stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IN 18/80 AUGUSTUS

C.J. RUSMAN

# B-SPLINE ALGORITHMS

kruislaan 413 1098 SJ amsterdam

*Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.*

*The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).*

1980 Mathematics subject classification: 65D05, 65D07

ACM-Computing Reviews-category: 5.13

# B-Spline Algorithms

by

C.J. Rusman

ABSTRACT

B-spline algorithms, given by C. de Boor, are being made suitable for the graphical system of the Mathematical Centre, while the theory leading to the algorithms is treated in detail. An alternative algorithm for B-spline interpolation is developed, and compared with the 'de Boor' algorithms.

KEY WORDS & PHRASES: *computer graphics, B-spline approximation, interpolation*

## CONTENTS

# 1. INTRODUCTION

This is an essay on algorithms for interpolation by polynomial splines, being determined by linear combinations of B-splines.

Assuming that hardware facilities would make it possible to plot polynomials from their coefficients, the ultimate goal was set at representing the polynomial splines by the coefficients of the composing polynomials.

Chapter 6 is the outcome of this endeavour. On the other hand, while studying the theory of polynomial splines, it became apparent that when having existing algorithms available in the graphical system of the Mathematical Centre, this would enrich the system. This is the reason why the rest of the essay is concerned with making the procedures of the 'de Boor-set' (cf. [2], [4]) suitable for the system.

The theory leading to the respective algorithms is quite self-contained, in that *complete* proofs are given for almost all the theorems. The few that make the exception would lead to considerations which are of no relevance for the presented work.

Although the greater part of the procedures can be used for splines of arbitrary order, in the examples and in the procedures of Chapter 6, the order has been fixed at 4.

The procedures, as presented here, are not given in their most efficient form, but in a way that facilitates reading in relation to the text.

Some mathematical foundations are given in Appendix A, the graphical system mentioned and programming language used are briefly described in Appendix B, while the procedures which do not follow directly from the presented theory have been put together in Appendix C.

## ACKNOWLEDGEMENT

This work was carried out as a "doctoraal scriptie" while working at the Mathematical Centre. The work was supervised by Prof. Th.J. Dekker and Drs. P.J.W. ten Hagen. The new algorithm of chapter 6 is based on a method developed by P.J.W. ten Hagen.

## 2. PIECEWISE POLYNOMIALS

Assume we want to construct a curve of the form $y = y(x)$, and that we require the curve to fit the set of datapoints $(x_0,y_0),\ldots,(x_n,y_n)$ where $(x_i,y_i) \in \mathbb{R}^2$, $i = 0,1,\ldots,n$ and $x_0 < x_1 <\ldots< x_n$.

Classical methods of numerical analysis which tackle the suggested problem by polynomial fitting, have, to their disadvantage, either the tendency to oscillate (fitting with high-degree polynomials) or insufficient fit at the datapoints (least squares approximation).

A way to circumvent these disadvantages is to construct a piecewise polynomial, consisting of successive low-degree polynomials, which interpolates the given datapoints.

<u>DEFINITION 2.1</u>. Let $(x_i)_{i=0}^{n}$ be a strictly increasing sequence, $x_i \in \mathbb{R}$, and let $k \in \mathbb{N}$. Let $p_0,p_1,\ldots,p_{n-1}$ be a sequence of polynomials of order $k$ (degree $k-1$), then the *piecewise polynomial* $f$, denoted by *pp*, is defined by:

$$f(x) := p_i(x) \quad \text{if } x_i < x < x_{i+1},\ i = 0,1,\ldots,n-1.$$

At the interior points $x_i$, which we, together with $x_0$ and $x_n$, will call the *breakpoints*, the pp $f$ now may be considered to have two values.

$$f(x_i^-) = p_{i-1}(x_i) \quad \text{and} \quad f(x_i^+) = p_i(x_i).$$

In order to obtain a single-valued function, we choose to define $f$ to be continuous from the right:

$$f(x_i) := p_i(x_i), \quad i = 1,2,\ldots,n-1,$$

and extend the domain of $f$ by $x_0$ and $x_n$:

$$f(x_0) := p_0(x_0)$$
$$f(x_n) := p_{n-1}(x_n)$$

Looking for a useful representation for a pp, that interpolates the given datapoints $(x_i,y_i)_{i=0}^n$, we consider the subinterval $[x_i,x_{i+1})$, $0 \le i \le n-1$. Let $p_i(x) = a_0 + a_1.x + \ldots + a_{k-1}.x^{k-1}$, then $p_i', p_i'', \ldots, p_i^{(k-2)}$ exist on $[x_i,x_{i+1})$, and the $k-1^{th}$ derivative is continuous and differentiable on $[x_i,x_{i+1})$. Then for every $x \in [x_i,x_{i+1})$ we have (Taylor)

$$p_i(x) = p_i(x_i) + (x-x_i).p_i'(x_i) + \ldots + \frac{(x-x_i)^k}{k!} . p_i^{(k)}(\xi)$$

where $\xi \in (x_i,x)$. But $p_i^{(k)}(x) \equiv 0$, so

$$\begin{aligned} p_i(x) &= p_i(x_i) + \ldots + \frac{(x-x_i)^{k-1}}{(k-1)!} . p_i^{(k-1)}(x_i) \\ &= c_{0,i} + c_{1,i}(x-x_i) + \ldots + c_{k-1,i} . \frac{(x-x_i)^{k-1}}{(k-1)!} \end{aligned}$$

where $c_{j,i} = p_i^{(j)}(x_i)$, $j = 0,1,\ldots,k-1$, $i = 0,1,\ldots,n-1$.

We give the representation of a pp f in terms of these derivatives $c_{j,i}$, which is particularly handy when f and some of its derivatives are to be evaluated at a number of points:

DEFINITION 2.2. The pp-representation for a pp f consists of

(i) order k

(ii) number of polynomial pieces n

(iii) the strictly increasing sequence $x_0,x_1,\ldots,x_n$ of its breakpoints

(iv) the matrix $(c_{j,i})_{j=0}^{k-1}{}_{i=0}^{n-1}$ of its *right* derivatives in the breakpoints $x_0,\ldots,x_{n-1}$.

From this definition it follows that

(1) $$f(x) = \sum_{i=0}^{n-1} \sum_{j=0}^{k-1} c_{j,i} . \frac{(x-x_i)^j}{j!}, \qquad x \in [x_0,x_n]$$

and

(2) $$f^{(m)}(x) = \sum_{j=m}^{k-1} c_{j,i} . \frac{(x-x_i)^{j-m}}{(j-m)!}, \qquad \begin{array}{l} x \in [x_i,x_{i+1}),\ 0 \le i \le n-2 \\ x \in [x_{n-1},x_n],\ \text{if } i = n-1. \end{array}$$

For the evaluation of the $m^{th}$ derivative of the pp f at the point $x \in [x_i,x_{i+1})$, we see from (2) that

$$f^{(m)}(x) = c_{m,i} + \frac{\Delta}{1}(c_{m+1,i} + \frac{\Delta}{2}(c_{m+2,i} + \frac{\Delta}{3}(\ldots(\frac{\Delta}{k-m-1} \cdot c_{k-1,i}))\ldots)$$

where $\Delta = x - x_i$, which is used in the following procedure:

```
double
ppval(breaks,order,numpol,pcoef,x,der)
        double breaks[], pcoef[][MAXORDER], x;
           int order, numpol, der;
        /* This procedure calculates the value of the 'der'th
         * derivative in the point x, of the piecewise polynomial
         * given in (breaks, order, numpol, pcoef).
         * The procedure uses  the procedure 'interval', that
         * gives in 'left' the number i such that
         * x[i] <= x < x[i+1].
         */
{       double val, delta;
           int n, left, f, l;

        val = 0;
        if (der < order)        /* else the procedure yields 0 */
           {
                interval(breaks,numpol,&left,x,&f,&lastleft);
                delta = x - breaks[left];
                for ( n = order-1; n >= der; n--)

                val = (val*delta)/(n+1-der) + pcoef[left][n];
           }
        return(val);
}
```

# 3. SPLINE FUNCTIONS

## 3.1. Introduction

The interpolation as suggested in the preceding section will generally result in a curve that is not smooth at the breakpoints. Since this is an undesirable feature, we seek a composite function $\phi$, which has the following properties:

(i) over each subinterval $[x_i,x_{i+1})$, $i = 0,1,\dots,n-2$, and over $[x_{n-1},x_n]$, $\phi$ is a polynomial,

(ii) at the breakpoints $x_1,\dots,x_{n-1}$, the function $\phi$ is as smooth as possible.

The smoothest pp on the breakpoints $x_0,\dots,x_n$, is the pp that has as many continuous derivatives at the interior breakpoints as possible. This leads to the definition of a spline function:

DEFINITION 3.1.1. Let $x_0,x_1,\dots,x_n$ be a strictly increasing sequence of points. Let $\phi$ be a function on $[x_0,x_n]$, which has the following properties:

(i) $\phi$ is a polynomial of order k on each subinterval $[x_i,x_{i+1})$, $i = 0,\dots,n-2$ and on $[x_{n-1},x_n]$

(ii) $\phi$ and its first k-2 derivatives are continuous at the points $x_1,\dots,x_{n-1}$.

Then $\phi$ is called a *spline function* of *order* k on the *knots* $x_0,x_1,\dots,x_n$, and the subintervals are called the *spans*.

We see that a spline function is a piecewise polynomial which is as smooth as can be, without simply reducing to a polynomial!

The desired interpolating function then is the function $\phi$ which has the properties i) and ii) of definition 3.1.1, and for which

$$\phi(x_i) = y_i, \quad i = 0,1,\dots,n.$$

We call this function the spline function with the knots $x_0,\dots,x_n$, which interpolates the datapoints $(x_0,y_0),\dots,(x_n,y_n)$. The interval $[x_0,x_n]$ is called the *support of the spline*.

If the order is 4, we call the spline function a *cubic spline*.

For interpolation problems, splines of *even* degree are not often used, since they have certain characteristics that make them less suitable ([1], ch.3).

## 3.2. Determination of a spline function

Consider the cubic spline function $\phi$ (order 4), with knots $x_0, x_1, \ldots, x_n$ and datapoints $(x_0,y_0),\ldots,(x_n,y_n)$. Since $\phi$ is cubic, $\phi''$ is linear in x. Consider the span $[x_{j-1},x_j]$, $1 \le j \le n$, and let $h_j = x_j - x_{j-1}$.
We have

(3.2.1) $$\phi''(x) = \phi''(x_j) . \frac{x-x_{j-1}}{h_j} + \phi''(x_{j-1}) . \frac{x_j-x}{h_j}$$

Two integrations then result in

(3.2.2) $$\phi(x) = \frac{\phi''(x_j)}{6.h_j} . (x-x_{j-1})^3 + \frac{\phi''(x_{j-1})}{6h_j} . (x_j-x)^3 + c_1.x + c_2$$

and we will try to determine $c_1, c_2$ and $\phi''(x_0),\ldots,\phi''(x_n)$. Take $x = x_j$:

(3.2.3) $$\phi(x_j) = \frac{\phi''(x_j).h_j^3}{6.h_j} + c_1.x_j + c_2$$

and $x = x_{j-1}$:

(3.2.4) $$\phi(x_{j-1}) = \frac{\phi''(x_{j-1}).h_j^3}{6h_j} + c_1.x_{j-1} + c_2.$$

Subtracting (3.2.4) from (3.2.3) gives

(3.2.5) $$\phi(x_j)-\phi(x_{j-1}) = (\phi''(x_j) - \phi''(x_{j-1})) . \frac{h_j^2}{6} + c_1.h_j.$$

This gives

(3.2.6) $$c_1 = \frac{\phi(x_j)-\phi(x_{j-1})}{h_j} - (\phi''(x_j) - \phi''(x_{j-1})) . \frac{h_j}{6}$$

and $c_2$ can now be evaluated from (3.2.3) and (3.2.6):

(3.2.6) $$c_2 = \frac{\phi''(x_j)-\phi''(x_{j-1})}{6} . h_j.$$

On the span $[x_{j-1},x_j]$ we get, after once integrating (3.2.1) and using (3.2.6):

$$\phi'(x) = \frac{\phi''(x_j)}{2.h_j}.(x-x_{j-1})^2 - \frac{\phi''(x_{j-1})}{2.h_j}.(x_j-x)^2 + \frac{\phi(x_j)-\phi(x_{j-1})}{h_j} - \frac{\phi''(x_j)-\phi''(x_{j-1})}{6}.h_j.$$

Take $x = x_j$:

$$\text{(3.2.7)} \qquad \phi'(x_j) = \frac{h_j}{6}.\phi''(x_{j-1}) + \frac{h_j}{3}.\phi''(x_j) + \frac{\phi(x_j)-\phi(x_{j-1})}{h_{j+1}}$$

Now on the span $[x_j,x_{j+1}]$:

$$\phi'(x) = \frac{\phi''(x_{j+1}).(x-x_j)^2}{2h_{j+1}} - \frac{\phi''(x_j)}{2h_{j+1}}.(x_{j+1}-x)^2 + \frac{\phi(x_{j+1})-\phi(x_j)}{h_{j+1}} - \frac{\phi''(x_{j+1})-\phi''(x_j)}{6}.h_{j+1}.$$

Take $x = x_j$:

$$\text{(3.2.8)} \qquad \phi'(x_j) = -\frac{h_{j+1}}{3}.\phi''(x_j) - \phi''(x_{j+1}).\frac{h_{j+1}}{6} + \frac{\phi(x_{j+1})-\phi(x_j)}{h_{j+1}}.$$

Since we have continuity of the first derivative at the knots $x_1,\dots,x_{n-1}$, we find by equalizing (3.2.7) and (3.2.8) for $j = 1,2,\dots,n-1$, exactly $(n-1)$ equations:

$$\text{(3.2.9)} \qquad \frac{h_j}{6}.\phi''(x_{j-1}) + \frac{h_j}{3}.\phi''(x_j) + \frac{h_{j+1}}{3}.\phi''(x_j) + \frac{h_{j+1}}{6}.\phi''(x_{j+1}) = -\frac{\phi(x_j)-\phi(x_{j-1})}{h_j} + \frac{\phi(x_{j+1})-\phi(x_j)}{h_{j+1}}.$$

We were trying to calculate the n+1 unknowns $\phi''(x_0),\dots,\phi''(x_n)$, so we need precisely two additional relations to enable us to determine the spline function $\phi$ unambiguously via (3.2.2) and (3.2.6).

The cubic spline can thus be constructed by using the n+1 function values in the knots, and just *two* other items of information, making (n+3) items of data in all.

The additional relations will, depending on any specific application, be chosen from one of the following possibilities:

(i) free ends : let the spline function be straight outside the interval $[x_0,x_n]$: $\phi''(x_0) = \phi''(x_n) = 0$. The resulting spline is called a *natural spline*

(ii) fixed ends : let the slope of the spline function be given at the endpoints: $\phi'(x_0) = a$, $\phi'(x_n) = b$. The resulting spline is called a *complete spline*

(iii) 'not a knot': let the first and second polynomial pieces coincide on $[x_0,x_2)$, and the last and one but last polynomial pieces coincide on $[x_{n-2},x_n]$. This means that $x_1$ and $x_{n-1}$ are not considered as knots, and we have n-3 polynomial pieces rather than n-1, together with two interpolation points which are not breakpoints! The additional relations are generated by taking $\phi'''$ continuous across the knots $x_1$ and $x_{n-1}$.

## 3.3. Example

Let $x_0,x_1,\ldots,x_n$ be a strictly increasing sequence of *equidistant* points, and let $\phi''(x_0) = \phi''(x_n) = 0$. Then equation (3.2.9) yields

$$\phi''(x_{j-1})+4.\phi''(x_j)+\phi''(x_{j+1}) = \frac{6}{h}\cdot\left(\frac{\phi(x_j)-\phi(x_{j-1})}{-h} + \frac{\phi(x_{j+1})-\phi(x_j)}{h}\right)$$

$$= A_j, \quad j = 1,2,\ldots,n-1.$$

In matrix form:

$$\begin{pmatrix} 4 & 1 & & & & & \\ 1 & 4 & 1 & & & & \\ & 1 & 4 & 1 & & & \\ & & 1 & 4 & 1 & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & 1 & 4 & 1 \\ & & & & & 1 & 4 \end{pmatrix} \begin{pmatrix} \phi''(x_1) \\ \vdots \\ \vdots \\ \phi''(x_{n-1}) \end{pmatrix} = \begin{pmatrix} A_1 \\ \vdots \\ \vdots \\ A_{n-1} \end{pmatrix}$$

## 4. B-SPLINES

### 4.1. Introduction

Consider a cubic spline $\phi$ with $\phi(x) = \phi'(x) = \phi''(x) = 0$ at both the endpoints of the range defined by the knots. Then those 6 relations enable us to compute a spline with 6-2 = 4 knots. This spline turns out to be $\phi(x) \equiv 0$.

If we take 5 knots $x_i, x_{i+1}, x_{i+2}, x_{i+3}, x_{i+4}$, we just need *one* additional piece of information. To ensure that the spline is not identically zero, we specify a function value at an internal knot.

By extending this spline from its endpoints by straight lines along the axis, we get a cubic spline over an indefinite number of spans, which is non-zero only on 4 adjacent spans. This function will be called a *B-spline* (Basic, Bell shaped, fundamental) *of order* 4, and is denoted by $M_{4,i}$ (see fig. 4.1.1)

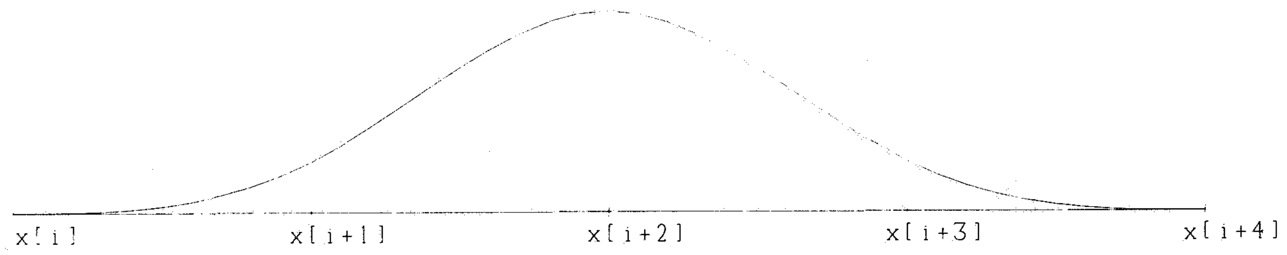


fig. 4.1.1

The B-spline is determined by the knot-set on which it is defined, and one function value $\neq 0$. The support of this spline is minimal: if we take away one knot, the B-spline degenerates into the function which is identically zero *everywhere*.

### 4.2. Definition of a B-spline

We will formally define a B-spline in terms of divided differences, and need one preliminary definition:

DEFINITION 4.2.1. The *truncated power function* $t_+^k$ is defined as:

$$t_+^k = \begin{cases} t^k & \text{if} \quad t \geq 0 \\ 0 & \text{if} \quad t < 0 \end{cases}$$

where $k \in \mathbb{N}^+$ and $t \in \mathbb{R}$. Now consider the function $f(x;z) = (z-x)_+^3$, of which we form the 4th divided difference with respect to the strictly increasing sequence $x_i, x_{i+1}, x_{i+2}, x_{i+3}, x_{i+4}$.

