MATHEMATICS OF COMPUTATION, VOLUME 27, NUMBER 123, JULY, 1973

Efficient Computer Manipulation of Tensor Products with Applications to Multidimensional Approximation

By V. Pereyra and G. Scherer

Abstract. The objective of this paper is twofold:

(a) To make it possible to perform matrix-vector operations in tensor product spaces, using only the factors ($n \cdot p^2$ words of information for $\bigotimes_{i=1}^{n} A_i$, $A_i \in \mathcal{L}(E^p, E^p)$) instead of the tensor-product operators themselves ($(p^2)^n$ words of information).

(b) To produce efficient algorithms for solving systems of linear equations with coefficient matrices being tensor products of nonsingular matrices, with special application to the approximation of multidimensional linear functionals.

1. Introduction. The use of multilinear algebra in applied numerical analysis has been rare. However, this is not the case in theoretical numerical analysis. In recent times, interest has grown in the use of tensor product interpolation rules in such different areas as multidimensional numerical quadrature [6], finite elements [4], interpolation and approximation.

Despite this widespread theoretical interest, there are practically no algorithms for performing the various tasks required by these applications. This paper attempts to start filling the gap.

We shall consider some of the basic operations in tensor spaces, and we shall indicate ways and means to perform them on a digital computer using a high level programming language. The aim is, of course, towards economy, both in arithmetic and storage, simplicity, sequential processing, and thus optimization in the manipulation of subscripts.

After giving some basic notations in Section 2, we pass on to describe an algorithm for performing the Kronecker product matrix-tensor multiplication $(A_1 \otimes \cdots \otimes A_k)\mathbf{x}$. It is clear from the beginning that a computer implementation of an algorithm which wants to be independent of the number of factors k must avoid the use of multi-indexed arrays. This holds even more if considerations on economy in index manipulations and storage are taken into account. It turns out, as we explain in Sections 3 and 4, that the whole process can be carried out sequentially and in a fairly simple manner.

In Section 5, we deal with systems of linear equations of the form

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

The idea in all cases is, of course, to be able to work with the factors A_i individually, and never form explicitly the tensor product $\otimes A_i$. This is achieved in a

Received March 8, 1972.

AMS (MOS) subject classifications (1970). Primary 65D15, 65F30; Secondary 65N30, 15-04, 15A69.

Key words and phrases. Computer manipulation of tensor products, multidimensional functional approximation, construction of finite elements, tensor product systems.

Copyright © 1973, American Mathematical Society

fairly straightforward way, and the practicality of the algorithms and the code offered are supported by the numerical results of Section 6, which show applications in tri-dimensional Lagrange interpolation.

This algorithm, coupled with the efficient, one-dimensional techniques of [1], [5], should provide a powerful tool in many applications.

Those readers only interested in the algorithmic part of this paper, and who are ready to accept the validity of the recursion (3.2), should direct their attention to Sections 4, 5, 6 and skip the more formal (and somewhat heavy) manipulations of Sections 2 and 3.

The authors would like to acknowledge the very expert and kind handling of this paper by the editor, and also the excellent suggestions of one of the referees.

2. Tensor Product Spaces. In this section, we shall introduce some necessary notation and well-known properties of tensor product spaces which will be needed in the sequel. For details, we refer to [3].

Let $U_1, U_2, \cdots, U_k$ be Euclidean spaces of dimensions $n_1, n_2, \cdots, n_k$, respectively, and let $U_1^*, U_2^*, \cdots, U_k^*$ be their duals.

As usual, if $\mathbf{x}$ is an element of the Euclidean space U, we denote by $\mathbf{x}^*$ the linear functional (element of U^*) defined by

$$\forall\ \mathbf{y} \in U,\ x^*(\mathbf{y}) = \langle \mathbf{x}, \mathbf{y} \rangle \quad \text{(where } \langle\ ,\ \rangle \text{ denotes inner product).}$$

The tensor product space U formed with $U_1, U_2, \cdots, U_k$ will be denoted by $U_1 \otimes U_2 \otimes \cdots \otimes U_k$, and it will be the set of all linear combinations of the symbols

$$\mathbf{x}_1 \otimes \mathbf{x}_2 \otimes \cdots \otimes \mathbf{x}_k, \quad \text{with } \mathbf{x}_i \in U_i. \tag{2.1}$$

Let us suppose that the tensor product space U is formed with s spaces $\{U_i\}$, $i = 1, \cdots, s$, and t spaces $\{U_{i_j}^*\}$, $j = 1, \cdots, t$, where the $U_{i_j}^*$ are the duals of some of the spaces U_i. Then we shall call its elements s times contravariant and t times covariant tensors, or (s, t)-tensors for short.

