

AD-A049 493

WISCONSIN UNIV MADISON MATHEMATICS RESEARCH CENTER
EFFICIENT COMPUTER MANIPULATION OF TENSOR PRODUCTS.(U)
NOV 77 C DE BOOR
MRC-TSR-1810

F/G 12/1

DAA629-75-C-0024

NL

UNCLASSIFIED

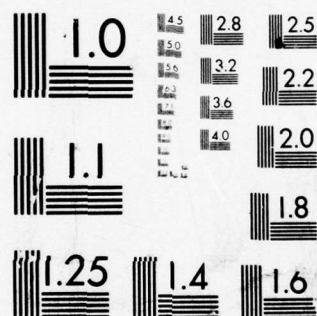
| OF |
AD
A049493



END
DATE
FILMED

3 - 78

DDC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A049493

HQ INU.
JDC FILE COPY

(13) 2 (13)

MRC Technical Summary Report #1810

EFFICIENT COMPUTER MANIPULATION
OF TENSOR PRODUCTS

Carl de Boor

Mathematics Research Center
University of Wisconsin-Madison
610 Walnut Street
Madison, Wisconsin 53706

See 1473
in
Book

November 1977

(Received October 6, 1977)

Approved for public release
Distribution unlimited

Sponsored by

U.S. Army Research Office
P. O. Box 12211
Research Triangle Park
North Carolina 27709

DDC
RECEIVED
FEB 2 1978
RECEIVED

ACCESSION NO.	
NTIS	Write Section ☒
DDC	Edit Section ☐
UNANNOUNCED	☐
JUSTIFICATION	
BY	
DISTRIBUTION/AVAILABILITY CODES	
Dist.	AVAIL. AND/OR SPECIAL
A	

UNIVERSITY OF WISCONSIN - MADISON
MATHEMATICS RESEARCH CENTER

EFFICIENT COMPUTER MANIPULATION
OF TENSOR PRODUCTS

Carl de Boor

Technical Summary Report #1810
November 1977

ABSTRACT

It is shown how to construct a modified version SUB'_i of a (presumably efficient) subroutine SUB_i for solving the linear system $A_i x = b$, $i=1, \dots, k$, so that the linear system

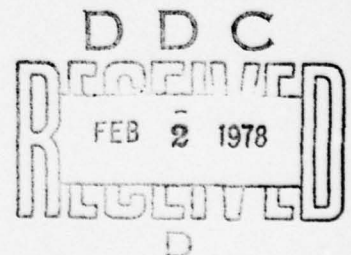
$$(A_1 \otimes \dots \otimes A_k) \underline{x} = \underline{b}$$

can be solved by just one call to each of the routines SUB'_i , $i=1, \dots, k$. Polynomial interpolation and spline interpolation in several variables are given as examples.

AMS(MOS) Subject Classifications: 65D15, 41A63.

Key Words: Mathematical Software, Tensor product, Multivariate, Interpolation, Approximation, Polynomial, Osculatory.

Work Unit Number 7 - Numerical Analysis.



SIGNIFICANCE AND EXPLANATION

The tensor product of linear approximation schemes is a very convenient mathematical construct to make efficient multivariate approximation schemes out of univariate ones. For example, we might know that the coefficient vector $\underline{a}$ for the polynomial

$$p(x) = \sum_i a_i p_i(x)$$

which agrees with the function f at $t_1, \dots, t_n$ can be computed as

$$\underline{a} = B_{\underline{t}} (f(t_i)) .$$

Then we know that

$$p(x,y) = \sum_i \sum_j a_{ij} p_i(x) p_j(y)$$

is a polynomial which agrees with the function f at all the points (t_i, u_j) of a rectangular grid provided the coefficient matrix (a_{ij}) is computed from the data matrix $(f(t_i, u_j))$ by

$$(a_{ij}) = (B_{\underline{t}} \otimes B_{\underline{u}}) (f(t_i, u_j)) := B_{\underline{t}} (f(t_i, u_j)) (B_{\underline{u}})^T .$$

Such a construction can also be used to obtain interpolants to functions of three or more variables, but explicit expressions for

$$(B_{\underline{t}} \otimes B_{\underline{u}} \otimes \dots \otimes B_{\underline{z}}) (f(t_i, u_j, \dots, z_n))$$

in terms of ordinary matrix multiplication are harder to come by and implement.

