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 \((n \cdot p^k)\) words of information for \(A_i, A_i \in \mathcal{L}(E_i, E_i)\) instead of the tensor-product operators themselves \((p^k)\) 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 \((A_1 \otimes \cdots \otimes A_s)x\). 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

\[(A_1 \otimes \cdots \otimes A_s)x = b.\]

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

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

$$\forall y \in U, x^*(y) = \langle x, y \rangle$$

(where $\langle , \rangle$ denotes inner product).

The tensor product space $U$ formed with $U_1, U_2, \ldots, U_k$ 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

$$x_1 \otimes x_2 \otimes \cdots \otimes x_k, \quad \text{with} \quad x_i \in U_i.$$

Let us suppose that the tensor product space $U$ is formed with $s$ spaces $\{U_i\}, i = 1, 2, \ldots, s$, and $t$ spaces $\{U_j^*\}, j = 1, 2, \ldots, t$, where the $U_j^*$ 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 $U = U_1 \otimes \cdots \otimes U_s \otimes U_1^* \otimes \cdots \otimes U_t^*$ can be generated as a linear combination of elements of the form

$$x_1 \otimes \cdots \otimes x_s \otimes x_1^* \otimes \cdots \otimes x_t^*,$$

where the $x_i \in U_i, x_i^* \in U_i^*$.

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

$$c'_i(x) = c'_i(x_1 \otimes \cdots \otimes x_i \otimes x_i^* \otimes \cdots \otimes x_t^*)$$

$$= x_i^*(x_i) \cdot (x_1 \otimes \cdots \otimes x_i \otimes x_i^* \otimes \cdots \otimes x_t^*),$$

where $i_s = i_i$.

The operator $c'_i$ is extended linearly to all $U$ and is called the contraction of the $i$th contravariant with the $i$th covariant component.

Given two tensors $u, v$, belonging to two spaces $U, V$, we define the tensor product of $u$ and $v$ as the element of $U \otimes V$:
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 \( \{e_i\} \) is a basis for \( U \), and \( \{e^*_i\} \) is the dual basis, then we can associate with \( A \) (in a 1-1 correspondence) the tensor \( T_A = \sum_i (Ae_i) \otimes e^*_i \), and obtain \( Ax \) through tensor multiplication and contraction

\[
(2.3) \quad Ax = \sum_i A e_i \otimes x \otimes e^*_i = \sum_i e^*_i(x) A e_i = \sum_i x_j A e_j.
\]

Observe that \( A e_i = a_i \) is the \( j \)th column of the matrix representation of \( A \) in the basis \( \{e_i\} \).

If \( A_{ij}, i = 1, \ldots, k, \) are linear operators from \( U_i \) into \( U_j \), 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

\[
(2.4) \quad (A_1 \otimes \cdots \otimes A_k)(x_1 \otimes \cdots \otimes x_k) = A_1 x_1 \otimes \cdots \otimes A_k x_k,
\]

and extended to all \( U \) by linearity.

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

\[
(2.5) \quad (A_1 \otimes \cdots \otimes A_k)^{-1} = A_1^{-1} \otimes \cdots \otimes A_k^{-1}.
\]

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 \( y = (A_1 \otimes A_2 \otimes \cdots \otimes A_k) x \), using a minimum of storage and index manipulation.

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

\[
 y = (A_1 \otimes A_2)x, \quad \text{where} \quad x = \sum_{i_1} \sum_{i_2} x_{i_1, i_2} e_{i_1, 1} \otimes e_{i_2, 2},
\]

and \( A_i = (a_{i_1 i_2 lx}), i_1 = 1, \ldots, n_1, j_1 = 1, \ldots, n_2, l_1 = 1, 2 \).

