Fast Matrix Operations in Computer Algebra

Zak Tonks & Gregory Sankaran & James H. Davenport
Departments of Mathematical Sciences and Computer Science
University of Bath, Bath BA2 7AY, United Kingdom
Email: {Z.P.Tonks,G.K.Sankaran,J.H.Davenport}@bath.ac.uk

***Abstract*—There has been surprisingly little written about the practical use of Strassen–Winograd (as opposed to interpolation-based, and therefore oriented towards matrices of dense polynomials) fast matrix methods in computer algebra. We show that Strassen–Winograd multiplication can be practically effective. We also derive a fraction-free method of fast matrix inversion, and investigate its efficiency.**

***Keywords*-matrices; fast matrix; multiplication; inversion;**

I. Introduction

Throughout this paper, we consider operations on $n \times n$ matrices over some commutative ring R, elements of $R^{n\times n}$, where we generally assume that n is a power of 2. When we say a matrix of size n, we mean that it is square, of dimension $n \times n$. F will be the field of fractions of R. By "elementary operations" we mean operations on elements of R. Since the initial appearance in [1] of methods for multiplying matrices in fewer than $O(n^3)$ elementary operations, there has been an enormous literature on what are loosely called "fast operations". The method in [1] was $O(n^{c_S})$ where $c_S = \log_2(7) \approx 2.807$, for multiplication and for matrix inversion. Roughly speaking, we can divide this literature into four classes:

1) developments in the area of inversion, e.g. [2];
2) improved implicit constants in the O notation, e.g. [3];
3) methods that are $O(n^\omega)$ for $\omega < c_S$, e.g. [4];
4) investigations into the practical behaviour of these algorithms, e.g. [5].

This paper falls largely in the fourth category. In particular we consider the case, common in computer algebra, where R is an integral domain (or at least a domain where enough elements are non zero divisors) rather than a field, and where the costs of operations grow as the elements grow. Conversely we assume exact arithmetic, and are not concerned with rounding errors or numerical stability.

II. Matrix Multiplication

Rather than [1], we consider [3], which from our point of view is strictly superior. We can summarise it as:

- $8A_1$ perform eight additions of the input matrix elements, or the results of previous additions;
- $7M$ perform seven multiplications of these;
- $7A_2$ perform seven additions of these products, or the results of previous additions.

By contrast, the standard algorithm's cost is $8M + 4A_2$.

A. The base case

If we regard $M = A_1 = A_2$ (the usual assumptions), then for 256×256 matrices, the elementary method requires $16777216M + 16711680A$ and Winograd's method $5764801M + 28496325A$.

In practice, of course, we do not recurse down all the way, and Table I shows the results of using the Naïve method for 2×2, 4×4 and 8×8 matrices. We see that 8×8 suffices, as this method is already faster for 16×16 matrices.

Table I
$M = A_1 = A_2$

Size	Naïve	Winograd	Wfrom2x2	Wfrom4x4	Wfrom8x8
2	12	22	12	12	12
4	112	214	144	112	112
8	960	1738	1248	1024	960
16	7936	13126	9696	8128	7680
32	64512	95722	71712	60736	57600
64	520192	685414	517344	440512	418560
128	4177920	4859338	3682848	3145024	2991360
256	33488896	34261126	26025696	22260928	21185280
512	268173312	240810922	183162912	156809536	149280000

Table of weighted number of operations for each algorithm against matrix dimension, where here we take $M = A_1 = A_2$. Note, "Wfromnxn" *means Winograd recursing as usual until matrix size n, then naïve.*

B. Linear Polynomials

Now let us suppose that the entries of M are univariate linear polynomials, so that an A_1 costs two coefficient operations, a multiplication costs $M = \frac{5}{2}A_1$, and $A_2 = \frac{3}{2}A_1$. Table II shows the effect of these data.

Table II
Linear Polynomials: $M = \frac{5}{2}A_1$,$A_2 = \frac{3}{2}A_1$

Size	Naïve	Winograd	Wfrom2x2	Wfrom4x4	Wfrom8x8
2	26	36	26	26	26
4	232	326	256	232	232
8	1952	2578	2088	1920	1952
16	16000	19230	15800	14624	14848
32	129536	139346	115336	107104	108672
64	1042432	994366	826296	768672	779648
128	8364032	7036338	5859848	5456480	5533312
256	67010560	49557470	41322040	38498464	39036288
512	536477696	348114706	290466696	270701664	274466432

Table of weighted number of operations for each algorithm against matrix dimension, where here we take linear polynomials.

As the degree increases, the disparities in costs will increase.

