stream cipher with provable security. In *Advances in Cryptology—Proc. of EUROCRYPT*, Lecture Notes in Computer Science. Springer-Verlag, 2005.

[9] Richard Brent and B. D. McKay. Determinants and ranks of random matrices over z_m. *Discrete Mathematics*, 66:35–49, 1987.

[10] M. Brickenstein and A. Dreyer. Polybori: A framework for grbner basis computations with boolean polynomials. In *Effective Methods in Algebraic Geometry*, 2007.

[11] J. Bunch and J. Hopcroft. Triangular factorization and inversion by fast matrix multiplication. *Math Comp.*, 28(125), 1974.

[12] Don Coppersmith. Solving homogeneous linear equations over gf(2) via block Wiedemann algorithm. *Mathematics of Computation*, 62(205):333–350, 1994.

[13] Nicolas Courtois. Personal communications with Nicolas Courtois.

[14] Dan Bernstein. Personal communications with Dan Bernstein.

[15] J. Gill. Computational complexity of probabilistic turing machines. *SIAM Journal on Computing*, 6(4):675–695, 1977.

[16] F. Gray. Pulse code communication, March 1953. USA Patent 2,632,058.

[17] Nicholas J. Hopper and Manuel Blum. Secure human identification protocols. In *ASIACRYPT*, pages 52–66, 2001.

[18] O. H. Ibara, S. Moran, and Hui R. A generalization of the fast LUP matrix decomposition algorithm and applications. *Journal of Algorithms*, 1(3):45–56, 1982.

[19] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations*. Addison-Wesley Professional, 2005.

[20] P. Montgomery. A block Lanczos algorithm for finding dependencies over gf(2). In *Advances in Cryptology—Proc. of EUROCRYPT*, Lecture Notes in Computer Science. Springer-Verlag, 1995.

[21] H. J. Ryser. Combinatorial properties of matrices of zeros and ones. *Canadian Journal of Mathematics*, 9:371–377, 1957.

[22] Francois-Xavier Standaert, Gilles Piret, Neil Gershenfeld, and Jean-Jacques Quisquater. SEA: a scalable encryption algorithm for small embedded applications. In *Smart Card Research and Advanced Application IFIP Conference (CARDIS'06)*, volume 3928, pages 222–236. Lecture Notes in Computer Science, Springer-Verlag, 2006.

[23] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(3), 1969.

[24] Lloyd Trefethen and David Bau, III. *Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997.

[25] J. H. van Lint and R. M. Wilson. *A Course in Combinatorics*. Cambridge University Press, second edition, 2001.

[26] Kenneth Koon-Ho Wong, Gregory V. Bard, and Robert H. Lewis. Partitioning multivariate polynomial equations via vertex cuts for algebraic cryptanalysis and other applications. Submitted to a Journal., 2008.

A Modified Gaussian Elimination

The method below performs a type of Gaussian Elimination, when one wants k rows of a large invertible matrix to be in RREF, with no rows of zeros, but yet one wants to work on as few rows as possible. This method only will process k rows, and will only abort if there is no possible pivot in some column in the entire matrix (i.e. the situation is hopeless). Therefore, in practice, we always use what is in this appendix.

Upon termination, there will be k rows at the top in RREF, and the remaining rows will follow underneath, untouched but possibly reordered. For use in M4RI, the addresses of the rows and columns must be incremented by an offset equal to k times the number of fully completed iterations.

This is meant to replace Stage 1 in the algorithm in practice, to make the probability of an abort to be zero unless the matrix was actually singular. However, the theoretical complexity analysis is ruined by this change, as Stage 1 is not of negligible duration in the worst case. If the input matrix is singular, the correct RREF or REF is computed nonetheless.

