MATHEMATICS OF COMPUTATION
Volume 00, Number 0, Pages 000–000
S 0025-5718(XX)0000-0

NEW PRACTICAL STRASSEN-LIKE APPROXIMATE MATRIX-MULTIPLICATION ALGORITHMS FOUND VIA SOLVING A SYSTEM OF CUBIC EQUATIONS

GREGORY V. BARD

Abstract. We have encapsulated the concept of matrix multiplication into a system of cubic equations. We discuss the properties of this system and the symmetry group of the set of solutions. Using solutions of these equations, we propose new algorithms for matrix multiplication, with explicit inequalities that bound the error. Also we calculate exactly the "hidden coefficient" in the Big-Oh notation for this class of algorithms. Moreover, we present a four-stage process for solving this system of cubic equations numerically, which we believe is new.

Specifically, by approximately solving such a system of these equations with 594 unknowns and 729 equations, we have found an approximate algorithm for multiplying 3×3 matrices in 22 steps, as compared to 23 steps by Laderman, 24 steps by Hopcroft and Kerr, 25 steps by Gastinel, and 27 steps by the naïve algorithm. This gives rise to a matrix multiplication algorithm which runs in time $n^{2.814\cdots}$ as compared to $n^{2.854\cdots}$, $n^{2.893\cdots}$, $n^{2.930\cdots}$, and n^3 respectively.

However, those running times are all inferior to Strassen's Algorithm, the currently fastest practical known method, which runs in time $n^{2.807\cdots}$. We have attempted to also solve these cubic equations in an instantiation with 4096 equations and 2304 or 2256 unknowns, a solution of which would give rise to an algorithm for approximate matrix multiplication which is practical and asymptotically faster than Strassen's algorithm, by performing 4×4 matrix multiplication in 48 or 47 steps. Lastly, we have investigated an algorithm for performing 5×5 matrix multiplication in 99 steps, which would exceed the current record, but is less than the 91 required to beat Strassen's algorithm. We do not have solutions to the 4×4 or 5×5 problems at this time.

Furthermore, we have discovered that these equations were known to Richard Brent several years ago [Bre70], and accordingly we name them the Brent Equations. In addition, if these equations can be solved exactly, over any field, then it will produce an asymptotically-fast exact algorithm over that field and all its finite-degree algebraic extensions. We are unable to do that at this time.

1. Introduction

Strassen's algorithm [Str69] for matrix multiplication stems from a shortcut for 2×2 matrix multiplication that requires 7 sub-matrix multiplies instead of $2^3 = 8$

2000 *Mathematics Subject Classification.* Primary 15A24; Secondary 15-04, 68W40.

Key words and phrases. Matrix Multiplication, Complexity, Polynomial System of Equations, Strassen's Algorithm, Laderman's Algorithm.

Computations for this work were entirely performed on `sage.math.washington.edu`, provided by The National Science Foundation, Grant No. DMS-0555776, to William Stein and the SAGE Project. Furthermore, a Fordham University Faculty Research Grant was essential.

$$\sum_{k=1}^{k=s} \alpha_{wxk}\beta_{yzk}\gamma_{ijk} = \delta_{iw}\delta_{xy}\delta_{jz} \qquad \forall \begin{cases} 1 \le i \le c & 1 \le w \le a \\ 1 \le x \le b & 1 \le y \le b \\ 1 \le z \le c & 1 \le j \le c \end{cases}$$

TABLE 1. The Brent Equations for Rectangular Shortcuts

multiplies. Likewise, Laderman's algorithm [Lad76] for 3×3 matrix multiplication requires 23 multiplies instead of $3^3 = 27$. Each gives rise, via repeated recursive calls, to a general matrix multiplication algorithm faster than the naïve method. The reason Laderman's is somewhat less commonly discussed is because its exponent, $\frac{\log 23}{\log 3} = 2.854\cdots$ is strictly inferior to Strassen's, or $\frac{\log 7}{\log 2} = 2.807\cdots$ and thus only Strassen's algorithm is implemented in many computer algebra systems. See Appendix D where these running times are derived, including more rigorous forms without the Big-Θ notation, with the coefficient exactly computed.

It is worth mentioning that a method for any of the following operations which runs in time $\Theta(n^\omega)$ also gives rise to algorithms for the others in time $\Theta(n^\omega)$, namely: matrix multiplication, matrix squaring, triangular matrix inversion, matrix inversion, LUP-factorization, and determinant calculation [Bar07, App. C].

Here, we present a general purpose method of finding such shortcuts, as well as a shortcut found. It is for 3×3 matrices in 22 steps, which beats the current record for 3×3 established by [Lad76], but is of only academic interest as it is slower than Strassen.

1.1. **From Shortcut to Algorithm.** The extension from shortcut to algorithm is simple. If one knows a matrix multiplication shortcut for $a \times b$ by $b \times c$ matrices, one can multiply an $a' \times b'$ by $b' \times c'$ by dividing the left into ab pieces of size $a'/a \times b'/b$, and the right into bc pieces of size $b'/b \times c'/c$, and proceed by recursion. Indeed, as a, b, c will most likely be less than six, the "pieces" are quite large. For this reason, we only count multiplications (an otherwise near-cubic operation) and not additions and subtractions (a quadratic operation) in the complexity measurements. The complexity if $a = b = c$ is given by $n^{\frac{\log s}{\log a}}$, where s is the number of steps required, and n is the dimension of the original matrix. See Appendix D for details, including the coefficients and tracking both the quadratic operations as well as the near-cubic operations.

2. The General Form of a Shortcut

When searching for an algorithm by solving a system of equations, it is challenging to represent a bound on the number of steps as some sort of algebraic or mathematical property. For simplicity, I will discuss an $n \times n$ times $n \times n$ multiplication, but rectangular formulae can be found similarly by carefully managing indices in the summations, and are given in Table 1.

Assume A, B, C are $n \times n$ matrices and we wish to multiply them in only s steps. We will use the following model, justified in Appendix A.

Each of the s products $P_1, \ldots, P_s$ will have a left matrix L_i and right matrix R_i, and we write $L_i R_i = P_i$. The L_i are simply linear combinations of the entries of A and the R_i are simply linear combinations of the entries of B. Using the following

notation:

$$L_k = \sum_{\forall i,j} \alpha_{ijk} A_{ij}$$

and likewise

$$R_k = \sum_{\forall i,j} \beta_{ijk} B_{ij}$$

we have the most general linear combination possible, but at the cost of $2n^2 s$ unknowns.

