

**stichting
mathematisch
centrum**



AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 177/81

AUGUSTUS

A.B. TOL

THE B-REPRESENTATION OF PIECEWISE POLYNOMIAL
PARAMETRIC CURVES AND LOCAL ADAPTION

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

The b-representation of piecewise polynomial parametric curves and
local adaption

by

Albert Tol

ABSTRACT

In this paper a description is provided for smooth curves satisfying a number of conditions, based on the theory of b-splines developed by C. de Boor. Local adaption to a new datapoint, cyclic curves and anti-cyclic continuations are the main features. Algorithms are presented as they are derived from the theory developed.

KEY WORDS & PHRASES: *b-splines, parametric curves, computer graphics*

CONTENTS

0.	INTRODUCTION	1
1.	THEORY OF pp-FUNCTIONS	3
1.1.	pp-functions	3
1.2.	Divided differences.	3
1.3.	b-splines.	4
1.4.	Properties of b-splines.	7
1.5.	Calculation of b-coefficients.	9
1.6.	Conversion b-representation \leftrightarrow pp-representation	10
2.	B-REPRESENTATION	13
2.1.	Advantages of the b-representation - cubic splines	13
2.2.	Advantages of the b-representation - local adaption.	15
3.	PARAMETRIC CURVES.	20
3.1.	pp-parametric curves	20
3.2.	The b-representation of cyclic pp-curves	22
3.3.	Anti-cyclic continuation	27
	References.	31
	Appendices.	32

0. INTRODUCTION

The main object of this paper is to provide a description for smooth curves satisfying a number of conditions, based on the theory of b-splines developed by C. DE BOOR ([1]). A condition may be for instance the occurrence of a given point on the curve, or, whether the curve is cyclic or not. The curves (pp-curves) are parametric curves: $P = P(t) = (p_1(t), \dots, p_{\dim}(t))$; each $p_i(t)$ is a piecewise polynomial function (pp-function).

The basic algorithms for the calculation of pp-functions with the aid of b-splines are described in detail by C.J. RUSMAN ([7]).

In the first chapters a concise description of the theory of pp-functions and b-splines is presented. The numerical aspects (e.g. the conditions of the b-matrices) are discerned, but not further elaborated (see [1], also for references). Special attention is given to a uniform notation per knots, datapoints, breakpoints etc. The theory as given in [1] for instance, lacks this uniformity. This makes it difficult to apply results in a different area. A uniform notation forms a basis for uniform datastructuring in a collection of algorithms. The algorithms as presented in this paper all use the same knot and datapoint organisation.

Chapters 7 and 8 give some reasons why the b-representation of pp-functions (and pp-curves), i.e. the representation by means of b-splines, is preferred to the pp-representation in most cases. If we want to plot a pp-function or pp-curve however, the work is easier done with the pp-representation (Chapter 6). pp-curves are introduced in Chapter 9. These allow for closed curves and so-called anti-cyclic continuations. The cyclic case is dealt with in Chapter 10. Special arrangements are made for anti-cyclic continuations (Chapter 11). For pp-functions methods are suggested to adapt the functions to a new function value (local adaption, Chapter 8). These methods can also be applied to pp-curves. The theory of Béziér-curves is often used to produce curves of a desired shape; the curve is shaped by changing the vertices of the corresponding polygon (see [4]). The Béziér-curve is a special case of pp-curves, the vertices being nothing more than the b-coefficients. So, the methods of local adaption in Chapter 8 give us a wider range of possibilities for producing smooth curves of a desired shape.

Some subjects in this paper are only briefly mentioned or not mentioned at all, but are worthwhile to be worked out or looked at:

- local adaption for pp-curves
- altering the knot-sequence in local adaption
- different methods of anti-cyclic continuation (e.g. with a fixed slope)
- different continuations (e.g. under a given angle).

The theory is described in close connection with the computer programs for the plotting of the pp-functions and pp-curves. To give an impression of what such programs may look like, a complete computer program for the plotting of pp-curves is presented in the appendix. Appendix B contains a newly developed algorithm for dynamically adapting the stepsize to the curvature.

The computer programs were tested and the pictures were drawn at the Mathematisch Centrum, Department of computer science, Amsterdam. I wish to thank the MC for putting its computers and plot-devices at my disposal. I also like to thank Paul ten Hagen, who introduced me to the subject, for his good advices.

1. THEORY OF pp-FUNCTIONS

In this chapter the theory of pp-functions as given in DE BOOR [1] is formulated, using a uniform notation for knots, datapoints and breakpoints. The notation as developed here greatly simplifies formulation of the results as given in Chapters 2 and 3. The algorithms derived from the theory use datastructures which reflect this uniform representation. As a result the various algorithms can be combined without any restructuring of data.

1.1. pp-functions

Let $E = \bigcup_{i=1}^{\ell+1} \xi_i \subset \mathbb{R}$, with $\xi_j < \xi_{j+1}$, $1 \leq j \leq \ell$. The collection of all functions $P(x): x \rightarrow \mathbb{R}$, $\xi_1 \leq x < \xi_{\ell+1}$, with the properties:

$$1) P_i(x) = \sum_{j=0}^{k-1} \frac{(x-\xi_i)^j}{j!} c(i,j), c(i,j) \in \mathbb{R}, \text{ for } \xi_i \leq x < \xi_{i+1},$$

$$P = \bigcup_{i=1}^{\ell} P_i.$$

2) With every ξ_i , $1 < i \leq \ell$, is an integer v_i , $0 \leq v_i \leq k$, associated;

$$N = \bigcup v_i;$$

$$\text{if } v_i > 0: P_{i-1}^{(j-1)}(\xi_i) = c(i,j-1) \text{ for } j = 1, \dots, v_i$$

constitutes a linear space $\mathbb{P}(k, E, N)$, with dimension $k \cdot \ell - \sum_{i=2}^{\ell} v_i$. k is the *order*, E the set of *breakpoints* and N the set of numbers of *continuation conditions* of $\mathbb{P}(k, E, N)$.

A member of $\mathbb{P}(k, E, N)$ is called a *pp-function* (piecewise polynomial function).

A pp-function with $v_i \geq k-1$ for $i = 2, \dots, \ell$ is a *spline function* or *spline*.

A breakpoint ξ_i with $v_i = k$ is a *pseudo-breakpoint*.

The *pp-representation* of a pp-function consists of k , E and a set of *pp-coefficients* $c(i,j)$, $i = 1, \dots, \ell$; $j = 0, \dots, k-1$.

1.2. Divided differences

Let $U t_i \subset \mathbb{R}$, $t_i \leq t_{i+1}$. The k -th *divided difference* $[t_i, \dots, t_{i+k}]g$ of a function g at t_i, \dots, t_{i+k} is the coefficient of x^k of the polynomial $p_{k+1}(x)$, the *interpolating polynomial*, of order $k+1$ (degree k) with the property:

$$p_{k+1}^{(m_j)}(t_j) = g^{(m_j)}(t_j), \quad m_j = \max(j - ii \mid t_j = t_{ii}, i \leq ii \leq i+k) \quad *)$$

Divided differences are most easily computed recursively using the formulas

$$[t_i, \dots, t_{i+k}]g = \frac{g^{(k)}(t_i)}{k!}, \quad \text{if } t_i = \dots = t_{i+k}$$

(note that $[t_i]g = g(t_i)$), and

$$[t_i, \dots, t_{i+k}]g = \frac{[t_{i+1}, \dots, t_{i+k}]g - [t_i, \dots, t_{i+k-1}]g}{t_{i+k} - t_i}$$

else (see for a derivation [2], p.277-278). The divided differences up to a desired k are commonly arranged in a *divided difference scheme*.

1.3. b-splines

Let $T = \cup t_i \subset \mathbb{R}$, $t_i \leq t_{i+1}$. The i -th *b-spline* of order k for *knot-sequence* T , $B_{i,k,T}$ (B_i for short), is defined by:

$$B_{i,k,T}(x) = (t_{i+k} - t_i)[t_i, \dots, t_{i+k}](t-x)_+^{k-1}, \quad x \in \mathbb{R},$$

with $(t-x)_+^{k-1} = (t-x)^{k-1}$ if $x < t$ and 0 else. With the linear space $\mathbb{P}(k, \mathbb{E}, N)$ we associate a collection of knot-sequences, $T(k, \mathbb{E}, N) = \cup_{i=1}^{n+k} t_i$, with

- 1) $t_1 \leq \dots \leq t_k = \xi_1$, $t_{n+k} \geq \dots \geq t_{n+1} = \xi_{\ell+1}$.
- 2) For $\xi_1 < \xi_i < \xi_{\ell+1}$ there is a j so that $t_j = t_{j+1} = \dots = t_{j+k-v_i-1} = \xi_i$; $k-v_i$ is the *multiplicity* of $t_j, \dots, t_{j+k-v_i-1}$; we also say $\bigcup_{ii=j}^{j+k-v_i-1} t_{ii}$ is a *multiple knot* if $k-v_i > 1$.
- 3) $n = \dim(\mathbb{P}(k, \mathbb{E}, N))$.

*) $\left. \left(\frac{d}{dt} \right)^r f(t) \right|_{t=t_j}$ is often shortened to $f^{(r)}(t_j)$, although it is actually incorrect.

We now are able to formulate the important theorem, which relates $\bigcup_i B_{i,k,T}$, $T \in T(k, \Xi, N)$, with $\mathbb{P}(k, \Xi, N)$:

THEOREM. $\bigcup_{i=1}^n (B_{i,k,T}|_{[\xi_1, \xi_{\ell+1})})$, $T \in T(k, \Xi, N)$, constitutes a basis for $\mathbb{P}(k, \Xi, N)$. (Hence the name "b-spline".) $n = \dim(\mathbb{P}(k, \Xi, N))$, so if we can prove that $B_i|_{[\xi_1, \xi_{\ell+1})} \in \mathbb{P}(k, \Xi, N)$, we only have to show that $B_1|_{[\xi_1, \xi_{\ell+1})}, \dots, B_n|_{[\xi_1, \xi_{\ell+1})}$ are linear independent.

i) $B_i|_{[\xi_1, \xi_{\ell+1})} \in \mathbb{P}(k, \Xi, N)$.

PROOF. From the divided difference scheme it is clear that there are numbers $d_j \in \mathbb{R}$ so that $[t_i, \dots, t_{i+k}]g = \sum_{j=i}^{i+k} d_j g^{(m_j)}(t_j)$. So for B_i we get:

$$B_i(x) = (t_{i+k} - t_i) \sum_{j=i}^{i+k} d_j (t_j - x)_+^{k-1-m_j} (k-1)! / (k-1-m_j)!.$$

This is a pp-function of order k with (not necessarily all) breakpoints in Ξ . By the definition of m_j and the construction of T we know that $m_j \leq k - v_{jj} - 1$ if $t_j = \xi_{jj}$. From the fact that $k-1-m_j \geq k-1-k+v_{jj}+1 = v_{jj}$, it follows that B_i has at least v_{jj} continuous derivatives at ξ_{jj} , so

$$B_i|_{[\xi_1, \xi_{\ell+1})} \in \mathbb{P}(k, \Xi, N). \quad \square$$

ii) Let the linear functional λ_i be defined by:

$$\lambda_i f = \sum_{r=0}^{k-1} (-1)^{k-1-r} \psi^{(k-1-r)}(\tau_i) f^{(r)}(\tau_i),$$

with $\psi(t) = (t_{i+1} - t) \dots (t_{i+k-1} - t) / (k-1)!$ and $t_i < \tau_i < t_{i+k}$.

Then $\lambda_i B_j = \delta_{ij}$ (the Kronecker delta) (DE BOOR & FIX, 1973).

PROOF.

$$\begin{aligned} \lambda_i (t-x)^{k-1} &= \sum_{r=0}^{k-1} (-1)^{k-1-r} \psi^{(k-1-r)}(\tau_i) (k-1) \dots (k-r) (-1)^r (t-\tau_i)^{k-1-r} (*) \\ &= \end{aligned}$$

*) with the convention $(k-1) \dots (k) = 1$.

$$\begin{aligned}
&= (-1)^{k-1} (k-1)! \sum_{r=0}^{k-1} \{ \psi^{(k-1-r)}(\tau_i) / (k-1-r)! \} (t-\tau_i)^{k-1-r} \\
&= (-1)^{k-1} (k-1)! \psi(t),
\end{aligned}$$

for ψ is a polynomial of order k . So for $(t-x)_+^{k-1}$ the following equation holds:

$$\lambda_i (t-x)_+^{k-1} = (-1)^{k-1} (k-1)! \psi(t) (t-\tau_i)_+^0.$$

Since

$$\left(\frac{d}{dt}\right)^{m_r} \lambda_i (t-x)_+^{k-1} = \lambda_i \left(\left(\frac{d}{dt}\right)^{m_r} (t-x)_+^{k-1}\right),$$

we have

$$\begin{aligned}
\lambda_i B_j &= (t_{j+k} - t_j) \sum_{r=j}^{j+k} d_r \lambda_i \left\{ \left(\frac{d}{dt}\right)^{m_r} (t-x)_+^{k-1} \right\} \Big|_{t=t_r} \\
&= (t_{j+k} - t_j) \sum_{r=j}^{j+k} d_r (-1)^{k-1} (k-1)! \left(\frac{d}{dt}\right)^{m_r} (\psi(t) (t-\tau_i)_+^0) \Big|_{t=t_r} \\
&= (t_{j+k} - t_j) (-1)^{k-1} (k-1)! [t_j, \dots, t_{j+k}] (\psi(t) (t-\tau_i)_+^0).
\end{aligned}$$

$[t_j, \dots, t_{j+k}] (\psi(t) (t-\tau_i)_+^0) = 0$ if $j \neq i$; if $j = i$, $\psi(t) (t-\tau_i)_+^0$ agrees with $\psi(x) (x-t_i) / (t_{i+k} - t_i)$ at t_i, \dots, t_{i+k} . The coefficient of x^k is

$$\frac{(-1)^{k-1}}{(k-1)! (t_{i+k} - t_i)},$$

so

$$\lambda_i B_i = \frac{(t_{i+k} - t_i) (-1)^{k-1} (k-1)! (-1)^{k-1}}{(k-1)! (t_{i+k} - t_i)} = 1. \quad \square$$

For $T \in T(k, \mathbb{E}, N)$ a pp-function $P \in \mathbb{P}(k, \mathbb{E}, N)$ can uniquely be written as $\sum_{i=1}^n \alpha_i B_{i,k,T} | [\xi_1, \xi_{\ell+1})$, $\alpha_i \in \mathbb{R}$. A *b-representation* of P consists of k , T and n *b-coefficients* α_i , $i = 1, \dots, n$.

If we know a priori that $P \in \mathbb{P}(k, \mathbb{E}, N)$ then the n *b-coefficients* going with a $T \in T(k, \mathbb{E}, N)$ are for instance determined

- 1) by giving n different function values (i.e., n different abscissae),
- 2) by giving n_1 different function values and $n-n_1$ additional (continuation) conditions.

1.4. Properties of b-splines

b1) $B_i(x) = 0$ for $x < t_i$ or $x > t_{i+k}$.

PROOF. For $x < t_i$ we have $(t-x)_+^{k-1} = (t-x)^{k-1}$ on $[t_i, t_{i+k}]$ and $B_i(x) = (t_{i+k} - t_i)[t_i, \dots, t_{i+k}](t-x)^{k-1} = 0$. (The coefficient of x^k is 0.) For $x > t_{i+k}$ $(t-x)_+^{k-1} \equiv 0$ on $[t_i, t_{i+k}]$ and $B_i(x) = (t_{i+k} - t_i)[t_i, \dots, t_{i+k}](t-x)_+^{k-1} = 0$ again. Consequently, if $t_j \leq x < t_{j+1}$, only B_{j-k+1}, \dots, B_j are possibly non-zero on x . \square

b2) $\sum_i B_i(x) = 1$.