Any element of $U = U_1 \otimes \cdots \otimes U_s \otimes U_{i_1}^* \otimes \cdots \otimes U_{i_t}^*$ can be generated as a linear combination of elements of the form

$$\mathbf{x}_1 \otimes \cdots \otimes \mathbf{x}_s \otimes \mathbf{x}_{i_1}^* \otimes \cdots \otimes \mathbf{x}_{i_t}^*,$$

where the $\mathbf{x}_l \in U_l$, $\mathbf{x}_{i_l}^* \in U_{i_l}^*$.

From a (s, t)-basis-tensor, we can obtain a $(s - 1, t - 1)$-basis-tensor by *contraction* of one contravariant and one covariant component, provided they belong to dual spaces

$$\begin{aligned} c_i^j(\mathbf{x}) &= c_i^j(\mathbf{x}_1 \otimes \cdots \otimes \mathbf{x}_s \otimes \mathbf{x}_{i_1}^* \otimes \cdots \otimes \mathbf{x}_{i_t}^*) \\ &= \mathbf{x}_{i_j}^*(\mathbf{x}_i) \cdot (\mathbf{x}_1 \otimes \cdots \otimes \mathbf{x}_s \otimes \mathbf{x}_{i_1}^* \otimes \cdots \otimes \mathbf{x}_{i_t}^*), \end{aligned} \tag{2.2}$$

where $i_j = i$.

The operator c_i^j is extended linearly to all U and is called the contraction of the ith contravariant with the i_jth covariant component.

Given two tensors $\mathbf{u}$, $\mathbf{v}$, belonging to two spaces U, V, we define the tensor product of $\mathbf{u}$ and $\mathbf{v}$ as the element of $U \otimes V$:

$$\begin{aligned}\mathbf{u} \otimes \mathbf{v} &= \Big(\sum u_{i_1,\cdots,i_k}^{j_1,\cdots,j_l} \mathbf{e}_{i_1} \otimes \cdots \otimes \mathbf{e}_{i_k} \otimes \mathbf{e}_{j_1}^* \otimes \cdots \otimes \mathbf{e}_{j_l}^*\Big) \\ &\quad \otimes \Big(\sum v_{i_1',\cdots,i_s'}^{j_1',\cdots,j_t'} \mathbf{f}_{i_1'} \otimes \cdots \otimes \mathbf{f}_{i_s'} \otimes \mathbf{f}_{j_1'}^* \otimes \cdots \otimes \mathbf{f}_{j_t'}^*\Big) \\ &= \sum u_{i_1,\cdots,i_k}^{j_1,\cdots,j_l} \cdot v_{i_1',\cdots,i_s'}^{j_1',\cdots,j_t'} \mathbf{e}_{i_1} \otimes \cdots \otimes \mathbf{e}_{i_k} \otimes \mathbf{f}_{i_1'} \\ &\quad \otimes \cdots \otimes \mathbf{f}_{i_s'} \otimes \mathbf{e}_{j_1}^* \otimes \cdots \otimes \mathbf{e}_{j_l}^* \otimes \mathbf{f}_{j_1'}^* \otimes \cdots \otimes \mathbf{f}_{j_t'}^*.\end{aligned}$$

Observe that we have chosen to collect first all the contravariant components, and then all the covariant ones. This we shall always do.

Any linear operator $A \in \mathcal{L}(U)$ can be thought of as a (1, 1) tensor, T_A. In fact, if $\{\mathbf{e}_i\}$ is a basis for U, and $\{\mathbf{e}_i^*\}$ is the dual basis, then we can associate with A (in a 1-1 correspondence) the tensor $T_A = \sum_i (A\mathbf{e}_i) \otimes \mathbf{e}_i^*$, and obtain $A\mathbf{x}$ through tensor multiplication and contraction

$$(2.3) \qquad A\mathbf{x} = c_2^1\Big(\sum_j A\mathbf{e}_j \otimes \mathbf{x} \otimes \mathbf{e}_j^*\Big) = \sum_j \mathbf{e}_j^*(\mathbf{x}) A\mathbf{e}_j = \sum_j x_j A\mathbf{e}_j.$$

Observe that $A\mathbf{e}_j = \mathbf{a}_j$ is the jth column of the matrix representation of A in the basis $\{\mathbf{e}_i\}$.

