\* Evaluation of B-splines for Solving Systems \* \* of Boundary Value Problems \* \* by \* Uri Ascher and Robert D. Russell \* Technical Report 77-14 \* November 1977 \* \* 

> Department of Computer Science The University of British Columbia Vancouver, British Columbia V6T 1W5

A general purpose collocation code COLSYS has been written, which is capable of solving mixed order systems of multi-point boundary value ordinary differential equations. The piecewise polynomial solution is given in terms of a B-spline basis.

Efficient implementation of algorithms to calculate with B-splines is a necessary condition for the code to be competitive. Here we describe these algorithms and the special features incorporated to take advantage of the specific environment in which they are used.

This paper was written with the financial support of the National Research Council of Canada under grants A 4306 and A 7871.

Abstract



# Evaluation of B-Splines for Solving Systems of Boundary Value Problems

by

U. Ascher<sup>†</sup> and R. D. Russell<sup>††</sup>

1. Introduction

In [1] a general purpose collocation code, COLSYS, for solving mixed order systems of multi-point boundary value ordinary differential equations is described. The piecewise polynomial approximate "global" solution is given in terms of a Bspline basis. This makes the solution process very flexible and stable.

The evaluation of the basis functions is a major cost for COLSYS, and a careful implementation of selected algorithms is needed to make the code competitive. Efficient algorithms for calculating with B-splines are given by deBoor [2], who also efficiently implements these algorithms in a Fortran package [3]. Here we describe the modifications we have made in these algorithms to better suit our needs. Our use of B-splines is somewhat special because:

- (i) we are solving a system of differential equations, so many repetitive calculations can be avoided,
- (ii) the continuity in the solution at the mesh points (knots) is more restricted here than in [2,3], and
- (iii) on many occasions we evaluate the B-splines at points which are placed in a regular fashion.
- + Computer Science Department, University of British Columbia. The work of this author was supported in part by NRC Grant #A4306.
- †† Mathematics Department, Simon Fraser University. The work of this author was supported in part by NRC Grant #A7871.

Specifically, suppose we have a set of d differential equations on an interval [a,b] of orders  $m_1 \le m_2 \le \ldots \le m_d$ . The collocation solution is constructed on a mesh

(1.1) If 
$$a = x_1 < x_2 < \dots < x_N < x_{N+1} = b$$

with the n-th component  $v_n(x)$ ,  $1 \le n \le d$ , belonging to  $P_{k+m_n,\Pi} \cap C^{(m_n-1)}[a,b]$ , by collocating at the k Gauss-Legendre points in each subinterval  $[x_i, x_{i+1}]$ i = 1, ..., N. Here

(1.2) 
$$p_{k+m_n} = \{w | w \text{ is a polynomial of degree } < k+m_n \text{ on each subinterval} \\ (x_i, x_{i+1}), 1 \le i \le N\}.$$

Let  $T = \{t_i\}$  be the knot sequence:  $j_{j=1}$ 

(1.3) 
$$t_{j} = \begin{cases} x_{1} & j \leq k + m_{d} \\ x_{i+1} & ik + m_{d} < j \leq (i+1)k + m_{d} & i=1,...,N-1 \\ x_{N+1} & Nk + m_{d} < j \end{cases}$$

Let N denote the j-th B-spline of order k [2]. The function v is given as a linear combination of the B-splines

(1.4) 
$$v_{n}(x) = \sum_{\substack{j=-k-m_{n}+2}}^{Nk} a_{j,n} a_{j,k+m_{n}}(x),$$

In fact, only k+m B-splines may be nonzero at x: If  $x \in [t_i, t_{i+1}]$ , then

(1.5) 
$$v_n(x) = \sum_{\substack{j=-k-m_n+1}}^{0} \alpha_{i+j,n} N_{i+j,k+m_n}(x),$$

We consider the efficient implementation of the following procedures for COLSYS: Evaluation of the B-splines at a point (§2), derivatives of the approxi-

mate solution (§3), derivatives of the basis functions (§4), highest order derivatives of the solution (§5), and we conclude with a brief summary of the uses of these procedures in the collocation code.

