Efficient Computer Manipulation of Tensor Products

CARL DE BOOR

University of Wisconsin-Madison

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

$$(A_1 \otimes \cdots \otimes A_k)\mathbf{x} = \mathbf{b}$$

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

Key Words and Phrases: tensor product, multivariate, interpolation, approximation, polynomial, spline, osculatory
CR Categories. 5.13

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

$$(A_1 \otimes \cdots \otimes A_k)\mathbf{x} = \mathbf{b} \tag{1}$$

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

Pereyra and Scherer propose to store arrays such as $\mathbf{x}$ and $\mathbf{b}$ 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 paper 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 SUB,b, n, x) which solves the ith 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 $\mathbf{x}$ and $\mathbf{b}$ are (to be) stored in Fortran fashion, i.e.,

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

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

The following simple procedure will then lead to an efficient way for solving eq. (1). For each i, enlarge the subroutine SUB_i to a subroutine $SUB_i'(b, n, m, x)$ which solves simultaneously $A_i x = b$ for m given right sides $b(\cdot, 1), b(\cdot, 2), \ldots, b$ $(\cdot, m)$, each of length $n = n_i$, and stores the corresponding solutions in $x(1, \cdot), x(2, \cdot), \ldots, x(m, \cdot)$. 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 jth column $b(\cdot, j)$ of the input array customarily in the jth *column* of the output array x and not, as I propose here, in the jth *row*.

LEMMA. *For $i = 1, \ldots, k$, let SUB_i be an expanded version, as described, of the routine SUB_i for solving $A_i x = b$, and set $N := n_1*n_2*\cdots*n_k$. Then, the following statements*

$\mathbf{b}_0 := b$
$CALL\ Sub_1'(\mathbf{b}_0, n_1, N/n_1, \mathbf{b}_1)$
$CALL\ SUB_2'(\mathbf{b}_1, n_2, N/n_2, \mathbf{b}_2)$

.
.
.

$CALL\ SUB_k'(\mathbf{b}_{k-1}, n_k, N/n_k, \mathbf{b}_k)$
$\mathbf{x} := \mathbf{b}_k$

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

PROOF. Let $\mathbf{x}_i$ be the k-dimensional array $\mathbf{x}_i := (A_1^{-1} \otimes \cdots \otimes A_i^{-1} \otimes 1 \otimes \cdots \otimes 1)\mathbf{b}$, $i = 0, \ldots, k$. Then

$$\mathbf{x}_i(j_1, \ldots, j_{i-1}, \cdot, j_{i+1}, \ldots, j_k) = A_i^{-1}\mathbf{x}_{i-1}(j_1, \ldots, j_i - 1, \cdot, j_{i+1}, \ldots, j_k) \qquad (2)$$

and our assertion is proved if we can establish that $\mathbf{b}_k = \mathbf{x}_k$. We prove this by showing that, for all i, $\mathbf{b}_i$ as generated by the succession of calls above is related to $\mathbf{x}_i$ in the following way: *If $\mathbf{b}_i$ is interpreted as a k-dimensional Fortran array, of dimension $(n_{i+1}, \ldots, n_k, n_1, \ldots, n_i)$, then*

$$\mathbf{b}_i(j_{i+1}, \ldots, j_k, j_1, \ldots, j_i) = \mathbf{x}_i(j_1, \ldots, j_k), \text{ all } \mathbf{j}, \qquad (3)$$

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

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

CALL $\text{SUB}_v'(\mathbf{b}_{v-1}, n_v, N/n_v, \mathbf{b}_v)$.

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

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

we have

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

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

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

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

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

which proves eq. (3) for $i = v$ and so advances the induction hypothesis.   Q.E.D.

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

Also, in typical situations, the various subroutines $\text{SUB}_1, \ldots, \text{SUB}_k$ 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 $(B_1 \otimes \cdots \otimes B_k)\mathbf{b}$, 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 $i = 1, \ldots, k$, let B_i be a (n_i, r_i)-matrix, and let $\text{SUB}_i'(b, n_i, m, x, r_i)$ be a subroutine which, for $j = 1, \ldots, m$, 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*

$\mathbf{b}_0 := \mathbf{b}$
$m := n_2 * \cdots * n_k$
CALL $\text{SUB}_1'(\mathbf{b}_0, n_1, m, \mathbf{b}_1, r_1)$
$m := m * r_1/n_2$

CALL SUB$_2'$(**b**$_1$, n_2, m, **b**$_2$, r_2)

$m := m*r_2/n_3$

.

.

.

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

x := **b**$_k$

form the k-dimensional array **x** = ($B_1 \otimes \cdots \otimes B_k$)**b**.

It is not even necessary that B_t be a matrix, i.e., a two-dimensional array. The more general situation in which B_t is a linear map which associates s_t-dimensional arrays with t_t-dimensional arrays is covered by the corollary as well since we can always interpret such s_t-dimensional and t_t-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 $a = (a_t)$ in a representation

$$g = \sum_t a_t \varphi_t$$

for the interpolant from certain information $(\lambda_t f)$ about f. Here, each λ_t is a *linear functional*, e.g., $\lambda_t f = f(x_t)$ or $\lambda_t f = f^{(r_t)}(x_t)$ or $\lambda_t f = \int \psi_t(x) f(x) dx$, etc., and g is so constructed that $\lambda_t g = \lambda_t f$, 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_t) to be linearly independent. All that is necessary is the assumption that $a = B(\lambda_t f)$ 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 $(\lambda_t f)$ of data appropriately into the vector a of coefficients.

