Achieving a log(n) Speed Up for Boolean Matrix Operations and Calculating the Complexity of the Dense Linear Algebra step of Algebraic Stream Cipher Attacks and of Integer Factorization Methods

Gregory V. Bard*

May 5, 2006

Abstract

The purpose of this paper is to calculate the running time of dense boolean matrix operations, as used in stream cipher cryptanalysis and integer factorization. Several variations of Gaussian Elimination, Strassen's Algorithm and the Method of Four Russians are analyzed. In particular, we demonstrate that Strassen's Algorithm is actually slower than the Four Russians algorithm for matrices of the sizes encountered in these problems. To accomplish this, we introduce a new model for tabulating the running time, tracking matrix reads and writes rather than field operations, and retaining the coefficients rather than dropping them. Furthermore, we introduce an algorithm known heretofore only orally, a "Modified Method of Four Russians", which has not appeared in the literature before. This algorithm is $\log n$ times faster than Gaussian Elimination for dense boolean matrices. Finally we list rough estimates for the running time of several recent stream cipher cryptanalysis attacks.

Keywords: Matrix Inversion, Matrix Multiplication, Boolean Matrices, GF(2), Stream Cipher Cryptanalysis, XL Algorithm, Strassen's Algorithm, Method of Four Russians, Gaussian Elimination, LU-factorization.

1 Introduction

The objective of this paper is to determine the complexity of solving a dense boolean system of linear equations via matrix operations. By boolean, or binary matrix, we mean a matrix whose entries are from the field with two elements. We describe a matrix inversion algorithm[1] which is a variant of the boolean matrix multiplication algorithm called the "Method of Four Russians" [ADKF70]. This inversion algorithm, which we name MM4R (or Modified Method of Four Russians), achieves $\log n$ speed-up over Gaussian Elimination on dense boolean matrices. Furthermore, we demonstrate that Strassen's Matrix Inversion Formula, when used with Strassen's Matrix Multiplication algorithm, is actually *slower* than Gaussian Elimination for matrices of feasible or near-feasible size (less than one trillion entries). We discuss the impact of these facts upon stream cipher cryptanalysis as well as factoring, both of which depend heavily upon dense matrix operations.

In order to prove these facts, we propose a simple but effective computational cost model for boolean matrix operations. This model counts matrix reads and writes rather than floating point operations. It retains the coefficients of complexity polynomials, allowing one to calculate the crossover point at which the running times of two competing methods are equal. Since many matrix operations are recursive, it is critically important to calculate these crossovers. Using this model, we calculate the complexity of eleven matrix operations, which are summarized in Table 1 on Page 2.

In Section 1.1, the computational cost model used in this paper for boolean matrix operations is defined. Section 2 describes the data structures used by these algorithms. Section 4 contains the analysis of three matrix multiplication algorithms: the naïve method, the Method of Four Russians, and

*Ph.D. Candidate, Applied Mathematics, University of Maryland & Lecturer, Computer Science, American University.

[1]While this algorithm may have been known anectodally among some, the author has not been able to find it in the literature. It was shown to me by Nicholas Courtois.

Algorithm	Overdefined	Square	Underdefined
System Upper-Triangularization			
Mod. Meth. of 4 Russians (UUTF)	$(n^3 + 1.5n^2m)/(\log n)$	$(2.5n^3)/(\log n)$	$(4.5nm^2 - 2m^3)/(\log m)$
Strassen's—LU-Factorization	—	$2.2876n^{\log 7}$	$2.2876m^{\log 7} + 3mn$
Dense Gaussian Elim. (UUTF)	$0.75mn^2 - 0.25n^3$	$0.5n^3$	$0.75nm^2 - 0.25m^3$
Back Substitution	$n^2 + m$	n^2	$m^2 + n$
Matrix Inversion			
Strassen's—U/L Triang. Inversion	—	$0.9337n^{\log 7}$	—
Strassen's—General Inversion	—	$6.4893n^{\log 7}$	—
Dense Gaussian Elim. (RREF)	$0.75mn^2$	$0.75n^3$	$1.5nm^2 - 0.75m^3$
Mod. Meth. of 4 Russians (RREF)	$(3n^3 + 3mn^2)/(2\log n)$	$(3n^3)/(\log n)$	$(6nm^2 + 3m^3)/(\log m)$

Algorithm	Rectangular $a \times b$ by $b \times c$	Square $n \times n$ by $n \times n$
Multiplication		
Method of 4 Russians	$(3b^2 \min(a, c) + 3abc)/(\log b) + b \max(a, c)$	$(6n^3)/(\log n)$
Naïve Multiplication	$2abc$	$2n^3$
Strassen's Algorithm	$2.3343(abc)^{(\log 7)/(3)}$	$2.3343n^{\log 7}$

Table 1: Algorithms and Asymptotic Performance

Strassen's Matrix Multiplication Algorithm. This analysis culminates in Section 4.4, which contains an ideal strategy for boolean matrix multiplication, using the building blocks currently available for general dense matrices. Section 5 analyzes three matrix inversion methods: Gaussian Elimination, Strassen's Matrix Inversion Formula, and the Modified Method of Four Russians. In order to give a sense of scale, Section 3 lists the sizes of dense linear systems to be solved in stream cipher cryptanalysis and integer factorization problems. The conclusions and consequences for stream cipher cryptanalysis are given in Section 6.

1.1 The Computational Cost Model

In papers on matrix operations over the real or complex numbers, the number of floating point operations is used as a measure of running time. This removes the need to account for assembly language instructions needed to manipulate index pointers, iteration counters, discussions of instruction set, and measurements of how cache coherency or branch prediction will impact running time. In the present paper, floating point operation counts are meaningless, for boolean matrices do not use floating point operations. Therefore, we propose that matrix entry reads and writes be tabulated, because addition (XOR) and multiplication (AND) are single instructions, while reads and writes on rectangular arrays are much more expensive. Clearly these data structures are non-trivial in size and so memory transactions will be the bulk of the computational burden.

From a computer architecture viewpoint in particular, these matrices cannot fit in the cache of the microprocessor, and so the fetches to main memory are a bottleneck. Even if exceptionally careful use of temporal and spatial locality guarantees effective caching (and it is not clear that this is possible), the data must still travel from memory to the processor and back! The bandwidth of buses has not increased proportionally to the rapid increase in the speeds of microprocessors. Given the relatively simple calculations done once the data is in the microprocessor's registers (i.e. single instructions), surely the memory transactions are the rate-determining step.

Naturally, the exponents of the complexity polynomials of the algorithms in our model will match those of the more flexible big-Oh notation counting all operations. On the other hand, counting exactly the number of memory reads and writes permits us to calculate more directly the coefficients of those polynomials. These coefficients are helpful in three principal ways. First and foremost, they help determine the crossover point at which two competing algorithms are of equal efficiency. This is absolutely critical as matrix multiplication and inversion techniques are often recursive. Second, they help one compare two algorithms of equal exponent. Third, they help calculate the total CPU running time required to solve a system. These coefficients are derived mathematically and are not dependent on machine architectures or benchmarks, but experiments will be performed to confirm them nonetheless.

Also, due to the variations of computer architectures, the coefficients given here may vary slightly. On the other hand, by deriving them mathematically, rather than experimentally, one need not worry about artifacts of particular architectures or benchmarks skewing the results.

When attempting to convert these memory operation counts into CPU cycles, remember that other instructions are needed to maintain loops, execute field operations, and so forth. Also, memory transactions are not one cycle each, but require several instructions to compute the memory address offsets. Yet, they can be pipelined. Thus one can estimate that about 40–60 CPU cycles are needed per matrix memory operation.

1.2 Notational Conventions

Precise performance estimates are useful, and so rather than the usual five symbols $O(n)$, $o(n)$, $\Omega(n)$, $\omega(n)$, $\Theta(n)$, we will use $f(n) \sim g(n)$ to indicate that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$$

in the case that an exact number of operations is difficult to state. While $O(n)$ statements are perfectly adequate for many applications, coefficients must be known to determine if algorithms can be run in a reasonable amount of time on particular target ciphers. Also, these coefficients have an enormous impact upon crossover points, as will be shown in Section 5.3.3, comparing Strassen's formula for matrix inversion with the Modified Method of Four Russians.

Let $f(n) \leq\sim g(n)$ signify that there exists an $h(n)$ and n_0 such that $f(n) \leq h(n)$ for all $n > n_0$, and $h(n) \sim g(n)$.

Matrices in this paper are over GF(2) unless otherwise stated, and are of size m rows and n columns. Denote ℓ as the lesser of n and m. If $n > m$ or $\ell = m$ the matrix is said to be **underdefined**, and if $m > n$ or $\ell = n$ then the matrix is said to be **overdefined**. The symbol $\ln x$ means $\log_e x$ and $\log x$ means $\log_2 x$.

The term **unit upper triangular** means that the matrix is upper triangular, and has only ones on the main diagonal. Obviously such matrices have determinant one, and so are non-singular. Likewise for **unit lower triangular**.

Finally please note that some authors use the term boolean matrix to refer to matrices operating in the semiring where addition is logical OR (instead of logical XOR) and multiplication is AND. Those matrices are used in graph theory, and some natural language processing. Algorithms from the real numbers can work well over this semiring [F70], and so are not discussed here.