The report proposes a simple way of modifying a subroutine for calculating the coefficient vector $\underline{a}$ from univariate data so that the construction of a tensor product interpolant reduces to calling these modified univariate routines in sequence. Somewhat surprisingly, this idea seems not to have been noticed before. It makes the construction and evaluation of tensor product approximants an easy thing.

The responsibility for the wording and views expressed in this descriptive summary lies with MRC, and not with the author of this report.

EFFICIENT COMPUTER MANIPULATION

OF TENSOR PRODUCTS

Carl de Boor

In [3], V. Pereyra and G. Scherer discuss the numerical solution of a linear system of the form

$$(A_1 \otimes \dots \otimes A_k) \underline{x} = \underline{b} \quad (1)$$

with A_i an invertible matrix of order n_i , $i=1, \dots, k$, and, correspondingly, both $\underline{x}$ and $\underline{b}$ k -dimensional arrays, of size $n_1 \times n_2 \times \dots \times n_k$. Such systems arise naturally when forming tensor products of univariate interpolation schemes.

Pereyra and Scherer propose to store arrays such as $\underline{x}$ and $\underline{b}$ with the last index running fastest and then have a scheme of applying A_k^{-1} , A_{k-1}^{-1} and so on down to and including A_1^{-1} , appropriately restoring the intermediate information so that application of A_i^{-1} involves only repeated ordinary matrix multiplication to a vector stored in consecutive locations in memory. When, as is more reasonable, application of $U_i^{-1} L_i^{-1}$ rather than of A_i^{-1} is wanted, with $L_i U_i$ a triangular factorization for A_i , a further complication arises and is dealt with.

It is the purpose of this note to describe a different procedure which I have used for some time and which is more direct and simpler than the Pereyra-Scherer procedure appears to be.

We assume that, for each i , we have available a Fortran subroutine

$$\text{SUB}_i(b, n, x)$$

which solves the i -th linear system $A_i x = b$ (of order $n = n_i$) for x , given b . Presumably, the routine does this in an efficient way, taking advantage of any special structure A_i might have such as bandedness, positive definiteness etc.

We further assume that the k-dimensional arrays $\underline{x}$ and $\underline{b}$ are (to be) stored in Fortran fashion, i.e.,

$$\underline{x}(i_1, i_2, \dots, i_k) = \underline{x}(i_1 + n_1(i_2 - 1 + n_2(i_3 - 1 + \dots + n_{k-1}(i_k - 1) \dots)))$$

if we refer to $\underline{x}$ also as an equivalent one-dimensional array.

The following simple procedure will then lead to an efficient way for solving (1): For each i , enlarge the subroutine SUB_i to a subroutine

$\text{SUB}_i'(b, n, m, x)$

which solves simultaneously $A_i x = b$ for m given right sides $b(*, 1), b(*, 2), \dots, b(*, m)$, each of length $n = n_i$, and stores the corresponding solutions in $x(1, *), x(2, *), \dots, x(m, *)$. Thus, the dimension statement for the arguments b and x in SUB_i' reads

DIMENSION $b(n, m), x(m, n)$

and the change otherwise consists in putting every statement involving b or x appropriately into a DO loop. In this, care should be taken to leave statements which do not depend on the particular right side outside such loops.

Many library routines for specific linear problems already provide this facility for dealing with several right sides in one call, since the work in solving $A_i x = b$ for an additional right side b is usually much less than the work for solving such a system the first time. But such routines return the solution corresponding to the j -th column $b(*, j)$ of the input array customarily in the j -th column of the output array x and not, as I propose here, in the j -th row.

Lemma. For $i=1, \dots, k$, let SUB_i' be an expanded version, as described, of the routine SUB_i for solving $A_i x = b$, and set

$$N := n_1 * n_2 * \dots * n_k .$$

Then, the following statements

$\underline{b}_0 := \underline{b}$

CALL SUB₁'($\underline{b}_0, n_1, N/n_1, \underline{b}_1$)

CALL SUB₂'($\underline{b}_1, n_2, N/n_2, \underline{b}_2$)

.

CALL SUB_k'($\underline{b}_{k-1}, n_k, N/n_k, \underline{b}_k$)

$\underline{x} := \underline{b}_k$

will produce the solution $\underline{x}$ of (1).

