ALGORITHMS FOR MATRIX MULTIPLICATION

BY

R. P. BRENT

STAN-CS-70-157
MARCH 1970

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

ALGORITHMS FOR MATRIX MULTIPLICATION

BY

R. P. Brent

March 1970

The preparation of this manuscript was supported in part by the Office of Naval Research (NR 044 211), the National Science Foundation (GJ 798), and the Atomic Energy Commission (Stanford PA #18).

Contents

1. Introduction

If $A = (a_{ij})$ is an m x n matrix, and $B = (b_{jk})$ is an n x p matrix, then the matrix product $C = A.B$ is the m x p matrix (c_{ik}) defined by

$$c_{ik} = \sum_{j=1}^{n} a_{ij}.b_{jk} \tag{1.01}$$

for $1 \leq i \leq m$, $1 \leq k \leq p$.

Matrix multiplication and its special cases occur very frequently in numerical analysis. For example: the inner-product of two vectors (the case m = p = 1), matrix times vector multiplication (the case p = 1), back substitution when solving linear systems, iterative refinement (perhaps with several right hand sides at once), the power method for eigenvalues, in least squares problems, and many more. Hence, it is interesting to investigate algorithms for matrix multiplication, and in particular to see in what circumstances it is possible to do better than the straightforward implementation of the definition (1.01).

It is clear that advantage may often be taken of special properties of A, B or C, e.g. sparseness or symmetry, if such properties are known a priori. We shall only consider the general case where no such helpful properties are known. For practical applications, we need only consider matrices over the rational, real and complex fields, although the definition above makes sense for matrices over any ring. The algorithms described will all be applicable to the problem of multiplication of matrices over an arbitrary commutative ring, and it will later be important that, for some of the algorithms, the ring need not even be commutative.

If the algorithms are to be implemented on a digital computer, then simply counting arithmetic operations can be rather misleading, for loads, stores and address computations are also important. The best test is to implement the algorithms and see how fast they actually run, and even then the conclusion may depend on the programmer, compiler and machine used. Also, from a practical point of view, storage requirements and roundoff errors may be vitally important. Hence, after describing several different algorithms in Sec. 2, I shall discuss their numerical properties in Sec. 3, and describe some experimental results in Sec. 4. In Sections 5 and 6 an attempt to find some new algorithms is described, and in Sec. 7 the results are summarized and some conclusions drawn. The notation of the definition (1.01) will be used in Secs. 2 to 4.

2. Known Results

2.1 The Normal Method

To evaluate the inner-product in the definition (1.01) takes n multiplications and n - 1 additions. Hence, the m.p elements c_{ik} can be found in mnp multiplications and m(n - 1)p additions, and about the same number of loads, stores and address computations.

If we count only multiplications then this straightforward method is known to be optimal in some important special cases. If m = p = 1 then we have the case of a vector inner-product, and a simple dimensionality argument shows that, in general, n multiplications are necessary. If p = 1 then we have the case of matrix times vector multiplication, and mn multiplications are necessary in general (Winograd, see [1]). In the general case, however, less than mnp multiplications are necessary: Strassen's , method shows this even when m = n = p = 2. Dimensionality arguments give the lower bound max (mn, np, pm), but usually this is too low, and the best possible result is not known. For more details, see Secs.5 and 6.

2.2 Winograd's Method

Winograd [7] has given a method based on the following identity:

$$\sum_{j=1}^{2\lfloor n/2\rfloor} a_{ij}b_{jk} = \sum_{j=1}^{\lfloor n/2\rfloor} (a_{i,2j-1} + b_{2j,k})(a_{i,2j} + b_{2j-1,k}) - \sum_{j=1}^{\lfloor n/2\rfloor} a_{i,2j-1}a_{i,2j} - \sum_{j=1}^{\lfloor n/2\rfloor} b_{2j-1,k}b_{2j,k} \tag{2.21}$$

Here $\lfloor x\rfloor$ means the greatest integer $y < x$, and analogously $\lceil x\rceil$ means the least integer $y > x$.

If n is even, the left side of (2.21) is just c_{ik}, but if n is odd, the term $a_{in}b_{nk}$ must be added to give c_{ik}. The point of Winograd's method is that the last two sums in (2.21) can be precomputed and, once this has been done, roughly half the usual number of multiplications are required to compute each c_{ik} using (2.21).

Supposing for simplicity that n is even, let us calculate the number of multiplications and additions involved in the computation of C by Winograd's method. We shall never distinguish between additions and subtractions. To compute

$$x_i = \sum_{j=1}^{n/2} a_{i,2j-1}a_{i,2j} \tag{2.22}$$

requires n/2 multiplications and (n/2 - 1) additions, and similarly for

$$y_k = \sum_{j=1}^{n/2} b_{2j-1,k}b_{2j,k} . \tag{2.23}$$

Hence, to precompute $x_1, x_2, \ldots, x_m$ and $y_1, y_2, \ldots, y_p$ takes (m + p)n/2 multiplications and (m + p)(n/2 - 1) additions*

Given x_i and y_k, to compute c_{ik} using (2.21) takes n/2 multiplications and (3n/2 + 1) additions. Thus the computation of the entire matrix product C takes (mp + m + p)n/2 multiplications and (3mp + m + p)n/2 + mp - m - p additions. From Sec. 2.1, we have saved (mp - m - p)n/2 multiplications at the expense of (mp + m + p)n/2 + 2mp - m - p additions, in comparison with the normal method.

Since mp - m - p = (m - 1)(p - 1) - 1, there is no gain at a.11 if m = 1 or p = 1, so the remarks above on the minimal number of multiplications required for matrix times vector multiplication are not contradicted.

Supposing for simplicity that $m = n = p > 1$, Winograd's method saves $(n - 2)n^2/2$ multiplications, at the expense of $(n^2 + 6n - 4)n/2$ additions. Hence, there is a. saving in the number of multiplications if $n > 4$ (recall that we assumed that n was even, but it may easily be verified that there is no saving for $n = 1$ or 3). If n is large then about $n^3/2$ multiplications have been traded for additions. If a multiplication takes w times as long as an addition, we see that

$$\frac{\text{Winograd time}}{\text{Normal time}} = \frac{w + 3}{2(w+1)} + O(n^{-1}), \tag{2.24}$$

so the most we can expect is a gain of nearly 50% if w and n are large. Since (2.24) neglects loads, stores etc. the gain will probably be rather less than this. Typically we might have $w = 2$ (say real multiplication) or $w = 4$ (say complex multiplication), giving savings of up to 17% and 30% respectively. In Sec. 4 we shall discuss how large n has to be for any gain in practice, and the important question of roundoff error will be discussed in Sec. 3.

2.3 Strassen's Method

Suppose there is an algorithm for the multiplication of $n_0 \times n_0$ matrices, for a certain fixed $n_0 > 1$, taking M multiplications and A additions. Suppose further that this algorithm is applicable for matrices over an arbitrary ring. In particular, we are not allowed to assume the commutative law for multiplication, so, for example, Winograd's method is excluded.

Let $v(k)$ and $w(k)$ be the number of multiplications and additions, respectively, required to multiply $n_0^k \times n_0^k$ matrices, for $k = 0, 1, 2 \ldots$.

We have
$$\left.\begin{aligned} &v(0) = 1, \quad w(0) = 0, \\ &v(1) \leq M, \quad w(1) < A. \end{aligned}\right\} \quad (2.31)$$

Now consider $n_0^{k+1} \times n_0^{k+1}$ matrices partitioned into n_0^2 blocks, each block an $n_0^k \times n_0^k$ matrix. Our matrices may be regarded as $n_0 \times n_0$ matrices with elements in the (noncommutative) ring of $n_0^k \times n_0^k$ matrices, so our algorithm is applicable. Applying it will take M multiplications, and A additions, of $n_0^k \times n_0^k$ matrices.

Hence
$$v(k+1) \leq M.v(k)$$
and
$$w(k+1) \leq M.w(k) + A.n_0^{2k}. \quad \left.\right\} \quad (2.32)$$

From (2.31) and (2.32) it follows by induction on k that

$$\left.\begin{aligned} &v(k) \leq M^k \\ \text{and} \quad &w(k) \leq \frac{A}{(M-n_0^2)}(M^k - n_0^{2k}) \end{aligned}\right\} \quad (2.33)$$

for any $k \geq 0$ (provided that $M \neq n_0^2$, but $M \leq n_0^2$ is impossible for $n_0 > 1$ anyway).

Now, in order to multiply n x n matrices for any n > 1, just take $k = \lceil \log_{n_0} n \rceil$ and embed the n x n matrices in $n_0^k \times n_0^k$ matrices with the last $n_0^k - n$ rows and columns zero, and use the above method. From (2.33), the number of arithmetic operations required is

$$O(M^{\log_{n_0} n}) = O(n^{\log_{n_0} M}) \quad \text{as } n \to \infty . \tag{2.34}$$

For example, the normal method with any $n_0 > 1$ has $M = n_0^3$, $\log_{n_0} M = 3$, giving $O(n^3)$ operations, which is no surprise.

From (2.34), square matrix multiplication can be done in $O(n^\beta)$ operations, where $\beta = \log_{n_0} M = (\log M)/(\log n_0)$. (It is interesting to note that β is independent of A.) Clearly there is a constant

$$\beta_0 = \inf \{ \beta \mid O(n^\beta) \text{ operations suffice} \} . \tag{2.35}$$

The normal method, and Winograd's method, both show that $\beta_0 \leq 3$, while the results discussed in Sec. 2.1 show that $\beta_0 \geq 2$. The actual value of β_0 is not known. While it might be considered "intuitively obvious" that $\beta_0 = 3$, this is false: as Strassen [5] has shown,

$$\beta_0 < \log_2 7 \doteq 2.8 . \tag{2.36}$$

Strassen's idea is to give an algorithm for the multiplication of 2 x 2 matrices over an arbitrary ring, with the algorithm involving 7 multiplications (instead of the usual 8) and 18 additions (instead of the usual 4). Putting $n_0 = 2$, $M = 7$ and $A = 18$ in the above, his result follows.

Strassen's algorithm is based on the following identities:

$$\text{if } \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$\left.\begin{aligned} \text{then} \quad c_{11} &= q_1 - q_3 - q_5 + q_7 , \\ c_{12} &= q_4 - q_1 , \\ c_{21} &= q_2 + q_3 , \\ \text{and} \quad c_{22} &= -q_2 - q_4 + q_5 + q_6 , \end{aligned}\right\}$$

$$\left.\begin{aligned} \text{where} \quad q_1 &= (a_{11} - a_{12})b_{22} , \\ q_2 &= (a_{21} - a_{22})b_{11} , \\ q_3 &= a_{22}(b_{11} + b_{21}) , \\ q_4 &= a_{11}(b_{12} + b_{22}) , \\ q_5 &= (a_{11} + a_{22})(b_{22} - b_{11}) , \\ q_6 &= (a_{11} + a_{21})(b_{11} + b_{12}) , \\ \text{and} \quad q_7 &= (a_{12} + a_{22})(b_{21} + b_{22}) . \end{aligned}\right\} \quad (2.37)$$

Strassen in [5] gives no hint of how the identities (2.37) were discovered, and they are certainly not immediately obvious. I shall give a "graphical" method which makes the ideas clearer, and which enables one to rediscover the identities (2.37) in a few minutes if they are not at hand. We want the four sums of products

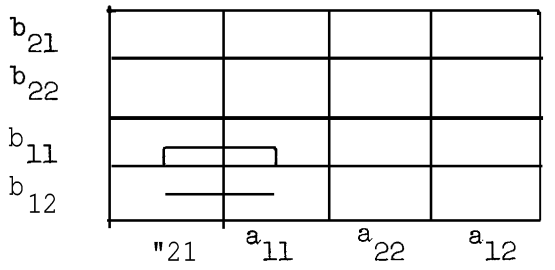
$$c_{ik} = a_{i1}b_{1k} + a_{i2}b_{2k} \qquad (i, k = 1, 2).$$

This might be represented diagrammatically thus:

b_{21}			21	11
b_{22}		x	22	12
b_{11}	21	11	y	
b_{12}	22	12		
	a_{21}	a_{11}	a_{22}	a_{12}

where we want the four sums of products which correspond to similarly labelled squares.