By using theorem A.1 we get

$$(4.2.1) \qquad \phi(x) := f[x;x_i,x_{i+1},x_{i+2},x_{i+3},x_{i+4}] = \sum_{k=0}^{4} \frac{(x_{i+k}-x)_+^3}{\omega'(x_{i+k})}$$

where

$$\omega(x) = \prod_{k=0}^{4} (x-x_{i+k}).$$

When we study this relation, we find that because of definition 4.2.1:

$$\phi(x) \equiv 0 \quad \text{if } x \geq x_{i+4}$$

and that because of theorem A3:

$$\phi(x) \equiv 0 \quad \text{if } x \leq x_i.$$

Each of the functions $(x_{i+k}-x)_+^3$ is continuous in the point $x_{i+k}$, $k = 0,1,2,3,4$, up to its second derivative, so $\phi$ is a cubic spline with knots $x_i$, $x_{i+1}, x_{i+2}, x_{i+3}, x_{i+4}$, which is identically zero for $x \leq x_i$ and $x \geq x_{i+4}$.

From the foregoing in section 4.1, we recognize $\phi$ as the cubic B-spline $M_{4,i}$. This leads to the definition of a general B-spline:

DEFINITION 4.2.2. Let $x_i, x_{i+1}, \ldots x_{i+n}$ be a strictly increasing sequence of points, and let

$$f_n(x;z) := (z-x)_+^{n-1}$$

where $n \in \mathbb{N}^+$.

The *B-spline* $M_{n,i}$ of order n is given by the $n^{th}$ divided difference of $f_n(x;z)$ with respect to $x_i, x_{i+1}, \dots, x_{i+n}$ for fixed x, i.e.

$$M_{n,i}(x) = f_n[x;x_i,x_{i+1},\dots,x_{i+n}]$$
$$= \sum_{j=i}^{i+n} \frac{(x_j-x)_+^{n-1}}{\omega'(x_j)}$$

where

$$\omega(x) = \prod_{k=0}^{n} (x-x_{i+k})$$

## 4.3. Normalization of a B-spline

By establishing definition 4.2.2, we have fixed the non-zero function value at some internal knot, which helped determine the B-spline as introduced in 4.1, in such a way that the property of the following theorem holds:

THEOREM 4.3.1.

$$\int_{-\infty}^{+\infty} M_{n,i}(x)dx = \frac{1}{n} .$$

For the proof of the theorem we need the following (cf. [10]): Consider functions of the class $C^{n+1}_{[a,b]}$, $a,b \in \mathbb{R}$, and let linear functionals of the following type be defined over this class:

$$\theta(f) = \int_a^b [a_0(x)f(x) + a_1(x).f'(x)+\dots+a_n(x).f^{(n)}(x)].dx$$
$$+ \sum_{i=1}^{j_0} b_{i0}.f(x_{i0}) + \sum_{i=1}^{j_1} b_{i1}f'(x_{i1})+\dots+ \sum_{i=1}^{j_n} b_{in}.f^{(n)}(x_{in})$$

where the $a_i(x)$ are assumed to be piecewise continuous over [a,b], and the points $x_{ij}$ lie in [a,b].

LEMMA 4.3.2. (Peano) *Let* $\theta(p) \equiv 0$ *for all polynomials* p *of degree* $\leq n-1$. *Then for all* $f \in C^n_{[a,b]}$:

$$\theta(f) = \int_a^b f^{(n)}(t).K(t).dt$$

*where*

$$K(t) = \frac{1}{(n-1)!}\,\theta_x[(x-t)_+^{n-1}].$$

The notation $\theta_x$ means that the functional $\theta$ is applied to $(x-t)_+^{n-1}$, *considered as a function of* $x$. $K(t)$ is called the *Peano kernel* of $\theta$.

PROOF. Recall Taylor's theorem:

$$f(x) = f(a) + f'(a).(x-a)+\ldots+f^{(n-1)}(a).\,\frac{(x-a)^{n-1}}{(n-1)!}$$
$$+ \frac{1}{(n-1)!}\int_a^x f^{(n)}(t).(x-t)^{n-1}dt$$

Write the last term as

$$\frac{1}{(n-1)!}\int_a^b f^{(n)}(t).(x-t)_+^{n-1}dt,$$

and apply $\theta$ to both sides of the expression: because $\theta$ annihilates all the polynomials on the right hand side, we get:

$$\begin{aligned}\theta(f) &= \frac{1}{(n-1)!}\,\theta\Big(\int_a^b f^{(n)}(t).(x-t)_+^{n-1}dt\Big)\\ &= \frac{1}{(n-1)!}\int_a^b f^{(n)}(t).\theta_x[(x-t)_+^{n-1}]dt\\ &= \int_a^b f^{(n)}(t).K(t).dt. \qquad \square\end{aligned}$$

LEMMA 4.3.3. *Let* $x_i, x_{i+1}, \ldots, x_{i+n}$ *be a strictly increasing sequence of points, and let* $f(x) = x^n$. *Then*

$$f[x_i, x_{i+1}, \ldots, x_{i+n}] = 1, \qquad \forall n \in \mathbb{N}.$$

PROOF. Induction on n:

$$n = 1:\ f(x) = x;$$

$$f[x_i,x_{i+1}] = \frac{x_i-x_{i+1}}{x_i-x_{i+1}} = 1.$$

Assume for $k < n$ and $f(x) = x^k$, that

$$f[x_i,x_{i+1},\dots,x_{i+k}] = 1.$$

Put $f(x) = x^n = x^{n-1}.x = g(x).h(x)$. With theorem A.4 we then have:

$$f[x_i,\dots,x_{i+n}] = \sum_{j=i}^{i+n} g[x_i,\dots,x_j].h[x_j,\dots,x_{i+n}]$$

$$= g[x_i,\dots,x_{i+n-1}].h[x_{i+n-1},x_{i+n}]$$

$$+ g[x_i,\dots,x_{i+n}].h(x_{i+n})$$

$$= 1.1 + 0.h(x_{i+n}) = 1.$$

The other terms are zero because of $h(x) = x$ and theorem A.3. □

<u>Proof of theorem 4.3.1.</u> (cf. [10]) Let $\theta_x$ be the functional defined by $\theta_x f = f[x_i,x_{i+1},\dots,x_{i+n}]$, where $x_i,x_{i+1},\dots,x_{i+n}$ is a strictly increasing sequence of points, and $f \in C^n_{[-\infty,\infty]}$: $\theta_x$ takes the nth divided difference of f with respect to $x_i,x_{i+1},\dots,x_{i+n}$ as a function of x. According to theorem A.3, the functional $\theta_x$ annihilates all the polynomials of degree $\le n-1$, and thus $\theta_x$ fulfills the hypothesis of lemma 4.3.2, so:

$$(4.3.1) \qquad \theta_x f = f[x_i,x_{i+1},\dots,x_{i+n}] = \int_{-\infty}^{+\infty} f^{(n)}(t)\cdot K(t)\cdot d(t)$$

where

$$(4.3.2) \qquad K(t) = \frac{1}{(n-1)!}\,\theta_x.[(x-t)_+^{n-1}].$$

For $f(x) = \frac{x^n}{n!}$ we get with lemma 4.3.3 and (4.3.1)

$$\frac{1}{n!} = \int_{-\infty}^{\infty} K(t).dt.$$

Now:

$$\frac{1}{n} = \int_{-\infty}^{\infty} (n-1)!.K(x).dx$$

and this gives with (4.3.2):

(4.3.3) $$\frac{1}{n} = \int_{-\infty}^{\infty} \theta_t[(t-x)_+^{n-1}]dx.$$

If we put $f_n(x;t) = (t-x)_+^{n-1}$, then $f_n \in C^n_{[-\infty,\infty]}$, and then (4.3.1) gives

$$\theta_t(f_n) = f_n[x;x_i,\dots,x_{i+n}]$$

Now (4.3.3) gives

$$\frac{1}{n} = \int_{-\infty}^{\infty} \theta_t[(t-x)_+^{n-1}]dx$$

$$= \int_{-\infty}^{\infty} \theta_t(f_n)dx$$

$$= \int_{-\infty}^{\infty} f_n[x;x_i,x_{i+1},\dots,x_{i+n}]dx = \int_{-\infty}^{\infty} M_{n,i}(x)dx. \qquad \square$$

<u>EXAMPLE</u>: (cf. [8]). Let

$$x_1 = x_0 + h,$$
$$x_2 = x_0 + 2h,$$
$$x_3 = x_0 + 3h,$$

and set

$$\theta(f) = -f(x_0) + 3.f(x_1) - 3.f(x_2) + f(x_3)$$

$\theta$ annihilates all the polynomials of degree 2, so following lemma 4.3.2 we have

$$K(t) = \frac{1}{2!} \cdot \theta_x[(x-t)_+^2].$$

This gives

$$\begin{aligned} 2.K(t) &= (x_3-t)^3-3(x_2-t)^2+3\ (x_1-t)^2 = (t-x_0)^2 && \text{if } x_0 \le t \le x_1 \\ &= (x_3-t)^2-(x_2-t)^2 && \text{if } x_1 \le t \le x_2 \\ &= (x_3-t)^2 && \text{if } x_2 \le t \le x_3. \end{aligned}$$

The Peano kernel K(t) has the following form for $x_0 = 0$, $h = 1$:

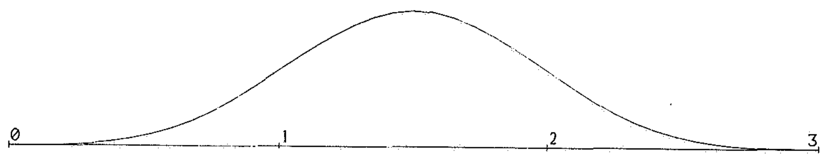


fig. 4.3.1

## 4.4. A recurrence relation for calculating B-splines

In order to calculate a B-spline, we can use the straightforward ways of definition 4.2.2:

- either form the divided difference table and compute $f_n[x,x_i,\ldots,x_{i+n}]$
- or compute the sum $\sum_{j=1}^{i+n} \frac{(x_j-x)_+^{n-1}}{\omega'(x_j)}$ .

Both ways have to their disadvantage that they include subtractions that make those methods numerically unstable.

A better method makes use of a recurrence relation by DE BOOR ([2],[3]) and COX [5]:

THEOREM 4.4.1. *Let* $x_i,x_{i+1},\ldots,x_{i+n}$ *be a non-decreasing sequence. Then*

(4.4.1) $$M_{n,i}(x) = \frac{(x-x_i)}{(x_{i+n}-x_i)} \cdot M_{n-1,i}(x) + \frac{(x_{i+n}-x)}{(x_{i+n}-x_i)} \cdot M_{n-1,i+1}(x)$$

*for all* $x \in \mathbb{R}$.

PROOF. Put $f_n(x;z) = (z-x)_+^{n-1}$. According to theorem A.1 we have

$$M_{n,i}(x) = f_n[x;x_i,x_{i+1},\ldots,x_{i+n}]$$

$$(4.4.2) \qquad = \sum_{j=i}^{i+n} \frac{f(x_j)}{\omega'_{i,n}(x_j)} .$$

where $\omega_{i,n}(x) = (x-x_i)(x-x_{i+1})\ldots(x-x_{i+n})$ and

$$\omega'_{i,n}(x_k) = \prod_{\substack{j=i\\ j\neq k}}^{i+n} (x_k-x_j)$$

$$= (x_k-x_i) . \prod_{\substack{j=i+1\\ j\neq k}}^{i+n} (x_k-x_j)$$

$$= (x_k-x_{i+n}) . \prod_{\substack{j=i\\ j\neq k}}^{i+n-1} (x_k-x_j)$$

We then have:

$$(4.4.3) \qquad \omega'_{i,n}(x_k) = (x_k-x_i).\omega'_{i+1,n-1}(x_k) = (x_k-x_{i+n}).\omega'_{i,n-1}(x_k).$$

Taking the right hand side of (4.4.1), we get with (4.4.2):

$$\frac{x-x_i}{x_{i+n}-x_i} . \sum_{j=i}^{i+n-1} \frac{(x_j-x)_+^{n-2}}{\omega'_{i,n-1}(x_j)} + \frac{x_{i+n}-x}{x_{i+n}-x_i} . \sum_{j=i+1}^{i+n} \frac{(x_j-x)_+^{n-2}}{\omega'_{i+1,n-1}(x_j)} =$$

$$\frac{x-x_i}{x_{i+n}-x_i} . \left( \frac{(x_i-x)_+^{n-2}}{\omega'_{i,n-1}(x_i)} + \sum_{j=i+1}^{i+n-1} \frac{(x_j-x)_+^{n-2}}{\omega'_{i,n-1}(x_j)} \right)$$

$$+ \frac{x_{i+n}-x}{x_{i+n}-x_i} . \left( \frac{(x_{i+n}-x)_+^{n-2}}{\omega'_{i+1,n-1}(x_{i+n})} + \sum_{j=i+1}^{i+n-1} \frac{(x_j-x)_+^{n-2}}{\omega'_{i+1,n-1}(x_j)} \right)$$

Using (4.4.3) this gives:

$$(4.4.4) \qquad \frac{(x_i-x)_+^{n-1}}{\omega'_{i,n}(x_i)} + \sum_{j=i+1}^{i+n-1} \frac{(x_j-x)_+^{n-2}}{(x_{i+n}-x_i)} . \left( \frac{x-x_i}{\omega'_{i,n-1}(x_j)} + \frac{x_{i+n}-x}{\omega'_{i+1,n-1}(x_j)} \right) +$$

$$+ \frac{(x_{i+n}-x)_+^{n-1}}{\omega'_{i,n}(x_j)} .$$

The $j^{th}$ term of the middle term of 4.4.4 becomes, using 4.4.3:

$$\frac{(x_j-x)_+^{n-2}}{x_{i+n}-x_i} \cdot \left(\frac{x-x_i}{\frac{\omega'_{i,n}(x_j)}{x_j-x_{i+n}}} + \frac{x_{i+n}-x}{\frac{\omega'_{i,n}(x_j)}{x_j-x_i}}\right) =$$

$$\frac{(x_j-x)_+^{n-2}}{x_{i+n}-x_i} \cdot \left(\frac{(x_j-x_{i+n}).(x-x_i)+(x_j-x_i)(x_{i+n}-x)}{\omega'_{i,n}(x_j)}\right) =$$

$$\frac{(x_j-x)_+^{n-2}}{x_{i+n}-x_i} \cdot \left(\frac{(x_{i+n}-x_i).(x_j-x)}{\omega'_{i,n}(x_j)}\right) = \frac{(x_j-x)_+^{n-1}}{\omega'_{i,n}(x_j)}$$

(4.4.4) then gives:

$$(4.4.5) \qquad \sum_{j=i}^{i+n} \frac{(x_j-x)_+^{n-1}}{\omega'_{i,n}(x_j)} .$$

Recalling definition 4.2.2 of the B-spline, we see that (4.4.5) is the B-spline $M_{n,i}(x)$. □

## 4.5. Calculating B-splines

We note that the recurrence relation (4.4.1) does not require the abscis x to be in any specific interval!

Repeated use of the relation generates a table that can be used to compute a B-spline. For instance, $M_{4,i}$ be computed from the table

$$\begin{array}{llll}
M_{1,i} & & & \\
 & M_{2,i} & & \\
M_{1,i+1} & & M_{3,i} & \\
 & M_{2,i+1} & & M_{4,i} \\
M_{1,i+2} & & M_{3,i+1} & \\
 & M_{2,i+2} & & \\
M_{1,i+3} & & & 
\end{array}$$

The table shows the need to have disposal of the first order B-splines, which can be found by means of the following theorem:

<u>THEOREM 4.5.1.</u>

$$M_{1,i}(x) = \begin{cases} (x_{i+1}-x_i)^{-1} & \text{if } x_i \leq x < x_{i+1} \\ 0 & \text{otherwise.} \end{cases}$$

<u>PROOF</u>:

$$M_{1,i}(x) = \sum_{j=i}^{i+1} \frac{(x_j-x)_+^0}{\omega'_{i,1}(x_j)}$$

$$= \frac{(x_i-x)_+^0}{x_i-x_{i+1}} + \frac{(x_{i+1}-x)_+^0}{x_{i+1}-x_i} . \qquad \square$$

Now if we take n to be fixed in e.g. the interval $(x_{i+1},x_{i+2})$ the table degenerates in:

$$\begin{array}{ccccc} & M_{2,i} & & & \\ M_{1,i+1} & & M_{3,i} & & \\ & M_{2,i+1} & & M_{4,i} & \\ & & M_{3,i+1} & & \end{array}$$

since all the remaining elements in the original table are zero by virtue of theorem 4.5.1. The actual calculation is numerically stable, because all the numbers $M_{k,i}$ are positive, and the relation (4.4.1) only performs additions:

<u>THEOREM 4.5.2.</u>

$$M_{k,i}(x) = \begin{cases} \text{positive} & \text{if } x_i < x < x_{i+k} \\ 0 & \text{otherwise.} \end{cases}$$

<u>PROOF</u>. By induction on k: k = 1: theorem 4.5.1. Assume the theorem holds for k = n: then, using 4.4.1:

$$M_{n+1,i} = \frac{x-x_i}{x_{i+n-1}-x_i} .M_{n,i} + \frac{x_{i+n+1}-x}{x_{i+n+1}-x} .M_{n,i+1}.$$

Since $M_{n,i}$ and $M_{n,i+1}$ are positive, it follows that $M_{n+1,i}$ is positive. □

The recurrence relation also leads to an algorithm for the *simultaneous* generation of the values at x of *all* the B-splines of a certain order, which are possibly not zero there. The matching table is:

$$\begin{array}{cccccc}
 & & & & & M_{n,i-n+1} \\
 & & & & & M_{n,i-n+2} \\
 & & & M_{4,i-3} & \cdots\cdots & \\
 & & M_{3,i-2} & & & M_{n,i-n+2} \\
 & M_{2,i-1} & & M_{4,i-2} & & \vdots \\
M_{1,i} & & M_{3,i-1} & & & \vdots \\
 & M_{2,i} & & M_{4,i-1} & & \vdots \\
 & & M_{3,i} & & & M_{n,i-2} \\
 & & & M_{4,i} & \cdots\cdots & \\
 & & & & & M_{n,i-1} \\
 & & & & & M_{n,i}
\end{array}$$

If all the entries in the last column are to be calculated, then the table presupposes the existence of 2.n knots. More precisely: If a sequence of knots $t_0, t_1, \ldots, t_s$ is available, and $M_{n,i-n+1}, \ldots, M_{n,i}$ are to be calculated for $x \in (t_i, t_{i+1})$, then $i-n+1 \geq 0$ and $i+n \leq s$. Now suppose we have calculated the j-1th column, $2 \leq j \leq n$, where n is the order of the desired B-splines. The jth column then follows with the recurrence relation (4.4.1).

$$(4.5.1) \qquad M_{j,i-j+1+r} = \frac{(x-t_{i-j+1+r}) . M_{j-1,i-j+1+r} + (t_{i+r+1} - x) . M_{j-1,i-j+r+2}}{t_{i+1+r} - t_{i-j+1+r}}$$

$$r = 0,1,2,\ldots,j-1, \qquad 2 \leq j \leq n.$$

$$r = 0: \quad M_{j-1,i-j+1} = 0$$

$$r = j-1: \; M_{j-1,i+1} = 0.$$

Suppose, we have stored $M_{j-1,i-j+1+r}$ $(r = 1,2,\ldots,j-1)$ in $(b_s)_{s=0}^{j-2}$ (there is no need to store the zero-values of $M_{j-1,i-j+1}$ and $M_{j-1,i+1}$!), and that we are going to store $M_{j,i-j+1+r}$ $(r = 0,1,\dots,j-1)$ in $(b'_s)_{s=0}^{j-1}$, then (4.5.1) gives the following system:

$$(4.5.2)\quad \begin{cases} b'_0 = \dfrac{t_{i+1}-x}{t_{i+1}-t_{i-j+1}}.b_0 \\ b'_s = \dfrac{(x-t_{i-j+s+1}).b_{s-1}+(t_{i+s+1}-x).b_s}{t_{i+1+s}-t_{i-j+1+s}} \\ 2 \le j \le n \\ s = 1,2,\dots,j-1 \\ b_{j-1} = 0 \end{cases}$$