1.3 To Invert or To Solve?

Generally, four basic options exist when presented with a system of equations, $A\vec{x} = \vec{b}$, over the reals as defined by a square matrix A. First, the matrix A can be inverted, but this is the most computationally intensive option. Second, the system can be adjoined by the vector of constants $\vec{b}$, and the matrix reduced into a triangular form $U\vec{x} = \vec{b'}$ so that the unknowns can be found via back-substitution. Third, the matrix can be factored into LU-triangular form ($A = LUP$), or other forms. Fourth, the matrix can be operated upon by iterative methods to converge to a matrix near to its inverse. Unfortunately, in finite fields concepts like convergence toward an inverse do not have meaning. This rules out option four.

The second option can be extended to solving the same system for two sets of constants $\vec{b_1}, \vec{b_2}$ but requires twice as much work during back substitution. However, back substitution is very cheap (quadratic) compared to the other steps (near cubic), and so this is desirable. In light of this, it is easy to see that even if many $\vec{b}$ were available, but fewer than $\min(m, n)$, one would pursue the second option rather than invert the matrix. Therefore, one would normally choose this option, to put the matrix in unit upper triangular form and use back substitution. Option three, to use LU factorization or another type of factorization will turn out to be the only way to use Strassen's matrix inversion formula. Option one, inverting the matrix, is given in the appendices, but would not normally be used. The exception is if the computation is taking place long before the interception of the message, and further the attack desires the plaintext as soon as possible upon receipt.

From the view of operational Stream Cipher Cryptanalysis, a large number of messages may be intercepted, but all in different keys. The matrix is dependent on the cipher only, but the constants depend on the messages. Once the matrix is inverted, all the keys can be found rapidly by matrix multiplication (assuming all the known plaintext is located in the same spots in each message, even

if those known plaintexts are not similar). Therefore this paper focuses on inverting or factoring the matrices (the first and third options), though some examples of the second option are given in the appendices.

2 Data Structure Choices

The most elementary way to store a matrix is as an array of scalars. Two-dimensional arrays are often stored as a series of one-dimensional arrays, in sequence, or an array of pointers to those arrays. In either case, it is not obvious if the linear arrays should be rows or columns. For example, in a matrix multiplication AB with the naïve algorithm, spatial locality will be enhanced if A's rows and B's columns are the linear data structure. The following three data structures are proposed:

An Array with Swaps The cells of the matrix are a two-dimensional array, with the rows being the linear data structure, since more of the work in the algorithms of this paper is performed upon rows than upon columns. Additionally, two one-dimensional arrays called row-swap and column-swap are to be used. Initially these are filled with the numbers $1, 2, \ldots, m$ and $1, 2, \ldots, n$. When a swap of rows or columns is called for, the numbers in the cells of the row-swap or column-swap corresponding to those rows are swapped. When a cell a_{ij} is called for, the result returned is $a_{r_i.c_j}$, with r_i representing the ith entry of the row-swap array, and c_j likewise. In this manner, row and column swaps can be executed in constant time, namely two writes each.

To facilitate the rapid taking of transposes, a "transpose flag" can be used. Initially false, every time the matrix is transposed, this single bit is flipped. When true, requests for a_{ij} return a_{ji} instead.

Permutation Matrices An identity matrix which has had rows or columns swapped is called a permutation matrix. It is only necessary to store a row-swap and column-swap array as before. The row-swap and column-swap arrays allow a quick look-up by calculating

$$a_{ij} = \begin{cases} 1 & \text{if } r_i = c_j \\ 0 & \text{if } r_i \neq c_j \end{cases}$$

This means that $a_{ij} = 1$ iff it is on the main diagonal after all row and column swaps have been resolved. For example, if rows 2 and 3 have been swapped in the 3×3 case, $r = \{1, 3, 2\}$ and $c = \{1, 2, 3\}$. One can see that $r_i = c_j$ iff (r_i, c_j) is $(1, 1)$ or $(2, 3)$ or $(3, 2)$. And indeed, the locations of the ones of the matrix are those coordinates.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Furthermore, "composition" of the swap arrays in the sense of

$$r''_i = r_{r'_i}$$

allows for the multiplication of two permutation matrices in time linear to the dimension. The inverse of a permutation matrix is its transpose, and using the "transpose bit" mentioned earlier, inversion can be done in constant time. Finally multiplying a permutation matrix by a vector can be done by simply swapping the elements of that vector according to the swap array.

3 Matrices Found by Algebraic Cryptanalysis and Factoring

The two principal applications of boolean matrices in cryptography are factoring integers and the algebraic attacks on stream ciphers. This section will discuss the typical sizes of dense matrices involved.

First, in stream cipher cryptanalysis, a stream cipher is converted to a system of boolean polynomial equations whose solution is the key to the cipher. The polynomial system is then converted to a boolean linear system. This system is often sparse but its density will rise very rapidly under Gaussian Elimination. The number of equations is m, the number of rows, and the number of variables is n, the number of columns.

- Courtois and Meier attack Toyocrypt with $\binom{128}{3} = 341,376$ equations [CM03]. One must guess that the system is slightly overdefined, so therefore near square.
- Hongjun Wu attacks Whitenoise with 80189 variables, and 80445 equations. [W].
- Courtois attacks Sfinks $2^{22.5}$ equations in once case and $2^{36.5}$ in another [C05]. One must guess that the system is slightly overdefined, so therefore near square.
- Hawkes and Rose state that the best known attacks against E-0, Lili-128, and Toyocrypt have $2^{18}, 2^{12},$ and 2^7 equations. It is clear from context that the matrices are nearly square [A02, HR04].

Second, in integer factorization, once all the relations have been found in an algorithm similar to the Quadratic Field Sieve, one must find a vector in the null space of a matrix. This can be done by first preconditioning the matrix [PS92], then using Sparse Gaussian Elimination and finally a applying dense matrix operation step on a matrix of reduced size [AGLL94].

The sizes of this final dense matrix operation in practice can be described by the process to factor the 120 digit prime number protecting the message "the magic words are squeamish ossifrage" from the paper of the same name [AGLL94]. A sparse $569,466 \times 524,339$ boolean matrix was reduced to a dense $188,614 \times 188,160$ dense boolean matrix, which was solved in 12 CPU hours of a Sun Sparc 10 workstation, at that time (see below for today's estimate).

To give some scale to the feasibility of the algorithms as applied to the attacks discussed here, note that a Cray X1E[2] is a 174 Tflop machine, performing 4.6×10^{21} operations in a year, or $11.5 \ldots 7.6 \times 10^{19}$ matrix memory operations using our assumption of 40–60 instructions times the number of reads/writes. Using 10^{20} matrix operations per year as a round value, one year of computation could be used to solve a system of size

- Gaussian Elimination: 5.85×10^6 rows and columns.
- Modified Method of Four Russians: 9.75×10^6 rows.
- LU Factorization with Strassen: 9.93×10^6 rows.
- Direct Inversion with Strassen: 6.85×10^6 rows.

This permits us to calculate how long the above attacks would take, ignoring the effects of sparsity which may be very important in some cases. These are calculated using the 10^{20} matrix operations per year figure from above, and Modified Method of Four Russians to Upper Triangular Form, or $2.5n^3/\log n$.

- Courtois and Meier vs. Toyocrypt: 5.41×10^{15} matrix ops, or 28 minutes.
- Hongjun Wu vs. Whitenoise: 7.91×10^{13} matrix ops, or 25 seconds.
- Courtois vs. Sfinks ($2^{22.5}$ case): 2.3×10^{19} matrix ops, or 84 days.
- Courtois vs. Sfinks ($2^{36.5}$ case): 6.28×10^{31} matrix ops, or half a trillion years.
- Best known attack on E-0: 2.5×10^{15} matrix ops, or 13 minutes.
- Best known attack on Lili-128: 1.43×10^{10} matrix ops, or 5 milliseconds.
- Squeamish Ossifrage: 9.57×10^{14} matrix ops, or 5 minutes.

One can conclude three things from this list. First, that matrix operation running times are highly dependent on dimension. Second, that algebraic attacks are indeed feasible against all but one of the above systems. Third, that further study is needed. However, one should also mention that while showing a quick running time is proof an attack is feasible, showing a slow running time is not proof of infeasibility, because sparse methods may be successful for the trillion year dense attack above.

4 Matrix Multiplication Algorithms

While this paper focuses on matrix inversion algorithms, many of them use matrix multiplication algorithms as subroutines. Therefore, matrix multiplication algorithms must be analyzed also. For comparison, we list three algorithms for matrix multiplication. These algorithms are candidates for the multiplication operations in Strassen's formula for matrix inversion, and so impact the complexity of matrix inversion directly.

A great deal of research was done in the period 1969–1987 on fast matrix operations [H, P]. Various proofs showed that many important matrix operations, such as QR-decomposition, LU-factorization,

[2]Data from the Cray website.

inversion, finding determinants, and finding the characteristic polynomial are no more complex than matrix multiplication, in the big-Oh sense [H, AHU, CLRS]. For this reason, many fast matrix multiplication algorithms were developed [S69, S81, S87]. One in particular was intended for GF(2), and has been named the Method of Four Russians, "after the cardinality and the nationality of its inventors" [AHU]. While the Method of Four Russians was conceived as a matrix multiplication tool, we show how to use a substantial modification of it for inversion. Of the general purpose algorithms, the most famous and frequently implemented of these is Volker Strassen's 1969 algorithm for matrix multiplication.