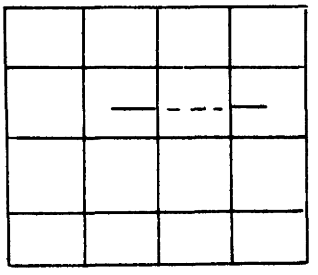
A product $(a_{21} \pm a_{11})(b_{11} \pm b_{12})$ might be represented as:



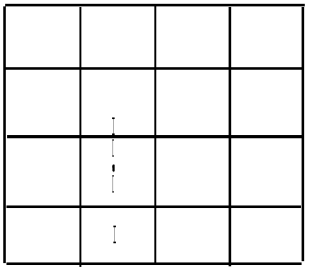
(the signs of the terms are not represented in the diagram)

Now consider the representations of the seven products q1, . . . q_7 of (2.37). For example,

q_1 →



and q_4 →



It is immediately obvious from the diagrams that we can combine q_1 and q_4 linearly to give terms involving the products $a_{11}b_{12}$, $a_{12}b_{22}$, and $a_{11}b_{22}$. It is conceivable that for a suitable combination the $a_{11}b_{22}$ term will drop out and leave c_{12}. If the reader now draws the representations of q_1, q_2, . . . , q_7 and sees how they combine according to (2.37) to give c_{11}, . . . , c_{22}, he will see that one could reconstruct the identities (2.37) from the easily remembered graphical representations, apart from ambiguities in sign. A little thought and juggling of signs will then give a set of identities equivalent to Strassen's (there may be a trivial permutation of the suffices).

It is interesting to experiment with other graphical representations and convince oneself that it is impossible to multiply 2 x 2 matrices in less than seven multiplications. Winograd [8] claims to have proved this.

In Sec. 4 we shall discuss how to implement Strassen's method for rectangular matrices, and how to avoid any wasteful "bordering" with zeros. The question of roundoff errors will be discussed in Sec. 3.

3. Error Analysis

The most important case in practice is that of real matrices and limited-precision floating-point computation. I shall use Wilkinson's notation [6], and assume all arithmetic operations are done in t-digit rounded binary arithmetic*, except that some operations may be especially noted to be done in double-precision (2t-digit). Wilkinson's assumptions concerning the method of rounding or truncating will be made. Some of these assumptions, e.g. binary arithmetic, do not hold for the IBM 360, and this will be discussed later. For simplicity, all matrices will be assumed to be square (n x n).

It will be convenient to use the norm

$$\|X\|_M = \max_{1\le i,j\le n} |x_{ij}| \qquad (3.01)$$

(note that $\|XY\|_M \le \|X\|_M . \|Y\|_M$ is generally false). This norm will usually be written just as $\|X\|$. The results obtained may be expressed in terms of more usual matrix norms by using the attainable bounds

$$\|X\|_M \le \|X\|_q \le n . \|X\|_M \quad , \qquad (3.02)$$

where q stands for 1, 2, ∞, or E.

Wilkinson [6] defines numbers t_1 and t_2 which are slightly less than t. Wherever t_1 or t_2 appear there is the implicit assumption that $n.2^{-t} < 0.1$, which is no restriction in practical cases.

* The analysis is similar with any base $\beta \ge 2$, and in most cases the same bounds will hold with 2^{-t} replaced by $\frac{1}{2}\beta^{1-t}$. For a discussion of Winograd's method, and some further applications of (2.21), with base $\beta > 2$, see [12].

3.1 The Normal Method

Wilkinson [6] shows that if

$$C = fl(A.B) = A.B + E \tag{3.11}$$

then
$$\|E\|_E \leq (2^{-t_1}) \cdot n^* \|A\|_E . \|B\|_E \quad . \tag{3.12}$$

He notes that if $\|AB\|_E \ll \|A\|_E . \|B\|_E$ then the relative error in C may be high. On the other hand, if the inner-products are accumulated in double-precision,

then
$$\|E\|_E \leq 2^{-t} . \|AB\|_E + \frac{3n}{2} . 2^{-2t_2} . \|A\|_E . \|B\|_E , \tag{3.13}$$

and hence the relative error in C will be low unless there is so much cancellation that
$$\frac{\|A\|_E . \|B\|_E}{\|AB\|_E} > \frac{2^t}{n} ,$$

To get a bound in terms of the norm $\|.\|_M$, consider a typical term in the product C. Such a term will be an inner-product

$$fl(\sum_1^n x_i y_i) = \sum_1^n x_i y_i + e \quad \text{say.}$$

If the sum is accumulated in the natural order, we have

$$|e| \leq 2^{-t_1} . (n.|x_1|.|y_1| + n.|x_2|.|y_2| + (n-1).|x_3|.|y_3| + \ldots + 2.|x_n|.|y_n|) , \tag{3.14}$$

$$\text{so} \quad |e| \leq 2^{-t_1} . \frac{(n^2 + 3n - 2)}{2} . \max|x_i| . \max|y_i| \quad . \tag{3.15}$$

As the x_i are elements of A, the y_i elements of B, (3.15) and the definition (3.01) give
$$\|E\|_M \leq 2^{-t_1} . \frac{(n^2 + 3n - 2)}{2} . \|A\|_M . \|B\|_M \quad . \tag{3.16}$$

(3.12) and (3.16) are of the same form

$$\|E\| \leq 2^{-t_1} . f(n) . \|A\| . \|B\| \quad , \tag{3.17}$$

and a bound of this form, with some reasonable f(n), is the best we can expect for any single-precision method.

For double-precision accumulation of inner-products, the bound corresponding to (3.13) is

$$\|E'\|_M \leq 2^{-t}.\|AB\|_M + \tfrac{3}{4}\,.\,(n^2 + 3n - 2).2^{-2t_2}.\|A\|_M.\|B\|_M \quad . \tag{3.18}$$

Again, unless there is exceptional cancellation, the relative error in C will be low.

3.2 Winograd's Method

First consider a simple inner-product

$$\left.\begin{aligned} p &= fl(\gamma - (\xi + \eta)) \ , \\ \text{where} \qquad \gamma &= fl\Big(\sum_1^{n/2} (x_{2j-1} + y_{2j})(x_{2j} + y_{2j-1})\Big) \ , \\ \xi &= fl\Big(\sum_1^{n/2} x_{2j-1}x_{2j}\Big) \ , \\ \text{and} \qquad \eta &= fl\Big(\sum_1^{n/2} y_{2j}y_{2j-1}\Big) \ , \end{aligned}\right\} \tag{3.21}$$

computed by Winograd's method (n even).

A simple example illustrates what can happen when limited-precision arithmetic is used. Suppose we are using G-decimal floating arithmetic, n = 2, $x_1 = x_2 = 1.000'+3$, $y_1 = y_2 = 1.000'-3$.

Then $\xi = 1.000'+6$

and $\eta = 1.000'-6$ (both exactly correct),

but $\gamma = 1.000'+6$ (instead of the exact $1.000002000001'+6$),

so $p = 0.000$ instead of 2.000 . The difficulty is in forming $fl(x_{2j-1} + y_{2j})$ etc. when the elements of x may differ widely in magnitude from the elements of $\underline{y}$. This conclusion will also follow from the rigorous error analysis below.

Let $a = \max |x_i|$ and $b = \max|y_i|$, (3.22)

and let $\xi = \sum_{1}^{n/2} x_{2j-1}x_{2j} + \epsilon_\xi$ etc.

From (3.15) with n replaced by n/2 we get

$$\left.\begin{aligned} |\epsilon_\xi| &\le 2^{-t_1}.a^2.(n^2 + 6n - 8)/8 , \\ \text{and similarly} \quad |\epsilon_\eta| &\le 2^{-t_1}.b^2.(n^2 + 6n - 8)/8 . \end{aligned}\right\} \tag{3.23}$$

If $fl(x + y) = x + y + \epsilon_{x+y}$ (x any x_i, Y any y_i)

then $$|\epsilon_{x+y}| \le 2^{-t}.(|x| + |y|)$$
$$< 2^{-t}.(a+b) . \tag{3.24}$$

Thus $$fl((x+y)(x'+y')) = (1 + \epsilon_0)(x + Y + \epsilon_1)(x' + Y' + \epsilon_2) \tag{3.25}$$
$$= (x + y)(x' + y') + \epsilon_z \quad \text{say,}$$

where $|\epsilon_0| \le 2^{-t}$ and $|\epsilon_1|, |\epsilon_2| \le 2^{-t}(a + b)'$.

By expanding (3.25) it follows that

$$|\epsilon_3| \le 2^{-t_3}.3.(a + b)^2 , \tag{3.26}$$

where t_3 is defined by

$$2^{-t_3} = 2^{-t} + 2^{-2t} + 2^{-3t} ,$$

(so in practice $t_3 \simeq t$).

Hence $$|\epsilon_\gamma| \le 2^{-t_3}.(3n/2).(a + b)^2 +$$
$$2^{-t_1}.((n^2 + 2n - 8)/8)(a+b)^2(1+3.2^{-t_3}) .$$

In all practical cases

$$(3n/2 + 3.2^{-t_1}((n^2 + 2n - 8)/8)).2^{-t_3} \le (3n/2) \bullet 2^{-t_1} ,$$

and with this assumption we get

$$|\epsilon_\gamma| \le 2^{-t_1}.((n^2 + 14n - 8)/8).(a + b)^2. \tag{3.27}$$

From (3.23) and (3.27), the error ϵ in p is bounded by

$$|\epsilon| \leq 2^{-t_1}\left[((n^2 + 14n - 8)/8)(a + b)^2 + ((n^2 + 6n - 8)/8)(a^2 + b^2) + |\gamma - \xi - \eta| + |\xi| + |\eta|\right] \qquad (3.28)$$

(terms of order 2^{-2t} have been neglected, but they may be dealt with as above (see [12])).

Now $|\gamma - \xi - \eta| \leq nab + O(2^{-t})$, $|\xi| \leq \frac{n}{2} a^2 + |\epsilon_\xi|$, $|\eta| \leq \frac{n}{2} b^2 + |\epsilon_\eta|$

and $a^2 + b^2 < (a + b)^2$,

so $$|\epsilon| \leq 2^{-t_1} \cdot \frac{n^2 + 12n - 8}{4} \cdot (a + b)^2 . \qquad (3.29)$$

By considering (3.29) with n replaced by n - 1 and a term added for the error in computing and adding $x_n y_n$, it may be shown that (3.29) holds whether n is even or odd, and bounds the error in computing an inner-product by Winograd's method. From (3.29) we obtain the bound

$$\|E\| \leq 2^{-t_1} \cdot \frac{n^2 + 12n - 8}{4} \cdot (\|A\| + \|B\|)^2 \qquad (3.210)$$

for matrix multiplication by Winograd's method. (A slightly stronger result than (3.29) can be obtained if a = b, see [12].)

Suppose $\|A\| / \|B\| = k$. (Assuming $k \neq 0$ or ∞)

Then

$$(\|A\| + \|B\|)^2 = (k + 2 + 1/k) \cdot \|A\| \cdot \|B\| ,$$

which shows that (3.210) will be much worse than (3.16) when k is very small or very large, and this is verified by the example above.

Scaling

Ignoring the cases $\|A\| = 0$ and $\|B\| = 0$, it is always possible to find an integer λ such that $1/2 \leq \frac{2^\lambda \|A\|}{2^{-\lambda}\|B\|} \leq 2$. Hence a practical

scheme would be to compute $\|A\|$ and $\|B\|$ (in $O(n^2)$ operations), find λ, and then apply Winograd's method to $2^{\lambda}A$ and $2^{-\lambda}B$ rather than to A and B. If this is done, then since

$$\max_{1/2\le k\le 2} (k + 2 + 1/k) = 9/2,$$

we get, in place of (3.210), the bound

$$\|E\| \le 2^{-t_1}\cdot\frac{9}{8}\cdot(n^2 + 12n - 8)\cdot\|A\|\cdot\|B\|, \qquad (3.211)$$

which is of the form (3.17) and is not much worse than (3.16). This shows that Winograd's method is feasible provided some form of scaling is used to make $\|A\| \sim \|B\|$. Without scaling, the results may easily lose all significance. This does not seem to have been mentioned by anyone recommending the use of Winograd's method: e.g. blindly following the procedure recommended in [2] could lead to disaster.