Proof. Let $\underline{x}_i$ be the k-dimensional array

$$\underline{x}_i := (A_1^{-1} \otimes \dots \otimes A_i^{-1} \otimes 1 \otimes \dots \otimes 1) \underline{b},$$

$i=0, \dots, k$. Then

$$\begin{aligned} \underline{x}_i(j_1, \dots, j_{i-1}, \dots, j_{i+1}, \dots, j_k) \\ = A_i^{-1} \underline{x}_{i-1}(j_1, \dots, j_{i-1}, \dots, j_{i+1}, \dots, j_k) \end{aligned} \quad (2)$$

and our assertion is proved if we can establish that $\underline{b}_k = \underline{x}_k$. We prove this by showing that, for all i , $\underline{b}_i$ as generated by the succession of calls above is related to $\underline{x}_i$ in the following way:

If $\underline{b}_i$ is interpreted as a k-dimensional Fortran array, of dimension $(n_{i+1}, \dots, n_k, n_1, \dots, n_i)$, then

$$\underline{b}_i(j_{i+1}, \dots, j_k, j_1, \dots, j_i) = \underline{x}_i(j_1, \dots, j_k), \text{ all } j, \quad (3)$$

for $i=0, \dots, k$. For $i=k$, (3) is indeed the desired statement that $\underline{b}_k = \underline{x}_k$.

Now, (3) holds for $i=0$ because of the initial assignment $\underline{b}_0 := \underline{b}$. Assuming (3) to hold for $i < v$, we consider the action of the

CALL SUB_v'($\underline{b}_{v-1}, n_v, N/n_v, \underline{b}_v$) .

SUB_v' considers $\underline{b}_{v-1}$ to be a two-dimensional array, $\underline{b}$ say, of dimension $(n_v, N/n_v)$.

Thus, with

$$s := j_{v+1} + n_{v+1}(j_{v+2}^{-1} + \dots + n_k(j_1^{-1} + \dots + n_{v-2}(j_{v-1}^{-1}) \dots)) \dots ,$$

we have

$$\begin{aligned} b(\cdot, s) &= \underline{b}_{v-1}(\cdot, j_{v+1}, \dots, j_k, j_1, \dots, j_{v-1}) \\ &= \underline{x}_{v-1}(j_1, \dots, j_{v-1}, \cdot, j_{v+1}, \dots, j_k) \end{aligned} \quad (4)$$

by induction hypothesis. SUB'_v then applies A_v^{-1} to each of these $m = N/n_v$ n_v -vectors $b(\cdot, s)$, thus obtaining the corresponding n_v -vector

$$\underline{x}_v(j_1, \dots, j_{v-1}, \cdot, j_{v+1}, \dots, j_k) ,$$

by (2) and (4), and stores this vector in

$$\begin{aligned} x(s, \cdot) &= \underline{b}_v(s + (N/n_v)(\cdot - 1)) \\ &= \underline{b}_v(j_{v+1} + n_{v+1}[j_{v+2}^{-1} + \dots + n_{v-2}(j_{v-1}^{-1}) \dots] + (N/n_v)(\cdot - 1)) \\ &= \underline{b}_v(j_{v+1}, \dots, j_k, j_1, \dots, j_{v-1}, \cdot) \end{aligned}$$

which proves (3) for $i = v$ and so advances the induction hypothesis;

Q.E.D.

We introduced the auxiliary arrays only for argument's sake. In calculations, two arrays, say $\underline{b}_1$ and $\underline{b}_2$, are sufficient, with $\underline{b}_1$ serving in place of all $\underline{b}_i$ with i odd, and $\underline{b}_2$ serving for all the others.

Also, in typical situations, the various subroutines $\text{SUB}_1, \dots, \text{SUB}_k$ are, in fact, just one routine called with additional arguments which differ with i . In such a case, only one extended version has to be written.

Finally, we put the above discussion in terms of solving a linear system, i.e., in terms of premultiplying a given vector by the inverse of a given matrix. We did this in order to make the point that we do not require the matrix by which we wish to premultiply to be present explicitly. Any Fortran subprogram $\text{SUB}_i(b, x)$ which has the effect of forming $x = B_i b$ for given b can serve as a basis for an extended version $\text{SUB}'_i(b, n, m, x)$ suitable for the calculation of $(B_1 \otimes \dots \otimes B_k) \underline{b}$, and the matrices B_i need not be square. We state this slight extension of the Lemma as a corollary for the record.

Corollary. For $i=1, \dots, k$, let B_i be a (n_i, r_i) -matrix, and let $SUB'_i(b, n_i, m, x, r_i)$ be a subroutine which, for $j=1, \dots, m$, forms the r_i -vector $B_i b(\cdot, j)$ (in some manner) from the n_i -vector $b(\cdot, j)$, and stores it in $x(j, \cdot)$. Then, the following statements

$\underline{b}_0 := \underline{b}$

$m := n_2 * \dots * n_k$

CALL $SUB'_1(\underline{b}_0, n_1, m, \underline{b}_1, r_1)$

$m := m * r_1 / n_2$

CALL $SUB'_2(\underline{b}_1, n_2, m, \underline{b}_2, r_2)$

$m := m * r_2 / n_3$

.