PROOF. From b1) follows $\sum_i B_i(x) = \sum_{i=j-k+1}^j B_i(x)$ if $t_j \leq x < t_{j+1}$.

$$\begin{aligned} \sum_{i=j-k+1}^j B_i(x) &= \sum_{i=j-k+1}^j ([t_{i+1}, \dots, t_{i+k}](t-x)_+^{k-1} \\ &\quad - [t_i, \dots, t_{i+k-1}](t-x)_+^{k-1}) \\ &= [t_{j+1}, \dots, t_{j+k}](t-x)_+^{k-1} \\ &\quad - [t_{j-k+1}, \dots, t_j](t-x)_+^{k-1} \\ &= 1 - 0 = 1. \end{aligned}$$

\square

$$b3) B_{i,k,T}(x) = \frac{t_{i+k} - x}{t_{i+k} - t_{i+1}} B_{i+1,k-1,T}(x) + \frac{x - t_i}{t_{i+k-1} - t_i} B_{i,k-1,T}(x).$$

PROOF. Let us look at the definition of divided difference. The interpolating polynomial p_{k+1} can be written as

$$p_1(x) + \sum_{i=2}^{k+1} (p_i(x) - p_{i-1}(x)) = \sum_{i=1}^{k+1} (x-t_1) \dots (x-t_{i-1}) [t_1, \dots, t_i] g$$

(the first term is $[t_1]g$). So $\sum_{r=i}^{i+k} (x-t_r) \dots (x-t_{r-1}) [t_i, \dots, t_r] g$. $\sum_{s=i}^{i+k} (x-t_{s+1}) \dots (x-t_{i+k}) [t_s, \dots, t_{i+k}] h$ agrees with gh at t_i, \dots, t_{i+k} and equals 0 for $r > s$ and $x = t_i, \dots, t_{i+k}$. Therefore $\sum_{r \leq s}$ also agrees with gh at t_i, \dots, t_{i+k} and by the definition of divided difference

$$[t_i, \dots, t_{i+k}]gh = \sum_{r=i}^{i+k} [t_i, \dots, t_r]g[t_r, \dots, t_{i+k}]h.$$

We use this formula for $(t-x)_+^{k-1} = (t-x)(t-x)_+^{k-2}$:

$$\begin{aligned} B_{i,k,T}(x) &= (t_{i+k} - t_i)[t_i, \dots, t_{i+k}](t-x)_+^{k-1} \\ &= (t_{i+k} - t_i)((t_i - x)[t_i, \dots, t_{i+k}](t-x)_+^{k-2} + [t_{i+1}, \dots, t_{i+k}](t-x)_+^{k-2}) \\ &= (t_{i+k} - t_i)((t_i - x) \frac{[t_{i+1}, \dots, t_{i+k}](t-x)_+^{k-2} - [t_i, \dots, t_{i+k-1}](t-x)_+^{k-2}}{t_{i+k} - t_i} \\ &\quad + [t_{i+1}, \dots, t_{i+k}](t-x)_+^{k-2}) \\ &= (t_{i+k} - x)[t_{i+1}, \dots, t_{i+k}](t-x)_+^{k-2} - (t_i - x)[t_i, \dots, t_{i+k-1}](t-x)_+^{k-2} \\ &= \frac{t_{i+k} - x}{t_{i+k} - t_{i+1}} B_{i+1,k-1,T}(x) + \frac{x - t_i}{t_{i+k-1} - t_i} B_{i,k-1,T}(x). \quad \square \end{aligned}$$

It is clear that $B_{i,1,T}(x) = 1$ for $t_i \leq x < t_{i+1}$ and 0 else, so, as a consequence, $B_{i,k,T} > 0$ for $t_i < x < t_{i+k}$.

$$b4) \frac{d}{dx}(\sum \alpha_i B_{i,k,T}(x)) = \sum (k-1) \frac{\alpha_i - \alpha_{i-1}}{t_{i+k-1} - t_i} B_{i,k-1,T}(x).$$

PROOF.

$$\begin{aligned} \frac{d}{dx} B_{i,k,T}(x) &= \frac{d}{dx}([t_{i+1}, \dots, t_{i+k}](t-x)_+^{k-1} - [t_i, \dots, t_{i+k-1}](t-x)_+^{k-1}) \\ &= -(k-1)([t_{i+1}, \dots, t_{i+k}](t-x)_+^{k-2} - [t_i, \dots, t_{i+k-1}](t-x)_+^{k-2}) \\ &= \frac{k-1}{t_{i+k-1} - t_i} B_{i,k-1,T}(x) - \frac{k-1}{t_{i+k} - t_{i+1}} B_{i+1,k-1,T}(x). \quad \square \end{aligned}$$

1.5. Calculation of b-coefficients

Let $P \in \mathbb{P}(k, \mathbb{E}, N)$ and $T \in T(k, \mathbb{E}, N)$. Furthermore, let *datapoints* τ_i , $i = 1, \dots, n$, $\xi_1 \leq \tau_i < \tau_{i+1} \leq \xi_{\ell+1}$, and $P(\tau_i)$, $i = 1, \dots, n$, be given. ($P(\xi_{\ell+1}) = P_\ell(\xi_{\ell+1})$.)

The questions that rise are:

- ⊙ How can we calculate the b-coefficients going with T and $P(\tau_i)$, $i = 1, \dots, n$?
 - ⊙ How must the τ_i 's be positioned to make the calculation possible in the first place?
- (It is clear that the problem is not solvable if we take for instance all the τ_i 's in the first ξ -interval.)

From property b1) follows

$$P(\tau_i) = \sum_{r=j-k+1}^j \alpha_r B_{r,k,T}(\tau_i) \quad \text{if} \quad t_j \leq \tau_i < t_{j+1}.$$

For $i = 1, \dots, n$ we get the n equations needed to determine $\alpha_1, \dots, \alpha_n$. We write: $(B_{ij})(\alpha_j)^T = (P(\tau_i))^T$, with (B_{ij}) the *b-matrix* of T and τ_i , $i = 1, \dots, n$. If $\tau_i \leq t_i$ or $\tau_i \geq t_{i+k}$, (B_{ij}) is not invertible. Let e.g. $\tau_i \leq t_i$. Then $B_{ij} = B_{j,k,T}(\tau_i) = 0$ for $j \geq i$. As τ_i , $i = 1, \dots, n$, is non-decreasing, we also have $B_{ii} = 0$ for $1 \leq ii < i$ and $j \geq i$, so the last $n-i+1$ columns only have possible non-zero elements in the last $n-i$ rows and are therefore dependent. Consequently: $t_i < \tau_i < t_{i+k}$. This makes (B_{ij}) banded with band-width $2k-1$.

The calculation of B_{ij} is conveniently performed with property b3) (subroutine 'bsplvx'): Suppose $t_j \leq \tau_i < t_{j+1}$; starting with $B_{j,1,T}(\tau_i) = 1$ and $B_{ii,1,T}(\tau_i) = 0$ for $ii \neq j$, we can compute $B_{j-k,k+1,T}(\tau_i), \dots, B_{j,k+1,T}(\tau_i)$ from $B_{j-k+1,k,T}(\tau_i), \dots, B_{j,k,T}(\tau_i)$, keeping in mind that $B_{j-k,k,T}(\tau_i) = B_{j+1,k,T}(\tau_i) = 0$:

$$\begin{aligned}
B_{j-k+ii-1,k+1,T}(\tau_i) &= \frac{t_{j+ii} - \tau_i}{(t_{j+ii} - \tau_i) + (\tau_i - t_{j-k+ii})} B_{j-k+ii,k,T}(\tau_i) \\
&+ \frac{\tau_i - t_{j-k+ii-1}}{(t_{j+ii-1} - \tau_i) + (\tau_i - t_{j-k+ii-1})} B_{j-k+ii-1,k,T}(\tau_i) \\
&= \frac{dr_{ii}}{dr_{ii} + dl_{ii-k+1}} B_{j-k+ii,k,T}(\tau_i) \\
&+ \frac{dl_{k+2-ii}}{dr_{ii-1} + dl_{k+2-ii}} B_{j-k+ii-1,k,T}(\tau_i),
\end{aligned}$$

with $dl_{jj} = \tau_i - t_{j+1-jj}$ and $dr_{jj} = t_{j+jj} - \tau_i$, $ii = 1, \dots, k+1$. Because (B_{ij}) is totally positive (no proof), the system can be solved without pivoting.

We store the non-zero elements of (B_{ij}) in an $n \times (2k-1)$ matrix. Sometimes the same b-matrix is later on used for another $(P(\tau_i))^T$ and instead of solving the system directly, we first perform an LU-decomposition on the condensed (B_{ij}) , which is saved (subroutine 'ludeco'). The theory on matrix-computations is clearly presented in e.g. [3], ch.3. With this LU-decomposition the solution is easily obtained (subroutine 'solsys').

1.6. Conversion b-representation \leftrightarrow pp-representation

If we want to plot a pp-function, i.e., to make a picture of it, we evaluate the function values of the pp-function at some points and draw straight lines between them in a certain coordinate-system. It is easier to evaluate the function values starting from the pp-representation, so, if we need many function values, as is usually the case with plotting, it is better to switch over to the pp-representation.

Suppose we have the disposal of a b-representation, k , T and α_i , $i = 1, \dots, n$, of a pp-function. The E corresponding with T is easily obtained. Note that $c(i,j) = \left(\frac{d}{dx}\right)^j P_i(\xi_i)$. So the problem reduces to calculating the function and the derivatives up to $k-1$ at ξ_1, \dots, ξ_ℓ (subroutine 'ppfppr'). With property b4) we can recursively calculate all the b-coefficients relevant for the function values of the $k-1$ derivatives at a certain

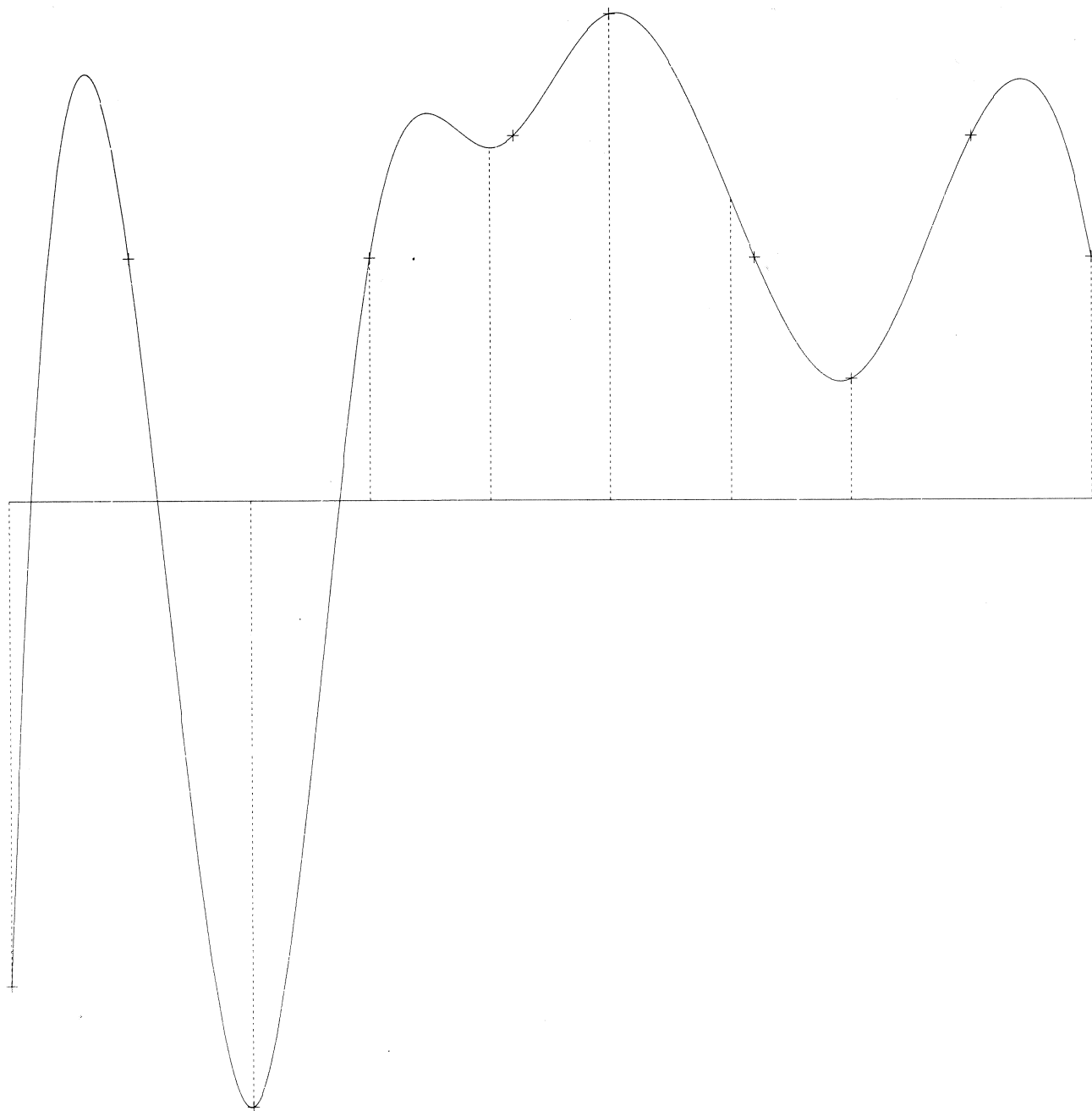


Fig. 1: pp-function of order 4 through 10 datapoints

ξ_{ii} . Let $t_{\text{left}} = \xi_{ii} \neq t_{\text{left}+1}$. A scratch-matrix s is constructed as follows:

$$1) s(i,1) = a_{\text{left}-k+i}, i = 1, \dots, k.$$

$$2) s(i,j+1) = (k-j) \frac{s(i+1,j) - s(i,j)}{t_{\text{left}+i} - t_{\text{left}+i-(k-j)}}, j = 1, \dots, k-1; i = 1, \dots, k-j;$$

$$t_{\text{left}+i} \neq t_{\text{left}+i-(k-j)}.$$

If $t_{\text{left}+i} = t_{\text{left}+i-(k-j)}$, $B_{\text{left}+i-(k-j),k-j,T}$ was already 0 and $s(i,j+1)$ is simply not calculated. With 'bsplvx' $B_{\text{left}-(k-j)+1,k-j,T(t_{\text{left}}), \dots, B_{\text{left},k-j,T(t_{\text{left}})}, 0 \leq j \leq k-1$, are determined. The pp-coefficients are then:

$$c(ii,j) = \sum_{i=1}^{k-j} s(i,j+1) B_{\text{left}+i-(k-j),k-j,T(t_{\text{left}})}, j = 0, \dots, k-1.$$

If on the other hand, we have the pp-representation of a pp-function P and want to have a b-representation, the work is done in three steps:

- 1) a $T \in T(k, \mathbb{E}, N)$ is constructed. First we determine N , then we take $k-v_i$ subsequent members of T equal to ξ_i .
- 2) We compute $P(\tau)$ for $\tau = \tau_1, \dots, \tau_n$, with $\tau_i < \tau_{i+1}$ and $t_i < \tau_i < t_{i+k}$.
- 3) With T and $P(\tau_1), \dots, P(\tau_n)$ the b-coefficients are fixed (see Chapter 5).

2. B-REPRESENTATION

2.1. Advantages of the b-representation - cubic splines

We compute the pp-coefficients of the spline function P of order 4 (*cubic spline*), with breakpoints $\xi_1, \dots, \xi_{\ell+1}$ and function values $P(\xi_i) = p_i$, $i = 1, \dots, \ell+1$, only using the pp-representation and not a b-spline (or any other) basis. For the $4 \times \ell$ unknowns $c(i, j)$, $i = 1, \dots, \ell$; $j = 0, \dots, 3$, we need as many equations:

- 1) $c(i, 0) = p_i$, $i = 1, \dots, \ell+1$;
- 2) $P_{i-1}(\xi_i) = c(i, 0)$, $i = 2, \dots, \ell$;
- 3) $P_{i-1}^{(1)}(\xi_i) = c(i, 1)$, $i = 2, \dots, \ell$;
- 4) $P_{i-1}^{(2)}(\xi_i) = c(i, 2)$, $i = 2, \dots, \ell$;
- 5) $P_1^{(3)}(\xi_2) = c(2, 3)$ and $P_{\ell-1}^{(3)}(\xi_\ell) = c(\ell, 3)$, making ξ_2 and ξ_ℓ pseudo-breakpoints.