4.1 Method of Four Russians Matrix Multiplication

This matrix multiplication algorithm is the original algorithm published by Arlazarov, Dinic, Kronrod, and Faradzev [ADKF70]. Consider a product of two matrices $AB = C$ where A is an $a \times b$ matrix and B is a $b \times c$ matrix, yielding an $a \times c$ for C. In this case, one could divide A into b/k vertical "stripes" $A_1 \dots A_{b/k}$ of k columns each, and B into b/k horizontal stripes $B_1 \dots B_{b/k}$ of k rows each. (For simplicity assume k divides b). The product of two stripes, $A_i B_i$ is an $a \times b/k$ by $b/k \times c$ matrix multiplication, and yields an $a \times c$ matrix C_i. The sum of all k of these C_i equals C.

$$C = AB = \sum_{i=0}^{i=k} A_i B_i$$

The algorithm itself proceeds as follows:

- for $i = 1, 2, \dots, b/k$ do
  - Make a Gray Code[3] table of the 2^k linear combinations of the k rows of B_i. Call the xth row T_x. (Costs $(3 \cdot 2^k - 4)c$ reads/writes, see Modified Method of Four Russians, in Section 5.1).
  - for $j = 1, 2, \dots, a$ do
    - Read the entries $a_{j,(i-1)k+1}, a_{j,(i-1)k+2}, \dots, a_{j,(i-1)k+k}$ and treat as a k bit binary number x. (Costs k reads.)
    - for $h = 1, 2, \dots, c$ do
      - Calculate $C_{jh} = C_{jh} + T_{xh}$ (Costs 3 reads/writes).

The innermost loop requires 3 steps, the next loop out requires $k + 3c$ steps, and then the next after that requires $(3 \cdot 2^k - 4)c + a(k + 3c)$ steps. Finally the entire algorithm requires

$$\frac{b((3 \cdot 2^k - 4)c + a(k + 3c))}{k} = \frac{3b2^k c - 4cb + abk + 3abc}{k}$$

memory operations. Substitute $k = \log b$, so that $2^k = b$, and observe,

$$\frac{3b^2 c - 4cb + ab \log b + 3abc}{\log b} \sim \frac{3b^2 c + 3abc}{\log b} + ab$$

For square matrices this becomes $\sim (6n^3)/(\log n)$.

Transposing the Matrix Product Since $AB = C$ implies that $B^T A^T = C^T$, one can transpose A and B, and transpose the product afterward. This would have running time $(3b^2 a + 3abc)/(\log b) + cb$ and some manipulations will show that this more efficient when $c < a$, for any $b > 1$. Therefore the final complexity is $\sim (3b^2 \min(a, c) + 3abc)/(\log b) + b \max(a, c)$. To see that the last term is not optional, substitute $c = 1$, in which case the last term becomes the dominant term.

4.2 Naïve Matrix Multiplication

For comparison, we calculate the complexity of the naïve matrix multiplication algorithm, for a product $AB = P$ with dimensions as before.

- for $i = 1, 2, \dots, a$
  - for $j = 1, 2, \dots, c$
    - Calculate $C_{ij} = A_{i1} B_{1j} + A_{i2} B_{2j} + \cdots A_{ib} B_{bj}$. (Costs $2b + 1$ reads/writes).

This clearly requires $2abc + ac \sim 2abc$ operations, or for square matrices $2n^3 + n^2$ operations.

[3] A k-bit Gray Code is all 2^k binary strings of length k, ordered so that each differs by exactly one bit in one position from each of its neighbors. For example, one 3-bit Gray Code is $\{000, 001, 011, 010, 110, 111, 101, 100\}$.

Crossover with Method of Four Russians Using the exact formula, or $(6n^3-4n^2+n^2\log n)/(\log n)$ for square matrices, via the Method of Four Russians, and $2n^3+n^2$ for the naïve algorithm, a crossover point can be found. It turns out to be very small, a 6×6 matrix requiring 482 for the Method of 4 Russians, versus 468 for the naïve algorithm, and a 7×7 requiring 712 for the Method of 4 Russians, and 735 for the naïve algorithm. In reality the crossover point maybe slightly higher due to machine architectures, but it is clear that the Method of 4 Russians should be prefered for matrices of any non-trivial size.

4.3 Strassen's Algorithm for Matrix Multiplication

Strassen's Algorithm for multiplying $AB=C$ consists of dividing the matrix into pieces as follows:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

The product is then given explicitly as 18 matrix additions and 7 matrix multiplications (See Appendix C). The matrices c_{11} and c_{22} must be square, but need not equal each other in size. The matrices c_{12} and c_{21} can be rectangular. For simplicity assume that A and B are both $2n\times 2n$ matrices. The 18 additions of a pair of $n\times n$ matrices requires $3n^2$ work (one write and two reads per entry) for a total of $54n^2$ operations. The seven multiplications are to be performed by repeated calls to Strassen's algorithm. In theory one could repeatedly call the algorithm until 1×1 matrices are the inputs, and multiply them with a logical AND operand. However, its unlikely that this is optimal. Instead, the program should switch from Strassen's algorithm to some other algorithm below some size n_0. Using the formula,

$$M(2n)=7M(n)+54n^2$$

where $M(n)$ is the number of reads and writes required to multiply a $n\times n$ matrix with Strassen's Algorithm. Some algebra yields (See Appendix C)

$$M(n)\sim (n^{\log 7})\frac{M(n_0)}{n_0^{\log 7}}$$

Therefore one can determine n_0 (to be calculated by the naïve algorithm or, more efficiently, the Method of Four Russians) by minimizing $(M(n_0)+18n_0^2)/n_0^{\log 7}$, which becomes

$$\frac{6n_0^3/(\log n_0)+18n_0^2}{n_0^{\log 7}}$$

which has a minimum at $n_0=610$. This is in contrast to the real and complex case, where Higham states that the Cray matrix operations library uses Strassen for dimension 130 and higher [H].

Crossover with Four Russians Clearly things would work better with powers of two (our equations assume $\log n$ is an integer, and ignore the consequences of rounding if it is not) so one can evaluate the above expression at $n_0=512$ or $n_0=1024$ and see that 512 is lower. Therefore the author recommends a crossover at 512. Square matrix multiplications of size less than 513 should be done via the Method of Four Russians, and of size 513 and higher by Strassen's Algorithm. The final performance is therefore

$$M(n)\sim (n^{\log 7})\frac{M(512)}{512^{\log 7}}\sim (n^{\log 7})\frac{\frac{6(512)^3}{\log 512}}{512^{\log 7}}\sim 2.3343n^{\log 7}$$

Strassen Without Four Russians If one were to attempt to use Strassen's Algorithm with naïve multiplication, the crossover would be the minimum of

$$\frac{M(n_0)+18n_0^2}{n_0^{\log 7}}=\frac{2n_0^3+19n_0^2}{n_0^{\log 7}}$$

which has a minimum at $n_0=40$, coefficient equal to 5.037, but for reasons mentioned above, powers of two are more convenient. One can try $n_0=32$ and $n_0=64$ and observe 32 is better, with $M(n)\sim 5.057n^{\log 7}$, almost twice as slow as the Strassen-Four Russian combination.

Pure Strassen In its purest form, Strassen's algorithm would be used to divide the matrices into 2×2 pieces, which can be multiplied in 8 reads and 4 writes, or 12 operations. The coefficient is then

$$M(n) = n^{\log 7} \frac{12 + 18 \cdot 2^2}{2^{\log 7}} = 12n^{\log 7}$$

which is more than 5 times slower than the Strassen-Russians combination.

4.4 Best Choices for Matrix Multiplication

We have explicitly calculated that the most efficient square matrix multiplication algorithms are

- For $n < 7$ (or slightly higher) by the naïve algorithm.
- For $7 \leq n \leq 512$ by the Method of Four Russians.
- For $n > 513$ by Strassen's Algorithm

It is more difficult to figure out where crossover should be for rectangular matrices. Schönhage and Strassen ([S87, S81] but not in [S69]) both state that an $a \times b$ by $b \times c$ multiplication is $(abc)^{(\log 7)/3}$ with Strassen's algorithm. Note that $7 \leq (abc)^{(\log 7)/3} \leq 512$ if and only if $8 \leq abc \leq 786$, therefore a good guideline is

- For $abc < 8$ (or slightly higher) by the naïve algorithm.
- For $8 \leq abc \leq 768$ by the Method of Four Russians.
- For $abc > 769$ by Strassen's Algorithm

5 Dense Matrix Triangulation Techniques

As stated earlier, even if one has several $\vec{b_1}, \vec{b_2}, \vec{b_3}, \ldots, \vec{b_n}$, it is far more efficient to solve $A\vec{x_i} = \vec{b_i}$ by appending the b_i as columns to the end of matrix A, and putting matrix A in unit upper triangular form (UUTF). Then, one can solve for each x_i by running the back substitution algorithm (See Appendix D) to obtain the x_i. The alternative is to invert A, and Appendix B contains algorithms for that approach, by putting A into row reduced echelon form (RREF).

5.1 Modified Method of Four Russians

The author would like to express gratitude to Nicolas Courtois who explained the following algorithm to him after Eurocrypt 2005 in Århus, Denmark. The Method of Four Russians, proposed in [ADKF70], is a boolean matrix multiplication algorithm (given in Section 4), and is not well known. However, the variant shown to the author is for inversion, given below, and it does not seem to be in the literature.