The matrix A has m rows and n columns.

```
function SidewaysGaussian(matrix A, integer k) do begin
    For 1 ≤ j ≤ k do begin
        //j is the ‘‘active’’ column
        found ← 0
        i ← j
        While (i ≤ m) AND (found = 0) do begin
            //i is the ‘‘candidate’’ row
            For 1 ≤ z ≤ j − 1 do ‡
                //clear the ones in the left of the new row
                If A_{i,z} ≠ 0 then
                    Add row z to row i and store in row i
            If A_{i,j} ≠ 0 then begin†
                found ← 1
                Swap row i and row j
                For 1 ≤ z ≤ j − 1 do
                    //clear the new column in all the old rows.
                    If A_{z,j} ≠ 0 then
                        Add row j to row z and store in row z.
            end
        end
        If found= 0 then abort in theory, or go to the next column in practice.
    end
    return A
end
```

Note to the Referee: *The author could not choose between two formats for the pseudocode. Therefore this algorithm is formatted one way, and the one on Page 3, is formatted in the other. Tell me which you think is better, and I will format both that way, or feel free to suggest a hybrid of the formats, or a new format entirely.*

The method here is more efficient than using $3k$ rows as explained earlier, however, the big-Oh running-time analysis is ruined because Stage 1 can become non-negligible in the worse case. This will be explored in the following appendix. However, in practice, this method is faster, because only precisely as many rows as are needed are processed, not $3k$ or k plus a small integer. Moreover, the true advantage of this option is that it will never abort unless there was no pivot column at all.

A.1 Complexity Model

In short, the complexity argument we are making is very similar to `ZPP`. In the complexity class `ZPP` a randomized algorithm is never permitted to be wrong, but has *expected* running time that is polynomial time [15]. The worse-case running time might be infinite. These are sometimes called "Polynomial-Time Las Vegas Algorithms."

The classical example of this a decision problem in `RP` $\cap$ `Co-RP`. Because it is in `RP` there is an algorithm that is never wrong if the answer is `NO`, but is only correct with probability $\geq 1/2$ if the answer is `YES`, (e.g. using Miller-Rabin to test primality). Because it is in `Co-RP` there is an algorithm that is never wrong if the answer is `YES`, but is only correct with probability $\geq 1/2$ if the answer is `NO`. This gives rise to the following program, which is never wrong. One can call the first algorithm. If the answer is `NO`, then this is certainly correct. If the answer is `YES`, then call the second algorithm. If this too is `YES`, then that too is certainly correct. However, if the second output is `NO`, then repeat from the beginning. The expected running-time is clearly finite, but for any running time, there is a positive probability that the program will run that length of time.

Here, we propose an algorithm which has two strategies for Stage One. If one uses "Sideways Gaussian Elimination", given in Appendix A, then the running time is expected $\Theta(n/\log n)$ and will be that with very high probability. However, for certain very strange inputs, it could be $\Theta(n^3)$. If one uses the traditional Stage One, which is an ordinary Gaussian Elimination on a particular sub-matrix, then the running time is always $\Theta(n/\log n)$, but it might abort. The probability of aborting was a problem in practice, but now with what is contained here, *it will never occur in practice.* In theory, it is nice, because the running time is never $\Theta(n^3)$. Furthermore, if an abort does occur in practice, a random permutation of the matrix can be taken, and the algorithm restarted from the beginning.

Thus one could define `ZNLNPP`, a complexity class where the expected running time is $\Theta(n \log n)$ but it might be longer, and `NLNP`, where the running time is always $\Theta(n \log n)$, but with a probability of an abort.

A.2 Using 3k Rows versus Sideways Gaussian Elimination

The algorithm given in Appendix A, "Sideways Gaussian Elimination" has the property that it will not fail to find a pivot unless the original matrix is singular— which would imply that no pivot exists and therefore the failure to find it is inevitable. Even then one can simply ignore the column and keep going to find the RREF.

One would think that since the alternative method—Sideways Gaussian Elimination, is better than ordinary Gaussian Elimination for $3k$ rows in practice, with very rare exceptions, that in theory this would be the case too. Furthermore, this is tempting because the fair-coin assumption could be dropped, and there would be no worries of aborts. Both Martin Albrecht and Clément Pernet have independently conjectured this to the author.