# 2. Evaluation of the B-splines at a point $x \in [t_i, t_{i+1})$

This is the evaluation of the  $N_{i+j,k+m_n}(x)$ . deBoor's algorithm reads [2]: <u>Algorithm 2A</u>: Let  $N_{i,1}(x) = 1$ . Do for  $\ell = 1, \dots, k + m_d - 1$ :  $N_{i-\ell,\ell+1}(x) = 0$ Do for  $j = 1, \dots, \ell$ :  $M_{i+j-\ell,\ell}(x) = N_{i+j-\ell,\ell}(x)/(t_{i+j} - t_{i+j-\ell})$   $N_{i+j-\ell-1,\ell+1}(x) = N_{i+j-\ell-1,\ell+1} + (t_{i+j} - x)M_{i+j-\ell,\ell}(x)$  $N_{i+j-\ell,\ell+1}(x) = (x - t_{i+j-\ell})M_{i+j-\ell,\ell}(x)$ 

The first saving when using B-splines arises because of the recursive way in which they are defined. One call to algorithm 2A produces all the B-splines that we need for all the components of the solution  $y \equiv (v_1, v_2, ..., v_d)$  and their derivatives (as we shall see). The algorithm produces the triangular array



The knot sequence (1.3) has the special structure that each interior mesh point  $x_i$  is repeated exactly k times. Since  $m_1 \leq \ldots \leq m_d \leq k$ , this implies that each B-spline has its support on at most 2 subintervals, and some of them have support on one subinterval only. Let  $\mathbf{i} = \mathbf{I}\mathbf{k} + \mathbf{m}_d$ . Then  $\mathbf{x} \in [\mathbf{x}_I, \mathbf{x}_{I+1})$ , and since  $1 \leq \mathbf{j} \leq \ell \leq \mathbf{k} + \mathbf{m}_d - 1$  in algorithm 2A,

(2.1) 
$$t_{i+j} - t_{i+j-\ell} = \begin{cases} h_{I-1} + h_{I} & \text{for } j \le k \text{ and } \ell \ge j+k \\ h_{I} & \text{for } j \le k \text{ and } \ell \le j+k-1 \\ h_{I} + h_{I+1} & \text{for } k+1 \le j \end{cases}$$

(2.2) 
$$t_{i+j} - x = \begin{cases} \rho h_{I} & \text{for } 1 \le j \le k \\ \rho h_{I} + h_{I+1} & \text{for } k+1 \le j \le k + m_{d} - 1 \text{ (or } 2k), \end{cases}$$

where

(2.3) 
$$h_{I} = X_{I+1} - X_{I}, \quad 1 \le I \le N.$$

Therefore, Algorithm 2A becomes

Algorithm 2B: Suppose  $x \in [x_1, x_{1+1}]$ . Set  $N_{i,1}(x) \equiv 1$ ;

$$\rho_1 = \frac{x_{I+1} - x}{x_{I+1} - x_{I-1}}; \quad \rho_2 = \frac{x_{I+1} - x}{x_{I+1} - x_I}; \quad \rho_3 = \frac{x_{I+1} - x}{x_{I+2} - x_I}.$$

Do for 
$$\ell = 1, ..., k$$
:  $N_{i-\ell,\ell+1}(x) = 0$   
(a)  
Do for  $j = 1, ..., \ell$ :  
(\*)  
 $N_{i+j-\ell-1,\ell+1}(x) = N_{i+j-\ell-1,\ell+1}(x) + \rho_2 N_{i+j-\ell,\ell}(x)$   
 $N_{i+j-\ell,\ell+1}(x) = (1 - \rho_2)N_{i+j-\ell,\ell}(x)$ 

Do for 
$$\ell = k + 1$$
, ...,  $k + m_d - 1$ :  $N_{i-\ell,\ell+1}(x) = 0$   
Do for  $j = 1, ..., \ell - k$ :  
(b)  $(*)$  with  $\rho_1$  replacing  $\rho_2$   
Do for  $j = \ell - k + 1$ , ...,  $k$ :  
(c)  $(*)$  with  $\rho_2$   
Do for  $j = k + 1, ..., \ell$ :  
(d)  $(*)$  with  $\rho_3$  replacing  $\rho_2$ 

Our implementation of algorithm 2B was found to be about 50% faster than the routine BSPLVN in [3] on the IBM 370/168. (This is somewhat machine dependent because a division operation is replaced by a multiplication, and memory references to arrays are saved). Note also that some storage is saved by eliminating the knot sequence.