In Gaussian Elimination (UUTF) of an $m \times n$ matrix, each iteration i operates on the submatrix $a_{ii} \ldots a_{mn}$, with the objective of placing a one at a_{ii} and a zero at every other entry of the column i below row i. In the Modified Method of Four-Russians (UUTF) algorithm, k columns are processed at once, producing a $k \times k$ identity matrix in the correct spot ($a_{ii} \ldots a_{(i+k-1),(i+k-1)}$), with all zeros below it, and leaving the region above it untouched.

Stage 1: Denote the first column to be processed in a given iteration as a_i. Then, perform Gaussian elimination on the first $3k$ rows after and including the ith row to produce an identity matrix in $a_{ii} \ldots a_{(i+k-1),(i+k-1)}$. (To know why it is reasonable to expect this to succeed, see Section 5.1.2).

Stage 2: Construct a table consisting of the 2^k binary strings of length k in a Gray Code. The first entry is all zeroes, and is associated with an all zero row of length equal to $n - (i + k) + 1$. The next entry is $0 \cdots 01$, and is associated with the entries from the ith row, from column $i + k$ until column n. The next entry after that is $0 \cdots 11$, and is associated with the entries from the ith and $(i + 1)$th rows, same columns, added together. Since any Gray Code has each line differing from the previous by exactly one bit, only one vector addition is needed to calculate each line of this table.

Stage 3: One can rapidly process the remaining rows from $i+k$ until the last row m by using the table. For example, suppose the jth row has entries $a_{ji} \ldots a_{j,(i+k-1)}$ in the columns being processed. Selecting the row of the table associated with this k-bit string, and adding it to row j will force the k columns to zero, and adjust the remaining columns from $i+k$ to n in the appropriate way, as if Gaussian Elimination had been performed.

5.1.1 Performance Analysis

Surely k must be small, otherwise exponential work is required to produce the 2^k entry table. But if 2^k is small compared to the number of rows m, then each row of the table will be used several times. This saves computation, which must be offset by the cost of table construction. The value of k is usually set to $\log n$ and numerical experimentation[4] has shown that $1 + \log \ell - \log\log \ell$ is a local minimum if not global. (Recall, $\ell = \min(m, n)$). Assume for simplicity that $\log n$ divides n.

- For $i = 0 \ldots \ell/k - 1$ do
  - Do Gaussian Elimination on rows $ki+1$ to $ki+3k$ to obtain a $k \times k$ identity matrix starting at $a_{(ki+1)(ki+1)}$.
  - For $j = 1 \ldots 2^k - 1$ do
    - Generate the $n - ik - k$ entries of this row of the Gray Code table by vector addition of the previous row, and the appropriate row of the matrix. Denote T_x the xth row of the table. (Costs 2 reads and 1 write per each of $n - ik - k$ entries).
  - For $j = ik + 3k + 1 \ldots m$ do
    - Read the k bit string at $a_{(ik+1),j} \ldots a_{(ik+k),j}$. (Costs k reads). Consider this string as a binary number x.
    - Set to zero the k entries at $a_{(ik+1),j} \ldots a_{(ik+k),j}$. (Costs k writes).
    - For $p = ik + k + 1 \ldots n$ do
      - Execute $a_{jp} = a_{jp} + T_{x,(p-ik-k)}$ (Costs 2 reads/1 write).

The first stage is a $3k \times n - ik$ underdefined Gaussian Elimination (RREF), which by the appendix requires $\sim 1.5(3k)(n-ik)^2 - 0.75((3k)^3)$ operations. This will be negligible.

The second stage, constructing the table, requires $3(n - ik - k)$ steps per row. The first row is all zeroes and can be hard-coded, and the second row is a copy of the appropriate row of the matrix, and requires $(n - ik - k)$ reads followed by writes. Thus we obtain $2(n - ik - k) + (2^k - 2)(3)(n - ik - k) = (3 \cdot 2^k - 4)(n - ik - k)$ steps.

The third stage, executed upon $(m - ik - 3k)$ rows (if positive) requires $2k + 3(n - ik - k)$ reads/writes per row. This becomes $(m - ik - 3k)(3n - 3ik - k)$ operations in total, when that total is positive. For example, in a square matrix the last 2 iterations of stage 1 will take care of all of these rows and so there maybe no work to perform in Stage 3 of those iterations. To denote this, let $pos(x) = x$ if $x > 0$ and $pos(x) = 0$ otherwise.

Adding steps one, two and three yields

$$
\begin{aligned}
&\sum_{i=0}^{i=\ell/k-1} 1.5(3k)^2(n-ik) - 0.75((3k)^3)(3\cdot 2^k - 4)(n-ik-k) + (pos(m-ik-3k))(3n-3ik-k) \\
&\left[\sum_{i=0}^{i=\ell/k-3} 1.5(3k)^2(n-ik) - 0.75((3k)^3)(3\cdot 2^k - 4)(n-ik-k) + (m-ik-3k)(3n-3ik-k)\right] \\
&+1.5(3k)^2(n-\ell+2k) - 0.75((3k)^3)(3\cdot 2^k - 4)(n-\ell+k) \\
&+1.5(3k)^2(n-\ell+k) - 0.75((3k)^3)(3\cdot 2^k - 4)(n-\ell) \\
\leq\ &\sim \frac{1}{4k}\left[2^k(-6k\ell + 12n\ell - 6\ell^2) - 6m\ell^2 - 6n\ell^2 + 4\ell^3 + 12mn\ell\right]
\end{aligned}
$$

Substituting $k = \log \ell$ and thus $2^k = \ell$,

$$\frac{1}{4\log \ell}\left(6n\ell^2 - 2\ell^3 - 6m\ell^2 + 12mn\ell\right)$$

[4]Experiments run in an OpenOffice spreadsheet, using the running time formulas given here, for square matrices.

Thus for the over-defined case ($\ell = n$) this is $(4n^3 + 6n^2m)/(4 \log n)$, and for the under-defined case ($\ell = m$) this is $(18nm^2 - 8m^3)/(4 \log m)$, and for square $(5n^3)/(2 \log n)$.

5.1.2 The Rank of $3k$ Rows

The reader may be curious why $3k$ rows are selected instead of k rows at the small Gaussian Elimination step (Stage 1 of each iteration). The answer is that the probability of k vectors of length $3k$ having rank k is very high, as proved below. The small Gaussian Elimination will only fail to produce the identity matrix followed by rows of zeroes, if this submatrix is not of full rank.

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

Proof: Consider the columns of the matrix as vectors. We now attempt to count the number of possible full rank matrices. The first vector can be any one of $2^{3k} - 1$ non-zero length $3k$ 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 $2^{3k} - 4$. 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

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

And this is the desired result. ♣
In the case $k = 5$, the actual probability of less than full rank is 9.46×10^{-4}, and the above formula has a relative error of 3.08×10^{-6}, and would be even more accurate for higher k. Also, note when $k = \log \ell$ then the probability of full rank is $1 - \ell^{-2}$. Since there will be $(\ell)/(\log \ell) - 1$ iterations, the probability of even one failure during all passes is less than $1/(\ell \log \ell)$, which is very low.

5.1.3 Using Bulk Logical Operations

The above algorithm can be improved upon if the microprocessor has instructions for 32-bit (or even 64-bit) logical operations. Stages 2 and 3 essentially consist of repeated row additions. The matrix can be stored in an 8-bits per byte format instead of the 1-bit per byte format, and long XOR operations can perform these vector additions. The $3k$ rows being operated upon in stage 1 must be "expanded" out to 1 bit per byte (assuming single byte addressing is available) and "compressed back" before stages 2 and 3. However, stages 2 and 3 can proceed 32 or 64 times as fast as normal if single-instruction logical operators are available.

The expansion and compression will be performed three times upon each cell in the first ℓ rows. An expansion consists of one read, and eight writes per byte, and a compression of eight reads and one write per byte, or 9/8 reads/writes per bit. Therefore the total cost is $3(2)(9/8)(\ell n)$, which is quadratic and therefore negligible in our analysis. Since only stages 2 and 3 were non-negligible, it is safe to say that the algorithm would proceed 32 or 64 times faster, for sufficiently large matrices.

5.2 Strassen's Formula for the Inversion of Real Matrices

In Strassen's original paper on matrix multiplication, the following formula appeared for matrix inversion. However, it can be used to show that for any matrix multiplication algorithm which is $\omega(n^2)$, matrix inversion can be done in time upper-bounded by matrix multiplication in the big-Oh sense [CLRS, AHU]. We refer to the following as Strassen's matrix inversion formula, as distinct from Strassen's matrix multiplication algorithm. (A more detailed exposition is found in [CLRS], using the same notation).

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

where $S = E - DB^{-1}C$, which is the Schur Complement of A with respect to B.

It is routine to check that the product of A and the matrix formula for A^{-1} yields the identity matrix, multiplying on the left or on the right. If an inverse for a matrix exists, it is unique, and so therefore this formula gives the unique inverse of A.

However, it is a clear requirement of this formula that B and S be invertible. One can show that if A and B are non-singular, then S is non-singular also [CLRS]. The problem is to guarantee that the upper-left submatrix, B, is invertible.

In contrast to the present situation, over the reals the possible noninvertibility of B is not a problem. It turns out that the upper-left submatrix of a symmetric positive-definite matrix is symmetric positive-definite. All symmetric positive-definite matrices are invertible, and so the above formula can be used. If A is not symmetric positive-definite, note that $A^T A$ will be. Furthermore,

$$A^{-1} = (A^T A)^{-1} A^T$$

Therefore any real or complex matrix can be inverted by calculating $A^T A$, applying Strassen's formula for matrix inversion, and multiplying by A^T on the right.