Consider the algorithm given for "Sideways Gaussian Elimination." But note the complexity of the innermost loop involves several row additions—in fact, a number of row additions equal to the weight of column i in rows $1, 2, \ldots, j-1$ and the weight of the new row j in columns $1, 2, \ldots, j-1$. This is expected to be $j-1$, and is at worse $2j-2$. Thus $\Theta(jn)$ operations are anticipated.

The outer loop will run a number of times equal to the number of linearly independent rows that one wishes to generate, namely k times. The inner loop causes much trouble.

The purpose of the inner loop is to find a pivot element for the active column, j. But the test performed at step † cannot be performed until the loop at ‡ is executed. In reality, we expect the inner loop to run very few times. But in worse case, it might run several times.

At best, each time we begin the inner loop, we discover that the first available row meets the requisite requirement $A_{ij} \neq 0$. But it is also possible that only the last row found will be the pivot row. For this example, let $j-1, j, j+1, j+2$ be the last three columns of a set of k columns being worked on in one Method of Four Russians elimination's Stage 1.

One would argue that if one must search all the rows in stage j, that the columns $1, 2, \ldots, j-1$ would all be now zeroed. Furthermore, in all but the pivot row, column j would be zero as well (otherwise an earlier column would

be selected as the pivot). Thus in terms of field operations, column $j+1$'s search would be very fast.

Thus, having the pivot in the first row or in the last row can both be a good outcome. Now consider that in column j it is the last row possible which is the pivot, in $j+1$ the first row possible, and in $j-2$ the last row possible. For both column j and column $j-2$, row additions must be performed in the loop marked with a double dagger.

Now let us generalize. Imagine for the odd numbered iterations, the last row chosen will be the pivot row, and for even numbered iterations, the first row chosen. In processing column x for Stage 1, if $x > 1$ is odd, then $m - x + 1$ rows will have to be checked. This will require a row addition in each case. Thus $(m-x+1)(n-x)$ field operations. Since this will occur for $x \in \{1, 3, 5, \ldots, n\}$ then

$$\sum_{x=1}^{x=\min(n,m)} (m-x+1)(n-x)$$

field operations are required, which becomes when $m \approx n$ equal to $\Theta(m^3)$. In other words, Stage 1 alone becomes longer than the entire algorithm.

Summary When a matrix is generated by fair coins, there is a positive probability of getting any particular matrix, including the all-zero matrix. Furthermore, if the running time is n^3 for even one of these input matrices, we are compelled to designate the algorithm as cubic-time. Clearly, using the "Sideways Gaussian Elimination", if the pivot were found in the last possible row, or even half way down, half the time and quickly the other half of the time, then we have m row additions to arrive at this conclusion, each of length a fraction of n (in fact that fraction is the ratio of the number processed columns to the number of columns remaining), and we do this $\min(m, n)$ times. If $m = n$, this is cubic time.

Instead, the algorithm given in the body of the paper would abort under these very difficult inputs. Therefore, firstly, there is never a case where the algorithm given in the body of the paper requires n^3 time, and secondly, the abort is repairable. But upon the abort, one could randomly permute the matrix and repeat. Under the random permutation, this outcome would be very rare given that the input matrix were similar. And so it is possible to calculate the number of trials expected, and it becomes clear that even 3 trials would be very unusual. And surely $n^3/\log n$ times 2 or 3 is less than n^3.

This is basically no different than getting $f'(x) = 0$ during Newton's Method. For a continuous non-constant function, the probability is zero for a uniformly distributed initial guess in some neighborhood, and so one can simply restart in the event of an abort.

The author does not believe this scenario to be typical or realistic, but it does explain why "Sideways Gaussian Elimination" is not a panacea. For the theoretical running times, we require the 3k rows approach, but in practice, we strongly recommend the "Sideways Gaussian Elimination" method.

B A Useful Lemma

Note to the Referee: *This particular appendix is included for the referee's convenience, but should probably be deleted.*

Surely this is well-known but it is useful here. The same argument can be used to calculate that an $n \times n$ matrix filled with fair coins is non-singular with probability $0.28879\cdots$, as $n \to \infty$.