There is another major saving to be done here. In algorithm 2B,  $\rho_2$  depends only on the relative position of x in its subinterval. But the collocation points have exactly the same relative position in each subinterval. Thus loops (a) and (c) can be evaluated once and stored for each of the k Gaussian points. used in [1], and while assembling the collocation equations, only loops (b) and (d) have to be executed kN times. The same remarks hold for any set of "regular" evaluation points; e.g., the points  $x_{I+1/6} = x_I + 1/6h_I$ ,  $x_{I+1/3} = x_I + 1/3h_I$  on which intermediate approximations are evaluated for estimating the error (see §2 of [1]).

In our code, algorithm 2B is divided into 2C, which does loops (a) and (c), and 2D, which does the rest. The total number of B-splines computed is  $\frac{1}{2}(k+m_d)(k+m_d-1)$ , while that computed in 2D is  $m_d(m_d-1)$ . Thus, if we ignore the one call to 2C per N "regular" evaluation points, the savings are about 50%, and

more if k>m.

Note that we have not utilized the symmetry of the collocation points (because it enters only in 2C and is not worth the complication in the program) and the nonconvex modification suggested in [4]. Even though we have not yet seen a case in which this nonconvex modification has significantly affected the accuracy in the solution, its improvement in efficiency is small enough that we have decided to play it safe.

## 3. Computation of derivatives of the approximate solution.

The n-th component of the approximate solution,  $v_n(x)$ , is given in terms of the coefficients  $\alpha_{j,n}$  by (1.4) or (1.5). Its r-th derivative is given by  $(x \in [t_i, t_{i+1}))$ 

(3.1) 
$$v_n^{(r)}(x) = (k+m_n-1)\dots(k+m_n-r) \sum_{j=-k-m_n+r+1}^{0} \alpha_{i+j,n}^{(r)} N_{i+j,k+m_n-r}^{(x)}(x)$$

where

(3.2) 
$$\alpha_{i+j,n}^{(r)} = \begin{cases} \alpha_{i+j,n} & r = 0 \\ \\ \frac{\alpha_{i+j,n}(r-1) - \alpha_{i+j-1,n}}{t_{i+j+k+m_n-r} - t_{i+j}} & r > 0 \end{cases}$$

In [3] a routine is given which prepares a table of divided differences, according to (3.2) with  $\alpha_{i+j,n}^{(r)}$  multiplied by  $(k+m_n-1)\dots(k+m_n-r)$ . We have written a similar routine which again uses the fact that  $\frac{1}{t_{i+j+k+m_n}-r-t_{i+j}}$  is known:

(3.3) 
$$t_{i+j+k+m_n-r} - t_{i+j} = \begin{cases} x_2 - x_1 & r+1 \le i+j \le m_n \\ x_{I+1} - x_I & (I-1)k+m_n+1 \le i+j \le Ik+r \\ x_{I+2} - x_I & Ik+r+1 \le i+j \le Ik+m_n \\ I=1, \dots, N \end{cases}$$

To compute  $z(\underline{v}) \equiv (v_1, v_1', \dots, v_1^{(m_1-1)}, v_2, \dots, v_d, \dots, v_d^{(m_d-1)})$  we need derivatives up to order  $m_n-1$ . Thus the resulting algorithm is

Algorithm 3A (with 
$$\alpha_{i+j,n}^{(0)} \equiv \alpha_{i+j,n}$$
):