Put:

$$\begin{cases} \delta_s^L := x - t_{i-s} \\ \delta_s^R := t_{i+1+s} - x \\ s = 0,1,2,\dots,n-1 \end{cases}$$

then

$$x - t_{i-j+s+1} = \delta_{j-s-1}^L$$

$$t_{i+s+1} - x = \delta_s^R$$

$$t_{i+s+1} - t_{i-j+s+1} = t_{i+s+1} - x + x - t_{i-j+s+1} = \delta_s^R + \delta_{j-s-1}^L.$$

This converts (4.5.2) into

$$(4.5.2')\quad \begin{cases} b'_0 = \dfrac{\delta_0^R}{\delta_0^R+\delta_{j-1}^L}.b_0 \\ b'_s = \dfrac{\delta_{j-s-1}^L.b_{s-1}+\delta_s^R.b_s}{\delta_s^R+\delta_{j-s-1}^L} \\ 2 \le j \le n \\ s = 1,2,\dots,j-1 \\ b_{j-1} = 0. \end{cases}$$

Starting with $M_{1,i} = b_0 = \frac{1}{t_{i+1}-t_i}$, repeated application of (4.5.2') for $j = 1,2,\ldots,n$ produces the values of all the B-splines at x of order n. This results in the following algorithm:

```
begin
    $b_0$ := 1/(t[i+1]-t[i]);
    $\delta^L_0$ := x - t[i];
    $\delta^R_0$ := t[i+1] - x;
    for j from 2 to n
    do
        $\delta^L_{j-1}$ := x - t[i+1-j];
        $\delta^R_{j-1}$ := t[i+j] - x;
        saved := 0;
        b[j-1] := 0
        for s from 0 to j-1
        do
            term := $\delta^L_{j-s-1}$ * saved;
            saved := b[s];
             b[s] := (term + saved * $\delta^R_s$)/($\delta^R_s$ + $\delta^L_{j-s-1}$)
        od
    od
end.
```

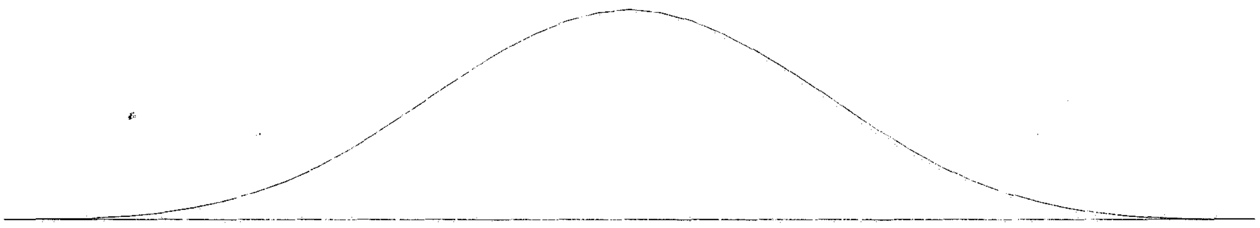
```
bsplval(knot,order,left,x,b)
        double *knot, x, *b;
        int     order, left;
       /*
        * The procedure ´bsplval´ uses generation of the complete
        * table implied by the recurrence relation (4.4.1)
        * Therefore, for a given knot set t[0],...,t[s] and a
        * given order, it will produce M[order,left-order+1], and
        * therefore left-order+1 >= 0, thus left >= order-1;
        * It will also produce M[order][left], and since the
        * recurrence relation uses t[left+order], it follows that
        * left+order <= s.
        * Then, if we use the procedure to calculate the B splines
        * on t[left],t[left+1] we will have to choose
        *           order-1 <= left <= s-order
        * The procedure produces in b[j] the values of the polynomial
        * of order ´order´, taken in x, that agrees with the B spline
        * M[order][left-order+j+1] on the interval t[left],t[left+1].
        *
        * b[0] contains M[order][left-order+1]
        * b[1] contains M[order][left-order+2]
        * b[2] contains M[order][left-order+3]
        *      .
        *      .
        * b[order-1] contains M[order][left]
        * There is no check on violation of the condition:
        *         order-1 <= left <= upperbound(knot)-order
        */
{       int i,j,r;
        double deltal[MAXORDER], deltar[MAXORDER];
        double term, saved;

        b[0] = 1/(knot[left+1] - knot[left]);
        deltal[0] = x - knot[left];
        deltar[0] = knot[left+1] - x;

        for (j=2; j<=order; j++)
            {   deltal[j-1] = x - knot[left+1-j];
                deltar[j-1] = knot[left+j] - x;
                saved = 0;
                b[j-1] = 0;
                for (r=0; r<j; r++)
                    {
                     term = deltal[j-r-1] * saved;
                     saved = b[r];
        b[r] = (term + saved * deltar[r]) / (deltar[r] + deltal[j-r-1]);
                    }
            }

}
```

```
/* Example 1:    */
main()
        /* We aim to calculate and plot the cubic B-spline M4,3 */
{       int i,left,last,flag;
        double t[11], b[4], x1, values[21];

        t[0]= -4.0; t[1]= -3.0; t[2]= -2.0;
        t[3]= -1.0;
        t[4]= -0.5;
        t[5]=  0.0;
        t[6]=  0.5;
        t[7]=  1.0;
        t[8]=  2.0; t[9]= 3.0; t[10]= 4.0;


        /* Leftmost evaluation point -1.0 ,
         * rightmost evaluation point 1.0 .
         * Step towards this last point with steps 0.1 .
         */

        x1= -1.0;

        /* Calculation: */
        for (i=0; i<=20; i++)
        {       interval(t,10,&left,x1,&flag,&last);
                /* if x1=t[7] take left=6: */
                left=min(left,6);
                /* Now for each x1 we have found the
                 * appropiate interval.
                 */

                bsplval(t,4,left,x1,b);
                /* Now b[0] contains M4,left-3 ,
                 *     b[1] contains M4,left-2 ,
                 *     b[2] contains M4,left-1 and
                 *     b[3] contains M4,left.
                 * On the span t[3],t[4] then b[3] contains M4,3
                 * on the span t[4],t[5] then b[2] contains M4,3
                 * on the span t[5],t[6] then b[1] contains M4,3 and
                 * on the span t[6],t[7] then b[0] contains M4,3.
                 */
                values[i] = b[6-left];
                printf("x = %f   M4,3 = %f\n", x1,values[i]);
                x1 = x1 + 0.1;
        }
/* Plotting  fig. 4.5.1 : */

pict(2,"M4,3");
         x1 = -1.0;
         for (i=0; i<=20; i++)
              {
               line(x1,values[i]);
               x1 += 0.1;
              }
         newpel();
         line(-1.0,0.0);
         line( 1.0,0.0);
         newpel();

         /* Put pen at position 0.5,0.5: */
         line(0.5,0.5);
         text("The B-spline M4,3");
         line(0.5,0.45);
         text("of example 1");
endpict();
}
```

```
RESULTS:


x = -1.000000   M4,3 = 0.000000
x = -0.900000   M4,3 = 0.000667
x = -0.800000   M4,3 = 0.005333
x = -0.700000   M4,3 = 0.018000
x = -0.600000   M4,3 = 0.042667
x = -0.500000   M4,3 = 0.083333
x = -0.400000   M4,3 = 0.141333
x = -0.300000   M4,3 = 0.207333
x = -0.200000   M4,3 = 0.269333
x = -0.100000   M4,3 = 0.315333
x = 0.000000   M4,3 = 0.333333
x = 0.100000   M4,3 = 0.315333
x = 0.200000   M4,3 = 0.269333
x = 0.300000   M4,3 = 0.207333
x = 0.400000   M4,3 = 0.141333
x = 0.500000   M4,3 = 0.083333
x = 0.600000   M4,3 = 0.042667
x = 0.700000   M4,3 = 0.018000
x = 0.800000   M4,3 = 0.005333
x = 0.900000   M4,3 = 0.000667
x = 1.000000   M4,3 = 0.000000
```

confer fig. 4.5.1

fig. 4.5.1
The B-spline M4,3
of example 1



```
/* Example 2:
 * We aim to calculate and plot the four polynomials that make up
 * a cubic Bspline. We remember from the procedure bsplval, that
 * it produces in b[j] the polynomial which agrees with the B spline
 * M4,left-4+j+1.
 * Hence b[6-left] contains  that polynomial for M4,3.
 * Further on we can take x anywhere we want, so we can calculate
 * b[6-left] and thus the polynomial over a bigger interval than that
 * where the B-spline M4,3 agrees with the polynomial.
 */
main()
{
        int i,j,left,s;
        double x,dx, biatx[4], t[11], pol[4][40];


        t[1]=t[2]=t[3]=t[0]= -3;
        t[4] = -2;
        t[5]=0;
        t[6]=1;
        t[8]=t[9]=t[10]=t[7]=3;

        /* Interval  -4,4 */
        x = -4;
        dx = 0.2;

        for (    i=0; i<=40; i++)
            {
                for (    left=3; left<7; left++)
                { bsplval(t,4,left,x,biatx);
                  pol[left-3][i] = biatx[6-left];
                }
                x = x + dx;
            }

pict(2,"bspline");
        with(); window(-1.0,-1.0,1.0,1.0);
                scale(0.3,2.0);
                thick(0.001);
        draw();
                for (j=0; j<=3; j++)
                    {
                        x = -4;
                        with();
                        draw(); for (i=0; i<=40; i++)
                                    {
                                     line(x, pol[j][i]);
                                     x = x + dx;
                                    }
                        ward();
                    }
        ward();
endpict();
}
```

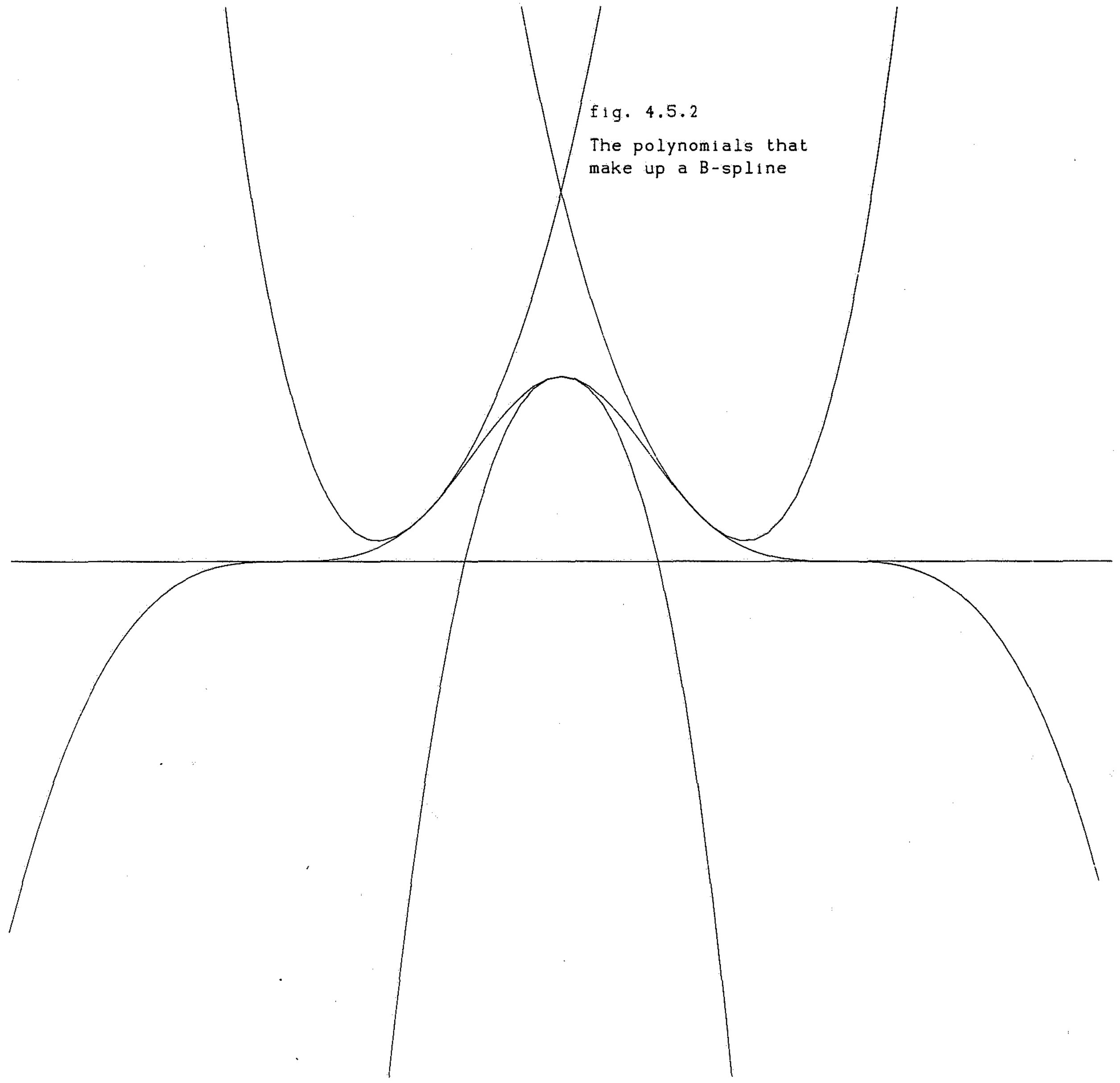


fig. 4.5.2

The polynomials that make up a B-spline

## 5. INTERPOLATION WITH SPLINES

### 5.1. Multiple knots

In (4.4.1) we did not require the knots to be distinct. This means that a given sequence of knots may contain identical knots!

If we have a non-decreasing sequence of knots $y_0, y_1, \ldots, y_s$, we will call $y_i$ a *$k_i$-fold knot*, or *a knot with multiplicity $k_i$*, if $y_i = y_{i+1} = \ldots = y_{i+k_i-1}$. The effect of a $k_i$-fold knot on a spline function $\phi$ of order n, is the decrease of continuous differentiability at $y_i$ to $n-1-k_i$ continuous derivatives.

If $k_i = n$, then there are no continuity requirements whatsoever, and we therefore restrict the multiplicity $k_i$ of any knot $y_i$ by the inequality $k_i < n$. If $k_i = 0$ for all $i \in \{0,1,2,\ldots,s\}$ then the spline function is continuous up to its $n-1^{th}$ derivative: the spline is actually just a polynomial and the knots are being called *pseudo-knots*.

The support of the B-spline $M_{n,i}$ is always the interval $[x_i, x_{i+n}]$. This implies that the width of the support of $M_{n,i}$ is equal to n spans: thus multiple knots induce spans of zero length, and a corresponding reduction of the support of the B-spline.

```
/* Example:
 * We now plot some B-splines with multiple knots.
 * Bearing in mind theorem 4.3.1, we expect the maximum
 * of a Bspline with multiple knots to get bigger as
 * the total mutliplicity increases!
 * In fig 5.1.1.' the same B-splines as in fig 5.1.1
 * are drawn on one horizontal axis.
 */
```

```
main()
{        int i,j,left,last,flag;
         double t[12], b[4], bspline[8][81], x1;

         t[0]=t[1]=t[2]=t[3]= -1.0;
         t[4]= -0.5;
         t[5]=  0.0;
         t[6]=t[7]= 0.5;
         t[8]=t[9]=t[10]=t[11]= 1.0;

         x1 = -1.0;

         for (i=0; i<=80; i++)
             {   interval(t,8,&left,x1,&flag,&last);
                 left=min(left,7);
                 bsplval(t,4,left,x1,b);
                 x1 += 0.025;

                 for (j=0; j<4; j++)
                      bspline[left-3+j][i] = b[j];
             }

         pict(2,"multiple knots");
          with();window(-1.0,-1.0,1.0,1.0);
                 scale(1.0,0.5);
          draw();
                 for (j=0; j<8; j++)
                     {   x1 = -1.0;
                         with(); translate(0.0, 0.75 - j*0.25);
                         draw(); for (i=0; i<=80; i++)
                                       {   line(x1,bspline[j][i]);
                                           x1 += 0.025;
                                       }

                         ward();
                     }
                 line(1.0,0.75);
                 text("M4,0");
                 line(1.0,0.50);
                 text("M4,1");
                 line(1.0,0.25);
                 text("M4,2");
                 line(1.0,0.00);
                 text("M4,3");
                 line(1.0,-0.25);
                 text("M4,4");
                 line(1.0,-0.50);
                 text("M4,5");
                 line(1.0,-0.75);
                 text("M4,6");
                 line(1.0,1.0);
                 text("M4,7");
          ward();
          line(0.5,0.85);
          text("fig. 5.1.1");
         endpict();
}
```