C. Experimental Data

Table III
THEORY WITH $M = 3.8A_1, A_2 = 2A_1$

Size	Naïve	Winograd	Wfrom2x2	Wfrom4x4	Wfrom8x8
2	38	49	38	38	38
4	339	428	357	339	339
8	2842	3349	2850	2726	2842
16	23245	24854	21355	20493	21299
32	188006	179609	155118	149082	154726
64	1512243	1279788	1108357	1066099	1105613
128	12130714	9048629	7848610	7552806	7829402
256	97176781	63700854	55300715	53230093	55166259
512	777938534	447347769	388546798	374052442	387605606

Table of weighted number of operations for each algorithm per matrix dimension, where here we take $M = 3.8A_1$, $A_2 = 2A_1$, such as for the polynomials used in experiments that produced the results in Table IV.

Table IV
RESULTS OF EXPERIMENTATION

Size	Naïve	Winograd	Wfrom2x2	Wfrom4x4	Wfrom8x8
8	0.094	0.172	0.047	0.032	0.046
16	0.391	1.34	0.5	0.391	0.359
32	3.02	9.13	3.36	2.80	2.64
64	23.6	64.6	24.8	19.7	20.2
128	204	463	185	150	158
256	1690	3500	1360	1100	1110
512	14300	25600	10000	8310	8900

CPU time (in seconds) to three significant figures for Naïve and Winograd variants per matrix dimension. Such matrices had entries that were polynomials in two variables, of degree 7 with 5 terms. Thus we might expect the timings of these multiplication algorithms on matrices of these polynomials to imitate the shape of Table III.

All the following experimentation was performed in Maple 2016[1]. Table IV shows the results of experimentation in the form of a table showing the CPU time taken to complete the relevant variants of these multiplication algorithms on two matrices of size n. Such matrices were over (randomly generated) polynomials of degree 7, of 5 terms, in two variables. These sparse polynomials obtained an experimental "MA ratio" of ≈ 3.8, i.e. here we have that a multiplication of two of these polynomials takes $\approx 3.8\times$ as much work as an addition of the same. It can be seen that Winograd recursing down to 4×4 matrices largely pairwise outperforms the other variants, and in fact this data for the most part reflects what can be seen in Table III, bar some anomalies where Winograd recursing to 8×8 outperforms the rest.

Note that Table III and IV together indeed suggest that in this case we should recurse down to 4×4 matrices.

III. INVERSION

[1] also explains how to invert a matrix. This assumes that the matrix is over a field, and that operations have unit cost. This is a much more dubious assumption in computer algebra, as if M is a matrix whose elements are polynomials of degree d, then the elements of M^{-1} are, generically, quotients of polynomials of degree $(n-1)d$ and nd. We in fact use the description in [2], which has the advantage of being applicable to solution of linear equations directly.

A. Schur Complements

We consider a block matrix $A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}$, and assume $A_{1,1}$ is invertible (see section V). Then

$$\begin{pmatrix} I & 0 \\ -A_{2,1}A_{1,1}^{-1} & I \end{pmatrix} \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} = \begin{pmatrix} A_{1,1} & A_{1,2} \\ 0 & \Delta \end{pmatrix}$$

where $\Delta = A_{2,2} - A_{2,1} \cdot A_{1,1}^{-1} \cdot A_{1,2}$ is known as the Schur Complement (See [6] for the history) of $A_{1,1}$ in A. In particular

$$\mathrm{Det}(A) = \mathrm{Det}(A_{1,1})\mathrm{Det}(\Delta) \tag{1}$$

since the premultiplier has determinant 1.

B. Bunch–Hopcroft Algorithm

This inversion algorithm (Algorithm 1) recurses on itself, inverting submatrices of half the size in any one iteration. Namely, the submatrices $A_{1,1}$ and $\Delta = A_{2,2} - A_{2,1} \cdot A_{1,1}^{-1} \cdot A_{1,2}$ are to be inverted, and we assume these are invertible (i.e. $\mathrm{Det}(A_{1,1}), \mathrm{Det}(\Delta) \neq 0$) for every iteration. Note that, in general, Δ will already be a matrix of fractions, because it involves $A_{1,1}^{-1}$.

Algorithm 1 Bunch Hopcroft Inversion. Inverts a matrix A over a field F. The matrices $A_{1,1}$ and Δ required to be non singular at every iteration.