The system is reduced to a set of $\ell+1$ equations with $c(i, 1)$, $i = 1, \dots, \ell+1$ as unknowns ($c(\ell+1, 1) = P_\ell^{(1)}(\xi_{\ell+1})$); we can write $P_i(x)$ as follows (see property b3)):

$$\begin{aligned} P_i(x) &= P(\xi_i) + (x - \xi_i)[\xi_i, \xi_i]P(x) + (x - \xi_i)^2[\xi_i, \xi_i, \xi_{i+1}]P(x) \\ &\quad + (x - \xi_i)^2(x - \xi_{i+1})[\xi_i, \xi_i, \xi_{i+1}, \xi_{i+1}]P(x). \end{aligned}$$

With $c(i, 1) = [\xi_i, \xi_i]P(x)$ we have:

$$\begin{aligned} c(i, 2) &= P^{(2)}(\xi_i) \\ &= 2[\xi_i, \xi_i, \xi_{i+1}]P(x) - 2(\xi_{i+1} - \xi_i)[\xi_i, \xi_i, \xi_{i+1}, \xi_{i+1}]P(x) \\ &= 2([\xi_i, \xi_{i+1}]P(x) - c(i, 1) - 2(c(i, 1) + c(i+1, 1) \\ &\quad - 2[\xi_i, \xi_{i+1}]P(x)))/(\xi_{i+1} - \xi_i) \end{aligned} \quad (1)$$

$$c(i,3) = 6(c(i,1) + c(i+1,1) - 2[\xi_i, \xi_{i+1}]P(x)) / (\xi_{i+1} - \xi_i)^2 \quad (2)$$

$$c(i-1,2) + c(i-1,3)(\xi_i - \xi_{i-1}) - c(1,2) = 0,$$

so for the equations of 4) we get

$$\begin{aligned} & c(i-1,1)(\xi_{i+1} - \xi_i) + 2c(i,1)(\xi_{i+1} - \xi_{i-1}) + c(i+1,1)(\xi_i - \xi_{i-1}) \\ & = 3((\xi_{i+1} - \xi_i)[\xi_{i-1}, \xi_i]P(x) + (\xi_i - \xi_{i-1})[\xi_i, \xi_{i+1}]P(x)), \quad i = 2, \dots, \ell. \end{aligned}$$

For the two equations of 5) we get:

$$\begin{aligned} & c(1,1)(\xi_3 - \xi_2) + c(2,1)(\xi_3 - \xi_1) \\ & = ((\xi_2 + 2\xi_3 - 3\xi_1)(\xi_3 - \xi_1)[\xi_1, \xi_2]P(x) + (\xi_2 - \xi_1)^2[\xi_2, \xi_3]P(x)) / (\xi_3 - \xi_1) \end{aligned}$$

and

$$\begin{aligned} & c(\ell,1)(\xi_{\ell+1} - \xi_{\ell-1}) + c(\ell+1,1)(\xi_{\ell} - \xi_{\ell-1}) \\ & = (\xi_{\ell+1} - \xi_{\ell})^2[\xi_{\ell-1}, \xi_{\ell}]P(x) + (3\xi_{\ell+1} - \xi_{\ell} - 2\xi_{\ell-1})(\xi_{\ell} - \xi_{\ell-1}) \cdot \\ & \quad \cdot [\xi_{\ell}, \xi_{\ell+1}]P(x)) / (\xi_{\ell+1} - \xi_{\ell-1}). \end{aligned}$$

Together we have a system of $\ell+1$ equations with a banded coefficient-matrix, band-width 3, which can be solved without pivoting. $c(i,2)$ and $c(i,3)$, $i = 1, \dots, \ell$, can then be computed with (1) and (2) (subroutine 'cubspl').

Now for a derivation by means of b-splines. We take a knot-sequence:

$$\begin{aligned} & t_1 \leq t_2 \leq t_3 \leq t_4 = \xi_1, \\ & t_i = \xi_{i-2}, \quad i = 5, \dots, \ell+1, \\ & \xi_{\ell+1} = t_{\ell+2} \leq t_{\ell+3} \leq t_{\ell+4} \leq t_{\ell+5}, \end{aligned}$$

and datapoints $\tau_i = \xi_i$, $i = 1, \dots, \ell+1$. (Note that the first and last two ξ -intervals agree with one t -interval.) After the determination of the b -coefficients, the b -representation can be converted to the pp -representation.

Using pp-representations only, forces us to start with a reduction step, reducing the $4 \times \ell$ equations to $\ell+1$ equations. The reduction algorithm changes with different order or continuation conditions and becomes more complicated with increasing order. Using a b-spline basis the alterations of the algorithm are simple and systematic and therefore allow parametrization.

2.2. Advantages of the b-representation - local adaption

Let P be a pp-function and $a \in \mathbb{R}$, $a \neq P(\xi)$, $\xi_1 \leq \xi \leq \xi_{\ell+1}$. Suppose we want to change P into a new pp-function P' so that

$$a1) \quad P'(\xi) = a$$

and P' stays as close to P as possible.

We can interpret this in different ways. In case P is a (cubic) spline function, constructed as described in the previous chapter, and ξ is a break-point ξ_j , we can compute the spline P' with $P'(\xi_i) = p_i$ for $i \neq j$ and $P'(\xi_j) = a$. Although in some sense the new spline stays as close to P as possible (all but one $P(\xi_i)$, $i = 1, \dots, \ell+1$, remain the same) the adaption is not local, because the whole spline, except at those ℓ points, changes. Moreover, the spline must be recomputed entirely.

To make the adaption local, function values at ξ_i 's adjacent to ξ_j must be 'released'. Now, suppose again we only have the pp-representation of a cubic spline function. If we change the function value at ξ_j , $P'(\xi_j) = a$, it is not possible to keep the spline unchanged for $x < \xi_{j-1}$ and $x \geq \xi_{j+1}$; we get ten equations for the eight pp-coefficients of the two intervals adjacent to ξ_j :

ξ_{j-1}	ξ_j	ξ_{j+1}
$P_{j-2}(\xi_{j-1}) = c(j-1,0)$	$c(j,0) = a$	$P_j(\xi_{j+1}) = c(j+1,0)$
$P_{j-2}^{(1)}(\xi_{j-1}) = c(j-1,1)$	$P_{j-1}(\xi_j) = c(j,0)$	$P_j^{(1)}(\xi_{j+1}) = c(j+1,1)$
$P_{j-2}^{(2)}(\xi_{j-1}) = c(j-1,2)$	$P_{j-1}^{(1)}(\xi_j) = c(j,1)$	$P_j^{(2)}(\xi_{j+1}) = c(j+1,2)$
	$P_{j-1}^{(2)}(\xi_j) = c(j,2)$	

If we release ξ_{j-1} or ξ_{j+1} we get thirteen equations for the twelve pp-coefficients of three intervals adjacent to ξ_j and if we release one more ξ_i , we finally get a solvable system of sixteen equations for sixteen pp-coefficients, for instance:

ξ_{j-3}	ξ_{j-2}	ξ_{j-1}	ξ_j	ξ_{j+1}
$P_{j-4}(\xi_{j-3}) = c(j-3,0)$	$P_{j-3}(\xi_{j-2}) = c(j-2,0)$	$P_{j-2}(\xi_{j-1}) = c(j-1,0)$	$c(j,0) = a$	$P_j(\xi_{j+1}) = c(j+1,0)$
$P_{j-4}^{(1)}(\xi_{j-3}) = c(j-3,1)$	$P_{j-3}^{(1)}(\xi_{j-2}) = c(j-2,1)$	$P_{j-2}^{(1)}(\xi_{j-1}) = c(j-1,1)$	$P_{j-1}(\xi_j) = c(j,0)$	$P_j^{(1)}(\xi_{j+1}) = c(j+1,1)$
$P_{j-4}^{(2)}(\xi_{j-3}) = c(j-3,2)$	$P_{j-3}^{(2)}(\xi_{j-2}) = c(j-2,2)$	$P_{j-2}^{(2)}(\xi_{j-1}) = c(j-1,2)$	$P_{j-1}^{(1)}(\xi_j) = c(j,1)$ $P_{j-1}^{(2)}(\xi_j) = c(j,2)$	$P_j^{(2)}(\xi_{j+1}) = c(j+1,2)$

There are three possibilities: the spline will change on (ξ_{j-3}, ξ_{j+1}) , (ξ_{j-2}, ξ_{j+2}) or (ξ_{j-1}, ξ_{j+3}) . If we raise the order of the spline by one, the above mentioned numbers will be: thirteen equations for ten unknowns, seventeen for fifteen and twentyone for twenty, so we have to release an extra ξ_i . In general we have to change the spline on one of the intervals (ξ_{j-k+i}, ξ_{j+i}) , $i = 1, \dots, k-1$, k the order, to keep the adaption local^{*}). We will make no attempt to solve the systems of equations. If we are working with a b-spline basis, everything will become much easier. Let k , T and α_i , $i = 1, \dots, n$, be a b-representation of the spline (of order k) and let $t_r = \xi_j$. By property b1) we know that $B_i(t_r) \neq 0$ for $i = r-k+1, \dots, r-1$. Adding $\frac{a-P(t_r)}{B_i(t_r)} B_i(x)$, $r-k+1 \leq i \leq r-1$, to the spline, gives a new spline function, which equals P for $x \leq t_i$ and $x \geq t_{i+k}$ and has function value a at t_r . So, by using a b-representation of the pp-function the same adaption is achieved in a far simpler way.

Although we can thus locally adapt a pp-function to a new function value by changing one b-coefficient, the result may not be what we wanted. Two additional conditions make the adaption more acceptable:

a2) $\max |P' - P| = a.$

a3) $\left. \frac{d}{dx}(P' - P) \right|_{x=\xi} = 0.$

^{*}) If the function values and derivatives up to $k-2$ are all zero at the end-points (which is the case for $P' - P$), we get the old, limited definition of b-splines.

Therefore, the following three options for adaption are suggested (sub-routine 'locadp'):

- 1) ξ is one of the underlying datapoints; the other datapoints are kept unchanged. The pp-function must be recomputed and the adaption will not be local, in general. The same (LU-decomposition of the) b-matrix can be used.
- 2) Let $i = \min(j \mid \left| \frac{d}{dx} B_{j,k,T}(\xi) \right| > 0) \ (k > 1)$.

If

$$\left| \frac{d}{dx} B_i(\xi) \right| < \left| \frac{d}{dx} B_{i-1}(\xi) \right|,$$

we add to the pp-function

$$\frac{a - P(\xi)}{B_i(\xi) - qB_{i-1}(\xi)} B_i(x) - \frac{a - P(\xi)}{B_i(\xi) - qB_{i-1}(\xi)} qB_{i-1}(x),$$

with

$$q = \frac{\frac{d}{dx} B_i(\xi)}{\frac{d}{dx} B_{i-1}(\xi)},$$

else:

$$\frac{a - P(\xi)}{B_{i-1}(\xi) - qB_i(\xi)} B_{i-1}(x) - \frac{a - P(\xi)}{B_{i-1}(\xi) - qB_i(\xi)} qB_i(x),$$

with

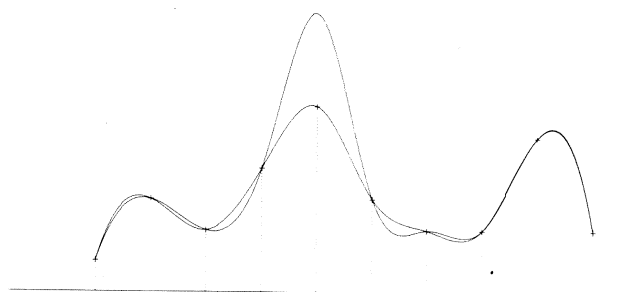
$$q = \frac{\frac{d}{dx} B_{i-1}(\xi)}{\frac{d}{dx} B_i(\xi)}.$$

The result is a pp-function satisfying conditions a1) and a3) (and maybe a2)). Note that only one b-coefficient will change, if $\frac{d}{dx} B_{i-1}(\xi) = 0$.

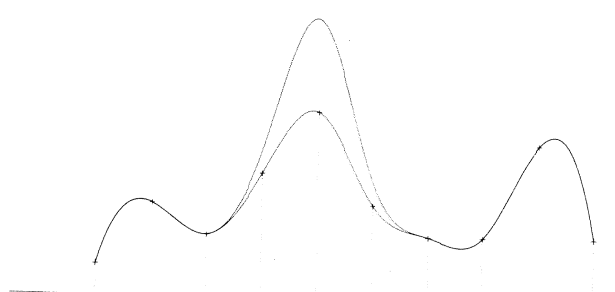
- 3) Let $t_i \leq \xi < t_{i+1}$.

We make use of property b2): $\sum_{j=i-k+1}^i B_j(\xi) = 1$.

We add: $\sum_{j=i-k+1-s}^{i+t-1} (a-P(\xi))B_j(x)$, $s \geq 0$, $t \geq 0$ ($t = 0$ can be used in case $\xi = t_i$).



a. option 1.



b. option 2.

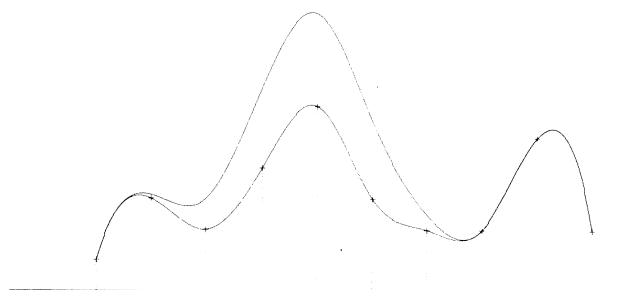
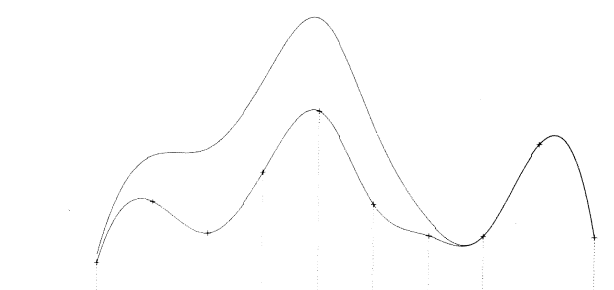
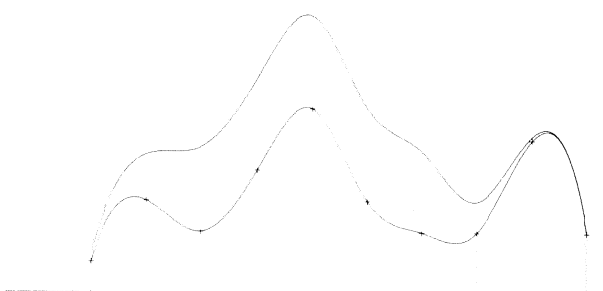
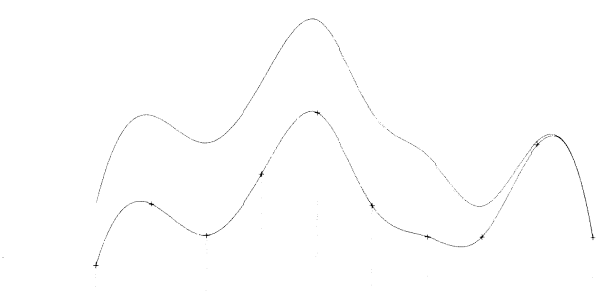
c. option 3: $t=0$, $s=0$.d. option 3: $t=0$, $s=1$.e. option 3: $t=1$, $s=1$.f. option 3: $t=1$, $s=2$.

Fig. 2: different methods of local adaption.

In this case a1), a2) and a3) are satisfied ($t > 0$); however, the function changes on a longer Interval as compared with the adapted pp-function in option 2. If s and t are big enough the whole function will shift over a distance a .

For the calculation of $B_{j,k,T}(\xi)$ and $\frac{d}{dx} B_{j,k,T}(\xi)$ the subroutine 'valuex' is used, which returns the function value or the value of a derivative at a given point. The method is roughly that of 'ppfppr'.

3. PARAMETRIC CURVES

3.1. pp-parametric curves

We are going to look at curves in a two dimensional space ($d = 2$). The discussion is easily generalized to curves in spaces of higher dimension. If p_1 and p_2 both are functions of the same parameter t , $p_1 = p_1(t)$ and $p_2 = p_2(t)$, the curve $P(p_1, p_2) = P(t) = (p_1(t), p_2(t))$ is called a *parametric curve*. An important difference with *implicit curves*, $P(p_1, p_2) = 0$ (P an algebraic expression in p_1 and p_2), is the fact, that parametric curves can have *multiple values*, i.e., $P(t_i) = P(t_j)$, $t_i \neq t_j$, whereas implicit curves can not (see Fig. 5).

Sometimes it is possible to parametrize an implicit cuve. Moreover, for the plotting of an ellipse the parametric form is better suited:

$$P(t) = (a \cos 2\pi t, b \sin 2\pi t), \quad 0 \leq t < 1.$$

The points $P(t_i)$ with $t_i = \frac{i}{n}$, $i = 0, \dots, n-1$, are nicely distributed over the ellipse. We use the parametric form of ellipses in subroutine 'plotkn' for plotting the knot-markers of pp-parametric curves.

A *pp-parametric curve*, or *pp-curve* for short, P is a parametric curve with both p_1 and p_2 members of the same linear space of pp-functions $\mathbb{P}(k, \mathbb{E}, N)$ and $P(t) = (p_1(t), p_2(t))$.