We shall show that \( y \) 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

\[
T_A = T_{A_1} \otimes T_{A_2}
\]

\[
= \sum_{i_1, i_2} \sum_{j_1, j_2} a_{i_1 j_1} e_{i_1, 1} \otimes e^*_{j_1, 1} \otimes \sum_{i_2, i_2} \sum_{j_2, j_2} a_{i_2 j_2} e_{i_2, 2} \otimes e^*_{j_2, 2}
\]

\[
= \sum_{i_1, i_2, j_1, j_2} a_{i_1 j_1} a_{i_2 j_2} e_{i_1, 1} \otimes e^*_{j_1, 1} \otimes e_{i_2, 2} \otimes e^*_{j_2, 2}.
\]

As in the one-dimensional case, we compute \( c_1^c_2 T_A \otimes x \).
We observe that this is equivalent to

\[ c^1_0(T_A \otimes c^1_0(T_A \otimes x)) = c^1_0\left( T_A \otimes \sum_{i_a} \sum_{s_a} a_{s_a,2}x_s e_{s,1} \otimes e_{s,1} \right) \]

\[ = \sum_{i_a} \sum_{s_a} \left( \sum_{i_s} a_{i_s,1} \left( \sum_{s_s} a_{s_s,2}x_{s_s} e_{s_s,1} \right) \right) e_{i_s,1} \otimes e_{s_s,2}, \]

which by definition is equal to the desired product

\[(A_1 \otimes A_2)x = y.\]

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

We shall systematically use the indices \(i, j, s\) for objects associated with \(U_i\) or its dual, which should make clear to what spaces, basis, etc. we are referring. Also, we will use, when possible, multi-indices \(i = (i_1, \ldots, i_s)\), etc.

With this notation, we have \(e_s = e_{i_s} \otimes \cdots \otimes e_{i_1}\).

We consider now the general case \(k \geq 2\).

Let \(x = \sum_j x_j e_j\), and, for \(A = A_1 \otimes \cdots \otimes A_k\), let

\[(3.1) \quad T_A = T_{A_1} \otimes \cdots \otimes T_{A_k}.\]

It is well known that

\[ y = (A_1 \otimes \cdots \otimes A_k)x = \sum_i \left( \sum_j a_{i,j} \cdots a_{i,1} x_j \right) e_i. \]

We shall obtain \(y\) recursively, in a form appropriate for implementation on a digital computer.

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

\[(3.2) \quad y^{(k-t-1)} = c_{k+1}(T_{A_{k-t}} \otimes y^{(k-t)}), \quad t = 0, 1, \ldots, k - 1.\]

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

We use the following notation:

\[i = (i_1, \ldots, i_s), \quad i^* = (i_{s+1}, \ldots, i_k).\]

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

**Proof.** By formula (3.2),

\[(3.3) \quad y^{(k-1)} = c_{k+1}(T_{A_k} \otimes y^{(k)}) = c_{k+1}(T_{A_k} \otimes x) \]

\[= c_{k+1}\left( \left( \sum_{i_{k+1}} a_{i_{k+1}} e_{i_{k+1}} \otimes e_{i_{k+1}}^* \right) \otimes \sum_i x_i e_i \right) \]

\[= \sum_{i_{k+1}} \sum_{i_{k+1}} \left( \sum_{a_{k+1}} a_{i_{k+1}} x_{i_{k+1}} e_{i_{k+1}} \right) e_{i_{k+1}} \otimes e_{i_{k+1}}. \]

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

We assume for an induction argument that

\[y^{(k-t)} = \sum_{i_{k-t}} \sum_{i_{k-t}} \left( \sum_{a_{k-t}} a_{i_{k-t}} x_{i_{k-t}} \cdots a_{i_{k-t}} x_{i_{k-t}} \right) e_{i_{k-t}} \otimes e_{i_{k-t}}.\]
If we now apply $T_{A_{k-l}}$:

$$T_{A_{k-l}} \otimes y^{(k-l)} = \sum_{i=1}^{n_1} \sum_{i=2}^{n_2} \sum_{i=3}^{n_3} a_{i_{k-l}, i_{k-l}, s_{k-l}} y_{i_{k-l}, i_{k-l}, s_{k-l}}$$

and finally

$$y^{(k-l-1)} = c^{k-l+1}_{k-l+1}(T_{A_{k-l}} \otimes y^{(k-l)})$$

as we wished to prove.

