Adaptive plotting of splines

Stein Dahl

Master's Thesis, Autumn 2019

Adaptive plotting of splines

Stein Dahl

December 27, 2019

Abstract

We investigate how to exploit the properties of splines in order to plot curves and surfaces more efficiently than by using general plotting methods. The most elaborate methods use knot insertion and the convergence of the piecewise linear control polygon to ensure a visually acceptable representation of the spline. Several strategies for finding adequate knot locations are explored.

Preface

Spline functions have several properties that make them almost a perfect tool for representing mathematical functions and discrete data of one variable, including parametric curves. One of these properties is the intuitive and simple relationship between the discrete representation (the control polygon) and the underlying function. This means that in principle, it is possible to develop algorithms that guarantee that all details in the function are reproduced in a plot. It has, however, been difficult to find algorithms that exploit this in an efficient way.

The goal of this project is to give an overview of existing plotting methods, before investigating whether it is possible to exploit the core properties of splines to develop an efficient, adaptive algorithm that is guaranteed to catch all details in the underlying function.

As mathematical spline theory emerged and matured from the 1970s onwards, many scholars noticed this close relationship between the spline and its control polygon and the potential for an adaptive plotting method. Farin [Far97, p. 157] formulated the idea as follows:

> If sufficiently many knots have been inserted into the knot sequence, the resulting control polygon will be arbitrarily close to the curve. Then, instead of plotting the curve directly, one simply plots the refined polygon. To obtain an *adaptive* rendering method, one would control the knot insertion process by inserting more knots where the curve is of high curvature and fewer knots where it is flat.

The idea is here, of course, only vaguely formulated, and an extensive literature review has not revealed any published work exploiting the idea and attempting to investigate its potential in detail. It remains, therefore, to express a working algorithm that specifies how curvature is to be computed or approximated, where knots are to be inserted, and which stopping criterions should be chosen. Some have called the answer to these questions a 'missing link' in spline theory.

It is tempting to attack an optimization problem such as the above by reverting to Machine Learning or some other brute force method. We will, however, attempt a more analytical approach, identifying structure where structure is to be found. It is our belief that results of such a classical analysis are more adept to further theory refinement than output data only interpretable by computers. At times, we will therefore apply geometric reasoning for solving the case at hand, and at other times we will make use of functional analysis.

The reader is assumed to be familiar with undergraduate level mathematics and to have some programming experience.

Layout of thesis. We will first review some of the basics of B-splines and specify notation and terminology that will be used throughout the thesis. This is the object of Chapter 1. Chapter 2 is also an introductory chapter, providing context to plotting as a step in producing graphics on a computer system, as well as reviewing some general plotting methods used for arbitrary graphical objects and functions. The core mathematical contributions of the thesis are presented in Chapter 3, where we explore and develop different plotting algorithms based on analysis of the control polygon. Chapter 4 extends the theory to tensor products, allowing us to represent surfaces using the same methods, and contains a few notes on a possible parallel implementation and other ideas for further refinement of the plotting algorithm.

Conventions. As will be stated explicitly in Chapter 1, we use the term *degree*, noted p, and not *order* when referring to the underlying piecewise polynomials of the splines. This simplifies certain formulas and gives a tighter relationship between the text and the geometric considerations of objects.

Acknowledgements. Mathematical thinking is developed at a young age. One should not, and I do not, take for granted any education, formal or informal, given or taken, in the early, middle or later years of life. Beyond this general gratitude, which I believe is well under-expressed by many in our era, I make explicit some of it to my supervisor Professor Knut Mørken.

Contents

List of Figures

CHAPTER 1

Background on Splines

This introductory chapter reviews some of the theory of splines relevant to plotting. Splines will be defined as linear combinations of B-splines before the introduction of the concepts of 'control polygon' and 'knot insertion'. We restrict ourselves to functions and curves, treating the case of surfaces in a later chapter. Most of the material in this chapter, unless stated otherwise, is taken from [De 01], first published in 1978 and the standard reference in spline theory for several decades.

1.1 B-splines and properties of B-splines

1.1.1 Definitions and notation

Splines are piecewise polynomials with special smoothness properties.

Definition 1.1 (B-splines, Cox–de Boor recursion formula)**.** Let p be a non-negative integer and $\mathbf{t} = \{t_i\}$ a sequence of non-decreasing real numbers of length at least $p+2$. The sequence $\mathbf{t}$ is called a *knot vector.* On the knot-vector $\mathbf{t}$ we define the jth B-spline of degree p by

$$B_{j,p,\mathbf{t}}(x) = \frac{x - t_j}{t_{j+p} - t_j} B_{j,p-1,\mathbf{t}} + \frac{t_{j+p+1} - x}{t_{j+p+1} - t_{j+1}} B_{j+1,p-1,\mathbf{t}} \tag{1.1}$$

for all real numbers x, with

$$B_{j,0,\mathbf{t}}(x) = \begin{cases} 1 & \text{if} \quad t_j \leq x < t_{j+1} \\ 0 & \text{otherwise.} \end{cases} \tag{1.2}$$

Some knot-vectors may lead to a '0/0'-situation in the recurrence relation. We then adopt the '$0/0 = 0$'-convention. Also, when understood from the context, we may omit some subscripts and use shorthands such as $B_j(x)$.

B-splines have quite a number of properties that will be central in the following analysis, and we state them here.

Lemma 1.2.

1. *Local knots.* The jth B-spline only depends on the knots $t_j, t_{j+1}, \ldots, t_{j+p+1}$.
2. *Local support.* If x is outside of the interval $[t_j, t_{j+p+1})$, then $B_j(x) = 0$.

3. *Active B-splines.* If x lies in the interval $[t_\mu, t_{\mu+1})$, then $B_j(x) = 0$ for $j < \mu - p$ and $j > \mu$.
4. *Positivity.* If x is in (t_j, t_{j+p+1}), then $B_j(x) > 0$.
5. *Partition of unity.* If x is in $[t_\mu, t_{\mu+1})$, then $\sum_{j=\mu-p}^{\mu} B_j(x) = 1$.
6. *Special values.* If $x = t_{\mu+1} = \cdots = t_{\mu+p} < t_{\mu+p+1}$, then $B_\mu(x) = 1$ and $B_j(x) = 0$ for $j \neq \mu$.

The knot vector controls the B-splines, and it is by manipulating the knot vector that we can enforce desired criteria in a curve or a plot. We call the *knot multiplicity* the number of times a knot occurs in $\mathbf{t}$. Some knot vectors lead to vacuous B-splines, i.e., the B-spline is identically zero, for instance when the multiplicity of a knot is greater than $p + 1$. As we shall see below, it is actually desirable to have knots of multiplicity $p + 1$ in both ends of a knot vector. This encourages the definition of knot vectors holding these properties.

Definition 1.3 ($(p + 1)$-regular knot vector)**.** A knot vector $\mathbf{t} = \{t_i\}_{i=1}^{n+p+1}$ is said to be $p + 1$-regular if

- The multiplicity of t_1 and t_{n+p+1} is $p + 1$.
- The multiplicity of all other knots is less than or equal to $p + 1$.

This implies $n \geq p + 1$.

When we refer to a *spline*, we mean a linear combination of B-splines defined on the same knot vector. The combination coefficients can be of arbitrary dimension.

Definition 1.4 (Spline space and B-spline coefficients)**.** Let $\mathbf{t} = \{t_i\}_{i=1}^{n+p+1}$ be a knot vector for n B-splines. Let $s \geq 1$ be an integer. The space of all linear combinations of these B-splines of degree p is the spline space $\mathbb{S}_{p,\mathbf{t}}^s$ defined by

$$\mathbb{S}_{p,\mathbf{t}}^s = \left\{ \sum_{j=1}^{n} \mathbf{c}_j B_{j,p} \mid \mathbf{c}_j \in \mathcal{R}^s \textit{ for } \quad 1 \leq j \leq n \right\}. \tag{1.3}$$

When $s = 1$, an element $f \in \mathbb{S}_{p,\mathbf{t}}^1 = \mathbb{S}_{p,\mathbf{t}}$ is called a *spline function* of degree p with knots $\mathbf{t}$. Similarly, when $s \geq 2$, an element $f \in \mathbb{S}_{p,\mathbf{t}}^s$ is called a *spline curve* of degree p. The coefficients $(\mathbf{c}_j)_{j=1}^n$ are called the *B-spline coefficients* of f.

Spline functions and spline curves inherit the properties stated in Lemma 1.2. We will at times lighten notation by refraining from using the bold $\mathbf{c}_j$, meaning a B-spline coefficient written c_j may have several components, understood from context.

1.1.2 Continuity and smoothness of splines

All knot vectors used in this paper will be $p+1$-regular. This is in fact without loss of generality. Indeed, assume first that the multiplicities in the end of the knot vector are less than $p + 1$. We can then simply extend the knot vector by adding more knots at the ends, and identify the old spline space with a new spline space where we set the appropriate number of first and/or last of the

B-spline coefficients to zero. Similarly, if a knot has multiplicity greater than $p+1$, we remove enough occurrences to obtain a $p+1$-regular knot vector, and remove the equal number of B-spline coefficients that were related to B-splines with empty support.

Example 1.5 (Reducing to $(p+1)$-regular knot vectors)**.** Let f be a spline of degree $p=2$ defined by the knot vector $\mathbf{t}=(0,0,1,1,1,1,2,2,2)$ and the coefficient vector $\mathbf{c}=(1,3,2,5,4,2)$. The knot vector $\mathbf{t}$ needs an extra 0 to ensure multiplicity three in the beginning of the knot vector. The third coefficient $c_3=2$ relates to a B-spline with empty support and can be removed. We thus associate f with $\tilde{f}$ defined on the knot vector $\tilde{\mathbf{t}}=(0,0,0,1,1,1,2,2,2)$ with coefficients $\tilde{\mathbf{c}}=(0,1,3,5,4,2)$.

The continuity and smoothness of a spline is directly linked to the knot-vector.

Proposition 1.6 (Spline smoothness)**.** *Let* $\mathbf{t}=\{t_i\}_{i=1}^{n+p+1}$ *be a* $(p+1)$*-regular knot vector and let* $f=\sum_{j=1}^{n} c_j B_{j,p,\mathbf{t}}$ *be a spline in* $\mathbb{S}_{p,\mathbf{t}}$*. If* x *occurs* m *times in* $\mathbf{t}$ *then* f *has continuous derivatives of order* $0,\dots,p-m$ *at* x*.*

This allows for simplifications in the context of plotting: if some interior knot has multiplicity exactly $p+1$, then the spline has a discontinuity in that knot. It is then natural to plot the two continuous parts of the curve separately. As to the adaptive plotting techniques developed later, properties of the control polygon (cf. Section 1.2) allow the stronger assumption that no interior knot has multiplicity more than $p-1$.

These checks and adaptations can easily be conducted during a preprocessing stage. Given a spline defined by its knot vector and its coefficient vector, a first call to a procedure `check_regular` performs dimension consistency checks and on no error returns a knot vector that is $p+1$-regular and an associated potentially modified/trimmed coefficient vector. This output is then sent to a procedure `check_continuous` which checks for discontinuities and, if applicable, splits the spline into continuous splines, each of which is forwarded to the plotting function.

1.1.3 Evaluating a Spline

In discussing the evaluation of splines, we confine ourselves to a high-level description of a few common methods. For more details, see for instance [BBB95; Sch07]. As we shall see, evaluating a spline is costly, especially for higher degrees, despite the clever but now classic methods applied.

Direct computation

The spline function f is a sum of n B-splines. From Item 3 on the facing page, however, on each knot interval $[t_\mu, t_{\mu+1})$ only $p+1$ of the B-splines are active.

Lemma 1.7. *For* $f=\sum_{j=1}^{n} c_j B_{j,p,\mathbf{t}}$ *and* $x\in[t_\mu,t_{\mu+1}), p+1\le\mu\le n$*, we have*

$$f(x)=f_\mu(x):=\sum_{j=\mu-p}^{\mu} c_j B_{j,p,\mathbf{t}}(x). \tag{1.4}$$

This finite sum is well represented using matrix operations. Let

$$\mathbf{c}_\mu := (c_{\mu-p}, \dots, c_\mu)^T, \tag{1.5}$$

and, using the shorthand $B_j = B_{j,p,\mathbf{t}}$,

$$\mathbf{B}_\mu := (B_{\mu-p}, \dots, B_\mu)^T. \tag{1.6}$$

Then equation Equation (1.4) is written

$$f(x) = \mathbf{B}_\mu^T \mathbf{c}_\mu. \tag{1.7}$$

On a side note, such matrix representations are more in the spirit of the current (2019) general programming best practices, as the advent of multicore processors and new processor architectures has made parallelizing and thus optimization more straightforward.

The vector $\mathbf{B}_\mu$ can be factored by exploiting the structure in the recurrence relation in Equation (1.1).

Lemma 1.8. *Let $\mathbf{t} = \{t_i\}_{i=1}^{n+p+1}$ be a knot vector for a B-spline of degree p and μ an integer such that $t_\mu < t_{\mu+1}$ and $p+1 \le \mu \le n$. For each positive integer k with $k \le p$ define the* B-spline matrix *$\mathbf{R}_k^\mu(x)$ of size $k \times (k+1)$ by*

$$\mathbf{R}_k^\mu(x) = \begin{pmatrix} \frac{t_{\mu+1}-x}{t_{\mu+1}-t_{\mu+1-k}} & \frac{x-t_{\mu+1-k}}{t_{\mu+1}-t_{\mu+1-k}} & 0 & \cdots & 0 \\ 0 & \frac{t_{\mu+2}-x}{t_{\mu+2}-t_{\mu+2-k}} & \frac{x-t_{\mu+2-k}}{t_{\mu+2}-t_{\mu+2-k}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{t_{\mu+k}-x}{t_{\mu+k}-t_\mu} & \frac{x-t_\mu}{t_{\mu+k}-t_\mu} \end{pmatrix}. \tag{1.8}$$

Then, for $x \in [t_\mu, t_{\mu+1})$,

$$\mathbf{B}_\mu(x)^T = (B_{\mu-p}(x), \dots, B_\mu(x)) = \mathbf{R}_1^\mu(x)\mathbf{R}_2^\mu(x)\cdots\mathbf{R}_p^\mu(x). \tag{1.9}$$

This means that if $f = \sum_{j=1}^n c_j B_{j,p,\mathbf{t}}$ is a spline in $\mathbb{S}_{p,\mathbf{t}}$ and x is restricted to the interval $[t_\mu, t_{\mu+1})$, then $f_\mu(x)$ is given by

$$f_\mu(x) = \mathbf{R}_1^\mu(x)\mathbf{R}_2^\mu(x)\cdots\mathbf{R}_p^\mu(x)\mathbf{c}_\mu. \tag{1.10}$$

Accumulating the matrix products from right to left yields the computations

$$\mathbf{c}_{\mu,p-k+1} := \mathbf{R}_k^\mu(x)\mathbf{c}_{\mu,p-k} \quad \text{for} \quad k = p, p-1, \dots, 1, \tag{1.11}$$

with $\mathbf{c}_{\mu,0} = \mathbf{c}_\mu$ and $f(x) = \mathbf{c}_{\mu,p}$, where in each step the length of the vector $\mathbf{c}_{\mu,p-k+1}$ is reduced with 1 until obtaining the scalar $\mathbf{c}_{\mu,p}$. This approach thus reveals a 'triangular algorithm' for computing the value of a spline, see Figure 1.

The situation is similar for the derivatives of a spline.

Lemma 1.9. *Let x be a number in $[t_\mu, t_{\mu+1})$. Then the rth derivative, with $0 \le r \le p$, of the vector of B-splines $\mathbf{B}_\mu(x)^T = (B_{\mu-p}(x), \dots, B_\mu(x))$ is given by*

$$D^r\mathbf{B}_\mu(x)^T = \frac{p!}{(p-r)!}\mathbf{B}_{p-r}^\mu(x)^T D\mathbf{R}_{p-r+1}^\mu \cdots D\mathbf{R}_p^\mu. \tag{1.12}$$

$$\begin{tikzcd}[column sep=small]
 & & & & & & & & c_{\mu-4,0} \arrow[dll, "t_{\mu+1}-x"'] \\
 & & & & & & c_{\mu-3,1} \arrow[dll, "t_{\mu+1}-x"'] & t_{\mu+1}-t_{\mu-3} & \\
 & & & & c_{\mu-2,2} \arrow[dll, "t_{\mu+1}-x"'] & t_{\mu+1}-t_{\mu-2} & & & c_{\mu-3,0} \arrow[ull, "x-t_{\mu-3}"] \arrow[dll, "t_{\mu+2}-x"'] \\
 & & c_{\mu-1,3} \arrow[dll, "t_{\mu+1}-x"'] & t_{\mu+1}-t_{\mu-1} & & & c_{\mu-2,1} \arrow[ull, "x-t_{\mu-2}"] \arrow[dll, "t_{\mu+2}-x"'] & t_{\mu+2}-t_{\mu-2} & \\
c_{\mu,4} & t_{\mu}-t_{\mu-1} & & & c_{\mu-1,2} \arrow[ull, "x-t_{\mu-1}"] \arrow[dll, "t_{\mu+2}-x"'] & t_{\mu+2}-t_{\mu-1} & & & c_{\mu-2,0} \arrow[ull, "x-t_{\mu-2}"] \arrow[dll, "t_{\mu+3}-x"'] \\
 & & c_{\mu,3} \arrow[ull, "x-t_{\mu}"] & t_{\mu+2}-t_{\mu} & & & c_{\mu-1,1} \arrow[ull, "x-t_{\mu-1}"] \arrow[dll, "t_{\mu+3}-x"'] & t_{\mu+3}-t_{\mu-1} & \\
 & & & & c_{\mu,2} \arrow[ull, "x-t_{\mu}"] & t_{\mu+3}-t_{\mu} & & & c_{\mu-1,0} \arrow[ull, "x-t_{\mu-1}"] \arrow[dll, "t_{\mu+4}-x"'] \\
 & & & & & & c_{\mu,1} \arrow[ull, "x-t_{\mu}"] & t_{\mu+4}-t_{\mu} & \\
 & & & & & & & & c_{\mu,0} \arrow[ull, "x-t_{\mu}"]
\end{tikzcd}$$

Figure 1: *A triangular algorithm for computing the value of a spline. Example with* $p = 4$*. A quantity* $c_{\mu-k,j}$ *is a convex combination of the two quantities at the beginning of the two arrows pointing at this value. The numerator weight is written along the arrow and the common denominator weight is written between the two arrows.*

Each of the r rightmost factors $D\mathbf{R}_k^\mu$ denote the matrix obtained by differentiating each entry in $\mathbf{R}_k^\mu(x)$ with respect to x, thus these are independent of x. This means that if $f = \sum_{j=1}^n c_j B_{j,p,\mathbf{t}}$ is a spline in $\mathbb{S}_{p,\mathbf{t}}$ and x is restricted to the interval $[t_\mu, t_{\mu+1})$, then the rth derivative of f at x is given by

$$D^r f_\mu(x) = \frac{p!}{(p-r)!}\mathbf{R}_1^\mu(x)\cdots\mathbf{R}_{p-r}^\mu(x)D\mathbf{R}_{p-r+1}^\mu\cdots D\mathbf{R}_p^\mu\mathbf{c}_\mu, \tag{1.13}$$

and that the main computational effort for this evaluation lies in the $p-r$ first matrix factors, similarly as for a spline of degree $p-r$.