CALL $SUB'_k(\underline{b}_{k-1}, n_k, m, \underline{b}_k, r_k)$

$\underline{x} := \underline{b}_k$

form the k -dimensional array $\underline{x} = (B_1 \otimes \dots \otimes B_k) \underline{b}$.

It is not even necessary that B_i be a matrix, i.e., a two-dimensional array. The more general situation in which B_i is a linear map which associates s_i -dimensional arrays with t_i -dimensional arrays is covered by the corollary as well since we can always interpret such s_i -dimensional and t_i -dimensional arrays Fortran fashion as equivalent one-dimensional arrays.

We give some simple examples in the next section.

Tensor products of univariate interpolation schemes. The following material concerning tensor products of univariate interpolation schemes is well known and is mentioned here only in order to illustrate the use and usefulness of the simple idea expounded earlier. (A simple account giving proofs and details can be found, e.g., in [1].)

The construction of a (univariate) linear interpolant g to some function f usually involves the calculation of the coefficients $a = (a_i)$ in a representation

$$g = \sum_i a_i \varphi_i$$

for the interpolant from certain information $(\lambda_i f)$ about f . Here, each λ_i is a linear functional, e.g.,

$$\lambda_i f = f(x_i) \quad \text{or} \quad \lambda_i f = f^{(r_i)}(x_i) \quad \text{or} \quad \lambda_i f = \int \psi_i(x) f(x) dx \quad \text{etc.}$$

and g is so constructed that

$$\lambda_i g = \lambda_i f, \quad \text{all } i.$$

At the level of the present discussion, there is no reason to require the representation for g to be irredundant, i.e., to require the sequence (φ_i) to be linearly independent. All that is necessary is the assumption that

$$a = B(\lambda_i f)$$

for some matrix B . The matrix B is commonly not known explicitly (although it could, of course, be determined). Rather, some procedure or subprogram SUB is available which transforms the vector $(\lambda_i f)$ of data appropriately into the vector a of coefficients.

For example, consider the construction of the polynomial $p = p_f$ of degree $< n$ which agrees with f at the n distinct points $x_1, \dots, x_n$. In its Newton form, p_f looks like

$$p_f(x) = \sum_{i=1}^n [x_1, \dots, x_n] f \cdot \prod_{j=i+1}^n (x - x_j) \quad (5)$$

with the coefficient $[x_1, \dots, x_n] f$ the so-called divided difference for f at the points $x_1, \dots, x_n$, $i=1, \dots, n$, i.e.,

$$[x_1, \dots, x_j] f := \begin{cases} f(x_i) & , \quad i = j \\ ([x_{i+1}, \dots, x_j] f - [x_1, \dots, x_{j-1}] f) / (x_j - x_i), & i < j \end{cases} \quad (6)$$

These coefficients can therefore be determined as final entries in a so-called divided difference table, for instance as in the following subprogram


```

SUBROUTINE POLINT (X, F, N)

DIMENSION X(N), F(N)

NML = N-1

IF (NML .LE. 0) RETURN

DO 10 K=1,NML

    NMK = N-K

    DO 10 I=2,NMK

10      F(I) = (F(I+1) - F(I))/(X(I+K) - X(I))

    END                                RETURN

```

Here, the array F contains $F(i) = f(x_i)$, $i=1, \dots, n$, on input and $F(i) = [x_i, \dots, x_n]f$, $i=1, \dots, n$, on output. (For details concerning divided differences and the Newton form (5), see, e.g., [2].)

Once the coefficient vector a in the representation $\sum_i a_i \varphi_i$ for the interpolant g has been determined, one may evaluate g in various ways. Typically, one then wants to find λg for various linear functionals λ such as $\lambda g = g(x)$, some x , or $\lambda g = g^{(j)}(x)$, or $\lambda g = \int \psi g$ for some ψ , etc. . All of these values can be obtained from the vector $a = (a_i)$ by applying to it a matrix consisting of just one row, viz. the matrix $[\lambda \varphi_1, \lambda \varphi_2, \dots]$. Thus evaluation of the interpolant at some linear functional λ is just another linear procedure or subprogram which applies some matrix B to the vector a .