Finally, the entries of the answer matrix will be linear combinations of these s product matrices. Namely,

$$C_{ij} = \sum_{k=1}^{s} \gamma_{ijk} P_k$$

yielding a total of $3n^2 s$ unknowns.

Note that the L_i, R_i, P_i, and the entries of A, B, C are all in the same "general-purpose" non-commutative ring. For example, if multiplying a 1000×1000 real matrix by another, with $n = 4$, then this ring is $M_{250}(\mathbb{R})$. Therefore, you can see that multiplications are far more costly than additions and subtractions, validating our earlier remark.

2.1. **The Constraint Satisfaction Problem Model.** The system of equations of this paper is an example of the constraint satisfaction problem model, where constraints to the solution of some problem are modeled as equalities and inequalities cutting up a space into a feasible and infeasible region. Any point in the feasible region is a solution. For this paper, we have equalities only.

We know matrix multiplication is a bilinear[1] map. Therefore, because the space of pairs of $n \times n$ matrices has a basis, if our algorithm is a bilinear map, then it suffices to prove that our algorithm is correct on each basis member. This is a powerful statement because (when $n = 4$ for example), there are finitely many basis elements ($4^4 = 256$ in this case) yet infinitely many matrices.

Let S_{ab} denote the matrix whose i, jth entry is given by $\delta_{ia}\delta_{bj}$. The n^2 matrices S_{ij} for $i, j \in \{1, 2, \ldots, n\}$ form a basis for the set of $n \times n$ matrices over any field F, as an n^2-dimensional F-vector space. This set of matrices taken in Cartesian Product with itself is a basis for all pairs of matrices, and thus we have n^4 basis elements to check the correctness of the algorithm.

Therefore, if $\phi(A, B)$ is the product of AB as computed by our algorithm, then

$$\phi(S_{ab}, S_{cd}) = S_{ab} S_{cd} = \delta_{bc} S_{ad} \qquad \forall a, b, c, d \in \{1, 2, \ldots, n\}$$

is sufficient for correctness, as it can be seen ϕ is bilinear.

This, however, is a matrix equation and we require field equations. We could instead state

$$entry_{ij}(\phi(S_{ab}, S_{cd})) = entry_{ij}(\delta_{bc} S_{ad}) = \delta_{bc}\delta_{ia}\delta_{jd} \qquad \forall a, b, c, d, i, j \in \{1, 2, \ldots, n\} \tag{1}$$

which is n^6 equations.

[1] A map $\phi : M \times M \to M$ where M is an R-module is said to be bilinear if $\phi(ax + by, z) = a\phi(x, z) + b\phi(y, z)$ and also $\phi(x, ay + bz) = a\phi(x, y) + b\phi(x, z)$ for all $x, y, z \in M$ and all $a, b \in R$. Examples include matrix multiplication, the dot product of vectors, cross products in $\mathbb{R}^3$, and the integral of the product of two functions.

Now all that remains is to find $entry_{ij}(\phi(S_{ab}, S_{cd}))$. If $A = S_{ab}$ then $L_k = \alpha_{abk}$ and likewise if $B = S_{cd}$ then $R_k = \beta_{cdk}$. Therefore $P_k = \alpha_{abk}\beta_{cdk}$. And finally,

$$entry_{ij}(\phi(S_{ab}, S_{cd})) = C_{ij} = \sum_{k=1}^{s} \gamma_{ijk} P_k = \sum_{k=1}^{s} \gamma_{ijk}\alpha_{abk}\beta_{cdk} \tag{2}$$

Substituting Equation 2 into Equation 1 we obtain

$$\sum_{k=1}^{s} \gamma_{ijk}\alpha_{abk}\beta_{cdk} = \delta_{bc}\delta_{ia}\delta_{jd} \qquad \forall a, b, c, d, i, j \in \{1, 2, \ldots, n\}$$

which is a system of n^6 cubic equations with $3n^2 s$ unknowns, that we name The Brent Equations. The rectangular form has been given in Table 1.

During the solving process, we prefer to consider the following n^6 functions, of which we are trying to find a common root.

$$f_{ijabcd}(\cdots) = -\delta_{bc}\delta_{ia}\delta_{jd} + \sum_{k=1}^{s} \gamma_{ijk}\alpha_{abk}\beta_{cdk}$$

Finally, note that the Brent Equations have particularly compact and simple first and second partial derivatives. This is crucial, as computing the Jacobian and Hessian is far faster than expected, compared to a dense system of the same number of equations and unknowns, and the method of solution used depends upon on those matrices.

Matrix Size	Steps	Equations	Unknowns	Complexity
3×3	22	729	594	$N^{2.814\cdots}$
4×4	48	4096	2304	$N^{2.792\cdots}$
5×5	99	15,625	7425	$N^{2.855\cdots}$

2.2. **Algebraic Extensions.** This now is all the information for an exact algorithm, if an exact solution were to be found. The author has no exact solution over any field so far, but such a solution would be valid not only for the field solved in, but also for any finite-degree algebraic extensions. For matrices over the same field as the exact solution, see the Algorithm in Table 2. The extension-field variation is described in Appendix C but just uses the classical idea of representing a degree-d algebraic extension of a field $\mathbb{F}$ as a set of $d \times d$ matrices whose entries are from $\mathbb{F}$.

2.3. **Approximate Solutions and Algorithms.** A "very good" approximate solution, however, has been found for several special cases to be discussed shortly. This produces an approximation algorithm for matrix multiplication. However, all matrix multiplication algorithms used in practice over $\mathbb{R}$ are indeed approximate, because there are a finite number of elementary particles in the universe, and therefore the full $\aleph_1$ view of the number line cannot be represented on a computer. Thus floating-point numbers are used, and we will discuss an error model in Section 4 and a modification to the algorithm, which reduces error, in Table 3.

3. Solving the Equations

The algorithm presented here is conglomeration of what is found in [Kel87], [Kel03], and [Avr03], but is different from what is found there because the number of equations and unknowns is not the same, and because we use norms other than the L-2 norm.

Recall for $n \times n$ matrix multiplication in s steps then the number of equations is n^6 and the number of unknowns is $3n^2 s$. If we wish to beat Strassen then

Input: A matrix A of size $a \times b$, and a matrix B of size $b \times c$.
Output: A matrix C of size $a \times c$.
Note: We assume $n|a, n|b, n|c$.