The triangular nature of the algorithm infers a quadratic computational complexity in the degree of the spline. In a careful implementation, the operation count is as follows.

Proposition 1.10 (Operational cost for spline evaluation)**.** *The evaluation in a point of a spline of degree p given in B-spline form has an operational count of*

$$R(p) = \frac{3}{2}p(p+1). \tag{1.14}$$

Evaluation of an rth derivative has an operational count of

$$R^r(p) = \frac{3}{2}(p-r)(p-r+1). \tag{1.15}$$

In addition comes some work, for instance a binary search, to find the correct index μ for a given x. In the context of plotting an interval $[a, b)$, μ is typically found for the first point to evaluate, i.e., the left bound a, and this value is either reused or increased according to a few simple if-tests, meaning its computational cost is negligible when considering the total plotting process.

Other spline representations and conversions between forms

A spline has other possible representations than the B-spline expansion. Restricted to one knot interval $[t_\mu, t_{\mu+1})$, the spline f is a polynomial of degree p and we may write

$$f_\mu(x) = \sum_{i=0}^{p} w_{i,\mu}(x - t_\mu)^i, \qquad x \in [t_\mu; t_{\mu+1}), \tag{1.16}$$

for certain coefficients $w_{i,\mu}$. To find these coefficients, we may first express each of the $p+1$ contributing B-splines as polynomials of degree p in canonical form, i.e., such that

$$B_{j,p,\mathbf{t}}(x) = \sum_{i=0}^{p} b_{j,i}(x - t_j)^i, \qquad j = \mu - p, \ldots, \mu, \tag{1.17}$$

for coefficients $b_{j,i}$ to be determined. This is achieved by writing out the recurrence relation for each B-spline, thus sequentially unloading the degree from the lower-degree basis splines onto its coefficients. This allows for identification of coefficients of same-order terms in Equation (1.16).

Once this new form is obtained, it allows for relatively cheap evaluation - only linear in the degree of the spline. Popular methods include using nested multiplication (Horner's method), i.e., rewriting Equation (1.16) with $y = x - t_\mu$ as

$$f_\mu(x) = w_{0,\mu} + y(w_{1,\mu} + y(w_{2,\mu} + \ldots + y(w_{p-1,\mu} + yw_{p,\mu})\cdots), \tag{1.18}$$

which requires only p multiplications and $p-1$ additions. The conversion from the B-spline form, however, has certain disadvantages. First, there is a considerable conversion cost in the degree of the spline – quadratic, since we have to write out the recurrence relations. Second, we risk decreasing the numerical precision [COX72; Far97]. For lower degree splines ($p = 2, 3$), the breakpoint as to whether it pays to carry out the conversion is experimentally found to be only about two evaluations per knot interval [Sch07, Ch. 5], but more evaluations for higher degrees. This suggests a conversion usually is justified for use in plotting in practical situations.

For the special case of knot vectors where each knot appears with multiplicity $p+1$ (*Bernstein knot vectors*), we obtain *Bézier curves*, for which there are additional high-performing evaluation methods, but these methods will not be considered here.

1.2 The control polygon of a Spline

1.2.1 Definitions and initial properties

Given a spline, some parameter values have special properties.

Definition 1.11 (Knot averages (Greville abscissae))**.** Let $\mathbf{t} = \{t_i\}_{i=1}^{n+p+1}$ be a knot vector. Let $\mathbf{t}^* = \{t_j^*\}_{j=1}^{n}$ with

$$t_j^* = \frac{1}{p}(t_{j+1} + \ldots + t_{j+p}). \qquad \text{(Greville)}$$

The vector $\mathbf{t}^*$ holds the *knot averages*.

We observe that the definition of $p+1$-regular knot vectors ensures the value of end-points is conserved when computing the knot averages,, i.e., for a $p+1$-regular knot vector $\mathbf{t}$, we have $t_1^* = t_1$ and $t_n^* = t_n$. The special value property from Lemma 1.2 leads to a stronger result.

Lemma 1.12 (Knot with multiplicity p)**.** *Let f be a spline of degree p with coefficients $\mathbf{c}$ and knot vector $\mathbf{t}$. If $t_{\mu+1} = t_{\mu+2} = \ldots = t_{\mu+p} < t_{\mu+p+1}$, then $f(t_\mu^*) = c_\mu$.*

From here, it is necessary to distinguish the case of explicit curves and parametric curves.

Definition 1.13 (Control points, control polygon)**.** Let f be a spline in $\mathbb{S}_{p,\mathbf{t}}^s$, with $s = 1$ or $s = 2$. The *control points* $\{\mathbf{P}_j\}_{j=1}^n$ of the spline are given by

$$\mathbf{P}_j = \begin{cases} (t_j^*, c_j) & \text{if} \quad s = 1 \\ \mathbf{c}_j & \text{if} \quad s = 2. \end{cases} \tag{1.19}$$

The *control polygon* $\Gamma_{p,\mathbf{t}}(f)$ of f is the piecewise linear interpolant of the points $\{\mathbf{P}_j\}_{j=1}^n$,, i.e., the polygon obtained by connecting neighbouring control points by straight lines. Formally,

$$(1-t)\mathbf{P}_j + t\mathbf{P}_{j+1} \quad \text{for} \quad j = 1, \ldots, n \text{ and } \quad t \in [0,1)\,.$$

For spline functions, the knot averages give the position along the first axis. Figure 2 shows an example of a control polygon (dashed black) for a spline function (red). The control points are indicated with black circles. The values of the knot vector are indicated along the first axis as filled, red circles underneath the graphs. Stacked circles signifies multiplicities. In particular, the end knots here are of multiplicity $p + 1 = 4 + 1 = 5$.

We notice that the B-spline representation of a spline gives almost immediate access to the control polygon in that the control point coordinates are readily available. Additionally, there is an intuitive relationship between the spline and its control polygon: the spline appears as nothing but a smoother version of the control polygon. This variation diminishing property is part of the shape preserving properties of B-splines: A spline is bounded by the extrema of its control polygon, it is monotone if the control polygon is monotone, and it is convex if the control polygon is convex.

Additionally, from the properties listed in Lemma 1.2, it is clear that a point on the spline is a weighted finite sum of only positive weights that sum to one, leading to the notion of convex hull.

Lemma 1.14 (Convex hull)**.** *Let $f = \sum_{j=1}^n c_j B_{j,p,\mathbf{t}}$ be a spline and $x \in [t_\mu, t_{\mu+1})$. Then $B_j(x)$ is in the convex hull of the $p+1$ control points $\mathbf{P}_{\mu-p}, \ldots, \mathbf{P}_\mu$.*

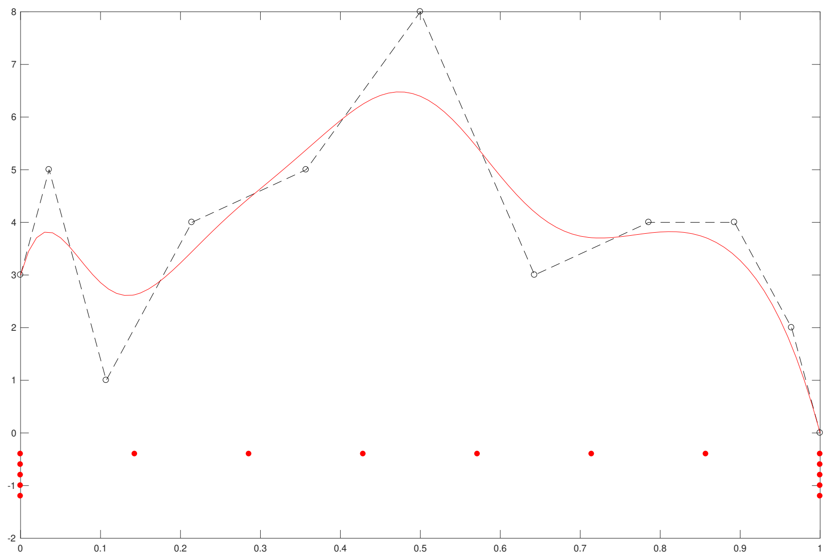


Figure 2: *A spline with its corresponding control polygon. You can 'guess' the shape of the spline.*

Some early authors may use the term *hodograph* instead of control polygon when splines are reduced to Bézier curves [For72]. We provide an example of computing control point coordinates in such a case. Let $p = 2$ and the knot vector $\mathbf{t}$ and B-spline coefficients $\mathbf{c}$ of a spline f be defined as follows:

$$\begin{aligned}\mathbf{t} =&(0,0,0,1,1,2,2,3,3,3),\\ \mathbf{c} =&(-2,-1,0,1,-3,0,2,3).\end{aligned}$$

Then the knot averages and corresponding computed values are

$$\begin{aligned}\mathbf{t}^* =&(0,\frac{1}{2},1,\frac{3}{2},2,\frac{5}{2},3)\\ f(t^*) =&(-2,-1,0,1,-3,0,2,3)\\ =& \,\mathbf{c}.\end{aligned}$$

1.2.2 Convergence of the control polygon to the spline function

The closer the knots are, the better the control polygon approximates the spline itself. To show this, we follow the reasoning as described in [LM08, Ch. 9], where proofs are also provided.

Firstly, considerations of particularly chosen quasi-interpolants lead to a result on the B-spline coefficients. More precisely, the size of the B-spline coefficient c_j can be bounded in terms of the size of the spline on the interval $[t_{j+1}, t_{j+p}]$:

Lemma 1.15 (Bound on B-spline coefficients)**.** *For any spline $f = \sum_{j=1}^n c_j B_{j,p,\mathbf{t}}$ the size of the B-spline coefficients c_j is bounded by*

$$|c_j| \leq \tilde{K}_p ||f||_{[t_{j+1},t_{j+p}]} \qquad \textit{for} \quad j = 1,\ldots,n, \tag{1.20}$$

where

$$\tilde{K_p} = \frac{2^p}{p!}(p(p-1))^p \tag{1.21}$$

depends only on p.

By considering the error in the first order Taylor expansion of a spline, which is itself a spline, it is possible to show that a control point is close to the spline when the knot spacing is small:

Lemma 1.16 (Control points are close to the spline)**.** *Let* $f = \sum_{j=1}^{n} c_j B_{j,p,\mathbf{t}}$ *be a spline in* $\mathbb{S}_{p,\mathbf{t}}$*. Then*

$$|c_j - f(t_j^*)| \leq \hat{K_p}(t_{j+1} - t_{j+p})^2 ||D^2 f||_{[t_{j+1},t_{j+p}]}, \tag{1.22}$$

where the operator D^2 *denotes one-sided differentiation from the right, and where*

$$\hat{K_p} = \frac{2^{p-1}}{p!}(p(p-1))^p. \tag{1.23}$$

The third step involes showing that the whole control polygon, and not only the single control points, converges to the spline. This is mainly a geometric argument, where on each interval $\left[t_j^*, t_{j+1}^*\right]$ the linear interpolant between the end point sample points is compared to both the spline and its control polygon on that interval. The result is stated in the following theorem.

Theorem 1.17 (Quadratic convergence of the control polygon)**.** *Let* $f = \sum_{j=1}^{n} c_j B_{j,p,\mathbf{t}}$ *be a spline in* $\mathbb{S}_{p,\mathbf{t}}$*. Then*

$$||\Gamma_{p,\mathbf{t}}(f) - f||_{[t_1^*,t_n^*]} \leq K_p h^2 ||D^2 f||_{[t_1,t_n+p+1]}, \tag{1.24}$$

where $h = \max_i \{t_{i+1} - t_i\}$*, and the constant* K_p *only depends on* p*. In fact, we have:*

$$K_p = \frac{2^{p-1}}{(p-2)!}p^3(p-1) + \frac{1}{8}. \tag{1.25}$$

This theorem is what initiated the study of the control polygon as a viable substitute for the spline when plotting. The term 'quadratic' refers to the exponent 2 in the factor h (*mesh size*) on the right hand side.

1.3 Knot insertion

1.3.1 Knot refinement

The study of the control polygon reveals that closely spaced knots (small mesh) ensure a control polygon that is close to the spline. This motivates the process of knot insertion, i.e., extension of a given knot vector by adding new knots.

Definition 1.18 (Knot refinement)**.** Let

$$\mathbf{t} = \{t_i\}_{i=1}^{n_0+p+1} \tag{1.26}$$

and

$$\bar{\mathbf{t}} = \{\bar{t}_j\}_{j=1}^{n_1+p+1} \tag{1.27}$$

be two knot vectors such that $n_0 < n_1$. The knot vector $\bar{\mathbf{t}}$ is a *refinement* of $\mathbf{t}$ if any real number occurs at least as many times in $\bar{\mathbf{t}}$ as in $\mathbf{t}$. We may use the abusive notation $\mathbf{t} \subset \bar{\mathbf{t}}$.

It is intuitive and can be proven that a spline f in the *coarse* spline space $\mathbb{S}_{p,\mathbf{t}}$ is also an element of the *finer* spline space $\mathbb{S}_{p,\bar{\mathbf{t}}}$, since the spline satisfies the smoothness conditions of this finer space. This means that the spline can be expressed in two different bases, and knot insertion can be considered a change of basis from the B-spline basis in $\mathbb{S}_{p,\mathbf{t}}$ to the B-spline basis in $\mathbb{S}_{p,\bar{\mathbf{t}}}$. Following this argument we obtain the equations

$$f = \sum_{i=1}^{n_0} c_i B_{i,p,\mathbf{t}} = \sum_{j=1}^{n_1} \bar{c}_j B_{j,p,\bar{\mathbf{t}}}. \tag{1.28}$$

There are numerous algorithms available to compute the new coefficients $\bar{\mathbf{c}} = \{\bar{c}_j\}_{j=1}^{n_1+1}$, a problem which is widely discussed in literature, as finding algorithms for computing the new spline coefficients on a refined knot vector has been a vivid area of research for many decades [BZ92; De 01; Lyc89]. Among the most noteworthy methods are Oslo-Algorithm II [CLR80; LM86] and Böhm's method [Boe80; BP85]. We present the latter in the next subsection.

As knot insertion will constitute a crucial part of the plotting techniques discussed later, we must make a few notes on efficiency. It is possible to evaluate the theoretical efficiency of the different knot insertion algorithms and find exact expressions of the computational cost of their execution, typically by counting the number of floating point operations involved. For examples of such work, see [Boe85; LCM85]. When facing practical situations, however, this efficiency depends on many factors some of which are difficult to categorize and identify 'on the fly', i.e., without a too severe computational penalty. Factors to consider include whether there is a curve or a surface setting, the relation between the old and the new knot vector as well as the multiplicity of the inserted knots. Other factors to consider are the numerical stability of the different methods, the simplicity of their implementation, hardware configurations and their memory usage.

To keep things simple, we will consider only one of these methods throughout our analysis. As we shall see, the most developed methods are based on a sequential knot insertion process. One study indicates that the algorithm with the overall best performance in such a setting, both for curves and for surfaces, is the classical Böhm's method [Fug93].

This choice is further supported by the observation that a major software provider [Mat19], clearly aiming for simplicity, implements Böhm's method and not any of the other methods developed despite the age of other algorithms approaching many decades. Indeed, software documentation indicates that old Fortran methods have simply been migrated to the new platform. Programmers may well interpret this as a cue to revise the source code and increase the practical efficiency of many related operations.

1.3.2 A knot insertion algorithm: Böhm's method

Böhm's method inserts knots sequentially, one at a time, and updates the B-spline coefficients between each insertion. The formulas are simple and explicit, and involve only convex combinations, which are computationally stable.

Lemma 1.19 (Böhm's method). *Let* $\mathbf{t} = \{t_i\}_{i=1}^{n+p+1}$ *be a given knot vector and let* $\bar{\mathbf{t}} = \{\bar{t}_j\}_{j=1}^{n+p+2}$ *be the knot vector obtained by inserting a knot* z *in* $\mathbf{t}$ *in the* *interval* $[t_\mu, t_{\mu+1})$. *If*

$$f = \sum_{i=1}^{n} c_i B_{i,p,\mathbf{t}} = \sum_{j=1}^{n+1} \bar{c}_j B_{j,p,\bar{\mathbf{t}}}, \tag{1.29}$$

then the $\{\bar{c}_j\}_{j=1}^{n+1}$ *can be expressed in terms of the* $\{c_i\}_{i=1}^{n}$ *through the formulas*

$$\bar{c}_j = \begin{cases} c_j & \textit{if} \quad 1 \le j \le \mu - 1, \\ \frac{z-t_j}{t_{j+p}-t_j} c_j + \frac{t_{j+p}-z}{t_{j+p}-t_j} c_{j-1} & \textit{if} \quad \mu - p + 1 \le j \le \mu, \\ c_{j_1} & \textit{if} \quad \mu + 1 \le j \le n+1. \end{cases} \tag{1.30}$$

The crucial observation is that inserting a knot provides one additional point in the control polygon, and that this operation is much cheaper than evaluating a spline. Indeed, inserting a knot involves computing p new coefficients. Each new coefficient is a convex combination of two values. In a clever implementation such a computation involves 4 additions, 2 multiplications and 1 division.

Lemma 1.20. *Knot insertion is 'linear' in the degree of the spline, whereas evaluation of splines is 'quadratic' in the degree of the spline.*

CHAPTER 2

General considerations for plotting of curves

In this chapter we review some fundamental concepts pertaining to computer plotting in general. The first section describes the very basic hardware and software elements that enable plotting on a screen. Although computer graphics represents a vivid area of research with many commercial purposes, meaning technology is quickly outdated, the principles laid out in this section should hold ground for some time. Classical sources on the topic are [Fol+90; Hug+14], whereas [**topdown**; Shr+13] represent more recent sources. In Section 2.2, we give criteria for the visual acceptability of the representation of a curve. Section 2.3 is devoted to explaining some standard and well-used methods for rendering general curves. These methods serve as benchmarks when comparing with the methods developed in later chapters.

2.1 Plotting on a raster display

2.1.1 Computer displays

Digital screens consist of pixels of fixed width and distance. The connected graphics device can normally control the light output of each individual pixel.

Although the exact number of pixels and the aspect ratio differ between screens, the technical standards organization Video Electronics Standards Association[1] (VESA) maintains an effort to standardize video displays across producers. Modern (2019) resolutions both for personal and professional use range in the couple of thousands pixels in each x- and y-direction. See Table 1 for a few typical configurations and rough estimates of the relative pixel density (pixels per inch, PPI).

It is clear that the perceived quality of the screen also depends on viewing distance, meaning manufacturers are likely to use a size-independent universal parameter such as pixels-per-degree (PPD) when promoting their screens, but the general idea given above of the precision of typical devices is accurate enough for our use.