Input: $A \in F^{n\times n}$
Output: $B \in F^{n\times n}$ such that $A \cdot B = B \cdot A = I$
begin algorithm BHInversion(A)
if $n = 1$ **then**
  $B \leftarrow \frac{1}{A_{1,1}}$
else
  $A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}$
  $A_{1,1}^{-1} \leftarrow$ BHInversion($A_{1,1}$)
  $\Delta \leftarrow A_{2,2} - A_{2,1} \cdot A_{1,1}^{-1} \cdot A_{1,2}$
  $\Delta^{-1} \leftarrow$ BHInversion(Δ)
  $\lambda \leftarrow \Delta^{-1} \cdot A_{2,1} \cdot A_{1,1}^{-1}$
  $\epsilon \leftarrow A_{1,1}^{-1} \cdot A_{1,2}$
  $B \leftarrow \begin{pmatrix} A_{1,1}^{-1} + \epsilon \cdot \lambda & -\epsilon \cdot \Delta^{-1} \\ -\lambda & \Delta^{-1} \end{pmatrix}$
end if
end algorithm
return B

The correctness of Algorithm 1 can be verified by direct computation of $B \cdot A$.

[1]See https://doi.org/10.15125/BATH-00460 for implementations of all algorithms & testing procedures.

C. Towards a fraction-free version

The first thing to observe is that, although M^{-1} is a matrix of fractions, these are not random fractions. Rather, $\left(M^{-1}\right)_{i,j} = (\mathrm{Adj}(M))_{i,j}/\mathrm{Det}(M)$, where the adjugate $\mathrm{Adj}(M)$ is defined by $(\mathrm{Adj}(M))_{i,j}$ being the determinant of M with row i and column j deleted (and hence is a polynomial if the entries of M are polynomials).

Hence, rather than work with elements of $F^{n\times n}$, we will consider abstractly the localisation $(R^{n\times n})_{R^\dagger}$ where $R^\dagger$ is the non zero-divisors of R, or more concretely equivalence classes of pairs $(M,d) \in R^{n\times n}\times R^\dagger$ under the equivalence relation for fractions: $(M_1,d_1)\sim(M_2,d_2)$ precisely when $d_2M_1 - d_1M_2 = 0$. For convenience we call this structure $R_\dagger^{n\times n}$. We have the usual rules

$$(M_1,d_1)\cdot(M_2,d_2) = (M_1M_2, d_1d_2) \tag{2}$$

$$(M_1,d_1)+(M_2,d_2) = (d_2M_1 + d_1M_2, d_1d_2), \tag{3}$$

except that an important optimisation is that, if we know a common factor c of d_1, d_2, the second can be replaced by

$$(M_1,d_1)+(M_2,d_2) = (\frac{d_2}{c}M_1 + \frac{d_1}{c}M_2, \frac{d_1}{c}d_2). \tag{4}$$

In addition, $(M,d)^{-1} = (d\cdot\mathrm{Adj}(M), \mathrm{Det}(M))$. Henceforth, when we say "fraction–free inverse" of M, we mean the pair $(\mathrm{Adj}(M), \mathrm{Det}(M))$, and likewise "fraction–free inversion" means the method that produces this pair from M.

D. A fraction-free algorithm

The fraction-free form of Algorithm 1 is given in Algorithm 2. A slight subtlety is that we have written $B_{1,1}$, which in Algorithm 1 is $A_{1,1}^{-1}+\epsilon\cdot\lambda$, rather as $(A_{1,1}^{adj}, a_{1,1}^{adj})\cdot\left[(I^{\frac{n}{2}\times\frac{n}{2}},1)+(A_{1,2},1)\cdot(\Lambda,\lambda)\right]$ to recognise the common factor of $A_{1,1}^{-1}$, and to avoid (3) generating unnecessarily large denominators. In practice we would use (4), knowing that $a_{1,1}^{adj}$ was a common factor.

Theorem 1: Algorithm 2 is correct, and all the computations in it are fraction-free.

Since Algorithm 2 is a translation of Algorithm 1, which is correct, the correctness follows (and can also be verified by direct manipulation).

The statement about the fraction-free nature is more subtle: there are four divisions $\frac{d}{b_{1,1}}$ etc. occurring in Algorithm 2. However, we are *not* asserting that these divisions, in isolation, are exact, and indeed sometimes they are not. Rather, we are asserting that the entries of the matrix $B_{1,1} = \frac{d}{b_{1,1}}B'_{1,1}$ etc. are free of fractions, i.e. lie in R. This statement is now obvious: since $A\cdot B = d_1\cdot d_2\cdot I_{n\times n}$, the elements of B are d_1 times the elements of $\mathrm{Adj}(A)$, and therefore lie in R.

Algorithm 2 Fraction free formulation of Bunch-Hopcroft Inversion of a matrix A over a ring R. The matrices $A_{1,1}$ and Δ are required to be non singular at every iteration.