Lemma 1 *A random $\mathbb{GF}(2)$ matrix of dimension $3k \times k$, filled by fair coins, has full rank with probability $\approx 1 - 2^{-2k}$.*

Proof: Consider the columns of the matrix as vectors. One can attempt to count the number of possible full rank matrices. The first vector can be any one of $2^{3k} - 1$ length $3k$ non-zero vectors. The second one can be any non-zero vector distinct from the first, or $2^{3k} - 2$ choices. The third one can be any non-zero vector not equal to the first, the second, or their sum, or $2^{3k} - 4$ choices. The ith vector can be any vector not in the space spanned by the previous $i-1$ vectors (which are linearly independent by construction). Thus $2^{3k} - 2^{i-1}$ choices are available. Therefore, the probability of any k vectors of length $3k$ being linearly independent is

$$\frac{\prod_{i=1}^{i=k} \left(2^{3k} - 2^{i-1}\right)}{(2^{3k})^k} = \prod_{i=1}^{i=k} \left(1 - 2^{i-1}2^{-3k}\right)$$

$$\begin{aligned} &\approx 1 - \sum_{i=1}^{i=k} 2^{i-1} 2^{-3k} \\ &\approx 1 - 2^{-3k}(2^k - 1) \approx 1 - 2^{-2k} \end{aligned}$$

and this is the desired result.

C Implementation Details

Note to the Referee:*I would prefer to keep this appendix but if we need to get the page-count down, as the instructions for authors says 8 pages, I would delete this section.* The following small tricks helped improve the implementation of the algorithm while writing the software library.

C.1 Bulk Logical Operations

Some microprocessors have operations which can do 32-bit, 64-bit or even 128-bit logical-AND and logical-XOR. This allows one to add 32, 64, or 128, field elements in a single instruction. In order to take advantage of this, the main method of storing a row in our library is to store the entries for the first 64 columns in the first memory word, the second 64 columns in the second memory word, and so forth.

To read a single bit, take a logical-AND with the word containing this bit, with a mask consisting of all zeroes, and a one in the position of the bit of interest. This logical-AND will be non-zero if and only if the original bit of interest were zero.

Therefore, reading a single bit becomes slightly slower, but an add becomes roughly 64× faster, which is certainly a net gain, since Stage 2 and Stage 3 are mostly vector additions. Therefore, we simply use this technique all the time.

C.2 Fixed Addressing Does Not Help

With the above in mind, it might make sense to choose $k = 8$ or $k = 16$ to simplify the addressing, since one can easily read bytes or 16-bit words. In these cases, the columns that will generate the codewords will all come from a single read instruction, and not several read instructions.

For example, with $k = 11$, for the third iteration, columns $23, \ldots, 33$ would be the source of the codewords, and would stretch across three bytes (bits 16–23, bits 24–31, and bits 32–39).

It turns out that this is all moot, because reading the data for the codewords is a very minor operation, whereas k has a significant influence on the running time of the algorithm. Therefore, going with fixed addressing, while it lowered the complexity of the code, was not useful in improving the running time.

C.3 Operating on A^T

In computer algebra, frequently we work with A^T rather than with A. For example in matrix multiplication via the naïve algorithm, it makes sense to transpose the right-hand matrix, which is a quadratic and therefore cheap operation, so that the cubicly many memory reads will be row-wise in all cases, leading to spatial locality.

In our algorithm, the running time is $\Theta(nm \min(m,n)/\log m)$, and so if $m \ll n$ we would rather have $\Theta(nm \min(m,n)/\log n$ This will impact each application as follows.

For inverting a matrix, this is of no use, as $RREF\left([A|I]^T\right) = [I|0]^T$. For LUP factorization, if $A^T = LUP$ then $A = P^T L^T U^T$. However, if one desires an inverse, one can perform the LUP factorization, then use Strassen's Matrix Inversion Formula [23] (here Equation 1, along with Strassen's Algorithm for Matrix Multiplication and The Method of Four Russians for Matrix Multiplication) to calculate the inverse after rapidly inverting L, U, P, see Subsection 5.2.