[1] https://vesa.org/about-vesa/missionvision/

Display standard	Pixel resolution	PPI
VGA	640x480	30
HD	1280x720	45
WQXGA	2560x1600	90
Apple Retina	2560x1600	220
4K UHD	3840x2160	90

Table 1: *Some typical resolutions of displays. The number of pixels per inch (PPI) is an approximate indication for screens of the physical size using the standard indicated.*

Observation 2.1. *Screens used for plotting have a relative pixel density of more than 45 pixels per inch,, i.e., some 20 pixels per centimeter. Printers generally have higher pixel densities.*

2.1.2 Rendering

In computer graphics, *rendering* is the process of turning raw information about objects and their relative position, and visible features such as colors, shading, shadows, reflection, transparency and refraction, into a digital 2D image. The simplest objects are referred to as *primitives* and for most systems these include points, lines, triangles and circles.

Observation 2.2. *The most common method of a graphical system to render a curve is to approximate the curve by straight lines. In other words, curves are not inherently 'smooth' on the screen, but the approximation by a polygonal curve is generally good enough to make this appearance.*

2.1.3 Rasterization

The finite precision nature of the rectangular grid of the pixels means that objects in general cannot be represented accurately. The digital image obtained from the rendering process must thus go through a *rasterizer* stage to determine exactly which pixels are to hold which intensities in order to represent the image. For modern Graphics Processing Units (GPUs), which in general are built using a modular approach, rasterization is the responsibility of the highly optimized *Render Output Unit (ROP)* of the GPU.

Important mechanisms in the rasterizer stage are algorithms for drawing lines and circles. For instance, *Bresenham's line algorithm* determines the pixels to activate to create the illusion of a straight line. It takes on input the line end points (x_0, y_0) and (x_1, y_1) and computes the slope of the line $m = \frac{y_1 - y_0}{x_1 - x_0}$. The input coordinates are here assumed to be integer coordinates corresponding to a pixel on the screen, with $x_0 < x_1$ and $y_0 < y_1$. The slope is assumed to be between 0 and 1. If any of these conditions do not hold, adjustments by symmetry properties or reversing roles of input data ensure the generality of the algorithm.

The algorithm activates one pixel in each column between and including the first and last columns. The first pixel (x_0, y_0) and the last pixel (x_1, y_1) are always activated. In the next column, there is a check for whether the center of $(x_0 + 1, y_0)$ on the same level, or $(x_0 + 1, y_0 + 1)$ one level up is closer to the value of the line in the center of that column. The closest one is activated, and

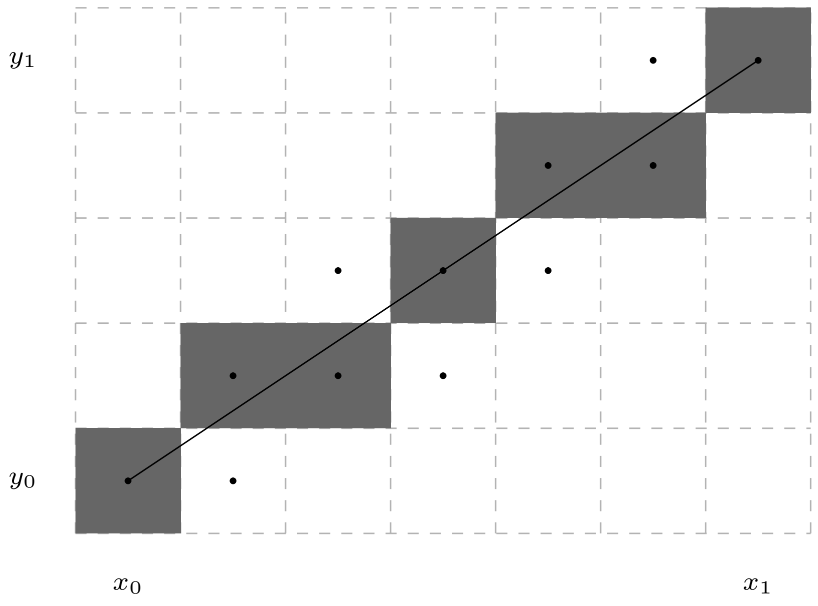


Figure 3: *Principle of Bresenham's line drawing algorithm. For a given column, the pixel whose center in the vertical direction is closest to the line is chosen.*

the scheme continues to the next column until reaching the last column. See Figure 3.

Other notable rasterizing algorithms are Wu's algorithm and the midpoint circle algorithm. We note that the object to be drawn usually has a thickness which exceeds the pixel width. This allows for *antialiaising*, i.e., activation of neighbouring pixels with varying intensity in order to 'round off' the sharp pixel edges.

Observation 2.3. *Geometric primitives are handled by the graphical system using highly optimized, low-level software and/or hardware mechanisms. When developing plotting tools for spline curves, it suffices to provide an array of points $P_i = (x_i, y_i)$ through which straight lines are to be drawn.*

2.2 Visual quality

When considering the visual quality of a piecewise linear approximation representing a smooth curve, we identify two different aspects of the representation: the visual smoothness, which is related to curvature, and the proximity of the approximation.

2.2.1 Visual smoothness and curvature

Since the polygonal lines of raster images and raster displays are not inherently smooth, they will not accurately represent smooth curves. Typically, if we use too few segments, the resulting approximative curve representation will contain easily identifiable sharp corners violating the smooth nature of the original curve. It could be tempting to simply increase the number of segments. Such a brute method, however, reduces the efficiency of any plotting algorithm and should not be used indiscriminantly: increasing the number of segments should be done intelligently. As we shall see later, this is rarely the case, even in commercial software.

Clearly, then, a good representation of a smooth curve avoids sharp corners. How sharp a corner becomes depends on the curve's *curvature*, i.e., how much the original curve 'bends' at some point. In other words, high curvature means that the curve locally deviates quite a bit from a straight line, whereas small curvature implies that a straight line could be an adequate representation. We visualized the phenomenon in Figure 4a for a cubic spline, where the two curves have the same arc length, but different curvature.

Arc length represents the other element relevant for plotting smooth curves. Consider for instance two curves where one is a magnified (scaled) version of the other. When magnifying, arc length increases, and any un-smooth detail such as a corner will occupy more visual area and stand out. See the cubic splines in Figure 4b for an example. We sum up these intuitive remarks in the following observation.

Observation 2.4. *Areas of a curve with high curvature need shorter segments, i.e., more plotting points. Also, given two curves with the same curvature and different arc length, the curve with the higher arc length requires more segments.*

2.2.2 Proximity of an approximation

Consider the curve segment (red) and the two proposed approximations (black and blue) in Figure 5. The approximation to the left oscillates and does not reflect the smoothness of the curve it approximates, but the approximation is still fairly close to the curve. The approximation to the right is further away from the curve, but gives a better image of the general allure of it. Intuitively, in a such a visual setting, most people will claim that the rightmost approximation better represents the original curve.

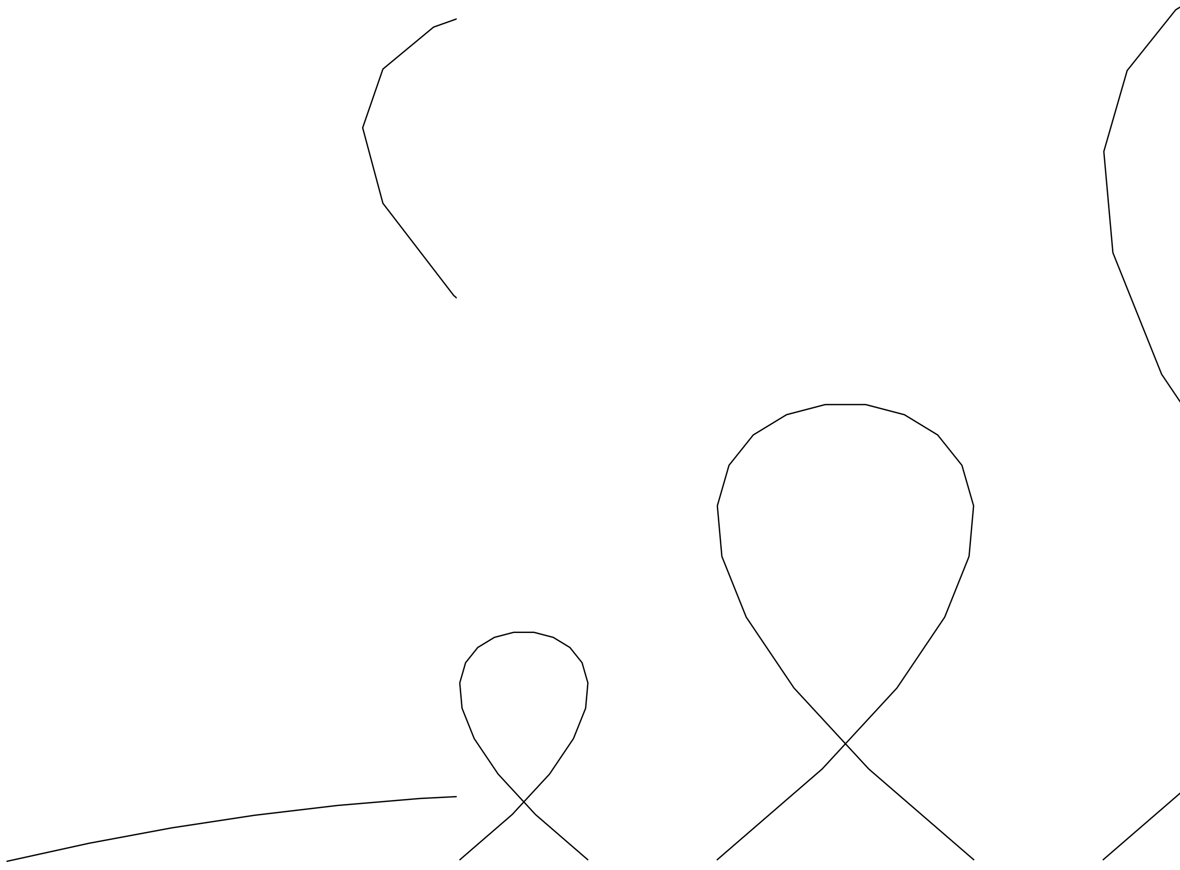
This somewhat subjective judgement should be reflected in the norm we choose to measure the proximity of an approximation. In the general case, it is not trivial to define a norm which incorporates both shape resemblance (geometry) and distance error (function values). In the case of spline approximation by the control polygon, however, from the convexity properties of the control polygon, we know that the situation to the left in Figure 5 will not occur: the geometric shape resemblance is guaranteed. When choosing a norm, we may thus restrict candidates to norms based on customary distance.

A precise method computes in every parameter value the distance between the curve f and the approximation g, then uses integrals to give an expression of a mean deviation. Such an expression would have the form

$$L_2(f,g) = \sqrt{\frac{1}{t_n^* - t_1^*} \int_{t=t_1^*}^{t_n^*} |f(t) - g(t)|^2 \, \mathrm{d}t}. \tag{2.1}$$

A drawback with this norm is that poor performance in certain areas can be masked by great performance in other areas. A safer option is basing the norm on some maximum deviation

$$L_\infty(f,g) = \max_{t \in \left[t_1^*, t_n^*\right]} |f(t) - g(t)|. \tag{2.2}$$



(a) Same arc length, different curvature *(b) Different arc length, same curvature*

Figure 4: *Visual smoothness depends on curvature and arc length. Figure reworked from [Eri93].*

If this value is small enough, i.e., corresponding to only a few pixels, visual (human) confirmation of the adequacy of the approximation should not be necessary.

In an initial stage of developing new plotting methods, there may be a great number of tests to perform. In the case where the above computation is limiting with our available resources, we can do even simpler. The following definition may be used to filter out clearly inadequate representations.

Definition 2.5 (Control polygon approximation error)**.** For a spline f, the *control polygon approximation error* is written $\|\Gamma_f - f\|_{CP}$ and is given by

$$\|\Gamma_f - f\|_{CP} = \max_{i=1}^{n} \|\gamma_f(t_i^*) - f(t_i^*)\|.$$

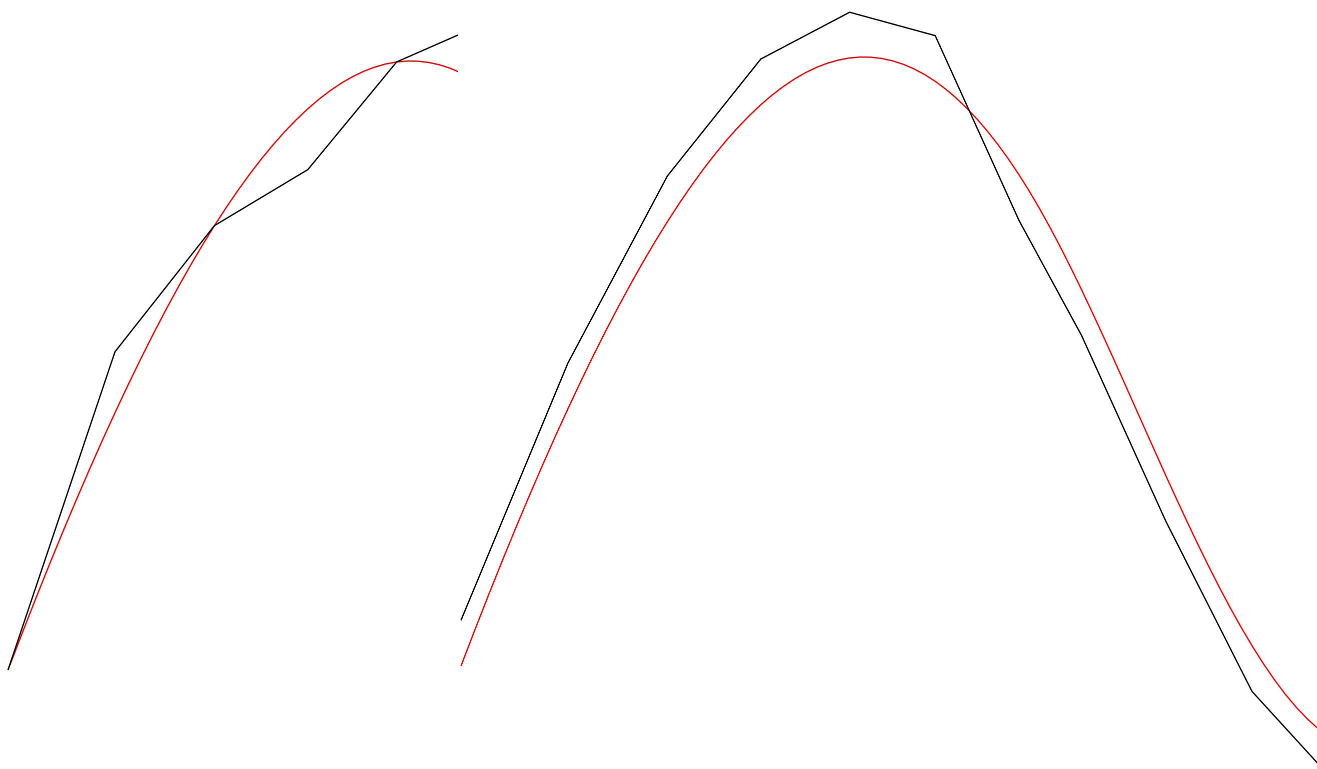


Figure 5: *Comparing different approximations: shape resemblance matters. A 'good' screen approximation mimics the geometric shape of the original curve.*

2.3 Existing plotting techniques based on sampling

An intuitive and common way of plotting a curve is to evaluate the function or parametric curve in a number of points and draw straight lines between these points. Below follows a description of a few different methods for how the sample points can be chosen. The interval to represent is $[a, b) \subset [t_{p+1}, t_{n+1}]$ and we are allowed to use at most N points to represent the graph.

2.3.1 Global uniform sampling

A very straightforward method is a uniform sampling of the interval $[a, b)$. The sample values x_i are then given by

$$x_i = a + (i-1)h, \qquad \text{for} \quad i = 1, \dots, N, \tag{2.3}$$

with the *step length* h given by

$$h = \frac{b-a}{N-1}. \tag{2.4}$$

This method is used extensively in all commercial software. For instance, the MATLAB Curve Fitting Toolbox contains a function `fnplt` for plotting

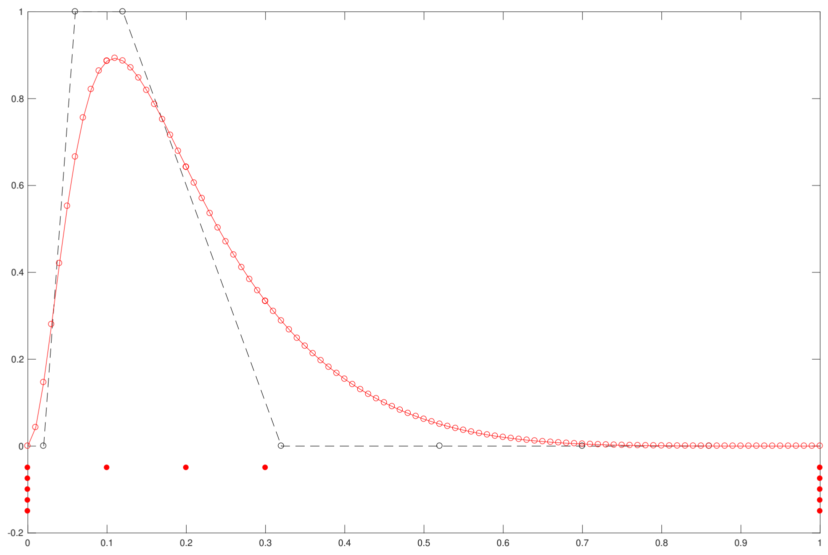


Figure 6: *Uniform sampling leading to unnecessary computations. The many samples in the left part of the curve are justified, but the right part of the curve could be equally well represented using fewer points. Example of quartic (p=4) spline.*

of splines which uses the value $N = 101$ [Mat16]. Values for N of this size provide good results in many situations, but with this naive method there is no distinguishing of parts of a curve which are 'straight' and parts with high curvature. This means we waste computational resources that could be better spent on other areas of the curve. See Figure 6 for an example.

Another drawback with such a method is the portability from functions to parametric curves. When dealing with functions, the N values are spread out along the horizontal axis and the curve moves across the screen only once. When dealing with parametric curves, the situation is different, as we may encounter splines which twist and twirl many times across the screen, giving room for even more details and a much longer arc length, thus requiring more sample points for the illusion of smoothness. Figure 7 shows how the MATLAB default value N leads to a plot of poor quality in such a situation.

2.3.2 Local uniform sampling: simple approach

One way to remedy these flaws is to use information in the object to plot when distributing the sample points, i.e., we construct consecutive subintervals $[a_i, b_i)$ of $[a, b)$ and keep the samples uniform on each subinterval. The number of samples N_i per subinterval $[a_i, b_i)$ may be fixed or vary, but we should have $\sum N_i \approx N$. For general objects, the subintervals may be chosen on the basis of a symbolical analysis of the functional expression, such as identification of elementary functions and exponents giving hints about variations and derivatives.