Input: $(A,d_1)\in R^{n\times n}\times R^\dagger$

Output: $(B,d_2)\in R^{n\times n}\times R^\dagger$ such that $A\cdot B = B\cdot A = d_1d_2I$

begin algorithm FFBHInversion(A)

if $n=1$ **then**

$B\leftarrow(d_1)$, $d_2\leftarrow A_{1,1}$

else

$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2}\end{pmatrix}$

$(A_{1,1}^{adj}, a_{1,1}^{adj}) \leftarrow$ FFBHInversion$(A_{1,1},1)$

$(\Delta,\delta)\leftarrow(A_{2,2},1)+(-A_{2,1},1)\cdot(A_{1,1}^{adj},a_{1,1}^{adj})\cdot(A_{1,2},1)$

$(\Delta^{adj},\delta^{adj})\leftarrow$ FFBHInversion(Δ,δ)

$(\Lambda,\lambda)\leftarrow(\Delta^{adj},\delta_{adj})\cdot(A_{2,1},1)\cdot(A_{1,1}^{adj},a_{1,1}^{adj})$

$(B'_{1,1},b_{1,1})\leftarrow(A_{1,1}^{adj},a_{1,1}^{adj})\cdot\left[(I^{\frac{n}{2}\times\frac{n}{2}},1)+(A_{1,2},1)\cdot(\Lambda,\lambda)\right]$

$(B'_{1,2},b_{1,2})\leftarrow(-A_{1,1}^{adj},a_{1,1}^{adj})\cdot(A_{1,2},1)\cdot(\Delta^{adj},\delta^{adj})$

$d_2\leftarrow\mathrm{Det}(A)$

$d\leftarrow d_1d_2$

$B_{1,1}\leftarrow\frac{d}{b_{1,1}}B'_{1,1}$

$B_{1,2}\leftarrow\frac{d}{b_{1,2}}B'_{1,2}$

$B_{2,1}\leftarrow\frac{-d}{\lambda}\Lambda$

$B_{2,2}\leftarrow\frac{d}{\delta^{adj}}\Delta^{adj}$

$B\leftarrow\begin{pmatrix} B_{1,1} & B_{1,2}\\ B_{2,1} & B_{2,2}\end{pmatrix}$

end if

E. Experimental Data

Fig. 1 shows the results of experimentation on Algorithm 2 using Winograd and naïve for matrix multiplication respectively, and an implementation of inversion via Gaussian Elimination using Dodgson-Bareiss (see [7]) for exact cancellations. Dodgson-Bareiss also lends a way of calculating the determinant of the matrix in the process, thus giving an implementation that gives an equivalent output to Algorithm 2. Where Winograd's multiplication was used, recursion on Winograd until matrix size 4 was used, as this was established as sufficient in Section II.

The matrices inverted in these experiments were over integers for purposes of testing. We acknowledge that it would be preferred to do these experiments over matrices of sparse multivariate polynomials as per Section II.C, and that there are approaches to inverting matrices of integers modulo small primes. Despite this, we present these results to demonstrate that Algorithm 2 works on matrices of

integers, and outperforms a fraction-free approach using Gaussian Elimination in this case. The intention, of course, is to use the algorithm on matrices of sparse multivariate polynomials, using Winograd's fewer multiplications to achieve less cost. However in particular the authors must point to one shortcoming of the current formulation, that lends itself to a potential improvement. For a matrix M of size n with entries being polynomials of degree d, we have that $Adj(M)$ is a matrix of size n with entries of polynomials of degree $d(n-1)$. In particular, we must draw attention to the fact that the matrix $\Delta = a_{1,1}^{adj} A_{2,2} - A_{2,1} A_{1,1}^{adj} A_{1,2}$ is a matrix of size $\frac{n}{2}$ with entries that are polynomials of degree $d(\frac{n}{2}+1)$ due to matrix multiplication, and we need to compute the fraction-free inverse of this! Clearly the cost of the problem is more than expected in this case. This of course also raises memory concerns. Integers suffer less from this bloat, and as such experiments to produce Fig. 1 completed reliably.

We see that Algorithm 2 with Winograd outperforms the other algorithms used here, even if marginally. The marginal performance improvement of Algorithm 2 with Winograd, and indeed its lower performance at matrix size 16 can be attributed to the lower "MA ratio" of integers compared to polynomials. As such expected number of operations of multiplications of matrices of integers done within 2 should be similar to those shown in Table I. In Table I we see that Winograd recursing to size 4 outperforms Naïve at size 16 - but Algorithm 2 on matrices of size 16 features multiplications of matrices of size 8, and hence we can explain Algorithm 2 with Naïve's performance at matrix size 16.