A more sophisticated form of scaling could be used, but it is important to keep the time for scaling to a minimum, or Winograd's method becomes slower than the normal method. The extra time taken by scaling will be considered in Sec. 4.

If it is easy to accumulate inner-products in double-precision then this may as well be done. The error bound will still be like (3.211) though, unless the terms $a_{i,2j-1} + b_{2j,k}$ and $a_{i,2j} + b_{2j-1,k}$ of (2.21) are computed in double-precision. Then we get a bound

$$\|E\| \le 2^{-t}\cdot\|A\|\cdot\|B\| + 2^{-2t}\cdot\tfrac{7}{2}(p+n2 + 12n - 8)\cdot\|b\|\cdot\|B\|, \qquad (3.212)$$

provided that the terms x_i and y_k of (2.22), (2.23) are kept in double-precision, and assuming scaling as above. (3.212) is very similar to (3.18) and the same remarks apply.

3.3 Strassen's Method

Assuming a bound $\|E\| \leq 2^{-t}.f(n).\|A\| \bullet \|B\|$ (3.31)

for n x n matrices, it is possible to deduce a similar expression for 2n x 2n matrices, if the multiplication of these matrices is reduced to the multiplication and addition of n x n matrices using Strassen's identities (2.37). This gives $f(2n)$ in terms of $f(n)$, and as (3.31) is certainly true when $n = 1$ (with $f(1) = 1$), we can find $f(n)$ for n an integral power of 2. If the "bordering" method is used for general n then the zeros will have no effect on the error, so the bound for the next power of two may be used.

To express $f(2n)$ in terms of $f(n)$, let A, B, and C be 2n x 2n matrices (deviating slightly from our usual notation), and regard A, B, and C as 2 x 2 matrices with n x n blocks. Consider forming $C = fl(A.B)$ using the identities (2.37). Terms of order 2^{-2t} will be ignored, for although they may be dealt with by replacing t by $t' \simeq t$ as we replaced t by t_1, t_2 and t_3 in Sec. 3.2, this complicates the argument, and the *results* are not significantly different. For brevity

let $a = \|A\|_M$, $b = \|B\|_M$. (3.32)

The error in computing q_i of (2.37) will be denoted by E_{qi}, so for example $fl((a_{11} - a_{12})b_{22}) = (a_{11} - a_{12})b_{22} + E_{q1}$ (where a_{11}, a_{12}, b_{22} and E_{q1} are n x n matrices). Similarly, the error in computing c_{ij} of (2.37) will be denoted by E_{ij}. Thus

$$C = fl(A.B) = A.B + E, \text{ where } E = \left(\begin{array}{c|c} E_{11} & E_{12} \\ \hline E_{21} & E_{22} \end{array}\right).$$

Since $q_1 = fl((a_{11} - a_{12}).b_{22})$, where the n x n matrix multiplication is done by Strassen's method with the error bound (3.31), and the matrix addition is done in the usual way, we have

$$\|E_{q1}\| \leq 2^{-t}(n + f(n))(\|a_{11}\| + \|a_{12}\|).\|b_{22}\| ,$$

$$\text{so } \|E_{q1}\| \leq 2^{-t}.2ab.(n + f(n)) , \tag{3.33}$$

and similarly for E_{q2}, E_{q3}, and E_{q4} . For i = 5, 6 and 7 we get the bound

$$\|E_{qi}\| \leq 2^{-t}.4ab.(2n + f(n)) \tag{3.34}$$

in the same way.

Now it follows from (2.37), neglecting terms in 2^{-2t}, that

$$\|E_{12}\| \leq \|E_{q1}\| + \|E_{q4}\| + 2^{-t}(\|q_1\| + \|q_4\|) , \tag{3.35}$$

but

$$\|q_i\| \leq \left\{ \begin{array}{ll} 2nab & \text{for } i = 1, 2, 3, 4 \\ 4nab & \text{for } i = 5, 6, 7 , \end{array} \right\} \tag{3.36}$$

so from (3.33), (3.35) and (3.36) we obtain

$$\|E_{12}\| \leq 2^{-t}.4ab.(2n + f(n)), \tag{3.37}$$

and clearly the same bound holds for E_{21}. Similarly we have

$$\|E_{11}\| \leq \|E_{q1}\| + \|E_{q3}\| + \|E_{q5}\| + \|E_{q7}\| +$$

$$2^{-t}(3\|q_1\| + 3\|q_3\| + 2\|q_5\| + \|q_7\|) \tag{3.38}$$

(assuming q_1, q_3, q_5 and q_7 are added in this order),

$$\text{so } \|E_{11}\| \leq 2^{-t}.ab.(44n + 12f(n)) , \tag{3.39}$$

and similarly for E_{22}.

From (3.37) and (3.39) we see that

$$\|E\| \leq 2^{-t}.(44n + 12f(n)).\|A\|.\|B\| , \tag{3.310}$$

so (3.31) will hold if f satisfies f(1) = 1 and $f(2n) = 44n + 12f(n)$. (3.311)

By induction on k, it follows from (3.311) that

$$f(2^k) = \tfrac{1}{5}(27.12^k - 22.2^k) \ , \tag{3.312}$$

$$\text{so} \quad f(2^k) < \tfrac{27}{5}.12^k = \tfrac{27}{5}.(2^k)^{\log_2 12} \ . \tag{3.313}$$

Hence, for general n, taking k such that $n < 2^k < 2n$,

$$\left.\begin{aligned} &\text{we have} \quad \|E\| \le 2^{-t}.65n^c.\|A\|.\|B\| \\ &\text{where} \quad c = \log_2 12 \simeq 3.58 \ . \end{aligned}\right\} \tag{3.314}$$

(3.314) gives a bound for the error in matrix multiplication by Strassen's method, as described in Sec. 2.3. The bound is of the form (3.17), although the function $54n^{3.58}$ increases rather more rapidly than we would like. On the other hand, all the error estimates obtained here are rather pessimistic, for the individual rounding errors are unlikely to be correlated in the worst possible way. If our bound is $2^{-t}f(n)\|A\|.\|B\|$ then the actual error is probably about $2^{-t}\sqrt{f(n)}\,\|A\|.\|B\|$ (see Sec. 4.6).

The analysis above assumes that a "pure" form of Strassen's method is used. In practice it turns out that Strassen's identities will be applied until the matrices to be multiplied are of order ~ 100 or less, and then the normal method will be used (see Sec. 4.3). Supposing we have matrices of order $2^k.n_0$, and apply Strassen's identities k times, multiplying the matrices of order n_0 by the normal method. Then (3.311) holds with

$$f(n_0) = (n_0^2 + 3n_0 - 2)/2 \qquad \text{(from 3.16)} \ ,$$

so, assuming $n_0 > 6$, we have

$$f(2^k n_0) \le 16^k n_0^2 \ . \tag{3.315}$$

Thus, for n x n matrices, the bound becomes $\|E\| \le 2^{-t}.4^k.n^2.\|A\|.\|B\|$. (3.316)

Since k will be very small in practice, the bound (3.316) is not too bad. Comparing it with (3.16), it appears that we may lose up to two bits of accuracy, compared to that of the normal method, each time Strassen's identities are applied recursively.

In using Strassen's method there does not seem to be much point in doing some of the arithmetic in double-precision, unless it can all be done in double-precision, when the above bounds hold with t replaced by 2t (and a factor of 3/2 with Wilkinson's assumptions about the method of rounding or truncating).

It is interesting to note that with Strassen's method there is no point in scaling the matrices so that $\|A\| \sim \|B\|$. This is because, unlike Winograd's identity, Strassen's identities never involve the addition of an element of A to an element of B.

3.4 Complex Arithmetic

The above analysis is based on the assumptions that $fl(x + y) = x(1 + \epsilon_1) + y(1 + \epsilon_2)$ and $fl(xy) = xy(1 + \epsilon_3)$ where $|\epsilon_i| \leq 2^{-t}$, $i = 1, 2, 3$*. These assumptions will be valid for complex arithmetic too, provided that t is decreased by a small amount (2 or 3) depending on how the arithmetic is done. Hence, with this small change in t, the above bounds will hold for complex matrix multiplication. Similar remarks apply to real arithmetic done on a decimal or hexadecimal machine (e.g. the IBM 360). A curious anomaly which appeared when Winograd's method was being tested on an IBM 360/67 computer is described in Sec. 4.6.

---

* A stronger assumption about addition, used in Section 3.2, was not really necessary (see [12]).

4. Implementation

In order to compare the normal, Winograd's and Strassen's methods in practice, they were all implemented in ALGOLW [10] on an IBM 360/67 computer. Doubtless all three methods would run faster if coded in, say, FORTRAN-H or assembly language, but their relative speeds would probably be about the same. While it would be easy enough to code the normal method and Winograd's method in FORTRAN or assembly language, for Strassen's method it is very convenient to have a language which allows recursive procedure calls. The simplest way to code Strassen's method in a language like FORTRAN would be to limit the depth of recursion and duplicate any subroutines which would naturally be called recursively. The three methods were tested on both real and complex matrices, with results which will be summarized below.

All three methods were coded in the form of a pure procedure, with calling sequence

name (A, B, C, M, N, P)

to form C := A.B , where A is an M x N matrix (dimensioned (1 :: M, 1 :: N)), B is N x P, and C is M x P. Calls such as name (A, A, A, N, N, N) are valid, and correct results should be returned for any M, N and P > 1, provided enough temporary storage is available.

At first the procedures were coded so that the "inner loops" involved references to doubly-subscripted array elements. In ALGOLW such references take considerably longer than references to singly-subscripted array elements [11], and it was found that all the procedures could be speeded up by passing cross-sections of two-dimensional arrays as parameters to procedures which then operated on them as one-dimensional

arrays. (This is not allowed in ALGOL-60.) For example, instead of:

```
For I := 1 until M do
for J := 1 until N do A(I,J) := B(I,J);
```

we use:

```
For I := 1 until M do assign (A(I,*),B(I,*),N);
```

where we have defined

```
Procedure assign (real array A, B(*); integer value N);
for J := 1 until N do A(J)   := B(J);
```

The second form will execute faster provided N > 10 . As this device speeded up the normal method rather more than Strassen's method, it is clear that a comparison of the three methods depends on the language and the programming techniques used to implement them.

The implemention of each method will now be described in more detail. The procedure for the real and complex cases are very similar, and listings for the real case are given in the Appendix.

4.1 The Normal Method

(Procedure MATMULT, see Appendix, lines 288-311.) There are no particular difficulties in the implementation of this method. Because of the possibility that C is the same as A or B in the call, the product is formed in a temporary array Q and then transferred to C. Thus M.P words of temporary storage are used. Inner-products are accumulated in double-precision, for in ALGOLW this is very nearly as fast as accumulation in single-precision. Hence the error bounds (3.13) and (3.18) are applicable (with the alteration noted in Sec. 3.4), and in most cases each c_{ij} will be the correctly rounded result, although this can not be guaranteed.

4.2 Winoarad's Method

(Procedure WINOGRAD, see Appendix, lines 219-285.) Again the implementation is fairly straight-forward. The matrices A and B are scaled as described in Sec. 3.2, and the scaled matrices are stored temporarily in arrays D and E. Strictly speaking, scaling should be done to the nearest power of 16 rather than 2, for scaling by powers of 2 could introduce roundoff errors on the 360, and these errors have not been taken into account in the error analysis (Sec. 3.2). Taking account of these errors gives the error bound

$$\|E\| \leq 2^{-t_1} Kn^2 . \|A\| . \|B\| , \tag{4.21}$$

where K is a small constant, instead of (3.211). In the complex case, $|R(x)| + |I(x)|$ rather then $|x|$ was used to save time. This increases the error bound by a factor of at most 1.15 .

The inner-products x_i and y_k of (2.22), (2.23) are computed and stored in the arrays X and Y. As stated above, it is not significantly harder to compute and save the x_i and y_k in double-precision, so this is done.