For splines, the knot intervals $[t_i, t_{i+1})$ are in a first instance a candidate for the subintervals $[a_i, b_i)$. After checking for empty intervals (multiple knots),

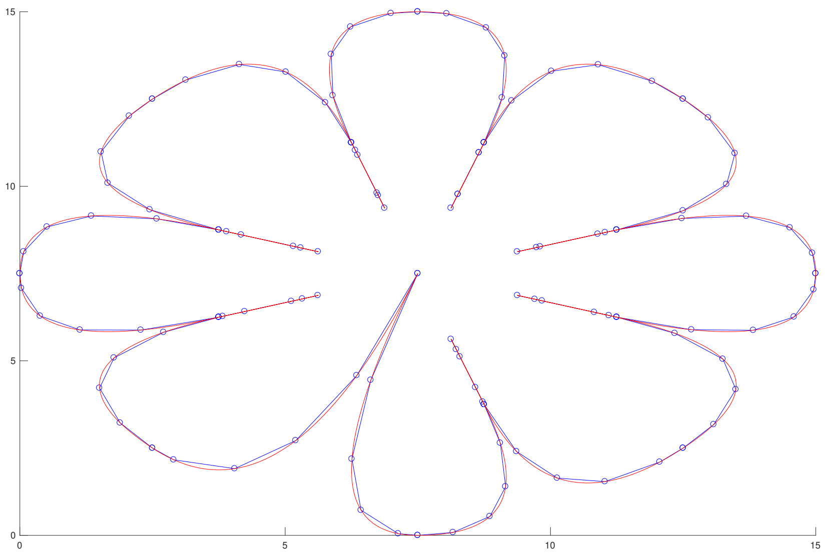


Figure 7: *Uniform sampling might miss features. The spline is plotted using a uniform sampling of $N = 101$ points (blue). Some features of the actual spline (red) are lost due to a long arc length.*

the step length on interval $[t_i, t_{i+1})$ would be given by

$$h_i = \frac{t_{i+1} - t_i}{N_i - 1}. \tag{2.5}$$

The value N_i is kept fixed as long as we don't use more information about the spline. A drawback with this method stems from a potentially misleading relationship between the knot vector and the geometry of the curve.

For instance, in Figure 8, the three parabolic pieces on a uniform knot vector were obtained by generating a quadratic spline with only one active coefficient. We then inserted a few knots, an operation which does not alter the graph of the spline, but according to the scheme above we would plot the three very similar pieces of the spline differently. In other words, this method does not pick up on the spline's variations as intended.

The idea of local uniform sampling will be further developed in Section 3.2, where we use more information from the spline than simply the parameter/knots.

2.3.3 Adaptive sampling

The general impression is that most software developers rarely make use of more advanced methods to plot functions. In the widely used cross-platform Application Programming Interface OpenGL for instance, curves are not a geometric primitive. Drawing curves thus means sampling the function and then calling a line drawing function `GL_LINES`, for instance by the commands

```
glBegin(GL_LINES);
        glVertex2f(0, 0);
        glVertex2f(1, 1);
glEnd();
```

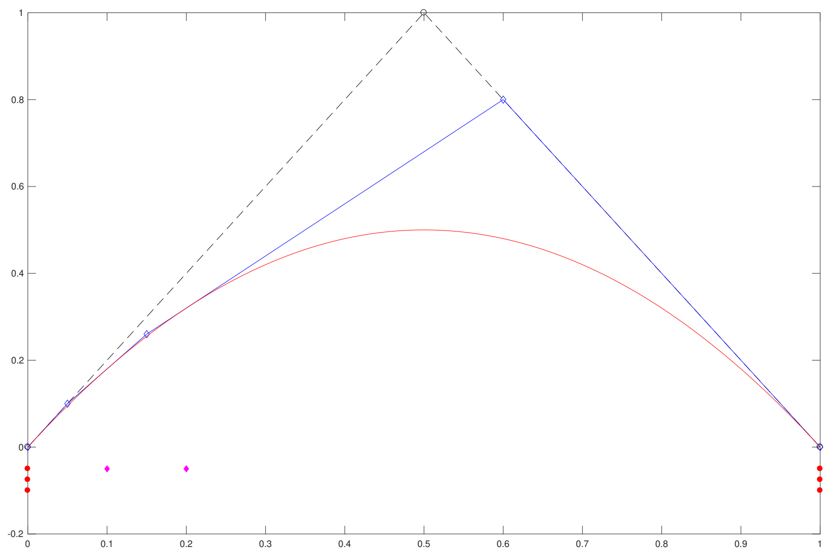


Figure 8: *A complicated knot vector but a simple geometric shape.*

which draws a line from the center of the viewing area (screen or window) to the top right of the viewing area [Khr19b; Shr+13]. There is no procedure 'GL_SPLINES' in the standard libraries, meaning splines are treated as any other object. The situation is similar for same-purpose APIs such as Microsoft's Direct3D [Mic19b], Khronos' Vulkan [Khr19a] and Apple's Metal [App19], although Microsoft's graphics device interface GDI+ includes a procedure `graphics.DrawCurve` for drawing cardinal splines (the knots are integers) and Bézier curves [Mic19a]. The implementation of the latter is not publicly available and possibly a business secret, but the author has no indication that any sophisticated method is used.

As to other software, we observe that the general purpose programming language Java includes quadratic and cubic Bézier curves in their collection of geometric primitives [Ora19] through the classes QuadCurve2D and CubicCurve2D. These specific objects are rendered in a particular fashion: After an initial sampling producing line segments, the distance from each segment to control points is measured. If the distance exceeds a threshold (default value is 15 pixels), the curve segment is recursively subdivided in a left and right part producing shorter segments and new control points. We inspire from this method in Section 3.2.3.

The most notable exception as to plotting techniques is nevertheless found in the Wolfram Mathematica scientific computing system[2]. The following is a high-level description of their plotting algorithm [Wol19a; Wol19b]:

1. The function is evaluated at N_0 uniformly spaced values.
2. The angle between successive line segments is examined. If the angle between two consecutive segments is less than α_δ, the algorithm continues

[2]urlhttps://www.wolfram.com/mathematica/

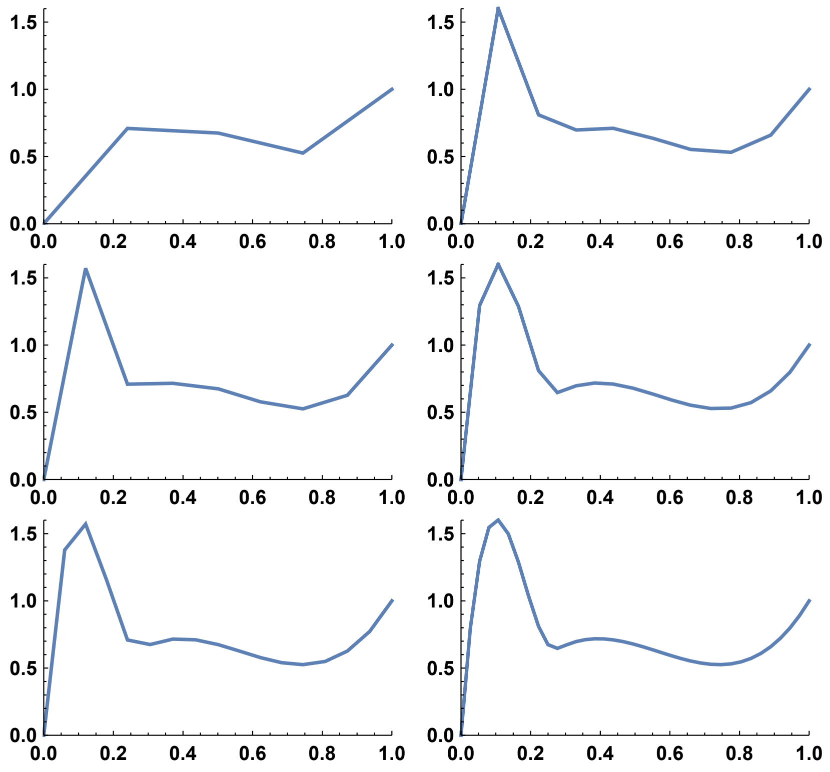


Figure 9: *Mathematica's adaptive plotting function. Plotting the same spline by varying the number of initially sampled points and the maximum number of allowed recursions in the adaptive method. See the text for details.*

to the next angle. If not, it recursively subdivides the domain of the two segments holding the angle.

3. The recursion continues until the angle criterion or the recursion depth limit Λ is reached.

This means the algorithm in a first instance conducts a rough uniform sampling over the domain of interest, then refines regions of high curvature using geometric arguments. The user can control the plotting parameters N_0 (PlotPoints), α_δ (accessed through ControlValue) and Λ (MaxRecursion). A code example is provided in Appendix A. The effect of these plotting parameters can be observed in Figure 9, where we plotted a cubic spline. In the plots in the left column, the spline is initially sampled with $N_0 = 5$ points, whereas in the right column plots we sampled the spline using $N_0 = 10$ points. The top, middle and bottom plots were refined using maximum $\Lambda = 0, 1$ and 2 recursive subdivisions, respectively. The best result is obtained in the bottom right plot. We observe that this plot provides adequate (but not good) quality using an impressively low 2 recursive subdivisions.

CHAPTER 3

Usage of a control polygon for plotting

The purpose of this chapter is to develop new ideas to improve the efficiency of plotting splines. As stated in Section 1.1.3, evaluation of a spline given as a weighted sum of B-splines is quadratic in the degree of the spline, thus the necessity of simplicity in any devised method to ensure any additional calculation does not exceed the cost of simply plotting (evaluating) more points using conventional methods. We will not consider the idea of raising the degree of the spline [Far97, p. 67] nor using derivative algorithms [SSF94], as these methods are already well studied.

Instead, the key to success is here assumed to lie in the easily manipulable control polygon, which by the convexity and convergence properties stated in Section 1.2 governs the spline. Hence, by drawing the control polygon, we can under reasonable assumptions guarantee that all details in the spline are reproduced without resorting to quasi-redundant and/or costly evaluations. This observation of spline curves will enable us to circumvent the general difficulty of knowing whether an approximation of a random curve is 'good enough'. We also note that such an approach to plotting gives an almost immediate rough representation of the curve on the entire plotting interval, which is advantageous for animation and real-time rendering.

We begin our discussion with a description of classic manipulations of the control polygon. Then follows an analysis of several mostly geometric aspects of the control polygon. After a few results on how the control polygon is affected by knot insertion, we propose a few criteria and rules for where knots should be inserted. Implementation examples of algorithms are given in Appendix A.

3.1 Cutting corners and subdivision

This is a classic approach where we start with a control polygon and conduct successive refinements on it until it is considered smooth enough to represent a curve [Boo87]. The methods mentioned here may be termed *geometric construction algorithms*, since they treat the control polygon purely as a geometric object and are independent of the underlying mathematical description.

The idea of Chaikin's algorithm [Cha74] is to cut the corners of the control polygon at a quarter of the length of the two adjoining segments in each round.

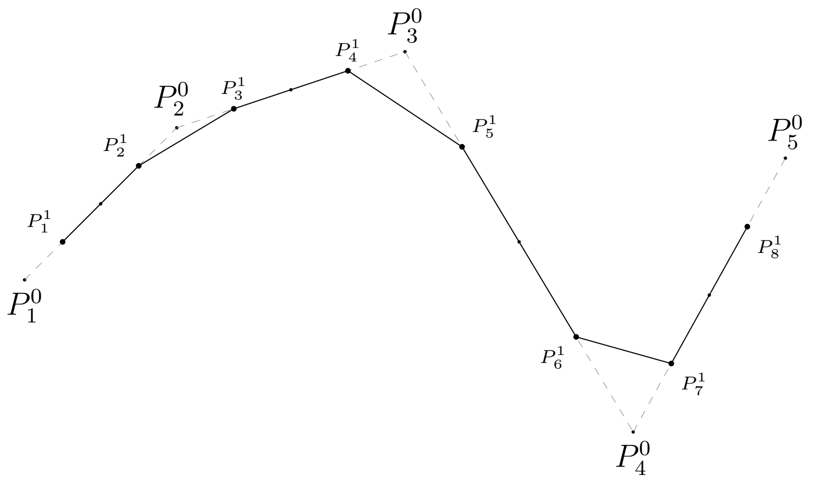


Figure 10: *Principle of Chaikin's algorithm. Each corner is cut at a quarter of the length of an adjoining segment. All iterations are the same and one iteration is shown.*

The end points of the control polygon are also treated as if they were corners. See Figure 10. If after round k we have the control points $\{P_i^k\}$, we may describe this method using barycentric logic on the inner segments to obtain the formulas

$$P_{2i}^{k+1} = \frac{1}{4}\left(3P_i^k + P_{i+1}^k\right) \tag{3.1}$$

$$P_{2i+1}^{k+1} = \frac{1}{4}\left(P_i^k + 3P_{i+1}^k\right). \tag{3.2}$$

Both for estimating the number of rounds that should be sufficient to properly represent a curve and for efficient implementation purposes, we now count the number of points obtained by this method. Starting with n points, each of the $n-2$ inner points (corners) are cut, giving $2n-4$ inner points. Adding the two new end points, we obtain $2n-2$ points in the new control polygon when the round is complete. The number of points N_k after round k thus satisfies the recurrence relation $N_{k+1} = 2N_k - 2$, with initial condition $N_0 = n$. Using standard methods, we find the explicit expression

$$N_k = (n-2)2^k + 2. \tag{3.3}$$

These points are obtained using only few computational resources, i.e., few arithmetic operations. More precisely, the computation of each new point requires only 1 addition and 2 multiplications for each dimension, in our case two dimensions, and this regardless of any other parameter such as spline degree. The total number of arithmetic operations Q_k involved to find the N_k points is thus

$$Q_k = 3 \cdot 2 \sum_{i+1}^{k} (n-2)2^k + 2 \tag{3.4}$$

$$= 12k + 12(n-2)(2^k - 1). \tag{3.5}$$

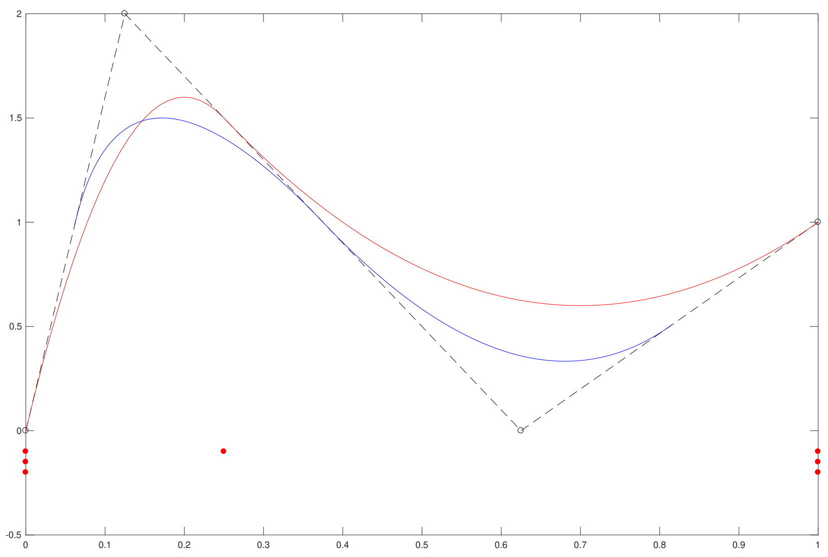


Figure 11: *Chaikin's method does not in general converge to the spline. The method converges to a quadratic Bézier curve (blue), not the spline curve (red) associated with the original control polygon (dashed black). The spline is here also quadratic, and greater differences are in general observed for other spline degrees.*

This gives a ratio of the number of operations per final point $R_k = Q_k/N_k$, with asymptotic value 12 (reached after only 5-7 rounds for typical values of n $(5 \leq n \leq 30)$), meaning the cost of each final point used in the representation is close to 12 operations. We state this result here for later reference.

Proposition 3.1 (Computational cost in Chaikin's algorithm)**.** *The cost of each final point obtained with Chaikin's algorithm is close to 12 operations in practical situations.*

For the example provided in Figure 10, we have $n = 5$, thus to exceed $N_k = 100$ points, we only need $\kappa = 6$ rounds, giving $N_6 = 194$ points. The number of operations involved for obtaining these points is $Q_6 = 2340$ giving an average cost of $R_6 \approx 12.06$ operations per final point.

The limit curve obtained by this method is a quadratic Bézier curve [Rie75], i.e., a spline curve of degree $p = 2$ where all knots are of multiplicity $p + 1 = 3$. As clearly shown in the example in Figure 11, the Chaikin limit curve and a spline curve may differ greatly, let alone the end points, and Chaikin's method is thus not a reliable and satisfactory method for general plotting of splines.

It is of course possible to envision varying the cutting points of the segments to a different ratio, as exemplified in Figure 12. Another idea is to couple the subdivision cutting ratio with ratios in the knot vector, hoping to obtain more general kinds of limit curves. It turns out that such regimes do produce other types of limit kurves, namely rational B-splines [GW93; Rie75]. In any case, these curves are not what what we are looking for in this context, and the idea of traditional corner cutting by subdivision will here not be pursued further. For more information on subdivision methods in general, see [Cav+91].

One comment is nevertheless in order. In a very time-constrained

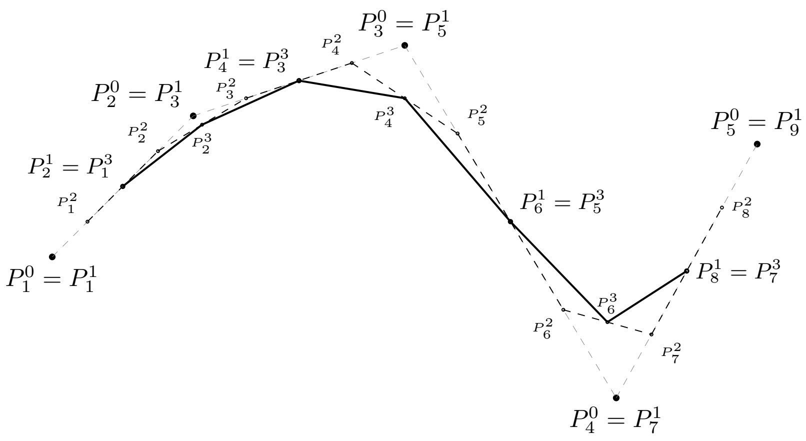


Figure 12: *Subdivision method: midpoint refinement. Three steps of an algorithm proposed by Lane and Riesenfeld [LR80]. With this method, the control polygon is refined in each iteration by finding and using middle points of segments. More precisely, the first iteration refines the original control polygon by adding control points on the midpoint of every control polygon segment. Subsequent iterations cut the corners of the previous control polygon at half the length of the adjoining segments. The method provably converges fast for B-splines with uniform knot vectors and B-splines with only* $(p+1)$*-multiple knots (Bézier curves).*

environment, the simplicity and efficiency of such geometric methods could be of interest, as they quite immediately give a rough sketch of the desired curve.

3.2 Geometric aspects of the control polygon

We now analyse different geometric properties of the control polygon and how these properties may be related to the spline and used in computations. The closeness in shape between the spline and its control polygon is suggested by Theorem 1.17.

The strategies are sorted roughly according to the complexity of the precalculations. That is, the simplest, most naïve method requiring the least computations is placed first. For clarity, we introduce the following notation.