1) Partition matrix A into n^2 submatrices of size $a/n \times b/n$, named $A_{11}, \ldots, A_{nn}$.
2) Partition matrix B into n^2 submatrices of size $b/n \times c/n$, named $B_{11}, \ldots, B_{nn}$.
3) For $k = 1$ to s do begin
3.1) $L_k = \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_{ijk} A_{ij}$
3.2) $R_k = \sum_{i=1}^{n} \sum_{j=1}^{n} \beta_{ijk} B_{ij}$
3.3) $P_k = L_k \times R_k$
3.4) end
3.5) For $i = 1$ to n do begin
3.5.1) For $j = 1$ to n do begin
3.5.1.1) $C_{ij} = \sum_{k=1}^{s} \gamma_{ijk} P_k$
3.5.2) end
3.6) end
4) Reassemble $C_{11}, \ldots, C_{nn}$ into C.
5) Return C.

Given α_{ijk}, β_{ijk}, γ_{ijk}, for $i, j \in \{1, 2, \ldots, n\}$ and $k \in \{1, 2, \ldots, s\}$, exact solutions to the $3n^2 s$-variable cubic system with n^6 equations, the above algorithm for $a \times b$-by-$b \times c$ matrix multiplication is induced. When $a = b = c = m$, it runs in time $\Theta(m^\omega)$, where $\omega = \log s / \log n$.

TABLE 2. The Algorithm Induced by an Exact Solution

$s = n^{2.807-\epsilon}$, and so we have $3n^{4.807-\epsilon}$ unknowns. Thus even if $n = 3$ there are far more equations than unknowns.

3.1. **Multidimensional Newton's Method.** Newton's Method, given by

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

has a vector analog called Multidimensional Newton's Method (MDNM) [Avr03]. While Newton's Method tries to find a zero of some $f(x)$, the MDNM finds a common zero of n functions of an n-dimensional vector $\vec{x}$, for example $\vec{F}(\vec{x}) = (f_1(\vec{x}), \ldots, f_n(\vec{x}))$.

The MDNM formula is

$$\vec{x}_{i+1} = \vec{x}_i - J^{-1} \vec{F}(\vec{x}_i)$$

where J is the Jacobian. More precisely for our case,

$$J_{ij} = \frac{d}{dx_j} f_i(\vec{x})$$

But in our case, J is not square, and thus it has no inverse. One response might be to replace $J^{-1}\vec{F}(\vec{x})$ with $\vec{z}$ such that $J\vec{z} = \vec{F}(\vec{x})$. In practice, this system was often inconsistent, and so no such $\vec{z}$ existed. Using $J(J^T J)^{-1}$, the Moore-Penrose

Pseudo-Inverse [Moo20] [Pen55] of J, which is identical to solving $J\vec{z} = \vec{F}(\vec{x})$ in the least-squared sense, was also totally ineffective.

Many other methods were tried as well, until the following was found to work.

3.2. **Stage 1: Gradient Descent.** We return to our case of m equations, and n unknowns, with $m > n$. Define now

$$g_2(\vec{x}) = \sum_{i=1}^{m} f_i(\vec{x})^2$$

which is clearly zero if and only if all the $f_i(\vec{x})$ are zero. Searching for a zero of $g(\vec{x})$ by gradient descent (sometimes called "method of steepest descent") is extremely slow but certain to work with arbitrarily high precision given enough time. We used this as a pre-processing step to turn a randomly generated initial $\vec{x}$ into a "moderate" approximate solution.

An interesting point is that $g(\vec{x})$ is merely the L-2 norm of the $\vec{F}(\vec{x})$ mentioned earlier, but without the final square root. We found that substituting the L-4 norm instead of the L-2 norm produced much better results. Norms of higher value, such as L-6 and L-8 would be used after the L-4 gradient descent reached a relative improvement below 10^{-5} per iteration. If the p norm is used, and p is odd, then an absolute value is required, but for even p it is not. Note, we will always omit the pth root.

Thus because

$$g_p(\vec{x}) = \sum_{i=1}^{m} |f_i(\vec{x})|^p$$

we have, for odd[2] $p \geq 3$,

$$\frac{d}{dx_j} g_p(\vec{x}) = \sum_{i=1}^{m} p f_i(\vec{x})^{p-1} \mathrm{sgn}(f_i(\vec{x})) \frac{d}{dx_j} f_i(\vec{x})$$

and equivalently, for even p,

$$\frac{d}{dx_j} g_p(\vec{x}) = \sum_{i=1}^{m} p f_i(\vec{x})^{p-1} \frac{d}{dx_j} f_i(\vec{x})$$

and lastly, for $p = 1$,

$$\frac{d}{dx_j} g_1(\vec{x}) = \sum_{i=1}^{m} \mathrm{sgn}(f_i(\vec{x})) \frac{d}{dx_j} f_i(\vec{x})$$

where

$$\mathrm{sgn} f(x) = \begin{cases} 1 & x > 0 \\ d.n.e. & x = 0 \\ -1 & x < 0 \end{cases}$$

but note that the d.n.e. will occur with almost probability zero in a floating-point computation. Nonetheless, it is an annoyance.

[2]Note, since $p - 1$ is even we can remove the absolute values.

Faster Gradient Calculation. Computationally speaking, however, it is much faster instead to use

$$\frac{d}{dx_j} g_p(\vec{x}) = \sum_{i=1}^{m} a_i(\vec{x}) J_{ij}$$

the J_{ij} being the ijth entry of the Jacobian defined in Section 3.1 and the function $a_i(\vec{x})$ given by

$$a_i(\vec{x}) = \begin{cases} pf_i(\vec{x})^{p-1}\mathrm{sgn}(f_i(\vec{x})) & p \geq 3 \text{ and odd} \\ pf_i(\vec{x})^{p-1} & p \text{ is even} \\ \mathrm{sgn}(f_i(\vec{x})) & p = 1 \end{cases}$$

because then the values of $a_1, a_2, \ldots, a_m$ can be calculated once each, and not n times each, when finding the entire gradient vector.

3.3. **Stage 2: Fletcher-Reeves-Polak-Ribiere.** The Fletcher-Reeves [Fr64] and Polak-Ribiere [PR69] methods are nearly identical examples of non-linear conjugate gradient methods. Like gradient descent, at each iteration, a search direction is chosen and a step length must be found, and the point updated. Unlike gradient descent, where the search direction is mereley the current gradient, these methods have the search direction $\vec{s_t}$ in iteration t equal to

$$\vec{s_t} = \vec{g} + \beta \vec{s}_{t-1}$$

where $\vec{g}$ is the gradient. Thus $\beta = 0$ produces gradient descent.