If A_i, $i = 1, \cdots, k$, are linear operators from U_i into U_i, then $A = A_1 \otimes \cdots \otimes A_k$ is a linear operator from $U = U_1 \otimes \cdots \otimes U_k$ into itself, defined by

$$(2.4) \qquad (A_1 \otimes \cdots \otimes A_k)(\mathbf{x}_1 \otimes \cdots \otimes \mathbf{x}_k) = A_1\mathbf{x}_1 \otimes \cdots \otimes A_k\mathbf{x}_k,$$

and extended to all U by linearity.

If the A_i are nonsingular, then it is well known that A is nonsingular and that

$$(2.5) \qquad (A_1 \otimes \cdots \otimes A_k)^{-1} = A_1^{-1} \otimes \cdots \otimes A_k^{-1}.$$

As is usually done, whenever there is no possibility of confusion, we shall use the same symbol to designate a linear operator or its matrix (tensor) representation in a given basis.

3. Computation with Tensor Product Linear Operators. Our objective now is to describe the computation of $\mathbf{y} = (A_1 \otimes A_2 \otimes \cdots \otimes A_k)\cdot\mathbf{x}$, using a minimum of storage and index manipulation.

To fix ideas, let us consider first in detail the case $k = 2$:

$$\mathbf{y} = (A_1 \otimes A_2)\mathbf{x}, \quad \text{where} \quad \mathbf{x} = \sum_{s_1}\sum_{s_2} x_{s_1 s_2} \mathbf{e}_{s_1,1} \otimes \mathbf{e}_{s_2,2},$$

and $A_l = (a_{i_l j_l, l})$, $i_l = 1, \cdots, n_l$, $j_l = 1, \cdots, n_l$, $l = 1, 2$.

We shall show that $\mathbf{y}$ can be obtained via transformation of $A_1 \otimes A_2$ into a tensor, followed by a tensor multiplication and two index contractions.

The matrix $A = A_1 \otimes A_2$ is associated with the $(n_1 \cdot n_2, n_1 \cdot n_2)$-tensor

$$\begin{aligned}T_A &= T_{A_1} \otimes T_{A_2} \\ &= \sum_{i_1, j_1} a_{i_1 j_1, 1} \mathbf{e}_{i_1,1} \otimes \mathbf{e}_{j_1,1}^* \otimes \sum_{i_2, j_2} a_{i_2 j_2, 2} \mathbf{e}_{i_2,2} \otimes \mathbf{e}_{j_2,2}^* \\ &= \sum_{i_1, i_2, j_1, j_2} a_{i_1 j_1, 1} a_{i_2 j_2, 2} \mathbf{e}_{i_1,1} \otimes \mathbf{e}_{i_2,2} \otimes \mathbf{e}_{j_1,1}^* \otimes \mathbf{e}_{j_2,2}^*.\end{aligned}$$

As in the one-dimensional case, we compute $c_3^1 c_4^2(T_A \otimes \mathbf{x})$.

We observe that this is equivalent to

$$c_3^1(T_{A_1} \otimes c_3^1(T_{A_2} \otimes \mathbf{x})) = c_3^1\Big(T_{A_1} \otimes \sum_{i_2} \sum_{s_1}\Big) \sum_{s_2} a_{i_2 s_2, 2} x_{s_1 s_2} (\mathbf{e}_{i_2,2} \otimes \mathbf{e}_{s_1,1})$$

$$= \sum_{i_1} \sum_{i_2} \Big(\sum_{s_1} a_{i_1 s_1, 1} \Big(\sum_{s_2} a_{i_2 s_2, 2} x_{s_1 s_2}\Big)\Big) \mathbf{e}_{i_1,1} \otimes \mathbf{e}_{i_2,2},$$

which by definition is equal to the desired product

$$(A_1 \otimes A_2)\mathbf{x} = \mathbf{y}.$$

In order to simplify the notation, we shall, in the following, omit the indices identifying the spaces.

We shall systematically use the indices i_l, j_l, s_l for objects associated with U_l or its dual, which should make clear to what spaces, basis, etc. we are referring. Also, we will use, when possible, multi-indices $\mathbf{i} = (i_1, \cdots i_k)$, etc.

With this notation, we have $\mathbf{e}_\mathbf{s} \equiv \mathbf{e}_{s_1} \otimes \cdots \otimes \mathbf{e}_{s_k}$.

We consider now the general case $k \geqq 2$.