Notation 3.2. Let $\{\mathbf{P}_i\}_{i=1}^n$ be the control points of a spline $f = \sum_{j=1}^n c_j B_{j,p,\mathbf{t}}$ with coefficients in $\mathbb{R}^s$, $s = 1$ or 2. We note $\mathbf{P}_i^j$ the jth component of control point i. We also introduce $\Delta\mathbf{P}_i^j = \mathbf{P}_{i+1}^j - \mathbf{P}_i^j$.

We rarely need to consider the entire control polygon at the same time. From Lemma 1.2 we know that on any interval $[t_\mu, t_{\mu+1})$ only $p+1$ B-splines contribute, namely the B_j with index $j = \mu - p, \ldots, \mu$. This motivates the definition of the local control polygon of the spline segment associated with that knot interval, similarly with the convex hull property in Lemma 1.14.

Definition 3.3 (Local control polygon)**.** Let $f(t) = \sum_{j=1}^n c_j B_{j,p,\mathbf{t}}$ be a spline and Γ its control polygon. Given a knot index μ and a range parameter $r \in \{1, 2, \ldots, p+1\}$, we define the *local control polygon* $\Gamma_{\mu,r}$ as the restriction of

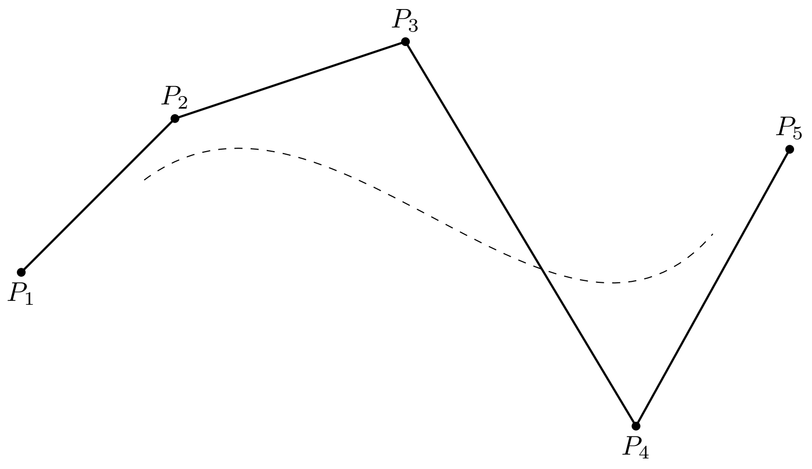


Figure 13: *Local control polygon. Restriction of a spline to one knot interval.*

Γ to the r vertices $\{\mathbf{P}_i\}_{i=\mu-p}^{\mu-p+r-1}$ and their joining edges. The *full local control polygon* is obtained when $r = p+1$, in which case we note $\Gamma_\mu = \Gamma_{\mu,p+1}$.

Figure 13 shows a full local control polygon and the corresponding active spline segment of the curve for $p = 4$. In this section, we will hardly need the value of the knot interval index corresponding to a local control polygon and will use a simple control point numbering. The above definition will first find its active use in Section 3.3 when we insert knots.

We will not always want to consider every consecutive local control polygon as they may overlap and thus, with some methods, give redundant information. For instance, $\Gamma_{\mu,r}$ and $\Gamma_{\mu+1,r}$ have $r-1$ vertices and $r-2$ edges in common. This motivates an alternative partitioning of the control polygon.

Definition 3.4 (Partitioning of a control polygon)**.** Let $f = \sum_{j=1}^{n} c_j B_{j,p,\mathbf{t}}$ be a spline and $S < n$ a number of intervals. Let $\{\sigma(i)\}_{i=0}^{S}$ specify $S+1$ strictly increasing indices of the knot average vector with $\sigma(0) = 1$ and $\sigma(S) = n$. Then $\Sigma = \{\sigma(i)\}_{i=0}^{S}$ defines a *partitioning of the control polygon.*

It remains to describe how to choose S and Σ. For Bézier curves, a natural choice is the inner break points which hold knots of multiplicity $(p+1)$, allowing curve splitting. We could inspire from this and set the inner break points to all knots with higher multiplicity than one, but this excludes all splines with only simple knots. Another possibility is simply to let σ take on the values of the inner knot averages or every multiple index of the knot averages, for instance every 3rd or pth knot average.

3.2.1 The length of the control polygon lines

We first look at the length of the segments in the control polygon and their relation to the spline arc length.

Definition 3.5 (Arc length)**.** Let f be a continuous curve from $[a, b]$ to $\mathbb{R}^s$ and $\mathbf{t} = \{t_i\}_{i=1}^{N}$ a uniform partition of that interval. The *length of the curve* f,

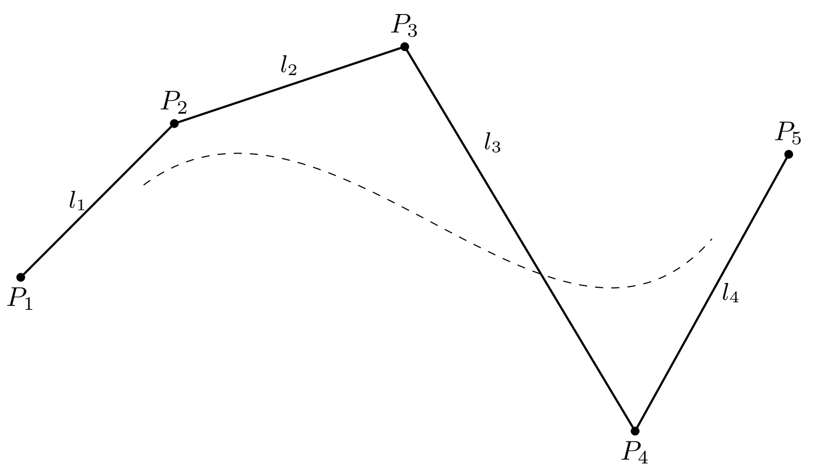


Figure 14: *The length of a local control polygon.*

written $\mathcal{L}(f)$, has value

$$\mathcal{L}(f) = \lim_{N\to\infty} \sum_{i=1}^{N} \|f(t_i) - f(t_{i-1})\|. \tag{3.6}$$

If f is a continuously differentiable function, then by a common transformation we can pass to the integral, but in our setting, if we wanted to use arc length in a computation, these kind of expressions involve many spline evaluations and are therefore very expensive. The corresponding computations on the control polygon, on the other hand, are much cheaper.

Definition 3.6 (Length of spline control polygon)**.** Let $\{\mathbf{P}_i\}_{i=1}^n$ be the control points of a spline $f = \sum_{j=1}^n c_j B_{j,p,\mathbf{t}}$ with coefficients in $\mathbb{R}^s$, $s = 1$ or 2. The length of the control polygon between $\mathbf{P}_\mu$ and $\mathbf{P}_\nu$ $(\mu < \nu)$ is

$$L_{\mu,\nu} = \sum_{i=\mu}^{\nu-1} l_i, \tag{3.7}$$

with

$$l_i = \sqrt{(\Delta\mathbf{P}_i^1)^2 + (\Delta\mathbf{P}_i^2)^2}. \tag{3.8}$$

The length of the entire control polygon is given by $L(f) = L_{1,n}$.

The intuition that the length of the control polygon might be a reasonable approximation to the length of the spline is further supported by the following result, due to [KH95]. A simple illustration of the situation is provided in Figure 14.

Lemma 3.7. *Let* $\mathbf{t}$ *be a* $(p+1)$*-regular knot vector and* $f = \sum_{j=1}^n c_j B_{j,p,\mathbf{t}}$ *a spline. The length of the control polygon* $L(f)$ *converges quadratically to the arclength* $\mathcal{L}(f)$ *of the spline under insertion of a dense set of knots.*

We now first pursue the strategy of local uniform plotting (without knot insertion), but allow for more flexibility than in Section 2.3.2 by extracting additional information about the spline in order to find adequate subintervals $[a_i, b_i)$ and a reasonable rule for varying the number of points N_i per subinterval. The idea is taken from [Eri93].

Assuming a partitioning Σ of the control polygon, the average control polygon segment length $\bar{L}$ (independent of Σ) can be written

$$\bar{L} = \frac{1}{S} \sum_{i=0}^{S-1} L_{\sigma(i),\sigma(i+1)}. \tag{3.9}$$

For an index such that $L_{\sigma(i),\sigma(i+1)} \approx \bar{L}$, we would want the number of plot points close to $\bar{N} = \frac{1}{S}N$, for instance by rounding down to the nearest integer $\lfloor \frac{1}{S}N \rfloor$. The fraction of the N allowed points across the intervals is then set proportionately to the local control polygon length compared to the total length of the entire control polygon. The number of plot points per subinterval is then

$$N_i = \max \left\{ N_{min}, \left\lfloor \bar{N} \frac{L_{\sigma(i),\sigma(i+1)}}{\bar{L}} \right\rfloor \right\}, \tag{3.10}$$

where the max-function ensures we have at least N_{min} points per subinterval, typically $N_{min} = 2$.

We want this precomputational setup to be as cheap as possible, and one could skip the challenging square root in the definition of l_i without changing the mechanism of the method. Without the square root and assuming dimension $s = 2$, we need 3 additions and 2 multiplications per term, that is approximately $5n$ arithmetic operations to find the $L_{\sigma(i),\sigma(i+1)}$. In addition comes S operations to find $\bar{L}$ and S operations to compute the N_i, provided the max- and floor-functions are free and we buffer the fraction $\bar{N}/\bar{L}$. For a worst case $S \sim n$, the precomputations for testing lengths then require approximately $7n$ operations.

In computer graphics, there is a common approximation used when computing with lengths [Rap91], which can reduce the cost of computing the values in Equation (3.7).

Observation 3.8. *Given points $\mathbf{P}_i, \mathbf{P}_{i+1}$ one can estimate the length l_i of the segment $\mathbf{P}_i\mathbf{P}_{i+1}$ by $\tilde{l}_i \approx l_i$ defined by*

$$\tilde{l}_i = \max \left\{ \Delta\mathbf{P}_i^1, \Delta\mathbf{P}_i^2 \right\} + \frac{1}{2} \min \left\{ \Delta\mathbf{P}_i^1, \Delta\mathbf{P}_i^2 \right\}, \tag{3.11}$$

The computation of $\tilde{l}_i$ requires only 1 addition, a cheap bit shift for the division and two fairly cheap max/min operations. In total, this means the equivalent of about 2 arithmetic operations.

Applying this alternative simplification almost halves the overall complexity of the above algorithm.

3.2.2 The angles of the local control polygon

From Observation 2.4 we established that length considerations alone are unlikely to produce satisfactory results. They must be coupled with considerations of other properties, such as curvature. Geometrically, curvature measures the rate of change of direction of a curve. In fact, the link between a curve and its curvature is well studied in classic geometry theory.

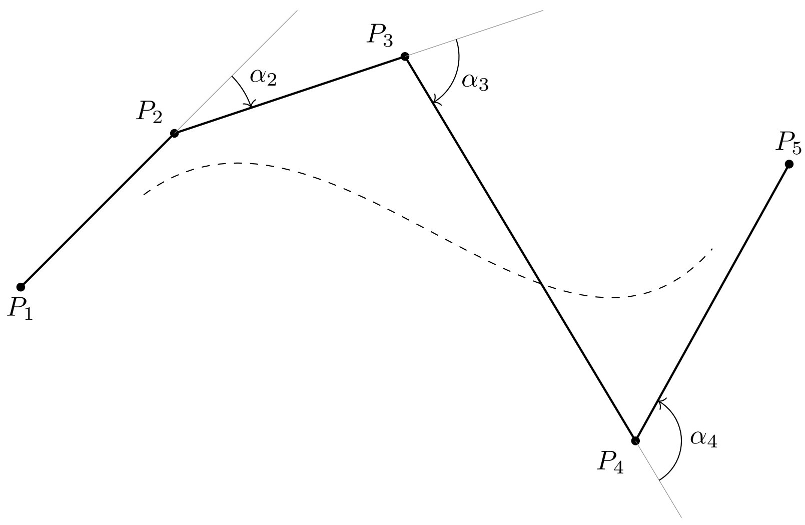


Figure 15: *The angles of a local control polygon. The angle of change from a straight line is taken as an approximation of curvature.*

Theorem 3.9 (The fundamental theorem of curves)**.** *The shape of a plane curve is completely determined by its curvature, and under some differentiability conditions, for a plane curve $f(t)$, the curvature κ is given by*

$$\kappa = \frac{|\det(f', f'')|}{\|f'\|^3}. \tag{3.12}$$

As we saw for lengths, this expression is computationally expensive, as it involves two spline evaluations as well as other computations for every point at which we want to compute curvature. We need something cheaper, and again the simple geometry of the control polygon inspires an approximation scheme. More precisely, although not perfectly correlated, the angle of change of a corner of the control polygon mimics the directional change of the curve. See Figure 15. The expression for the angles in Definition 3.11 are found using the following dot product identity:

Lemma 3.10. *The geometric and algebraic definitions of the dot product are equivalent. That is, given two vectors $\Delta\mathbf{P}_{i-1}, \Delta\mathbf{P}_i$ and noting α_i the angle between them and $\|\cdot\|$ the Euclidean norm, we have*

$$\Delta\mathbf{P}_{i-1} \cdot \Delta\mathbf{P}_i = \Delta\mathbf{P}^1_{i-1}\Delta\mathbf{P}^1_i + \Delta\mathbf{P}^2_{i-1}\Delta\mathbf{P}^2_i = \|\Delta\mathbf{P}_{i-1}\|\|\Delta\mathbf{P}_i\| \cos\alpha_i. \tag{3.13}$$

Definition 3.11 (Angle of spline control polygon)**.** Let $\{\mathbf{P}_i\}_{i=1}^n$ be the control points of a spline $f = \sum_{j=1}^n c_j B_{j,p,\mathbf{t}}$ with coefficients in $\mathbb{R}^s$, $s = 1$ or 2. The total angle of the control polygon between $\mathbf{P}_\mu$ and $\mathbf{P}_\nu$ $(\mu < \nu)$ is

$$A_{\mu,\nu} = \sum_{i=\mu}^{\nu-1} \alpha_i, \tag{3.14}$$

with

$$\alpha_i = \arccos\left(\frac{\Delta\mathbf{P}^1_{i-1}\Delta\mathbf{P}^1_i + \Delta\mathbf{P}^2_{i-1}\Delta\mathbf{P}^2_i}{\sqrt{(\Delta\mathbf{P}^1_{i-1})^2 + (\Delta\mathbf{P}^2_{i-1})^2}\sqrt{(\Delta\mathbf{P}^1_i)^2 + (\Delta\mathbf{P}^2_i)^2}}\right) \tag{3.15}$$

for $i = 2, \dots, n-1$ and $\alpha_1 = \alpha_n = \frac{\pi}{2}$.

As the lower limit of summation in the definition shows, we need one point preceding $\mathbf{P}_\mu$ and one point succeeding $\mathbf{P}_{\nu-1}$ to compute the inner angles. This choice makes the angle-method less sensitive to the choice of Σ since the overlap in use of points makes sure we also compute angles at the joints $\mathbf{P}_{\sigma(i)}$. We note that each α_i takes values in the interval $[0, \pi)$. The end-point angles α_1 and α_n are defined so as not to discriminate nor overcompensate the end-segments.

Following the same logic as for lengths and assuming subintervals given by Σ, we define the average control polygon angle $\bar{A}$ (independent of Σ) by

$$\bar{A} = \frac{1}{S}\sum_{i=0}^{S-1} A_{\sigma(i),\sigma(i+1)}. \tag{3.16}$$

We could also allow a point distribution according to the equation

$$N_i = \max\left\{N_{min}, \left\lfloor \bar{N}\frac{A_{\sigma(i),\sigma(i+1)}}{\bar{A}} \right\rfloor\right\}. \tag{3.17}$$

This distribution, however, is likely to give unwanted bias, especially if there are i such that $\sigma(i)$ and $\sigma(i+1)$ are close. Let's say for instance $A_{\sigma(i),\sigma(i+1)} = A_{\mu,\mu+1}$. Then the number of points awarded to $[t_\mu, t_{\mu+1})$ depends only on α_μ, not on $\alpha_{\mu+1}$. One solution is to modify Equation (3.14) to include an extra term α_ν. The method would then depend more on Σ, but exhibit more symmetry.

Now to the cost, which might seem high due to the presence of division, square root and arccos. A few sources point to a rough estimate for each of these operations [Mar19; Mar90; Str99]. We will for ease of comparison between methods make use of this estimate.

Observation 3.12. *The operations of division, $\sqrt{\cdot}$ and* arccos *each cost about the same as 10 multiplications on modern (2019) chipsets. The $\frac{1}{\sqrt{\cdot}}$-operation can also be counted as 10 multiplications.*

Each α_i requires 3 additions, 7 multiplications, 2 inverse square roots and 1 inverse cosine. According to Observation 3.12, this means a comparative 40 operations per computed value, and thus 40n operations to find the $A_{\sigma(i),\sigma(i+1)}$. In addition comes as before the S operations to find $\bar{A}$ and the S operations to compute the N_i. In conclusion, for a worst case $S \sim n$, the precomputations for testing angles require approximately 42n operations.

We can try to mitigate the cost of both the arccos function and the square-root function. In Figure 16 we applied the transformation $T(x) = -\frac{\pi}{2}x|x| + \frac{\pi}{2}$ (blue) to mimic the arccos-function (red). To obtain the square of the arccos-argument in each α_i, we need 3 additions, 8 multiplications and 1 division. Saving the sign of the numerator before squaring, we can now count 1 multiplication and 1 addition for the transformation T. Counting the division as above, this means approximately 23 operations per angle and a complexity of

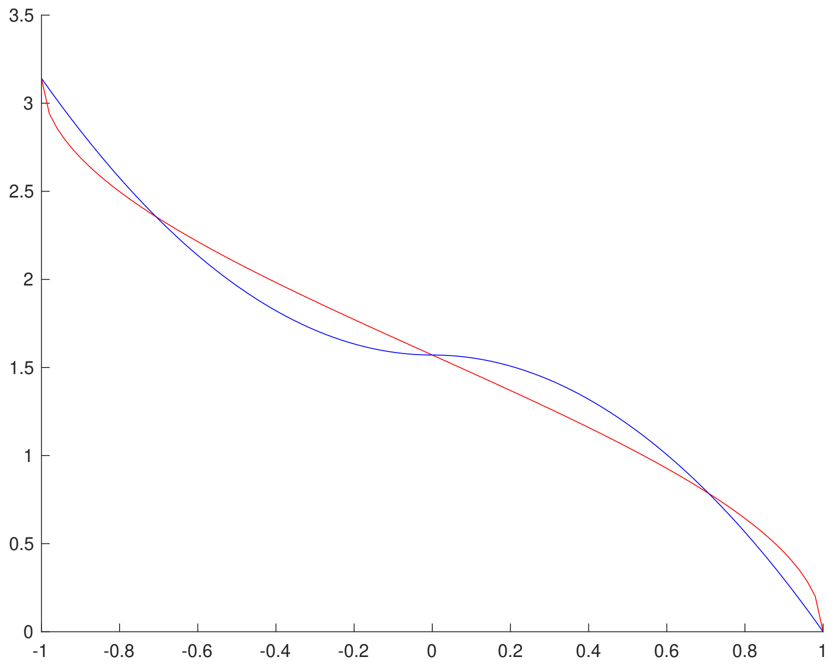


Figure 16: *Mimicing the* arccos*-function (red) to save computational cost.*

25n for the entire scheme. There is no order of magnitude saved, but there is a considerable 40% reduction cost that could pay off provided the approximation is not too rough.