fig. 5.1.1

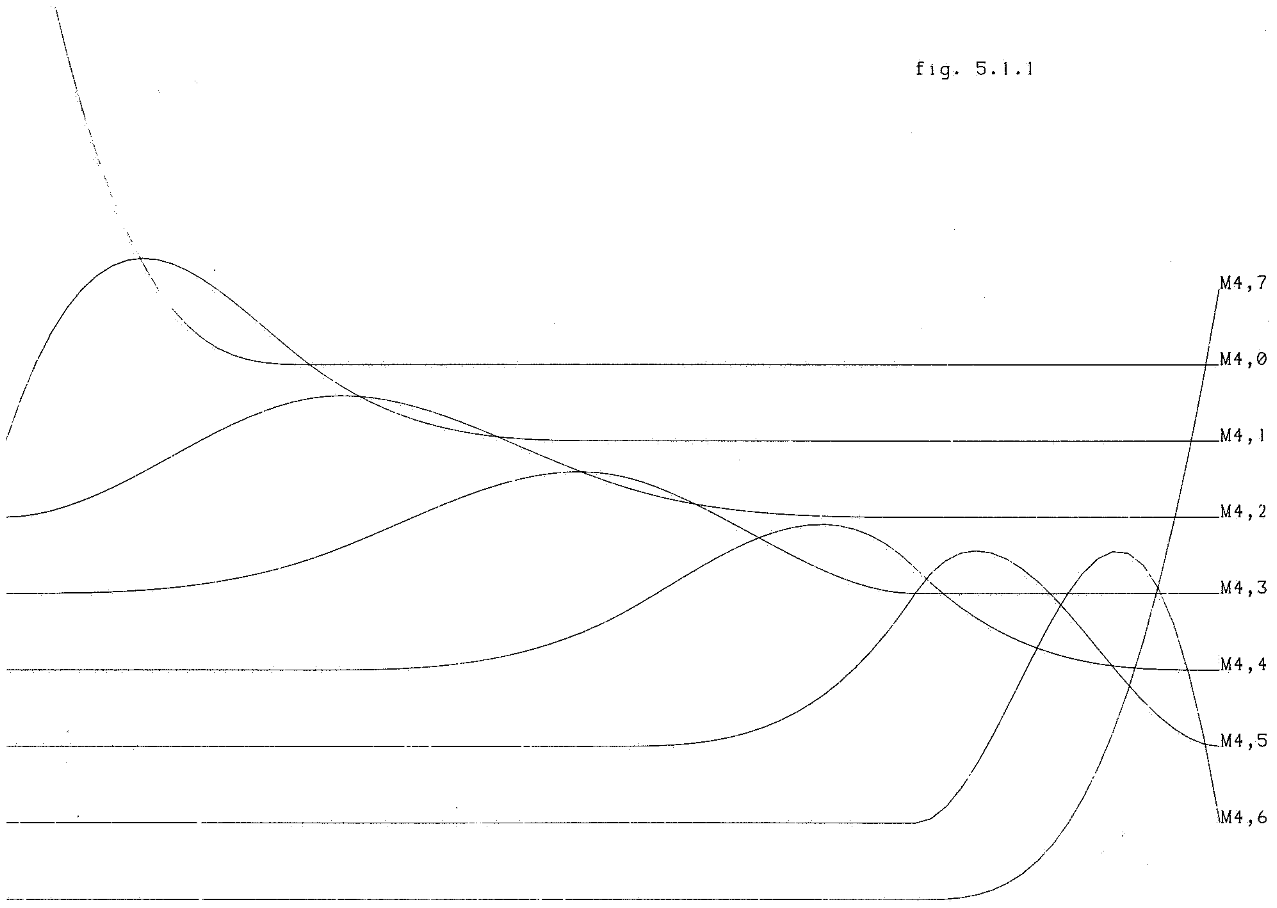
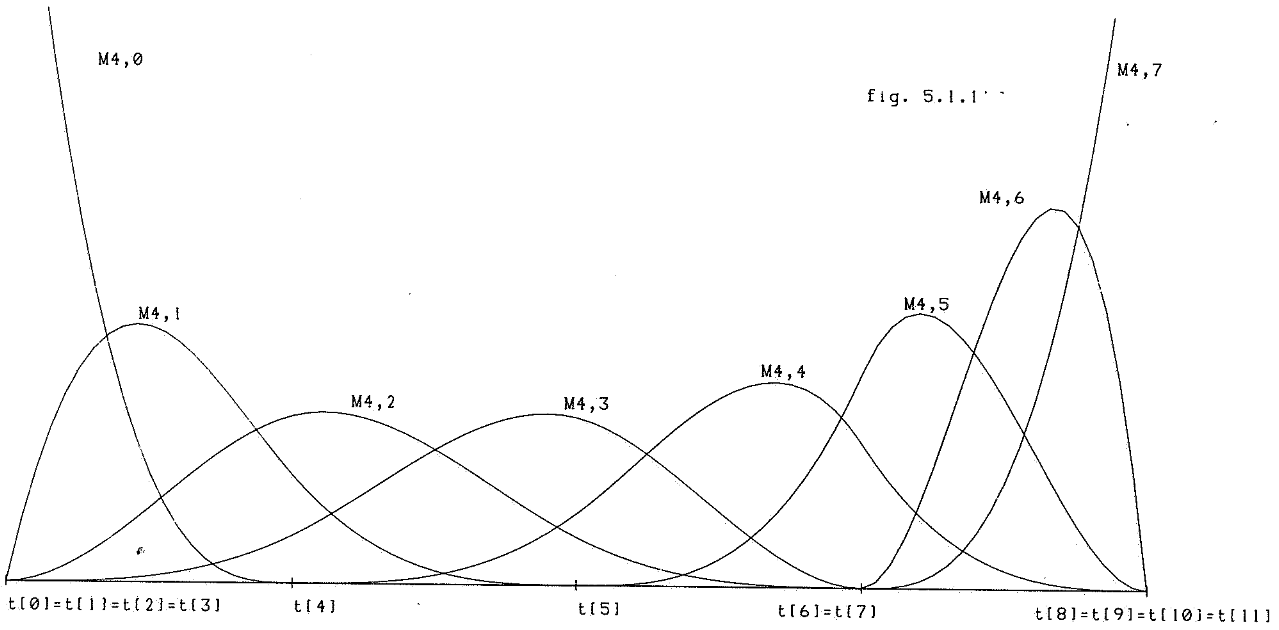


fig. 5.1.1''



## 5.2. The B-spline basis

We denote the linear space of spline functions of order k on the non-decreasing knot sequence $\xi = \{y_0, y_1, \ldots, y_s\}$ by $S_{k,\xi}$.

Without proof, we now give the theorem of CURRY and SCHOENBERG (1966), which postulates that any $\phi \in S_{k,\xi}$ can be expressed as the sum of multiples of $k^{th}$ order B-splines, that are defined on a knot set that envelopes $\xi$.

A proof can be found in [2] or [6].

THEOREM 5.2.1.

(i) *Let* $\xi = \{y_0, y_1, \ldots, y_s\}$ *be a non-decreasing knot sequence. We consider a* $\phi \in S_{k,\xi}$.

(ii) *Form* $\nu = \{x_0, x_1, \ldots, x_\ell\}$ *from* $\xi$ *by taking in increasing order, the numbers in* $\xi$ *at most once. Then* $\nu$ *is a strictly increasing sequence. Assume that for* $i = 0,1,\ldots,\ell$, *the number* $x_i$ *occurs precisely* $\alpha_i$ *times in* $\xi$, *i.e. multiplicity* $(y_i) = \alpha_i$. *Put*

$$n := k + \sum_{i=1}^{\ell-1} \alpha_i . \tag{5.2.1}$$

(iii) *Let* $\{s_i\}_{i=0}^{2k-1}$ *be any non-decreasing sequence, such that*

$$s_0 \le s_1 \le \ldots \le s_{k-1} \le x_0$$

*and*

$$x_\ell \le s_k \le s_{k+1} \le \ldots \le s_{2k-1}.$$

*Now* $\phi$ *can be expressed as the sum of multiples of B-splines* $M_{k,0}, M_{k,1}, \ldots, M_{k,n-1}$ *defined on the* (n+k) *knots* $t_0, t_1, \ldots, t_{n+k-1}$, *which are given by:*

$$
\left.\begin{array}{l}
\left.\begin{array}{l}
\left.\begin{array}{rcl} t_0 &=& s_0 \\ t_1 &=& s_1 \\ t_2 &=& s_2 \\ &\vdots& \\ t_{k-1} &=& s_{k-1} \end{array}\right\} \#s:\ k \\
\left.\begin{array}{rcl} t_k &=& y_1 \\ t_{k+1} &=& y_2 \\ &\vdots& \\ t_{n-2} &=& y_{s-2} \\ t_{n-1} &=& y_{s-1} \end{array}\right\} \#y:\ \sum_{i=1}^{\ell-1} \alpha_i
\end{array}\right\} \#:\ n \\
\left.\begin{array}{rcl} t_n &=& s_k \\ &\vdots& \\ t_{n+k-2} &=& s_{2k-2} \\ t_{n+k-1} &=& s_{2k-1} \end{array}\right\} \#s:\ k
\end{array}\right\} \#t:\ n+k
$$

*$\phi$ is to be considered as a function on* $[t_{k-1}, t_n]$.

COROLLARY. *The* B-*splines* $M_{k,0}, M_{k,1}, \ldots, M_{k,n-1}$ *form a basis for* $S_{k,\xi}$, *with dimension* n *as determined in* (5.2.1).

The theorem leaves open the choice of the first k and the last k knots. A convenient choice is $t_0=t_1=\ldots=t_{k-1}=x_0$ and $t_n=t_{n+1}=\ldots=t_{n+k-1} = x_\ell$, which then allows us to include the choice of these knots under the same pattern as the choice of the other knots by taking

$$\alpha_0 = \alpha_\ell = k.$$

The choice of $\xi$ specifies the desired amount of smoothness at a point in terms of the number of knots at that point, with *fewer* knots corresponding to *more* continuity.

EXAMPLE. Let:

$$y_0 = y_1 = -2$$

$$y_2 = -1$$

$$y_3 = 0$$
$$y_4 = y_5 = 1$$
$$y_6 = y_7 = y_8 = 2$$

$k = 4.$

Then:

| | |
|---|---|
| $x_0 = -2$ | $\alpha_0 = 2$ |
| $x_1 = -1$ | $\alpha_1 = 1$ |
| $x_2 = 0$ | $\alpha_2 = 1$ |
| $x_3 = 1$ | $\alpha_3 = 2$ |
| $x_4 = 2$ | $\alpha_4 = 3$ |

$n := 4 + (1+1+2) = 8$ (5.2.1).

Take:

$$t_0 = t_1 = t_2 = t_3 = -2$$
$$t_4 = -1$$
$$t_5 = 0$$
$$t_6 = t_7 = 1$$
$$t_8 = t_9 = t_{10} = t_{11} = 2.$$

Now any spline $\phi$ of order 4 with knots $\{y_0,\ldots,y_8\}$ can be expressed as $\sum_{j=0}^{7} c_j M_{4,j}$, $c_j \in \mathbb{R}$ (cf. fig. 5.1.1), where the $M_{4,j}$ are being taken on the knots $t_0,t_1,\ldots,t_{11}$.

## 5.3. B-representation for splines

By virtue of theorem 5.2.1, every spline has a representation in terms of B-splines, of which we now give the definition:

DEFINITION 5.3.1. The *B-representation* for a spline function $\phi$ of order k on the knots $y_0, y_1, \ldots, y_s$ consists of:

(i) k: order of $\phi$

(ii) n: dimension of B-spline basis, obtained from (5.2.1).

(iii) the vector $t = (t_i)_{i=0}^{n+k-1}$, containing the *extended knot sequence*, constructed in the way of theorem 5.2.1.

(iv) the vector $\beta = (\beta_i)_{i=0}^{n-1}$ of the coefficients of $\phi$ with respect to the basis $(M_{k,i})_{i=0}^{n-1}$.

In terms of these quantities, the value of $\phi$ at a point $x \in [t_{k-1}, t_n]$, is given by

$$f(x) = \sum_{i=0}^{n-1} \beta_i . M_{k,i}(x).$$

In particular, if $t_j \leq x \leq t_{j+1}$, for some $j \in [k-1, n-1]$, then

$$f(x) = \sum_{i=j-k+1}^{j} \beta_i . M_{k,i}(x).$$

From this we can see a nice point of representing a spline function as a sum of B-splines: because of the local support of the B-splines, only a finite (viz. order of the spline) number of B-splines contribute to the spline, while they can be calculated accurately with the algorithm of 4.5.

## 5.4. Interpolation

We now are in a position to tackle the problem of kth order spline interpolation using B-splines. There are several possibilities, but they all include an extension of the given knot set by 2(k-1) knots, which can be taken arbitrarily (cf. theorem 5.2.1). First of all we can define the strictly increasing knot sequence $t_0, t_1, \ldots, t_\ell$ beforehand, extend it, and ask for the kth order spline $\phi$ which interpolates a given function g in the data points $(x_0, g(x_0)), (x_1, g(x_1)), \ldots, (x_m, g(x_m))$, with the abscissae

$x_0, x_1, \ldots, x_m$ all in $[t_0, t_\ell]$. The points $x_0, \ldots, x_m$ will be called *nodes*.

SCHOENBERG and WHITNEY [13] have shown that there exists exactly one $k^{th}$ order spline $\phi$ that agrees with g at $x_0, x_1, \ldots, x_m$, where $m = k+\ell-2$, if and only if $M_{k,i}(x) \neq 0$, $i = 0,1,\ldots,m$, and that

$$\phi = \sum_{i=0}^{m} a_i \cdot M_{k,i}$$

for certain coefficients $a_i$. These coefficients can be found from the linear system

$$\sum_{i=0}^{m} a_i \cdot M_{k,i}(x_j) = g(x_j), \qquad j = 0,1,\ldots,m.$$

Another possibility is to ask for the $k^{th}$ order spline which interpolates the data points $(x_0,y_0),(x_1,y_1),\ldots,(x_\ell,y_\ell)$, and to take the nodes $x_0, x_1, \ldots, x_\ell$ as knots of the spline. Extend this knotset, and use theorem 5.2.1:

$$\phi = \sum_{i=0}^{k+\ell-2} c_i \cdot M_{k,i}.$$

We can determine the coefficients $c_i$ via the system

$$(5.4.1) \qquad \sum_{i=0}^{k+\ell-2} c_i \cdot M_{k,i}(x_j) = y_j, \qquad j = 0,1,\ldots,\ell$$

but see directly that we will need k-2 additional relations to come to a solution of the system!

We shall restrict ourself to the last possibility, and take $k = 4$, thus needing 2 additional relations; these can be obtained as discussed in section 3.2.

<u>EXAMPLE</u>. Assume that in the points $x_0 < x_1 < \ldots < x_{10}$ the values $y_0, y_1, \ldots, y_{10}$ are given. Take $x_0, x_1, \ldots, x_{10}$ as knots $t_3, t_4, \ldots, t_{13}$, and extend this set with 3 additional knots $\leq t_3$, and 3 additional knots $\geq t_{13}$. As we have seen, there is no prescription for the choice of the additional knots. In order to have 3 non-zero B-splines at $t_3$ and $t_{13}$, we choose $t_0 = t_1 = t_2 = x_0 - 1$ and $t_{14} = t_{15} = t_{16} = x_{10} + 1$. Compute at each node the three B-splines

that are non-zero there, e.g. $M_3$, $M_4$ and $M_5$ at $t_6$.
This can be done with 'bsplval' (cf. 4.5). If the value at $t_6$ is $y_3$, we have from (5.4.1):

$$y_3 = c_3 . M_{4,3} + c_4 . M_{4,4} + c_5 . M_{4,5}$$

in which $c_3$, $c_4$ and $c_5$ are the unknowns. For each of the 11 nodes there will be a similar equation. Since there are 11 + 2 = 13 B-splines, there are 13 coefficients to be determined, and so two more equations are needed to be able to compute the coefficients:

<u>POSSIBILITY 1</u>: free ends at $x_0$ and $x_{10}$: $\phi''(x_0) = \phi''(x_{10}) = 0$. We then get

$$0 = c_0 . M''_{4,0} + c_1 . M''_{4,1} + c_2 . M''_{4,2}$$

$$0 = c_{10} . M''_{4,10} + c_{11} . M''_{4,11} + c_{12} . M''_{4,12}$$

and this leads to

$$(5.4.2) \qquad \begin{pmatrix} M''_0 & M''_1 & M''_2 & & & & \\ M^0_0 & M^0_1 & M^0_2 & 0 & & & \\ 0 & M^1_1 & M^1_2 & M^1_3 & 0 & & \\ & 0 & M^2_2 & M^2_3 & M^2_4 & 0 & \\ & & \ddots & \ddots & \ddots & \ddots & \ddots \\ & & & & & & 0 \\ & & & 0 & M^{10} & M^{10}_{11} & M^{10}_{12} \\ & & & & M''_{10} & M''_{11} & M''_{12} \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ \vdots \\ \vdots \\ c_{11} \\ c_{12} \end{pmatrix} = \begin{pmatrix} 0 \\ y_0 \\ y_1 \\ \vdots \\ \vdots \\ \vdots \\ y_{10} \\ 0 \end{pmatrix}$$

$(M^j_i = M_{4,i}(x_j))$.

<u>POSSIBILITY 2</u>: built-in ends at $x_0$ and $x_{10}$: $\phi'(x_0) = a$, $\phi'(x_0) = b$, $a,b \in \mathbb{R}$.
We then get

$$a = c_0 . M'_{4,0} + c_1 . M'_{4,1} + c_2 . M'_{4,2}$$

$$b = c_{10} . M'_{4,10} + c_{11} . M'_{4,11} + c_{12} . M'_{4,12}$$

which leads to a similar system as (5.4.2).

POSSIBILITY 3: Let $x_1$ and $x_9$ not be knots. This means that we choose the first and second cubic polynomial pieces to coincide in $[x_0,x_2]$, and have the prescribed value in $x_1$. This leads to:

$$\begin{pmatrix}
M_0(x_0) & M_1(x_0) & M_2(x_0) & & & & & & \\
M_0(x_1) & M_1(x_1) & M_2(x_1) & M_3(x_1) & & & & & \\
 & M_1(x_2) & M_2(x_2) & M_3(x_2) & & & & & \\
 & & M_2(x_3) & M_3(x_3) & M_4(x_3) & & & & \\
 & & & \ddots & \ddots & \ddots & & & \\
 & & & & & M_7(x_8) & M_8(x_8) & M_9(x_9) & \\
 & & & & & M_7(x_9) & M_8(x_9) & M_9(x_9) & M_{10}(x_9) \\
 & & & & & & M_8(x_{10}) & M_9(x_{10}) & M_{10}(x_{10})
\end{pmatrix}
\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_8 \\ c_9 \\ c_{10} \end{pmatrix}
=
\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_8 \\ y_9 \\ y_{10} \end{pmatrix}$$

In this situation, we have a B-spline basis of 11 B-splines, instead of the 13 of possibility 1) and 2), and of course the extended knot set has only 15 knots.

In all three possibilities, a banded linear system needs to be solved, and for the possibilities 1) and 2), we even need to be able to calculate derivatives of B-splines.

In next sections we shall develop algorithms which tackle these problems.

## 5.5. Differentiation of splines and B-splines

In order to find the derivatives of a B-spline, we go back to the definition of a B-spline, and to the theory of divided differences. Consider $f_k(x;z) = (z-x)_+^{k-1}$ (cf. definition 4.2.2). If we take the $n^{th}$

divided difference of $f_k$, with respect to $z = x_i, x_{i+1}, \ldots, x_{i+n}$, then we get with theorem A1:

$$f_k[x;x_i,x_{i+1},\ldots,x_{i+k}] = \sum_{j=i}^{i+n} \frac{f_k(x;x_j)}{\omega'(x_j)}$$

$$= \sum_{j=i}^{i+n} \frac{(x_j-x)_+^{k-1}}{\omega'(x_j)}$$

Then

$$\frac{d}{dx} f_k[x;x_i,x_{i+1},\ldots,x_{i+n}] = -(k-1)\,.\sum_{j=i}^{i+n} \frac{(x_j-x)_+^{k-2}}{\omega'(x_j)}$$

$$= -(k-1)\,.f_{k-1}[x;x_i,\ldots,x_{i+n}].$$

This gives for $n = k-1$:

$$\frac{d}{dx} f_k[x;x_i,x_{i+1},\ldots,x_{i+k-1}] = -(k-1)\,.f_{k-1}[x;x_i,\ldots,x_{i+k-1}]$$

$$= -(k-1)\,.M_{k-1,i}.$$

Now, using the definition of divided differences (def. A.1), we get:

$$M_{n,i}(x) = f_n[x;x_i,x_{i+1},\ldots,x_{i+n}]$$

$$= \frac{f_n[x;x_i,x_{i+1},\ldots,x_{i+n-1}]-f_n[x;x_{i+1},x_{i+1},x_{i+2},\ldots,x_{i+n}]}{(x_i-x_{i+n})}$$

and

$$\frac{d}{dx} M_{n,i}(x) = \frac{-(n-1)\,.M_{n-1,i}+(n-1)\,.M_{n-1,i+1}}{(x_i-x_{i+n})}$$

(5.5.1) $$\frac{d}{dx} M_{n,i}(x) = \frac{n-1}{x_{i+n}-x_i}\,.(M_{n-1,i}-M_{n-1,i+1})$$

We shall derive a formula for the $i^{th}$ derivative of a spline function

$$\phi = \sum_i \alpha_i\,.M_{n,i}$$

Taking $\alpha_i = 0$, $i \neq p$, and $\alpha_p = 1$, then gives the $j^{th}$ derivative of the B-spline $M_{n,p}$.