5.3 Inversion of Boolean Matrices

Unfortunately, the reduction to positive definite matrices does not work without modification over GF(2). Consider

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \qquad A^T A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

Both A and $A^T A$ have det = 1, thus are invertible. Yet in both cases the upper-left hand 2x2 submatrices have det = 0, and therefore are not invertible. Thus Strassen's formula for inversion is unusable without modification.

If the matrix is unit upper (or lower) triangular with all ones on the main diagonal, then it clearly has det = 1, and thus invertible. Likewise its upper-left hand submatrices (of any size) also are unit upper (or lower) triangular, and so also have det = 1 and are invertible. For this reason, Strassen's inversion formula can be used to invert matrices that are unit upper/lower triangular.

5.3.1 Triangular Inversion with Strassen's Algorithm

When A is a $2n \times 2n$ upper-triangular matrix (with ones on the main diagonal), the Schur Complement of A with respect to B (which was denoted S earlier) is merely E. Thus Strassen's formula becomes

$$A = \begin{bmatrix} B & C \\ 0 & E \end{bmatrix} \quad \Rightarrow \quad A^{-1} = \begin{bmatrix} B^{-1} & -B^{-1}CE^{-1} \\ 0 & E^{-1} \end{bmatrix}$$

Since A is unit upper triangular, so are B and E. Thus the cost is two calls to a triangular inversion algorithm, of size $n \times n$, and two calls to a multiplication algorithm, of size $n \times n$, plus $4n^2$ work to fill in the output matrix.

$$I_t(2n) = 2I_t(n) + 2M(n) + 4n^2$$

The subscripts here denote that these algorithms are for the triangular case. Since matrix inversion and matrix multiplication are the same complexity in the big-Theta sense [CLRS], $I_t(n) \sim kn^{(\log 7)}$, and $I_t(2n) \sim 2^{(\log 7)} I_t(n) \sim 7I_t(n)$, yielding,

$$\begin{aligned} 7I_t(n) &\sim 2I_t(n) + 2M(n) + 4n^2 \\ I_t(n) &\sim (2M(n) + 4n^2)/(5) \\ I_t(n) &\sim 0.9337n^{\log 7} \end{aligned}$$

with the substitution $M(n) = 2.3343n^{(\log 7)}$ calculated in Section 5.3.2, a general formula for the multiplication of large boolean matrices. One can also verify that our assumption of $I_t(2n) = 7I_t(n)$ is consistent. (If Strassen's Algorithm were used naïvely, this would be $I_t(n) \sim 4.8n^{\log 7}$, or more than five times slower).

5.3.2 LU-Factorization

The generalized algorithm for doing LU-Factorization using multiplication and inverting subroutines of smaller size is given in [AHU], originally from [BH74], but is not straightforward. The full details are specified in Appendix E, but we summarize as follows.

The algorithm can take any full rank underdefined $m \times n$ matrix A (thus $m < n$) for input. The goal is to output an $m \times m$ lower triangular matrix L, an $m \times n$ upper-triangular matrix U, and a $n \times n$ permutation matrix P, such that $A = LUP$. Also, both L and U will have only ones on the main diagonal and so are non-singular.

It does this by cutting A in half horizontally, into two $m/2 \times n$ pieces, repeatedly. When down to single $1 \times n$ rows, the LU-factorization is easy. The permutation matrix P is that which puts a non-zero element in the first cell of A. The upper-triangular matrix is $U = AP$, and the lower triangular matrix L is the 1×1 matrix [1]. Assembling the LU-factorizations of the larger pieces from those of the smaller takes many steps. The reader is encouraged to try a full-rank real-valued matrix with no zeroes using the full specification in Appendix E, because this algorithm works over any field. The absence of zeroes and full rank imply that the permutation matrices will always be the identity.

The computational cost of this factoring is given by

$$F(m, n) \sim 2.2876m^{\log 7} + 1.5mn$$

and for square matrices

$$F(n, n) \sim 2.2876n^{\log 7}$$

(See Appendix E for details).

5.3.3 Crossover Between LU-Factorization and Modified Method of 4 Russians

We have calculated that $\sim 2.2876n^{log7}$ is the number of matrix operations for Strassen's algorithm for LU factorization, while $\sim 0.5n^3$ and $\sim 2.5n^3/logn$ are the number required for Gaussian Elimination and the Modified Method of Four Russians for upper triangular form. Big Oh notation would make one believe that Strassen's algorithm is the best choice. However, we will show that this is misleading. Solving the following inequality for n,

$$2.2876n^{log7} < 2.5n^3/logn$$

one obtains that $n = 6.65 \times 10^6$ rows. Recall, this is the point where the two algorithms are of *equal* performance. The point at which Strassen is *twice* as fast as the Modified Method of Four Russians is $n = 1.02 \times 10^9$ rows. These matrices are have 4.4×10^{13} and 1.04×10^{18} entries respectively.

Meanwhile, we can calculate the crossover between Gaussian Elimination and the Modified Method of Four Russians, by solving

$$0.5n^3 > 2.5n^3/logn$$

The crossover here is at $n = 32$ rows, or 1024 elements. It is also interesting to see where the Modified Method of Four Russians is twice as fast as Gaussian Elimination, and that point is $n = 1024$, or 1.05×10^6 elements. Therefore the Modified Method of Four Russians should be used for the solution of systems of linear boolean equations, if $32 < n < 6.65 \times 10^6$.

5.3.4 Parallelism

It should be noted that Strassen's formula for matrix inversion lends itself particularly well to parallel computing environments. Suppose there are 2^d machines available. Then during LU-factorization each horizontal $n/(2^d) \times n$ strip of the matrix can be factored by one particular processor. This means that $d - 1$ levels of recursion would consist of transferring the task to new processors, and the work would be performed locally beginning at the dth level. The inverses and multiplications during the recombine step could be handled similarly, divided so that each processor has exactly one piece to work upon. The following inequality would be used to check if parallel execution of Strassen's formula is better than running the Modified Method of Four Russians on one machine:

$$2^{-d}6.4893n^{\log 7} \leq (3n^3)/(\log n)$$

For example, if a million by million square dense matrix is the result of the sparse stage, then the Modified Method of Four Russians requires 1.505×10^{17} steps. Using Strassen's formula requires $2^{-d}4.532 \times 10^{17}$. Thus with four machines available, Strassen's formula would take less absolute (clock) time, but even with two machines it would be better to leave one idle and have the other use the Modified Method of Four Russians. In the case of four machines, however, they could *each independently* solve four separate stream ciphers using the Modified Method of Four Russians in only slightly longer time (1.505×10^{17} versus 1.133×10^{17}). If a particular cipher needs to be solved *post haste* then massive parallelism could work (though careful analysis would need to be made as to how to handle the enormous communications overhead) but otherwise, to maximize the total amount of matrices inverted per unit time, parallelism is better utilized by each machine working on its own problem in isolation.

6 Conclusions

A new model for the computational cost of boolean matrix operations has been proposed, namely that of counting the number of matrix reads and writes, (to the input and output matrices but also including to large matrix-like temporary data structures). Instead of $O(f(n))$ or $\Theta(f(n))$ notation, $\sim f(n)$ notation has been used, permitting the study of coefficients. Particular coefficients for many algorithms have been calculated and appear in Table 1. For matrix multiplication, a strategy has been shown balancing the naïve algorithm, the Method of Four Russians and Strassen's Algorithm. This hybrid algorithm has a low coefficient. For solving a linear system of equations, the Modified Method of Four Russians has been shown to be dominant for all practical matrix sizes, despite big-Θ analysis to the contrary, when compared to Strassen's formula or dense Gaussian Elimination. Strassen's formula for matrix inversion can take advantage of parallelism, but this is only useful in lowering latency between submission of a problem and a solution, not the throughput of solutions solved per unit time.

7 Acknowledgments

The author would like to recognize first and foremost, Nicolas Courtois, whose work inspired this paper and whose ideas, so freely given in conversation, have been indispensable. We are deeply indebted to Patrick Studdard and Lawrence Washington for proofreading this paper. The author would also like to recognize Daniel Bernstein for his encouragement and suggestions.

References

[AHU] A. Aho, J. Hopcroft, and J. Ullman. "Chapter 6: Matrix Multiplication and Related Operations." *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[AGLL94] D. Atkins, M. Graff, A. Lenstra, P. Leyland. "The Magic Words are Squeamish Ossifrage." *Advances in Cryptology: ASIACRYPT'94* 1994.

[ADKF70] V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev. "On Economical Construction of the Transitive Closure of a Directed Graph." *Dokl. Akad. Nauk. SSSR* No. 194 (in Russian), English Translation in *Soviet Math Dokl.* No. 11, 1970.

[A02] F. Armknecht. "A Linearization Attack on the Bluetooth Key Stream Generator." IACR E-print, 2002, No. 191.

[A04] F. Armknecht. "Improving Fast Algebraic Attacks." Proceedings of Fast Software Encryption '04, Springer-Verlag *Lecture Notes in Computer Science*, 2004.

[AA05] F. Armknecht, and G. Ars. "Introducing a New Variant of Fast Algebraic Attacks and Minimizing Their Successive Data Complexity." Proceedings of Mycrypt '05, Springer-Verlag *Lecture Notes in Computer Science*, 2005.