For example, the evaluation of the interpolating polynomial (5) at some point $x = \text{ARG}$ proceeds customarily by Nested Multiplication, as in the following function subprogram

```

FUNCTION POLVAL (X, F, N, ARG)

DIMENSION X(N), F(N)

POLVAL = F(1)

IF (N .LE. 1) RETURN

DO 10 K=1,N

10      POLVAL = POLVAL*(ARG-X(K)) + F(K)

    RETURN

END

```

Note that, once again, the matrix B to be applied to the coefficient vector a (in the array F) is not formed explicitly.

Suppose now that we have, for each of the k independent variables $t_1, \dots, t_k$, a linear interpolation scheme. This means that, for $r=1, \dots, k$, we have a matrix B_r which associates with each data vector $(\lambda_i^r f)$ a coefficient vector $(a_i^r) = B_r(\lambda_i^r f)$, giving the interpolant $g_r = \sum_i a_i^r \varphi_{i,r}$ for $f = f(t_r)$. Further, for all appropriate integer vectors $\underline{i} = (i_1, \dots, i_k)$, let $\lambda_{\underline{i}}$ be a linear functional on some appropriate class of functions f of k variables for which

$$\lambda_{\underline{i}} f = (\lambda_{i_1}^1 f_1)(\lambda_{i_2}^2 f_2) \cdots (\lambda_{i_k}^k f_k)$$

whenever

$$f(t_1, \dots, t_k) = f_1(t_1)f_2(t_2)\dots f_k(t_k), \quad \text{all } t_1, \dots, t_k.$$

For example, if $k=3$ and $\lambda_1^r f = f(\alpha_r)$, $\lambda_2^r f = f''(\beta_r)$ and $\lambda_3^r f = \int_{a_r}^{b_r} f(t) dt$, then

$$\lambda_{(1,1,1)} f := f(\alpha_1, \alpha_2, \alpha_3)$$

$$\lambda_{(1,2,3)} f := \int_{a_3}^{b_3} (\partial/\partial t_2)^2 f(\alpha_1, \beta_2, t_3) dt_3$$

$$\lambda_{(2,2,1)} f := (\partial^4/\partial t_1^2 \partial t_2^2) f(\beta_1, \beta_2, \alpha_3)$$

would serve. Also, let

$$\varphi_{\underline{i}}(t_1, \dots, t_k) := \varphi_{i_1,1}(t_1)\varphi_{i_2,2}(t_2)\dots\varphi_{i_k,k}(t_k).$$

Then we can construct an interpolant

$$g = \sum_{\underline{i}} a_{\underline{i}} \varphi_{\underline{i}}$$

for a function f of the k variables $t_1, \dots, t_k$ as follows: Calculate the k -dimensional array $\underline{a} = (a_{\underline{i}})$ as

$$\underline{a} = (B_1 \otimes \dots \otimes B_k)(\lambda_{\underline{i}} f)$$

from the k -dimensional array $(\lambda_i f)$ of data. This function g is then indeed an interpolant to f in the sense that

$$\lambda_i g = \lambda_i f, \text{ all } i.$$

The calculation of the coefficient array a is, of course, easily effected as described in the corollary above.

To follow up on the example of polynomial interpolation, an appropriately extended version POLNTE of the subprogram POLINT would require a separate output array, D say, for the calculated divided differences. Otherwise, only the statement labelled 10,

```
10      F(I) = (F(I+1) - F(I))/(X(IPK) - X(I))
```

needs to be put into an additional loop over the data sets, with the difference $X(IPK) - X(I)$ calculated outside that loop, of course. We get

```

SUBROUTINE POLNTE (X, F, N, M, D)
  DIMENSION X(N),F(N,M),D(M,N)
  DO 5 I=1,N
    DO 5 J=1,M
5      D(J,I) = F(I,J)
  NM1 = N-1
  IF (NM1 .LE. 0) RETURN
  DO 10 K=1,NM1
    NMK = N-K
    DO 10 I=1,NMK
      DIFF = X(I+K) - X(I)
      DO 10 J=1,M
10      D(J,I) = (D(J,I+1)-D(J,I))/DIFF
      RETURN
  END
```

Note that this routine functions appropriately even for $M = 1$, the only difference compared to POLINT being that the output is now to be found in D and not in F . Note further that it takes $N(N-1)/2$ adds and divides per data set to form $B(\lambda, f)$. Since the matrix B^{-1} is upper triangular in this case, explicit application of B by backsubstitution would take no fewer operations and would require the generation and storage of B (or its inverse).