In all, $(n + 2)(m + p)$ words of temporary storage are used, which is about twice as much as for the normal method if $m = n = p$. The sums $(a_{i,2j-1} + b_{2j,k})$ and $(a_{i,2j} + b_{2j-1,k})$ of (2.21) are computed in single-precision, and then the inner-product involving them is computed, as usual, in double-precision. If n is odd then the necessary correction is made, and the final result fl(C) is formed. It is interesting to note that if the sums $(a_{i,2j-1} + b_{2j,k})$ and $(a_{i,2j} + b_{2j-1,k})$ were computed in double-precision, we would be using double-precision throughout,

and the bound (3.212) would apply. Unfortunately,the extra time taken to do this slows the procedure down so that it is never faster than the normal method, so the sums could only be computed in single-precision, and the best error bound we can get is of the form of (4.21).

4.3 Strassen's Method

(Procedure STRASSEN, see Appendix, lines 6-216.) The method implemented is the following: First, if m, n and p are sufficiently small, normal matrix multiplication is used (see below for the precise criterion). Otherwise, m is replaced by $2\lfloor m/2 \rfloor$, n by $2\lfloor n/2 \rfloor$, and p by $2\lfloor p/2 \rfloor$. A is partitioned into four m/2 by n/2 matrices and B into four n/2 by p/2 matrices, ignoring the last row and/or column if necessary. The block 2 by 2 matrices are multiplied using Strassen's identities (2.37), which involves seven recursive calls to STRASSEN to compute the m/2 by p/2 products $q_1, \ldots q_7$ (actually C is used in place of Q7 to save storage). Finally, the result is corrected if the original m, n or p were odd. This avoids wasting space and time by filling up the arrays with zeros as described in Sec. 2.3 . In case C coincides with A or B, some values needed for the correction step have been saved in arrays S1 and S2.

Actually implementing the identities (2.37) is tedious but straightforward. The fast, general-purpose procedure OP is used to take advantage of the facility, noted above, for passing cross-sections of arrays as parameters to procedures. In forming c_{11} and c_{22}, the terms $q_1 \ldots q_4$ are added before $q_5 \cdots q_7$, for otherwise the error bound would be increased slightly. All arithmetic is done in single-precision except

for the accumulation of inner-products when normal matrix multiplication is used, so the error bound (3.316) is applicable. Because of the double-precision accumulation of inner-products, the term $4^k n^2$ in this bound may be replaced by $5.12^k n_0$.

Procedure IDENTITIES uses the temporary arrays T, U, Q1, Q2, .*a , Q6 , taking $(mn + np + 6pm)/4$ words. Since the procedure is called recursively, at any one time we may need $\leq (mn + np + 6pm)(4^{-1} + 4^{-2} + 4^{-3} + \ldots)$

$$= (mn + np + 6pm)/3 \text{ words of temporary storage.} \quad (4.31)$$

The arrays S1 and S2, and the stack space required for recursive procedure calls, will be negligible if m, n and p are reasonably large. The space for the array Q, used when normal matrix multiplication is invoked, may be absorbed into (4.31). Hence the temporary storage used is roughly bounded by (4.31), and if $m = n = p$ this is $8n^2/3$ words, or slightly more than that required by Winograd's method and 8/3 times that required by the normal method. For all three methods, the temporary storage requirements can be reduced if C is not allowed to overlap A or B.

4.4 Comparison of the Three Methods

The three procedures described above were run under the same conditions (idle with "nocheck" option) for various test matrices A and B. Some running times for the case of square matrices are given in Table 1. In each case the depth of recursion in procedure STRASSEN was kept at exactly one.

Table 1 Running Times (in 1/60 sec.)

m = n = p	Real case			Complex case		
	Normal	Winograd	Strassen*	Normal	Winograd	Strassen*
20	28	34	42	53	53	66
30	83	88	107	167	150	187
40	184	184	221	384	330	401
50	347	336	392	731	615	742
60	584	557	636			

*Strassen's method with exactly one recursion. Run times varied slightly, but were constant to ± 1%.

By counting operations it is clear that the running time of each method should be a cubic in n, and for Strassen's method the coefficients will depend on the depth of recursion. It turns out that the constant term is negligible, and the times in Table 1 are given to + 1% by cubics $T(n) = an^3 + bn^2 + cn$ with the following coefficients:

Table 2 Cubic Coefficients, $T = an^3 + bn^2 + cn$, in μ sec.

		a	b	c
Real	Normal	40	270	2000
	Winograd	37	200	9500
	Strassen*	36	650	8000
Complex	Normal	90	320	2000
	Winograd	73	220	11500
	Strassen*	80	790	8000

Some interesting conclusions may be drawn from Tables 1 and 2. Comparing the normal method with Winograd's method, we see that Winograd's will be faster if $37n^3 + 200n^2 + 9500 < 40n^3 + 270n^2 + 2000$, i.e. if $n \geq 40$ in the real case, and if $73n^3 + 220n^2 + 11500 < 90n^3 + 320n^2 + 2000$, i.e. if $n > 21$ in the complex case, which may be verified by inspection of Table 1. As $n \to \infty$, Winograd's method will run in $37/40 = 92\%$ of the normal time in the real case, and in $73/90 = 81\%$ of the normal time in the complex case. The gains are significant for reasonably small n: e.g. for $n = 100$ Winograd's method will save 7% (real) or 18% (complex). Hence, for moderately large matrices, Winograd's method leads to significant, though not spectacular, savings, and is worthwhile especially in the complex case.

It is worth noting here that it does not pay to reduce the multiplication of two complex n by n matrices to three multiplications of real n by n matrices (plus some additions) by using

$$(A + Bi)(C + Di) = (E - F) + (G - E - F)i, \text{ where } E = AC,\ F = BD, \text{ and } G = (A + B)(C + D), \tag{4.41}$$

for complex matrix multiplication takes less than three times as long as real matrix multiplication (using any of the three methods).

It follows from Table 2 that Strassen's method will be faster than the normal method if $n \geq 110$ in the real case, and if $n \geq 60$ in the complex case. Hence procedure STRASSEN should check to see if $n < n_0$ (with n_0 set at 110 or 60), and if so use the normal method. If $n \geq n_0$ then Strassen's identities should be used to reduce n to n/2, and the same test applied recursively. This is what the procedure actually does, except that n_0 is not compared just with n, but also with m and p in case the matrices are rectangular. It can be seen by counting operations that the appropriate test is if $3mnp < n_0(mn + np + pm)$ rather

than if $n < n_0$. The times given in Table 1 were obtained with n_0 reduced so that Strassen's identities would be used exactly once.

By counting operations, it can easily be seen that the time $T_S(n)$ for multiplication of n by n matrices using Strassen's method should be given by

$$T_S(n) = \begin{cases} an^3 + bn^2 + en + d & \text{if } n < n_0 \\ 7T_S(n/2) + a'n^2 + b'n + c' & \text{if } n \geq n_0 . \end{cases} \tag{4.42}$$

From (4.42) it follows that, if

$$k = \max(0, \lfloor \log_2(n/n_0) \rfloor + 1) ,$$

then

$$\left.\begin{aligned} T_S(n) = {} & (\tfrac{7}{8})^k an^3 + ((\tfrac{7}{4})^k b + \tfrac{4}{3}((\tfrac{7}{4})^k - 1)a')n^2 \\ & + ((\tfrac{7}{2})^k c + \tfrac{2}{5}((\tfrac{7}{2})^k - 1)b')n \\ & + (7^k d + \tfrac{1}{6}(7^k - 1)c') \end{aligned}\right\} \tag{4.43}$$

The constants a, b, c and d should be those given for the normal method in Table 2 (d is negligible). The constants a', b' and c' determined to fit the data in Table 1 are:

Table 3	Constants in (4.42)	(μ sec.)	
Real case	a' = 190	b' = 4000	c' = 120000
Complex case	220	4000	120000

The constants in Tables 2 and 3 are not very well determined by the data (especially c and c'), and are not exactly consistent. For example, from (4.42) and (4.43) we should.have, in Table 2, $a_S = 7a_N/8$, while the Table gives $a_S = 36$ and $a_N = 40$. The consistency is about as good as can be expected though.

From (4.42) and (4.43) it follows that $T_S(n) = O(n^{\log_2 7})$ as $n \to \infty$,

so for sufficiently large matrices Strassen's method is arbitrarily faster than the normal method or Winograd's method. In practical cases, say for $n < 200$, the normal method or Winograd's method appears to be faster. By the above formulae we can estimate that Strassen's method will be faster than Winograd's only if $n > 270$ (real case) or $n > 280$ (complex case). On the other hand, these changeover points are very sensitive to changes in programming techniques etc., so it is conceivable that Strassen's method would be the fastest, in some language on some machine, for matrices of order ~ 150. In most practical cases, Winograd's method will be the fastest, except that the normal method will be faster for sufficiently small matrices.

4.5 Paged Machines

Some machines (e.g. the Burroughs B5500) have a fairly small physical memory but a large "virtual" memory. The user's program and data is divided into "pages", some of which may be held in fast core memory, and the others on a device such as a disc or drum. When reference is made to a page which is not in memory, a hardware interrupt occurs, and the required page is read into memory from the external device (to make room for it, a page may have to be saved on the device). We say that a "page fault" has occurred. As a relatively slow external device is involved, page faults are very time-consuming and should be avoided as much as possible. (For a discussion of the concepts of virtual memory, paging, segmentation etc. see Randell and Kuehner [9].)

Mc Kellar and Coffman [4] have considered the number of page faults which will occur when certain matrix operations, including multiplication, are performed on large matrices using a machine with paging like that

described above. They conclude that, for a slight modification of the normal method of matrix multiplication, it is better to store a large matrix by submatrices, with each submatrix fitting into a small number of pages, than by rows or columns. Even then, the number of page faults will increase like n^3 for sufficiently large n. Similar arguments would apply to Winograd's method, again suitably modified.

Unlike the normal method or Winograd's method, Strassen's method would perform well, with eventually $O(n^{2.8})$ page faults, even when simple row or column storage is used. This is because the only matrix operations on matrices with $n > n_0$ are assignment and addition operations, and these can be performed as efficiently when row or column storage is used as for any other method of storage. A few modifications to the procedure STRASSEN in the Appendix should be made. n_0 should be decreased if necessary so that n_0 by n_0 matrices can be multiplied in core (without any page faults). Also, inner loops should involve operations on one row rather than on one column, if row storage is used. Thus we should change double loops like

```
For J := 1 until N do for I := 1 until M do ...
```

to

```
For I := 1 until M do for J := 1 until N do ...
```

This also applies to the "implicit" loops when procedure OP is called: e.g. lines 138 -139 should be changed to

```
For I := 1 until M2 do
OP(T(I,*),A(I,*),A(I,*),M2,0,N2,-1);
```

Hence Strassen's method might be competitive with the other methods for smaller values of n on a paged machine than on a machine without paging.

4.6 Rounding Errors

The procedures were tested using matrices with elements uniformly distributed in (-1/2, +1/2), or with real and imaginary parts having this distribution. $\|E\|_E^2$ and $\|E\|_M$ were computed, assuming that the normal method gave exact results, which is reasonable considering the error bounds (3.13) and (3.18). As expected, the error bounds (3.211) and (3.316) of the form $\|E\| \leq 2^{-t} f(n) \|A\| . \|B\|$ were too pessimistic, and the actual $\|E\|$ was more like $2^{-t} \sqrt{f(n)} \|A\| . \|B\|$: See Table 4.