Clearly, $y^{(0)} = Ax$. □

4. Computer Implementation. The implementation of the computation of $Ax$, 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 $t_i$ of tensors of the form

$$T = \sum_i t_i e_i \otimes e_{i_2} \otimes \cdots \otimes e_{i_k}$$

will be stored in one-dimensional arrays according to:

begin

\begin{align*}
    j & := 1; \\
    \text{for } i_1 & := 1 \text{ step 1 until } n_1 \text{ do} \\
    \text{for } i_2 & := 1 \text{ step 1 until } n_2 \text{ do} \\
    \quad & \\
    \quad \quad \text{for } i_k & := 1 \text{ step 1 until } n_k \text{ do} \\
    \quad \quad \quad \text{begin} \\
    \quad \quad \quad T[j] & := t_{i_1, i_2, \ldots, i_k}; \\
    \quad \quad \quad j & := j + 1; \\
    \quad \quad \text{end} \\
    \quad \text{end;}
\end{align*}

end;

The implied order of the factor spaces $U_{i_1}, U_{i_2}, \ldots, U_{i_k}$, 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_{i_1}^{k}, \quad A(2) \leftarrow a_{i_2}^{k}, \ldots, \quad A(n_k + 1) \leftarrow a_{i_1}^{k-1}, \ldots, \quad A((n_1 \cdot n_2 \cdots n_k)^2) \leftarrow a_{i_n}^{n}. $

We also assume that the components of the vector $x$ 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 $c_{k+1}^{l}(T_{A_k} \otimes y^{(k)})$ is readily obtained by means of the code (confront (3.3)):

```plaintext
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 / := 1 step 1 wzz/z7 nk do
                            begin
                                SUM := SUM + A[I] X X[J];
                                /:= /+ 1;/:- J+ 1;
                            end;
                        Y[k] := SUM; k := k + 1;
                    end;
                Inic := I;
                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 $y^{(k-1)}$ 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 $s_{k-1}$ of (3.3). This index $s$ simply counts the number of subvectors of dimension $n_s$ in which the one-dimensional array $X$ must be subdivided.

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