Now, to illustrate the lemma and its corollary, suppose that we require the polynomial interpolant $p = p(x, y, z)$ to data

$$f(x_i, y_j, z_k), \quad i=1, \dots, n_x; \quad j=1, \dots, n_y; \quad k=1, \dots, n_z.$$

We load $f(x_i, y_j, z_k)$ into $F(i, j, k)$, x_i into $X(i)$, y_j into $Y(j)$ and z_k into $Z(k)$, for all appropriate i, j, k . Then

$$N := n_x * n_y * n_z$$

CALL POLNTE ($X, F, n_x, N/n_x, D$)

CALL POLNTE ($Y, D, n_y, N/n_y, F$)

CALL POLNTE ($Z, F, n_z, N/n_z, D$)

to get the appropriate polynomial coefficients of the polynomial interpolant p into the 3-dimensional array D .

If we wish to evaluate this interpolant at some point $(\hat{x}, \hat{y}, \hat{z})$, we have to procure an extended version of the function routine POLVAL. The output for such a routine will consist now of more than one number, we must therefore give up on having a function. Otherwise, it is again only the assignment statement $POLVAL = F(1)$ and statement 10 which need to be put into a loop over the data sets. Here is an extended version POLVLE of POLVAL.

```

SUBROUTINE POLVLE (X, D, N, M, ARG, VALUE)
DIMENSION X(N), D(N,M), VALUE(M)
DO 5 J=1,M
5   VALUE(J) = D(1,J)
IF (N .LE. 1) RETURN
DO 10 K=2,N
FACTOR = ARG - X(K)
DO 10 J=1,M
10  VALUE(J) = VALUE(J)*FACTOR + D(K,J)
RETURN
END

```

Now, to find $p(\hat{x}, \hat{y}, \hat{z})$,

```

CALL POLVLE (X, D, nx, N/nx,  $\hat{x}$ , TEMP1)
CALL POLVLE (Y, TEMP1, ny, nz,  $\hat{y}$ , TEMP2)
CALL POLVLE (Z, TEMP2, nz, 1,  $\hat{z}$ , ANSWER)

```

to get $p(\hat{x}, \hat{y}, \hat{z}) = \text{ANSWER}$. Note that TEMP1 must be of size $n_y * n_z$ and contains the necessary information to evaluate the bivariate polynomial $p(\hat{x}, y, z)$ for any choice of y and z . Again, TEMP2 is of size n_z and contains the appropriate coefficients of the polynomial $p(\hat{x}, \hat{y}, z)$ in the single variable z . In particular, if p is to be evaluated at all points of a regular grid, it is most efficient to evaluate p along lines parallel to the z -axis.

As an example of some of the difficulties one might encounter, we now discuss briefly osculatory polynomial interpolation. Here, the interpolant is again of the form (5), but now some of the interpolation points $x_1, \dots, x_n$ might coincide. This requires an extension of (6) which reads as follows:

$$[x_1, \dots, x_j] f := f^{(j-i)}(x_i) / (j-i)!, \quad \text{if } x_i = \dots = x_j. \quad (6a)$$

By insisting that, for given data points $x_1, \dots, x_n$, we have

$$x_i = x_j \text{ implies } x_i = x_{i+1} = \dots = x_j ,$$

(6) and (6a) cover all eventualities. The point of this extension is that now p_f agrees with f in the sense that

$$p^{(r)}(z) = f^{(r)}(z) \text{ in case the number } z \text{ appears (at least) } r+1 \text{ times in the sequence } x_1, \dots, x_n .$$

This explains the term "osculatory".

The following program for the construction of the coefficients in (5) is based on (6) and (6a) and can be found, in somewhat different notation, in [2].

```

SUBROUTINE POLOSC (X, F, N)
C INPUT MUST SATISFY THE FOLLOWING.
C IF X(I-1) .NE. X(I) = X(I+J) .NE. X(I+J+1), THEN
C X(I+L) = X(I) AND F(I+L) = (D**L)F(X(I)), L=0,...,J .
C (HERE, X(0), X(N+1) .NE. X(I), I=1,...,N, BY DEFINITION.)
      DIMENSION X(N),F(N)
      NML = N-1
      IF (NML) .LE. 0)          RETURN
      DO 10 K=1,NML
        FLOATK = K
        NMK = N-K
        FLAST = F(1)
        DO 9 I=1,NMK
          DX = X(I+K) - X(I)
          IF (DX .EQ. 0.)          GO TO 7
          F(I) = (F(I+1) - FLAST)/DX
          FLAST = F(I+1)
        GO TO 9
      7      F(I) = F(I+1)/FLOATK
      9      CONTINUE
     10     F(NMK+1) = FLAST
          RETURN
      END