Both Fletcher-Reeves and Polak-Ribiere give formulae for β, but these are chosen to ensure that $\vec{s_t} H \vec{s_t}_{-1} = 0$, where H is the Hessian matrix. Interestingly, this is done without actually computing the Hessian itself. Furthermore, these algorithms have the property that they are essentially as fast as gradient descent, but on the other hand, have the quadratic termination property: given a quadratic system of equations, they will converge exactly to the global minimum (given exact arithmetic operations), provided that H is positive-definite. Naturally, for a quadratic system of equations, H is constant. Further, if H is positive-definite, on can see that there is a unique global minimum, and no local minima. Furthemore, the numerical stability of the algorithms is quite good [Avr03].

3.4. **Stage 3: The Dancing-Norm Descent.** Norms higher than L-8 required more precision than the C langauge `long double` or "quadruple precision" could provide. Switching to MPFR [FHL+07], an arbitrary-precision floating-point library, we selected 400 bits of precision and would perform gradient descent with an L-p norm selected uniformly at random from $\{1, 2, \ldots, 9\}$, and changed whenever the relative improvement in the current norm between two adjacent operations falls below 10^{-6}.

Each norm represents a distinct surface in m-dimensional space, and a non-global-but-local-minimum or a saddle-point of one norm's surface might not exist on another norm's surface. On the other hand, the set of global minima are identical for all norms. And thus it is understandable that the altering of the norm is productive.

3.5. **Stage 4: A Local Quadratic Approximation.** For an n-valued scalar function f the Hessian H is a matrix such that $H_{ij} = d^2 f/(dx_i dx_j)$. More plainly, the Hessian is the Jacobian of the gradient. Obviously for a continuous f, the

Hessian is symmetric because of Clairaut's Theorem, noting that a polynomial will always have continuous partial derivatives existing at all points.

For even p we have

$$H_{jk}(\vec{x}) = \sum_{i=1}^{m} p f_i(\vec{x})^{p-2} \left[f_i(\vec{x}) \frac{d^2}{dx_j dx_k} f_i(\vec{x}) + (p-1) \frac{d}{dx_j} f_i(\vec{x}) \frac{d}{dx_k} f_i(\vec{x}) \right]$$

for odd $p \geq 3$ we have

$$H_{jk}(\vec{x}) = \sum_{i=1}^{m} p f_i(\vec{x})^{p-2} \mathrm{sgn}(f_i(\vec{x})) \left[f_i(\vec{x}) \frac{d^2}{dx_j dx_k} f_i(\vec{x}) + (p-1) \frac{d}{dx_j} f_i(\vec{x}) \frac{d}{dx_k} f_i(\vec{x}) \right]$$

and for $p = 1$ we have

$$H_{jk}(\vec{x}) = \sum_{i=1}^{m} \mathrm{sgn}(f_i(\vec{x})) \frac{d^2}{dx_j dx_k} f_i(\vec{x})$$

If an n-valued scalar function $q(\vec{x})$ has gradient $\vec{v}$ and Hessian H, but all third-order derivatives equal to zero, then it is given by

$$\frac{\vec{x}^T H \vec{x}}{2} + \vec{x} \circ \vec{v} + c = q(\vec{x})$$

where c is some real number. Near to some point $\vec{x}$, we can choose $c = g_p(\vec{x})$, and choose $\vec{v} = \bigtriangledown g_p(\vec{x})$, and likewise H to be the Hessian of g_p at $\vec{x}$. Just as (one-dimensional) Newton's Method approximates a function as a line, and MDNM as a plane, here we approximate locally as a quadric surface, $q(\vec{x})$.

We solve $H\vec{z} = \vec{v}$ for $\vec{z}$ and apply $\vec{x}_{i+1} = \vec{x} - \lambda \vec{z}$ for some real λ found via step-search. In doing so, we are actually using MDNM on the n equation and n unknown system of equations given by $\bigtriangledown q(\vec{x}) = 0$, which is certainly true at a local minimum.

3.6. **Step Search via Newton's Method and Armijo's Rule.** Furthermore, we found (one-dimensional) Newton's Method was an excellent step search algorithm after using Armijo's Rule [Kel03] to get global convergence. The function

$$g_p(\vec{x} + \vec{g}k)$$

where $\vec{x}$ is the current position and $\vec{g}$ is the gradient there was treated as a single-variable function of k the step-length. The initial guess was -1 for the first iteration, and in all later iterations, the final answer in the previous iteration.

4. Error Analysis

Let $Error(A, B) = \phi(A, B) - AB$, where $\phi(A, B)$ is the product according to our algorithm. Since matrix multiplication is a bilinear map and $\phi(A, B)$ is a bilinear map, then $Error$ is bilinear as well.

All $n \times n$ matrices can be written as linear combinations of the S_{ij}s as follows

$$A = \sum_{ij} A_{ij} S_{ij}$$

This means that

$$Error(A, B) = Error\left(\sum_{ab} A_{ab} S_{ab}, \sum_{cd} B_{cd} S_{cd} \right) = \sum_{abcd} A_{ab} B_{cd} Error(S_{ab}, S_{cd})$$

and in particular, taking the ijth entry

$$entry_{ij}(Error(A,B)) = \sum_{abcd} A_{ab}B_{cd}entry_{ij}(Error(S_{ab},S_{cd}))$$

Substituting $Error(S_{ab},S_{cd}) = \phi(S_{ab},S_{cd}) - S_{ab}S_{cd}$, and furthermore recalling from previous work that

$$entry_{ij}(S_{ab}S_{cd}) = \delta_{bc}\delta_{ia}\delta_{jd}$$

as well as the fact that

$$entry_{ij}(\phi(S_{ab},S_{cd})) = \sum_{k=1}^{s} \gamma_{ijk}\alpha_{abk}\beta_{cdk}$$

we obtain

$$entry_{ij}(Error(A,B)) = \sum_{abcd} A_{ab}B_{cd}\left[\left(\sum_{k=1}^{s}\gamma_{ijk}\alpha_{abk}\beta_{cdk}\right) - \delta_{bc}\delta_{ia}\delta_{jd}\right]$$

or more plainly,

$$entry_{ij}(Error(A,B)) = \sum_{abcd} A_{ab}B_{cd}f_{ijabcd}(\vec{x}) \tag{3}$$

Recall, in numerical analysis it frequently occurs that if an error term is precise enough, then it can be calculated and the answer adjusted by it, leaving the remaining error to be caused by secondary sources of error (e.g. Romberg Integration, [Rom55]). Here, that would require n^4 or 256 additional sub-matrix multiplies, or 303 total sub-matrix multiplies, yielding a recursive complexity of $n^{4.122}$, which is worse than the naïve method.