[AK03] F. Armknecht and M. Krause. "Algebraic Attacks on Combiners with Memory." *Advances in Cryptology:* CRYPTO 2003.

[BH74] J. Bunch and J. Hopcroft. "Triangular Factorization and Inversion by Fast Matrix Multiplication." *Math Comp.* No. 28:125, 1974.

[C02] N. Courtois. "Higher Order Correlation Attacks, XL Algorithm and Cryptanalysis of Toyocrypt." Proceedings of ICISC '02, Springer-Verlag *Lecture Notes in Computer Science*, 2002.

[C03] N. Courtois. "Fast Algebraic Attacks on Stream Ciphers with Linear Feedback." *Advances in Cryptology:* CRYPTO 2003.

[C04] N. Courtois. "Algebraic Attacks on Combiners with Memory and Several Outputs." Proceedings of ICISC '04, Springer-Verlag *Lecture Notes in Computer Science*, 2004.

[C05] N. Courtois. "Cryptanalysis of Sfinks." Proceedings of ICISC '05, Springer-Verlag *Lecture Notes in Computer Science*, 2005.

[CLRS] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. "Chapter 28: Matrix Operations." *Introduction to Algorithms, Second Edition*. MIT Press, McGraw-Hill Book Company, 2001.

[CM03] N. Courtois, and W. Meier. "Algebraic Attacks on Stream Ciphers with Linear Feedback." *Advances in Cryptology:* EUROCRYPT 2003.

[CP02] N. Courtois, and J. Pieprzyk. "Cryptanalysis of Block Ciphers with Overdefined Systems of Equations." *Advances in Cryptology:* ASIACRYPT 2002.

[F70] M. Furman. "Application of a Method of Fast Multiplication of Matrices in the Problem of Finding the Transitive Closure of a Graph." *Dokl. Akad. Nauk. SSSR* No. 194 (in Russian), English Translation in *Soviet Math Dokl.* No. 3, 1970.

[HR04] P. Hawkes, and G. Rose. "Rewriting Variables: the Complexity of Fast Algebraic Attacks on Stream Ciphers." *Advances in Cryptology:* CRYPTO 2004.

[H] N. Higham. "Chapter 23: Fast Matrix Multiplication." *Accuracy and Stability of Numerical Algorithms, Second Edition.* SIAM, 2002.

[Ing] A. Ingham and R. Vaughan. The Distribution of Prime Numbers. Cambridge University Press, Reprint Edition, Cambridge, 1990. (Original is 1934).

[P] V. Pan. *How to Multiply Matrices Faster.* Springer-Verlag, 1984.

[PS92] C. Pomerance and J. Smith. "Reduction of Huge, Sparse Matrices over Finite Fields via Created Catastrophes." *Experimental Mathematics*. Vol. 1, No. 2. 1992.

[S81] A. Schönhage. "Partial and Total Matrix Multiplication." SIAM Journal of Computing, Vol 10, No. 3, August 1981.

[SPCK00] A. Shamir, J. Patarin, N. Courtois, and A. Klimov. "Efficient Algorithms for solving Overdefined Systems of Multivariate Polynomial Equations." *Advances in Cryptology:* CRYPTO 2004.

[S69] V. Strassen. "Gaussian Elimination is not Optimal." Numerische Mathematik, Vol 13, No 3. 1969.

[S87] V. Strassen. "Relative Bilinear Complexity and Matrix Multiplication." J. Reine Angew. Math. Vols 375 & 376, 1987.

[TB] L. Trefethen, and D. Bau. "Numerical Linear Algebra." SIAM. 1997.

[W] H. Wu. "Breaking the Stream Cipher Whitenoise." IACR E-print, 2003, No. 250.

[wiki] "Gaussian Elimination." From Wikipedia—The Free Encyclopedia.

A Dense Gaussian Elimination

A.1 Dense Gaussian Elimination to Reduced Row Echelon Form

The algorithm known as Gaussian Elimination is very familiar, and has been known since antiquity in China[5]. It has many variants, but two are useful to us. As a subroutine for calculating the inverse

[5]The important Chinese mathematical text *The Nine Chapters on the Mathematical Art*, written about 150 BC, known as *Jiuzhang Suanshu*, has the earliest known presentation of Gaussian Elimination [wiki].

of a matrix, we refer to adjoining an $n \times n$ matrix with the $n \times n$ identity matrix to form an $n \times 2n$ matrix. This will be processed to output the $n \times n$ identity on the left, and A^{-1} on the right. The second is to solve a system directly, in which case one column is adjoined with the constant values. The more useful variant, which finishes with a triangular rather than identity submatrix in the upper-left, is listed in Appendix A.2. (That variant requires 2/3 as much time when used for solving a system of equations, but is not useful for finding matrix inverses). Since Gaussian Elimination is probably known to the reader, it is not described here, but it has the following cost analysis.

- For each column $i = 1, 2, \ldots, \ell$
  - Search for a non-zero entry in region $a_{ii} \ldots a_{mn}$ (Expected cost is 2 reads). Call this entry a_{xy}.
  - Swap rows i and x, swap columns i and y. (Costs 4 writes).
  - For each row $j = 1, 2, \ldots, m$, but not row i
    - If $a_{ji} = 1$ (Costs 1 read) then for each column $k = i, i+1, \ldots, n$
      - Calculate $a_{jk} = a_{jk} + a_{ik}$. (Costs 2 reads, 1 write).

The total number of reads and writes is given by

$$\begin{aligned} &= \sum_{i=1}^{i=\ell} 6 + (m-1)(1 + 0.5(3)(n-i+1)) \\ &= 1.5nm\ell - 0.75m\ell^2 + 1.75m\ell - 1.5n\ell + 0.75\ell^2 + 4.25\ell \\ &\sim 1.5nm\ell - 0.75m\ell^2 \end{aligned}$$

Thus for the overdefined case ($\ell = n$) one obtains $0.75mn^2$, and for underdefined ($\ell = m$) the total is $1.5nm^2 - 0.75m^3$. For a square matrix this is $0.75n^3$.

A.2 Dense Gaussian Elimination to Upper Triangular Form

Below we list an alternative form of the Gaussian Elimination algorithm, which leaves in the last step an upper-triangular matrix rather than an identity matrix in the upper-left $\ell \times \ell$ submatrix. This is not useful for finding the inverse of a matrix, but for solving a system of m equations in n unknowns. Here it is assumed that one column is adjoined that contains the constants for a system of linear equations.

- For each column $i = 1, 2, \ldots, \ell$
  - Search for a non-zero entry in region $a_{ii} \ldots a_{mn}$ (Expected cost is 2 reads). Call this entry a_{xy}.
  - Swap rows i and x, swap columns i and y. (Costs 4 writes).
  - For each row $j = i+1, i+2, \ldots, m$
    - If $a_{ji} = 1$ then for each column $k = i, i+1, \ldots, n$
      - Calculate $a_{jk} = a_{jk} + a_{ik}$ (Costs 2 reads, and 1 write).

The total number of reads and writes is given by

$$\begin{aligned} &= \sum_{i=1}^{i=\ell} 6 + (m-i)(1 + 0.5(3)(n-i+1)) \\ &= \sum_{i=1}^{i=\ell} 6 + (m-i)(2.5 + 1.5 * n - 1.5 * i) \\ &= 1.5nm\ell - 0.75m\ell^2 - 0.75n\ell^2 + 0.5\ell^3 + 2.5m\ell - 1.25\ell^2 - 0.75m\ell - 0.75n\ell + 0.75\ell^2 + 5\ell \\ &\sim 1.5nm\ell - 0.75m\ell^2 - 0.75n\ell^2 + 0.5\ell^3 \end{aligned}$$

Thus for the overdefined case ($\ell = n$) one obtains $0.75mn^2 - 0.25n^3$, and for underdefined ($\ell = m$) the total is $0.75nm^2 - 0.25m^3$. For a square matrix this is $0.5n^3$.

B Matrix Inversion Algorithms

B.1 Reduced Row Echelon Form in the Modified Method of Four Russians

Like Gaussian Elimination, it is possible to change the Modified Method of Four Russians for inversion to output a reduced row echelon form matrix, with ones on the main diagonal. This is useful for finding A^{-1}.

- For $i = 0 \ldots \ell/k - 1$ do
  - Do Gaussian Elimination on rows $ik + 1$ to $ik + 3k$ to obtain a $k \times k$ identity matrix starting at $a_{(ki+1)(ki+1)}$.
  - For $j = 1 \ldots 2^k - 1$ do
    - Generate the $n - ik - k$ entries of this row of the Gray Code table by vector addition of the previous row, and the appropriate row of the matrix. Denote T_x the xth row of the table. (Costs 2 reads and 1 write per each of $n - ik - k$ entries).
  - For $j = 1$ to ik and also $j = ik + 3k + 1$ to m do
    - Read the k bit string at $a_{(ik+1),j} \ldots a_{(ik+k),j}$. (Costs k reads). Treat this string as a binary number x.
    - Set to zero the k entries at $a_{(ik+1),j} \ldots a_{(ik+k),j}$. (Costs k writes).
    - For $p = ik + k + 1 \ldots n$ do
      - Calculate $a_{jp} = a_{jp} + T_{x,(p-ik-k)}$ (Costs 2 reads/1 write).