```

The construction of an efficient extension of POLOSC is made difficult by the fact that the local variable FLAST depends on the data F but is active through various

statements which are independent of the data F and should therefore not be put inside a loop over the various data sets. One way out is to make $FLAST$ an array of length M , either local or as an argument, which then requires the four groups of statements

```
FLAST = F(1)
F(I) = (F(I+1) - F(I))/DX ; FLAST = F(I+1)
F(I) = F(I+1)/FLOATK
F(NMK+1) = FLAST
```

each be put into a loop over the different data sets.

An alternative way consists in a reorganization of the entire calculation which avoids the temporary saving of terms which depend on F , possibly at the cost of a slight increase in F -independent work. For the record, here is such a subprogram. Note that the input information in F is to be arranged differently, too.

```
SUBROUTINE POLSCN (X, F, N)
C INPUT MUST SATISFY THE FOLLOWING.
C IF X(I-1) .NE. X(I) = X(I+J) .NE. X(I+J+1), THEN
C X(I+L) = X(I) AND F(I+L) = (D**(J-L))F(X(I)), L=0,...,J.
C (HERE, BY DEFINITION, X(0), X(N+1) .NE. X(I), I=1,...,N.)
  DIMENSION X(N),F(N)
  NML = N-1
  IF (NML .LE. 0) RETURN
  DO 3 NEXTP1=2,N
    IF (X(NEXTP1) .NE. X(1)) GO TO 4
  3 CONTINUE
  NEXTP1 = N+1
  4 DO 10 K=1,NML
    NEXT = NEXTP1-1
    FLOATK = FLOAT(K)
    NMK = N-K
    DO 9 I=1,NMK
      IF (NEXT .EQ. I) GO TO 5
      F(I) = F(I)/FLOATK
    GO TO 9
```

```

5      NEXT = NEXT+1
      IF (NEXT .GT. NMK)          GO TO 7
      IF (X(NEXT+K) .EQ. X(NEXT)) GO TO 5
7      F(I) = (F(NEXT) - F(I))/(X(I+K) - X(I))
9      CONTINUE
10     NEXTPl = MAXO(2,NEXTPl-1)

                                RETURN

```

END

We do not bother to carry out here the extension of this routine because it is straightforward. Aside from an initial transfer of $F(i,j)$ to $D(j,i)$, all i,j , only two statements,

$$F(I) = F(I)/\text{FLOATK}$$

$$F(I) = (F(\text{NEXT}) - F(I))/(X(I+K) - X(I))$$

need to be put into a loop over the data sets, with the difference $X(I+K) - X(I)$ formed outside such a loop (and, of course, F replaced by $D(j,.)$).

We close with an example in which the "matrix" B is three-dimensional, taking vectors to matrices, viz. complete cubic spline interpolation. A typical implementation of this scheme (see, e.g., [2]) starts off with an array C , of dimension $(4,n+1)$, which contains the following information initially:

$$C(1,i) = f(x_i), \quad i=1, \dots, n+1$$

$$C(2,1) = f'(x_1), \quad C(2,n+1) = f'(x_{n+1})$$

This says that the data $(\lambda_i f)$ about f in this scheme consist of the vector $(f(x_1), \dots, f(x_{n+1}), f'(x_1), f'(x_{n+1}))$. After passing through a subroutine

SPLINE (X, C, N) ,

the array C contains the coefficients of the polynomial pieces which make up the interpolating cubic spline, i.e.,

$$C(j,i) = g^{(j-1)}(x_i)/(j-1)!, \quad j=1, \dots, 4 \quad \text{and} \quad i=1, \dots, n$$

For an extended version, it would seem reasonable to introduce a separate input array, F say, with

$$(F(1), \dots, F(n+3)) = (f(x_1), \dots, f(x_{n+1}), f'(x_1), f'(x_{n+1}))$$

The calling statement of the extended version then might be

SPLNEE (X, F, N+3, M, C, N)

with F and C dimensioned internally as

F(N+3,M), C(M,4,N)

Thus, if SPLNEE is used as SUB_i in the corollary above, then

$$n_i = N+3, \quad r_i = 4*N$$

Consequently, bicubic spline interpolation, on a mesh $(x_i)_1^{n+1}$ by $(y_j)_1^{m+1}$, would be carried out by