Do for n = 1, ..., d:  
Do for r = 1, ..., m\_n^{-1:  

$$\rho_1 = (k+m_n-r)/(x_2-x_1)$$
  
Do for  $\ell = r+1, ..., m_n$ :  
 $(*) \alpha_{\ell,n}^{(r)} = (\alpha_{\ell,n}^{(r-1)} - \alpha_{\ell-1,n}^{(r-1)}) \times \rho_1$   
Do for I = 1, ..., N:  
 $\rho_1 = (k+m_n-r)/(x_{I+1}-x_I)$   
 $\rho_2 = (k+m_n-r)/(x_{I+2}-x_I)$   
Do for  $\ell = (I-1)k+m_n+1, ..., Ik+r:$   
 $(*) with \rho_1$   
Do for  $\ell = Ik+r+1, ..., Ik+m_n:$   
 $(*) with \rho_2$ 

There are many occasions in COLSYS where  $\underline{z}(\underline{v})$  has to be evaluated. The most frequent evaluations are made during the Newton iterations for a nonlinear problem, where the solution from the former iteration is evaluated to determine the coefficients of the new linearized problem. Other evaluations are made for estimating the error and producing the final approximate solution for the user.

There are two alternative ways to evaluate z(y).

### Algorithm 3B

- (a) Generate once the divided difference table for  $\alpha_{i,n}^{(r)}$ , i=1, ..., kN+m<sub>n</sub>, r = 1, ..., m<sub>n</sub>-1, n = 1, ..., d, using algorithm 3A.
- (b) For each x, x  $\varepsilon$  [t<sub>i</sub>,t<sub>i+1</sub>) form the nonzero B-splines, up to order k+m<sub>d</sub>, using algorithm 2B.
- (c) (For each x), form  $v_n^{(r)}(x) = \sum_{\substack{j=-k-m_n+r+1 \\ n}}^{0} \alpha_{i+j,n}^{(r)} N_{i+j,k+m_n-r}(x)$ r = 0, ...,  $m_n-1$ , n = 1, ..., d.

#### Algorithm 3C

(a) Generate once the piecewise-polynomial coefficients at the knots: for  $x = x_1, x_2, ..., x_N$  do algorithm 3B, with r in algorithm 3A going up to  $k+m_1-1$ .

(b) For each x, x 
$$\in [x_1, x_{1+1})$$
, form  $v_n^{(r)}(x) = \sum_{\substack{j=r \\ j=r}}^{k+m_n-1} \frac{v_n^{(j)}(x_1)}{(j-r)!} (x-x_1)^{j-r}$ .

One disadvantage of 3C as compared to 3B is obviously that the amount of storage required for the divided difference table is more than doubled. It also requires more initialization and, in fact, if the number of points x at which z(y(x)) is required is less than N, algorithm 3B must be more effective. However, when the number of points x is very large, the situation is different. In the collocation example given in [3], it is algorithm 3C which is used for the nonlinear iterations.

We shall compare the relative efficiency of the two algorithms in our setting. For this, consider two cases.

- (i) The approximation is to be evaluated at M<sub>1</sub> points x, irregularly distributed in the interval [a,b].
- (ii) The approximation is to be evaluated at  $M_1 = M_2 \times N$  regularly distributed

points x,  $M_2$  points per subinterval at the same relative positions: x -  $x_I = \rho (x_{I+1} - x_I)$  where there are only  $M_2$  different  $\rho$ 's. Consider N large, compared to  $M_2$ , k, and d.

First, algorithm 3A involves 2N divisions,  $2N+kN+m_n-r$  subtractions and  $kN+m_n-r$  multiplications for each n and r. This amounts to about (k+2)N (m\*-d) multiplications for 3B, and about (k+2)N (m\*-d) + (k+2)Nkd multiplications for 3C, where r goes up to  $k+m_n-1$ . Here,  $m* = \int_{n=1}^{d} m_n \cdot n^{n+1}$ 

For step (b) in algorithm 3B we have about  $M_1(k+m_d)(k+m_d-1)$  operations for case (i), and  $2M_1m_d(m_d-1)$  operations for case (ii), where the 2C part can be ignored.