Stages one and two are identical to the variant for unit upper triangular form. Stage three is identical except that it runs for more steps, namely $m-3k$ iterations instead of $pos(m-3k-ik)$ (Recall $pos(x) = x$ if x is positive, 0 otherwise). Assume $m > 3k$. Adjusting for these changes, we obtain

$$\begin{aligned}
&\sum_{i=0}^{i=\ell/k-1} 1.5(3k)^2(n-ik) - 0.75((3k)^3)(3 \cdot 2^k - 4)(n-ik-k) + (m-3k)(3n-3ik-k) \\
&\sim \frac{1}{4k}\left[2^k(12n\ell - 6\ell^2 - 6k\ell) + 12mn\ell - 6m\ell^2\right]
\end{aligned}$$

Substituting $k = \log \ell$ and thus $2^k = \ell$,

$$\frac{1}{4\log \ell}\left(12n\ell^2 - 6\ell^3 + 12mn\ell - 6m\ell^2\right)$$

Thus for the over-defined case ($\ell = n$) this is $(3n^3 + 3mn^2)/(2\log n)$, and for the under-defined case ($\ell = m$) this is $(6nm^2 - 3m^3)/(\log m)$, and for square $(3n^3)/(\log n)$.

B.2 Strassen's Algorithm for Matrix Inversion

After using LU-factorization, as described in Section E, one has $A = LUP$. Since $A^{-1} = P^{-1}U^{-1}L^{-1}$, and $P^{-1} = P^T$, then only two unit triangular matrix inversions and one matrix multiplication are required to invert A.

$$\begin{aligned}
I(n) &= F(n,n) + 2I_t(n) + M_s(n) \\
I(n) &= (2.2876 + 1.8674 + 2.3343)n^{\log 7} \\
I(n) &= 6.4893n^{\log 7}
\end{aligned}$$

C Strassen's Algorithm for Matrix Multiplication

To find:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

Use the following steps:

- Calculate 10 sums, namely: $s_1 = a_{12} - a_{22}$, $s_2 = a_{11} + a_{22}$, $s_3 = a_{11} - a_{21}$, $s_4 = a_{11} + a_{12}$, $s_5 = a_{21} + a_{22}$, $s_6 = b_{21} + b_{22}$, $s_7 = b_{11} + b_{22}$, $s_8 = b_{11} + b_{12}$, $s_9 = b_{12} - b_{22}$, and $s_{10} = b_{21} - b_{11}$.
- Calculate 7 products, namely: $m_1 = s_1 s_6$, $m_2 = s_2 s_7$, $m_3 = s_3 s_8$, $m_4 = s_4 b_{22}$, $m_5 = a_{11} s_9$, $m_6 = a_{22} s_{10}$, and $m_7 = s_5 b_{11}$.
- Calculate 8 sums, namely: $s_{11} = m_1 + m_2$, $s_{12} = -m_4 + m_6$, $s_{13} = -m_3 + m_2$, $s_{14} = -m_7 + m_5$, $c_{11} = s_{11} + s_{12}$, $c_{12} = m_4 + m_5$, $c_{21} = m_6 + m_7$, and $c_{22} = s_{13} + s_{14}$.

One can see that this consists of 18 matrix additions and 7 matrix multiplications,

$$M(2n) = 7M(n) + 54n^2$$

allowing one to calculate, for a large matrix,

$$\begin{aligned}
M(4n_0) &= 7^2 M(n_0) + (4+7) \cdot 54n_0^2 \\
M(8n_0) &= 7^3 M(n_0) + (16 + 7 \cdot 4 + 7^2) \cdot 54n_0^2 \\
M(16n_0) &= 7^4 M(n_0) + (64 + 16 \cdot 7 + 7^2 \cdot 4 + 7^3) \cdot 54n_0^2 \\
M(2^i n_0) &= 7^i M(n_0) + (4^{i-1} + 4^{i-2}7 + 4^{i-3}7^2 + \cdots + 4 \cdot 7^{i-2} + 7^{i-1}) 54n_0^2 \\
M(2^i n_0) &\approx 7^i M(n_0) + 7^{i-1}(1 + 4/7 + 16/49 + 64/343 + \cdots) 54n_0^2 \\
M(2^i n_0) &\approx 7^i M(n_0) + 7^i 18n_0^2 \\
M(n) &\approx \frac{7^{\log n}}{7^{\log n_0}} M(n_0) + \frac{7^{\log n}}{7^{\log n_0}} 18n_0^2 \\
M(n) &\approx n^{\log 7} \frac{M(n_0) + 18n_0^2}{n_0^{\log 7}} \\
M(n) &\sim n^{\log 7} \frac{M(n_0)}{n_0^{\log 7}}
\end{aligned}$$

C.1 Crossover between Method of Four Russians and Strassen's Algorithm

Since the Modified Method of Four Russians is $\Theta(n^3/\log n)$ and Strassen's algorithm is $\Theta(n^{\log 7})$ it is true that Strassen's Algorithm will be more efficient for large matrices. However, solving the inequality below will determine at what n will Strassen's Algorithm defeat the Modified Method of Four Russians (for inversion of a square matrix).

$$6.4893 n^{\log 7} \leq (3n^3)/(\log n)$$

The crossover point is roughly n=3,370,000,000. Since this is a dense matrix algorithm, that means the matrix would have 1.1357×10^{19} entries, or require (at one bit per byte) 11 Exabytes of memory. The number of instructions required to calculate the inversion is 3.628×10^{27} operations, or 115 million years for a teraflop machine. In the next Section we show that Dense Gaussian Elimination is not efficient for any non-trivially sized matrix. Therefore, the Modified Method of Four Russians is the winning strategy for matrix inversion.

D Back Substitution

We assume the algorithm is familiar and trivial to the reader, so we simply list the performance analysis.

First, we example the underdefined case, where $A\vec{x} = \vec{b}$, and A is upper triangular $m \times n$, $\vec{x}$ is $n \times 1$ and $\vec{b}$ is m by 1. Recall underdefined means $m < n$. The following algorithm sets the "extra" $n - m$ degrees of freedom of $\vec{x}$ to zero, and then solves for the m remaining values of $\vec{x}$.

- $x_m \leftarrow b_m$
- For i:=$m-1$ down to 1 do
  - temp $\leftarrow b_i$
  - For j:=m down to $i+1$ do
    - $temp \leftarrow a_{ij} x_j$

- $x_i \leftarrow temp$

- For i:=m+1 to n do
  - $x_i \leftarrow 0$

The the inner loop is two matrix reads per iteration, and runs $m-i$ iterations. Each run of the outer loop is then $1+(m-i)(2)+1$ or $2m-2i+2$. Since i runs from $m-1$ down to 1, the outer loop comes to $(2m+2)(m-1)-2(m^2-m)/2$, or more simply m^2+m-2. The later loop costs one write per iteration, or $n-m$ writes in total. Therefore the whole algorithm costs $m^2+n-1 \sim m^2+n$. For square matrices, this is $\sim n^2$.

In the overdefined case, $m > n$ but the last $m-n$ rows are all zero, since A is upper triangular. If the last $m-n$ entries of $\vec{b}$ are not all zero, then the system is inconsistent (it takes $m-n$ matrix reads to check this). If the last $m-n$ entries of $\vec{b}$ are zero, then it suffices to ignore the all-zero rows of A, and all zero tail of $\vec{b}$, and proceed as if the matrix were square. As noted above, this is $\sim n^2$ steps, for a total of $\sim n^2+m$.

E LU-Factorization

Given the reasonable running time for matrix inversion in the triangular case, it is desirable to provide an algorithm for LU-factorization, which reduces any matrix to a product of triangular and permutation matrices. The variant shown here is taken from Aho, Hopcroft, and Ullman's textbook [AHU] and always returns (L,U,P) such that $A=LUP$. This would then enable the calculation $A^{-1}=P^{-1}U^{-1}L^{-1}=P^TU^{-1}L^{-1}$.

Initially A is an $n \times n$ matrix. However, the algorithm functions by repeatedly cutting the matrix in half horizontally, and calling itself on the two $n/2 \times n$ pieces. Once these are factored, some steps are required to convert this to an LU-factorization of A. Therefore the algorithm is written to accept any $m \times n$ matrix as input, such that $m < n$ (underdefined). This implies $m/2 < n$ and $m/4 < n$ and so forth, and so the overdefined case cannot occur. The symbol $F(m,n)$ is the cost of this algorithm on an $m \times n$ matrix. The output matrices L, U, and P, are of size $m \times m$, $m \times n$, and $n \times n$ respectively. An excellent proof of correctness is found in [AHU].

Also, note that $M_s(n)$ is the cost of multiplying a square $n \times n$ matrix, $M_r(a,b,c)$ that of multiplying an $a \times b$ matrix by a $b \times c$ matrix, and $A_r(a,b)$ that of adding two $a \times b$ matrices. These are $2.3343n^{\log 7}$, and $(a,b,c)^{(\log 7)/3}$, and $3ab$ respectively.