Let $\mathbf{x} = \sum_\mathbf{j} x_\mathbf{j} \mathbf{e}_\mathbf{j}$, and, for $A = A_1 \otimes \cdots \otimes A_k$, let

$$T_A = T_{A_1} \otimes \cdots \otimes T_{A_k}. \tag{3.1}$$

It is well known that

$$\mathbf{y} = (A_1 \otimes \cdots \otimes A_k)\mathbf{x} = \sum_\mathbf{i} \Big(\sum_\mathbf{j} a_{i_1 j_1} \cdots a_{i_k j_k} \mathbf{x}_\mathbf{j}\Big) \mathbf{e}_\mathbf{i}.$$

We shall obtain $\mathbf{y}$ recursively, in a form appropriate for implementation on a digital computer.

We define $\mathbf{y}^{(k)} = \mathbf{x}$, and in general

$$\mathbf{y}^{(k-t-1)} = c_{k+1}^1(T_{A_{k-t}} \otimes \mathbf{y}^{(k-t)}), \qquad t = 0, 1, \cdots, k-1. \tag{3.2}$$

In the following lemma, we prove that the tensor $\mathbf{y}^{(0)}$ obtained in this form is really $\mathbf{y} = A\mathbf{x}$.

We use the following notation:

$$\mathbf{i}_t = (i_1, \cdots, i_t), \qquad \mathbf{i}_t^* = (i_{t+1}, \cdots, i_k).$$

LEMMA 1. *Let* $\mathbf{y}^{(0)}$ *be obtained recursively by formula* (3.2). *Then* $\mathbf{y}^{(0)} = \mathbf{y}$.

Proof. By formula (3.2),

$$\begin{aligned} \mathbf{y}^{(k-1)} &= c_{k+1}^1(T_{A_k} \otimes \mathbf{y}^{(k)}) = c_{k+1}^1(T_{A_k} \otimes \mathbf{x}) \\ &= c_{k+1}^1\Big(\Big(\sum_{\mathbf{i}_k, \mathbf{j}_k} a_{i_k j_k} \mathbf{e}_{i_k} \otimes \mathbf{e}_{j_k}^*\Big) \otimes \sum_\mathbf{i} x_\mathbf{s} \mathbf{e}_\mathbf{s}\Big) \\ &= \sum_{i_k} \sum_{\mathbf{s}_{k-1}} \Big(\sum_{s_k} a_{i_k s_k} x_{\mathbf{s}_{k-1} s_k}\Big) \mathbf{e}_{i_k} \otimes \mathbf{e}_{\mathbf{s}_{k-1}}. \end{aligned} \tag{3.3}$$

(Observe the change in the order of the basis vectors!)

We assume for an induction argument that

$$\mathbf{y}^{(k-t)} = \sum_{\mathbf{i}_{k-t}^*} \sum_{\mathbf{s}_{k-t}} \Big(\sum_{\mathbf{s}_{k-t}^*} a_{i_{k-t+1}, s_{k-t+1}} \cdots a_{i_k s_k} x_{\mathbf{s}_{k-t}, \mathbf{s}_{k-t}^*}\Big) \mathbf{e}_{\mathbf{i}_{k-t}^*} \otimes \mathbf{e}_{\mathbf{s}_{k-t}}.$$

If we now apply $T_{A_{k-t}}$:

$$T_{A_{k-t}} \otimes \mathbf{y}^{(k-t)} = \sum_{i_{k-t}} \sum_{\mathbf{i}_{k-t}*} \sum_{\mathbf{s}_{k-t}} \sum_{j_{k-t}} a_{i_{k-t}, j_{k-t}} y^{(k-t)}_{\mathbf{i}_{k-t}*, \mathbf{s}_{k-t}} \cdot \mathbf{e}_{i_{k-t}} \otimes \mathbf{e}_{\mathbf{i}_{k-t}*} \otimes \mathbf{e}_{\mathbf{s}_{k-t}} \otimes \mathbf{e}^*_{j_{k-t}},$$

and finally

$$\begin{aligned}
\mathbf{y}^{(k-t-1)} &= c^1_{k+1}(T_{A_{k-t}} \otimes \mathbf{y}^{(k-t)}) \\
&= \sum_{i_{k-t}} \sum_{\mathbf{i}_{k-t}*} \sum_{\mathbf{s}_{k-t-1}} \left(\sum_{s_{k-t}} a_{i_{k-t} s_{k-t}} y^{(k-t)}_{\mathbf{i}_{k-t}*, \mathbf{s}_{k-t}} \right) \cdot \mathbf{e}_{i_{k-t}} \otimes \mathbf{e}_{\mathbf{i}_{k-t}*} \otimes \mathbf{e}_{\mathbf{s}_{k-t-1}} \\
&= \sum_{\mathbf{i}_{k-t-1}*} \sum_{\mathbf{s}_{k-t-1}} \left(\sum_{\mathbf{s}_{k-t-1}*} a_{i_{k-t} s_{k-t}} \cdots a_{i_k s_k} x_{\mathbf{s}_{k-t-1} \mathbf{s}_{k-t-1}*} \right) \cdot \mathbf{e}_{\mathbf{i}_{k-t-1}*} \otimes \mathbf{e}_{\mathbf{s}_{k-t-1}},
\end{aligned}$$

as we wished to prove.