Table 4 $\quad \|E\|_M / (2^{-t} \sqrt{f(n)} \|A\|_M \|B\|_M)$

n	Real Strassen	Complex Strassen	Complex Winograd
30	0.27	0.28	0.28
40	0.20	0.83	0.24

(taking $f(n) = \begin{cases} \frac{9}{8}(n^2 + 12n - 8) & \text{for Winograd,} \\ 4^k n^2 & \text{for Strassen,} \end{cases}$ and $t = 21$)

A surprising result occurred with Winograd's method in the real case. The single-precision results agreed exactly with those given by the normal method! This might be expected if the error bound (3.212), rather than (3.211), were applicable' The anomaly is apparently caused by the special nature of the test matrices and the characteristics of floating-point arithmetic on the 360/67. As the elements of A and B were uniformly distributed in (-1/2, +1/2), about 7/8 of them would have absolute values in (1/16,1/2) . Since the 360 is a hexadecimal machine, any two such numbers will be added <u>exactly</u>. This means that at least 49/64 of the sums $(x_{2j-1}$ □ ϕ_{2j} ◊ and $(x_{2j} + y_{2j-1})$ of (3.21) will be formed exactly. As

remarked in Sec. 3.2, this means that we are effectively using at least double-precision most of the time. Presumably the few errors made in computing the above sums were not enough to affect the rounded single-precision results, although it seems strange that all the elements of a 50 x 50 product should agree, even to the last bit, when computed by two such different methods. In the complex case this anomaly disappears, for a rounding error will usually be made in adding either the real or the imaginary parts of the above sums.

5. Strassen-like Methods

For 2 x 2 matrix multiplication, both the normal method and Strassen's method may be described as follows: given the a_{ij} and b_{kL}, we form products $q_1, \ldots, q_T$ of the form

$$q_p = (\sum \alpha_{ijp} a_{ij})(\sum \beta_{kLp} b_{kL}) \quad , \tag{5.01}$$

and then the c_{nm} are linear combinations of the q_p, i.e. there are constants γ_{mnp} such that

$$c_{nm} = \sum_{p=1}^{T} \gamma_{mnp} q_p \quad . \tag{5.02}$$

Substituting (5.01) in (5.02), equating coefficients, and using the definition of matrix multiplication, gives the set of equations

$$\sum_{p=1}^{T} \alpha_{ijp} \beta_{kLp} \gamma_{mnp} = \delta_{ni} \delta_{jk} \delta_{Lm} \quad , \tag{5.03}$$

where δ is Kronecker's delta. (The subscripts on the c_{nm} were reversed to increase the symmetry of (5.03).) For the multiplication of M x N matrices by N x P matrices, (5.03) gives $(MNP)^2$ equations as i, j, k, L, m, and n range over the integers $1 \leq i,n < M$, $1 \leq j,k < N$, $1 \leq L,m < P$. For example, in the 2 x 2 case with T = 7, we have 64 equations in 84 unknowns, and Strassen's identities show that there is a solution. Strassen's solution has the nice property that all the α_{ijp}, β_{kLp} and γ_{mnp} are 0 or +1 . Note that, if a solution of (5.03) exists, it will certainly not be unique.

Strassen's method applied to 4 x 4 matrices shows that the equations (5.03) have an (integral) solution when M = N = P = 4, T = 49 (there are 4096 equations in 2352 unknowns). In general Strassen's method shows that there is a solution with $T = 7^k$ when $M = N = P = 2^k$.

If there is a real solution with M = N = P and a certain T, then matrices of order n can be multiplied in $O(n^{\log_N T})$ arithmetic operations by a simple extension of the method described at the beginning of Sec. 2.3. While an integral or rational solution is desirable, in theory a real or even a complex solution would suffice.

The problem leading to equation (5.03) can be generalized in the following way: suppose $a_1, \ldots, a_I$ and $b_1, \ldots, b_J$ are non-commuting variables, σ_{ijk} is a given three-dimensional array of real or complex numbers, and we want to compute the K sums of products $q_k = \sum \sigma_{ijk} a_i b_j$ $(k = 1, \ldots K)$ in as few multiplications as possible. Then we want the least possible T and scalars $\alpha_{it}, \beta_{jt}, \gamma_{kt}$ such that from the T products

$$p_t = \Big(\sum_i \alpha_{it} a_i\Big)\Big(\sum_j \beta_{jt} b_j\Big) \quad , \ 1 \le t \le T \ , \tag{5.04}$$

we can form the q_k as linear combinations of the p_t ,

$$q_k = \sum_{t=1}^{T} \gamma_{kt} p_t \quad , \ 1 \le k \le K. \tag{5.05}$$

Combining (5.04) and (5.05) and equating coefficients gives

$$\sum_{t=1}^{T} \alpha_{it}\beta_{jt}\gamma_{kt} = \sigma_{ijk} \qquad \text{for } 1 \le i \le I, \quad 1 \le j \le J, \quad 1 \le k \le K, \tag{5.06}$$

and clearly (5.03) is a special case of (5.06).

To sharpen the upper bound (2.36) for the constant β_0 defined by (2.35), we could look for solutions of (5.03) with M = N = P and $\log_N T < \log_2 7$. For example, we would like to find solutions with N = 2, T = 6 or N = 3, T = 21 or N = 4, T = 48. As (5.03) is a special case of (5.06), and as it is convenient to avoid triple subscripts wherever possible, we shall first consider (5.06).

In the case I = 1 it is not difficult to show that the minimal T for which a solution of (5.06) exists is the rank of the J x K matrix (σ_{1jk}), and similarly if J or K = 1, If I, J and K are greater than unity then there does not seem to be any such simple theorem,- and examples with I = J = K = 2 show that the minimal T may depend on whether the α_{it}, β_{jt} and γ_{kt} are allowed to be rational, real, or complex. This is so even if all the σ_{ijk} are integral. Hence we are led to try numerical methods for solving special cases of (5.06). If these methods find a real solution, then it is worthwhile to try to find an integral solution, but if no real solution exists there is no point in looking for an 'integral solution.

5.1 Least Squares Approach

Because of the large number of equations (4096 for N = 4), conventional numerical methods like Newton's method are impractical for finding a solution of (5.06). The problem may be regarded as one of function minimization: we want to minimize the sum of squares of residuals of the set of equations (5.06). If $\underline{\beta}$ and $\underline{\gamma}$ are fixed , then (5.06) is a set of linear equations in the α_{it*} Hence we could find a least-squares solution of this (overdetermined) system, then fix $\underline{\gamma}$, $\underline{\alpha}$ and find a least squares solution for $\underline{\beta}$, then for $\underline{\gamma}$, and repeat the cycle. The sum of squares of residuals will converge to some non-negative number, and hopefully this will be zero. Even this method would be impractical, except that the coefficient of α_{it} in the system of linear equations happens to be independent of i. In other words, the matrix of coefficients has I identical T x T blocks along the main diagonal, and zeros elsewhere, so each least-squares problem splits up into a number of smaller ones.

Writing x_t for α_{it}, we want the least squares solution of $Ax = \underline{b}$, where $A = (\beta_{jt}\gamma_{kt})_{(j,k),t}$. (5.11)

The solution is given by $\underline{x} = (A^TA)^{-1}A^T\underline{b}$ (in the real case) , (5.12)

and we have

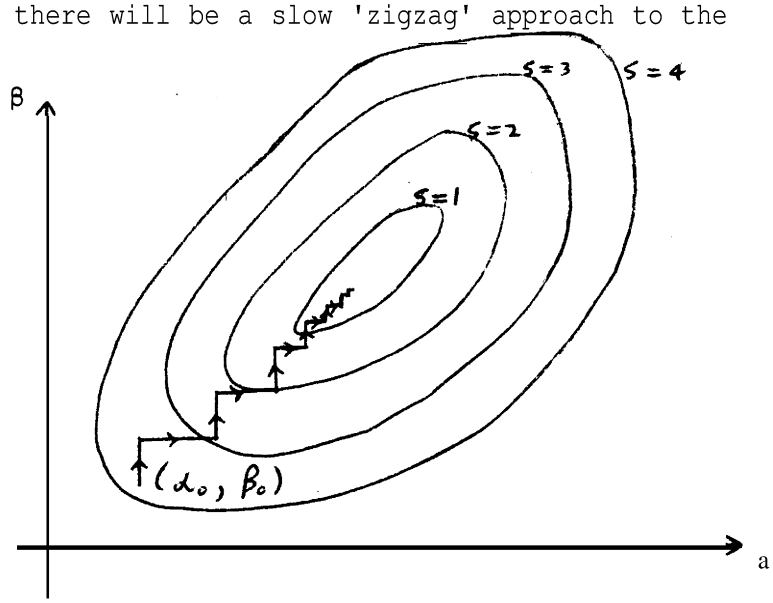
$$A^TA = ((\sum_j \beta_{jt}\beta_{ju})(\sum_k \gamma_{kt}\gamma_{ku}))_{t,u} \tag{5.13}$$

$$\text{and } A^T\underline{b} = (\sum_{j,k} \beta_{jt}\gamma_{kt}\sigma_{ijk})_t \ . \tag{5.14}$$

As noted above, (5.13) is independent of i, but (5.14) depends on i.

5.2 Acceleration of Convergence

It is not clear how one should make a good initial guess at a solution of (5.06), but in any case, with randomly chosen $\underline{\alpha}$, $\underline{\beta}$, and $\underline{\gamma}$, the initial rate of convergence is rapid. Unfortunately the convergence soon slows down. One possible difficulty may be illustrated by a two-dimensional example: suppose we try to minimize $s(\alpha,\beta)$ by fixing β, minimizing s with respect to a, fixing α and minimizing s with respect to β, etc. If the contour lines of s are ellipses as illustrated in the diagram below, there will be a slow 'zigzag' approach to the minimum.



In the case illustrated, the following algorithm will speed up convergence:

1/ i := 0; Guess α_0, β_0 .

2/ Find δ^{α} to minimize $s(\alpha_i + \delta^{\alpha},\beta_i)$.

3/ Find δ^{β} to minimize $s(\alpha_i + \delta^{\alpha},\beta_i + \delta^{\beta})$.

4/ Find w to minimize $s(\alpha_{i+1},\beta_{i+1})$,

where $\alpha_{i+1} = \alpha_i + w\delta^{\alpha}, \beta_{i+1} = \beta_i + w\delta^{\beta}$.

5/ i := i + 1 .

6/ Go back to 2/ .

In the simple case of a quadratic function $s(\alpha,\beta)$, this algorithm will find the minimum in one cycle.

The same idea can be used in our more general problem. If $s(\underline{\alpha}, \underline{\beta}, \underline{\gamma})$ is the sum of squares of residuals, we find $\underline{\delta}^{\alpha}$ to minimize $s(\underline{\alpha}+\underline{\delta}^{\alpha},\underline{\beta},\underline{\gamma})$,
then δ^{β} to minimize $s(\underline{\alpha} + \underline{\delta}^{\alpha},\beta + \underline{\delta}^{\beta},\underline{\gamma})$,
then $\underline{\delta}^{\gamma}$ to minimize $s(\underline{\alpha} + \underline{\delta}^{\alpha},\underline{\beta} + \underline{\delta}^{\beta},\underline{\gamma} + \underline{\delta}^{\gamma})$,
then w to minimize $s(\underline{\alpha}',\underline{\beta}',\underline{\gamma}')$ where $\alpha' = \alpha + w\delta^{\alpha}$ etc.
Since

$$s(\alpha',\beta',\gamma') = \sum_{i,j,k}\left[\sum_{t}(\alpha_{it} + w\delta^{\alpha}_{it})(\beta_{jt} + w\delta^{\beta}_{jt})(\gamma_{kt} + w\delta^{\gamma}_{kt})\right]^2, \qquad (5.21)$$

we can express s as a sixth degree polynomial in w, and then w can be chosen to minimize this polynomial (globally).

6. Search for New Algorithms by the Least Squares Method

A program was written to try to find a solution of (5.03) using the least-squares approach described in Sec. 5. Although it would be interesting to look for complex solutions, only the real case was considered.