To solve $Ax = b$, one could perform that LUP-factorization of A^T, and then find $P^T z = b$, followed by $L^T y = z$ and $U^T x = y$. The last three "back-solves" are quadratic or better and therefore cheap operations. For finding the rank, note that the rank of A is the rank of A^T as well.

Thus in each case, we can use the transpose and run in time equivalent to $\Theta(nm \min(m,n)/\log \max(m,n))$. However, in fairness, some applications such as F4 Gröbner Bases calculations simply require the RREF of A, and taking the transpose does not help, for reasons similar to the case of A^{-1} above.

C.4 Cache-Friendly Operations

It turns out that performance is greatly affected if the Gray Code tables can fit in the L2 cache or not. This leads to an optimization whereby several Gray Code tables are used simultaneously, but with smaller values of k. This is described in [5] for matrix multiplication using the original Method of Four Russians Matrix Multiplication Algorithm given in [4].

D Working in $\mathbb{GF}(q)$, with $3 \leq q \leq 7$

The algorithm will proceed identically for $\mathbb{GF}(q)$ as for $\mathbb{GF}(2)$, but one has to use a Gray Code over $\mathbb{GF}(q)$. The Gray Code will still enumerate all the vectors in the subspace generated by the k rows. The only difference is that we must *subtract* and not *add* in Stage 3. The running time will have the base of the logarithm be equal to the size of the field.

Therefore, doing this over even $\mathbb{GF}(7)$ might be non-productive, because $\log_7 n$ is unlikely to be large enough to overcome the overhead of the algorithm. However, we did not attempt an implementation over fields other than $\mathbb{GF}(2)$. We imagine that over $\mathbb{GF}(3)$ the algorithm would be an improvement, but we know of no applications.

D.1 Generating the Gray Code Itself

The generation of a Gray Code of size ℓ over a finite alphabet of size q, given that of size $\ell - 1$ is done as follows: list the Gray Code of size $\ell - 1$ forwards, with a prefix of the first alphabet character attached to each codeword. Then list the Gray Code of size $\ell - 1$ backwards, with a prefix of the second alphabet character attached to each codeword. Next, list the Gray Code of size $\ell - 1$ forwards, with a prefix of the third alphabet character attached to each codeword. This is repeated until all alphabet characters have been prefixed to a codeword.

This can be done recursively to generate a Gray Code of any length, because the Gray Code of length 1 is simply the alphabet itself, starting with 0.

It is obvious that this is a Gray Code of length ℓ because of three facts. First, there are q times as many words in the new code as the Gray Code of length $\ell - 1$, which makes for the correct size. Second, there are no duplicate words in the new Gray Code, because each "pass" over the alphabet has a distinct prefix, and the Gray Code of length $\ell - 1$ has no duplicates. Third, each codeword differs in exactly one spot from the word before it. This is true if the two codewords come from the same $\ell - 1$ word pass, because the Gray Code of length $\ell - 1$ has that property. This is true if the two codewords come from distinct passes, because they are either both the first codeword or both the last codeword from the code of length $\ell - 1$, but with distinct prefixes.

E Comparison to and Combining with Strassen's Methods

Because the running time is $\Theta(n^3/\log n)$, the reader may be surprised to see so much time spent on an algorithm of this complexity, since Strassen's Algorithm runs in time $\Theta(n^{2.807\cdots})$. Recall that Strassen's paper [23] contained three algorithms: one for multiplication of matrices, one for the inversion of non-singular matrices, and one for the determinant. We call the first Strassen's Matrix Multiplication Algorithm (SMMA) and the second Strassen's Matrix Inversion Formula (SMIF), here labeled as Equation 1.

As during SMMA, during SMIF the matrix is cut into quadrants repeatedly. Yet if certain submatrices are not invertible, the algorithm must abort, as the inverses of those submatrices are used in the formula. (See [7, Ch 5.8] for details). This was resolved by Bunch and Hopcroft [11], and further by Ibara, Moran, and Hui (IMH) [18]. These algorithms are more complex and therefore entail a very large coefficient. Note, the IMH algorithm is sometimes called the LQUP algorithm.