Clearly, $\mathbf{y}^{(0)} = A\mathbf{x}$. □

4. Computer Implementation. The implementation of the computation of $A\mathbf{x}$, using a high level computer language, is facilitated very much by the developments of Section 3.

Our aim is to avoid the wasteful use of multi-indexed arrays by organizing the data in such a way that its processing is as sequential as possible. This will tend to minimize index manipulation and paging.

The main bridge between multilinear algebra and computer programming is provided by the following storage convention:

Storage Convention. The components $t_{\mathbf{i}}$ of tensors of the form

$$T = \sum_{\mathbf{i}} t_{\mathbf{i}} \mathbf{e}_{i_1} \otimes \mathbf{e}_{i_2} \otimes \cdots \otimes \mathbf{e}_{i_k}$$

will be stored in one-dimensional arrays according to:

```
begin
   j := 1;
   for i1 := 1 step 1 until n1 do
   for i2 := 1 step 1 until n2 do
            .
            .
            .
   for ik := 1 step 1 until nk do
   begin
   T[j] := t_{i1 i2 ... ik};
   j := j + 1;
   end
end;
```

The implied order of the factor spaces $U_{i_1}, U_{i_2}, \cdots, U_{i_k}$, given by the subindices of the basis vectors, is of primary importance. That order changes in the recursive algorithm of Section 3, and this will be the cue for the data handling.

We assume that the matrices A_i are stored row by row in a linear array A, starting with A_k and going backwards up to A_1. Thus

$$A(1) \leftarrow a_{11}^k,\ A(2) \leftarrow a_{12}^k,\ \cdots,\ A(n_k^2 + 1) \leftarrow a_{11}^{k-1},\ \cdots,\ A((n_1 \cdot n_2 \cdots n_k)^2) \leftarrow a_{n_1 n_1}^1.$$

We also assume that the components of the vector $\mathbf{x}$ are given in a linear array X as explained in the storage convention above.

We define $m_k = \prod_{i \neq k} n_i$.

With the data stored in this fashion, the first contraction $c_{k+1}^1(T_{A_k} \otimes \mathbf{y}^{(k)})$ is readily obtained by means of the code (confront (3.3)):

```
procedure CONTRACT (nk, mk, N, A, X); integer nk, mk, N; real array A, X;
        begin       integer I, J, k, Inic, i, s, t;
                    real SUM;
                    real array Y[1 : N];
        label 1:    k := 1; Inic := 1;
        label 2:    for i := 1 step 1 until nk do
                    begin J := 1;
                          for s : = 1 step 1 until mk do
                          begin I := Inic; SUM := 0;
                                for t := 1 step 1 until nk do
                                begin SUM := SUM + A[I] × X[J];
                                      I := I + 1; J := J + 1;
                                end;
                            Y[k] := SUM; k := k + 1;
                          end;
                        Inic : = I;
        label 3:    end;
        for i := 1 step 1 until N do
        X[i] := Y[i];
finish: end;
```

This code follows exactly the ordering of the indices indicated by the tensor products and the storage convention. It produces a result $\mathbf{y}^{(k-1)}$ that is stored in such a way that the next contraction can also be performed sequentially, and so on.

Observe that the index s has taken the place of the multi-index $\mathbf{s}_{k-1}$ of (3.3). This index s simply counts the number of subvectors of dimension n_k in which the one-dimensional array X must be subdivided.

If we put $m_i = \prod_{i \neq j} n_i$, $N = \prod_{j=1}^k n_j$, and $M = \sum_{j=1}^k n_j^2$, then the complete product $\mathbf{y} = A\mathbf{x}$ can be obtained by means of the following procedure:

```
procedure TENSOR PRODUCT (M, N, k, n, A, X); integer M, N, k;
                         integer array n[1 : k]; real array A[1 : M]; X[1 : N];
        begin            integer mk, nk, L;
                         nk := n[k]; mk := N/nk;
        other matrix:    CONTRACT (nk, mk, N, A, X);
                         if (k = 1) go to finish;
                         k := k − 1; mk := N/n[k];
                         go to other matrix;
finish: end;
```

5. Tensor Product Systems of Equations. We would like to consider now the solution of systems of equations of the form

$$(W_1 \otimes \cdots \otimes W_k)\mathbf{y} = \mathbf{x} \tag{5.1}$$

where the W_i are nonsingular linear operators from the n_i-dimensional vector space U_i $(i = 1, \cdots, k)$ into itself. Naturally, $\mathbf{x}, \mathbf{y} \in U_1 \otimes \cdots \otimes U_k$.