Step (c) in algorithm 3B has

$$\begin{array}{cccc} d & m_{n} - 1 & d \\ M & \Sigma & \Sigma & (k+m_{n} - r) = M_{1} & \Sigma \left[ (k+\frac{1}{2})m_{1} + \frac{1}{2}m_{2}^{2} \right] = M_{1} \left[ (k+\frac{1}{2})m + \frac{1}{2}\hat{M} \right] = M_{1} \overline{M}$$

multiplications, where  $\hat{M} = \sum_{n=1}^{\infty} m^2$  and  $\overline{M} = (k+\frac{1}{2})m^{k+\frac{1}{2}}\hat{M}$ .

Thus the overall amount of work measured by the approximate number of multiplications for algorithm 3B is

(3.4) 
$$W_1 = (m^{*}-d)(k+2)N + M_1[(k+m_d)(k+m_d-1) + \overline{M}]$$

in case (i), and

(3.5) 
$$W_2 = N\{(m^*-d)(k+2)+M_2[2(m_d^2-m_d) + \overline{M}]\}$$

in case (ii).

For step (b) of algorithm 3C the number of multiplications is  $M_1[\overline{M} + 2(k+m_d-2)]$ . Thus the overall amount of work for algorithm 3C is

(3.6) 
$$\overline{W} = N\{(m^{*}-d)(k+2) + kd(k+2) + 2(m_{d}^{2}-m_{d}) + \overline{M} + \frac{d}{2}k(k+1)\} + M_{1}[\overline{M} + 2(k+m_{d}-2)].$$

We consider now when  $\overline{W}$  is cheaper than  $W_1$  or  $W_2$ . Case (i).  $\overline{W} \leq W_1$ .

$$\mathbb{N}\{kd(k+2) + 2(m_d^2 - m_d) + \overline{M} + \frac{d}{2}k(k+1)\} \le M_1[(k+m_d)(k+m_d-1) - 2(k+m_d-2)].$$

Since the right hand term is positive,  $\overline{W} \leq W_1$  only when  $M_1 \geq N\lambda$ , where

(3.7) 
$$\lambda = \frac{kd(k+2) + 2(m_d^2 - m_d) + \overline{M} + \frac{d}{2}k(k+1)}{(k+m_d)(k+m_d^{-1}) - 2(k+m_d^{-2})}$$

Note that  $\lambda$  grows with d. For d = 1, m = 2 and k = 3,  $\lambda = \frac{34}{14} \approx 2.5$ . For d = 3, m<sub>1</sub> = 1, m<sub>2</sub> = 3 = m<sub>3</sub> and k = 4,  $\lambda = \frac{155}{32} \approx 4.8$ .

Case (ii).  $\overline{W} \leq W_2$ .

For this to

$$kd(k+2) + 2(m_d^2 - m_d) + \overline{M} + \frac{d}{2}k(k+1) \le M_2[2(m_d^2 - m_d) - 2(k+m_d - 2)]$$
  
happen we need  $2(m_d^2 - m_d) - 2(k+m_d - 2) > 0$ . Let

(3.8) 
$$\mu = \frac{kd(k+2) + 2(m_d^2 - m_d) + \overline{M} + \frac{d}{2}k(k+1)}{2[m_d^2 - 2m_d - k + 2]}$$

Then algorithm 3C is faster only when  $\mu > 0$  and  $M_2 \ge \mu$ . This is never the case for  $m_d \le 2$ ! For d = 1,  $m_d = 3$  and k = 4,  $\mu = \frac{64}{2} = 32$  and  $\mu$  grows with d. For d = 1,  $m_d = 4$  and k = 5,  $\mu = \frac{104}{10} = 10.4$ .

Thus, in practice algorithm 3B is always preferable to algorithm 3C in the "regular" case. This is the case in the nonlinear Newton iterations in COLSYS, except for the first iteration where the former mesh is not the same as the current one. Based on the above considerations, our code never converts to the piecewise polynomial representation 3C. The additional storage and computational and programming overhead are not compensated in general by speed.