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

$$p_f(x) = \sum_{t=1}^{n} [x_t, \ldots, x_n] f \cdot \prod_{j=t+1}^{n} (x - x_j) \tag{5}$$

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

$$[x_t, \ldots, x_j] f := \begin{cases} f(x_t), & i = j \\ ([x_{t+1}, \ldots, x_j] f - [x_t, \ldots, x_{j-1}] f)/(x_j - x_t), & i < j. \end{cases} \tag{6}$$

These coefficients can therefore be determined as final entires in a so-called

divided difference table, for instance, as in the following subprogram:

```
      SUBROUTINE POLINT (X, F, N)
      DIMENSION X(N), F(N)
      NM1 = N − 1
      IF (NM1 .LE. 0)                    RETURN
      DO 10 K = 1, NM1
          NMK = N − K
          DO 10 I = 1, NMK
10            F(I) = (F(I + 1) − F(I))/(X(I + K) − X(I))
                                         RETURN
      END
```

Here, the array F contains $F(i) = f(x_i)$, $i = 1, \ldots, n$, on input and $F(i) = [x_i, \ldots, x_n]f$, $i = 1, \ldots, n$, 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 $[\lambda \varphi_1, \lambda \varphi_2, \ldots]$. Thus evaluation of the interpolant at some linear functional λ is just another linear procedure or subprogram which applied 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 of coefficients (in the array F) is not formed explicitly.

Suppose now that we have, for each of the k independent variables $t_1, \ldots, t_k$, a linear interpolation scheme. This means that, for $r = 1, \ldots, k$, 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 $\mathbf{i} = (i_1, \ldots, i_k)$, let λ_i be a linear functional on some appropriate class of functions f of k variables for which

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

whenever $f(t_1, \ldots, t_k) = f_1(t_1)f_2(t_2) \ldots f_k(t_k)$, all $t_1, \ldots, t_k$. 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

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

$$\lambda_{(1,2,3)} f := \int_{\alpha_3}^{b_3} (\partial/\partial t_2)^2 f(\alpha_1, \beta_2, t_3) \, dt_3$$

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

would serve. Also, let $\varphi_i(t_1, \ldots, t_k) := \varphi_{i_1,1}(t_1)\varphi_{i_2,2}(t_2 \ldots \varphi_{i_k,k}(t_k)$.

Then we can construct an interpolant $g = \sum_i a_i \varphi_i$ for a function f of the k variables $t_1, \ldots, t_k$ as follows: Calculate the k-dimensional array $\mathbf{a} = (a_i)$ as $\mathbf{a} = (B_1 \otimes \cdots \otimes B_k)(\lambda_i f)$ from the k-dimensional array $(\lambda_i f)$ of data. This function g is then indeed an interpolant to f in the sense that $\lambda_i g = \lambda_i f$, all i. The calculation of the coefficient array $\mathbf{a}$ 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 labeled 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 $B(\lambda_i f)$. 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 $f(x_i, y_j, z_k)$, $i = 1, \ldots, n_x$; $j = 1, \ldots, n_y$; $k = 1, \ldots, n_z$. 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 three-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 POLVE (X, D, n_x, N/n_x, $\hat{x}$, TEMP1)
CALL POLVE (Y, TEMP1, n_y, n_z, $\hat{y}$, TEMP2)
CALL POLVE (Z, TEMP2, n_z, 1, $\hat{z}$, ANSWER)

to get p $(\hat{x}, \hat{y}, \hat{z})$ = ANSWER. Note that TEMP1 must be of size n_y*n_z and contains the necessary information to evaluate the *bi*variate 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 the 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 of eq. (5), but now some of the interpolation points $x_1, \ldots, x_n$ might coincide. This requires an extension of eq. (6) which reads as follows:

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

By insisting that, for given data points $x_1, \ldots, x_n$, we have $x_i = x_j$ implies $x_i = x_{i+1} = \ldots = x_j$, eqs. (6) and (6a) cover all eventualities. The point of this extension is that now p_f agrees with f in the sense that $p^{(r)}(z) = f^{(r)}(z)$ in case the number z appears (at least) $r + 1$ times in the sequence $x_1, \ldots, x_n$. This explains the term *osculatory.*

The following program for the construction of the coefficients in eq. (5) is based on eqs. (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)
      NM1 = N − 1
      IF (NM1 .LE. 0)                      RETURN
      DO 10 K = 1, NM1
         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)
      NM1 = N − 1
      IF (NM1 .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, NM1
           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  NEXTP1 = MAX0(2, NEXTP1 - 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)/FLOATK
F(I) = (F(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 $D(j, \cdot)$).

   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:

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

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

This says that the data (λ, f) about f in this scheme consist of the vector $(f(x_1), \ldots, f(x_{n+1}), f'(x_1), f'(x_{n+1}))$. 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., $C(j, i) = g^{(j-1)}(x_i)/(j - 1)!, j = 1, \ldots, 4$ and $i = 1, \ldots, n$.

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

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

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,' in the corollary above, then n_i = N + 3, 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

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

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

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

$$i, j = 0, \ldots, 3; r = 1, \ldots, n; s = 1, \ldots, m. \quad (7)$$

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 COEF(i, j, r, s) = F(i, r, j, s). The coefficient array F, organized as in eq. (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. DE BOOR, C. Appendix to "Splines and histograms" by I.J. Schoenberg Rep. MRC TSR 1273, Oct 1972, in *Spline Functions and Approximation Theory*, A Meir and A Sharma, Eds., Birkhauser Verlag, Basel, Switzerland, 1973, pp. 329–358.
2 CONTE, S., AND DE BOOR, C *Elementary Numerical Analysis*. McGraw-Hill, New York, second ed., 1972.
3 PEREYRA, V., AND SCHERER, G. Efficient computer manipulation of tensor products with applications to multidimensional approximation *Math. Comput* 27 (1973), 595–605.