From (2.5), we have that the solution to this system is simply given by

$$\mathbf{y} = (W_1^{-1} \otimes \cdots \otimes W_k^{-1})\mathbf{x}. \tag{5.2}$$

Thus, after inversion of the W_i, we can apply the algorithm of Section 4 directly. Of course, it is seldom wise to solve a system of equations by inverting the matrix of coefficients [2, Chapter 2, Section 1.1], and we shall look for a different approach.

Let V_i, L_i be upper and lower triangular matrices respectively such that $V_i = L_i W_i$. Thus,

$$\begin{aligned}(L_1 \otimes L_2 \otimes \cdots \otimes L_k)\cdot(W_1 \otimes W_2 \otimes \cdots \otimes W_k)\\ = (V_1 \otimes V_2 \otimes \cdots \otimes V_k).\end{aligned} \tag{5.3}$$

The system (5.1) becomes, after multiplication by $\otimes L_i$,

$$(V_1 \otimes \cdots \otimes V_k)\mathbf{y} = (L_1 \otimes \cdots \otimes L_k)\mathbf{x}. \tag{5.4}$$

We assume that the nonzero elements of the matrices V_i and L_i are stored in one-dimensional arrays V and L in the same way as the A_i were stored in Section 4.

With a small modification in the code to take into account the special form of the factors L_i, we can use the same procedure described in Section 4 in order to obtain

$$\mathbf{b} = (L_1 \otimes \cdots \otimes L_k)\mathbf{x}. \tag{5.5}$$

To compute the solution of the system $(V_1 \otimes \cdots \otimes V_k)\mathbf{y} = \mathbf{b}$ given by

$$\mathbf{y} = (V_1^{-1} \otimes \cdots \otimes V_k^{-1})\mathbf{b}, \tag{5.6}$$

basically, we will use, recursion (3.2).

For this purpose, A_{k-t} is replaced by V_{k-t}^{-1}, and $\mathbf{y}^{(k)}$ by $\mathbf{b}$. If we make the assumption that $\mathbf{b}$ and all the intermediary vectors $\mathbf{y}^{(k-t)}$ are stored in the same way as in (3.2), then each step of the recursion, in this case is equivalent to the solution of m_{k-t} upper triangular systems with matrix of coefficients V_{k-t}. The right-hand sides of these systems are the vectors obtained by partitioning the present $\mathbf{y}^{(k-t)}$ in subvectors of length n_{k-t}.

Here we have to point out an important difference between the algorithm of Section 4 and the present one. In the code described in Section 4, the vectors $\mathbf{y}^{(k-t-1)}$ resulted automatically in the appropriate storage mode. This was a consequence of a convenient ordering of the loops, and it was possible because the components of $\mathbf{y}^{(k-t-1)}$ were computed one at a time.

In the present situation though, we shall process $\mathbf{y}^{(k-t)}$ by whole blocks of length n_{k-t}, and the returning blocks *will not be in the proper ordering* for sequential processing of $\mathbf{y}^{(k-t-1)}$.

Let O_t be the ordering associated with the tensor product space

$$U = U_{k-t+1} \otimes U_{k-t+2} \otimes \cdots \otimes U_k \otimes U_1 \otimes U_2 \otimes \cdots \otimes U_{k-t}, \qquad t = 0, 1, \cdots, k-1,$$

according to the storage convention of Section 4.

For a given multi-index s, let $\pi\mathbf{s} = \hat{\mathbf{s}}$ be the cyclic permutation that sends s_i into $\hat{s}_{i+1}$ if $1 \leqq i < k$, and s_k into $\hat{s}_1$.

For $0 \leqq t \leqq k-1$, π^t is defined by repeated application of π. Clearly, $(\mathbf{s}^*_{k-t}, \mathbf{s}_{k-t}) = \pi^t\mathbf{s}$.

We shall also use $\mathbf{K} = (1, \cdots, k)$.