The pp-representation of a pp-curve of dimension d consists of k , \mathbb{E} and a set of pp-coefficients $c(m, i, j)$, $m = 1, \dots, d$; $i = 1, \dots, \ell$; $j = 0, \dots, k-1$. We might add a number v_1 (or $v_{\ell+1}$) to N , $N^+ = N \cup v_1$, so that

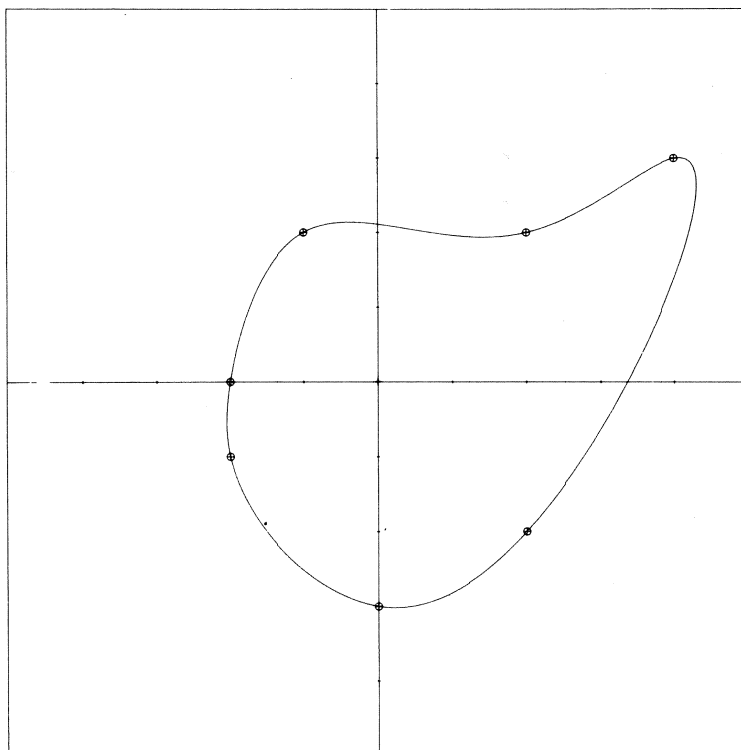
$$P_{\ell}^{(j-1)}(\xi_{\ell+1}) = (c(1, 1, j-1), c(2, 1, j-1))$$

for $j = 1, \dots, v_1$ (if $v_1 > 0$), with

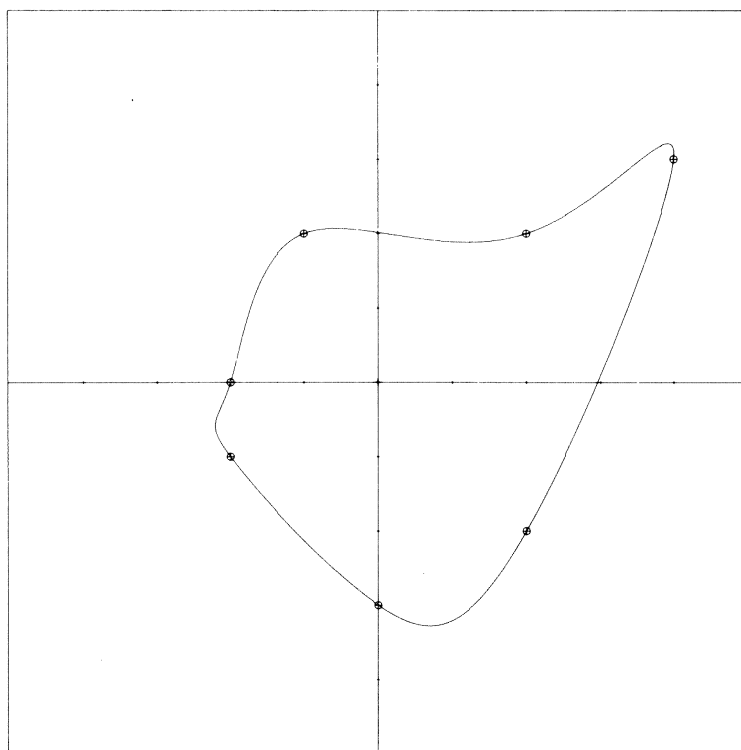
$$P_i^{(j-1)}(\xi) = (p_1^{(j-1)}(\xi), p_2^{(j-1)}(\xi)).$$

The collection of pp-curves with dimension d , order k , breakpoints \mathbb{E} and numbers of continuation conditions N^+ is denoted by $\mathbb{P}(d, k, \mathbb{E}, N^+)$.

$\mathbb{P}(d, k, \mathbb{E}, N) = \mathbb{P}(d, k, \mathbb{E}, N^+) \Big|_{v_1=0}$. Members of $\mathbb{P}(d, k, \mathbb{E}, N^+)$ are said to be *cyclic* if $v_i > 0$, $i = 1, \dots, \ell$.



a. datapoints chord-distant (ndist=1)



b. knots equidistant (ndist=2).

Fig. 3: cyclic pp-curve of order 4
(the knots are indicated by o, the datapoints by +).

3.2. The b-representation of cyclic pp-curves

Our aim is to construct a cyclic pp-curve through $n-k+1$ points $P(\tau_1), P(\tau_2), \dots, P(\tau_{n-k+1})$, with $\tau_i < \tau_{i+1}$, using the theory of b-splines we have dealt with so far (subroutine 'ppcinc'). The datapoints τ_i , $i = 1, \dots, n-k+1$ and the knots t_i , $i = 1, \dots, n+k$, are not fixed for the present.

A datapoint τ_i is placed in the last ξ -interval where $B_{i,k,T^+} \neq 0$ (T^+ is defined below):

$$\xi_j \leq \tau_i < \xi_{j+1}, \quad B_{i,k,T^+}(\tau_i) \neq 0$$

and

$$B_{i,k,T^+}(\xi_{j+1} + \epsilon) = 0, \quad \epsilon > 0.$$

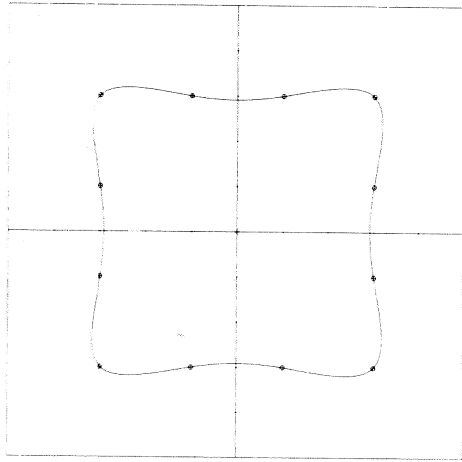
The format of the file, from which the points are read, is kept simple; three kinds of points are distinguished:

- code 0 points: points that are datapoints and not knots^{*)}
- code 1 points: points that are only knots (*floating knots*)
- code 2 points: points that are datapoints and at the same time knots.

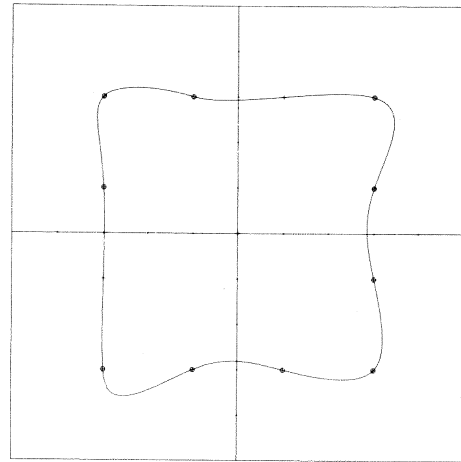
An input file may look like:

2	$p_1(\tau_1)$	$p_2(\tau_1)$	knot & datapoint
0	$p_1(\tau_2)$	$p_2(\tau_2)$	datapoint
1			knot
0	$p_1(\tau_3)$	$p_2(\tau_3)$	datapoint
2	$p_1(\tau_4)$	$p_2(\tau_4)$	knot & datapoint
3	dt1	dt2	anti-cyclic continuation (see next chapter)
2	$p_1(\tau_5)$	$p_2(\tau_5)$	knot & datapoint
4			skip interval while plotting
2	$p_1(\tau_6)$	$p_2(\tau_6)$	knot & datapoint
5			end of input.

^{*)} We use the word knot (datapoint) for $t_i(\tau_i)$, but also for $P(t_i)(P(\tau_i))$.

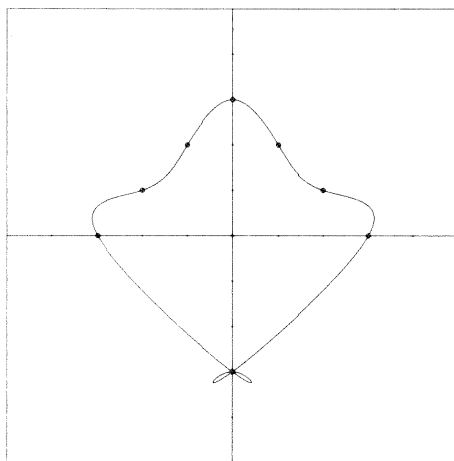


a. Without multiple knots

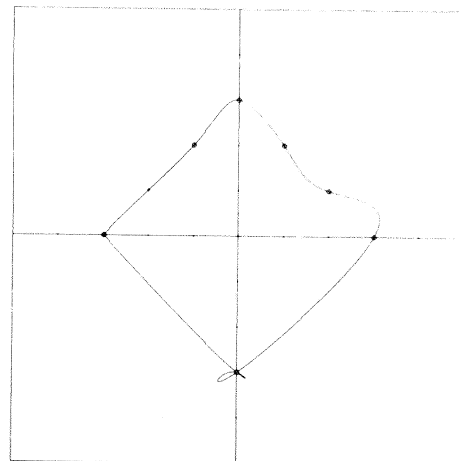


b. With two double knots

Fig. 4: Cyclic pp-curve (order 4, ndist=1) with multiple knots



a. Cyclic



b. Non-cyclic

Fig. 5: pp-curve (order 4, ndist=2) with a triple datapoint.

The following rules must be obeyed:

- r1) the first point (item) must be a knot
- r2) the last point must be equal to the first
- r3) an anti-cyclic continuation must follow a datapoint
- r4) a skip must follow a knot
- r5) a knot (code 1) must be followed by a datapoint (code 0).

The data are digested according to the following:

- 1) Two options for the distances between knots and datapoints are considered:
 - a) The breakpoints are equidistant: $\xi_{i+1} - \xi_i = c$, $i = 1, \dots, \ell$.
Datapoints of code 1 are equally spaced between the breakpoints.
 - b) The datapoints are *chord-distant*: $\tau_{i+1} - \tau_i = \|P(\tau_{i+1}) - P(\tau_i)\| = ((p_1(\tau_{i+1}) - p_1(\tau_i))^2 + (p_2(\tau_{i+1}) - p_2(\tau_i))^2)^{\frac{1}{2}}$.
Floating knots are placed halfway the two adjacent datapoints.

If the datapoints are chord-distant, an additional restriction is imposed upon the inputfile:

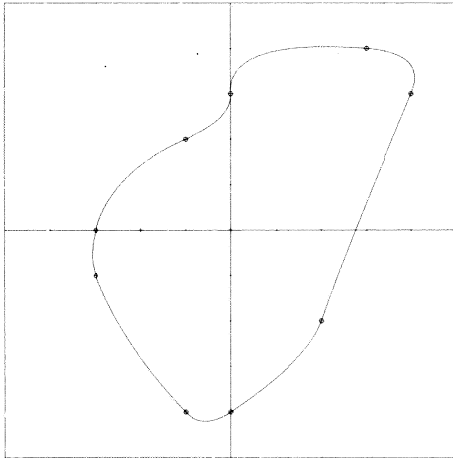
- r6) If the chord-distant option holds, two consecutive datapoints must not be equal.
- 2) The multiplicity of a knot is equal to the number of preceding points of code 0 and 2 counting backwards as far as the first encountered point of code 1 or 2 inclusive.
The first knot has multiplicity 1.
So, in the above example we get knots of multiplicity 1,2,1,1 and 1.
The first knot is t_k ($t_k = 0$), the last t_{n+1} . The first datapoint is τ_1 .

- 3) The extension of the knot-sequence t_1, \dots, t_{k-1} and t_{n+2}, \dots, t_{n+k} , is fixed by:
 - a) $t_i = t_k - (t_{n+1} - t_{n+1-k+i})$, $i = 1, \dots, k-1$.
 - b) $t_{n+1+i} = t_{n+1} + (t_{k+i} - t_k)$, $i = 1, \dots, k-1$.
 The knot-sequence t_1, \dots, t_{n+k} is denoted by T^+ .

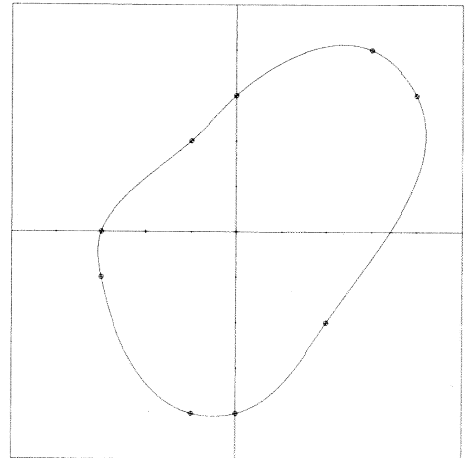
The b-coefficients $\alpha_{m,i}$, $m = 1, \dots, d$; $i = 1, \dots, n$, can now be determined:

- 1. For each co-ordinate we have $k-1$ *cyclic conditions*:

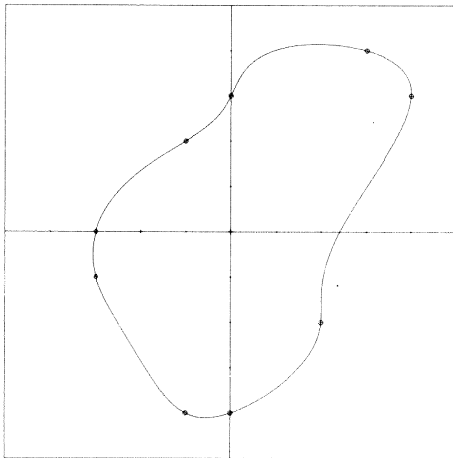
$$\alpha_{m,n-k+1+i} = \alpha_{m,i}, \quad 1 \leq m \leq d, \quad i = 1, \dots, k-1.$$



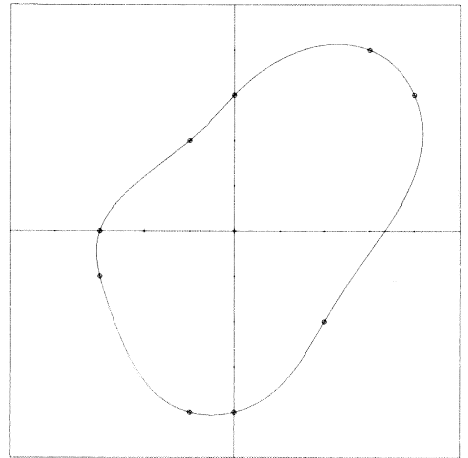
a. order 3



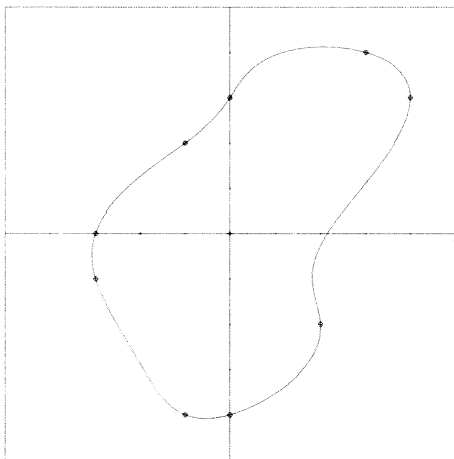
b. order 4



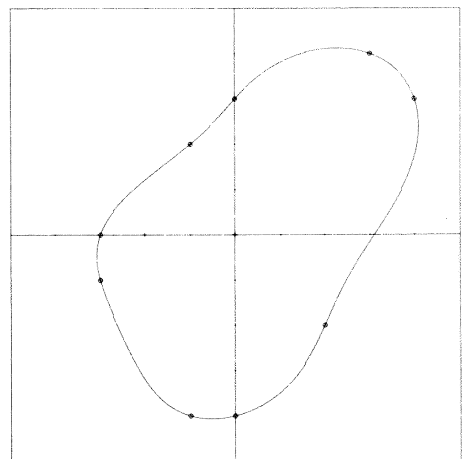
c. order 5



d. order 6



e. order 7



f. order 8

Fig. 6: Cyclic pp-curve (ndist=1)

2. For $\alpha_{m,i}$, $m = 1, \dots, d$; $i = 1, \dots, n-k+1$, we have d systems of $n-k+1$ equations:

$$\sum_{r=j-k+1}^j \alpha_{m,r} B_{r,k,T^+(\tau_i)} = p_m(\tau_i)$$

(provided that $t_j \leq \tau_i < t_{j+1}$), $m = 1, \dots, d$; $i = 1, \dots, n-k+1$.