CALL SPLNEE (X, F, n+3, m+3, C, n)

CALL SPLNEE (Y, C, m+3, 4*n, F, m)

with F initially of dimension (n+3, m+3) and containing the data

$$F = \begin{bmatrix} f(x_1, y_1) & \dots & f(x_1, y_{m+1}) & f_y(x_1, y_1) & f_y(x_1, y_{m+1}) \\ \cdot & & \cdot & \cdot & \cdot \\ \cdot & & \cdot & \cdot & \cdot \\ \cdot & & \cdot & \cdot & \cdot \\ f(x_{n+1}, y_1) & \dots & f(x_{n+1}, y_{m+1}) & f_y(x_{n+1}, y_1) & f_y(x_{n+1}, y_{m+1}) \\ f_x(x_1, y_1) & \dots & f_x(x_1, y_{m+1}) & f_{xy}(x_1, y_1) & f_{xy}(x_1, y_{m+1}) \\ f_x(x_{n+1}, y_1) & \dots & f_x(x_{n+1}, y_{m+1}) & f_{xy}(x_{n+1}, y_1) & f_{xy}(x_{n+1}, y_{m+1}) \end{bmatrix}$$

After the two calls, F contains the polynomial coefficients of the interpolating bicubic spline,

$$F(i+1, r, j+1, s) = (\partial/\partial x)^i (\partial/\partial y)^j g(x_r, y_s), \quad i, j=0, \dots, 3 \quad (7)$$

$$r=1, \dots, n, \quad s=1, \dots, m$$

Note the difference between this way of storing the coefficients and the customary way followed by the various available routines which return the coefficients in some array COEF containing

$$\text{COEF}(i,j,r,s) = F(i,r,j,s) \quad .$$

The coefficient array F , organized as in (7), lends itself easily to evaluation by extended univariate evaluation routines.

In summary, the approach to tensor products advocated here allows one to do the detailed programming work in the univariate context. The resulting programs are then strung together to give or evaluate a tensor product interpolant (or, effect multiplication by a tensor or Kronecker product of matrices) with an ease which mirrors the ease of the mathematical construction of tensor products.

REFERENCES

1. C. de Boor, Appendix to 'Splines and Histograms' by I. J. Schoenberg, MRC TSR #1273, October 1972; in "Spline Functions and Approximation Theory", A. Meir & A. Sharma eds., Birkhäuser Verlag, Basel, 1973, 329-358.
2. S. Conte & C. de Boor, "Elementary Numerical Analysis. 2nd Edition", McGraw-Hill, New York, 1972.
3. V. Pereyra & G. Scherer, Efficient Computer Manipulation of Tensor Products with Applications to Multidimensional Approximation, Math. Comp. 27 (1973) 595-605.

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER #1810 <u>MRC-TSR-1810</u>	2. GOVT ACCESSION NO. <u>9</u>	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) <u>EFFICIENT COMPUTER MANIPULATION OF TENSOR PRODUCTS.</u>	5. TYPE OF REPORT & PERIOD COVERED <u>Summary Report, no specific reporting period</u>	
7. AUTHOR(s) <u>Carl de Boer</u>	6. PERFORMING ORG. REPORT NUMBER	
9. PERFORMING ORGANIZATION NAME AND ADDRESS Mathematics Research Center, University of 610 Walnut Street Madison, Wisconsin 53706	8. CONTRACT OR GRANT NUMBER(s) <u>15 DAAG29-75-C-0024</u>	
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Office P.O. Box 12211 Research Triangle Park, North Carolina 27709	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Work Unit Number 7 - Numerical Analysis	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	12. REPORT DATE <u>11 November 1977</u>	
	13. NUMBER OF PAGES 17 <u>12</u> <u>21 p.</u>	
	15. SECURITY CLASS. (of this report) UNCLASSIFIED	
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Mathematical Software, Tensor product, Multivariate, Interpolation, Approximation, Polynomial, Osculatory.		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) It is shown how to construct a modified version SUB_i of a (presumably efficient) subroutine SUB_i for solving the linear system $A_i x = b$, $i=1, \dots, k$, so that the linear system $(A_1 \otimes \dots \otimes A_k) x = b$ can be solved by just one call to each of the subroutines SUB_i , $i=1, \dots, k$. Polynomial interpolation and spline interpolation in several variables are given as examples.		

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

221 200 J08

78