On the other hand, since the f_{ijabcd} are half-positive and half-negative, (in fact in the 3×3 in 26 steps case they had a mean of 2.1682×10^{-6} and a standard deviation of 1.3589×10^{-4}), if $A_{ab}B_{cd}$ can be made half-positive and half-negative, then many of the n^4-term sum will cancel out.

4.1. **Re-centering to Reduce Error.** We return to the general case of A being an $a\times b$ matrix and B being a $b\times c$ matrix, and our algorithm is to find $AB = C$, an $a\times c$ matrix.

Therefore, let μ_A be the median value of the entries of A and likewise μ_B. Furthermore define $\hat{A} = A - \mu_A U_{ab}$, and likewise $\hat{B} = B - \mu_B U_{bc}$, where U_{xy} is an $x\times y$ matrix of all ones. Observe,

$$AB = \left(\hat{A}+\mu_A U_{ab}\right)\left(\hat{B}+\mu_B U_{bc}\right) = \hat{A}\hat{B} + \mu_A U_{ab}\hat{B} + \mu_B U_{bc}\hat{A} + \mu_A\mu_B U_{ab}U_{bc}$$

which would produce the effect that half of the entries of $\hat{A}$ and of $\hat{B}$ would be positive or negative. But also note, this is not the same as saying that half of the $\hat{A}\hat{B}$ will be positive or negative, as the signs might be correlated.

Luckily, there are easy ways to calculate the last three products above. Surely $U_{ab}U_{bc} = bU_{ac}$. But also, to calculate $U_{ab}\hat{B}$, we need only find the first row of that product, and copy it repeatedly, which is a quadratic time operation. Likewise with $\hat{A}U_{bc}$, we need only find the first column, and copy repeatedly.

Nonetheless, these steps produce essentially no error, yet almost certainly destroy a great deal of it. Moreover, these extra steps have quadratic thus negligible complexity, and so they should be always taken when the field operations are not exact (i.e. floating-point). Thus we have the Algorithm given in Table 3.

Input: A matrix A of size $a \times b$, and a matrix B of size $b \times c$.
Output: A matrix C of size $a \times c$.

1) Let μ_A be the median entry of matrix A.
2) Let μ_B be the median entry of matrix B.
3) Let $\hat{A} = A - \mu_A U_{ab}$.
4) Let $\hat{B} = B - \mu_B U_{bc}$.
– Note U_{xy} is an $x \times y$ matrix of all ones.
5) Partition matrix $\hat{A}$ into $n \times n$ submatrices of size $a/n \times b/n$, named $\hat{A}_{11}, \ldots, \hat{A}_{nn}$.
6) Partition matrix $\hat{B}$ into $n \times n$ submatrices of size $b/n \times c/n$, named $\hat{B}_{11}, \ldots, \hat{B}_{nn}$.
7) For $k = 1$ to s do begin
7.1) $L_k = \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_{ijk} \hat{A}_{ij}$
7.2) $R_k = \sum_{i=1}^{n} \sum_{j=1}^{n} \beta_{ijk} \hat{B}_{ij}$
7.3) $P_k = L_k \times R_k$
7.4) end
7.5) For $i = 1$ to n do begin
7.5.1) For $j = 1$ to n do begin
7.5.1.1) $\hat{C}_{ij} = \sum_{k=1}^{s} \gamma_{ijk} P_k$
7.5.2) end
7.6) end
8) Reassemble $\hat{C}_{11}, \ldots, \hat{C}_{nn}$ into $\hat{C}$.
9) $C = \hat{C} + \mu_A U_{ab} \hat{B} + \mu_B \hat{A} U_{bc} + (\mu_A \mu_B b) U_{ac}$
– Note that AU_{bc} can be calculated by finding the first column of that product, and then repeating it, thus in quadratic time.
– Note that $U_{ab}B$ can be calculated by finding the first row of that product, and then repeating it, thus in quadratic time.
10) Return C.

Given α_{ijk}, β_{ijk}, γ_{ijk}, for $i, j \in \{1, 2, \ldots, n\}$ and $k \in \{1, 2, \ldots, s\}$, approximate solutions to the $3n^2 s$-variable cubic system with n^6 equations. The above approximate algorithm for $a \times b$-by-$b \times c$ matrix multiplication is induced. When $a = b = c = m$, it runs in time $O(m^\omega)$, where $\omega = \frac{\log s}{\log n}$.

TABLE 3. The Approximate Algorithm Induced by an Approximate Solution

4.2. **Bounding the Frobenius Error Norm.** Since the Frobenius Norm $||A||_f$ of a matrix A is merely the sum of the squares of the matrix entries A_{ij}, we can apply this to Equation 3 to obtain

$$||Error(A,B)||_f = \sum_{ij} entry_{ij}^2(Error(A,B)) = \sum_{ij} \left[\sum_{abcd} A_{ab} B_{cd} f_{ijabcd}(\vec{x}) \right]^2$$

By applying the Cauchy-Schwartz Inequality twice,

$$entry_{ij}^2(Error(A,B)) = \left[\sum_{abcd} A_{ab}B_{cd}f_{ijabcd}(\vec{x})\right]^2 \leq \left[\sum_{abcd} A_{ab}\right]^2 \left[\sum_{abcd} B_{cd}\right]^2 \left[\sum_{abcd} f_{ijabcd}\right]^2$$

which can be rewritten as

$$entry_{ij}^2(Error(A,B)) \leq \left(n^2||A||_f\right)\left(n^2||B||_f\right)\left(\sum_{abcd} f_{ijabcd}^2(\vec{x})\right)$$

and therefore

$$||Error(A,B)||_f = n^4||A||_f||B||_f\left(\sum_{ijabcd} f_{ijabcd}^2(\vec{x})\right)$$

and thus minimizing the L_2 norm of the residual vector is a way of provably minimizing the error of the algorithm.