As a consequence of the cyclic conditions the numbers $B_{r,k,T^+(\tau_i)}$ will not be situated anymore within the band of the b-matrix with band-width $2k-1$, if $r > n-k+1$. We have: $B_{ij} = B_{n-k+1+j,k,T^+(\tau_i)}$, if $1 \leq j \leq k-1$. Only the last $k-1$ rows may have non-zero elements in the first column.

Two ways of solving the systems are considered:

- a. Without pivoting (subroutines 'ludacy' and 'solsyc').

The first $n-2k+2$ rows are condensed into a $(n-2k+2) \times (2k-1)$ matrix, the last $k-1$ rows are stored in a $(k-1) \times (n-k+1)$ matrix.

- b. With complete pivoting (subroutines 'ludcyp' and 'solscyp').

The permutation which must be performed on the input $(p_m(\tau_1), \dots, p_m(\tau_{n-k+1}))$ is kept in the 0-th column. The permutation which must be performed on the output $(\alpha_{m,1}, \dots, \alpha_{m,n-k+1})$ is kept in the 0-th row.

In some cases it is possible (not guaranteed) to solve the systems without pivoting, for instance, if all datapoints are knots and the order is not too high.

Good results, i.e., pp-curves that are nicely smooth, are obtained with order 4, simple knots and datapoints chord-distant (see figures).

The same method can be used to produce non-cyclic pp-curves. This is done by reducing the multiplicity of the multiple knot at ξ_2 by $[k/2]-1$ and that of the multiple knot at $\xi_{\ell+1}$ by $k-[k/2]-1$. However, there are some limitations:

1. the first and the last read point must be code 2 points
2. the multiplicity of the multiple knot at $\xi_{\ell+1}$ must be $k-[k/2]$
3. the multiplicity of the multiple knot at ξ_2 must be at least $[k/2]$
4. the multiplicity of the multiple at ξ_2, \dots, ξ_{ℓ} must not exceed $k-[k/2]$ (after reducing).

It is not necessary anymore that the first and the last point are equal. The extension of the knot-sequence can be taken arbitrary.

The different methods of local adaption as dealt with in Chapter 8 can also be applied to pp-parametric curves. If a point (p_1, p_2) on the curve is shifted, a procedure must be yielded, which finds the underlying (or one of the underlying) $\xi, \xi_1 \leq \xi < \xi_{\ell+1}$.

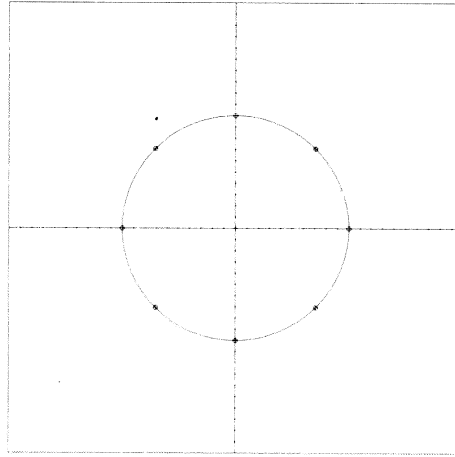


Fig. 7: Cyclic pp-curve (order 4) through eight points equidistant on a circle.

3.3. Anti-cyclic continuation

A parametric curve P has an *anti-cyclic continuation* at ξ , if

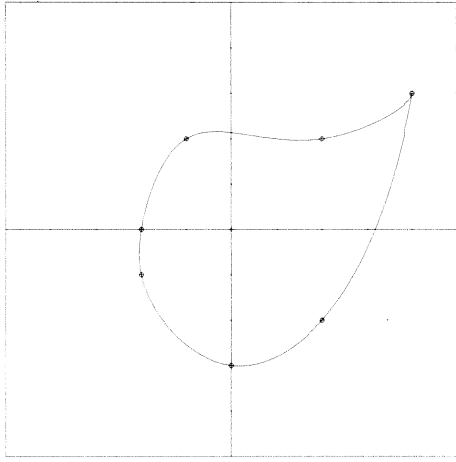
$$\lim_{x \uparrow \xi} P^{(1)}(x) = - \lim_{x \uparrow \xi} P^{(1)}(x),$$

i.e.,

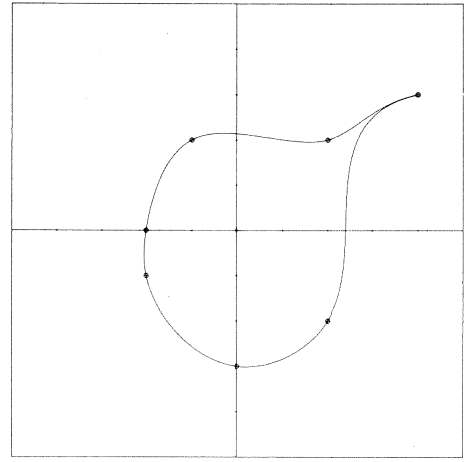
$$\lim_{x \uparrow \xi} p_m^{(1)}(x) = - \lim_{x \uparrow \xi} p_m^{(1)}(x), \quad m = 1, \dots, d.$$

As to pp-curves there are several ways to impose an anti-cyclic continuation upon the curve. One way is to give anti-cyclic boundary conditions (see [4], p.127). A drawback of this method is, that only one anti-cyclic continuation can be effectuated.

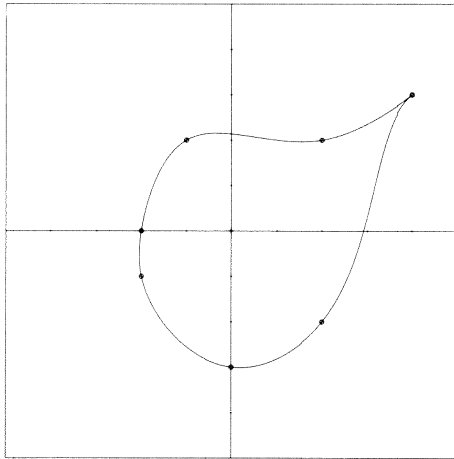
Using the (a) b-representation we can construct an anti-cyclic continuation at a breakpoint ξ_{ii} as follows: Let t_i be the (multiple) knot going with ξ_{ii} . Between t_j and t_{j+1} two additional knots are inserted, t_a and t_b say, with $t_a - t_j = dt1$, $t_b - t_a = dt2$ and $t_{j+1} - t_b = t_{j+1} - t_j$ ($dt1, dt2 \geq 0$). If $dt1 = 0$, then $dt2 \neq 0$. Consequently, two extra b-coefficients (for each co-ordinate) are to be determined.



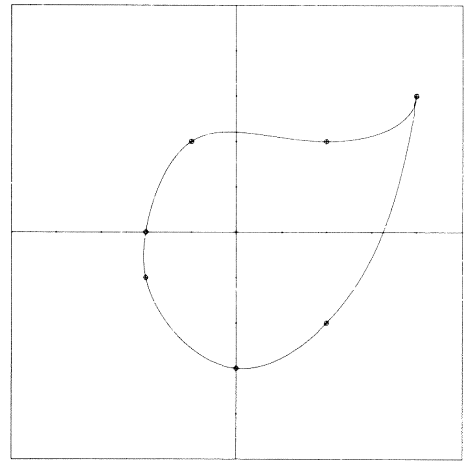
a. (2,0)-anti-cyclic



b. (0,2)-anti-cyclic



c. (1,1)-anti-cyclic



d. (1000,0)-anti-cyclic

Fig. 8: Cyclic pp-curve (order 4, ndist=1) with an anti-cyclic continuation (anti-clockwise)

The anti-cyclic continuation is effectuated by the two conditions:

- 1) $P(t_b) = P(t_j)$
- 2) $P^{(1)}(t_b) + P^{(1)}(t_j) = 0$.

After the calculation of the pp-curve the one or two intervals between t_j and t_b are omitted.

We use property b4) to write the second equation as a linear combination of b-coefficients:

$$p_m^{(1)}(t_j) = \frac{d}{dx} \left(\sum_i \alpha_{m,i} B_{i,k}(x) \right) \Big|_{x=t_j} = \sum_i (k-1) \frac{\alpha_{m,i}^{-\alpha_{m,i-1}}}{t_{i+k-1} - t_i} B_{i,k-1}(t_j).$$

Since $B_{i,k-1}(t_j) = 0$ for $i < j-k+2$ or $i > j-1$, we have:

$$\begin{aligned} p_m^{(1)}(t_j) &= (k-1) \sum_{i=j-k+2}^{j-1} \frac{\alpha_{m,i}^{-\alpha_{m,i-1}}}{t_{i+k-1} - t_i} B_{i,k-1}(t_j) \\ &= (k-1) \left(\frac{\alpha_{m,j-k+2}^{-\alpha_{m,j-k+1}}}{t_{j+1} - t_{j-k+2}} B_{j-k+2,k-1}(t_j) + \dots + \frac{\alpha_{m,j-1}^{-\alpha_{m,j-2}}}{t_{j+k-2} - t_{j-1}} B_{j-1,k-1}(t_j) \right) \\ &= (k-1) (-w_{j-k+2}^{\alpha_{m,j-k+1}} + w_{j-k+2}^{\alpha_{m,j-k+2}} - w_{j-k+3}^{\alpha_{m,j-k+2}} + \\ &\quad + w_{j-k+3}^{\alpha_{m,j-k+3}} - \dots - w_{j-1}^{\alpha_{m,j-2}} + w_{j-1}^{\alpha_{m,j-1}}), \end{aligned}$$

with

$$w_i = \frac{B_{i,k-1}(t_j)}{t_{i+k-1} - t_i}.$$

So, we get

$$p_m^{(1)}(t_j) = (k-1) (-w_{j-k+2}^{\alpha_{m,j-k+1}} + \sum_{i=j-k+2}^{j-2} (w_i - w_{i+1}) \alpha_{m,i} + w_{j-1}^{\alpha_{m,j-1}}),$$

or

$$p_m^{(1)}(t_j) = \sum_{i=j-k+1}^{j-1} (w_i - w_{i+1}) \alpha_{m,i},$$

with

$$w_{j-k+1} = w_j = 0.$$

We can alter the shape of the curve by changing the parameters $dt1$ and $dt2$. We could say, that the pp-curve is $(dt1, dt2)$ -anti-cyclic at ξ_{ii} (see Fig. 8). Of course, more than one anti-cyclic continuation is possible (see Fig. 9).

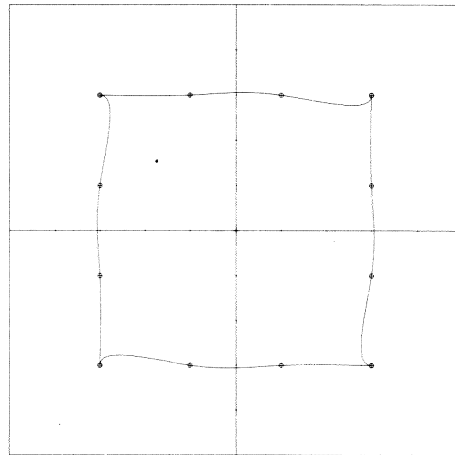


Fig. 9: Cyclic pp-curve (order 4) with four $(2,0)$ -anti-cyclic continuations (anti-clockwise)

REFERENCES

- [1] BOOR, C. DE, *A practical guide to splines*, Springer Verlag, 1978.
- [2] DAHLQUIST, G. a.o., *Numerical methods*, Prentice Hall, 1974.
- [3] STEWART, G.W., *Introductions to matrix computations*, Academic Press, 1973.
- [4] ROGERS, D.F. & J.A. ADAMS, *Mathematical elements for computer graphics*, McGraw-Hill, 1976.
- [5] KERNIGHAN, B.W. & D.M. RITCHIE, *The C programming language*, Prentice Hall, 1978.
- [6] HAGEN, P.J.W. TEN, a.o., *ILP-Intermediate language for pictures*, Mathematical Centre Tracts 130, Mathematisch Centrum, 1980.
- [7] RUSMAN, C.J., *B-spline algorithms*, Mathematisch Centrum rapport IN 18/80, Mathematisch Centrum, 1980.

APPENDIX

Remarks

1. The programs are written in C, an Algol/Pascal-like programming language developed by DENNIS RITCHIE at the Bell Laboratories, New Jersey. See for a description [5].
2. Some subroutines form part of the interface CILP with the graphical language ILP: 'pict', 'with', 'mdcontrol', 'draw', 'ward', 'scale', 'endpict', 'newpel' and 'line'. ILP is described in [6].
3. The pictures are drawn on a high-resolution display (HRD).
4. The contents of the appendix is enumerated a) - h) as follows:

a) 'testpr31.c'

with the subroutines

'ppcinc' pp-curve interpolation with complete pivoting in the cyclic case.
 'ppfint' pp-function interpolation
 'ludeco' LU-decomposition of banded b-matrix without pivoting
 'solsys' solution of system of equations without pivoting
 'bsplvx' b-spline values at x
 'ludecp' LU-decomposition of b-matrix with complete pivoting
 'solscp' solution of system of equations with complete pivoting
 'ppcpr' conversion of b-representation of pp-curve to pp-representation
 'ppfpr' conversion of b-representation of pp-function to pp-representation
 'plotpc' plotting pp-curve
 'plotdp' plotting datapoint-markers of pp-curve
 'plotkn' plotting knot-markers of pp-curve
 'axes2d' axes 2-dimensional.

- b) subroutine 'plotpf' plotting pp-function
- c) subroutine 'cubspl' cubic spline interpolation
- d) subroutine 'locadp' local adaption of pp-function
- e) subroutine 'ludecy' LU-decomposition cyclic without pivoting
- f) subroutine 'solsysc' solution of system of equations cyclic without pivoting
- g) subroutine 'valuex' value of (derivative of) pp-function at x
- h) subroutine 'interv' left endpoint of interval containing x.

APPENDIX A

```

1
2  #include <cilp.h>
3  #include <math.h>
4  #include <stdio.h>
5  #define MAXP 30
6  #define K 4
7  #define KP 5
8  #define KPKM1P 8
9  #define DIMP 3
10 #define DIM2P 5
11 double sqrt();
12 double sin();
13 double cos();
14 FILE *fd, *fopen();
15
16 main()
17 /* calls ppcinc ppcppr plotpc plotkn plotdp axes2d */
18 /* computes and plots pp-curves */
19
20 { double p[MAXP][DIMP], q[MAXP][MAXP], ql[MAXP][KPKM1P];
21   double bcoefp[DIMP][MAXP], t[MAXP], tau[MAXP];
22   double coefp[DIMP][MAXP][KP], breakp[MAXP];
23   double rscale[DIMP], origin[DIMP], pb[MAXP];
24   int dia;
25   int k, l, n, dim, cyc, ndist, plotc, numberstep;
26   fd = fopen("out", "w");
27   scanf(" %f %f", &rscale[1], &rscale[2]);
28   scanf(" %d", &dia);
29   k = K;
30   dim = DIMP - 1;
31   scanf("%d %d %d %d", &cyc, &ndist, &plotc, &numberstep);
32   ppcinc(dim, cyc, ndist, k, &n, p, q, ql, bcoefp, t, tau, pb);
33   ppcppr(dim, bcoefp, t, n, k, coefp, breakp, &l);
34   if (plotc == 1)
35   { pict(2, "pp-curve");
36     with(); scale(rscale[1], rscale[2]);
37     if (dia == 1)
38     { mdcontrol("HRD:diazo");
39       mdcontrol("HRD:feed");
40       mdcontrol("HRD:title:#pp-curve#");
41     }
42     draw();
43   }
44   plotpc(plotc, numberstep, dim, k, l, coefp, breakp, pb);
45   if (plotc == 1)
46   { plotkn(rscale, breakp, coefp, l, cyc, pb);
47     plotdp(rscale, p, n, cyc);
48     axes2d(rscale, origin, 20);