While Magma [1] is closed source, the running times of inversions of $\mathbb{GF}(2)$ matrices of various size in Magma, when plotted in a log-log graph, demonstrate an eventual slope of $2.807\ldots$, thus indicating that Strassen's family of algorithms is being used. Our algorithm is faster than both Gaussian Elimination and the combination of SMIF/SMMA used by Magma for $n \times n$ matrices of intermediate size ($100 < n < 64000$) on certain architectures, see Section 6.1. For the latest running times, on several architectures, including newer versions of Magma, see `http://m4ri.sagemath.org/`.

Because a matrix of size $64,000 \times 64,000$ has roughly 4.1 billion entries, we were not able to test larger sizes.

For matrices of larger dimension, using IMH will allow the LUP factorization of A to be done by multiplies and inverses of non-singular matrices. Recall that Strassen's class of algorithms, Bunch-Hopcroft, and IMH each work by repeatedly cutting the matrix into pieces and then performing a recursive call on each submatrix. While this can be

done until single element sub-matrices are reached, that would be inefficient. Instead, one cuts until the sub-matrices are smaller than some n_0, at which time they are multiplied, inverted, or factored by "other techniques," and the final answer re-assembled. The M4RI algorithm can become that "other technique", instead of Gaussian Elimination or some variant of it.

While this only provides a constant speed-up, the speed-up should be roughly $0.23 \log_2 n_0$ where n_0 is the cross-over point, and so could be as large as 3.22–3.68 since $16,000 \leq n_0 \leq 64,000$ on the architectures tested.

Currently, such operations are planned for SAGE but are not implemented yet, as most users rarely enter matrices of size $> 64,000$ rows and columns.

F Historical Notes

Note to the Referee: *This will almost certainly be deleted, only to be retained in "the full version" of the paper on my webpage, along with any data tables the referee wishes to remove. Unless, of course, the referee wants this appendix to remain.*

The paper [6] refers to its algorithm as Kronrod's Algorithm, after one of the authors, a nomenclature not currently in use but advocated by Dan Bernstein [14], and this author. The origin of the alternative name "Method of 4 Russians" is obscure, but [4] says that it is named for the "cardinality and nationality of its inventors." Later, it came to be known that not all the authors are Russians.

Kronrod's algorithm finds one step of the transitive closure of a graph. This is equivalent to squaring a matrix over the boolean semiring. However, it is clear how to convert it to multiplication, and the M4RM algorithm for matrix multiplication (for the semiring) appeared in [4, Ch. 6]. The extension to $\mathbb{GF}(2)$ is not difficult, and is found in [7, Ch. 5]. Implementation details are given in [5].

Given that M4RI is difficult to say, Clément Pernet suggested the name "Mary", equivocating the 4 with an "A".

The Gray Code [16] is often thought to refer to a color midway between white and black, but actually refers to Frank Gray, who patented a vacuum tube assembly to generate the code, which he believed must have been known for quite some time.

Size	128	256	362	512	724	1020	1448	2048
Best k	5 or 6	6	7	7 or 8	8	8 or 9	9	9
M4RI	0.09571	0.650	1.68	4.276	11.37	29.12	77.58	204.1
Gauss	0.1871	1.547	4.405	12.34	35.41	97.99	279.7	811.0
Ratio	1.954	2.380	2.622	2.886	3.114	3.365	3.605	3.974
Ratio / $\log_2(n)$	0.279	0.298	0.308	0.321	0.328	0.337	0.343	0.361

Table 1: Experiment 1— Optimal Choices of k, and running time in seconds, using Opt 0.