```plaintext
procedure TENSOR PRODUCT (M, N, k, n, A, X); integer M, N, k;
begin
    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;
```

License or copyright restrictions may apply to redistribution; see https://www.ams.org/journal-terms-of-use
5. Tensor Product Systems of Equations. We would like to consider now the solution of systems of equations of the form

\[(W_1 \otimes \cdots \otimes W_k)y = x\]

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

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

\[y = (W_1^{-1} \otimes \cdots \otimes W_k^{-1})x.\]

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,

\[(L_1 \otimes L_2 \otimes \cdots \otimes L_k) \cdot (W_1 \otimes W_2 \otimes \cdots \otimes W_k) = (V_1 \otimes V_2 \otimes \cdots \otimes V_k).\]

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

\[(V_1 \otimes \cdots \otimes V_k)y = (L_1 \otimes \cdots \otimes L_k)x.\]

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

\[b = (L_1 \otimes \cdots \otimes L_k)x.\]

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

\[y = (V_1^{-1} \otimes \cdots \otimes V_k^{-1})b,\]

basically, we will use, recursion (3.2).

For this purpose, \(A_{k-1}\), is replaced by \(V_{k-1}^{-1}\), and \(y^{(k)}\) by \(b\). If we make the assumption that \(b\) and all the intermediary vectors \(y^{(k-1)}\) 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 \(m_{k-1}\) upper triangular systems with matrix of coefficients \(V_{k-1}\). The right-hand sides of these systems are the vectors obtained by partitioning the present \(y^{(k-1)}\) into subvectors of length \(n_{k-1}\).

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 \(y^{(k-1)}\) 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 \(y^{(k-1)}\) were computed one at a time.

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

Let \(O_i\) be the ordering associated with the tensor product space.
\[
U = U_{k-t+1} \otimes U_{k-t+2} \otimes \cdots \otimes U_k \otimes U_1 \otimes U_2 \otimes \cdots \otimes U_{k-t},
\]
\(t = 0, 1, \cdots, k - 1,\)

according to the storage convention of Section 4.

For a given multi-index \(s\), let \(\pi s = \hat{s}\) be the cyclic permutation that sends \(s_i\) into \(s_{i+1}\) if \(1 \leq i < k\), and \(s_k\) into \(s_1\).

For \(0 \leq t \leq k - 1\), \(\pi^t\) is defined by repeated application of \(\pi\). Clearly, \((s_{k-t}, s_{k-1}) = \pi^t s\).

We shall also use \(K = (1, \cdots, k)\).

For a given tensor \(B \subseteq U\), each of its components

\[
b_{s-k+1, s_{k-2}, \ldots, s_1, s_0} = b_{s-k+1, s_{k-1}} = b_{\pi^t s}
\]

has the linear address (in the order \(O_t\), and calling \(s = x\) and \(K = \pi^t K\))

\[
(add_{O_t}(b_{\hat{s}}) = \sum_{j=1}^{k-1} (\hat{s}_j - 1) \prod_{i=1}^{k-1} n_{\hat{s}_i} + \hat{s}_k.
\]

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

\[
(add_{O_t}(b_{\hat{s}}) = (p - 1)n_{\hat{s}_k} + q.
\]

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

We now need to compute the linear address corresponding to \(b_{s-k+1, s_{k-2}, \ldots, s_1, s_0}\), in the ordering \(O_{t+1}\), associated with the tensor product \(U_{s-k+1} \otimes U_{s_{k-2}} \otimes \cdots \otimes U_{s_0}\).

Of course, we know that if \(s' = \pi s\) and \(K' = \pi K\), then

\[
(add_{O_{t+1}}(b_{\pi s}) = (s'_k - 1) \prod_{i=2}^{k} n_{K'_i} + \sum_{j=2}^{k-1} (s'_j - 1) \prod_{i=2}^{k-1} n_{K'_i} + s'_k.
\]

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

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

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

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

\[
(add_{O_{t+1}}(b_{\pi s}) = (q - 1)m_{\hat{s}_k} + p.
\]

In the Appendix, we give an ALGOL procedure which reorders the returning blocks of the process of \(b^{(k-1)}\), 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 \((W_1 \otimes \cdots \otimes W_k)y = x\) with \(W_1\) 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 TENS.
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 \( n_i = n + 1 \).

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

* \( ||r||_\infty \) 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 \( x \), 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 \( x \) by

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

License or copyright restrictions may apply to redistribution; see https://www.ams.org/journal-terms-of-use
or \( k := 0 \) step 1 until \( n \) do
\begin{align*}
X[L] &:= \alpha[I] \times n; \\
L &:= L + 1;
\end{align*}
end;

then we shall have that \( P(z_1, z_2, z_3) = z_i^k \), since \( f(z_1, z_2, z_3) \) is interpolated exactly.

Thus, the solution vector will be
\[
y_{(n+1)\times 1} = 1; \quad y_i = 0 \text{ otherwise.}
\]

Some further data on these computer runs:

- CPU time for DGELG; \( n = 5 \): 2’42.39”.
- CPU time for TENS; \( n = 5, 10, 15 \): 55.40”.
- Storage for TENS; \( n = 15 \): 64000 bytes.
- Storage for DGELG; \( n = 5 \): 150,000 bytes.

By using the procedures pvdand 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 \( y^{(k-1)} \)
(see Section 5, (5.8), (5.10)).

\[
\begin{align*}
&\text{procedure ORDENAR (X, Y, mk, nk, N);} \\
&\text{Comment Parameter list:} \\
&\text{X[t]: starting vector, length: } \prod_{k} n_k = N, \\
&\text{Y[s]: reordered vector;} \\
&\text{integer mk, nk, N, t, I, J, s;} \\
&\text{real array X, Y;} \\
&\text{begin} \\
&\quad t := 1; \\
&\quad \text{for } J := 1 \text{ step 1 until } mk \text{ do} \\
&\quad \text{for } I := 1 \text{ step 1 until } nk \text{ do} \\
&\quad \quad \text{begin} \\
&\quad \quad \quad s := (I - 1) \times mk + J; \\
&\quad \quad \quad Y[s] := X[t]; \\
&\quad \quad \quad t = t + 1; \\
&\quad \quad \text{end;}
&\text{end;}
\end{align*}
\]

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