Returning to Observation 2.4, we are motivated to combine the two previous tests, such that the number of points awarded N_i is estimated on the basis of considerations of both lengths and angles. This can be done for instance by means of a weight $\omega \in [0, 1]$ giving the rule

$$N_i = \max\left\{N_{min}, \left\lfloor \bar{N}\left(\omega\frac{L_{\sigma(i),\sigma(i+1)}}{\bar{L}} + (1-\omega)\frac{A_{\sigma(i),\sigma(i+1)}}{\bar{A}}\right)\right\rfloor\right\}. \qquad (3.18)$$

The combined test reduces to using only the angles when $\omega = 0$, and to using only the lengths when $\omega = 1$.

One apparent weakness with this method and similar renditions of it using other properties of the local control polygon such as those mentioned below, is the cases where the number of knot intervals is small, for instance 1 or 2. Such cases would need a preprocessing stage where knots are inserted to obtain a satisfactory number of knot intervals, say at least 10. For this reason, we will abandon the idea of local uniform plotting.

Scoping back to the pure angle method, we must point out the lack of sensitivity to sign changes in oriented angles. Under the assumption that the curve follows the variations of the control polygon, two consecutive angles of the control polygon with opposing signs signifies the existence of an inflexion point in the same area, meaning the directional rate of change (derivative in the functional case) is close to 0. On the other hand, the presence of two consecutive angles with the same sign signifies high curvature. In Figure 15, the distinction can be observed between angles at P_2 and P_3 (same sign) and at P_3 and P_4

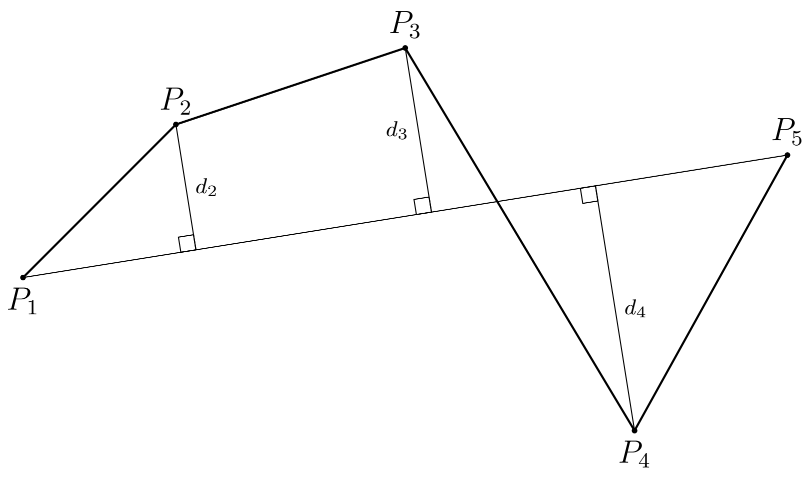


Figure 17: *Base line $[P_1P_5]$ of a local control polygon and the distances to the base line from the inner control points.*

(opposing sign). A closer study of dot products will address this issue, as shown in the next section.

3.2.3 The base line of the local control polygon

The concept of base line is taken from a remark in [BBB95] and is visualized in Figure 17. As mentioned in Section 2.3.3, the programming language Java already exploits the idea for lower degree Bézier curves. The method is based on measuring how *flat* the control polygon is, not through the use of angles, but of distances. A flat control polygon means that the spline does not vary much in that area, and that both the spline and the control polygon can be approximated by a line, or at least that they should not require many points to be represented faithfully.

Definition 3.13 (Base line)**.** Let $\{\mathbf{P}_i\}_{i=1}^n$ be the control points of a spline $f = \sum_{j=1}^n c_j B_{j,p,\mathbf{t}}$ with coefficients in $\mathbb{R}^s$, $s = 1$ or 2. The *base line* of the control polygon between $\mathbf{P}_\mu$ and $\mathbf{P}_\nu$ $(\mu < \nu)$ is the line segment $\mathbf{P}_\mu\mathbf{P}_\nu$.

When testing for flatness, we must distinguish cases where the inner control points are 'inside' the end points, i.e., that their projection falls on the line segment defined by the end points, and not on the extended line defined by the same points. Simply measuring the distance to the line indiscriminantly would give misleading results, since the latter situation represents a control polygon much more pointed than the former situation. We choose a simple solution.

Definition 3.14 (Distance to base line)**.** Given a local control polygon $\{\mathbf{P}_i\}_{i=\mu}^{\nu}, (\mu < \nu + 1)$ and noting $d(\cdot,\cdot)$ the usual Euclidean distance function, the distance d_i to the base line of a point $\mathbf{P}_i, \mu < i < \nu$ is given by

$$d_i = \begin{cases} d(\mathbf{P}_i, \mathbf{P}_\mu\mathbf{P}_\nu) & \text{if the projection of } \mathbf{P}_i \text{ lies on } \mathbf{P}_\mu\mathbf{P}_\nu \\ \min\{d(\mathbf{P}_i,\mathbf{P}_\mu), d(\mathbf{P}_i,\mathbf{P}_\nu)\} & \text{if not.} \end{cases} \tag{3.19}$$

This means that if the projection of a point $\mathbf{P}_i$ lies outside the local base line, the distance to the nearest end point is taken as the distance to the base line.

The value is obtained using dot products. Since we are only interested in relative coordinates, we can save some computations by first moving the origin to $\mathbf{P}_\mu$, thus obtaining new coordinates $\tilde{\mathbf{P}_i}$ and $\tilde{\mathbf{P}_\nu}$. We then check whether $\tilde{\mathbf{P}_i}$ lies to the right of the origin O on the axis defined by $\tilde{\mathbf{P}_\nu}$ by computing the dot product $\tilde{\mathbf{P}_i} \cdot \tilde{\mathbf{P}_\nu}$. A negative value means that $\tilde{\mathbf{P}_i}$ lies to the left of the origin on that axis, and that the Euclidean distance to $\tilde{\mathbf{P}_\mu}$ should be returned. With a positive dot product, we move the origin to $\tilde{\mathbf{P}_\nu}$ to perform a similar check on that side. If the inner control point $\tilde{\mathbf{P}_i}$ is found to lie between the end points, we find the square distance to the base line by computing the Pythagorean difference between the norm of $\tilde{\mathbf{P}_i}$ and the norm of its projection on $\tilde{\mathbf{P}_\nu}$. As before, we have the option to keep the squared value to save computations.

We count 4 additions for the first change of origin. For each dot product we must count 1 addition and 2 multiplications. Assuming the most common situation is that the inner control points are between the end points, which is also the most expensive situation, we count two dot product checks, and 2 new additions for the second change of origin. For the norm of the projected vector we need 1 addition, 3 multiplications and 1 division. The final distance is obtained using 2 additions and 2 multiplications. The cost of computing the distance to the base line is therefore 11 additions, 9 multiplications and 1 division. Applying the conversion in Observation 3.12 with 1 division having the cost of roughly 10 multiplications, we land at the equivalent of approximately 30 arithmetic operations per point calculated for this algorithm. We could simplify somewhat using Observation 3.8, but the relative gain is negligible.

An improvement of the above scheme checks whether two consecutive inner control points lie on the same side of the base line, a situation which must be considered 'more flat' than the case where two such points lie on either side of the base line. The check can be conducted using the following lemma from elementary geometry.

Lemma 3.15. *Let $(\mathbf{P}_\mu \mathbf{P}_\nu)$ be a line and $\mathbf{P}_i, \mathbf{P}_j$ two points in the plane. Noting $\times$ the cross product, the two points $\mathbf{P}_i, \mathbf{P}_j$ are on the same side of $(\mathbf{P}_\mu \mathbf{P}_\nu)$ if and only if*

$$[(\mathbf{P}_i - \mathbf{P}_\mu) \times (\mathbf{P}_\nu - \mathbf{P}_\mu)] \cdot [(\mathbf{P}_j - \mathbf{P}_\mu) \times (\mathbf{P}_\nu - \mathbf{P}_\mu)] > 0. \tag{3.20}$$

The cross product requires, as the dot product, 1 addition and 2 multiplications, giving after buffering of recurring values 9 additions and 6 multiplications for the above calculation in the plane. If for an inner control point we check only one neighbour, the count is 45 arithmetic operations for this procedure. If we check both neighbours, with some reuse of values, the count is 56 arithmetic operations.

Introducing the notion of base line reveals another possibility for a method checking for flatness. The base line is the shortest path between the local control polygon end points, and the local control polygon must be fairly flat if the total length of the local control polygon is almost as short. Given a partitioning Σ, we can therefore study the ratios

$$r_i = \frac{L_{\sigma(i),\sigma(i+1)}}{d(\mathbf{P}_{\sigma(i)}, \mathbf{P}_{\sigma(i+1)})} \qquad \text{for } i = 0, \dots, S-1. \tag{3.21}$$

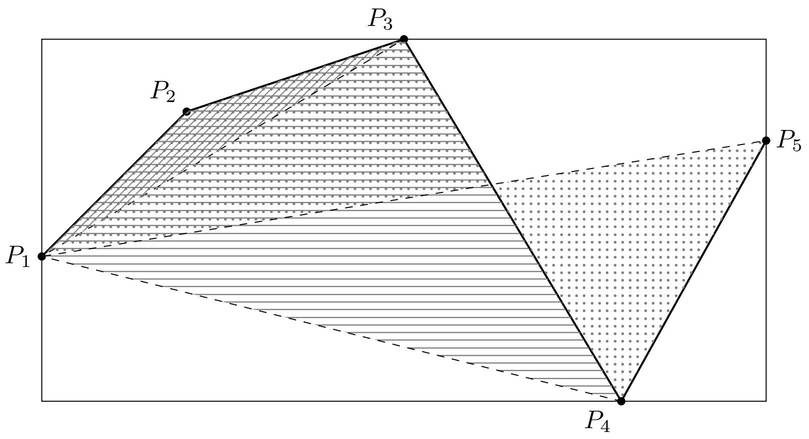


Figure 18: *Area considerations of local control polygon.*

According to this specification, we must interpret values close to 1 from above as belonging to local control polygons that are flat, and higher values as belonging to local control polygons in need of increased plotting effort. The cost is $\nu - \mu + 1$ evaluations of distance and 1 division. If we use square distances, we count 1 addition and 2 multiplications for each of the distances. The method makes sense only if each local control polygon has at least one inner control point. In fact, the cost is highest when $S \sim n/2$, i.e., when we typically have $\nu = \mu + 2$. In this situation, the cost is about 19 operations per local control polygon. Choosing full local control polygons, i.e., $\nu = \mu + p$ the cost is about $3p + 13$ operations per local control polygon.

3.2.4 Area considerations of the local control polygon

It is possible to contemplate other schemes. One possibility is to introduce a notion of area in the local control polygon and direct plotting effort to places with higher values. The notion of area would pick up on scaling, i.e., with two local control polygons having same shape but different sizes, the bigger one is favoured. However, with more than one inner control point, we could need to define how to distinguish between situations where an inner segment crosses the base line, giving in effect two smaller areas, and situations without such crossings.

Another option is to consider the area of the convex hull of the local control polygon, or as an approximation, using the rectangular box containing all points and performing checks on the inner points. See Figure 18 for a visualization of some of the options.

If we restrict the area considerations to local control polygons with only one inner point, that is to triangles, the ambiguous situations mentioned above will not arise, and we have already developed the tools needed for such tests: the distance to the base line and the length of the base line. It remains to determine an adequate method for combining these numbers. The simple multiplication of the two might be the better option, but weighing regimes, arithmetic, geometric and harmonic averaging regimes can also be considered, all trying to account for possible and probable situations. These ideas will for the sake of conciseness not be developed further in this paper.

3.3 Placement of knots relative to local control polygon

We will now find reasonable choices for exactly where the new knots should be inserted, once the local control polygon or the precise control point with assumed highest potential for visual quality improvement has been determined.

Studying knot placement is not new. In the context of using splines as interpolation objects. Carl de Boor [Boo73] found a criteria for good placement of knots when wanting to *interpolate* a function using a broken line. The idea is, as before, to place the knots more densely where curvature is high.

Theorem 3.16 (Placement of knots when interpolating with a broken line)**.** *If $g \in \mathcal{C}^{(2)}(a...b)$, that is, if g has two continuous derivatives inside the interval (a, b), and $|g''|$ is monotone near a and b, and $\int^b |g''(x)|^{\frac{1}{2}}\,\mathrm{d}x < \infty$, then, with $t_2 < \ldots < t_{n-1}$ chosen so that*

$$\int_a^{t_i} |g''(x)|^{\frac{1}{2}}\,\mathrm{d}x = \frac{i-1}{n-1}\int_a^b |g''(x)|^{\frac{1}{2}}\,\mathrm{d}x, \qquad \textit{all } i$$

we have quadratic convergence in n of the broken line interpolant at $\mathbf{t}$.

Our situation is similar: we want to approximate a curve by broken lines. Our lines do not interpolate at the knots, but Lemma 1.16 guarantees that the lines have end-points close to the interpolation sites.

To proceed further, it is necessary to discuss a few more of the theoretical properties of B-splines. This helps to clearly see the local relationship between knots and control points.

3.3.1 Control points affected by a single knot insertion

Recall that a knot average t^*_μ involves only the knots with indices $j = \mu+1, \ldots, \mu+p$. The knot averages with index less than $\mu - p + 1$ are thus not affected by the knot insertion, and similarly for the old knot averages corresponding to the new knot averages with index greater than μ. The remaining p new knot averages are all new.

Lemma 3.17. *Assume we inserted the knot $z \in [t_\mu, t_{\mu+1})$ in $\mathbf{t}$ to obtain the new knot vector $\bar{\mathbf{t}}$, and computed new coefficients $\bar{\mathbf{c}}$ from the old coefficients $\mathbf{c}$. Then $z = \bar{t}_{\mu+1}$ and*

$$\bar{t}^*_i = \begin{cases} t^*_i & \textit{for} \quad i = 1, \ldots, \mu - p, \\ t^*_i - \frac{1}{p}t_{\mu+1} + \frac{1}{p}\bar{t}_{\mu+1} & \textit{for} \quad i = \mu - p + 1, \ldots, \mu, \\ t^*_{i-1} & \textit{for} \quad i = \mu + 1, \ldots, n + 1. \end{cases} \tag{3.22}$$

There is another possible approach for obtaining these equations. For spline functions, the p new knot averages are also the first axis coordinates for control points, thus satisfy the relations in Böhm's algorithm (Lemma 1.19). More precisely, for these indices we may write

$$\bar{t}^*_i = \frac{\bar{t}_{\mu+1} - t_i}{t_{i+p} - t_i} t^*_i + \frac{t_{i+p} - \bar{t}_{\mu+1}}{t_{i+p} - t_i} t^*_{i-1}, \tag{3.23}$$

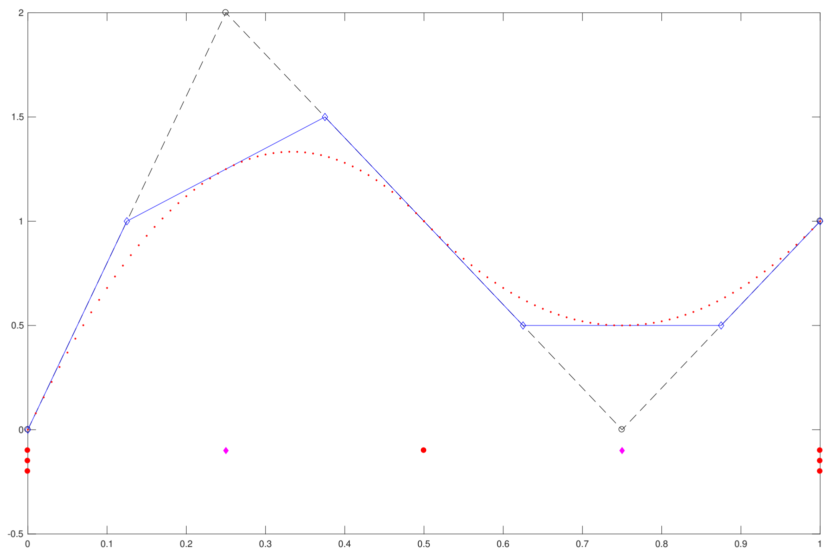


Figure 19: *Knot insertion consequence for control polygon: even degree (p=2).*

which after reordering gives Equation (3.22).

Böhm's algorithm actually provides more information about how a local control polygon is affected by a single knot insertion. We see from the formulas that indeed $p+1$ of the old control points are involved, and that the new coefficients are convex combinations of the old coefficients. More precisely, the new coefficients simply lie on each of the p line segments of the relevant old local control polygon. This means that $p-1$ old vertices vanish, and are replaced by p new vertices.

Figures 19 and 20 illustrate these mechanisms in the cases of even and odd degree. The original control polygon is in dashed black and the refined control polygon in blue. The value of original knots is indicated with red circles and the value of inserted knots with pink diamonds.

3.3.2 On the reduction in size of the B-spline coefficients.

In our context, the desired effect of knot insertion is a substantial change in B-spline coefficient sizes, since this means the new control polygon wraps around the spline even tighter than the old control polygon. This is related to the more specific problem of finding how much the size of B-spline coefficients can be reduced by inserting one new knot. The two following lemmas are due to [Tom93] in a discussion on *condition numbers*.

Lemma 3.18. *Let $f = \sum_{j=1}^{n} c_j B_{j,p,\mathbf{t}}$ be a spline and $\bar{c}$ the spline coefficients obtained by inserting one new knot into $\mathbf{t}$. Then the spline coefficients satisfy*

$$\frac{\max_i |c_i|}{\max_j |\bar{c}_j|} \leq \delta_p, \tag{3.24}$$

with $\delta_p \leq p$. A necessary condition for equality in Equation (3.24) is that the knot insertion must produce a Bernstein knot vector.