For a given tensor $B \in U$, each of its components

$$b_{s_{k-t+1}, s_{k-t+2}, \cdots, s_k, s_1, \cdots, s_{k-t}} = b_{\mathbf{s}^*_{k-t}, \mathbf{s}_{k-t}} = b_{\pi^t\mathbf{s}}$$

has the linear address (in the order O_t, and calling $\hat{\mathbf{s}} \equiv \pi^t\mathbf{s}$, $\hat{\mathbf{K}} = \pi^t\mathbf{K}$)

$$\text{(5.7)} \qquad \mathrm{add}_{O_t}(b_{\hat{\mathbf{s}}}) = \sum_{j=1}^{k-1} (\hat{s}_j - 1) \prod_{i=j+1}^{k} n_{\hat{K}_i} + \hat{s}_k.$$

Putting $p - 1 = \sum_{j=1}^{k-2} (\hat{s}_j - 1) \prod_{i=j+1}^{k-1} n_{\hat{K}_i} + (\hat{s}_{k-1} - 1)$, $q = \hat{s}_k$, we obtain, from (5.7),

$$\text{(5.8)} \qquad \mathrm{add}_{O_t}(b_{\hat{\mathbf{s}}}) = (p-1)n_{\hat{K}_k} + q.$$

Observe that p is the block number in the partition of B in blocks of size $n_{\hat{K}_k}$, while q is the position of the component in its block.

We now need to compute the linear address corresponding to $b_{\mathbf{s}^*_{k-t}, \mathbf{s}_{k-t}}$ in the ordering O_{t+1}, associated with the tensor product $U_{\mathbf{s}^*_{k-t-1}} \otimes U_{\mathbf{s}_{k-t-1}}$.

Of course, we know that if $\mathbf{s}' = \pi\mathbf{s}$ and $\mathbf{K}' = \pi\hat{\mathbf{K}}$ then

$$\text{(5.9)} \qquad \mathrm{add}_{O_{t+1}}(b_{\hat{\mathbf{s}}}) = (s'_1 - 1) \prod_{i=2}^{k} n_{K_i'} + \sum_{j=2}^{k-1} (s'_j - 1) \prod_{i=j+1}^{k} n_{K_i'} + s'_k.$$

But, since $n_{K_i'} = n_{\hat{K}_k}$, it turns out that $\prod_{i=2}^{k} n_{K_i'} = m_{\hat{K}_k}$. Also, since $s'_j = \hat{s}_{j-1}$, we have that

$$\sum_{j=2}^{k-1} (s'_j - 1) \prod_{i=j+1}^{k} n_{K_i'} + s'_k = \sum_{j=1}^{k-2} (\hat{s}_j - 1) \prod_{i=j+1}^{k-1} n_{\hat{K}_i} + \hat{s}_{k-1} = p,$$

and $s'_1 = \hat{s}_k = q$, where p and q were defined just before formula (5.8).

Therefore, replacing these values in (5.9), we obtain the simple address mapping

$$\text{(5.10)} \qquad \mathrm{add}_{O_{t+1}}(b_{\pi^t\mathbf{s}}) = (q-1)m_{\hat{K}_k} + p.$$

In the Appendix, we give an ALGOL *procedure* which reorders the returning blocks of the process of $\mathbf{b}^{(k-t)}$, using formulae (5.8), (5.10).

6. Numerical Tests and Possible Applications. The algorithm described in the earlier sections was used for the solution of systems of the form $(W_1 \otimes \cdots \otimes W_k)\mathbf{y} = \mathbf{x}$ with W_i transposed Vandermonde matrices. The program was written in FORTRAN-G and tested on an IBM 360/50 computer. It was run in double-precision, which corresponds to 14 hexadecimal digits in the mantissa. We shall refer to this program as *procedure* TENSP.

For the solution of the resulting transposed Vandermonde systems, we used a double-precision version of the *procedure* dvand (see [1]).

The results of our *procedure* TENSP were compared with a program that uses the full Kronecker product matrix and solves the system (5.1) by Gaussian elimination using the IBM supplied routine DGELG. It is quite obvious that this second approach is counter indicated in practically all cases, but we include it since we know of cases in which it is used anyway. Our hopes of seeing a complete breakdown due to ill-conditioning were disappointed because another factor made it impossible to proceed with values of n above 5: storage. In fact, for $k = 3$ and $n = 6$, we would have needed 373248 bytes of memory, just for storing the matrix! Also, the factor time was growing very fast. Compare with the relevant data for our algorithm. Observe also that there is a significant loss of accuracy.

The table below shows the results obtained in the case $k = 3$, with all the W_i equal and for different sizes $n_i \equiv n + 1$.