- If $m \neq 1$
  - Denote B the rows $1,2,\ldots m/2$ of A. (No cost).
  - Denote C the rows $m/2+1, m/2+2, \ldots m$ of A. (No cost).
  - Call $(L_1,U_1,P_1) \leftarrow Factor(B)$ (Costs $F(m/2,n)$).
  - Let $D=CP_1^{-1}$. (Costs $4n$ reads/writes).
  - Denote E the columns $1,2,\ldots,m/2$ of U_1. (No cost).
  - Denote F the columns $1,2,\ldots,m/2$ of D. (No cost).
  - Let $F'=FE^{-1}$. (Costs $M_s(m/2)+I_t(m/2)$).
  - Let $G=D-F'U_1$. (Costs $M_r(m/2,m/2,n)+A_r(m/2,n)$).
  - Denote G' the columns $m/2+1, m/2+2, \ldots, n$ of G. (No cost).[6]
  - Call $(L_2,U_2,P_2) \leftarrow Factor(G')$ (Costs $F(m/2,n-m/2)$).
  - Extend P_2 to a $n \times n$ matrix by

$$P_3 = \begin{bmatrix} P_2 & 0 \\ 0 & I_{m/2} \end{bmatrix}$$

    (Costs $2(n-m/2)$ reads/writes, since this is nothing more than copying the swap list).
  - Let $H=U_1P_3^{-1}$. (Costs $4n$ reads/writes).
  - Let $P=P_3P_1$. (Costs $6n$ reads/writes).
  - Let

$$L = \begin{bmatrix} L_1 & 0 \\ F' & L_2 \end{bmatrix}$$

[6] The first $m/2$ columns of G are all zeroes.

(Costs $3(m/2)^2$ reads and $4m^2$ writes).

– Let

$$U = \begin{bmatrix} \leftarrow U \rightarrow \\ 0 \quad U_2 \end{bmatrix}$$

(Costs $3(m/2)^2$ reads and $4m^2$ writes).

– Return (L, U, P).

- else (when $m = 1$)
  - Find an entry $a_{1j} \neq 0$ in A. (Costs at most n reads).
  - Let L be the 1×1 matrix [1]. (Costs 1 write).
  - Let P be the identity matrix with columns 1 and j swapped. (Costs 2 writes).
  - Let $U = AP$ (Costs 2 writes).
  - Return (L, U, P).

Once can see that the recursive call requires

$$\begin{aligned} F(m,n) &= F(m/2,n) + 4n + M_s(m/2) + I_t(m/2) + M_r(m/2,m/2,n) + A_r(m/2,n) + \\ &\quad F(m/2,n-m/2) + 2(n-m/2) + 4n + 6n + 4.75m^2 + 4.75m^2 \\ F(m,n) &= F(m/2,n) + F(m/2,n-m/2) + \mu_1(m/2)^{\log 7} + \mu_2(m/2)^{\log 7} + \mu_1(m^2n/4)^{(\log 7)/3} \\ &\quad +1.5mn + 9.5m^2 + 24n + 2m \\ F(m,n) &\sim F(m/2,n) + F(m/2,n-m/2) + \mu_3(m/2)^{\log 7} + \mu_1(m^2n/4)^{(\log 7)/3} \end{aligned}$$

Now since an $m/2 \times n - m/2$ matrix is smaller than an $m/2 \times n$ matrix, one can say that $F(m/2, n - m/2) \leq F(m/2, n)$, or that $F(m/2, n) + F(m/2, n - m/2) < 2F(m/2, n)$. We now have

$$F(m,n) \sim 2F(m/2,n) + \mu_3(m/2)^{\log 7} + \mu_1(m^2n/4)^{(\log 7)/3}$$

Let the symbols μ_1 and μ_2 refer to the coefficients of square matrix multiplication and triangular matrix inversion, 2.3343 and 0.9337, respectively. Their sum is μ_3, or 3.268. The symbol $\omega = \log 2$ and $\gamma = \omega/3$

We begin with three recursive steps:

$$\begin{aligned} F(m,n) &\sim 2F(m/2,n) + \mu_3(m/2)^{\omega} + \mu_1(m^2n/4)^{\gamma} \\ F(2m_0,n_0) &\sim 2F(m_0,n_0) + m_0^{\omega}\mu_3 + \mu_1 m_0^{2\gamma} n_0^{\gamma} \\ F(4m_0,n_0) &\sim 4F(m_0,n_0) + 2m_0^{\omega}\mu_3 + (2m_0)^{\omega}\mu_3 + 2m_0^{2\gamma}n_0^{\gamma}\mu_3 + (2m_0)^{2\gamma}n_0^{\gamma}\mu_1 \\ &\sim 4F(m_0,n_0) + (2+2^{\omega})m_0^{\omega}\mu_3 + (2+2^{2\gamma})m_0^{2\gamma}n_0^{\gamma}\mu_1 \\ F(8m_0,n_0) &\sim 8F(m_0,n_0) + (2^2+2\cdot 2^{\omega})m_0^{\omega}\mu_3 + (4m_0)^{\omega}\mu_3 + (2^2+2\cdot 2^{2\gamma})m_0^{2\gamma}n_0^{\gamma}\mu_1 + (4m_0)^{2\gamma}n_0^{\gamma}\mu_1 \\ &\sim 8F(m_0,n_0) + (2^2+2\cdot 2^{\omega}+4^{\omega})m_0^{\omega}\mu_3 + (2^2+2\cdot 2^{2\gamma}+4^{2\gamma})m_0^{2\gamma}n_0^{\gamma}\mu_1 \end{aligned}$$

Which generalizes to

$$\begin{aligned} F(2^i m_0,n_0) &\sim 2^i F(m_0,n_0) + (2^{i-1} + 2^{i-2}2^{\omega} + \cdots + 2^2(2^{i-3})^{\omega} + 2(2^{i-2})^{\omega} + (2^{i-1})^{\omega})m_0^{\omega}\mu_3 \\ &\quad +(2^{i-1} + 2^{i-2}2^{2\gamma} + \cdots + 2^2(2^{i-3})^{2\gamma} + 2(2^{i-2})^{2\gamma} + (2^{i-1})^{2\gamma})m_0^{2\gamma}n_0^{\gamma}\mu_1 \end{aligned}$$

Which can become

$$\begin{aligned} F(2^i m_0,n_0) &\sim 2^i F(m_0,n_0) + (2^{\omega i-1})\left((\frac{2}{2^{\omega}})^i + (\frac{2}{2^{\omega}})^{i-1} + (\frac{2}{2^{\omega}})^{i-2} + (\frac{2}{2^{\omega}})^{i-3} + \cdots + 1\right)m_0^{\omega}\mu_3 \\ &\quad +(2^{2\gamma i-1})\left((\frac{2}{2^{2\gamma}})^i + (\frac{2}{2^{2\gamma}})^{i-1} + (\frac{2}{2^{2\gamma}})^{i-2} + (\frac{2}{2^{2\gamma}})^{i-3} + \cdots + 1\right)m_0^{2\gamma}n_0^{\gamma}\mu_1 \\ &\sim 2^i F(m_0,n_0) + \frac{2^{\omega i-1}}{1-2^{1-\omega}}m_0^{\omega}\mu_3 + \frac{2^{2\gamma i-1}}{1-2^{1-2\gamma}}m_0^{2\gamma}n_0^{\gamma}\mu_1 \end{aligned}$$

One extreme is to allow the algorithm to run until it has partitioned the $m \times n$ matrix into m strips that are $1 \times n$. In this case, $m_0 = 1$, $i = \log m$, and $2^i = m$. This gives us

$$F(m, n_0) \sim mF(1, n_0) + \frac{2^{\omega(\log m)-1}\mu_3}{1 - 2^{1-\omega}} + \frac{2^{2\gamma(\log m)-1}n_0^\gamma \mu_1}{1 - 2^{1-2\gamma}}$$

But note $2^{\omega \log m - 1} = 2^{(\log m^\omega)-1} = m^\omega/2$ and therefore

$$F(m, n_0) = mF(1, n_0) + \frac{\mu_3}{2(1 - 2^{1-\omega})}m^\omega + \frac{n_0^\gamma \mu_1}{2(1 - 2^{1-2\gamma})}m^{2\gamma}$$

But, $2\gamma < \omega$ since $\gamma = \omega/3$, so we can drop the last term.

$$F(m, n_0) \sim mF(1, n_0) + \mu_4 m^\omega$$

Where

$$\mu_4 = \frac{\mu_3}{2(1 - 2^{1-\omega})} \approx 2.2876$$

This only leaves $F(1, n_0)$. To perform the LU-factorization of a single row, the L is the 1×1 matrix consisting of a one, the U is the vector with a one swapped into the first location, and the P is a permutation matrix that does swap that one into the first location. This can be calculated by searching for a one (at most n reads), performing the swap (2 writes), and copying the vector into U, ($2n$ reads and writes). Thus $F(1, n_0) \leq 3n_0 + 2$, and

$$F(m, n) \sim 3mn + \mu_4 m^\omega = 3mn + 2.2876m^{2.807}$$

For square matrices,

$$F(n) \sim 3n^2 + \mu_4 n^\omega = 2.2876n^{2.807}$$

Sub-Optimality Since $\Theta(n^{2.807})$ is asymptotically less than $\Theta(n^3)$, one knows that this algorithm as written will *eventually* be faster than LU-factorization or triangularization with Gaussian Elimination or the Modified Method of Four Russians. However, for smaller sizes, surely simpler algorithms will be faster. Therefore, there is a crossover "horizon" of ordered pairs (m, n), on the one side of which this algorithm should be avoided and upon the other side of which it should be used.

That being the case, one should not go to $F(1, n)$, but to some $F(m', n)$ and no smaller, and then switch to the algorithms that are faster for smaller matrices. This will lower the coefficient of the algorithm. The author believes that this horizon will probably be found experimentally and not analytically, but is open to suggestions. Future work will report on such experiments. Nonetheless, the results here are a valid upper bound to the cost of LU factorization via Strassen's algorithm.