IV. A further improvement

The (Δ, δ) in Algorithm 2 is related to the true Δ of Algorithm 1, henceforth called Δ_t, in that every row has been multiplied by $\delta = a_{1,1}$. Hence $\mathrm{Det}(\Delta) = \left(a_{1,1}^{adj}\right)^{n/2} \mathrm{Det}(\Delta_t)$. But $\mathrm{Det}(A) = \mathrm{Det}(A_{1,1})\mathrm{Det}(\Delta)$ from (1), and $\mathrm{Det}(A_{1,1}) = a_{1,1}^{adj}$ from Theorem 1. So $\mathrm{Det}(\Delta) = \left(a_{1,1}^{adj}\right)^{n/2-1} \mathrm{Det}(A)$. $\mathrm{Det}(\Delta)$ crops up as δ^{adj}.

Hence the line $d_2 \leftarrow \mathrm{Det}(A)$ can be replaced by $d_2 \leftarrow \delta^{adj} / \left(a_{1,1}^{adj}\right)^{n/2-1}$, and we can afford[2] to cancel a common factor of $\left(a_{1,1}^{adj}\right)^{n/2-1}$ from $(\Delta^{adj}, \delta^{adj})$.

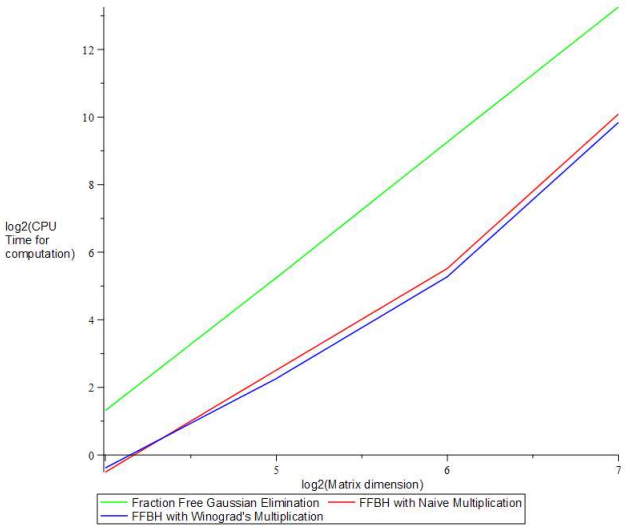
V. Conclusion

Despite initial appearances, it *is* possible to rewrite the algorithm from [2] to be fraction-free. There are some non-trivial improvements that we have yet to evaluate.

All this assumes that the relevant matrices $A_{1,1}$ and Δ are non-singular. Bunch and Hopcroft explain how to handle this in the case when R is an integral domain, and we propose no changes [2]. If R is not an integral domain then the process would need to be reformulated.

[2]Further developments on this improvement to be found at https://researchportal.bath.ac.uk/en/publications/on-fast-matrix-inversion.

Figure 1. Experimental Data for Fraction Free Inversion



A log–log plot of CPU time taken to compute the fraction free inverse of a matrix A over the integers against matrix dimension. The methods used include FFBH (Algorithm 2) with Winograd, FFBH with elementary multiplication, and a fraction free implementation of Gaussian Elimination.

Acknowledgment

We are grateful for support by the Bath Institute for Mathematical Innovation and the H2020-FETOPEN-2016-2017-CSA project SC^2 (712689).

References

[1] V. Strassen, "Gaussian Elimination is not Optimal," *Numer. Math.*, vol. 13, pp. 354–356, 1969.

[2] J. Bunch and J. Hopcroft, "Triangular factorization and inversion by fast matrix multiplication," *Mathematics of Computation*, vol. 28, pp. 231–236, 1974.

[3] S. Winograd, "On multiplication of 2 x 2 matrices," *Linear algebra and its applications*, vol. 4, pp. 381–388, 1971.

[4] F. Le Gall, "Powers of Tensors and Fast Matrix Multiplication," in *Proceedings ISSAC 2014*, 2014, pp. 296–303.

[5] P. D'Alberto, M. Bodrato, and A. Nicolau, "Exploiting parallelism in matrix-computation kernels for symmetric multiprocessor systems: Matrix-multiplication and matrix-addition algorithm optimizations by software pipelining and threads allocation," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 2:1–2:30, 2011.

[6] R. Cottle, "Manifestations of the Schur Complement," *Linear Algebra and its Applications*, vol. 8, pp. 189–211, 1974.

[7] E. Bareiss, "Sylvester's Identity and Multistep Integer-Preserving Gaussian Elimination," *Mathematics of Computation*, vol. 22, pp. 565–578, 1968.