TABLE 1

$n + 1$	$\|\|r\|\|_\infty$ TENSP*	$\|\|r\|\|_\infty$ DGELG
3	—	0.0
4	0.4996×10^{-15}	0.3035×10^{-13}
5	0.0	0.2043×10^{-10}
6	0.63144×10^{-14}	—
10	0.7835×10^{-15}	—
15	0.3472×10^{-14}	—

* $\|\|r\|\|_\infty$ is the maximum norm of the difference between the exact and the computed solutions.

The vector alpha $[0 : n]$ defines the matrices W_i, and its elements are of the form alpha $[j] = j/n$.

Given a function $f(z_1, z_2, z_3)$ and an appropriate right-hand side vector $\mathbf{x}$, the solution to the system of linear equations will be formed by the coefficients of the polynomial

$$P(z_1, z_2, z_3) = \sum_{i_1, i_2, i_3 = 0}^{n} y_{i_1 i_2 i_3} z_1^{i_1} z_2^{i_2} z_3^{i_3}$$

that interpolates f at certain points.

In fact, if we consider the function $f(z_1, z_2, z_3) = z_1^n$, and define $\mathbf{x}$ by

```
L := 1;
for I := 0 step 1 until n do
for J := 0 step 1 until n do
```

```
or  k := 0 step 1 until n do
    begin
    X[L] := alpha [I] ** n;
    L := L + 1;
    end;
```

then we shall have that $P(z_1, z_2, z_3) = z_1^n$, since $f(z_1, z_2, z_3)$ is interpolated exactly. Thus, the solution vector will be

$$y_{(n+1)^2+1} = 1; \qquad y_i = 0 \quad \text{otherwise}.$$

Some further data on these computer runs:

CPU time for DGELG; $n = 5$: 2′42.39″.
CPU time for TENSP; $n = 5, 10, 15$: 55.40″.
Storage for TENSP; $n = 15$: 64000 bytes.
Storage for DGELG; $n = 5$: 150,000 bytes.

By using the *procedures* pvand of [1], or vanderconf and dualconf of [5], it is possible to apply this algorithm to a large variety of problems in multidimensional interpolation of the Lagrange and Hermite type, numerical hyper-cubature, construction of tensor product finite elements, etc. The most remarkable points, we think, are the efficiency and simplicity of the algorithms and, overall, their ability to solve problems which are traditionally avoided because of their purported ill-conditioning. We mean by this, problems involving the solution of Vandermonde systems which, for moderate sizes, can be solved directly, accurately, and efficiently by our methods. See [1], [5] for more details.

Appendix. Below, we give an ALGOL *procedure* for the reordering of $\mathbf{y}^{(k-t)}$ (see Section 5, (5.8), (5.10)).

```
procedure ORDENAR (X, Y, mk, nk, N);
Comment Parameter list:
X[t]: starting vector, length: ∏_{i=1}^{k} n_i = m_k × n_k = N,
Y[s]: reordered vector;
integer mk, nk, N, t, I, J, s;
real array X, Y;
begin     t := 1;
          for J := 1 step 1 until mk do
          for I := 1 step 1 until nk do
          begin     s := (I − 1) × mk + J;
                    Y[s] := X[t];
                    t = t + 1;
          end;
end;
```

Departamento de Computación
Universidad Central de Venezeula
Apartado 59002
Caracas, Venezuela 105

Departamento de Física Atómica y Molecular
Instituto Venezolano de Investigaciones Científicas
Apartado 1827
Caracas, Venezuela

1. Å. BJÖRCK & V. PEREYRA, "Solution of Vandermonde systems of equations," *Math. Comp.*, v. 24, 1970, pp. 893–903. MR **44** #7721.

2. E. ISAACSON & H. B. KELLER, *Analysis of Numerical Methods,* Wiley, New York, 1966. MR **34** #924.

3. N. JACOBSON, *Lectures in Abstract Algebra,* Vol. 1, Van Nostrand, Princeton, N.J., 1951. MR **12,** 794.

4. G. BIRKHOFF, M. H. SCHULTZ & R. S. VARGA, "Piecewise Hermite interpolation in one and two variables with applications to partial differential equations," *Numer. Math.*, v. 11, 1968, pp. 232–256. MR **37** #2404.

5. G. GALIMBERTI & V. PEREYRA, "Solving confluent Vandermonde systems of Hermite type," *Numer. Math.*, v. 18, 1971, pp. 44–60.

6. P. J. DAVIS & P. RABINOWITZ, *Numerical Integration,* Blaisdell, Waltham, Mass., 1967. MR **35** #2482.