k	1,024	1,536	2,048	3,072	4,096	6,144	8,192	12,288	16384
5	870	2,750	6,290	20,510	47,590	—*	—*	—*	–
6	760	2,340	5,420	17,540	40,630	132,950	—*	1,033,420	–
7	710	2,130	4,850	15,480	35,540	116,300	—*	903,200	–
8	680	2,040	4,550	14,320	32,620	104,960	242,990	798,470	–
9	740	2,100	4,550	13,860	30,990	97,830	223,270	737,990	1,703,290
10	880	2,360	4,980	14,330	31,130	95,850	215,080	690,580	1,595,340
11	1,170	2,970	5,940	16,260	34,020	99,980	218,320	680,310	1,528,900
12	1,740	4,170	7,970	20,470	41,020	113,270	238,160	708,640	1,557,020
13	2,750	6,410	11,890	29,210	55,970	147,190	295,120	817,950	1,716,990
14	4,780	10,790	19,390	45,610	84,580	208,300	399,810	1,045,430	–
15	8,390	18,760	33,690	77,460	140,640	335,710	623,450	1,529,740	–
16	15,290	34,340	60,570	137,360	246,010	569,740	1,034,690	2,440,410	–

*Indicates that too many aborts occurred due to singular submatrices.

Table 2: Experiment 3: Running times, in msec, Optimization Level 0

error of k	1,024	1,536	2,048	4,096	6,144	8,192	12,288	16384
-4	—	—	48.0%	53.6%	38.7%	–	32.8%	—
-3	27.9%	34.8%	26.6%	31.1%	21.3%	–	17.4%	—
-2	11.8%	14.7%	11.7%	14.7%	9.5%	13.0%	8.5%	11.4%
-1	4.4%	4.4%	3.3%	5.3%	2.1%	3.8%	1.5%	4.3%
Exact	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
+1	8.8%	2.9%	3.4%	0.5%	4.3%	1.5%	4.2%	1.8%
+2	29.4%	15.7%	17.3%	9.8%	18.2%	10.7%	20.2%	12.3%
+3	72.1%	45.6%	47.7%	32.4%	53.6%	37.2%	53.7%	—
+4	155.9%	104.4%	110.8%	80.6%	117.3%	85.9%	124.9%	—
+5	304.4%	214.2%	229.1%	172.9%	250.2%	189.9%	258.7%	—
+6	602.9%	428.9%	458.9%	353.8%	494.4%	381.1%	—	—

Table 3: Percentage Error for Offset of K, From Experiment 3

Dimension	4,000	8,000	12,000	16,000	20,000	24,000	28,000	32,000
Gaussian	19.00	138.34	444.53	1033.50	2022.29	3459.77	5366.62	8061.90
7	7.64	–	–	–	–	–	–	–
8	7.09	51.78	–	–	–	–	–	–
9	6.90	48.83	159.69	364.74	698.67	1195.78	–	–
10	7.05	47.31	151.65	342.75	651.63	1107.17	1740.58	2635.64
11	7.67	48.08	149.46	332.37	622.86	1051.25	1640.63	2476.58
12	–	52.55	155.51	336.11	620.35	1032.38	1597.98	2397.45
13	–	–	175.47	364.22	655.40	1073.45	1640.45	2432.18
14	–	–	–	–	–	–	1822.93	2657.26
Min	6.90	47.31	149.46	332.37	620.35	1032.38	1597.98	2397.45
Gauss/M4RI Ratio	2.75	2.92	2.97	3.11	3.26	3.35	3.36	3.36
Ratio $/ \log_2 n$	0.230	0.225	0.219	0.222	0.228	0.230	0.227	0.225

Table 4: Experiment 4: Optimization Level 3, Flexible k

	Computer I		Computer II		Computer III	
Dimension	MAGMA	M4RI	MAGMA	M4RI	MAGMA	M4RI
10,000 × 10,000	2.693	1.451	3.283	2.509	8.069	3.418
16,384 × 16,384	8.476	6.500	11.204	10.741	25.828	19.987
20,000 × 20,000	14.417	11.566	16.911	19.776	43.256	30.829
32,000 × 32,000	45.739	40.450	57.761	86.071	157.134	118.121
64,000 × 64,000	286.716	291.252	355.477	640.742	892.481	874.609

Table 5: Experiment 5: Comparison between Latest Code and MAGMA