Assume n to be a fixed integer, and put

$$\phi(x) = \sum_{i=r-n+1}^{s-1} \alpha_i . M_{n,i}(x)$$

where $x \in [t_r,t_s]$. Then, with (5.5.1), we have

$$\phi'(x) = \sum_{i=r-n+1}^{s-1} \alpha_i . \frac{d}{dx} M_{n,i}(x)$$

$$(5.5.2) \qquad \phi'(x) = \sum_{i=r-n+1}^{s-1} \alpha_i . \frac{n-1}{x_{i+n}-x_i} . (M_{n-1,i}-M_{n-1,i+1})$$

$$= \sum_{i=r-n+1}^{s-1} \beta_i . (n-1) . (M_{n-1,i}-M_{n-1,i+1})$$

$$= (n-1) . \{\beta_{r-n+1} . M_{n-1,r-n+1} - \beta_{r-n+1} . M_{n-1,r-n+2} + \beta_{r-n+2} . M_{n-1,r-n+2}$$

$$- \beta_{r-n+2} . M_{n-1,r-n+3} + \ldots + \beta_{s-1} . M_{n-1,s-1}$$

$$- \beta_{s-1} . M_{n-1,s}\}$$

$$= (n-1) . \{\beta_{r-n+1} . 0 + \sum_{i=r-n+2}^{s-1} (\beta_i - \beta_{i-1}) . M_{n-1,i} + \beta_{s-1} . 0\}$$

since $x \in [t_r,t_s]$, and thus $M_{n-1,i} = 0$ if $i \notin [r-n+2,s-1]$

$$(5.5.3) \qquad \phi'(x) = \sum_{i=r-n+2}^{s-1} (n-1) . (\beta_i - \beta_{i-1}) . M_{n-1,i}$$

where

$$\beta_i = \frac{\alpha_i}{x_{i+n}-x_i} .$$

If we put $\alpha_i^1 = (n-1) . (\beta_i - \beta_{i-1})$, then (5.5.3) gives:

$$(5.5.4) \qquad \phi'(x) = \sum_{i=r-n+2}^{s-1} \alpha_i^1 . M_{n-1,i}$$

where

$$\alpha_i^1 = (n-1).(\beta_i-\beta_{i-1})$$

$$= (n-1).(\frac{\alpha_i}{x_{i+n}-x_i} - \frac{\alpha_{i-1}}{x_{i+n-1}-x_{i-1}})$$

$$i = r-n+2,\ldots,s-1.$$

Repeat the operation of (5.5.2) on (5.5.4):

$$\phi''(x) = \sum_{i=r-n+2}^{s-1} \alpha_i^1 . \frac{n-2}{x_{i+n-1}-x_i} . (M_{n-2,i}-M_{n-2,i+1})$$

$$= \sum_{i=r-n+3}^{s-1} (n-2).(\beta_i^1-\beta_{i-1}^1).M_{n-2,i}$$

where

$$\beta_i^1 = \frac{\alpha_i^1}{x_{i+n-1}-x_i} .$$

If we put $\alpha_i^2 = (n-2).(\beta_i^1-\beta_{i-1}^1)$, then this gives:

$$\phi''(x) = \sum_{i=r-n+3}^{s-1} \alpha_i^2.M_{n-2,i}$$

where

$$\alpha_i^2 = (n-2).(\beta_i^1-\beta_{i-1}^1)$$

$$= (n-2).(\frac{\alpha_i^1}{x_{i+n-1}-x_i} - \frac{\alpha_{i-1}^1}{x_{i+n-2}-x_{i-1}})$$

$$i = r-n+3,\ldots,s-1.$$

Repeated operation of (5.5.2) produces the following formula for the $j^{th}$ derivative of a spline:

$$(5.5.5) \qquad D^j(\sum_{i=r-n+1}^{s-1} \alpha_i.M_{n,i}) = \sum_{i=r-n+1+j}^{s-1} \alpha_i^j.M_{n-j,i}$$

where $\alpha_i^0 = \alpha_i$, $i = r-n+1,\ldots,s-1$

$$\alpha_i^j = (n-j)(\frac{\alpha_i^{j-1}}{x_{i+n-j+1}-x_i} - \frac{\alpha_{i-1}^{j-1}}{x_{i+n-j}-x_{i-1}})$$

$j = 1,\dots,n-1$, $i = r-n+1+j,\dots,s-1$.

If we take $\alpha_i = 0$ if $i \neq p$, and $\alpha_p = 1$ we get from (5.5.5):

$$(5.5.6) \qquad D^j M_{n,p} = \sum_{i=r+1+j-n}^{s-1} \alpha_i^j . M_{n-j,i}$$

where

$$\alpha_p^0 = 1$$

$$\alpha_i^j = (n-j).\left(\frac{\alpha_i^{j-1}}{x_{i+n-j+1}-x_i} - \frac{\alpha_{i-1}^{j-1}}{x_{i+n-j}-x_{i-1}}\right)$$

and

$$j = 1,2,\dots,n-1, \; i = r-n+1+j,\dots,s-1.$$

```
double
splderiv(knot,order,basdim,bcoef,x,der)
        double knot[], bcoef[], x;
           int order, basdim, der;
        /* The procedure calculates the value of the der'th derivative
         * of the spline given in its B-representation
         *      (knot, order, basdim, bcoef),  at
         * the point x. According to theorem (5.2.1), the spline is to
         * be considered as a function on the interval
         * knot[order-1],knot[basdim]. The point x shall therefore
         * be restricted to be in this interval.
         * The method is to locate the interval knot[left],knot[left+1]
         * such that knot[left] <= x < knot[left+1], or
         *           knot[left] <= x <= knot[left+1] when left+1=basdim,
         * and then use the formula (5.5.5) to do the calculation.
         * In this calculation  only  the for the interval relevant
         * B-splines are used, and  the summation  is done over
         * 'order-der' indices.
         */
{       double value, alpha[MAXBASDIM], b[MAXORDER], dl, dr,saved,trans;
           int left, f,  i, j, imax, ll;

        if ( x < knot[order-1] || x > knot[basdim] )
             bsplerror(DERIV);

        imax = basdim + order - 1;
        value = 0;

        if (der < order)        /* else value stays 0 */
       {interval(knot, imax, &left, x, &f, &ll);

        /* if x = knot[basdim],  take left = basdim-1 : */
        left = min(left,basdim-1);

        /* first difference the coefficients 'der' times. */
        for ( i = left-order+1; i <= left; i++)
              alpha[i] = bcoef[i];

        for ( j = 1; j <= der; j++)
            {  saved = alpha[left-order+j];
               for ( i = left-order+1+j; i <= left; i++)
                   {   dr = knot[i+order+1-j] - knot[i];
                       dl = knot[i+order  -j] - knot[i-1];

                       trans = alpha[i];
                       alpha[i] = (order-j)*(trans/dr - saved/dl);
                       saved = trans;
                   }
            }

        /* Now calculate the values of the B-splines of order (order-der)
         * at x for the relevant B-splines, for the interval
         * knot[left],knot[left+1].
         */

        bsplval(knot,order-der,left,x,b);

        /* and now b[0] contains M[order-der][left-order+1+der],
         * and b[order-der-1] contains  M[order-der][left].
         * Calculate the sum.
         */

        for ( i = left-order+1+der; i <= left; i++)
              value += alpha[i] * b[i-left+order-1-der];

       }
        return(value);
}
```

## 5.6. Solving the linear, banded system

We aim at solving the system

$$\sum_{j=0}^{n-1} a_j . M_{k,j}(x_i) = y_i , \qquad i = 0,1,\dots,n-1.$$

Since all $M_{k,j}$ are positive (cf. theorem 4.5.2) the system has a positive definite matrix, and Gauss-elimination *without* a pivoting strategy can be performed (cf. [2]).

We will store the band matrix M of degree n in the following way:

Let

$a$ = # diagonals *above* the main diagonal

$u$ = # diagonals *under* the main diagonal.

Use a double-subscripted array A[i][j] for the storage of the a+u+1 diagonals. Put the most upper diagonal in A[0], the main diagonal in A[a], and the lowest diagonal in A[a+u], in the following way:

$$\begin{pmatrix} a_{00} & a_{01} & & & & & \\ a_{10} & a_{11} & a_{12} & & & & \\ a_{20} & a_{21} & a_{22} & \ddots & & & \\ & a_{21} & a_{32} & \ddots & \ddots & & \\ & & a_{42} & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & \ddots & \ddots \\ & & & & \ddots & \ddots & a_{n-2,n-1} \\ & & & & 0 & 0 & a_{n-1,n-1} \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & a_{01} & a_{12} & \cdots & a_{n-2,n-1} \\ a_{00} & a_{11} & a_{22} & \cdots & a_{n-1,n-1} \\ a_{10} & a_{21} & a_{32} & \cdots & 0 \\ a_{20} & a_{31} & a_{42} & \cdots 0 & 0 \end{pmatrix}$$

In the new representation, the columns correspond to the columns of the original matrix.

```
decband(mat,dim,above,under)
        double mat[][MAXPOINTS];
        int    dim,above,under;
        /* The diagonals of the matrix are stored in the rows of mat
         * Above = #diagonals above main diagonal,
         * Under = #diagonals under main diagonal.
         * There are above+under+1 rows in mat.
         * The most upper diagonal is stored in row mat[0], the main
         * diagonal in row mat[above].
         * Dim is the rank of the bandmatrix.
         */
{       int middle,i,j,k,jmax,kmax;
        double fac,pivot;
        middle = above;
        if (under == 0)         /* upper triangular matrix. */
           {    for (i=0; i<dim; i++)
                    {if (mat[middle][i] == 0)
                         bsplerror(SINGULAR);
                    }
           }
        else
        if (above == 0)    /* Then mat is a lower triangular matrix. */
                                /* In order to be able to use the
                                 * solving procedure, we must get
                                 * all 1's on the main diagonal !
                                 * Divide column elements by the
                                 * corresponding diagonal elements.
                                 */
           {    for (i=0; i<dim; i++)
                    {pivot = mat[middle][i];    /* diagonal element */
                     if (pivot == 0)
                         bsplerror(SINGULAR);
                     else
                        {jmax = min(under,dim-1-i);
                         for (j=1; j<= jmax; j++)
                              mat[middle+j][i] /= pivot;
                        }
                    }
           }
        else
           {    for (i=0; i<dim; i++)   /* column 0 through dim-2. */
                    {pivot = mat[middle][i];
                     if (pivot == 0)
                         bsplerror(SINGULAR);
                     jmax = min(under,dim-1-i);
                     kmax = min(above,dim-1-i);
                     for (j=1; j<=jmax; j++)
                         {fac = mat[middle+j][i] / pivot;
                          for (k=1; k<=kmax; k++)
                          mat[middle-k+j][i+k] -= fac * mat[above-k][i+k];

                          mat[middle+j][i] = fac;
                         }
                    }
                if ( mat[middle][dim-1] == 0)
                     bsplerror(SINGULAR);
            }
}
```

```
solband(mat,dim,above,under,b)
        double mat[][MAXPOINTS], b[];
        int    above,under,dim;
        /* Matrix stored as in decband. */
        /* solution in b. */
{
        int i,j,jmax,middle;
        middle = above;

        if (under == 0 && above == 0)   /* One (main) diagonal */
           {for (i=0; i<dim; i++)
                 b[i] /= mat[0][i] ;
           }
        else
           {if (under > 0)      /* then forward substitution */
               {for (i=0; i<dim-1; i++)
                    {
                     jmax = min(under,dim-1-i);
                     for (j=1; j<= jmax; j++)

                          b[j+i] -= b[i] * mat[middle+j][i];

                    }
               }
            if (above > 0)      /* then backward substitution */
               {for (i=dim-1; i>=0; i--)
                    {
                     jmax = min(above,i);
                     b[i] /= mat[middle][i];


                     for (j=1; j<=jmax; j++)

                          b[i-j] -= b[i] * mat[middle-j][i];

                    }
                }
        }
}
```

```
brep(xnode,ynode,basdim ,t,bcoef,extra,phiacc0,phiacc1)
        double xnode[],ynode[],t[],bcoef[],phiacc0,phiacc1;
           int basdim,extra;
        /* The procedure generates the B-representation (t,4,basdim,bcoef)
         * for the spline interpolant of order 4, interpolating the nodes
         * xnode in ynode. The 2 extra conditions are being determined
         * by the value of 'extra'.
         * This feature accounts for the choice of order 4.
         * Let nnode = #nodes
         * Input:
         *      The nodes xnode and their y-values ynode.
         *      The integer basdim :
         *              -the number of nodes decide basdim:
         *              -in case extra = FREE or BUILT then basdim=nnode+2
         *              -in case extra = NOT then basdim=nnode.
         *      The parameters extra, phiacc0 and phiacc1:
         *
         * If extra = FREE      free ends: phi''=0 at node[0] and
         *                                   at node[nnode-1].
         *                      basdim = nnode + 2
         *    extra = BUILT     built in ends:
         *                                 phi'=phiacc0 at node[0],
         *                                 phi'=phiacc1 at node[nnode-1].
         *                      basdim = nnode + 2
         *    extra = NOT       node[1] and node[nnode-2] are not knots.
         *                      basdim = nnode
         * Output:
         *    The knots t
         *    The B-coefficients bcoef.
         *
         * In the main program, t should be declared as an array of
         * (nnode+6) doubles in case extra = FREE or BUILT, and of (nnode+4)
         * doubles if extra = NOT, because of the extension with 2*3 knots.
         *
         * bcoef should be declared as an array of basdim doubles.
         */
```

```
{
   double b[4], mat[5][MAXPOINTS ], splderiv();

      int i,j,n;

   n = basdim - 3;


   if ( extra == NOT )
  {
   /* Create the extended knot set: */
   t[0]=t[1]=t[2]=xnode[0]-1.0;
   t[3]=xnode[0];
   for (i=4; i<basdim; i++)
           t[i]=xnode[i-2];
   t[basdim]=xnode[basdim-1];
   t[basdim+1]=t[basdim+2]=t[basdim+3]=xnode[basdim-1]+1.0;


   /* Fill the bandmatrix: */

   bsplval(t,4,3,xnode[0],b);
   for (i=0; i<=2; i++)
         mat[2-i][i] = b[i];


   bsplval(t,4,3,xnode[1],b);
   for (i=0; i<=3; i++)
         mat[3-i][i] = b[i];

   for (j=2; j<=basdim-3; j++)
        {   bsplval(t,4,j+2,xnode[j],b);
            for (i=0; i<=2; i++)
                  mat[3-i][j-1+i] = b[i];
        }


   bsplval(t,4,basdim-1,xnode[basdim-2],b);
   for (i=0; i<=3; i++)
         mat[4-i][3+i] = b[i];


   bsplval(t,4,basdim-1,xnode[basdim-1],b);
   for (i=0; i<=2; i++)
         mat[4-i][4+i] = b[i+1];



   /* Fill y. We use bcoef for y. */
   for (i=0; i<basdim; i++)
         bcoef[i] = ynode[i];

   /* Now solve the system: */
   decband(mat,basdim,2,2);
   solband(mat,basdim,2,2,bcoef);
  }


   else
```

```
{
 /* Create the extended knot set: */
 t[0]=t[1]=t[2]=xnode[0]-1.0;
 for (i=0; i<=n; i++)
         t[3+i] = xnode[i];
 t[n+4]=t[n+5]=t[n+6]=xnode[n]+1.0;


 /* Fill the bandmatrix: */
 /* Calculate, depending on 'extra', the appropiate B-spline
  * values in node[0] and node[n], and put them in 'mat':
  */
 for (i=0; i<=2; i++)
     {   bcoef[i] = 1;
         mat[2-i][i] = splderiv(t,4,basdim,bcoef,xnode[0],2-extra);
         bcoef[i] = 0;
     }
 for (i=0; i<=2; i++)
     {   bcoef[n+i] = 1;
         mat[4-i][n+i]=splderiv(t,4,basdim,bcoef,xnode[n],2-extra);
         bcoef[n+i] = 0;
     }
 /* Now fill the rest of the bandmatrix: */
 for (j=0; j < n; j++)
     {   bsplval(t,4,j+3,xnode[j],b);
         for (i=0; i<=2; i++)
              mat[3-i][i+j] = b[i];
     }


         bsplval(t,4,basdim-1,xnode[n],b);
         for (i=0; i<=2; i++)
              mat[3-i][i+n] = b[i+1];


 /* Fill y according to the value of 'extra'.
  * For y we use bcoef.
  */
 if (extra == BUILT)
    {    bcoef[0] = phiacc0;
         bcoef[basdim-1] = phiacc1;
    }
 else
    {    bcoef[0] = 0;
         bcoef[basdim-1] = 0;
    }

 for ( i=1; i<basdim-1; i++)
         bcoef[i] = ynode[i-1];

 /* Solve the system: */

 decband(mat,basdim,2,1);
 solband(mat,basdim,2,1,bcoef);
}
```

```
double
splval(knot,order,basdim,bcoef,x)
        double knot[], bcoef[], x;
           int order, basdim;
        /* The procedure calculates the value of a spline in the
         * point x, given its B-representation.
         * This work can also be done by 'splderiv' with der=0.
         */
{
        double b[MAXORDER], val;
           int left,lastleft,flag,i;

        val = 0;

        interval(knot,basdim+order-1,&left,x,&flag,&lastleft);
        /* If x=knot[basdim], take left = basdim-1: */
        left=min(left,basdim-1);

        bsplval(knot,order,left,x,b);

        for (i=1; i<=order; i++)
             val += bcoef[left-order+i] * b[i-1];
        return(val);
}
```

```
/* Example 1:
 * We interpolate 5 points of the B-spline M[4][3], with
 * endconditions: free ends and built in ends.
 * First we determine the B-representation of the interpolating
 * spline, and then calculate at some points the value of
 * the spline via 'splval'.
 * We see from the comment in 'brep', that we have to take
 * basdim = 7, and declare 5+6 knots.
 */


main()
{
        double xnode[5], ynode[5], bcoef[7], knot[11], x, splval();
           int i, j, basdim, left, lastleft, flag;

        basdim = 7;

        xnode[0] = -1.0;  ynode[0] = 0.0;
        xnode[1] = -0.5;  ynode[1] = 0.083333;
        xnode[2] =  0.0;  ynode[2] = 0.333333;
        xnode[3] =  0.5;  ynode[3] = 0.083333;
        xnode[4] =  1.0;  ynode[4] = 0.0;


        for (j=0; j<=1; j++)
        {    /*  extra=j=0: free ends, extra=j=1: built in ends. */
                 brep(xnode,ynode,basdim,knot,bcoef,j,0.0,0.0);
                 for (i=0; i<7; i++)
                         printf("bcoef[%d] = %f\n", i, bcoef[i]);
                 x = -1.0;
                 for (i=0; i<=20; i++)
             {   printf("x=%f value=%f\n", x,splval(knot,4,basdim,bcoef,x));
                 x += 0.1;
             }
                 printf("\n\n");
        }
}
```