The positive definite symmetric matrix A^TA is found from (5.13) and $A^T\underline{b}$ is found from (5.14), taking advantage of the identity.

$$\sum_{k,L,m,n} \beta_{kLu}\gamma_{mnu}\delta_{ni}\delta_{jk}\delta_{Lm} = \sum_{L} \beta_{jLu}\gamma_{Liu} \quad . \tag{6.01}$$

6.1 Calculation of $s(\underline{\alpha},\underline{\beta},\underline{\gamma})$

We shall use two or three subscripts on the a, β and y as convenient. The sum of squares of residuals of (5.06) is

$$s(\underline{\alpha},\underline{\beta},\underline{\gamma}) = \sum_{i,j,k} \left[\sum_{t} \alpha_{it}\beta_{jt}\gamma_{kt} - \sigma_{ijk} \right]^2 , \tag{6.11}$$

so

$$\begin{aligned} s(\alpha,\beta,\gamma) = & \sum_{i,j,k} \left[\sum_{t} \alpha_{it}\beta_{jt}\gamma_{kt} \right]^2 \\ & - 2 \sum_{i,j,k} \left(\sigma_{ijk} \sum_{t} \alpha_{it}\beta_{jt}\gamma_{kt} \right) \\ & + \sum_{i,j,k} \sigma_{ijk}^2 \end{aligned} \tag{6.12}$$

The straightforward evaluation of (6.11) for matrix multiplication with M = N = P takes $\sim 2N^6T$ operations (just counting multiplications). Using (6.12) instead, the last two terms give no problems, in fact

$$\sum_{i,j,k} \sigma_{ijk}^2 = \sum_{i,j,k,L,m,n} (\delta_{ij}\delta_{kL}\delta_{mn})^2 = M.N.P \tag{6.13}$$

and $$\sum_{i,j,k} \sigma_{ijk} \sum_{t} \alpha_{it}\beta_{jt}\gamma_{kt}$$

$$= \sum_{i,j,k,L,m,n,t} \alpha_{ijt}\beta_{kLt}\gamma_{mnt}\delta_{ni}\delta_{jk}\delta_{Lm}$$

$$= \sum_{i,j,L,t} \varepsilon_{ijt}\beta_{jLt}\gamma_{Lit} \quad , \tag{6.14}$$

and the evaluation of (6.14) requires only $\sim 2N^3T$ operations. The first term in (6.12) is

$$\sum_{i,j,k} \left(\sum_{t} \alpha_{it}\beta_{jt}\gamma_{kt}\right)^2 = \sum_{t,u} \left[\left(\sum_{i} \alpha_{it}\alpha_{iu}\right)\left(\sum_{j} \beta_{jt}\beta_{ju}\right)\left(\sum_{k} \gamma_{kt}\gamma_{ku}\right)\right] , \tag{6.15}$$

and the right side of (6.15) involves $\sim 3N^2T^2/2$ operations (50% are saved by symmetry). Since we are interested in values of $T \sim N^{2.8}$, s can be found from (6.12) -(6.15) in $\sim 3N^{7.6}/2$ operations instead of $\sim 2N^{8.8}$ using (6.11) . Hence it is much faster to use (6.12) - (6.15), although this involves some loss of accuracy.

6.2 Quadratic Approximation

At first the coefficients of w in the sixth degree polynomial p(w) of (5.21) were calculated using $\underline{\alpha}$, $\underline{\beta}$, $\underline{\gamma}$, $\underline{\delta}^{\alpha}$, $\underline{\delta}^{\beta}$ and $\underline{\delta}^{\gamma}$, and the global minimum of p(w) was found. Evaluation of the coefficients of p(w) was rather time-consuming, and it was noticed that the minimum usually occurred for $1 < w < 2$, and in this range p(w) was approximated very well by the quadratic fitting p(0), p(1) and p(2).

Since $p(0) = s(\underline{\alpha},\underline{\beta},\underline{\gamma})$ is already known,

and both $p(1) = s(\underline{\alpha}+\underline{\delta}^{\alpha}, \underline{\beta}+\underline{\delta}^{\beta}, \underline{\gamma}+\underline{\delta}^{\gamma})$

and $p(2) = s(\underline{\alpha}+2\underline{\delta}^{\alpha}, \underline{\beta}+2\underline{\delta}^{\beta}, \underline{\gamma}+2\underline{\delta}^{\gamma})$ may be found by the method of Sec. 6.1, the program was speeded up considerably by using the quadratic approximation, and the rate of convergence was not noticeably diminished.

As a precaution, necessary for the first few iterations anyway, w was constrained to lie in [1, 3] . Once w was chosen, $s(\underline{\alpha}+w\underline{\delta}^{\alpha}, \underline{\beta}+w\underline{\delta}^{\beta}, \underline{\gamma}+w\underline{\delta}^{\gamma})$ was computed (using previously calculated inner-products like $\sum_i \alpha_{iu}\alpha_{iv}$), and a check made that it was less than p(1) and p(2) . After the first few iterations these precautions usually turned out to be unnecessary. Note that, once $s_x = \sum_i (\alpha_{iu} + x\delta_{iu}^{\alpha})(a_{iv} + x\delta_{iv}^{\alpha})$ is found for x = 0, 1 and 2, we can find any s_x from

$$s_{x2} = \tfrac{1}{2}((y^2 - y)s_0 + (2 - 2y^2)s_1 + (y^2 + y)s_2), \text{ where } y = x - 1 .$$

This device was also used to save some-time. There is a danger of numerical instability unless $|\frac{1}{2}(y^2 - y)| \leq 1$, i.e. unless $0 \leq x \leq 3$, which is one reason why w was constrained to lie in [1, 3] .

I f M = N = P, the number of operations (just counting multiplications) per complete cycle is $\sim(15N^2+T)T^2/2$. Since $N^2 \leq T \leq N^3$ for the cases of interest, this grows very rapidly with N. On the other hand, we are trying to solve N^6 nonlinear equations in $3N^2T$ unknowns, so it would be surprising if any other method could do much better.

6.3 Summary of Results

The attempt to lower the bound (2.36) was unsuccessful, but some interesting negative results were obtained. For 2 x 2 matrices, many solutions were found with T = 7, but s never fell below 1 for T = 6, strongly indicating that Strassen's method gives the minimal number of multiplications for 2 x 2 matrices (at least for real $\underline{\alpha}$, $\underline{\beta}$ and $\underline{\gamma}$). With T = 7 each iteration took about 0.2 sec. and convergence was fairly fast, and appeared to be linear.

Trying T = 1, 2, . . . 7 for 2 x 2 matrices, it was found that

$$\inf(s) + T = \begin{cases} 7 \text{ if } T = 5, 6 \text{ or } 7 \\ 8 \text{ if } T = 1, 2 \text{ or } 3 \\ 7.59 \text{ if } T = 4 . \end{cases}$$

Thus the minimal sum of squares of residuals is usually integral, but appears to be nonintegral for T = 4.

3 x 3 matrices may be multiplied in 26 multiplications by using Strassen's method on a 2 x 2 submatrix-. It appears that there is also a solution with T = 25: the program (taking 3 sec./iteration) reduced s to 0.183 in 33 iterations, and s was still slowly decreasing. Knuth has found a solution, involving 'cube roots of unity, with T = 24. However, $\log_3 24 > \log_2 7$, and in fact $\log_3 21 < \log_2 7 < \log_3 22$, so a solution with $T \leq 21$ is necessary to improve the bound (2.36). When the program was run with T = 21, s appeared to be tending to 2 rather than to zero. If the rule $\inf(s(T)) + T > T_{min}$, which was observed for the 2 x 2 case, holds generally, this would indicate that for 3 x 3 matrices $T_{min} < 23$.

For 4 x 4 matrices the program was run with T = 48, to try to improve on Strassen's 49. Unfortunately, each iteration took 18 sec., and convergence was slow, so lack of computer time forced a return to smaller problems.

Various cases of small rectangular matrices were investigated. For example, the program was run with M = P = 2, N = 4 and with M = P = 4, N = 2 . In these cases the smallest T for which s appeared to be tending to zero was exactly the T to be expected by partitioning the matrices and

applying Strassen's method. Convergence often slowed as s approached 1, and speeded up again once $s < 1$, and there was no case in which $s < 1$ was attained, but for which s failed to tend to zero. Perhaps $s(\underline{\alpha}, \underline{\beta}, \underline{\gamma})$ has some local minima or saddle points, but they all have $s \geq 1$.

To summarize the results: although nothing has been rigorously proved, it appears likely that, to improve on the bound (2.36), matrices of size at least 4 x 4 must be investigated. It is plausible that there are no (real) methods better than Strassen's for the 2 x 2 or 3 x 3 case, and if this is so it is unlikely that any new method could be of much practical use, although it would certainly be of theoretical interest. A practical method needs to have rational $\underline{\alpha}$, $\underline{\beta}$ and $\underline{\gamma}$, and to be fast for reasonably small matrices most of the components of $\underline{\alpha}$, $\underline{\beta}$ and $\underline{\gamma}$ should vanish.

7. Conclusion

While the normal method takes $O(n^3)$ operations to multiply n x n matrices, Strassen's method shows that $O(n^{2.8})$ suffice. In practice, though, the normal method is faster for $n < 100$. Winograd's method, while still taking $O(n^3)$ operations, trades multiplications for additions and is definitely faster than the normal method for moderate and large n, with a. gain of up to about 10% for real matrices and up to about 20% for complex matrices. The gain would be greater for double or multiple-precision arithmetic.

Floating-point error bounds can be given for Strassen's and Winograd's methods, and the bounds are comparable to those for the normal method if the same precision arithmetic is used. With Winograd's method the necessity for prescaling can not be emphasized too strongly (see also [12]).

Provided scaling is used, Winograd's method can be recommended, especially in the complex case, unless very high accuracy is essential. It is much easier to code than Strassen's method. Possibly Strassen's method would be preferable when working with large matrices on a paged machine.

Attempts to lower the constant $\log_2 7 = 2.8...$ given by Strassen's method were unsuccessful. A completely new approach seems necessary in order to bring the upper and lower bounds on the computational complexity of matrix multiplication much closer together. For matrices of reasonable size, though, it seems unlikely that any new method could be very much faster than the known methods on a. serial computer.

Acknowledgement

I would like to thank R. Floyd and J. Herriot for their helpful advice, and CSIRO (Australia) for its generous financial support.

References

1. Floyd, R. W. Unpublished notes.

2. Fox, B. L. "Accelerating LP Algorithms", CACM 12, 7 (July 1969), 384 - 385.

3. Knuth, D. E. "The Art of Computer Programming", Vol. II, "Seminumerical Algorithms", Addison Wesley, 1969.

4. Mc Kellar, A. C. & Coffman, E. G. "Organizing Matrices and Matrix Operations for Paged Memory Systems", CACM 12, 3 (March 1969) 153 - 165.

5. Strassen, V. "Gaussian Elimination is Not Optimal", Numer. Math. 13, 354 - 356 (1969).

6. Wilkinson, J. H. "Rounding Errors in Algebraic Processes", H.M.S.O., 1963.

7. Winograd, S. "A New Algorithm for Inner-product", IEEE Trans. C-17 (1968), 693 - 694.

8. Winograd, S. Unpublished communication.

9. Randell, B. & Kuehner, J. "Dynamic Storage Allocation Systems", CACM 11, 5 (May 1968), 297 - 306.

10. Wirth, N. & Hoare, C. "A Contribution to the Development of ALGOL", CACM 9, 6 (June 1966), 413 - 431.

11. Bauer, H. & Becker, S. & Graham, S. "ALGOLW Implementation", Tech. Report No. CS98, Computer Science Department, Stanford Uni., (May 1968).

12. Brent, R. P. "Error Analysis of Algorithms for Matrix Multiplication and Triangular Decomposition Using Winograd's Identity', to appear.