5. Symmetry of Solutions

As it turns out, the symmetries of the algebraic variety formed by the intersection of the zero-sets of all these polynomials is quite large with a rich symmetry group. To see this, first note that for $n \times n$ multiplication in s steps, picking any $\sigma \in S_s$ will allow one to map one solution to another via

$$\alpha_{i,j,k} \mapsto \alpha_{i,j,\sigma(k)}, \qquad \beta_{i,j,k} \mapsto \beta_{i,j,\sigma(k)}, \qquad \gamma_{i,j,k} \mapsto \gamma_{i,j,\sigma(k)}$$

Clearly this maps any root of the Brent Equation

$$-\delta_{bc}\delta_{ia}\delta_{dj} + \sum_{k=1}^{k=s} \alpha_{ijk}\beta_{ijk}\gamma_{ijk} = f_{ijabcd}$$

to another root because it just reorders the large sum. More intuitively, we can see that renaming the s product matrices $P_1, P_2, \ldots, P_s$ does not really change the algorithm's functionality.

While this is a symmetry of size $s!$ for any coefficient field, there is a further symmetry that is infinite for infinite fields and, furthermore, continuous for complete (i.e. topologically closed) coefficient fields.

Let c_k and c'_k for $k \in \{1, 2, \ldots, s\}$ be non-zero elements of the coefficient field, and let $c''_k = 1/(c_k c'_k)$. Then the mapping

$$\alpha_{i,j,k} \mapsto c_k\alpha_{i,j,k}, \qquad \beta_{i,j,k} \mapsto c'_k\beta_{i,j,k}, \qquad \gamma_{i,j,k} \mapsto c''_k\gamma_{i,j,k}$$

will result in

$$-\delta_{bc}\delta_{ia}\delta_{dj} + \sum_{k=1}^{k=s} \alpha_{ijk}\beta_{ijk}\gamma_{ijk}c_k c'_k c''_k$$

which obviously will preserve roots.

It is surprising that the solution set has an infinite symmetry group, of dimension $2s$, (for any infinite field) because of the fact that the equations form an over-defined system, and therefore zero or a few solutions would be anticipated.

Even for a finite field, of size p^r, the second symmetry group has size $(p^r - 1)^2$. It is notable that any operation of the first group and any from the second will commute. Thus their direct product (as groups) is also a subgroup of the true

symmetry group of the solutions. Thus the true symmetry group of the set of solutions is of size divisible by $s!(p^r-1)^2$.

Naturally none of this matters if no solution exists. Still, it is of interest, because over finite fields the base-field solutions are spectacularly complex and so a Gröbner Basis that describes them would also have to be rather complicated, and thus hard to find.

6. Experimental Results

The experiments are mostly finished but converting the data into tables for LaTeX is a slow process, and will be done in early January.

7. Acknowledgements

Three pieces of software made this work possible. First, the NTL library (Number Theory Library) for large integer operations [Sho]. Second, built on the first, is the arbitrary-precision floating-point-arithmetic library MPFR [FHL+07]. Third, the author's own Gradient Descent/MDNM/Hessian Inverting code named "descender", which was converted to MPFR by Alexander Golec, then in the summer between his first and second year at Fordham University, paid for by a Fordham University Faculty Research Grant. We plan to release our software as an open-source tool for the general scientific community.

First and foremost, the author is indebted to the monographs [Kel87], [Kel03], and [Avr03], and their authors C T Kelley and M Avriel. In no particular order, the author would like to thank Nicolas T. Courtois for suggesting the problem; the author's thesis advisor, Lawrence C. Washington encouraged him to continue the project when he was considering aborting it; the SAGE community and William Stein in particular for algebraic hints; Dan Bernstein for suggesting the use of MPFR and the need for high-precision; the NSF for the machine `sage.math.washington.edu`, paid for by Grant No. DMS-0555776, and again William Stein for access to it; John Osborne of the University of Maryland for help with step search algorithms and multi-dimensional Newton's method; Justin Domke, of the University of Maryland Computer Science Department, for suggesting several algorithms; and Peter Wolfe, of the University of Maryland mathematics department for teaching an excellent course in numerical analysis, without which this work would not have been possible; and Volker Strassen, for both [Str69] and his later papers, which inspired this work.

References

[Avr03] Mordecai Avriel, *Nonlinear programming: Analysis and methods*, Dover Publications, 2003.

[Bar07] Gregory Bard, *Algorithms for the solution of linear and polynomial systems of equations over finite fields, with applications to cryptanalysis.*, Ph.D. thesis, Department of Applied Mathematics and Scientific Computation, University of Maryland at College Park, August 2007.

[Bar08] ———, *"extending sat-solvers to low-degree extension fields of gf(2)"*, In Preparation, Draft Available on Author's Web Page, 2008, `http://www.math.umd.edu/~bardg`.

[Bre70] Richard Brent, *Algorithms for matrix multiplication*, Tech. Report Report TR-CS-70-157, Department of Computer Science, Stanford, March 1970.

[FHL+07] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann, *Mpfr: A multiple-precision binary floating-point library with correct rounding*, ACM Trans. Math. Softw. **33** (2007), no. 2, 13.

[Fr64] R. Fletcher and C. M. reeves, *Function minimization by conjugate gradients*, Computer J. **7** (1964), 149–154.

[Kel87] C T Kelley, *Iterative methods for linear and nonlinear equations*, Society for Industrial and Applied Mathematics, 1987.

[Kel03] C. T. Kelley, *Fundamentals of algorithms: Solving nonlinear equations with newton's method*, Society for Industrial and Applied Mathematics, 2003.

[Lad76] J. Laderman, *A noncommutative algorithm for multiplying* 3×3 *matrices using 23 multiplications*, Bulletin of the American Mathematical Society **82** (1976), 126–128.

[Moo20] E. Moore, *On the reciprocal of the general algebraic matrix*, Bulletin of the American Mathematical Society **26** (1920).

[Pen55] R. Penrose, *A generalized inverse for matrices*, Proc. of the Cambridge Phil. Soc. **51** (1955).

[PR69] E. Polak and G. Ribiere, *Note sur la convergence de methodes de directions conjugees*, Rev. Fr. Inform. Rech. Oper. **16** (1969), 35–43.

[Rom55] W. Romberg, *Vereinfachte numerische integration*, Norske Videnskabers Selskab Forhandlinger (Trondheim) **28** (1955), no. 7, 30–36.