RESULTS:

```
Free ends:                                 Built in ends:

bcoef[0] = 0.000001                        bcoef[0] = -0.000001
bcoef[1] = 0.000000                        bcoef[1] = 0.000001
bcoef[2] = -0.000001                       bcoef[2] = -0.000001
bcoef[3] = 0.999999                        bcoef[3] = 0.999999
bcoef[4] = -0.000001                       bcoef[4] = -0.000001
bcoef[5] = 0.000000                        bcoef[5] = 0.000001
bcoef[6] = 0.000001                        bcoef[6] = -0.000000
x=-1.000000 value=0.000000                 x=-1.000000 value=0.000000
x=-0.900000 value=0.000667                 x=-0.900000 value=0.000667
x=-0.800000 value=0.005333                 x=-0.800000 value=0.005333
x=-0.700000 value=0.018000                 x=-0.700000 value=0.018000
x=-0.600000 value=0.042666                 x=-0.600000 value=0.042666
x=-0.500000 value=0.083333                 x=-0.500000 value=0.083333
x=-0.400000 value=0.141333                 x=-0.400000 value=0.141333
x=-0.300000 value=0.207333                 x=-0.300000 value=0.207333
x=-0.200000 value=0.269333                 x=-0.200000 value=0.269333
x=-0.100000 value=0.315333                 x=-0.100000 value=0.315333
x=0.000000 value=0.333333                  x=0.000000 value=0.333333
x=0.100000 value=0.315333                  x=0.100000 value=0.315333
x=0.200000 value=0.269333                  x=0.200000 value=0.269333
x=0.300000 value=0.207333                  x=0.300000 value=0.207333
x=0.400000 value=0.141333                  x=0.400000 value=0.141333
x=0.500000 value=0.083333                  x=0.500000 value=0.083333
x=0.600000 value=0.042666                  x=0.600000 value=0.042666
x=0.700000 value=0.018000                  x=0.700000 value=0.018000
x=0.800000 value=0.005333                  x=0.800000 value=0.005333
x=0.900000 value=0.000667                  x=0.900000 value=0.000667
x=1.000000 value=0.000000                  x=1.000000 value=0.000000
```

```
/* Example 2:
 * We now interpolate 7 points of the B-spline M[4][3], with
 * endcondition ´not a knot´.
 * The second and sixth node are not taken as knots, but
 * merely as interpolation points. We see from the comment
 * in ´brep´ that we have to take basdim = 7, and declare
 * 7+4 = 11 knots.
 */

main()
{
        double xnode[7], ynode[7], bcoef[7],knot[11], splval(), x;
           int i, basdim;

        basdim = 7;

        xnode[0] = -1.0;  ynode[0] = 0.0;
        xnode[1] = -0.8;  ynode[1] = 0.005333;
        xnode[2] = -0.5;  ynode[2] = 0.083333;
        xnode[3] =  0.0;  ynode[3] = 0.333333;
        xnode[4] =  0.5;  ynode[4] = 0.083333;
        xnode[5] =  0.8;  ynode[5] = 0.005333;
        xnode[6] =  1.0;  ynode[6] = 0.0;


        brep(xnode,ynode,basdim,knot,bcoef,-1,0.0,0.0);


        for (i=0; i<7; i++)
                printf("bcoef[%d] = %f\n", i, bcoef[i]);
        x = -1.0;
        for (i=0; i<=20; i++)
           {
                printf("x=%f splval=%f\n", x,splval(knot,4,basdim,bcoef,x));
                x += 0.1;
            }
}
```

```
                        RESULTS:

                        bcoef[0] = 0.000006
                        bcoef[1] = -0.000001
                        bcoef[2] = -0.000001
                        bcoef[3] = 0.999999
                        bcoef[4] = -0.000001
                        bcoef[5] = -0.000001
                        bcoef[6] = 0.000006
                        x=-1.000000 splval=0.000000
                        x=-0.900000 splval=0.000666
                        x=-0.800000 splval=0.005333
                        x=-0.700000 splval=0.018000
                        x=-0.600000 splval=0.042666
                        x=-0.500000 splval=0.083333
                        x=-0.400000 splval=0.141333
                        x=-0.300000 splval=0.207333
                        x=-0.200000 splval=0.269333
                        x=-0.100000 splval=0.315333
                        x=0.000000 splval=0.333333
                        x=0.100000 splval=0.315333
                        x=0.200000 splval=0.269333
                        x=0.300000 splval=0.207333
                        x=0.400000 splval=0.141333
                        x=0.500000 splval=0.083333
                        x=0.600000 splval=0.042666
                        x=0.700000 splval=0.018000
                        x=0.800000 splval=0.005333
                        x=0.900000 splval=0.000666
                        x=1.000000 splval=0.000000
```

```
bstopp(knot,order,basdim,bcoef,breaks,numpol,pcoef)
        double knot[], bcoef[], breaks[], pcoef[][MAXORDER];
           int basdim, *numpol, order;
        /* The procedure converts the B-representation
         * (knot,order,basdim,bcoef) of a spline into its
         * PP-representation (breaks,order,numpol,pcoef).
         * The breakpoints are located, and then for each
         * breakpoint interval, the 'order' relevant B-coefficients
         * are differenced according to formula (5.5.6), whereafter the
         * first 'order-1' derivatives of the spline are calculated.
         */
{       double dr,dl,alpha[MAXORDER][MAXBASDIM], val,b[MAXORDER];
           int i, j, left, count;
        val = 0;
        count = 0;
        for (left = order-1; left < basdim; left++)
            {   /* Find succesive breakpoints. */

                if (knot[left+1] != knot[left])

               {breaks[count] = knot[left];
                /* Now construct table of coefficients that decide
                 * the sum of (5.5.6): for j=0,1,...,order-1,
                 * calculate the column alpha[j][i].
                 */
                /* j=0: */
                        for (i=left-order+1; i <= left; i++)
                                alpha[0][i] = bcoef[i];
                for (j = 1; j <= order-1; j++)
                    {   for (i=left-order+1+j; i <= left; i++)
                            {   dr = knot[i+order+1-j] - knot[i];
                                dl = knot[i+order  -j] - knot[i-1];

        alpha[j][i] = (order-j)*(alpha[j-1][i]/dr - alpha[j-1][i-1]/dl);
                            }
                    }

                /* Now calculate the value of the spline, and its first
                 * (order-1) derivatives in breaks[count]:
                 */
                for (j = 0; j <= order-1; j++)
                    {   bsplval(knot,order-j,left,breaks[count],b);
                        for (i = left-order+1+j; i <= left; i++)

                        val += alpha[j][i] * b[i-left+order-1-j];

                        pcoef[count][j] = val;
                        val = 0;
                    }
                count++;
               }
            }
        if (knot[basdim-1] != knot[basdim])
            breaks[count] = knot[basdim];
        *numpol = count;
}
```

```
/* Example 3:
 * We convert the B-representation of M4,3 into its
 * PP-representation, and then calculate M4,3 by means of
 * 'ppval'.
 */

main()
{        double t[11],bcoef[7],b[11],pcoef[4][MAXORDER],ppval(),x1,dx;
            int i,j,numpol;

         /* Construct the B-representation: */
         t[0]=t[1]=t[2]=t[3]= -1.0;
         t[4]= -0.5;
         t[5]=  0.0;
         t[6]=  0.5;
         t[7]=t[8]=t[9]=t[10]= 1.0;

         /* Basdim = 7, so b[0],...,b[6]. */

         bcoef[0]=bcoef[1]=bcoef[2]= 0.0;
         bcoef[3]= 1.0;
         bcoef[4]=bcoef[5]=bcoef[6]= 0.0;

         /* Construct the pp-representation via 'bstopp': */
         bstopp(t,4,7,bcoef,b,&numpol,pcoef);
         /* Now M4,3 in PP-rep. (b,4,numpol,pcoef) */

         for (i=0; i<numpol; i++)
                 {
                 for (j=0; j<4; j++)
                 printf("pcoef[%d][%d] = %f\n", i,j, pcoef[i][j]);
                 printf("break[%d] = %f\n\n", i, b[i]);
                 }
         printf("break[%d] = %f\n\n", numpol,b[numpol]);
         x1 = -1.0;
         dx = 0.1;

         for (i=0; i<=20; i++)
         {
           printf("x1=%f val=%f\n", x1, ppval(b,4,numpol,pcoef,x1,0) );
           x1 += dx;
         }
}
```

RESULTS:

```
pcoef[0][0] = 0.000000
pcoef[0][1] = 0.000000
pcoef[0][2] = 0.000000
pcoef[0][3] = 4.000000
break[0] = -1.000000

pcoef[1][0] = 0.083333
pcoef[1][1] = 0.500000
pcoef[1][2] = 2.000000
pcoef[1][3] = -12.000000
break[1] = -0.500000

pcoef[2][0] = 0.333333
pcoef[2][1] = 0.000000
pcoef[2][2] = -4.000000
pcoef[2][3] = 12.000000
break[2] = 0.000000

pcoef[3][0] = 0.083333
pcoef[3][1] = -0.500000
pcoef[3][2] = 2.000000
pcoef[3][3] = -4.000000
break[3] = 0.500000

break[4] = 1.000000
```

```
x1=-1.000000 val=0.000000
x1=-0.900000 val=0.000667
x1=-0.800000 val=0.005333
x1=-0.700000 val=0.018000
x1=-0.600000 val=0.042667
x1=-0.500000 val=0.083333
x1=-0.400000 val=0.141333
x1=-0.300000 val=0.207333
x1=-0.200000 val=0.269333
x1=-0.100000 val=0.315333
x1=0.000000 val=0.333333
x1=0.100000 val=0.315333
x1=0.200000 val=0.269333
x1=0.300000 val=0.207333
x1=0.400000 val=0.141333
x1=0.500000 val=0.083333
x1=0.600000 val=0.042667
x1=0.700000 val=0.018000
x1=0.800000 val=0.005333
x1=0.900000 val=0.000667
x1=1.000000 val=0.000000
/* See also the results of example 1 of section 4.5 */
```

# 6. A NEW ALGORITHM FOR B-SPLINE INTERPOLATION

## 6.1. Introduction

So far, we have seen two different representations of an interpolating curve: the pp-representation where the polynomials are given in terms of the right derivatives at the break points, and the B-representation where the polynomials are given as a linear combination of B-splines.

Anticipating hardware facilities that, given their coefficients, perform the plotting of polynomials, we seek a new representation for our interpolating curve, in terms of its composing polynomials. We shall derive a polynomial representation for a B-spline, and with this, arrive at an algorithm which enables us to express the interpolating function as a composition of polynomials of which we know the coefficients.

## 6.2. B-spline Polynomials

Consider the B-spline $M_{4,0}$ on its knots $x_0,x_1,\dots,x_4$, and call the polynomials that make up this B-spline

$p_{-1}$ on $(-\infty,x_0]$,
$p_0$ on $[x_0,x_1]$,
$p_1$ on $[x_1,x_2]$,
$p_2$ on $[x_2,x_3]$,
$p_3$ on $[x_3,x_4]$, and
$p_4$ on $[x_4,\infty)$.

Now recall the definition of a B-spline (def. 4.2.2), then:

$$M_{4,0}(x) = \sum_{j=0}^{4} \frac{(x_j-x)_+^{n-1}}{\omega_0'(x_j)}$$

where

$$\omega_0'(x_j) = \prod_{\substack{\kappa=0 \\ \kappa\neq j}}^{4} (x_\kappa-x_j)$$

We then have

$$
\begin{aligned}
p_4(x) &= 0\\
p_3(x) &= \frac{(x_4-x)^3}{\omega'(x_4)}\\
p_2(x) &= \frac{(x_4-x)^3}{\omega'(x_4)} + \frac{(x_3-x)^3}{\omega'(x_3)}\\
p_1(x) &= \frac{(x_4-x)^3}{\omega'(x_4)} + \frac{(x_3-x)^3}{\omega'(x_3)} + \frac{(x_2-x)^3}{\omega'(x_2)}\\
p_0(x) &= \frac{(x_4-x)^3}{\omega'(x_4)} + \frac{(x_3-x)^3}{\omega'(x_3)} + \frac{(x_2-x)^3}{\omega'(x_2)} + \frac{(x_1-x)^3}{\omega'(x_1)}\\
p_{-1}(x) &= \frac{(x_4-x)^3}{\omega'(x_4)} + \frac{(x_3-x)^3}{\omega'(x_3)} + \frac{(x_2-x)^3}{\omega'(x_2)} + \frac{(x_1-x)^3}{\omega'(x_1)} + \frac{(x_0-x)^3}{\omega'(x_0)}\\
&= 0.
\end{aligned}
$$

In general, if we write

$$
\begin{aligned}
M_i = M_{4,i} &= p_{-1}^i \quad \text{on} \quad (-\infty, x_i]\\
&= p_0^i \quad \text{on} \quad [x_i, x_{i+1}]\\
&= p_1^i \quad \text{on} \quad [x_{i+1}, x_{i+2}]\\
&= p_2^i \quad \text{on} \quad [x_{i+2}, x_{i+3}]\\
&= p_3^i \quad \text{on} \quad [x_{i+3}, x_{i+4}]\\
&= p_4^i \quad \text{on} \quad [x_{i+4}, \infty)
\end{aligned}
$$

and

$$\Delta p_j^i = p_j^i - p_{j-1}^i , \qquad j = 0,1,\ldots,4$$

and

$$\omega_i'(x_j) = \prod_{\substack{\kappa=i \\ \kappa\neq j}}^{4+i} (x_\kappa - x_j)$$

then we get:

(6.2.1)
$$\Delta p_0^i = \frac{(x-x_i)^3}{\omega_i'(x_i)}$$

$$\Delta p_1^i = \frac{(x-x_{i+1})^3}{\omega_i'(x_{i+1})}$$

$$\Delta p_2^i = \frac{(x-x_{i+2})^3}{\omega_i'(x_{i+2})}$$

$$\Delta p_3^i = \frac{(x-x_{i+3})^3}{\omega_i'(x_{i+3})}$$

$$\Delta p_4^i = \frac{(x-x_{i+4})^3}{\omega'(x_{i+4})}$$

## 6.3. B-spline Interpolation

For a given knotset $x_0, x_1, \ldots, x_n$ the polynomials on the spans i+1 and i+2 are given as

$$pp_i = c_{i-3}\cdot M_{i-3} + c_{i-2}\cdot M_{i-2} + c_{i-1}\cdot M_{i-1} + c_i\cdot M_i$$

$$pp_{i+1} = c_{i-2}\cdot \bar{M}_{i-2} + c_{i-1}\cdot \bar{M}_{i-1} + c_i\cdot \bar{M}_i + c_{i+1}\cdot \bar{M}_{i+1}$$

where

$$M_{i-\kappa} = p_\kappa^{i-\kappa} \quad \text{on} \quad [x_i, x_{i+1}] \quad \text{for} \quad \kappa = 0,1,2,3$$

and

$$\bar{M}_{i-\kappa} = p_{\kappa+1}^{i-\kappa} \quad \text{on} \quad [x_{i+1}, x_{i+2}] \quad \text{for} \quad \kappa = -1,0,1,2.$$

We then have, for $x \in [x_{i+1}, x_{i+2}]$:

$$\begin{aligned} pp_{i+1} - pp_i = \; & c_{i-3} \cdot (0 - M_{i-3}) \\ & + c_{i-2} \cdot (\bar{M}_{i-2} - M_{i-2}) \\ & + c_{i-1} \cdot (\bar{M}_{i-1} - M_{i-1}) \\ & + c_i \cdot (\bar{M}_i - M_i) \\ & + c_{i+1} \cdot (\bar{M}_{i+1} - 0) \\ = \; & c_{i-3} \cdot (p_4^{i-3} - p_3^{i-3}) \\ & + c_{i-2} \cdot (p_3^{i-2} - p_2^{i-2}) \\ & + c_{i-1} \cdot (p_2^{i-1} - p_1^{i-1}) \\ & + c_i \cdot (p_1^i - p_0^i) \\ & + c_{i+1} \cdot (p_0^{i+1} - p_{-1}^{i+1}) \end{aligned}$$

because $p_4^{i-3} \equiv 0$ and $p_{-1}^{i+1} \equiv 0$ on $[x_{i+1}, x_{i+2}]$. This gives

$$\begin{aligned} pp_{i+1} - pp_i &= c_{i-3} \cdot \Delta p_4^{i-3} + c_{i-2} \cdot \Delta p_3^{i-2} + c_{i-1} \cdot \Delta p_2^{i-1} + c_i \cdot \Delta p_1^i \\ &\qquad + c_{i+1} \cdot \Delta p_0^{i+1} \\ &= c_{i-3} \cdot \frac{(x-x_{i+1})^3}{\omega'_{i-3}(x_{i+1})} + c_{i-2} \cdot \frac{(x-x_{i+1})^3}{\omega'_{i-2}(x_{i+1})} + \\ &\qquad c_{i-1} \cdot \frac{(x-x_{i+1})^3}{\omega'_{i-1}(x_{i+1})} + c_i \cdot \frac{(x-x_{i+1})^3}{\omega'_i(x_{i+1})} + c_{i+1} \cdot \frac{(x-x_{i+1})^3}{\omega'_{i+1}(x_{i+1})} \end{aligned}$$

$$(6.3.1) \qquad \begin{aligned} pp_{i+1} - pp_i &= \left\{ \frac{c_{i-3}}{\omega'_{i-3}(x_{i+1})} + \frac{c_{i-2}}{\omega'_{i-2}(x_{i+1})} + \frac{c_{i-1}}{\omega'_{i-1}(x_{i+1})} + \frac{c_i}{\omega'_i(x_{i+1})} + \right. \\ &\qquad \left. + \frac{c_{i+1}}{\omega'_{i+1}(x_{i+1})} \right\} \cdot (x-x_{i+1})^3 \\ &= A_i \cdot (x-x_{i+1})^3 . \end{aligned}$$

We now have found a way to compute $pp_{i+1}$ from $pp_i$ , and can determine the interpolating curve by taking a 'starting' polynomial on a certain span, and use (6.3.1) to calculate the next polynomial! An algorithm for this interpolation now proceeds as follows:

(1) Calculate the coefficients $c_{i-3},\ldots,c_{i+1}$;

(2) Calculate for a certain i (e.g. i = 0) all right derivatives of the interpolating function from the B-spline values:

$$y_i := pp_i(x_i)$$

$$y_i' := pp_i'(x_i)$$

$$y_i'' := pp_i''(x_i)$$

$$y_i''' := pp_i'''(x_i)$$

Then the starting pp equals:

$$pp_i(x) = y_i + (x-x_i)\cdot y' + \frac{(x-x_i)^2}{2}\cdot y_i'' + \frac{(x-x_i)^3}{6}\cdot y_i'''$$

$$= \frac{y_i'''}{6}\cdot x^3 + \left(\frac{y_i''}{2} - \frac{y_i'''\cdot x_i}{2}\right)\cdot x^2 + \left(y_i' - x_i y_i'' + \frac{x_i^2\cdot y_i'''}{2}\right)\cdot x +$$

$$+ y_i - x_i\cdot y_i' + \frac{x_i^2\cdot y_i''}{2} - \frac{x_i^3\cdot y_i'''}{6}\ .$$

If we put

$$pp_i(x) = a_i x^3 + b_i x^2 + c_i x + d_i$$

this gives:

(6.3.2)
$$a_i = \frac{y_i'''}{6}$$
$$b_i = \frac{y_i''}{2} - \frac{y_i''' \cdot x_i}{2}$$
$$c_i = y_i' - y_i'' \cdot x_i + \frac{y_i''' \cdot x_i^2}{2}$$
$$d_i = y_i - y_i' \, x_i + \frac{y_i'' \cdot x_i^2}{2} - \frac{y_i''' \cdot x_i^3}{6}$$

(3) Calculate $A_i$ from (6.3.1).

(4) Given the coefficients $a_i$, $b_i$, $c_i$ and $d_i$, it follows:

(6.3.3)
$$a_{i+1} = a_i + A_i$$
$$b_{i+1} = b_i - 3 \cdot x_{i+1} \cdot A_i$$
$$c_{i+1} = c_i + 3 \cdot x_{i+1}^2 \cdot A_i$$
$$d_{i+1} = d_i - x_{i+1}^3 \cdot A_i$$

and also:

$$a_{i-1} = a_i - A_{i-1}$$
$$b_{i-1} = b_i + 3 \cdot x_i \cdot A_{i-1}$$
$$c_{i-1} = c_i - 3 \cdot x_i^2 \cdot A_{i-1}$$
$$d_{i-1} = d_i + x_i^3 \cdot A_{i-1}$$

Hence, given a starter according to (6.3.2), we can find the pp coefficients on the adjacent spans and so on!

Numerical stability can be checked by comparing the actual pp-value at a knot with the given value. If the difference gets too big, one can calculate a new starter!