```

```

49     ward();
50     endpict();
51 }
52 }
53
54 int ppcinc(dim, cyc, ndist, k, n, p, q, ql, bcoefp, t, tau, pb)
55 /* calls ppfint, ludecp/ludepp and solscp/solspp */
56 /* reads points, continuation conditions and plot-instructions
57 * from an inputfile and calculates the b-coefficients of the
58 * pp-curve through the points, according to the given order,
59 * knot-distance option and cyc option.
60 * see for the format of the inputfile and the restrictions
61 * imposed on it, the following paper: 'the b-representation of
62 * piecewise polynomial parametric curves and local adaption', ch. 3.2.
63 * input : dim : dimension.
64 *         cyc : cyclic (cyc = 1) or not cyclic (cyc = 2).
65 *         ndist : tau is chord-distant (ndist = 1) or t is
66 *         equidistant (ndist = 2).
67 *         k : order.
68 * output : n : number of b-coefficients.
69 *         p[i][0] : pointcodes.
70 *         p[i][1],...,p[i][dim] : datapoints as read from file.
71 *         t : knotsequence.
72 *         tau : abscissae of the datapoints.
73 *         q : b-matrix, (n-k+1) x (n-k+1); used in case cyc = 1.
74 *         ql : condensed b-matrix, n x (2k-1); used in case cyc = 2.
75 *         bcoefp : matrix of b-coefficients, dim x n.
76 */
77
78 int dim, cyc, ndist, *n, k;
79 double p[MAXP][DIMP], q[MAXP][MAXP], ql[MAXP][KPKM1P];
80 double bcoefp[DIMP][MAXP], t[MAXP], tau[MAXP], pb[MAXP];
81 { double sumsq, dist, bcoef[MAXP], gtau[MAXP], tauii, dt;
82   int mult, ii, jj, i, j, m, nmkpl, mm, left, km2div2, tt;
83 /* m : current pointcode;
84 *   m = 0 : datapoint, not a knot.
85 *   m = 1 : floating knot (only if ndist = 2).
86 *   m = 2 : datapoint & knot.
87 *   m = 3 : anti-cyclic continuation.
88 *   mm = 0 : last read knot was datapoint
89 *   mm = 1 : last read knot was floating
90 */
91   for (i = 1; i < MAXP; ++i)
92     for (j = 1; j < MAXP; ++j)
93       q[i][j] = 0.0;
94   sumsq = 0.0;
95   dist = 0.0;
96   mult = 1;
97   ii = 0;
98   i = k;
99   pb[(tt = 1)] = -1.0;
100   scanf("%d", &m);
101   if (m == 1 || m == 2)
102     { t[k] = 0.0;
103       if (m == 2)
104         { for (j = 1, ++ii ; j <= dim; ++j)

```

```

105     scanf("%f", &p[l][j]);
106     p[l][0] = m;
107     tau[l] = t[k];
108     mm = 0;
109 }
110 else
111     mm = 1;
112     ++i;
113 }
114 else
115 { fprintf(fd, "\nwrong data(1)\n");
116     return(1);
117 }
118 scanf("%d", &m);
119 while (m <= 3)
120 { if (m == 3)
121     { for (j = 1; j <= 2; ++j)
122         { ++ii;
123             scanf(" %f ", &dt);
124             tau[ii] = t[i] = t[i-1] + dt;
125             if (t[i] > t[i-1]) pb[tt++] = t[i-1];
126             ++i;
127         }
128         p[ii-1][0] = 3.0;
129         p[ii][0] = 2.0;
130         for (jj = 1; jj <= dim; ++jj)
131             { p[ii-1][jj] = 0.0;
132                 p[ii][jj] = p[ii-2][jj];
133             }
134         mm = 0;
135         mult = 1;
136     }
137     else if (m != 1)
138     { for (j = 1, ++ii ; j <= dim; ++j)
139         { scanf("%f", &p[ii][j]);
140             if (ndist == 1)
141                 sumsq += (p[ii][j]-p[ii-1][j]) * (p[ii][j]-p[ii-1][j]);
142         }
143         p[ii][0] = m;
144         if (ndist == 1)
145             dist += sqrt(sumsq);
146         sumsq = 0.0;
147     }
148     if (m == 0)
149     /* datapoint, not a knot */
150     { if (ndist == 1)
151         tau[ii] = t[i-1] + dist;
152         ++mult;
153     }
154     else if (m == 1)
155     /* floating knot */
156     { if ((mult == 1 && mm == 1) || ndist == 1)
157         { fprintf(fd, "\nwrong data(2)\n");
158             return(1);
159         }
160         dist = 1.0 / mult;

```

```

161     tau[ii-mult+2] = t[i-1] + dist;
162     t[i] = t[i-1] + 1.0;
163     ++i;
164     for ( j = 3; j <= mult; ++j, ++i)
165     { tau[ii-mult+j] = tau[ii-mult+j-1] + dist;
166       t[i] = t[i-1];
167     }
168     if (mm == 0 && mult > 1)
169     { t[i] = t[i-1];
170       ++i;
171     }
172     mult = 1;
173     mm = 1;
174   }
175   else if (m == 2)
176   /* knot & datapoint */
177   { if (ndist == 1)
178     { tau[ii] = t[i] = t[i-1] + dist;
179       ++i;
180       dist = 0.0;
181     }
182     else
183     { if (mult == 1 && mm == 1)
184       { fprintf(fd, "\nwrong data(3)\n");
185         return(1);
186       }
187       dist = 1.0 / mult;
188       t[i] = t[i-1] + 1.0;
189       tau[ii-mult+1] = t[i-1] + dist;
190       ++i;
191       for (j = 2; j <= mult; ++j)
192         tau[ii-mult+j] = tau[ii-mult+j-1] + dist;
193     }
194     for (j = 3; j <= mult; ++j, ++i)
195     t[i] = t[i-1];
196     if (mm == 0 && mult > 1)
197     { t[i] = t[i-1];
198       ++i;
199     }
200     mult = 1;
201     mm = 0;
202   }
203   scanf("%d", &m);
204 }
205 if (t[i-1] < tau[ii])
206 { fprintf(fd, "\nwrong data(4)\n");
207   return(1);
208 }
209 *n = i - 2;
210 /* printing of the points after digesting */
211 for (j = 1; j <= *n + k; ++j)
212 { fprintf(fd, "%f   %f   ", t[j], tau[j]);
213   for (m = 0; m <= dim; ++m)
214     fprintf(fd, "%f   ", p[j][m]);
215   fprintf(fd, "\n");
216 }

```

```

217 fprintf(fd, "\n");
218 if (cyc == 2)
219 { km2div2 = (k - 2) / 2;
220   if (t[i-1] > tau[i]) --*n;
221   if (t[k+1] < t[k+1+km2div2] || t[*n+1-km2div2] < t[*n+1])
222   { fprintf(fd, "\nwrong data(5)\n"); return(1); }
223   for (i = 1; i <= *n - k - k + 3; ++i)
224     t[i+k] = t[i+k+km2div2];
225   *n -= k-2;
226 }
227 if (*n < 2 * k)
228 { fprintf(fd, "\nwrong data(6)\n");
229   return(1);
230 }
231 /* extension t */
232 for (i = 1; i < k; ++i)
233 { t[i] = t[k] - (t[*n+1] - t[*n+1-k+i]);
234   t[*n+1+i] = t[*n+1] + (t[k+i] - t[k]);
235 }
236 /* printing of the points after extension t */
237 for (i = 1; i <= *n + k; ++i)
238 { fprintf(fd, "%f %f ", t[i], tau[i]);
239   for (j = 0; j <= dim; ++j)
240     fprintf(fd, "%f ", p[i][j]);
241   fprintf(fd, "\n");
242 }
243 fprintf(fd, "\n");
244 if (cyc == 2)
245   for (j = 1; j <= dim; ++j)
246   { for (i = 1; i <= *n+1; ++i)
247     gtau[i] = p[i][j];
248     if (ppfint(tau, gtau, t, *n, k, ql, bcoefp[j]) == 2)
249       return(2);
250   }
251   else
252   { nmkpl = *n - k + 1;
253     /* fill matrix */
254     left = k;
255     for (ii = 1; ii <= nmkpl; ++ii)
256     { if (p[ii][0] != 3.0)
257       { tauii = tau[ii];
258         for (; tauii >= t[left+1]; ++left);
259         bsplvx(t, k, 1, tauii, left, bcoef);
260         for (i = left - k + 1, j = 1; j <= k; ++i, ++j)
261         { if (i > nmkpl) i -= nmkpl;
262           q[ii][i] += bcoef[j];
263         }
264       }
265     }
266     else
267     for (j = 1; j <= 2; ++j)
268     { bcoef[0] = bcoef[k-1] = 0.0;
269       bsplvx(t, k-1, 1, tauii, left, bcoef);
270       for (i = left - k + 2, jj = 1; jj <= k - 1; ++i, ++jj)
271       { bcoef[jj] /= (t[i+k-1] - t[i]);
272         if (i > nmkpl + 1) i -= nmkpl;
273         q[ii][i-1] -= (bcoef[jj-1] - bcoef[jj]) * (k - 1);

```



```

273     }
274     if (j == 1)
275     { if (t[left] < tau[i]) left += 3;
276       else if (t[left] == t[left-1]) ++left;
277       else left += 2;
278       tau[i] = t[left];
279     }
280   }
281 }
282 /* printing of the b-matrix before lu-decomposition */
283 for (j = 0; j <= nmkpl; j++)
284 { for (i = 0; i <= nmkpl; ++i)
285   { for (ii = j; ii <= j + 5 && ii <= nmkpl; ++ii)
286     fprintf(fd, " %f", q[i][ii]);
287     fprintf(fd, "\n");
288   }
289   fprintf(fd, "\n");
290   j += 6;
291 }
292 m = ludec(p, nmkpl);
293 /* printing of the b-matrix after lu-decomposition */
294 for (j = 0; j <= nmkpl; j++)
295 { for (i = 0; i <= nmkpl; ++i)
296   { for (ii = j; ii <= j + 5 && ii <= nmkpl; ++ii)
297     fprintf(fd, " %f", q[i][ii]);
298     fprintf(fd, "\n");
299   }
300   fprintf(fd, "\n");
301   j += 6;
302 }
303 if (m == 2)
304 { fprintf(fd, "\nb-matrix in ppcinc not invertible\n");
305   return(2);
306 }
307 for (j = 1; j <= dim; ++j)
308 { for (i = 1; i <= nmkpl; ++i)
309   bcoef[i] = p[i][j];
310   solscp(q, nmkpl, bcoef);
311   for (i = 1; i <= nmkpl; ++i)
312   { bcoefp[j][i] = bcoef[i];
313     if (i < k)
314       bcoefp[j][nmkpl+i] = bcoef[i];
315   }
316 }
317 }
318 return(0);
319 }
320
321 int ppfint(tau, gtau, t, n, k, q, bcoef)
322 /* calls bsplvx solsys luenco */
323 /* calculates the n b-coefficients (bcoef) of a pp-function with:
324  * knotsequence t,
325  * datapoints (tau[i], gtau[i]), i=1,...,n.
326  * order k.
327  * q is the condensed b-matrix (n x (2k-1)).
328  */

```

```

329 double tau[MAXP], gtau[MAXP];
330 double t[MAXP], q[MAXP][KPKM1P], bcoef[MAXP];
331 int n, k;
332 { int i, npl, ilplmx, j, jj, kml, left;
333   double tau_i;
334   npl = n+1;
335   kml = k-1;
336   left = k;
337   for (i = 1; i <= n; ++i)
338     for (j = 1; j <= k+kml; ++j)
339       q[i][j] = 0.0;
340   for (i = 1; i <= n; ++i)
341     { tau_i = tau[i];
342       ilplmx = ((i + k) < npl) ? (i + k) : npl;
343       left = (left > i) ? left : i;
344       if (tau_i < t[left])
345         { fprintf(fd, "b-matrix in ppfint not invertible (1)\n");
346           return(2);
347         }
348       do
349         { if (tau_i < t[left + 1]) break;
350           else ++left;
351         }
352       while (left < ilplmx);
353       if (left >= ilplmx)
354         { --left;
355           if (tau_i > t[left+1])
356             { fprintf(fd, "b-matrix in ppfint not invertible (2)\n");
357               return(2);
358             }
359         }
360       bsplvx(t, k, 1, tau_i, left, bcoef);
361       for (j = left-i+1, jj = 1; jj <= k; ++j, ++jj)
362         q[i][jj] = bcoef[jj];
363     }
364   if (ludeco(q, k, n) == 2)
365     { fprintf(fd, "b-matrix in ppfint not invertible (3)\n");
366       return(2);
367     }
368   for (i = 1; i <= n; ++i)
369     bcoef[i] = gtau[i];
370   solsys(q, k, n, bcoef);
371   return(1);
372 }
373
374 int ludeco(q, k, n)
375 /* lu-decomposition of banded n x n matrix, bandwidth 2k-1,
376  * without pivoting; the band is stored in q (n x (2k-1)).
377  */
378 double q[MAXP][KPKM1P];
379 int k, n;
380 { int l, nr, i, j, jj, kml, ipl;
381   double qik;
382   kml = k-1;
383   for (i = 1; i <= n; ++i)
384     { l = k;

```

```

385     ipl = i+1;
386     nr = ((nr = i+kml) <= n) ? nr : n;
387     if ((qik = q[i][k]) == 0) return(2);
388     for (j = ipl; j <= nr; ++j)
389     { l -= 1;
390       if (q[j][l] != 0.0)
391       { q[j][l] /= qik;
392         for (jj = 1; jj <= kml; ++jj)
393           q[j][l+jj] -= q[j][l] * q[i][k+jj];
394       }
395     }
396   }
397   return(1);
398 }
399
400 solsys(q, k, n, bcoef)
401 /* solves a system of equations;
402  * to be used with ludeco;
403  * the right-values are expected in bcoef, the solution
404  * is put back in bcoef.
405  */
406 double q[MAXP][KPKMLP], bcoef[MAXP];
407 int k, n;
408 { int l, i, j, kml, kpl, kpkml;
409   kml = k-1;
410   kpl = k+1;
411   kpkml = k+kml;
412   /* forward step */
413   for (i = 2; i <= n; ++i)
414   { l = (k > i)? (k-i+1) : 1;
415     for (j = 1; j <= kml; ++j)
416       bcoef[i] -= q[i][j] * bcoef[j-k+i];
417   }
418   /* backward step */
419   for (i = n; i >= 1; --i)
420   { l = (n-i < k)? (n-i+k) : kpkml;
421     for (j = 1; j >= kpl; --j)
422       if (q[i][j] != 0.0)
423         bcoef[i] -= q[i][j] * bcoef[i+j-k];
424     bcoef[i] /= q[i][k];
425   }
426 }
427
428 bsplvx(t, jhigh, indexx, x, left, biatx)
429 /* calculates the values of all possibly non-zero b-splines
430  * at x of order
431  *     jhigh, if index = 1,
432  *     max(jhigh, j+1), if index = 2.
433  * further input: t, the knot sequence; left, an integer,
434  * such that t[left] <= x < t[left+1].
435  * if index = 1, the calculation starts from the beginning
436  * (i.e. with order = 1);
437  * if index = 2, the calculation continues where it left of.
438  * The value of j and the auxiliary arrays deltal and
439  * deltar are therefore saved.
440  * output: biatx[i], 1 <= i <= order, with

```

```

441     * biatx[i] = b[left-order+i][order](x).
442     */
443     double t[MAXP], x, biatx[KP];
444     int jhigh, indexx, left;
445     { static double deltal[20], deltar[20];
446         static int j;
447         int i, jpl;
448         double saved, term;
449         if (indexx == 1)
450             { j = 1;
451               biatx[1] = 1;
452             }
453         if ((indexx == 1 && j < jhigh) || indexx == 2)
454             do
455             { jpl = j + 1;
456               deltar[j] = t[left+j] - x;
457               deltal[j] = x - t[left+1-j];
458               saved = 0.0;
459               for (i = 1; i <= j; ++i)
460               { term = biatx[i] / (deltar[i] + deltal[jpl-i]);
461                 biatx[i] = saved + deltar[i] * term;
462                 saved = deltal[jpl-i] * term;
463               }
464               biatx[jpl] = saved;
465               j = jpl;
466             }
467             while (j < jhigh);
468     }
469
470     int ludecp(q, n)
471     /* lu-decomposition with complete pivoting of an n x n matrix q;
472      * the permutation to be performed on the input (output)
473      * is kept in the 0-th column (row).
474      */
475     double q[MAXP][MAXP];
476     int n;
477     { int i, j, pc, pr, ii, ipl;
478       double pivot, hulp, abspiv, absq;
479       q[1][0] = q[0][1] = 1.0;
480       for (i = 2; i <= n; ++i)
481         q[i][0] = q[0][i] = q[0][i-1] + 1.0;
482       for (i = 1; i <= n; ++i)
483       { ipl = i + 1;
484         pivot = q[i][i];
485         abspiv = (pivot > 0.0)? pivot : -pivot;
486         pr = pc = i;
487         for (ii = i; ii <= n; ++ii)
488           for (j = i; j <= n; ++j)
489             { absq = (q[ii][j] > 0.0)? q[ii][j] : -q[ii][j];
490               if (absq > abspiv)
491               { pc = j;
492                 pr = ii;
493                 pivot = q[ii][j];
494                 abspiv = (pivot > 0.0)? pivot : -pivot;
495               }
496             }