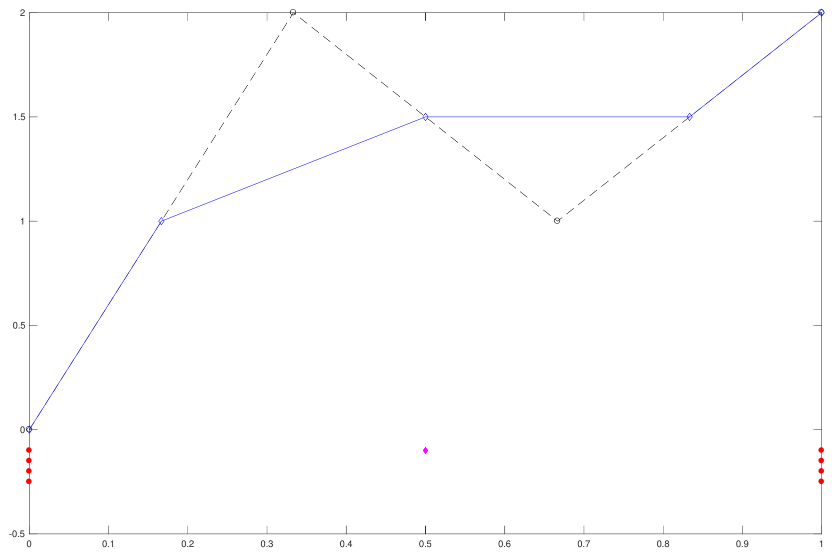


Figure 20: *Knot insertion consequence for control polygon: odd degree (p=3).*

Lemma 3.19. *Let* $f = \sum_{j=1}^{n} c_j B_{j,p,\mathbf{t}}$ *be a spline on the* uniform *knot vector* $\mathbf{t}$ *and* $\bar{c}$ *the spline coefficients obtained by inserting the knot with value* z *into* $\mathbf{t}$*. For obtaining the quantity*

$$\max_{\mathbf{c},z} \frac{\max_i |c_i|}{\max_j |\bar{c}_j|} \tag{3.25}$$

it is necessary that z *is inserted midway between two old knots in the case of* p *even, and that* z *increases the multiplicity of an existing knot in the case of* p *odd.*

Lemmas 3.18 and 3.19 describe two optimal or 'best case' scenarios. Although we certainly cannot hope to achieve such effects of our knot insertions on general knot vectors and for general coefficient values, they might give some hint of as to what should be set as default values.

3.3.3 Placing knots

The rule for placing a knot depends on the geometric criterion chosen from Section 3.2. In any case, we must find the value of the knot by studying its position relative to a control point, for which we know exact values (Equation (3.23)). The mechanisms and ideas mentioned below need further study, but are included here for completeness.

With the length method, we have no immediate natural option. We can of course insert a knot at the knot average of the beginning or end of the corresponding segment. It is an advantage that the value is already computed. We can also insert a knot midway between these values, an option which respects symmetry.

With the angle method and the distance to base line method, if we study individual points and not all inner points of a local control polygon, we can also insert a knot at the corresponding knot average, i.e., at the control point exhibiting most 'unflatness'.

A few of the methods find a local control polygon to refine, not specific control points, such as with the ratio method. One possibility is to insert a few knots, for instance p, uniformly inside that local control polygon.

Another idea involves imitating Chaikin's algorithm, where we insert a knot so that a new knot average appears midway between two old segments.

In an implementation, it may be necessary to check we do not raise the multiplicity of existing knots, for instance to avoid multiplicity $p + 2$. If a desired knot value is not valid, we can simply perturb the value slightly. We can also experiment with the opposite: inserting a same knot multiple times.

3.4 Selecting a stopping criterion

We now elaborate on reasonable stopping criteria. As discussed in Section 2.2, visual quality depends on the smoothness of the approximation and the proximity of the approximation. There are a few choices available for how to determine when visual quality is assumed to be satisfactory and thus to define a stopping criterion for the iterations in the adaptive plotting algorithm:

1. The proximity can be controlled based on the convergence bounds given in Section 1.2.2. A drawback is that we must have an idea of the norm of the second derivative of the spline, a computation that we would prefer to avoid.

2. The smoothness can be controlled by computing angles, lengths or other geometric aspects in the local control polygon, on the pattern of the method described in Section 2.3.3.

3. The proximity can also be controlled by introducing a threshold ϵ for the difference in some measure of the old and the new control polygon, i.e., we stop when inserting a new knot only alters the control polygon slightly. Such a slight alteration should be measured against the size of the whole spline, assumed to fill the screen. If for instance the new and old control points only differ in a distance of δ pixels, say $\delta = 5$, we may assume that the approximation in that area is good enough.

The second method is easy to implement once the methods from Section 3.2 are in place. On the other hand, it is not necessarily trivial to find correct threshold values. These must be acquired through experimental testing. For angles, a starting point for testing is a threshold value $\alpha_\tau = 1$ degree. For the distance to base line method, a conversion to pixel distance can be used (see below).

We choose the last method as the principle for our stopping method and will now develop it further. First, we find $\mathbf{\Delta c}$ the length of the diagonal of the rectangle containing the spline by performing max/min-tests on the control points. Noting then ϵ the control point movement threshold distance, δ the corresponding chosen limit for pixel distance and ρ the diagonal length of the screen measured in number of pixels, we have

$$\epsilon = \frac{\delta}{\rho}\mathbf{\Delta c}. \tag{3.26}$$

The movement of the control points can be deduced from Equation (1.30), and can be counted as low cost, since the weighing factors are already computed by the insertion algorithm.

From an algorithmic perspective, it makes sense not to perform such penalizing checks too often, but for instance only when we exceed 100 points and for every 10 iterations.

There should be an additional check in place, guaranteeing that the algorithm terminates, for instance by limiting the number of knots that can be inserted, say 300 knots.

3.5 Adaptive algorithm

3.5.1 A note on the term 'adaptive'

We have used the therm 'adaptive methods' on numerous occasions.

Definition 3.20 (Adaptive plotting of splines)**.** Methods which in each iteration analyze some information residing in the spline object and take well-considered decisions based on newly computed information from it. The term also assumes some generality, in that such methods should not be limited to special cases but apply to a wide family of functions, in our case all B-splines regardless of degree.

This means that the term is to be understood as different from 'static' or 'brute force' methods which plot objects without analyzing or interacting with them, i.e., methods which basically apply the same principle to every point or every segment of the object to be plotted.

3.5.2 General assumptions

As indicated in Chapter 1, we can assume all knot vectors to be $(p+1)$-regular knot vectors, lest some preprocessing. Similarly we impose the restriction that a knot has multiplicity at most $p-1$, since otherwise, we know that the spline and its control polygon coincide, and thus the refined control polygons to the left and right of a knot with multiplicity p could and would be drawn independently.

Similarly, we will assume that the interval to be plotted is the entire support of the spline. If one wanted to zoom to a specific part of the curve, the local nature of B-splines means we could discard all computations outside a (small) buffer area around the interval of interest. This issue would also typically be addressed in a cheap preprocessing stage.

3.5.3 General algorithmic formulation

The adaptive plotting strategies proposed below all share a common structure. On input is given $f = \sum_{j=1}^{n} c_j B_{j,p,\mathbf{t}}$, a spline in $\mathbb{S}_{p,\mathbf{t}}$. One iteration consists of three basic steps:

1. The first step consists of applying the criterion $\mathcal{C}$ for determining the section of the control polygon assumed to hold the greatest potential for improving visual quality, running from left to right on the knot average and

coefficient vector. This method returns the parameters $\mu \in \{1, 2, \ldots, n\}$ and $r \in \{1, 2, \ldots, p+1\}$, which together indicate that a knot should be inserted in $\left[t^*_\mu, t^*_{\mu+r}\right)$. The number r controls the locality of the method, i.e., lower values describe the most local methods and higher values describe more global methods.

2. The second step consists of the rule $\mathcal{R}$ for deciding exactly where in this interval the knot is to be inserted. This method returns the value of the new knot z.

3. The third step is a knot insertion algorithm $\mathcal{I}$, which inserts the new knot z into the knot vector and computes the new coefficients.

The iterations continue until the stopping criteria $\mathcal{S}$ is met.

The general description above can be coupled with an initial phase of uniform knot insertion, on the pattern of the standard plotting method used in Mathematica as described in Section 2.3.3. The number of knots inserted in such a stage can have a default value, but can also be found in order to guarantee a maximum segment length of the control polygon. The details of this are omitted.

Between iterations in the general algorithm, it is possible to shift from one partitioning of the control polygon Σ_1 to another Σ_2. This ensures more equal treatment of each control point in the long run. The same is true if we implement the different criteria $\mathcal{C}$ and the different rules $\mathcal{R}$ with similar syntax. This allows for even more dynamic schemes. Determining when to change between criteria and rules remains to be described.

3.6 On testing the newly developed methods

For av given algorithm, it is possible to judge its complexity by prescribing the number of arithmetic operations or by indicating its order, as we have attempted in the preceding sections. We saw that the criteria, rules and knot insertions were all operations linear in complexity. For lower degrees, we do not expect to compete with simple evaluation methods. For higher degrees, however, we do expect a gain.

We note that in practical situations these indications of magnitude do not always communicate accurately and truthfully the target metric in which we are really interested: actual time consumption. Actual time consumption depends heavily on the implementation of the algorithm in question, especially when arrays and lists make up a considerable part of the data. In working with the theme at hand, we have to the best of our efforts kept a consistent data structure and avoided evidently bad programming habits that could favour one algorithm over the other. We shall nonetheless humbly acknowledge the possibility of suboptimal routines.

To a large degree, and increasingly with modern computers, CPU-time also depends on the underlying algorithms which are called, software versions, hardware configurations and hardware optimizations that may or may not be in place. Such dependencies are especially valid for the elementary functions $\sqrt{\cdot}$ and arccos that we saw in Section 3.2.

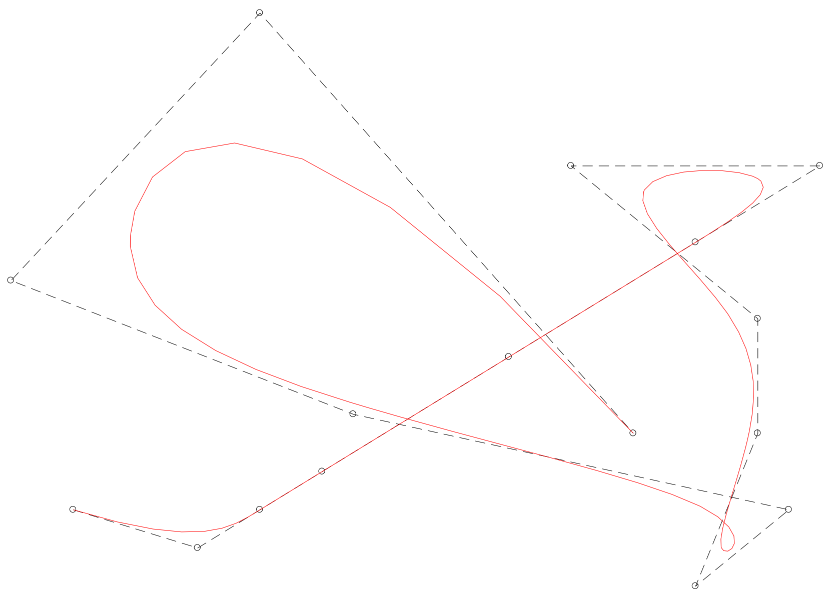


Figure 21: *Test curve [Eri93].*

With these uncertainties, it is not uncalled for to study the algorithms' real time consumption on a variety of problems and using at least more than one test-platform, hereby aspiring for a just and statistically robust comparison of these algorithms. Information about software versions and hardware configurations used in our preliminary testing regime can be found in Appendix A.

Future testing of the algorithms can use a palette of low (p=3) and high (p=25) degree splines. One appealing test curve for low degree is shown in Figure 21. It contains a few almost straight segments and areas with varying curvature. Similar curves can easily be generated for higher degrees. In a first instance, results should be evaluated using the control polygon approximation error given in Definition 2.5.

CHAPTER 4

Extensions and improvements

The aim of this chapter is to show the viability of the methods developed in the previous chapter for surfaces generated by tensor product splines, and to discuss modifications of the methods for efficient parallel computing. Sections 4.3 and 4.4 provide concluding remarks.

4.1 B-spline surfaces

4.1.1 Tensor products

A common way to produce surfaces is by a tensor product construction.

Definition 4.1. Let p, q be integers. Let $\mathbf{t} = \{t_i\}_{i=1}^{n+p+1}$ and $\mathbf{s} = \{s_j\}_{j=1}^{m+q+1}$ be two knot vectors. The tensor product of the two spaces $\mathbb{S}_{p,\mathbf{t}}$ and $\mathbb{S}_{q,\mathbf{s}}$, denoted $\mathbb{S}_{p,\mathbf{t}} \otimes \mathbb{S}_{q,\mathbf{s}}$, is the family of all functions f of the form

$$f(x,y) = \sum_{i=1}^{n} \sum_{j=1}^{m} c_{i,j} B_{i,p,\mathbf{t}}(x) B_{j,q,\mathbf{s}}(y), \tag{4.1}$$

where the coefficients $\{c_{i,j}\}_{i,j=1}^{n,m}$ are real numbers

For a tensor product function, the degree, dimension and smoothness may differ in x- and y-directions. The two directions are usually taken to be orthogonal.

As we saw the graphics device taking care of drawing lines between two points in the case of curves, it takes care of representing the surface by projecting the rectangle defined by four points. There are also other methods to represent surfaces realistically, for instance by drawing triangles and by using shadowing [Hug+14].

Equation (4.1) shows that the spline structure from the univariate case is kept. In particular, if we fix one of the variables, the expression reduces to a spline function. This gives a natural framework for applying the methods developed in Chapter 3.

4.1.2 Algorithm adjustments to the tensor product case

We first set $x = t_{i_1}$ and run the adaptive plotting algorithm for the spline function thus obtained. This leads to new knot vector $\mathbf{s}^1$, a refinement of $\mathbf{s} = \mathbf{s}^0$.

We can then continue working in the same direction by setting $x = t_{i_2}$ and applying the algorithm again, but now on the function

$$\tilde{f}(x, y) = \sum_{i=1}^{n} \sum_{j=1}^{m+1} \tilde{c}_{i,j} B_{i,p,\mathbf{t}}(x) B_{j,q,\mathbf{s}^1}(y). \tag{4.2}$$

It is also possible to change directions and set $y = s_{j_1}$ and proceed in a similar fashion.

The simpler option is nevertheless in a first phase successively to fix x such that x runs through all original knots in $\mathbf{t}$, and then in a second phase fix y such that it runs through all knots on the final refinement $\bar{\mathbf{s}}$ of $\mathbf{s}$ obtained from the first phase.

We do not know how many knots $\bar{\mathbf{s}}$ contains before the second phase starts, meaning it is not trivial to judge the complexity of the algorithm. But we know we will at least need to run the algorithm n times in the first direction and m times in the second direction. For reference and comparison, the MATLAB `fnplt`-function by default evaluates a spline function in a 51x51 point grid.

It may be tempting to seek criteria for the visual quality of a representation of a surface using a concept of *total curvature* rather than the approach of reducing the surface to two lines and checking the visual quality of the lines. Gaussian curvature is in that case not an option, since it involves the computation of two exact curvatures which we deem to expensive. One option is to modify the concept of *distance to base line* from the study of curves, to a concept of *distance to base polygon.* It remains to define such a concept precisely, or to find other approaches to measuring curvature. Undoubtedly, such schemes must be approximative.

4.1.3 Propagation of refinement

The knot vectors produce rectangular *knot patches.* If we refine a knot vector in one direction, the inserted knots will be used across values in the other direction. This means some regions of the surface may receive an excessive number of knots, i.e., more knots than required to represent that particular part of the surface adequately. See Figure 22.

This *propagation of refinement* is not a weakness inherent to B-splines, but rather a deficiency of the tensor product construction well known in the computer graphics research environment. One solution is appropriately combining basis functions at different refinement levels, thus modifying existing surfaces by locally adding patches representing finer detail. The study of such methods is outside the scope of this thesis, but the principle is well presented in for instance [FB88; JRK15; Vuo+11]. See Figure 23 for a visualization of the principle.

4.2 Modification of algorithms to parallel implementation

We succinctly address how the plotting algorithms may be modified to reap the potential of parallel computing.

The central concept in parallel computing is work division. The propensity of the problem to allow for work division governs the theoretical gain of using multiple parallell processes versus one serial process.

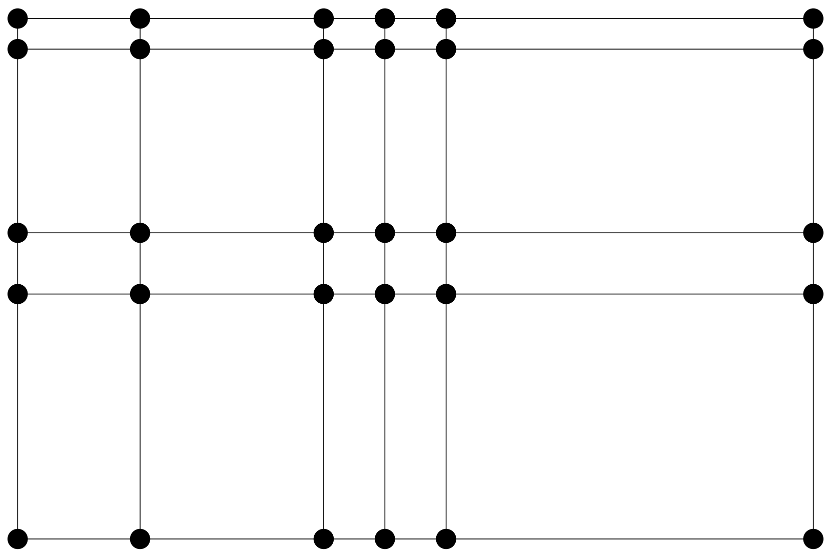
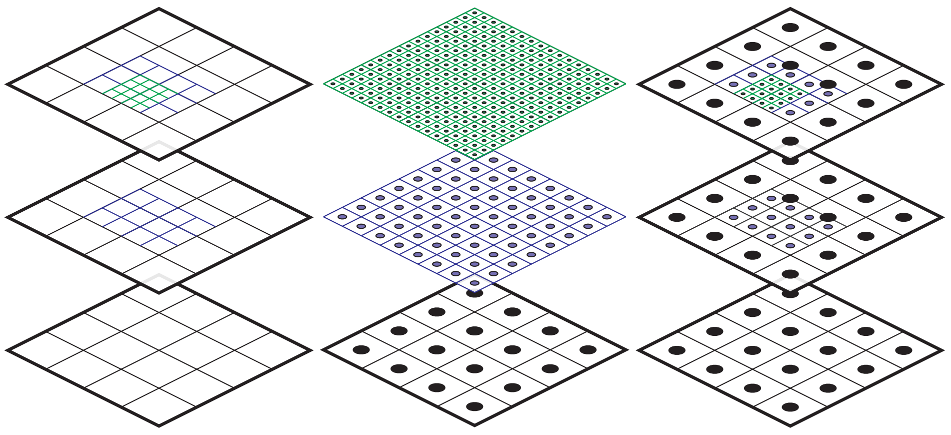


Figure 22: *Propagation of refinement. The knot lines for a tensor product spline surface.*



(a) *Meshes* ***(b)*** *Tensor product basis* ***(c)*** *Hierarchical basis*

Figure 23: *Principle of a hierarchical basis in the 2D case. The original mesh is refined in two steps (left). The tensor-product basis is constructed for the entire grid using the finest knot span available (middle). The Hierarchical basis consists of the coarser basis functions from the previous level, but those contained in the refined region are now substituted by the available finer basis functions. Figure reworked from [JRK15].*

For the surface case, work division can be carried out in the following fashion. Keeping the direction fixed, each process chooses a knot value and finds candidates for knots. When all processes have returned candidates, the union of them is inserted and the coefficients are updated. Several rounds may be necessary before work is completed in the first direction. Refinement in the second direction may be conducted similarly.