(6.3.2) is performed in the following procedure 'starter', which is written for order 4.

```
starter(t,basdim,bcoef,left,coef)
        double t[],bcoef[],coef[];
           int basdim,left;
        /* The procedure calculates the polynomial
         * coefficients  on the interval
         * t[left+3],t[left+4], of the spline
         * given in its B-representation.
         * coef[3] is the leading coefficient!
         * For the calculations we use 6.3.2
         */
{
        double y[4],x,splderiv();
           int i;

        x = t[left+3];
        for (i=0; i<4; i++)
             y[i] = splderiv(t,4,basdim,bcoef,x,i);
             /* y[i] now holds the i-th derivative of
              * the spline in xnode[left].
              */

        coef[3] = y[3] / 6;
        coef[2] = (y[2] - y[3]*x)/2;
        coef[1] = y[1] - y[2]*x + y[3]*x*x/2;
        coef[0] = y[0] - y[1]*x + y[2]*x*x/2 - y[3]*x*x*x/6;
}
```

The calculations of $A_i$, and of the new coefficients, are performed in the procedure 'nextpol', also written for the order 4.

```
nextpol(t,basdim,bcoef,left,coef)
        double t[],bcoef[],coef[];
           int basdim,left;
        /* The procedure calculates, given the coefficients
         * of the ´preceding´ polynomial and the B-rep, the coefficients
         * of the next polynomial, using 6.3.3  and
         * 6.3.1, on the interval t[left+3],t[left+4], where left >=1.
         * It assumes the knots t[3],...,t[basdim] to be distinct.
         */
{
        double alpha[5],A,ti;
           int i,j;

        i = 3 + left;
        ti = t[i];

        alpha[0] = (t[i-4]-ti)*(t[i-3]-ti)*(t[i-2]-ti)*(t[i-1]-ti);
        alpha[1] = (t[i-3]-ti)*(t[i-2]-ti)*(t[i-1]-ti)*(t[i+1]-ti);
        alpha[2] = (t[i-2]-ti)*(t[i-1]-ti)*(t[i+1]-ti)*(t[i+2]-ti);
        alpha[3] = (t[i-1]-ti)*(t[i+1]-ti)*(t[i+2]-ti)*(t[i+3]-ti);
        alpha[4] = (t[i+1]-ti)*(t[i+2]-ti)*(t[i+3]-ti)*(t[i+4]-ti);

        A = 0;
        for (j=0; j<5; j++)
                A += bcoef[i-4+j] / alpha[j];

        coef[3] += A;
        coef[2] -= 3*ti*A;
        coef[1] += 3*ti*ti*A;
        coef[0] -= ti*ti*ti*A;
}
```

## 6.4. Measurements

Now the different methods of approach to B-spline interpolation will be given and clocked for order 4, and the newley developed method of section 6.3 will be judged only on speed.

Given a set of nodes and corresponding data, the B-representation of the interpolating spline is determined, and taken as input by three procedures:

'splineval': calculates the splinevalues directly via 'bsplval'.

'splinecon': converts the B-representation to PP-representation and then uses the algorithm of 'ppval' for the calculation of the splinevalues.

'splinepol': determines the coefficients of the 'first' polynomial via 'starter', and successively those of the following polynomials via 'nextpol'.
It then calculates the splinevalues by 'polval':

```
splinecon(xnode,ynode,knot,basdim,bcoef,delta)
        double xnode[],ynode[],knot[],bcoef[],delta;
           int basdim;
        /* The procedure converts the B-representation of
         * the spline to the PP-representation, and then
         * calculates and prints the values of the spline with the
         * algorithm of ´ppval´.
         * It assumes the nodes 0,1,... to coincide with
         * the knots 3,4,...,basdim-1.
         * Cf. ´bstopp´
         */
{
        double dr,dl,alpha[4][MAXBASDIM],val,b[4],x,pcoef[4],del;
           int i,j,left;

        for (left=3; left<basdim; left++)
        {
        /* Construct the table of coefficients that decide
         * the sum of (5.5.6).
         */
                for (i=left-3; i<=left; i++)
                     alpha[0][i] = bcoef[i];
           for (j=1; j<=3; j++)
           {    for (i=left-3+j; i<=left; i++)
                {    dr = knot[i+5-j] - knot[i];
                     dl = knot[i+4-j] - knot[i-1];
        alpha[j][i] = (4-j)*(alpha[j-1][i]/dr - alpha[j-1][i-1]/dl);
                }
           }
```

```
      /* Now calculate the coefficients: */
      for (j=0; j<=3; j++)
      {    bsplval(knot,4-j,left,knot[left],b);
           val = 0;
           for (i=left-3+j; i<=left; i++)
                   val += alpha[j][i] * b[i-left+3-j];
           pcoef[j] = val;
      }

      /* Now calculate the spline values (cf. 'ppval'): */
      for (x=knot[left]; x<knot[left+1]; x += delta)
          {del = x - knot[left];
           val = 0;
           for (j=3; j>=0; j--)
                val = (val*del)/(j+1) + pcoef[j];
           printf("x=%f  splinecon=%f\n", x,val);
          }
   }
   del = knot[basdim] - knot[basdim-1];
   val = 0;
   for (j=3; j>=0; j--)
           val = (val*del)/(j+1) + pcoef[j];
   printf("x=%f  splinecon=%f\n",knot[basdim],val);
}



splineval(xnode,ynode,knot, basdim,bcoef,delta)
        double xnode[],ynode[],knot[],bcoef[],delta;
           int basdim;
        /* The procedure calculates and prints the values of the spline
         * from the B-representation.
         * Cf. 'splderiv' with der=0.
         * It assumes the nodes 0,1,... to coincide with
         * the knots 3,4,...,basdim-1.
         */
{
        double b[MAXORDER],val,x;
           int left,i;


        for (left=3; left<basdim; left++)
        {
                for (x=knot[left]; x<knot[left+1]; x += delta)
                {
                        bsplval(knot,4,left,x,b);
                        val = 0;
                        for (i=0; i<4; i++)
                             val += bcoef[left-3+i]*b[i];

                        printf("x=%f  splineval=%f\n", x,val);
                }
        }
        bsplval(knot,4,basdim-1,knot[basdim],b);
        val = 0;
        for (i=0; i<4; i++)
                val += bcoef[basdim-4+i]*b[i];
        printf("x=%f  splineval=%f\n",knot[basdim],val);
}
```

```
splinepol(xnode,ynode,knot,basdim,bcoef,eps,delta)
        double xnode[],ynode[],knot[],bcoef[],eps,delta;
           int basdim;
        /* The procedure calculates and prints the spline values
         * by determining the first polynomial,
         * and then succesively determining the
         * coefficients of the following polynomials
         * as suggested in 6.3
         * For every call to 'nextpol', the splinevalue in the
         * right endpoint of the considered interval is
         * compared with the given y-value
         * in that point, and if the difference is bigger
         * than the by eps prescribed value, a new starting
         * polynomial for that  interval is
         * determined by 'starter'.
         * The procedure assumes the nodes 0,1,... to coincide with
         * the knots 3,4,...,basdim-1.
         */
{
        double co[4],coef[4],x,y,abs(),polval(),splval();
           int left,i;

        /* Calculate the coefficients of the first polynomial */

        starter(knot,basdim,bcoef,0,coef);
        for (x=knot[3]; x<knot[4]; x += delta)
             printf("x=%f  splinepol=%f\n", x,polval(coef,x));

        /* Calculate the coefficients of the succeeding polynomials */
        for (left=1; left<basdim-3; left++)
        {
                /* In case a new starting polynomial has to be
                 * calculated, we have to have disposal of the
                 * coefficients of the preceding polynomial.
                 */
                for (i=0; i<4; i++)
                     co[i] = coef[i];

                nextpol(knot,basdim,bcoef,left,coef);

                /* Calculate value in right endpoint: */
                y = polval(coef,xnode[left+1]);

                /* Compare with given value: */
                if ( abs(y-ynode[left+1]) >= eps )
                   {
                        printf("\n Determination of a new starting");
                        printf(" polynomial for \n");
                        printf(" interval x[%d],x[%d]. \n\n",left,left+1);

                        for (i=0; i<4; i++)
                             coef[i] = co[i];

                        starter(knot,basdim,bcoef,left,coef);
                   }


                for (x=xnode[left]; x<xnode[left+1]; x += delta)
                     printf("x=%f  splinepol=%f\n", x,polval(coef,x));
        }
        printf("x=%f  splinepol=%f\n", xnode[basdim-3], y);
}
```

In the following example, we compare the different methods of calculating the splinevalues.

All three methods use the B-representation supplied by 'brep', and are clocked.

'times' is a system procedure which amongst others returns the number of 1/50 seconds that the user process has run.

```
main()
{
        double x[I+1],y[I+1],bcoef[I+3],knot[I+7];
        int i;
        struct tbuffer {
                long pu;
                long ps;
                long cu;
                long cs;
                       };
        struct tbuffer buffer;
        /* buffer needed for system call ´times´
         * Time is counted in 1/50 seconds.
         */
        /* On the file ´file´ 200 random doubles
         * are available, and fetched with ´scanf´
         */
        for (i=0; i<=I; i++)
        {       scanf("%f",&x[i]);
                x[i]=x[i]*10;

                scanf("%f",&y[i]);
                y[i]=y[i]*2;
        }

        /* Get the nodes in strictly increasing order: */
        sort(x,I);
        for (i=0; i<=I; i++)
        printf("x[%d]=%f   y[%d]=%f\n", i,x[i],i,y[i]);

        /* Compute the B-coefficients, and clock
         * the process.
         */
        times(&buffer);
        printf("\n time in brep:%D\n", buffer.pu);
        brep(x,y,I+3,knot,bcoef,FREE,0.0,0.0);
        times(&buffer);
        printf(" time out brep: %D\n", buffer.pu);
        /* Now compare the different methods of calculation: */
        times(&buffer);
        printf("\n time in splineval: %D\n", buffer.pu);
        splineval(x,y,knot,I+3,bcoef,0.5);
        times(&buffer);
        printf(" time out splineval: %D\n", buffer.pu);

        times(&buffer);
        printf("\n time in splinecon: %D\n", buffer.pu);
        splinecon(x,y,knot,I+3,bcoef,0.5);
        times(&buffer);
        printf(" time out splinecon: %D\n", buffer.pu);

        times(&buffer);
        printf("\n time in splinepol: %D\n", buffer.pu);
        splinepol(x,y,knot,I+3,bcoef,0.000001,0.5);
        times(&buffer);
        printf("time out splinepol: %D\n", buffer.pu);
}
```

```
RESULTS for I=5:


x[0]=4.217110    y[0]=1.785398
x[1]=5.000000    y[1]=1.594526
x[2]=6.891550    y[2]=0.829098
x[3]=7.117930    y[3]=1.569692
x[4]=8.308720    y[4]=0.004216
x[5]=9.877540    y[5]=0.553642

 time in brep:3
 time out brep: 7

 time in splineval: 7
x=4.217110  splineval=1.785398
x=4.717110  splineval=1.766887
x=5.000000  splineval=1.594526
x=5.500000  splineval=0.896056
x=6.000000  splineval=0.171771
x=6.500000  splineval=0.064967
x=6.891550  splineval=0.829098
x=7.117930  splineval=1.569692
x=7.617930  splineval=1.639553
x=8.117930  splineval=0.429318
x=8.308720  splineval=0.004216
x=8.808720  splineval=-0.408532
x=9.308720  splineval=-0.095311
x=9.808720  splineval=0.479855
x=9.877540  splineval=0.553642
 time out splineval: 14
 time in splinecon: 14
x=4.217110  splinecon=1.785398
x=4.717110  splinecon=1.766887
x=5.000000  splinecon=1.594526
x=5.500000  splinecon=0.896056
x=6.000000  splinecon=0.171771
x=6.500000  splinecon=0.064967
x=6.891550  splinecon=0.829098
x=7.117930  splinecon=1.569692
x=7.617930  splinecon=1.639553
x=8.117930  splinecon=0.429318
x=8.308720  splinecon=0.004216
x=8.808720  splinecon=-0.408532
x=9.308720  splinecon=-0.095311
x=9.808720  splinecon=0.479855
x=9.877540  splinecon=0.553642
 time out splinecon: 20

 time in splinepol: 20
x=4.217110  splinepol=1.785398
x=4.717110  splinepol=1.766887
x=5.000000  splinepol=1.594526
x=5.500000  splinepol=0.896056
x=6.000000  splinepol=0.171771
x=6.500000  splinepol=0.064967
x=6.891550  splinepol=0.829098
x=7.117930  splinepol=1.569692
x=7.617930  splinepol=1.639553
x=8.117930  splinepol=0.429318
x=8.308720  splinepol=0.004216
x=8.808720  splinepol=-0.408532
x=9.308720  splinepol=-0.095311
x=9.808720  splinepol=0.479855
x=9.877540  splinepol=0.553642
time out splinepol: 26
```

If we take the steplength = 0.01 instead of 0.5 in the procedures 'splineval', 'splinecon' and 'splinepol', we get for I = number of nodes the following table:

| I | Time splineval | Time splinecon | Time splinepol |
|---|---|---|---|
| 5 | 200 | 180 | 180 |
| 10 | 350 | 260 | 230 |
| 50 | 450 | 320 | 300 |
| 100 | 474 | 350 | 330 |
| 500 | 580 | 630 | 500 |

```
In the last situation, where I=500, several times a new
starting polynomial is determined:
x[174]=4.007690    y[174]=1.908110
x[175]=4.044240    y[175]=1.105360
x[176]=4.078700    y[176]=1.445834
x[177]=4.104570    y[177]=0.435832
x[178]=4.122260    y[178]=0.939470
x[179]=4.123160    y[179]=0.050510
x[180]=4.159770    y[180]=1.114572
x=4.007690   splinepol=1.908120
x=4.044240   splinepol=1.105370
x=4.078700   splinepol=1.445844

 Determination of a new starting polynomial for
 interval x[177],x[178].

x=4.104570   splinepol=0.435832
x=4.122260   splinepol=0.939470
x=4.123160   splinepol=0.050510
x=4.159770   splinepol=1.114572
x=4.192670   splinepol=0.698628
```

Now if we assume that hardware facilities make it possible to plot polynomials directly from the coefficients, it is interesting to know how much of the total time of 'splinepol' is actually taken by the determination of the coefficients:

```
main()
{
        double coef[MAXPOINTS][4], co[4], x, y, abs(), polval(), splval();
        double xnode[I+1],ynode[I+1], bcoef[I+3], knot[I+7],delta,eps;
           int left,i,basdim;
        struct tbuffer {
                long pu;
                long ps;
                long cu;
                long cs;
                        };
        struct tbuffer buffer;



        for (i=0; i<=I; i++)
        {       scanf("%f",&xnode[i]);
                xnode[i] = xnode[i]*10;

                scanf("%f",&ynode[i]);
                ynode[i]=ynode[i]*2;
        }
        delta = 0.5;
        eps = 0.00001;

        sort(xnode,I);
        for (i=0; i<=I; i++)
        printf("xnode[%d]=%f ynode[%d]=%f\n",i,xnode[i],i,ynode[i]);

        brep(xnode,ynode,I+3,knot,bcoef,FREE,0.0,0.0);


        times(&buffer);
        printf(" time in starter/nextpol: %D\n", buffer.pu);
        starter(knot,I+3,bcoef,0,co);
        for (i=0; i<4; i++)
             coef[0][i]=co[i];

        for (left=1; left<I; left++)
            {
                nextpol(knot,I+3,bcoef,left,co);
                y = polval(co,xnode[left+1]);
                if (abs(y-ynode[left+1]) >= eps)
                    {
                        for (i=0; i<4; i++)
                             co[i]=coef[left-1][i];
                        starter(knot,I+3,bcoef,co);
                    }
                for (i=0; i<4; i++)
                     coef[left][i]=co[i];
            }

        times(&buffer);
        printf("time out starter/nextpol: %D\n", buffer.pu);



        times(&buffer);
        printf("time in polval: %D\n", buffer.pu);
        for (left=0; left<I; left++)
        {    for (x=xnode[left]; x<xnode[left+1]; x+=delta)
             {
                printf(" x=%f ", x);
                printf(" polval=%f\n", polval(coef[left],x));
             }
        }
        printf(" x=%f", xnode[I]);
        printf(" polval=%f\n", y);
        times(&buffer);
        printf("time out polval: %D\n", buffer.pu);
}
```

```
The clocked tmes are:
        time in starter/nextpol:  49
        time out starter/nextpol:  56

        time in polval:  56
        time out polval:  73
```

The determination of the coefficients thus takes approximately 1/3 of the total time.
Considering that the procedure 'splinepol' is at least as quick as the other procedures, it is probable that the new method is very efficient in case the work of 'polval' can be done by the hardware!

## REFERENCES

[1] AHLBERG, J.H., E.N. NILSON & J.L. WALSH, *The theory of splines and their applications,* Academic Press, 1967.

[2] BOOR, C. de, *A practical guide to splines,* Springer Verlag, 1978.

[3] BOOR, C. de, *On calculating with B-splines,* Journal of Approximation Theory 6, 50-62 (1972).

[4] BOOR, C. de, *Package for calculating with B-splines,* MRC Technical Summary Report nr. 1333, 1973, University of Wisconsin.

[5] COX, M.G., *The Numerical evaluation of B-splines,* National Physical Laboratory DNAC 4, 1971.

[6] CURRY, H.B. & I.J. SCHOENBERG, *On Polya Frequency Functions IV: The fundamental spline functions and their limits,* Journal d'Analyse Mathematique 17, 71-107 (1966).

[7] DAHLQUIST, G. & A. BJÖRK, *Numerical Methods,* Prentice Hall, 1974.

[8] DAVIS, P.J., *Interpolation and Approximation,* Blaisdell Publishing Company, 1963.

[9] FAUX, I.D. & H.J. PRATT, *Computational Geometry for Design and Manufacture,* Ellis Horwood, 1979.

[10] GREVILLE, T.N.E. (ed.), *Theory and Applications of Spline Functions,* Academic Press, 1969.

[11] KERNIGHAN, B.W. & D.M. RITCHIE, *The C programming language,* Prentice Hall, 1978.

[12] STEFFENSEN, J.F., *Interpolation,* Chelsea Publishing Company, 1950.

[13] SCHOENBERG, I.J. & A. WHITNEY, *On Polya Frequency Functions III:* Trans. Amer. Mathem. Soc. 74, 246-259 (1953).

[14] HAGEN, T. & P.J.W. ten HAGEN, P. KLINT & H. NOOT, *ILP Intermediate language for pictures,* IW 68/77 Mathematisch Centrum (1977).