## 4. Derivatives of B-splines.

In order to assemble the collocation equations, an algorithm is needed to evaluate derivatives of the basis functions. The desired output is (x  $\in$  [t<sub>i</sub>,t<sub>d+1</sub>))  $N_{i-k-m_{1}+1,k+m_{1}}, \dots, N_{i,k+m_{1}}, N_{i-k-m_{2}+1,k+m_{2}}, \dots, N_{i-k-m_{d}+1,k+m_{d}}, \dots, N_{i,k+m_{d}}, \dots,$ 

Of course, the formula (3.1) for derivatives of a B-spline combination can be used here, with  $\alpha_{i+j,n}^{(0)} = \delta_{j\ell}$  for  $N_{i+\ell,k+m}$ . The resulting algorithm follows.

#### Algorithm 4A:

Do for n = 1, ..., d (each component)  
Do for 
$$\ell = -k-m_n+1$$
, ..., 0: (each B-spline)  
Do for j =  $-k-m_n+1$ , ..., 0 (initialization)  
 $\alpha_{i+j,n,\ell}^{(0)} = \delta_{j\ell}$   
Do for r = 1, ...,  $m_n$  (each derivative)  
Do for j =  $-k-m_n+r+1$ , ..., 0 (triangular array of  $\alpha$ 's)  
 $\alpha_{i+j,n,\ell}^{(r)} = \frac{\alpha_{i+j,n,\ell}^{(r-1)} - \alpha_{i+j-1,n,\ell}^{(r-1)}}{t_{i+j+k+m_n}-r - t_{i+j}}$   
D<sup>r</sup> N<sub>i+\ell,k+m<sub>n</sub></sub>(x) = 0 (initialization)  
Do for j =  $-k-m_n+r+1$ , ..., 0 (sum accumulation)  
Do for j =  $-k-m_n+r+1$ , ..., 0 (sum accumulation)  
Do for j =  $-k-m_n+r+1$ , ..., 0 (sum accumulation)  
Do for j =  $-k-m_n+r+1$ , ..., 0 (sum accumulation)  
D<sup>r</sup> N<sub>i+l,k+m<sub>n</sub></sub>(x) = D<sup>r</sup> N<sub>i+j,k+m<sub>n</sub></sub>(x) +  $\alpha_{i+j,n,\ell}^{(r)} N_{i+j,k+m_n}-r$ (x)  
D<sup>r</sup> N<sub>i+\ell,k+m<sub>n</sub></sub>(x) = (k+m-1)...(k+m\_n-r) D<sup>r</sup> N<sub>i+\ell,k+m<sub>n</sub></sub>

It is apparent that there are a number of savings that can be made to improve algorithm 4A. First, if  $m_n = m_{n+1}$ , there is no need to repeat the computations for both orders, since they are the same. Hence, in the beginning the program isolates the set of strictly increasing orders and applies the algorithm only to them. The output array is then filled by copying available data where needed. We shall assume henceforth, without restricting generality, that  $m_1 < m_2 < \ldots < m_d$ .

Second, note that most of the  $\alpha_{i+j,m,\ell}^{(r)}$  are zeros, with a very simple zero structure. By moving the loop on  $\ell$  inside the loop on j, the loop on  $\ell$  goes only from j to j+r. This has been noted and implemented in [3].

Now, again apply our knowledge of t i+j+k+m -r - t i+j:

(4.1) 
$$t_{i+j} - t_{i+j-k-m_n+r} = \begin{cases} h_{I-1} + h_I & j \le m_n - r \\ h_I & m_n - r + 1 \le j \le k \\ h_I + h_{I+1} & k+1 \le j \le k + m_n - r \end{cases}$$

where we shift the index j to run from 1 to  $k+m_n-r$  and break it into three domains, so that in each the divided difference involves multiplication by the same constant (similar to algorithm 2B).