A P P E N D - I X

ALGOLW procedures and test program

```
0001 1-     BEGIN COMMENT:
0002 --                 TEST PROGRAM FOR PROCEDURE STRASSEN, WITH NOGRAD & MATMULT,
0003 --                 FILE IS BRENT.TESTSTRASSEN ON SYS09;
0004 --
0005 --
0006 --     PROCEDURE STRASSEN (REAL ARRAY A, B, C(*,*);
0007 --                         INTEGER VALUE M, N, P);
0008 2-       BEGIN COMMENT :
0009 --                 IF A IS AN M X N MATRIX, AND B IS AN N X P MATRIX,
0010 --                 THEN THE M X P PRODUCT MATRIX A. B IS RETURNED IN C.
0011 --                 A MODIFIED FORM OF STRASSEN'S METHOD IS USED WHEN
0012 --                 M, N, AND P ARE SUFFICIENTLY LARGE.  IT IS BASED ON THE
0013 --                 FOLLOWING IDENTITIES WHICH HOLD IN THE 2X2 CASE:
0014 --
0015 --                 C11 = Q1 - Q3 - Q5 + Q7,
0016 --                 C12 = Q4 - Q1,
0017 --                 C21 = Q2 + Q3, AND
0018 --                 C22 = Q5 + Q6 - Q2 - Q4,      WHERE
0019 --                 Q1  = (A11 - A12).B22,
0020 --                 Q2  = (A21 - A22).B11,
0021 --                 Q3  = A22.(B11 + B21),
0022 --                 Q4  = A11.(B12 + B22),
0023 --                 Q5  = (A11 + A22).(B22-B11),
0024 --                 Q6  = (A11 + A21).(B11 + B12), AND
0025 --                 Q7  = (A12 + A22).(B21 + B22)
0026 --
0027 --                 A, B AND/OR C MAY BE IDENTICAL OR OVERLAPPING IN THE
0028 --                 CALL TO STRASSEN.  IN THE CASE M=N=P THE INTERMEDIATE
0122 --                 STORAGE REQUIRED IS ABOUT 8N**2/3 REAL WORDS.  THIS
0030 --                 COULD BE REDUCED TO N**2 (OR MORE GENERALLY
0031 --                 (MN + NP + PM)/3) BY BUILDING UP THE PRODUCT AFTER
0032 --                 EACH CALL TO STRASSEN IN EVENMULT, BUT THEN C COULD
0033 --                 NOT OVERLAP A OR B, AND THE PROCEDURE WOULD BE
0034 --                 RATHER SLOWER.
0035 --
0036 --                 IF 3MNP/(MN+NP+PM)<=NO THEN NORMAL MATRIX MULTIPLICATION
0037 --                 IS USED.  THIS IS BECAUSE STRASSEN'S IDENTITIES SAVE
0038 --                 TIME ONLY IF A MULTIPLICATION TAKES LONGER THAN 14
0033 --                 ADDITIONS, WHICH IS CERTAINLY FALSE FOR MATRICES SMALLER
0040 --                 THAN 14 X 14,  OR A LITTLE LARGER.  THE NUMBER NO
0041 --                 IS MACHINE AND COMPILER-DEPENDENT, BUT 100 IS ABOUT
0042 --                 OPTIMAL FOR ALGOLW ON THE 360/67 (WITH NO ARRAY BOUNDS
0043 --                 CHECKING).
0044 --
0045 --                 THE TIME FOR PROCEDURE STRASSEN IS ABOUT THE SAME AS
0046 --                 FOR THE NORMAL METHOD FOR SMALL M, N AND P, BUT FOR
0047 --                 LARGE M,N AND P THE TIME MULTIPLIES BY 7 (RATHER
0048 --                 THAN 8) EACH TIME M, N AND P ARE DOUBLED.  ACCURACY
0049 --                 IS NOT MUCH WORSE THAN FOR MATRIX MULTIPLICATION BY
0050 --                 THE USUAL METHOD WITH ALL OPERATIONS DONE IN SINGLE
0051 --                 PRECISION.
0052 --
0053 --                 R RRFNT, JULY 1969;
0054 --
0055 --       REAL PROCEDURE IP(REAL ARRAY A, B(*); INTEGER VALUE N);
0056 3-         BEGIN COMMENT:
0057 --                   RETURNS THE INNER PRODUCT OF THE N-VECTORS A AND B;
0058 --
0059 --         LONG REAL S;
0060 --         S := 0L;
0061 --         FOR I := 1 UNTIL N DO S := S + A(I)*B(I);
0062 --         ROUNDTOREAL(S)
0063 -3         END IP;
0664 --
0065 --       PROCEDURE EOP(REAL ARRAY A, B,C(*); INTEGER VALUE M1, M2, M3, F);
0066 3-         BEGIN COMMENT:
```

```
0067 --                          EFFECTIVELY DOES:
0068 --                          FOR I := 1 UNTIL M1 DO
0069 --                          A(I) := B(I + M2) + F*C(I + M3)
0070 --                          WHERE F = 0, +1 OR -1.
0071 --                          NOTE THAT IN ALGOLW 1-D ARRAY ACCESSES ARE MUCH
0072 --                          FASTER THAN 2-D ACCESSES;
0073 --
0074 --            IF F > 0 THEN
0075 4-              BEGIN IF M2 = 0 THEN
0076 5-                BEGIN IF M3 = 0 THEN
0077 6-                  BEGIN FOR I := 1 UNTIL M1 DO A(I):=B(I)+C(I)
0078 -6                  END
0079 --                ELSE FOR I := 1 UNTIL M1 DO A(I):=B(I)+C(I+M3)
0080 -5                END
0081 --              ELSE
0082 5-                BEGIN IF M3=0 THEN
0083 G-                  BEGIN FOR I := 1 UNTIL M1 DO A(I):=B(I+M2)+C(I)
0084 -G                  END
0085 --                ELSE FOR I := 1 UNTIL M1 DO A(I) := B(I + M2) + C(I + M3)
0086 -5                END
0087 -4              END
0088 --            ELSE IF F < 0 THEN
0089 4-              BEGIN IF M2 = 0 THEN
0090 5-                BEGIN IF M3 = 0 THEN
0091 G-                  BEGIN FOR I := 1 UNTIL M1 DO A(I):=B(I)-C(I)
0092 -6                  END
0093 --                ELSE FOR I := 1 UNTIL M1 DO A(I):=B(I)-C(I+M3)
0094 -5                END
0095 --              ELSE
0096 5-                BEGIN IF M3 = 0 THEN
0097 6-                  BEGIN FOR I := 1 UNTIL M1 DO A(I):=B(I+M2)-C(I)
0098 -G                  END
0099 --                ELSE FOR I := 1 UNTIL M1 DO A(I):=B(I+M2)-C(I+M3)
0100 -5                END
0101 -4              END
0102' --           ELSE
0103 4-              BEGIN IF M2 = 0 THEN
0104 5-                BEGIN FOR I := 1 UNTIL M1 DO A(I):=B(I)
0105 -5                END
0106 --              ELSE FOR I := 1 UNTIL M1 DO A(I):=B(I+M2)
0107 -4              END
0108 -3            END OP;
0109 --
0110 --
0111 --          COMMENT: IF M, N, OR P SMALL USE NORMAL MATRIX MULTIPLICATION.
0112 --                   THE CONSTANT NO MENTIONED ABOVE IS REDUCED TO 29 FOR
0113 --                   CHECKING PURPOSES;
0114 --
0115 --
0116 --          IF (3*M*N*P) <= (29*(M*N + N*P + P*M)) THEN
0117 --
0118 3-            BEGIN COMMENT: WE USE A TEMPORARY ARRAY Q IN CASE C = A OR B;
0119 --            REAL ARRAY Q (1 : : M, 1 : : P);
0120 --            FOR I := 1 UNTIL M DO FOR J := 1 UNTIL P DO
0121 --            Q(I,J) := IP(A(I,*), B(*,J), N);
0122 --            FOR I := 1 UNTIL MD O OP(C(I,*),Q(I,*),Q(I,*),P, 0, 0, 0)
0123 -    3       END
0124 --
0125 --          ELSE
0126 --
0127 3-            BEGIN COMMENT: USE STRASSEN'S METHOD;
0128 --
0129 --            PROCEDURE IDENTITIES;
0130 4-              BEGIN COMMENT:
0131 --                          THE IDENTITIES ARE PUT HERE TO AVOID SEGMENT
0132 --                          OVERFLOW;
```

```
0133 --
0134 --          REAL ARRAY T(1 :: M2, 1 :: N2);
0135 --          REAL ARRAY U(1 :: N2, 1 :: P2);
0136 --          REAL ARRAY Q1, Q2, Q3, Q4, Q5, Q6 (1 :: M2, 1 :: P2);
0137 --
0138 --          FOR J := 1 UNTIL N2 DO
0139 --          OP   (T(*, J), A(*, J), A(*, J + N2), M2, 0, 0, -1);
0140 --          FOR I := 1 UNTIL N2 DO
0141 --          OP   (U(I,*), B(I + N2, *), B(I,*), P2, P2, 0, 0);
0142 --          STRASSEN(T, U, Q1, M2, N2, P2);
0143 --          FOR I := 1 UNTIL M2 DO
0144 --          OP   (T(I,*), A(I + M2, *), A(I + M2, *), N2, 0, N2, -1);
0145 --          STRASSEN(T, B, Q2, M2, N2, P2);
0146 --          FOR I := 1 UNTIL M2 DO
0147 --          OP   (T(I,*), A(I + M2, *), A(I,*), N2, N2, 0, 0);
0148 --          FOR I := 1 UNTIL N2 DO
0149 --          OP   (U(I,*), B(I,*), B(I + N2,*), P2, 0, 0, 1);
0150 --          STRASSEN(T, U, Q3, M2, N2, P2);
0151 --          FOR J := 1 UNTIL P2 DO
0152 --          OP   (U(*, J), B(*, J + P2), B(*, J + P2), N2, 0, N2, 1);
0153 --          STRASSEN (A, U, 44, M2, N2, P2);
0154 --          FOR I := 1 UNTIL M2 DO
0155 --          OP   (T(I,*), A(I,*), A(I + M2, *), N2, 0, N2, 1);
0156 --          FOR I := 1 UNTIL N2 DO
0157 --          OP   (U(I,*), B(I + N2, *), B(I,*), P2, P2, 0, -1);
0158 --          STRASSEN(T, U, Q5, M2, N2, P2);
0159 --          FOR I := 1 UNTIL M2 DO
0160 --          OP   (T(I,*), A(I,*), A(I + M2, *), N2, 0, 0, 1);
0161 --          FOR J := 1 UNTIL P2 DO
0162 --          OP   (U(*, J), B(*, J), B(*, J + P2) N2 ,0 ,0 1);
0163 --          STRASSEN (T, U, Q6, M2, N2, P2);
0164 --          FOR J := 1 UNTIL N2 DO
0165 --          OP (T(*, J), A(*, J + N2), A(*, J + N2), M2, 0, M2, 1);
0166 --          FOR I := 1 UNTIL N2 DO
0167 --          OP (U(I,*), B(I + N2, *), B(I + N2, *), P2, 0, P2, 1);
0168 --          STRASSEN(T, U, C, M2, N2, P2);
0169 --
0170 --          FOR I := 1 UNTIL M2 DO FOR J := 1 UNTIL P2 DO
0171 5-            BEGIN
0172 --            C (I, J)      := Q1(I, J) - Q3 (I, J) + C (I, J) - Q5(I, J);
0173 --            C (I, J + P2) := Q4(I, J) - Q1(I, J);
0174 --            C (I + M2, J) := Q2(I, J) + Q3(I, J);
0175 --            C (I+M2, J+P2) := Q5(I, J) + Q6(I, J) - (Q2(I, J) + Q4(I, J))
0176 -5           END
0177 -4         END IDENTITIES;
0178 --
0179 --       REAL ARRAY S1(1:: P);
0180 --       REAL ARRAY S2(1 :: M);
0181 --       INTEGER M2, N2, P2;
0182 --      M2 := M DIV 2; N2 := N DIV 2; P2 := P DIV 2;
0183 --
0184 --       COMMENT:  THIS PART MUST BE DONE NOW IN CASE C = A OR B;
0185 --
0186 --       IF (2*M2) < M THEN
0187 4-          BEGIN FOR J := 1 UNTIL 2*P2 DO
0188 --           S1(J) := IP(A(M,*), B(*, J), N)
0189 - 4          END;
0190 --
0191 --       IF (2*P2) < P THEN