[Sho] Victor Shoup, *Ntl library*, Software Package.

[Str69] Volker Strassen, *Gaussian elimination is not optimal*, Numerische Mathematik **13** (1969), no. 3.

Appendix A. Justification of the Model

We wish to derive a shortcut for multiplication of two $n \times n$ matrices over any ring, using only s multiply operations, and then use this to multiply two matrices of dimension $N \times N$, with $n|N$.

Divide the matrices A and B into n^2 pieces each, so that each piece is a $N/n \times N/n$ matrix. Since there will be s matrix products, which will be $N/n \times N/n$ matrices as well, denote the result of these s matrix products $P_1, P_2, \ldots, P_s$.

Surely the ac pieces of the answer matrix C will be linear combinations of these products. The reason this is true is that we have spent our alloted s multiplications, and only additions, subtractions and scalar multiplications remain. We do not, however, know the coefficients of this linear combination, but we can write

$$C_{ij} = \sum_{k=1}^{k=n} \gamma_{ijk} P_k$$

where the γ_{ijk} are $n^2 s$ unknowns. What is less obvious, perhaps, is that one can claim that all those P's have the following form:

$$P_k = \left(\sum_{i=1}^{i=n} \sum_{j=1}^{j=n} A_{ij} \alpha_{ijk} \right) \times \left(\sum_{i=1}^{i=n} \sum_{j=1}^{j=n} B_{ij} \beta_{ijk} \right)$$

To see why this is true, consider the naïve formula for C_{ij}:

$$C_{ij} = \sum_{k=1}^{k=n} A_{ik} B_{kj}$$

There are no $B_{wx}B_{yz}$, nor $A_{wx}A_{yz}$, nor $B_{wx}A_{yz}$ terms, nor any terms with one, three or more matrices. Therefore, we know that in any valid formula for C_{ij}, there must only be terms of the form $A_{wx}B_{yz}$. By virtue of the fact that matrix multiplication is a bilinear map, clearly there can be no constants added or subtracted.

For P_1, then, it is clear that the product must be of some linear combination of the A_{ij} times some linear combination of the B_{ij}. On the other hand, for later P's, one might wonder if P_j can appear in the formula for P_i, with $j < i$. After all, the P's are matrices just like the A's and the B's.

If so, then of course P_i either can be written as a linear combination of A's and B's, or not. If not, then upon substituting the formula for P_j into P_i, and repeatedly to remove all P terms, then we will see that there must be terms of some type other than $A_{wx}B_{yz}$, because otherwise it could be written as such a linear combination of A's and B's. Yet we know that matrix terms of no other form can be found in the final formula. Therefore, it must be the case that the formula can be written as a linear combination of A's and B's.

Since we have now proven that P_i can be written as a linear combination of A's and B's, we can, without loss of generality, ignore the possibility of including P_j in the formula for P_i for $j < i$.

The question then arrises if this formulation for a potential algorithm is universal. Perhaps instead of 48 sub-matrix multiplies we use 47 and 1 sub-matrix inversion. This is unlikely because such an algorithm would fail if the particular sub-matrix were singular. Furthermore, if both input matrices were all zero, any linear combination of submatrices would remain all zero, and so there is no doubt that all those would be singular.

Nonetheless, we cannot quite take into account the possibility of some exotic operation, not normally part of the ring of square matrices. Such an exotic operation might allow for a shortcut that cannot be written this way. But in the absence of that, all shortcuts can be written in this way and so any certification that the system has no solutions is a certification that no shortcut exists with the particular specified number of steps and dimension.

Appendix B. Writing Strassen, Naïve, and Lademan this way

B.1. **The Naïve Algorithm.** The naïve algorithm uses 8 products to compute a 2×2 matrix multiplication. Recall this is really 8 products of the size $n/2 \times n/2$ to perform a $n \times n$ matrix multiplication.

$$\begin{aligned}
P_1 &= (A_{11}) \times (B_{11}) \\
P_2 &= (A_{11}) \times (B_{12}) \\
P_3 &= (A_{12}) \times (B_{21}) \\
P_4 &= (A_{12}) \times (B_{22}) \\
P_5 &= (A_{21}) \times (B_{11}) \\
P_6 &= (A_{21}) \times (B_{12}) \\
P_7 &= (A_{22}) \times (B_{21}) \\
P_8 &= (A_{22}) \times (B_{22})
\end{aligned}$$

$$\begin{aligned}
C_{11} &= P_1 + P_3 \\
C_{12} &= P_2 + P_4 \\
C_{21} &= P_5 + P_7 \\
C_{22} &= P_6 + P_8
\end{aligned}$$

B.2. **Strassen's Algorithm.** The naïve algorithm uses 7 products to compute a 2×2 matrix multiplication. Recall this is really 7 products of the size $n/2 \times n/2$ to perform a $n \times n$ matrix multiplication.

$$\begin{aligned}
P_1 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\
P_2 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\
P_3 &= (A_{11} - A_{21}) \times (B_{11} + B_{12}) \\
P_4 &= (A_{11} + A_{12}) \times (B_{22}) \\
P_5 &= (A_{11}) \times (B_{12} - B_{22}) \\
P_6 &= (A_{22}) \times (B_{21} - B_{11}) \\
P_7 &= (A_{21} + A_{22}) \times (B_{11})
\end{aligned}$$

$$\begin{aligned}
C_{11} &= P_1 + P_2 - P_4 + P_6 \\
C_{12} &= P_4 + P_5 \\
C_{21} &= P_6 + P_7 \\
C_{22} &= P_2 - P_3 + P_5 - P_7
\end{aligned}$$

B.3. **Laderman's Algorithm.** The naïve algorithm uses 23 products to compute a 3×3 matrix multiplication. Recall this is really 23 products of the size $n/3 \times n/3$ to perform a $n \times n$ matrix multiplication.

$$\begin{aligned}
P_1 &= (A_{11} + A_{12} + A_{13} - A_{21} - A_{22} - A_{32} - A_{33}) \times (B_{22}) \\
P_2 &= (A_{11} - A_{21}) \times (B_{12} - B_{22}) \\
P_3 &= (A_{22}) \times (\cdots \\
&\vdots \quad \vdots \quad \vdots \\
P_{22} &= (A_{33}) \times (B_{21}) \\
P_{23} &= (A_{33}) \times (B_{33}) \\
C_{11} &= P_6 + P_{14} + P_{19} \\
C_{12} &= P_2 + P_3 + P_4 + P_6 + P_{14} + P_{16} + P_{17} \\
&\vdots \quad \vdots \quad \vdots \\
C_{33} &= P_6 + P_7 + P_8 + P_9 + P_{23}
\end{aligned}$$

[I will arrange for an undergraduate to do the rest of the typing.]