```

```

497     if (pivot == 0.0) return(2);
498     if (pc != i)
499     for (j = 0; j <= n; ++j)
500     { hulp = q[j][i];
501       q[j][i] = q[j][pc];
502       q[j][pc] = hulp;
503     }
504     if (pr != i)
505     for (j = 0; j <= n; ++j)
506     { hulp = q[i][j];
507       q[i][j] = q[pr][j];
508       q[pr][j] = hulp;
509     }
510     for (j = ipl; j <= n; ++j)
511     if (q[j][i] != 0.0)
512     { q[j][i] /= pivot;
513       for (ii = ipl; ii <= n; ++ii)
514       q[j][ii] -= q[j][i] * q[i][ii];
515     }
516   }
517   return(1);
518 }
519
520 solscp(q, n, bcoef)
521 /* solves a system of equations;
522  * to be used with ludecp;
523  * the right-values are expected in bcoef, the solution
524  * is put in bcoef again.
525  */
526 double q[MAXP][MAXP], bcoef[MAXP];
527 int n;
528 { int i, j;
529   double hulp[MAXP];
530   for (i = 1; i <= n; ++i)
531     hulp[i] = bcoef[(int) q[i][0]];
532   for (i = 1; i <= n; ++i)
533     bcoef[i] = hulp[i];
534   for (i = 2; i <= n; ++i)
535     for (j = 1; j < i; ++j)
536     if (q[i][j] != 0.0)
537     bcoef[i] -= q[i][j] * bcoef[j];
538   for (i = n; i >= 1; --i)
539   { for (j = n; j > i; --j)
540     if (q[i][j] != 0.0)
541     bcoef[i] -= q[i][j] * bcoef[j];
542     bcoef[i] /= q[i][i];
543   }
544   for (i = 1; i <= n; ++i)
545     hulp[(int) q[0][i]] = bcoef[i];
546   for (i = 1; i <= n; ++i)
547     bcoef[i] = hulp[i];
548 }
549
550 ppcppr(dim, bcoefp, t, n, k, coefp, breakp, 1)
551 /* calls ppfprr */
552 /* computes the pp-representation of a pp-curve starting from a

```

```

553  * b-representation.
554  * input : dim : dimension.
555  *         bcoefp : b-coefficients
556  *         t : knotsequence.
557  *         n : number of b-coefficients.
558  *         k : the order.
559  * output : coefp : pp-coefficients (dim x 1 x k)
560  *         breakp : sequence of breakpoints.
561  *         l : number of intervals.
562  */
563  int n, k, *l, dim;
564  double t[MAXP], coefp[DIMP][MAXP][KP], breakp[MAXP];
565  double bcoefp[DIMP][MAXP];
566  { int i;
567    for (i = 1; i <= dim; ++i)
568      ppfprr(t, bcoefp[i], n, k, breakp, coefp[i], 1);
569  }
570
571  ppfprr(t, bcoef, n, k, breakp, coef, 1)
572  /* calls bsplvx */
573  /* computes the pp-representation of a pp-function starting
574  * from a b-representation;
575  * input : t : knotsequence.
576  *         bcoef : b-coefficients.
577  *         k : the order.
578  *         n : number of b-coefficients.
579  * output : breakp : sequence of breakpoints.
580  *         coef : the pp-coefficients.
581  *         l : number of intervals.
582  */
583  int n, k, *l;
584  double t[MAXP], bcoef[MAXP], breakp[MAXP], coef[MAXP][KP];
585  { double scrtch[KP][KP], diff, sum, biatx[KP];
586    int left, lsofar, i, j, kmj, jpl;
587    lsofar = 0;
588    breakp[1] = t[k];
589    for (left = k; left <= n; ++left)
590      if (t[left+1] > t[left])
591        { lsofar += 1;
592          breakp[lsofar+1] = t[left+1];
593          if (k == 1)
594            coef[lsofar][1] = bcoef[left];
595          else
596            { for (i = 1; i <= k; ++i)
597              scrtch[i][1] = bcoef[left-k+i];
598              for (jpl = 2; jpl <= k; ++jpl)
599                { j = jpl - 1;
600                  kmj = k - j;
601                  for (i = 1; i <= kmj; ++i)
602                    { diff = t[left+i] - t[left+i-kmj];
603                      if (diff > 0.0)
604                        scrtch[i][jpl] = ((scrtch[i+1][j] - scrtch[i][j])/diff)*kmj;
605                    }
606                }
607              bsplvx(t, 1, 1, t[left], left, biatx);
608              coef[lsofar][k] = scrtch[1][k];

```

```

609     for (jpl = 2; jpl <= k; ++jpl)
610     { bsplvx(t, jpl, 2, t[left], left, biatx);
611       kmj = k + 1 - jpl;
612       sum = 0.0;
613       for (i = 1; i <= jpl; ++i)
614       sum += biatx[i] * scrtch[i][kmj];
615       coef[lsofar][kmj] = sum;
616     }
617   }
618 }
619 *l = lsofar;
620 }
621
622 plotpc(plotc, numberstep, dim, k, 1, coefp, breakp, pb)
623 /* plots a pp-curve;
624  * plotc = 1 : the calculated (points on the) curve is (are)
625  *           given to a display.
626  * plotc = 2 : the points are put on a file.
627  * numberstep : steps per interval.
628  * pb : values of the left endpoints of intervals to be skipped.
629  */
630 double coefp[DIMP][MAXP][KP], breakp[MAXP], pb[MAXP];
631 int plotc, numberstep, dim, k, 1;
632 { double parpp[DIMP], h, dh;
633   int i, j, m, jj, tt;
634   if (plotc == 1 && dim != 2)
635   { fprintf(fd, "\nplotting only if dim = 2\n");
636     return;
637   }
638   if (plotc == 2)
639   { fprintf(fd, "\n          t");
640     for (i = 1; i <= dim; ++i)
641       fprintf(fd, "          p%ld(t)", i);
642     fprintf(fd, "\n\n");
643   }
644   tt = 1;
645   for (i = 1; i <= 1; ++i)
646   if (breakp[i] == pb[tt]) ++tt;
647   else
648   { dh = (breakp[i+1] - breakp[i]) / numberstep;
649     h = 0.0;
650     for (jj = 0; jj <= numberstep; ++jj)
651     { for (j = 1; j <= dim; ++j)
652       { parpp[j] = 0.0;
653         for (m = k; m >= 1; --m)
654           parpp[j] = (parpp[j] / m) * h + coefp[j][i][m];
655       }
656       if (plotc == 1)
657         line(parpp[1], parpp[2]) ;
658       else if (plotc == 2)
659       { fprintf(fd, " %10.6f", breakp[i] + h);
660         for (j = 1; j <= dim; ++j)
661           fprintf(fd, " %10.6f", parpp[j]);
662         fprintf(fd, "\n");
663       }
664       h += dh;

```

```

665     }
666   }
667   for (i = 1; i < DIMP; ++i)
668     coefp[i][1+1][1] = parpp[i];
669 }
670
671 plotdp(rscale, p, n, cyc)
672 /* plots the datapoint-markers (little crosses) of a pp-curve;
673  * rscale : scale factors.
674  */
675 double p[MAXP][DIMP], rscale[DIMP];
676 int n, cyc;
677 { int i;
678   double dpx, dpy, p1, p2;
679   dpx = 2.0 / (rscale[1] * 200);
680   dpy = 2.0 / (rscale[2] * 200);
681   if (cyc == 2) ++n;
682   for (i = 1; i <= n; ++i)
683     if (p[i][0] == 3) ++i;
684   else
685     { p1 = p[i][1];
686       p2 = p[i][2];
687       newpel();
688       line(p1, p2 - dpy);
689       line(p1, p2 + dpy);
690       newpel();
691       line(p1 - dpx, p2);
692       line(p1 + dpx, p2);
693     }
694 }
695
696 plotkn(rscale, breakp, coefp, 1, cyc, pb)
697 /* plots the knot-markers (little circles) of a pp-curve;
698  * rscale : scale factors.
699  */
700 double coefp[DIMP][MAXP][KP], breakp[MAXP], rscale[DIMP];
701 double pb[MAXP];
702 int l, cyc;
703 { int i, j, tt;
704   double dsx, dsy, px, py, dth;
705   tt = 1;
706   dth = 3.1416 / 8;
707   dsx = 2.0 / rscale[1];
708   dsy = 2.0 / rscale[2];
709   for (i = 1; i <= 1 + 1; ++i)
710     if (pb[tt] == breakp[i]) ++tt;
711   else
712     { if ((i == 1+1) && (cyc == 1)) return;
713       px = coefp[1][i][1];
714       py = coefp[2][i][1];
715       newpel();
716       for (j = 0; j <= 16; ++j)
717         line(px + dsx*sin(j*dth)/200, py + dsy*cos(j*dth)/200);
718     }
719 }
720

```



```

721 axes2d(rscale, or, nstr)
722 /* plots axes with origin (or[1], or[2]) and scale factors
723    * rscale[1] and rscale[2];
724    * approximately nstr points p[i] are indicated on the
725    * axes, such that  $p[i] = l[i] * 10^k$ ,  $l[i]$  and  $k$  integers.
726    */
727 int nstr;
728 double or[DIMP], rscale[DIMP];
729 { int i, j;
730   double w[DIMP], fac;
731   for (i = 1; i < DIMP; ++i)
732     w[i] = or[i];
733   for (i = 1; i < DIMP; ++i)
734     { w[i] = 1.0 / (rscale[i] * nstr);
735       j = 0;
736       fac = 1.0;
737       while (fac > w[i])
738         { --j; fac /= 10; }
739       while (fac < w[i])
740         { ++j; fac *= 10; }
741       newpel();
742       w[i] = -1.0 / rscale[i];
743       line(w[1], w[2]);
744       w[0] = w[i] = -w[i];
745       line(w[1], w[2]);
746       while (w[i] > 0.0) w[i] -= fac;
747       w[i % 2 + 1] += 1.0 / (rscale[i % 2 + 1] * 200);
748       for (w[i] = -w[0] + fac + w[i]; w[i] < w[0]; w[i] += fac)
749         { newpel();
750           line(w[1], w[2]);
751           w[i % 2 + 1] -= 1.0 / (rscale[i % 2 + 1] * 100);
752           line(w[1], w[2]);
753           w[i % 2 + 1] += 1.0 / (rscale[i % 2 + 1] * 100);
754         }
755       w[i] = or[i];
756     }
757   newpel();
758   for (i = 1; i < DIMP; ++i)
759     w[i] = 1.0 / rscale[i];
760   line(-w[1], -w[2]);
761   line(-w[1], w[2]);
762   line(w[1], w[2]);
763   line(w[1], -w[2]);
764   line(-w[1], -w[2]);
765 }

```

APPENDIX B

Subroutine 'plotpf'

This subroutine plots a pp-function. There are three options:

- 1) For each interval a fixed number (numberstep +1) of points is evaluated:

$$\left(\xi_j + \frac{\xi_{j+1} - \xi_j}{\text{numberstep}} \cdot i, P_j \left(\xi_j + \frac{\xi_{j+1} - \xi_j}{\text{numberstep}} \cdot i \right) \right), i = 0, \dots, \text{numberstep}.$$

The points are put on a file (plotcode 3).

- 2) Idem.

The points are given to a display, which draws straight lines between them (plotcode 2).

- 3) The distribution of the points is adapted to the curvature of the function (plotcode 1).

If the function is plotted with (not too many) mpoints equidistant on each interval, then, on places where the function has a strong curvature, it will show, that the plotting is done by drawing little straight lines.

Two parameters are given to the procedure

maxdev: maximum deviation in slope (in rad):

$$|\text{atan}(P_j^{(1)}(x_{i+1})) - \text{atan}(P_j^{(1)}(x_i))| \leq \text{maxdev},$$

x_i, x_{i+1} two consecutive abscissae.

Consequently, the angle between two consecutive lines lies between -2 maxdev and $+2 \text{ maxdev}$, if $P_j^{(2)}(x) \neq 0$ on the intervals corresponding with the lines.

maxdh: $|x_{i+1} - x_i| \leq \text{maxdh}$.

Suppose we have reached a point $(x_i, P_j(x_i))$ in the plotting process.

$dh := x_i - x_{i-1}$ (if $x_i = \xi_j$ we take $dh := \text{maxdh}$). If $|\text{atan}(P_j^{(1)}(x_{i+dh})) - \text{atan}(P_j^{(1)}(x_i))| > \text{maxdev}$, the value of dh is altered: $dh := \frac{1}{2}dh$, else $dh := 2dh$.

Again we look at the angle. If the $>$ -sign changes into \leq , a line is drawn between $(x_i, P_j(x_i))$ and $(x_{i+1}, P_j(x_{i+1}))$, $x_{i+1} = x_i + dh$.

If the \leq -sign changes into $>$, a line is drawn between $(x_i, P_j(x_i))$ and

$(x_{i+1}, P_j(x_{i+1})), x_{i+1} = x_i + \frac{1}{2}dh.$

If there is no change of sign, the halving or doubling of dh continues until the sign changes.

Of course, provisions are made in case $dh > \max dh$, or $x_i + dh > \xi_{j+1}$.

Figure 1 is drawn by this procedure. The numbers of steps per

ξ -interval turned out to be: 139, 37, 75, 51, 46, 51 and 96

($\max dh = 0.1$, $\max dev = 0.05$).

```

1
2 plotpf(plotcode, numberstep, maxdh, maxdev, k, 1, coef, breakp)
3 /* plots a pp-function.
4  * input : k, 1, coef and breakp : the pp-representation
5  *
6  *          numberstep : steps per interval (only used if
7  *          plotcode = 2 or plotcode = 3).
8  *          maxdh : maximum step-width (only used if plotcode = 1).
9  *          maxdev : maximum value of the angle (in rad) between
10 *          the slopes of the curve at two consecutive points
11 *          (only used if plotcode = 1).
12 *          plotcode = 3 : (numberstep + 1) equidistant points per
13 *          interval are evaluated and put on a file.
14 *          plotcode = 2 : idem, the points are given to a display
15 *          (which draws straight lines between them).
16 *          plotcode = 1 : the distances between the points depend
17 *          on the curvature of the function. The points are
18 *          given to a display.
19 */
20 int k, 1, plotcode, numberstep;
21 double maxdh, maxdev, coef[MAXP][KP], breakp[MAXP];
22 { int i, j, pc, a, nl, m;
23   double h, dh, ppx, ppf, ppfa, darc, darcn, darct;
24   if (plotcode == 1)
25     for (i = 1; i <= 1; ++i)
26     { h = 0.0; dh = maxdh; pc = 2; a = 0;
27       nl = 0;
28       ppf = coef[i][1];
29       darc = atan(coef[i][2]);
30       newpel();
31       line(breakp[i], ppf);
32       while (breakp[i] + h < breakp[i+1])
33       { if (a++ > 1000)
34         { fprintf(fd, "\n while in loop\n"); return; }
35         ppfa = 0.0;
36         for (j = k; j >= 2; --j)
37           ppfa = (ppfa / (j-1)) * (h + dh) + coef[i][j];
38         darcn = atan(ppfa);
39         if (((darcn > darc) ? (darcn - darc) : (darc - darcn)) > maxdev)
40         { dh /= 2;
41           if (pc == 0)
42           { h += dh;
43             ppf = 0.0;
44             for (j = k; j >= 1; --j)
45               ppf = (ppf / j) * h + coef[i][j];
46             darc = darct;

```

```

47         pc = 2;
48         line(breakp[i] + h, ppf);
49         ++nl;
50     }
51     else pc = 1;
52 }
53 else
54 { if (pc==1 || dh>=maxdh || breakp[i]+h+dh>=breakp[i+1])
55     { if (breakp[i] + h + dh >= breakp[i+1])
56         h = breakp[i+1] - breakp[i];
57         else
58             h += dh;
59         ppf = 0.0;
60         for (j = k; j >= 1; --j)
61             ppf = (ppf / j) * h + coef[i][j];
62         darc = darcn;
63         line(breakp[i] + h, ppf);
64         ++nl;
65         pc = 2;
66     }
67     else
68     { pc = 0;
69       if (dh < maxdh) dh *= 2;
70     }
71 }
72 darct = darcn;
73 }
74 fprintf(fd, "      %4d\n", nl);
75 }
76 else
77 for (i = 1; i <= 1; ++i)
78 { dh = (breakp[i+1] - breakp[i]) / numberstep;
79   h = 0.0;
80   if (plotcode == 2) newpel();
81   for (m = 0; m <= numberstep; ++m)
82   { ppf = 0.0;
83     for (j = k; j >= 1; --j)
84         ppf = (ppf / j) * h + coef[i][j];
85     ppx = breakp[i] + h;
86     if (plotcode == 2)
87         line(ppx, ppf);
88     else if (plotcode == 3)
89         fprintf(fd, "\n %10.6f  %10.6f", ppx, ppf);
90     h += dh;
91   }
92 }
93 coef[l+1][1] = ppf;
94 }