0192 4-          BEGIN FOR I := 1 UNTIL M DO
0193 --           S2(I) := IP(A(I,*), B(*,P), N)
0194 -4           END;
0195 --
0196 --       IDENTITIES;
0197 --
0198 --       f12 := 2*M2; N2 := 2*N2; P2 := 2*P2;
```

```
0199 --
0200 --          COMMENT :   IF M,N, OR P WAS ODD WE HAVE TO FI X UP THE BORDERS;
0201 --
0202 --          IF N2 < N THEN
0203 4-            BEGIN
0204 --            FOR I := 1 UNTIL M2 DO FOR J := 1 UNTIL P2 DO
0205 --            C(I,J) := C(I,J) + A(I,N)*B(N,J)
0206 -4            END;
0207 --
0208 --          IF M2 <M THEN
0209 4-            BEGIN F O R J:= 1 UNTIL P2 DO C(M,J):= S1(J)
0210 -4            END;
0211 --
0212 --          IF P2 < P THEN
0213 4-            BEGIN FOR I := 1 UNTIL M D O C(I,P):=S2(I)
0214 -4            END
0215 -3          END
0216 -2        END STRASSEN;
0217 --
0218 --
0219 --      PROCEDURE WINOGRAD(REAL ARRAY A,B,C(*,*); INTEGER VALUE M, N, P);
0220 2-        BEGIN COMMENT:
0221 --                      IF A IS AN M X N MATRIX AND B AN N X P MATRIX, THEN
0222 --                      THEIR PRODUCT A.B IS RETURNED IN C.  WINOGRAD'S METHOD
0223 --                      IS USED WITH PRESCALI NG TO ENSURE GOOD ACCURACY;
0224 --
0225 --          REAL PROCEDURE WP(REAL ARRAY A,B(*); LONG REAL VALUE X,Y);
0226 3-            BEGI N COMMENT:
0227 --                       RETURNS THE INNER PRODUCT OF THE N-VECTORS A AND B,
0228 --                       USING PRECOMPUTED X AND Y.   N IS GLOBAL;
0229 --
0230 --            LONG REAL S;
0231 --            S :=  -(X + Y);
0232 --            COMMENT:   IF THE NEXT STATEMENT IS REPLACED BY:
0233 --            FOR I := 2 STEP 2 UNTIL 2*(NDIV2) D O
0234 --            S :=  S +(LONG(A(I-1)) + LONG(B(I)))*(LONG(A(I)) + LONG(B(I-1))).,
0235 --            THEN THE CORRECTLY ROUNDED SINGLE-PRECISION RESULT IS USUALLY
0236 --            RETURNED (ASSUMING PRESCALING).  UNFORTUNATELY TH I S SLOWS DOWN
0237 --            THE ALGORITHM SD THAT IT IS NO LONGER FASTER THAN THE USUAL ONE;
0238 --            FOR I :=2 STEP 2 UNTIL 2*(N DIV 2) D O
0239 --            S :=  S  +(A(I -   1) + B(I))*(A(I) + B(I - 1));
0240 --            IF (N REM 2)> 0 THEN S := S +A(N)*B(N);
0241 --            ROUNDTOREAL(S)
0242 -3            END WP;
0243 --
0244 --          LONG REAL PROCEDURE XI(REAL ARRAY A(*));
0245 3-            BEGI N COMMENT: .
0246 --                       USED TO PRECOMPUTE THE FUNCTIONS OF A REQUIRED B Y WP;
0247 --
0248 --            LONG REAL S;
0249 --            S := 0L;
0250 --            FOR I :=  1  STEP  2  UNTIL  N  -  1  DO S := S +A(I)*A(I+1);
0251 --            S
0252 -3            END XI;
0253  --
0254 --          PROCEDURE MAX (REAL ARRAY A(*); REAL VALUE RESULT BD);
0255 --          FOR I := 1 UNTIL N DO IF BD <ABS(A(I)) THEN BD:=ABS(A(I));
025G --
0257 --          PROCEDURE MUL(REAL ARRAY A,B(*); REAL VALUE M);
0258 --          FOR I :=  1  UNTIL N DO A(I):=M*B(I);
0259 --
0260 --          REAL AMAX, BMAX, MULT;
0261 --          COMMENT:   THE ARRAYS D AND E ARE USED AS TEMPORARY STORAGE IN CASE
0262 --                     SOME OF A, B A N D C COINCIDF;
0263 --          REAL  ARRAY  D(1 :: M, 1  ::  N);
0264 --          REAL ARRAY E(1  ::  N, 1  ::  P);
```

```
0265 --          LONG REAL ARRAY X(1 :: M);
0266 --          LONG REAL ARRAY Y(1 :: P);
0267 --
0268 --          COMMENT:  A AND B ARE SCALED BY SUITABLE POWERS OF TWO TO GIVE GOOD
0269 --                    NUMERICAL PROPERTIES, AND THE SCALED MATRICES STORED IN
0270 --                    D AND E;
0271 --          AMAX := BMAX := 0.0;
0272 --          FOR I := 1 UNTIL M DO MAX(A(I,*),AMAX);
0273 --          FOR K := 1 UNTIL P DO MAX(B(*,K),BMAX);
0274 --          MULT := IF (AMAX>0) AND (BMAX>0)    THEN
0275 --                  2**(TRUNCATE((LOG(BMAX) - LOG(AMAX))/LOG(4) + 200.5) - 200)
0276 --                  ELSE 1.0;
0277 --          FOR I := 1 UNTIL M DO MUL(D(I,*),A(I,*),MULT);
0278 --          FOR K := 1 UNTIL P DO MUL(E(*,K),B(*,K),MULT);
0279 --          COMMENT:  NOW SOME CONSTANTS ARE PRECOMPUTED AND SAVED IN X AND Y;
0280 --          FOR I := 1 UNTIL M DO X(I):=XI(D(I,*));
0281 --          FOR K := 1 UNTIL P DO Y(K):=XI(E(*,K));
0282 --          COMMENT:  NOW THE INNER PRODUCTS ARE FOUND;
0283 --          FOR I := 1 UNTIL M DO FOR J := 1 UNTIL P DO
0284 --          C(I,J) := WP(D(I,*), E(*,J), X(I), Y(J))
0285 -2          END WINOGRAD;
0286 --
0287 --
0288 --        PROCEDURE MATMULT (REAL ARRAY A,B,C(*,*);
0289 --                           INTEGER VALUE M, N, P);
0290 2-          BEGIN COMMENT:
0291 --                    FORMS C := A.B IN THE USUAL WAY;
0292 --
0293 --          REAL PROCEDURE IP(REAL ARRAY A,B(*); INTEGER VALUE N);
0294 3-            BEGIN COMMENT:
0295 --                      RETURNS THE INNER PRODUCT OF THE N-VECTORS A AND B;
0296 --
0297 --            LONG REAL S;
0298 --            S := 0L;
0299 --            FOR I := 1 UNTIL N DO S := S + A(I)*B(I);
0300 --            ROUNDTOREAL(S)
0301 -3            END IP;
0302 --
0303 --          PROCEDURE ASSIGN(REAL ARRAY A,B(*);INTEGER VALUE N);
0304 --          FOR I := 1 UNTIL N DO A(I):=B(I);
0305 --
0306 --          COMMENT:  Q IS USED IN CASE C COINCIDES WITH A OR B;
0307 --          REAL ARRAY Q(1 : : M, 1 : : P);
0308 --          FOR I := 1 UNTIL M DO FOR J := 1 UNTIL P DO
0309 --          Q(I,J) := IP(A(I,*), B(*,J), N);
0310 --          FOR I := 1 UNTIL M DO ASSIGN(C(I,*),Q(I,*),P)
0311 -2          END MATMULT;
0312 --
0313 --
0314 --        INTEGER RAN1,RAN2,RAN3,RAN4;
0315 --        INTEGER ARRAY RAN5(0 :: 255);
0316 --
0317 --        PROCEDURE RANINIT(INTEGER VALUE R1);
0318 2-          BEGIN COMMENT:
0319 --                    MUST BE CALLED WITH ANY INTEGER R1
0320 --                    TO INITIALIZE PROCEDURE RANDOM;
0321 --
0322 --          INTOVFL := NULL;  COMMENT:  MASKS OFF INTEGER OVERFLOW;
0323 --          RAN1 := 1 ;  RAN2 := 2*ABS(R1) + 1 ;
0324 --          FOR I := 0 UNTIL 255 DO RAN5(I):= RAN2:=RAN2*65539
0325 -2          END RANINIT;
0326 --
0327 --        REAL PROCEDURE RANDOM;
0328 2-          BEGIN COMMENT:
0329 --                    USES TWO SIMPLE LEHMER GENERATORS OF THE FORM
0330 --                    X(N+1) = X(N)*A (MOD T) WITH
```

```
0331 --                          A1 = 11**11 (MOD T1) = 6435, T1=2**13-1= 8191,
0332 --                          A2 = 2**16+3 = 65539, T2 = 2**31 = 2147483643.
0333 --                          THE FIRST GENERATOR JUST POINTS TO THE TABLE OF
0334 --                          ENTRIES FOR THE SECOND GENERATOR, SO GOOD RANDOM
0335 --                          NUMBERSWITH A CYCLE LENGTH AT LEAST 2.10**12 ARE
0336 --                          PRODUCED.
0337 --                          THE IDEA IS DUE TO MACLAREN AND MARSAGLIA, SEE
0338 --                          KNUTH, VOL 2, PG 30, ALGORITHM M.
0339 --                          REAL OUTPUT UNIFORM IN (0,1).
0340 --                          NOTE THAT INTEGER RAN1, RAN2, RAN3, RAN4 AND
0341 --                          INTEGER ARRAY RAN5 (0::255) MUST BE DECLARED
0342 --                          GLOBALLY AND RANINIT MUST BE CALLED FOR
0343 --                          INITIALIZATION;
0344 --
0345 --          RAN1 := (RAN1*6435) REM 8191;
0346 --          RAN3 := RAN1 REM 256;
0347 --          RAN4 := RAN5 (RAN3);
0348 --          RAN2 := RAN5 (RAN3) := RAN2 * 65539;
0349 --          RAN4 * 0.465661287'.9
0350 -2        END RANDOM;
0351 --
0352 --
0353 --      PROCEDURE RANSET(REAL ARRAY A(*,*); INTEGER VALUE M, N);
0354 --        FOR I := 1 UNTIL M DO FOR J := 1 UNTIL N DO
0355 --        A(I,J) := RANDOM - 0.5;
0356 --
0357 --
0358 --      COMMENT : CALLING PROGRAM;
0359 --
0360 --      INTEGER R, M,N, P, T; REAL S, MAX, DEL, SW MAXW;
0361 --      READ(R);
0362 --      WHILE R ~= 0 DO
0363 2-      BEGIN READON(M,N,P); RANINIT(R); WRITE(" "); WRITE(" ");
0364 --      WRITE("R", R, "                M", M, "                N", N,
0365 --                                    P", P);
0366 3-        BEGIN
0367 --        REAL ARRAY A(1 :: M, 1 :: N);
0368 --        REAL ARRAY B(1 :: N, 1 :: P);
0369 --        REAL ARRAY C, D, E(1:: M, 1 :: P);
0370 --        RANSET ( A ,M, N);
0371 --        RANSET (B, N, P);
0372 --        T := TIME(1);
0373 --        MATMULT(A, B, D, M, N, P);
0374 --        WRITE ("MATMULT TIME ", TIME(1) - T);
0375 --        T := TIME(1);
0376 --        STRASSEN (A,B,C,M,N,P);
0377 --        WRITE ("STRASSEN TIME",.TIME(1)-T); T:=TIME(1);
0378 --        WINOGRAD(A,B,E, EI,N,P);
0379 --        WRITE("WINOGRAD TIME", TIME(1) - T);
0380 --        S := MAX := SW := MAXW := 0;
0381 --        FOR I := 1 UNTIL M DO FOR J := 1 UNTIL P DO
0382 4-          BEGIN DEL := ABS(C(I,J) - D(I,J));
0383 --          IF MAX < DEL THEN MAX := DEL;
0384 --          S := S +DEL*DEL;
0385 --          DEL := ABS(D(I,J) - E(I,J));
0386 --          IF MAXW < DEL THEN MAXW:= DEL;
0387 --          SW := SW + DEL*DEL
0388 -4           END;
0389 --        WRITE("S ", S , "          MAX", MAX );
0390 --        WRITE("SW", SW, "         MAXW", MAXW);
0391 --        READ(R)
0392 -3         END
0393 -2       END
0-394 -1     END.
```