Appendix C. Finite-Degree Algebraic Extensions

The reason for this is that a degree-d extension over a field $\mathbb{F}$ can be thought of as a set of $d \times d$-matrices over the field $\mathbb{F}$, and there is an isomorphism of fields between these representations. The field is generated by the powers of a matrix whose characteristic polynomial is the minimal polynomial of the field extension's primitive element.

Just as an example, $\mathbb{C}$ over $\mathbb{R}$, given by a root of x^2+1, can be written as

$$a+bi \mapsto a \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + b \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

or since $\mathbb{GF}(4)$, the finite field of eight elements, is given by an α which is a root of x^2+x+1 over $\mathbb{GF}(2)$, then

$$a+b\alpha \mapsto a \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + b \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

See [Bar08] for more examples over $\mathbb{GF}(2)$. So in short, an $n \times n$ matrix over a d-dimensional algebraic extension over F, a field in which The Brent Equations have been solved, yields a $nd \times nd$ matrix over F, which can be multiplied, inverted, squared, etc..., and transformed back. The transformation is quadratic time, and so $\Theta((nd)^\omega)$ operations for the matrix multiply is the rate-determining step, and for any particular fixed value of d this is $\Theta(n^\omega)$ time. Recall $\omega = \log s / \log n$ if a shortcut for $n \times n$ matrices in s steps can be found.

Appendix D. Complexity Calculations

The existence of a shortcut for $n \times n$ matrix multiplication using s multiplications, gives rise for an algorithm for general $N \times N$ matrix multiplication in $\Theta(N^{\log_s n})$ time, a fact which we will now prove. This proof appears in nearly every textbook which covers Strassen's Algorithm, but here we treat the secondary term rigorously, and keep the coefficients, giving a sharper result.

Suppose we can multiply two $n \times n$ matrices with s sub-matrix multiplications, and t sub-matrix additions, subtractions, and scalar multiplications. Then two $N \times N$ matrices, with $N \gg n$ can be multiplied via cutting it into $n \times n$ pieces, or with two recursive calls, into $n^2 \times n^2$ pieces, and so forth, until some threshold, namely $m_0 \times m_0$ is reached. The time required to multiply a $N \times N$ matrix is denoted $M(N)$. Note that scalar multiplications, additions, and subtractions of $N \times N$ matrices require N^2 field operations with the naïve methods for those operations, which are optimal.

Therefore, we have

$$\begin{aligned} M(N) &= sM(N/n) + t(N/n)^2 \\ &= s^2 M(N/n^2) + st(N/n^2)^2 + t(N/n)^2 \\ &= s^3 M(N/n^3) + s^2 t(N/n^3)^2 + st(N/n^2)^2 + t(N/n)^2 \\ &= s^r M(N/n^r) + \sum_{i=0}^{i=r} \frac{tN^2 s^i}{n^{2i+2}} \\ &= s^r M(N/n^r) + \frac{tN^2}{n^2} \left(\frac{1 - \left(s/n^2\right)^{r+1}}{1 - s/n^2} \right) \end{aligned}$$

and we can substitute

$$r = \log_n N/m_0 \text{ or more simply } n^r = N/m_0 \text{ thus } N/n^r = m_0$$

and obtain

$$M(N) = s^{\log_n N/m_0} M(m_0) + \frac{tN^2}{n^2} \left(\frac{1 - \left(s/n^2\right)^{1+\log_n N/m_0}}{1 - s/n^2} \right)$$

but applying the identity

$$s^{log_n N/m_0} = (N/m_0)^{\log_n s}$$

and likewise

$$\left(\frac{s}{n^2}\right)^{\log_n N/m_0} = \left(\frac{N}{m_0}\right)^{\log_n s/n^2} = \left(\frac{N}{m_0}\right)^{-2+\log_n s}$$

we see the clearer form

$$M(N) = \left(\frac{N}{m_0}\right)^{\log_n s} M(m_0) + \frac{tN^2}{n^2}\left(\frac{1-(s/n^2)\,(N/m_0)^{-2+\log_n s}}{1-s/n^2}\right)$$

or equivalently

$$M(N) = \left(\frac{N}{m_0}\right)^{\log_n s} M(m_0) - \frac{tN^2}{s-n^2} + \left(\frac{N}{m_0}\right)^{\log_n s}\left(\frac{tm_0^2 s}{n^2(s-n^2)}\right)$$

and finally

$$M(N) = \left(\frac{N}{m_0}\right)^{\log_n s}\left(M(m_0) + m_0^2\frac{ts}{n^2(s-n^2)}\right) - \frac{tN^2}{s-n^2}$$

the second term being quadratic and therefore negligible. The $M(m_0)$ would be $2m_0^3$ if naïve matrix multiplication were being used, otherwise it must be calculated.

D.1. **Explicit Cross-Over.** In our case, each L_i requires n^2 scalar multiplications, and n^2-1 additions, or $2n^2-1$ field operations. The R_i cost the same, and there are s of each of these. Each of the n^2 of the C_{ij} will cost s scalar multiplications, and $s-1$ additions. Thus we have

$$t = (n^2)(2s-1) + (2s)(2n^2-1) = 6sn^2 - n^2 - 2s$$

Just as examples, in our 3×3 in 22 steps, we see that $t = 1135$, $n = 3$, and $s = 22$, the number of field operations is

$$M(N) = \left(\frac{N}{m_0}\right)^{\log_3 22}\left(M(n_0) + \frac{24970}{117}m_0^2\right) - \frac{1135}{13}N^2$$

and if the base case of $m_0 \times m_0$ multiplication is done naïvely then the optimal $m_0 = 466$. From this, the cross-over with naïve matrix multiplication can be found to be

$$M(N) > 2N^3 \Leftrightarrow N > 1154$$

Naturally, this method ignores the effects of caching, and so is not realistic, but in practice, Strassen's Algorithm out-performs the naïve-method *earlier* than expected due to caching, thus we might expect ours to do so as well.

Department of Mathematics, Fordham University, The Bronx, NY, 10458, USA
E-mail address: bard@fordham.edu
URL: http://www.math.umd.edu/~bardg/