```

APPENDIX C

```

1
2  cubspl(tau, c, n)
3  /* cubic spline interpolation.
4   * input: tau[i], 1 <= i <= n, the knots,
5   *        (tau[i], c[i][1]), the given points,
6   *        n, number of points (n >= 3).
7   * output: c[i][2], c[i][3] and c[i][4], 1 <= i <= n-1,
8   *        the values of the first, second and third
9   *        derivatives at the left end points.
10  * for tau[i] <= x <= tau[i+1] we have
11  *      f(x) (= p[i](x)) = c[i][1] + (x - tau[i]) * c[i][2] +
12  *      + 1/2 * (x - tau[i])^2 * c[i][3] +
13  *      + 1/6 * (x - tau[i])^3 * c[i][4].
14  * the not-a-knot boundary condition is being used.
15  */
16  int n;
17  double tau[MAXP], c[MAXP][KP];
18  { int l, m;
19    double g, dtau, divdf3, divdf1;
20    l = n-1;
21    for (m = 2; m <= n; ++m)
22    { c[m][3] = tau[m] - tau[m-1];
23      c[m][4] = (c[m][1] - c[m-1][1]) / c[m][3];
24    }
25  /* calculation of the diagonal- (c[m][4]), next-to-diagonal-
26  * (c[m][3]) and right-elements (c[m][2]) together with the
27  * forward step of the Gauss-elimination.
28  */
29  c[1][4] = c[3][3];
30  c[1][3] = c[2][3] + c[3][3];
31  c[1][2] = (c[2][3] + 2 * c[1][3]) * c[2][4] * c[3][3];
32  c[1][2] += c[2][3] * c[2][3] * c[3][4];
33  c[1][2] /= c[1][3];
34  for (m = 2; m <= 1; ++m)
35  { g = (-c[m+1][3]) / c[m-1][4];
36    c[m][2] = g * c[m-1][2];
37    c[m][2] += 3*(c[m][3] * c[m+1][4] + c[m+1][3] * c[m][4]);
38    c[m][4] = g * c[m-1][3] + 2 * (c[m][3] + c[m+1][3]);
39  }
40  g = c[n-1][3] + c[n][3];
41  c[n][2] = (c[n][3] + 2 * g) * c[n][4] * c[n-1][3];
42  c[n][2] += c[n][3] * c[n][3] * (c[n-1][1] - c[n-2][1]);
43  c[n][2] /= c[n-1][3] * g;
44  /* c[n-1][4] was already overwritten */
45  c[n][4] = c[n-1][3];
46  g = (-g) / c[n-1][4];
47  c[n][4] = g * c[n-1][3] + c[n][4];
48  c[n][2] = (g * c[n-1][2] + c[n][2]) / c[n][4];

```

```
49  /* completion Gauss-elimination */
50  for (m = 1; m >= 1; --m)
51    c[m][2] = (c[m][2] - c[m][3] * c[m+1][2]) / c[m][4];
52  /* calculation of the functionvalues of the second and third
53  * derivatives in the left endpoints
54  */
55  for (m = 2; m <= n; ++m)
56  { dtau = c[m][3];
57    divdf1 = (c[m][1] - c[m-1][1]) / dtau;
58    divdf3 = c[m-1][2] + c[m][2] - 2 * divdf1;
59    c[m-1][3] = 2 * (divdf1 - c[m-1][2] - divdf3) / dtau;
60    c[m-1][4] = (divdf3 / dtau) * 6.0 / dtau;
61  }
62 }
```

APPENDIX D

```

1
2  locadp(x, fx, adapcode, t, n, k, bcoef, q, tau, gtau)
3  /* calls interv solsys valuex */
4  /* adapts (a b-representation of) a pp-function to a new
5   * function value.
6   * input : (x, fx) : new point.
7   *          t, n, k and bcoef : b-representation of the
8   *          pp-function.
9   *          q : the condensed b-matrix.
10  *          (tau[i], gtau[i]) : the underlying datapoints.
11  *          adapcode : adaption-code.
12  *          adapcode = 0 : x = tau[j] and the other
13  *          datapoints are kept unchanged (if adapcode = 0
14  *          and x != tau[j], 'locadp' will protest).
15  *          adapcode = -1 : two consecutive b-coefficients
16  *          change.
17  *          adapcode >= 1 : (k-2+adapcode) consecutive
18  *          b-coefficients change.
19  * (see for a description: 'the b-representation of piecewise
20  * polynomial parametric curves and local adaption', ch. 2.2)
21  */
22  double x, fx, t[MAXP], q[MAXP][KPKM1P], bcoef[MAXP];
23  double tau[MAXP], gtau[MAXP];
24  int n, k, adapcode;
25  { int i, left, tau_i, right;
26    double bsplc[MAXP], der, derpr, diff, bval, bvalpr;
27    double valuex();
28    interv(t, n, x, &left);
29    if (x < t[k] || x > t[n+1])
30    { printf("\n(%f, %f) not in interval", x, fx);
31      printf(" [%f, %f]\n", t[k], t[n+1]);
32      return;
33    }
34    if (adapcode == 0)
35    { interv(tau, n, x, &tau_i);
36      if (x != tau[tau_i])
37      { printf("\nadapcode = 0 en x != tau[i]\n");
38        return;
39      }
40      gtau[tau_i] = fx;
41      for (i = 1; i <= n; ++i)
42        bcoef[i] = gtau[i];
43      solsys(q, k, n, bcoef);
44    }
45    else if (adapcode == -1)
46    { for (i = 1; i <= n; ++i)
47        bsplc[i] = 0.0;
48      for (i = left-k+1; i <= left; ++i)

```

```

49     { bsplc[i] = 1.0;
50       der = valuex(t, bsplc, n, k, x, 1);
51       if (der > 0.0) break;
52       derpr = der;
53       bsplc[i] = 0.0;
54     }
55     diff = fx - valuex(t, bcoef, n, k, x, 0);
56     bval = valuex(t, bsplc, n, k, x, 0);
57     bsplc[i] = 0.0;
58     bsplc[i-1] = 1.0;
59     bvalpr = valuex(t, bsplc, n, k, x, 0);
60     printf("\n %f %f %f %f \n", bval, bvalpr, der, derpr);
61     if (der < derpr)
62     { diff /= (bval - (der / derpr) * bvalpr);
63       bcoef[i-1] -= diff * (der / derpr);
64       bcoef[i] += diff;
65     }
66     else
67     { diff /= (bvalpr - (derpr / der) * bval);
68       bcoef[i-1] += diff;
69       bcoef[i] -= diff * (derpr / der);
70     }
71     for (left = 1; tau[left] <= t[i-1]; ++left);
72     for (right = n; tau[right] >= t[i+k]; --right);
73     for (i = left; i <= right; ++i)
74       gtau[i] = valuex(t, bcoef, n, k, tau[i], 0);
75   }
76   else
77   { diff = fx - valuex(t, bcoef, n, k, x, 0);
78     if (x == t[left] && adapcode > 1) --left;
79     i = left;
80     adapcode -= 2;
81     left = left - k + 1 - (adapcode / 2);
82     left = (left < 1) ? 1 : left;
83     right = i + adapcode - (adapcode / 2);
84     right = (right > n) ? n : right;
85     for (i = left; i <= right; ++i)
86       bcoef[i] += diff;
87     for (i = 1; tau[i] <= t[left]; ++i);
88     left = i;
89     for (i = n; tau[i] >= t[right+k]; --i);
90     right = i;
91     for (i = left; i <= right; ++i)
92       gtau[i] = valuex(t, bcoef, n, k, tau[i], 0);
93   }
94 }

```


APPENDIX E

```

1
2  int ludecy(q1, q2, k, n)
3  /* lu-decomposition without pivoting of an n x n b-matrix, which
4   * is banded (bandwidth 2k-1) in the first n-k+1 rows (condensed
5   * in q1); the last k-1 rows are stored in q2.
6   */
7  double q1[MAXP][KPKM1P], q2[K][MAXP];
8  int k, n;
9  { int kpl, kml, nmkpl, i, ipkml, ipl, m, j, jj, l, nr, ipnmkpl;
10   double pivot;
11   kpl = k + 1;
12   kml = k - 1;
13   nmkpl = n - k + 1;
14   for (m = 1; m <= kml && q2[m][1] == 0.0; ++m);
15   /* m kan k zijn */
16   for (i = 1; i <= nmkpl; ++i)
17   { l = k;
18     ipkml = i + k - 1;
19     ipl = i + 1;
20     nr = (ipkml <= nmkpl)? ipkml : nmkpl;
21     if ((pivot = q1[i][k]) == 0.0) return(2);
22     for (j = ipl; j <= nr; ++j)
23     { l -= 1;
24       if (q1[j][l] != 0.0)
25       { q1[j][l] /= pivot;
26         for (jj = 1; jj <= kml; ++jj)
27           q1[j][l+jj] -= q1[j][l] * q1[i][k+jj];
28       }
29     }
30     for (j = m; j <= kml; ++j)
31     { q2[j][i] /= pivot;
32       for (jj = ipl, l = kpl; jj <= ipkml; ++jj, ++l)
33         q2[j][jj] -= q2[j][i] * q1[i][l];
34     }
35   }
36   for (i = 1; i <= kml; ++i)
37   { ipnmkpl = i + nmkpl;
38     if ((pivot = q2[i][ipnmkpl]) == 0.0) return(2);
39     ipl = i + 1;
40     for (j = ipl; j <= kml; ++j)
41     if (q2[j][ipnmkpl] != 0.0)
42     { q2[j][ipnmkpl] /= pivot;
43       for (jj = nmkpl + ipl; jj <= n; ++jj)
44         q2[j][jj] -= q2[j][ipnmkpl] * q2[i][jj];
45     }
46   }
47   return(1);
48 }

```

APPENDIX F

```

1
2  solsys(q1, q2, k, n, bcoef)
3  /* solves a system of equations;
4   * to be used with ludacy;
5   * the right-values are expected in bcoef, the solution is put
6   * in bcoef again.
7   */
8  double q1[MAXP][KPKM1P], q2[K][MAXP], bcoef[MAXP];
9  int k, n;
10 { int kml, nmkpl, l, i, j, ipnmkpl, jmkpipnmkpl, kpl, kpkml;
11     kml = k-1;
12     nmkpl = n - kml;
13     kpkml = k + kml;
14     kpl = k + 1;
15     for (i = 2; i <= nmkpl; ++i)
16     { l = (k > i) ? (kpl - i) : 1;
17       for (j = 1; j <= kml; ++j)
18         if (q1[i][j] != 0.0)
19           bcoef[i] -= q1[i][j] * bcoef[j-k+i];
20     }
21     for (i = 1; i <= kml; ++i)
22     { ipnmkpl = i + nmkpl;
23       if (q2[i][1] == 0.0)
24         for (j = 1; j <= kml; ++j)
25         { jmkpipnmkpl = ipnmkpl - k + j;
26           if (q2[i][jmkpipnmkpl] != 0.0)
27             bcoef[ipnmkpl] -= q2[i][jmkpipnmkpl] * bcoef[jmkpipnmkpl];
28         }
29       else
30         for (j = 1; j < ipnmkpl; ++j)
31           bcoef[ipnmkpl] -= q2[i][j] * bcoef[j];
32     }
33     for (i = kml; i >= 1; --i)
34     { ipnmkpl = i + nmkpl;
35       for (j = n; j > ipnmkpl; --j)
36         if (q2[i][j] != 0.0) bcoef[ipnmkpl] -= q2[i][j] * bcoef[j];
37       bcoef[ipnmkpl] /= q2[i][ipnmkpl];
38     }
39     for (i = nmkpl; i >= 1; --i)
40     { for (j = kpkml; j >= kpl; --j)
41       if (q1[i][j] != 0.0) bcoef[i] -= q1[i][j] * bcoef[i+j-k];
42       bcoef[i] /= q1[i][k];
43     }
44 }

```

APPENDIX G

```

1
2  double valuex(t, bcoef, n, k, x, jderiv)
3  /* calls interv */
4  /* returns the value at x of the jderiv-th derivative of the
5   * the pp-function with
6   *         order k,
7   *         knot sequence t[i], 1 <= i <= n+k,
8   *         b-coefficients bcoef[j], 1 <= j <= n.
9   */
10 double t[MAXP], bcoef[MAXP], x;
11 int n, k, jderiv;
12 { double dl[KP], dr[KP], aj[KP];
13   int kml, jcmn, imk, i, j, jcmax, nmi, jc, kmj, ilo, jj;
14   if (jderiv >= k) return(0.0);
15   if (interv(t, n+k, x, &i) != 0) return(0.0);
16   kml = k-1;
17   if (kml == 0) return(bcoef[i]);
18   jcmn = 1;
19   imk = i-k;
20   if (imk >= 0)
21     for (j = 1; j <= kml; ++j)
22       dl[j] = x - t[i+1-j];
23   else
24     { jcmn = 1 - imk;
25       for (j = 1; j <= i; ++j)
26         dl[j] = x - t[i+1-j];
27       for (j = i; j <= kml; ++j)
28         { aj[k-j] = 0.0;
29           dl[j] = dl[i];
30         }
31     }
32   jcmax = k;
33   nmi = n - i;
34   if (nmi >= 0)
35     for (j = 1; j <= kml; ++j)
36       dr[j] = t[i+j] - x;
37   else
38     { jcmax = k + nmi;
39       for (j = 1; j <= jcmax; ++j)
40         dr[j] = t[i+j] - x;
41       for (j = jcmax; j <= kml; ++j)
42         { aj[j+1] = 0.0;
43           dr[j] = dr[jcmax];
44         }
45     }
46   for (jc = jcmn; jc <= jcmax; ++jc)
47     aj[jc] = bcoef[imk+jc];
48   for (j = 1; j <= jderiv; ++j)

```

```

49     { kmj = ilo = k-j;
50       for (jj = 1; jj <= kmj; ++jj)
51         { aj[jj] = ((aj[jj+1] - aj[jj]) / (dl[ilo] + dr[jj])) * kmj;
52           ilo -= 1;
53         }
54     }
55     for (j = jderiv + 1; j <= kml; ++j)
56     { kmj = ilo = k - j;
57       for (jj = 1; jj <= kmj; ++jj)
58         { aj[jj] = aj[jj+1] * dl[ilo] + aj[jj] * dr[jj];
59           aj[jj] /= (dl[ilo] + dr[jj]);
60           ilo -= 1;
61         }
62     }
63     return(aj[1]);
64 }

```

APPENDIX H

```

1
2 int interv(xt, lxt, x, left)
3 /* input: xt[i], xt[i] <= xt[i+1], 1 <= i <= lxt.
4  * returns -1, if x < xt[1],
5  *           0, if xt[1] <= x <= xt[lxt] and
6  *           1, if x >= xt[lxt].
7  * computes in an economic way *left, such that
8  *   xt[*left] <= x < xt[*left+1], if
9  *   xt[1] <= x < xt[lxt].
10 */
11 int *left, lxt;
12 double xt[MAXP], x;
13 { static int ilo;
14   int ihi, istep, middle;
15   if (ilo == 0) ilo = 1;
16   ihi = ilo + 1;
17   if (ihi >= lxt)
18   { if (x >= xt[lxt]) { *left = lxt; return(1); }
19     if (lxt <= 1) { *left = 1; return(-1); }
20     ilo = lxt-1;
21     ihi = lxt;
22   }
23   if (x < xt[ihi] && x >= xt[ilo])
24   { *left = ilo; return(0); }
25   else if (x < xt[ilo])
26   { istep = 1;
27     do
28     { ihi = ilo;
29       ilo = ihi - istep;
30       if (ilo <= 1)
31       { ilo = 1;
32         if (x < xt[1]) { *left = 1; return(-1); }
33       }
34       istep += istep;
35     }
36     while (x < xt[ilo]);
37   }
38   else
39   { istep = 1;
40     do
41     { ilo = ihi;
42       ihi = ilo + istep;
43       if (ihi >= lxt)
44       { ihi = lxt;
45         if (x >= xt[lxt]) { *left = lxt; return(1); }
46       }
47       istep += istep;
48     }

```

```
49     while (x >= xt[ihi]);
50     }
51     for (;;)
52     { middle = (ilo + ihi) / 2;
53       if (middle == ilo) { *left = ilo; return(0); }
54       if (x < xt[middle])
55         ihi = middle;
56       else ilo = middle;
57     }
58 }
```


ONTVANGEN 9 5 OKT 1981 -