## APPENDIX A. Divided differences and some related theorems.

Let f be a function, defined on the points $x_0, x_1, \ldots$ We shall call this set of points a *net*, and assume the points $x_i$ to be distinct. They need not be equidistant or monotonely ordered. We define *divided differences* recursively:

DEFINITION A1. The 1st, 2nd, ...rth divided differences of f(x) with respect to the arguments $x_0, x_1, \ldots,$ are defined by the following system of equations:

$$f[x_0,x_1] = \frac{f(x_0)-f(x_1)}{x_0-x_1}$$

$$f[x_0,x_1,x_2] = \frac{f[x_0,x_1]-f[x_1,x_2]}{x_0-x_2}$$

$$\vdots$$

$$f[x_0,x_1,\ldots,x_r] = \frac{f[x_0,x_1,\ldots,x_{r-1}]-f[x_1,x_2,\ldots,x_r]}{x_0-x_r} .$$

THEOREM A1.

$$f[x_0,x_1,\ldots,x_n] = \sum_{j=0}^{n} \frac{f(x_j)}{\omega_n'(x_j)}$$

*where*

$$\omega_n(x) = \prod_{r=0}^{n} (x-x_r) .$$

PROOF.

$$\omega_n(x) = \prod_{r=0}^{n} (x-x_r) ,$$

so,

$$\omega_n'(x) = \sum_{r=0}^{n} \prod_{\substack{i=0 \\ i\neq r}}^{n} (x-x_i) ,$$

and thus,

$$\omega_n'(x_j) = \prod_{\substack{i=0 \\ i\neq j}}^{n} (x_j-x_i) .$$

Induction on n:

$$n=1:\ f[x_0,x_1] = \frac{f(x_1)-f(x_0)}{x_1-x_0}$$

$$= \frac{f(x_1)}{x_1-x_0} + \frac{f(x_0)}{x_0-x_1} = \sum_{j=0}^{1} \frac{f(x_j)}{\omega_1'(x_j)} .$$

Assume that the expression holds for n, then

$$f[x_0,x_1,\ldots,x_{n+1}] = \frac{f[x_0,x_1,\ldots,x_n]-f[x_1,x_2,\ldots,x_{n+1}]}{x_0-x_{n+1}}$$

$$=\left\{\sum_{j=0}^{n} \frac{f(x_j)}{\prod\limits_{\substack{i\neq j\\ i=0}}^{n}(x_j-x_i)} - \sum_{j=1}^{n+1} \frac{f(x_j)}{\prod\limits_{\substack{i\neq j\\ i=1}}^{n+1}(x_j-x_i)}\right\}\cdot \frac{1}{x_0-x_{n+1}}$$

$$=\left\{\frac{f(x_0).(x_0-x_{n+1})}{\prod\limits_{i\neq 0}^{n+1}(x_0-x_i)} + \sum_{j=1}^{n} \frac{f(x_j).(x_j-x_{n+1})}{\prod\limits_{\substack{i\neq j\\ i=0}}^{n+1}(x_j-x_i)}\right.$$

$$\left. - \frac{f(x_{n+1}).(x_{n+1}-x_0)}{\prod\limits_{i\neq n+1}^{n+1}(x_{n+1}-x_i)} - \sum_{j=1}^{n} \frac{f(x_j).(x_j-x_0)}{\prod\limits_{\substack{i\neq j\\ i=0}}^{n+1}(x_j-x_i)}\right\}\cdot \frac{1}{x_0-x_{n+1}}$$

$$= \frac{f(x_0)}{\prod\limits_{i\neq 0}^{n+1}(x_0-x_i)} + \sum_{j=1}^{n} \frac{f(x_j)}{\prod\limits_{\substack{i\neq j\\ i=0}}^{n+1}(x_j-x_i)} + \frac{f(x_{n+1})}{\prod\limits_{i\neq n+1}^{n+1}(x_{n+1}-x_i)}$$

$$= \sum_{j=0}^{n+1} \frac{f(x_j)}{\prod\limits_{\substack{i=0\\ i\neq j}}^{n+1}(x_j-x_i)}$$

$$= \sum_{j=0}^{n+1} \frac{f(x_j)}{\omega_{n+1}'(x_j)} . \qquad \square$$

COROLLARY. *The* n*th* *divided difference* $f[x_0,x_1,\ldots,x_n]$ *is a symmetric function of its* n+1 *arguments.*

LEMMA A2. *Let* p(x) *be the polynomial*

$$p(x) = c_0+c_1x + \ldots + c_n.x^n, \qquad c_i \in \mathbb{R},\ n \in \mathbb{N}.$$

Let $k \in \mathbb{N}$, and $k \leq n$. Then

$$p[x_0,x_1,\ldots,x_k] = c_n.A_k(x_0)$$

where $A_k(x_0)$ is a *monic polynomial* in $x_0$ of degree $(n-k)$.

PROOF. Induction on k:

$$\begin{aligned}
k=1:\ p[x_0,x_1] &= \frac{p(x_0)-p(x_1)}{x_0-x_1} \\
&= \frac{(c_0+c_1x_0+\ldots+c_nx_0^n)-(c_0+c_1x_1+\ldots+c_n.x_1^n)}{x_0-x_1} \\
&= \frac{c_1(x_0-x_1)+\ldots+c_n.(x_0^n-x_1^n)}{x_0-x_1} \\
&= c_1+c_2.(x_0+x_1)+\ldots+c_n.A_1.(x_0)
\end{aligned}$$

where $A_1(x_0)$ a monic polynomial of degree $(n-1)$.
Assume that the lemma holds for a certain k, $k < n$, then

$$\begin{aligned}
p[x_0,x_1,\ldots,x_{k+1}] &= \frac{p[x_0,x_1,\ldots,x_k]-p[x_1,\ldots,x_{k+1}]}{x_0-x_{k+1}} \\
\text{(corollary th.A1)} &= \frac{p[x_0,x_1,\ldots,x_k]-p[x_{k+1},x_1,\ldots,x_k]}{x_0-x_{k+1}} \\
&= c_n.\frac{A_k(x_0)-A_k'(x_{k+1})}{x_0-x_{k+1}} \\
&= c_n.\frac{A_k(x_0)-A_k'(x_{k+1})}{x_0-x_{k+1}} \\
&= c_n.\frac{x_0^{n-k}+\ldots-x_{k+1}^{n-k}-\ldots}{x_0-x_{k+1}} \\
&= c_n.\frac{(x_0-x_{k+1}).(x_0^{n-k-1}+\ldots)}{x_0-x_{k+1}} \\
&= c_n.A_{k+1}(x_0)
\end{aligned}$$

where $A_{k+1}(x_0)$ is a monic polynomial in $x_0$ of degree $(n-(k+1))$. □

THEOREM A3. *Let* $p(x)$ *be a polynomial of degree* $n$. *Then the* $(n+1)^{th}$ *divided difference*

$$p[x_0,x_1,\ldots,x_{n+1}] = 0.$$

PROOF. By lemma A2 we have

$$\begin{aligned} p[x_0,x_1,\ldots,x_{n+1}] &= \frac{p[x_0,x_1,\ldots,x_n]-p[x_{n+1},x_1,\ldots,x_n]}{x_0-x_{n+1}} \\ &= \frac{c_n-c_n}{x_0-x_{n+1}} \\ &= 0. \quad \square \end{aligned}$$

THEOREM A4. *Let* $f,g$ *and* $h$ *be functions, defined on the net* $\{x_i\}_{i=0}^{k}$, *with* $h(x) = f(x).g(x)$. *Write* $f[x] = f(x)$, $g[x] = g(x)$. *Then*

$$h[x_0,x_1,\ldots,x_k] = \sum_{r=0}^{k} f[x_0,\ldots,x_r].g[x_r,\ldots,x_k].$$

PROOF. Induction on k:

$$\begin{aligned} k=1:\ h[x_0,x_1] &= \frac{h(x_0)-h(x_1)}{x_0-x_1} \\ &= \frac{f(x_0).g(x_0)-f(x_1).g(x_1)}{x_0-x_1} \\ &= \frac{f(x_0).g(x_0)-f(x_0).g(x_1)+f(x_0).g(x_1)-f(x_1).g(x_1)}{x_0-x_1} \\ &= f(x_0).\frac{g(x_0)-g(x_1)}{x_0-x_1} + g(x_1).\frac{f(x_0)-f(x_1)}{x_0-x_1} \\ &= f(x_0).g[x_0,x_1] + g(x_1)f[x_0,x_1] \\ &= \sum_{r=0}^{1} f[x_0,\ldots,x_r].g[x_r,\ldots,x_k]. \end{aligned}$$

Assume that the theorem holds for a certain k, then

$$h[x_0,x_1,\dots,x_{k+1}] = \frac{h[x_0,\dots,x_k]-h[x_1,\dots,x_{k+1}]}{x_0-x_{k+1}}$$

$$= \frac{\sum_{r=0}^{k} f[x_0,\dots,x_r].g[x_r,\dots,x_k] - \sum_{r=1}^{k+1} f[x_1,\dots,x_r].g[x_r,\dots,x_{k+1}]}{x_0-x_{k+1}}$$

If we write $f_r^s = f[x_r,\dots,x_s]$, we get

$$\Big(\sum_{r=0}^{k} f_0^r.g_r^k - \sum_{r=1}^{k+1} f_1^r.g_r^{k+1}\Big)/(x_0-x_{k+1})$$

$$= (f_0^0.g_0^k - f_1^1.g_1^{k+1} + f_0^1.g_1^k - f_1^2.g_2^{k+1} + \dots$$

$$\dots + f_0^k.g_k^k - f_1^{k+1}.g_{k+1}^{k+1})/(x_0-x_{k+1})$$

$$= (f_0^0.g_0^k - f_0^0.g_1^{k+1} + f_0^0.g_1^{k+1} + f_0^1.g_1^k - f_0^1.g_2^{k+1} + f_0^1.g_2^{k+1} - f_1^2 g_2^{k+1} + \dots$$

$$\dots + f_0^k.g_k^k - f_0^k.g_{k+1}^{k+1} + f_0^k.g_{k+1}^{k+1} - f_1^{k+1}.g_{k+1}^{k+1})/(x_0-x_{k+1})$$

$$= f_0^0.\frac{g_0^k-g_1^{k+1}}{x_0-x_{k+1}} + \frac{f_0^0-f_1^1}{x_0-x_1} . \frac{x_0-x_1}{x_0-x_{k+1}} . g_1^{k+1}$$

$$+ f_0^1.\frac{g_1^k-g_2^{k+1}}{x_1-x_{k+1}} . \frac{x_1-x_{k+1}}{x_0-x_{k+1}} + \frac{f_0^1-f_1^2}{x_0-x_2} . \frac{x_0-x_2}{x_0-x_{k+1}} . g_2^{k+1} + \dots$$

$$\dots + f_0^k.\frac{(g_k^k-g_{k+1}^{k+1})}{x_k-x_{k+1}} . \frac{x_k-x_{k+1}}{x_0-x_{k+1}} + \frac{f_0^k-f_1^{k+1}}{x_0-x_{k+1}} . g_{k+1}^{k+1}$$

$$= f_0^0.g_0^{k+1} + f_0^1.g_1^{k+1} + f_0^2.g_2^{k+1} + \dots + f_0^{k+1}.g_{k+1}^{k+1}$$

$$= \sum_{r=0}^{k+1} f_0^r.g_r^{k+1}$$

$$= \sum_{r=0}^{k+1} .f[x_0,\dots,x_r].g[x_r,\dots,x_{k+1}]. \qquad \square$$

cf. [1], [7], [12].

# APPENDIX B

## The MC Graphical System and the C-programming Language

The Intermediate Language for Pictures (cf. [14]) defines the structure of an interactive graphical system, where pictures are being represented as ILP programs. The language is a special purpose language, an as such does not contain structures that would make it a *programming* language. A high level graphical languages arises from embedding ILP into an existing high-level general purpose language.

ILP contains four important features of graphical information:

1) elementary drawing actions.
2) modification of such drawing actions under control of state information.
3) structuring (and combining) states and actions.
4) specification of entry points for external references on which interaction and association of non-graphical data can be based.

The structuring of states and actions is achieved via the construction 'WITH state DRAW action'.

The present system is implemented on a PDP 11/45 under the UNIX7 operating system. UNIX is a trademark for a family of operating sysetms, developed by Bell labs. The available programming language is "C", which is the general purpose language in which the UNIX system itself is programmed. Unfortunately C does not allow for the above mentioned embedding of ILP, being the reason why an interface, CILP, has been developed.

CILP enables the writing in C of a 'picture program' in terms of ILP-constructions. Amongst others, it features the following:

- opening and closing procedures: '<u>pict</u>', '<u>endpict</u>'
- structuring procedures: '<u>with</u>', '<u>draw</u>', '<u>ward</u>'

    state between '<u>with</u>' and '<u>draw</u>'

    actions between '<u>draw</u>' and '<u>ward</u>'
- state procedures: e.g. '<u>translate</u>'
- action procedures: e.g. '<u>line</u>'
- aiding procedures: e.g. '<u>newpel</u>'

    '<u>newpel</u>' finishes an action.

The programming language C (cf. [11]), features economy of expression, modern control flow and data structuring capabilities, and a rich set of operators and data types. The features to be mentioned, as they facilitate the reading of the programs, are:

- types : the basic types in C are int, which represents an integer in the basic size provided by the machine architecture (PDP11: 16 bits), char, which represents a single byte, float, a single precision floating point number (PDP11: 32 bits), and double, a double precision floating point.
- statements : statements are grouped with braces {and}.
- control flow : control flow in C differs from other languages primarily in details of syntax.
  - The if - else statement has the form

    ```
    if (expr)
         statement 1.
    else statement 2.
    ```

    If 'expr' has a true (non-zero) value, then the first statement is done, else, if the optional else-part is there, the second statement is executed.
  - The while statement is simply

    ```
    while (expr)
          statement.
    ```

    The expression is evaluated and if it yields true the statement is executed, and the process repeats. When the expression becomes false, execution terminates.
  - The for-loop is

    ```
    for (expr1; expr2; expr3)
        statement.
    ```

    The first expression is done once, before the loop is proper entered. The second part is the test or condition, that controls the loop: if it yields true, the statement is evaluated. Then expr3 is done, and the condition re-evaluated. The loop terminates when the condition becomes false.

| | |
|---|---|
| - variables | : data declared outside the body of any function definition are static in lifetime, exist throughout the execution of the program, and are automatically initialized to 0. Variables declared within a function body are by default <u>automatic</u>: they come into existence when the function is entered and vanish when it is exited. If they are not set, they will contain garbage! |
| - pointers | : A pointer is a variable that contains the address of another variable. C has the unary operators & and *: the first when applied to a name, yields the address of the cell corresponding to the name, while the latter yields the value in the cell which is pointed to by its argument. - the statement px = &x; assigns to px the number that can be interpreted as the address of x; - the statement y = *px; assigns to y the value pointed to by px. |
| - arrays | : In C an array of 10 integers might be declared as `int Array[10];` Arrays always begin at zero: the elements of Array are Array[0],Array[1],...,Array[9]. When the name "array' appears in an expression it is converted to a pointer to the first member of the array. |
| - functions | : A function is a subprogram that returns an object of a given type: `double ppval();` declares the function 'ppval' to return a double. In a function, the arguments have to be declared so their types are known. The argument declarations go between the argument list and the opening left brace. In C, all function arguments are passed "by value". This means that the called function is given a copy of the value of its arguments, and it follows that the function cannot alter a variable. When necessary, it is possible to arrange |

for a function to modify a variable: The caller must provide the address of the variable, and the function must declare the argument to be a pointer. When the name of an array is used as an argument, the value passed to the function is actually the address of the beginning of the array. By subscripting this value, the function can access and alter any element of the array.

- C-preprocessor: Symbolic naming is not part of the syntax of C, but is provided for by a macro-processor which is automatically invoked as part of every C compilation. For example, given the definition

```
# define MAXORDER 4
```

the preprocessor replaces all occurrences of MAXORDER by 4. A second service of the preprocessor is library file inclusion: a source line of the form

```
# include "constants"
```

causes the contents of the file 'constants' to be inserted in the source at that point.

- operators : & - address operator
  - && - logical AND operator
  - || - logical OR operator
  - -- - decrement operator (by 1)
  - ++ - increment operator (by 1)
  - * - indirection operator
    multiplication operator.
  - ! - inequality operator.

The drawings were made on a High Resolution Display (HRD) of Laserscan. This is a system which consists of

- source of light (Argon Ion Laser)
- drawing system (Modulator with mirrors)
- internal film system (Projection on frosted silk screen)
- external film system (adds to the system a plotter).

The external film system makes a hard copy on a fiche, and can be invoked by the program through the procedure 'diazo' from the library 'piclib.h'.

APPENDIX C. In this appendix, some procedures which do not follow directly from the text, are being given:

1. polval
2. interval
3. indexmax
4. sort
5. bsplerror
6. min
7. abs

```
double
polval(coef,x)
        double coef[],x;
        /* The procedure calculates the value in x of the
         * polynomial of which the coefficients are given
         * in coef[].
         */
{
        double val;
           int i;

        val = coef[3];
        for (i=2; i>=0; i--)
             val = val*x + coef[i];

        return(val);
}



interval(knot,imax,left,x,flag,lastleft)
        double *knot, x;
        int imax, *left, *flag, *lastleft;
       /* The procedure 'interval', searches, for a given knot
        * sequence t, and a given abscis x, for the number i
        * with t[i] <= x < t[i+1] :
        *
        * OUTPUT:                LEFT:  FLAG:  LASTLEFT:
        *
        * t[i] <= x < t[i+1]     i      0      i
        *         x < t[0]       0      -1     0
        * t[imax] <= x           imax   +1     imax
        *
        * The procedure is designed to be efficient in the common
        * situation that it is called repeatedly, with x taken from
        * an increasing or decreasing sequence. The first guess for
        * left is therefore taken to be the value returned at the
        * previous call, and stored in the global variable LASTLEFT
        * If t[lastleft] <= x < t[lastleft+1], we are done  after
        * three comparisons. Otherwise, we repeatedly double the
        * difference istep = u - 1, while also moving 1 and u in the
        * direction of x, until t[1] <= x < t[u], afer which we use
        * bisection to get, in addition, 1+1=u.
        * LEFT = 1 is then returned.
        * This procedure is a copy of the routine given by de Boor [2].
        */
{       int 1,u,istep,middle;
        1= *lastleft;
        u= 1+1;
        istep= 1;
        if (u > imax)
           {u = imax;
            l = u - 1;
           }
        if (knot[l] <= x && x < knot[u])
           {*left = *lastleft = l;
            *flag = 0;
           }
        else
        if (x >= knot[imax])
           {*left = *lastleft = imax;
            *flag = 1;
           }
        else
        if (x < knot[0])
           {*left = *lastleft = 0;
            *flag = -1;
           }
        else
        if (x >= knot[u])
           {while (x >= knot[u])
                  {l = u;
                   if (istep > imax - 1)
                       u = imax;
                   else
                       u = l + istep;
                       istep = 2 * istep;
                  }
            while ((middle = (l + u)/2) != l)
                  {if (x >= knot[middle])
                       l = middle;
                   else
                       u = middle;
                  }

            *left = *lastleft = middle;
            *flag = 0;
           }
        else
           {while (x < knot[l])
                  {u = l;
                   if (istep > u )
                       l = 0;
                   else
                       l = u - istep;
                       istep = 2 * istep;
                  }
            while ((middle = (l + u)/2) != l)
                  {if (x >= knot[middle])
                       l = middle;
                   else
                       u = middle;
                  }

            *left = *lastleft = middle;
            *flag = 0;
           }
}
```

```
indexmax(x,n)
        double x[];    /* In: x[0]...x[n]   */
           int n;
        /* Produces index of max. element of str. increasing seq. x[]. */
{
        double m;
           int i,index;

        index=0;
        m=x[0];
        for (i=1; i<=n; i++)
            {if (x[i]>m)
                {m=x[i];
                 index=i;
                }
            }
        return(index);
}



sort(x,n)
        double x[];
           int n;
        /* Sorts the str. increasing array x[] to x[0]<x[1]<...<x[n]   */
{
        double hulp;
           int m,i,indexmax();
        for (i=n; i>=1; i--)
            {
             m=indexmax(x,i);
             hulp=x[i];
             x[i]=x[m];
             x[m]=hulp;
            }
}


bsplerror(n) int n;
{
        switch(n)
        {
        case SINGULAR     : fprintf(stderr,("bandmatrix singular."));
                            exit(1);
        case DERIV        : fprintf(stderr,("The given abscis x lies "));
                            fprintf(stderr,("outside interval on which"));
                            fprintf(stderr,("the spline is defined.."));
                            exit(1);
        }

}
```

```
min(i,j)        int i,j;
{
        int r;
        if (i<=j)
            r = i;
        else
            r = j;
        return(r);
}


double
abs(x) double x;
{       double y;
        if (x<0)
            y = -x;
        else
            y =  x;

        return(y);
}
```