The fourth improvement uses the fact that we have a system of differential equations. If the algorithm is performed for n = d, then a number of the  $\alpha_{i+j,n,\ell}^{(r)}$  may be determined directly by making the observation that

(4.2) 
$$\alpha_{i+j,n,\ell}^{(r)} = \begin{cases} \alpha_{i+j+(m_d-m_n),d,\ell}^{(r)} & j = 1, \dots, k - (m_d-m_n) - r + 1 \\ \alpha_{i+j,d,\ell}^{(r)} & j = m_d, \dots, k + m_n - r. \end{cases}$$

All of these points are incorporated in algorithm 4B, which is implemented in the code COLSYS.

<u>Algorithm 4B</u>: (Suppose  $\alpha_{i+j,n,\ell}^{(0)} = \delta_{j\ell}$  and  $D^r N_{j,k+m_n}(x)$ ,  $r \ge 1$ , are initialized to zero ahead of time. Assume  $x \in [t_i, t_{i+1}) = [x_I, x_{I+1})$  and omit the explicit dependence of  $N_{j,k+m_n}$  on x).

1. Set 
$$\rho_1 = \frac{1}{x_{I+1} - x_{I-1}}$$
;  $\rho_2 = \frac{1}{x_{I+1} - x_I}$ ;  $\rho_3 = \frac{1}{x_{I+2} - x_I}$ .

2. Do for r = 1, ..., m<sub>a</sub>: (each derivative)  
3. Do for j = 1, ..., m<sub>d</sub>-r: (largest order n = d)  
Do for 
$$\ell = j$$
, ..., j+r: (argest order n = d)  
 $p_{0}$  for  $\ell = j$ , ..., j+r: (largest order n = d)  
 $p_{1}$  for  $\ell = j$ , ..., j+r: (not support to the state of th

(multiply by constant)

5. Do for 
$$n = 1, ..., d$$
:  
 $c = 1$   
Do for  $r = 1, ..., m$ :  
 $c = c (k+m_n-r)$   
Do for  $\ell = 1, ..., k+m_n$ :  
 $D^r N_{i+\ell-k-m_n,k+m_n} = D^r N_{i+\ell-k-m_n,k+m_n} \times C$ 

## 5. Derivatives of highest order

When selecting a new mesh we need the values of the piecewise constant functions  $v_n^{(k+m_n-1)}(x)$  n=1,...,d. These are obtained in a straightforward manner, starting with the values  $\alpha_{i+j,n}^{(m_n-1)}$ , j=-k,...,0, previously obtained in algorithm 3A, and using (3.2), with  $t_{i+j+k+m_n-r} - t_{i+j} = h_I$ , to obtain the values  $\alpha_{i,n}^{(k+m_n-1)} = v_n^{(k+m_n-1)}(x)$  for  $x \in [t_i, t_{i+1}) = [x_I, x_{I+1})$ .

## 6. Using the B-spline algorithms in COLSYS

The use of the B-spline routines in the collocation code is summarized as follows:

- 1. Call 2C for each of the Gaussian points.
- 2. For each Newton iterate, evaluate the former solution and its derivatives using 3Å and 3B (the evaluation with 3Å is done beforehand and also serves to check convergence of the nonlinear iteration). Calculate B-splines with 2D and derivatives with 4B.
- Call 2C once for points at which the error is checked. Evaluate the solution and derivatives for error estimation on the given mesh using 2D, 3A and 3B.

- 4. Evaluate  $v_n^{(k+m_n-1)}$ , n=1,...,d, as described in section 5 to select the new mesh.
- 5. When the error tolerances are satisfied, evaluate y(x) and derivatives at user-specified points using algorithm 3B.

- U. Ascher, J. Christiansen and R. D. Russell, A collocation solver for mixed order systems of boundary value problems, Tech. Rep. #13, Computer Science, UBC, Vancouver, Canada (1977).
- 2. C. de Boor, On calculating with B-splines, J. Approx, Th., 6 (1972), 50-62.
- C. de Boor, Package for calculating with B-splines, SIAM J. Numer. Anal., <u>14</u> (1977), 441-472.
- R. D. Russell, Efficiencies of B-splines methods for solving differential equations, Proc. 5th Conf. on Numerical Math., Manitoba (1975), 599-617.