Zooming down to the univariate case, there is also some potential for work division. Given a partitioning Σ, each thread can compute values on independent local control polygons according to the given criteria, for instance the distance to base line-test. Values are returned to a common memory pool using a special sorting algorithm. If we use for instance *heap selection*, where we do not sort all values but keep an overview of say the 10% biggest values, there will always be knots to insert for an idling thread. The knot insertion process, either using Böhm's algorithm or the Oslo Algorithms, is also parallelizable. A separate thread keeps track of the stopping criterions.

4.3 Other ideas for algorithm improvements

4.3.1 Including arcs in approximation schemes

Note that, in accordance with Observation 2.3, it is possible to develop plotting methods which approximate curves using a combination of lines and ellipse arcs. More precisely, such a method would not cut corners, but round them off by drawing arcs. saving at least one iteration per corner of an adaptive algorithm. A reasonable assumption when dealing with the inherently smooth splines, is that a short, smooth arc is generally a better approximation than an angle of two straight segments.

4.3.2 Perception and psychological factors

As we have seen, plotting is not an accurate science, but a series of approximations. The goal is to create a truthful perception, read or viewed by a human eye. Perception is undoubtedly governed by habits, and the further development of plotting schemes could take advantage of such habits. This is especially true in a time-constrained environment, where the successive refinement of a curve could continue until the expiration of some time limit, say 10 milliseconds, ensuring a comfortable 100 frames per second on a computer screen.

More precisely, most people usually have a first look at the center of the screen, and the eye's center of attention has higher visual acuity than surrounding areas. This motivates prioritizing the middle areas of a screen in the adaptive algorithms, for instance by giving more weight to (prioritizing) its control points by means of a simple convolution function.

4.4 Final remarks

We stated in the Preface a hypothesis of the existence of plotting methods specific to splines that perform better than general plotting methods. The investigation of this problem first required a review of some B-spline theory in order to identify the key properties to exploit. A potential was confirmed in the gap between the first order complexity of knot insertion and the second order complexity of spline evaluation. It was then necessary to consult the literature on computer graphics to understand the mechanics behind plotting and to extract ideas behind the best available methods. This was the object of Chapters 1 and 2.

We pursued the idea of using the spline control polygon as an approximation to the spline. Failure of static corner-cutting methods to produce satisfactory results motivated a geometric analysis of the control polygon with focus on computational simplicity. This produced precise mathematical criteria for in which areas of the spline the plotting effort should be put. We devised schemes for local uniform plotting, but found more flexibility in adaptive methods based on inserting knots, and formulated rules for where knots should be inserted. It was possible to find several criteria for when the updated control polygon was of sufficient quality to properly represent the underlying spline. These elements were used to put together an algorithm for adaptive plotting of splines. A sketch of how these algorithms should be tested concluded Chapter 3.

The present chapter then showed how to apply the algorithm in the tensor product case and offered ideas to exploit in a parallel implementation of it. A few thoughts for additional improvements concluded the main part of the work.

It remains to carry out the testing regime of the algorithms and methods developed. In our initial experiments, a pre-stage uniform knot insertion preceding adaptive knot insertion at the knot averages based on the the distance-to-base line test yields results that are not unpromising, especially for higher degrees, as predicted by the complexity analysis. Future effort will remain directed at such experiments and at finding suitable combinations of the methods. It is a long-term goal to provide extensions to graphics libraries in order for these to treat general degree splines as geometric primitives.

APPENDIX A

Listings

This appendix contains a few code snippets referenced in the main text and implementation examples of some of the methods developed. Additional code and the code used to produce plots and figures in this paper can be found on GitHub[3].

Development and testing of algorithms was done using a MacBook Pro (2017) with Intel (dual) Core i5/2,3 GHz running OS X Version 10.14.6 and MATLAB 2016b.

A.1 Existing methods related to plotting

Java: Excerpt from class QuadCurve2D.java showing the principle of subdivision.

```
    /**
     * Subdivides the quadratic curve specified by the <code>src</code>
     * parameter and stores the resulting two subdivided curves into the
     * <code>left</code> and <code>right</code> curve parameters.
     * Either or both of the <code>left</code> and <code>right</code>
     * objects can be the same as the <code>src</code> object or
     * <code>null</code>.
     * @param src the quadratic curve to be subdivided
     * @param left the <code>QuadCurve2D</code> object for storing the
     *          left or first half of the subdivided curve
     * @param right the <code>QuadCurve2D</code> object for storing the
     *          right or second half of the subdivided curve
     * @since 1.2
     */
    public static void subdivide(QuadCurve2D src,
                                 QuadCurve2D left,
                                 QuadCurve2D right) {
        double x1 = src.getX1();
        double y1 = src.getY1();
        double ctrlx = src.getCtrlX();
        double ctrly = src.getCtrlY();
        double x2 = src.getX2();
        double y2 = src.getY2();
        double ctrlx1 = (x1 + ctrlx) / 2.0;
        double ctrly1 = (y1 + ctrly) / 2.0;
        double ctrlx2 = (x2 + ctrlx) / 2.0;
        double ctrly2 = (y2 + ctrly) / 2.0;
        ctrlx = (ctrlx1 + ctrlx2) / 2.0;
```

[3]https://github.com/sadahl/spline-aplot.git

```
        ctrly = (ctrly1 + ctrly2) / 2.0;
        if (left != null) {
            left.setCurve(x1, y1, ctrlx1, ctrly1, ctrlx, ctrly);
        }
        if (right != null) {
            right.setCurve(ctrlx, ctrly, ctrlx2, ctrly2, x2, y2);
        }
    }
```

Mathematica: Setting up a univariate cubic spline function in B-form. Showing the effect of altering parameters in Mathematica's adaptive plotting function. This code produced Figure 9.

```
p = 3;
knots = {0, 0, 0, 0, .25, .3, 1, 1, 1, 1};
controlpoints = {0, 3, .5, 1, 0, 1};
myspline[x_] :=
  Sum[controlpoints[[i + 1]] BSplineBasis[{p, knots}, i, x], {i, 0,
    5}];
Grid@Table[
  Plot[myspline[x], {x, 0, 1}, PlotPoints -> NN,
   MaxRecursion -> lambda,
   Method -> {"Refinement" -> {"ControlValue" -> .1}},
   PlotRange -> {{0, 1}, {0, 1.6}}], {lambda, {0, 1, 2}}, {NN, {5,
    10}}]
```

An implementation of Chaikin's algorithm.

```
function [c1n,c2n] = chaikin(c1,c2)
%Implements one round of Chaikins algorithm.
%   Takes as input
%   a 2D control polygon (c1,c2) (c1 may be the knot averages tstar)
%   c1 and c2 are assumed to be of same length, n.
%
%   Cuts the corners of the control polygon at one quarter of the length of
%   adjoining segments.
%   Returns the new control polygon, which better approximates a smooth
%   curve.

%allocating
n=length(c1);
c1n=zeros(1,2*n-2);
c2n=zeros(1,2*n-2);

%computing new values
for i = 1:n-1
    c1n(2*i-1)=(3*c1(i)+c1(i+1))/4;
    c2n(2*i-1)=(3*c2(i)+c2(i+1))/4;

    c1n(2*i)=(c1(i)+3*c1(i+1))/4;
    c2n(2*i)=(c2(i)+3*c2(i+1))/4;

end %end for

end
```

A.2 Spline methods in the MATLAB Curve Fitting Toolbox

Auxiliary function visualizing knots and inserted knots in the functional case.

```
function h = knotPlotter(t, c, newt)
%Plotting knots and their multiplicities. (functional case)
%Knots are shown as red circles slightly beneath the spline.
%t are the knots, shown in circles
%c are the coefficients of the spline
%newt, if provided, are the knots that were inserted, shown as diamonds

ymin=min(c); ymax=max(c); yamp=ymax-ymin;
ybaseline=ymin-yamp*.05; % underneath the lower part of cp
m = knt2mlt(t);
y=ybaseline-m*yamp*.025;
h = scatter(t, y, 'ro', 'filled', 'DisplayName', 'Knot placement');
hold on;

if nargin==3
    newt = sort(newt);
    newm = knt2mlt(newt);
    %shifting if some knots were already in t
    tshare=sort(intersect(newt,t));
    for i=1:length(tshare)
        element=tshare(i);
        nbinnewt=sum(~(newt-element));
        nbint=sum(~(t-element));
        myindex=find(newt==element,1);
        % first index in newm corresponding to tdiff(i)
        for j=myindex:myindex+nbinnewt-1
            newm(j)=newm(j)+nbint;
        end
    end

    newy=ybaseline-newm*yamp*.025;
    h = scatter(newt, newy, 'md', 'filled', 'DisplayName', 'Knot placement');
end %end if

end
```

Setting up a spline and inserting new knots. Visualizing the old and new control polygon and the knot vector.

```
%% Using the MATLAB Curve Fitting Toolbox - examples
% Studying the effect of one knot insertion (nonuniform)
% Simple example: quadratic B-spline, two non-zero coefficients
% Study the control polygon refinement in this simple case

% setting up the spline
p=2; %degree
knots=[0 .25 1]; % knots without multiplicities
coeffs=[0 2 0 1]; % spline coefficients
sp=spmak(augknt(knots,p+1),coeffs)% spmak uses order=degree+1
                                  % augknt ensures (p+1)-regular k.vector

% setting up window
scrsz = get(groot,'ScreenSize');
figure('Position',[1 scrsz(4)/1.3 scrsz(3)/1.3 scrsz(4)/1.3])

% initial sampling at the knot averages
xxstar=aveknt(sp.knots,p+1)
N=length(xxstar);
legcp='Control Polygon';
pc = plot(xxstar, sp.coefs,'--ok');%, 'DisplayName', legcp);
hold on;
```

```
%Choosing which knots to insert
xx=[.125 .75];

%Inserting knots using built-in function fnrfn
spr=fnrfn(sp,xx) % spr holds the new spline after insertion of new knots
xxstarnew=aveknt(spr.knots,spr.order)
N=length(xxstarnew);
% legcpr=sprintf('Refined control Polygon, N=%d',N);
% waitforbuttonpress;
% pause(.3);
pr = plot(xxstarnew, spr.coefs,'-db');
% set(pr,'XData',xxstarnew,'YData',spr.coefs);%,'DisplayName',mylegend);
% legend({legsp,legcp,mystr,mylegend},'Position', [.7 .7 .2 .2]);
% legend({legcp,mylegend},'Position', [.7 .7 .2 .2]);
legend('off'); drawnow;

knotPlotter(sp.knots, sp.coefs, xx);

fnpltpoints=fnplt(sp); %plotted with reference tool fnplt
% legsp='The spline';
ps=plot(fnpltpoints(1,:),fnpltpoints(2,:), 'r.'); hold off;
% saveas(gcf,'../figures/quadratic_insert','epsc');
```

A.3 Measuring properties in the local control polygons

A code snippet written in MATLAB based on the pseudo-code in the algorithm provided for computing the distance to base line.

```
function [ dSq ] = baselineDist(cs,ce,p)
%Returns the square of the distance d from a point to a line segment.
%   Uses dot products to compute distances.
%   If the projection is outside the segment between the points,
%   the method returns the distance to the closest point.
%   cs is the start point of the line segment
%   ce is the end point of the line segment
%   p is the specified point to which we measure distance.

%Moving the origin to cs
x2=ce(1)-cs(1);
y2=ce(2)-cs(2);
px=p(1)-cs(1);
py=p(2)-cs(2);

dotprod = px*x2 + py*y2;
% projlenSq;
if dotprod <= 0.0
    %the projection of p is outside the segment, on side of cs
    projlenSq=0;
else
    %moving origin to ce for similar check
    px=x2-px;
    py=y2-py;
    dotprod=px*x2 + py*y2;  % a dot b = -a dot -b
    if dotprod <= 0.0
        projlenSq = 0.0;
    else
        %p is between cs and ce
        projlenSq = dotprod * dotprod / (x2*x2 + y2*y2);
    end
end
```

```
dSq = max(0,px*px + py*py - projlenSq);

end
```

References

[App19] Apple Inc. *MTLPrimitiveType*. 2019. URL: https://developer.apple.com/documentation/metal/mtlprimitivetype (visited on 12/23/2019) (cit. on p. 20).

[BBB95] Bartels, R. et al. *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*. Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. Elsevier Science, 1995 (cit. on pp. 3, 32).

[Boe80] Boehm, W. “Inserting new knots into B-spline curves”. In: *Computer-Aided Design* vol. 12, no. 4 (1980), pp. 199–201 (cit. on p. 10).

[Boe85] Boehm, W. “On the efficiency of knot insertion algorithms”. In: *Computer Aided Geometric Design* vol. 2, no. 1 (1985), pp. 141–143 (cit. on p. 10).

[Boo73] Boor, C. de. “Good approximation by splines with variable knots”. In: *Spline functions and approximation theory*. Springer, 1973, pp. 57–72 (cit. on p. 35).

[Boo87] Boor, C. de. “Cutting corners always works”. In: *Computer Aided Geometric Design* vol. 4, no. 1 (1987), pp. 125–131 (cit. on p. 22).

[BP85] Böhm, W. and Prautzsch, H. “The insertion algorithm”. In: 1985 (cit. on p. 10).

[BZ92] Barry, P. J. and Zhu, R.-F. “Another knot insertion algorithm for B-spline curves”. In: *Computer Aided Geometric Design* vol. 9, no. 3 (1992), pp. 175–183 (cit. on p. 10).

[Cav+91] Cavaretta, A. et al. *Stationary Subdivision*. American Mathematical Society: Memoirs of the American Mathematical Society. American Mathematical Society, 1991 (cit. on p. 24).

[Cha74] Chaikin, G. M. “An algorithm for high-speed curve generation”. In: *Computer Graphics and Image Processing* vol. 3, no. 4 (1974), pp. 346–349 (cit. on p. 22).

[CLR80] Cohen, E. et al. “Discrete B-splines and subdivision techniques in computer-aided geometric design and computer graphics”. In: *Computer Graphics and Image Processing* vol. 14, no. 2 (1980), pp. 87–111 (cit. on p. 10).

[COX72] COX, M. G. “The Numerical Evaluation of B-Splines*”. In: *IMA Journal of Applied Mathematics* vol. 10, no. 2 (Oct. 1972), pp. 134–149 (cit. on p. 6).

[De 01] De Boor, C. *A practical guide to splines*. New York, 2001 (cit. on pp. 1, 10).

[Eri93] Eriksen, Ø. *Plotting av B-spline kurver og B-spline flater*. Oslo, 1993 (cit. on pp. 16, 28, 41).

[Far97] Farin, G. *Curves and surfaces for computer-aided geometric design: a practical guide*. San Diego, 1997 (cit. on pp. ii, 6, 22).

[FB88] Forsey, D. R. and Bartels, R. H. “Hierarchical B-spline refinement”. In: *Computer Graphics (ACM)* vol. 22, no. 4 (1988), pp. 205–212 (cit. on p. 43).

[Fol+90] Foley, J. D. et al. *Computer Graphics: Principles and Practice (2Nd Ed.)* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990 (cit. on p. 12).

[For72] Forrest, A. “Interactive interpolation and approximation by bézier polynomials”. In: *Computer Journal* vol. 15, no. 1 (1972), pp. 71–79 (cit. on p. 8).

[Fug93] Fugelli, P. *Skjøteinnsetting for B-splines : evaluering av nyutviklede og etablerte algoritmer*. Oslo, 1993 (cit. on p. 10).

[GW93] Goldman, R. and Warren, J. “An Extension of Chaiken’s Algorithm to B-Spline Curves with Knots in Geometric Progression”. In: *CVGIP: Graphical Models and Image Processing* vol. 55, no. 1 (1993), pp. 58–62 (cit. on p. 24).

[Hug+14] Hughes, J. et al. *Computer Graphics: Principles and Practice*. The systems programming series. Addison-Wesley, 2014 (cit. on pp. 12, 42).

[JRK15] Johannessen, K. A. et al. “On the similarities and differences between Classical Hierarchical, Truncated Hierarchical and LR B-splines”. In: (2015) (cit. on pp. 43, 44).

[KH95] Kobbelt, L. and Hartmut, P. “Approximating the length of a spline by its control polygon”. In: *Mathematical methods for Curves and Surfaces*. Vanderbilt University Press, 1995, pp. 291–292 (cit. on p. 27).

[Khr19a] Khronos Group. *Drawing: Primitive topologies*. 2019. URL: https://www.khronos.org/registry/vulkan/specs/1.0/html/chap19.html#drawing-primitive-topologies (visited on 12/23/2019) (cit. on p. 20).

[Khr19b] Khronos Group. *OpenGL: Primitive*. 2019. URL: https://www.khronos.org/opengl/wiki/Primitive (visited on 12/23/2019) (cit. on p. 20).

[LCM85] Lyche, T. et al. “Knot line refinement algorithms for tensor product B-spline surfaces”. In: *Computer Aided Geometric Design* vol. 2, no. 1 (1985), pp. 133–139 (cit. on p. 10).

[LM08] Lyche, T. and Morken, K. “Spline methods draft”. In: *Department of Informatics, Center of Mathematics for Applications, University of Oslo, Oslo* (2008) (cit. on p. 8).

[LM86] Lyche, T. and Mørken, K. “Making the Oslo algorithm more efficient”. In: *SIAM Journal on Numerical Analysis* vol. 23, no. 3 (1986), pp. 663–675 (cit. on p. 10).

[LR80] Lane, J. M. and Riesenfeld, R. F. “A Theoretical Development for the Computer Generation and Display of Piecewise Polynomial Surfaces”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* vol. PAMI-2, no. 1 (1980), pp. 35–46 (cit. on p. 25).

[Lyc89] Lyche, T. *Discrete B-splines and conversion problems.* Oslo, 1989 (cit. on p. 10).

[Mar19] Martin, F. *Instruction tables: List of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs.* 2019. URL: https://www.agner.org/optimize/instruction_tables.pdf (visited on 12/23/2019) (cit. on p. 30).

[Mar90] Markstein, P. W. “Computation of elementary functions on the IBM RISC System/6000 processor. (technical)”. In: *IBM Journal of Research and Development* vol. 34, no. 1 (1990), p. 111 (cit. on p. 30).

[Mat16] Mathworks. *fnplt, Plotting of a function f on its basic interval.* 2016. URL: https://se.mathworks.com/help/curvefit/fnplt.html (visited on 12/23/2019) (cit. on p. 18).

[Mat19] Mathworks. *MATLAB.* Version R2016b (9.1.0.441655). Dec. 19, 2019 (cit. on p. 10).

[Mic19a] Microsoft Inc. *GDI+: Overview of vector graphics.* 2019. URL: https://docs.microsoft.com/en-us/windows/win32/gdiplus/-gdiplus-overview-of-vector-graphics-about (visited on 12/23/2019) (cit. on p. 20).

[Mic19b] Microsoft Inc. *Programming Guide: Primitive Topologies.* 2019. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d11/d3d10-graphics-programming-guide-primitive-topologies (visited on 12/23/2019) (cit. on p. 20).

[Ora19] Oracle. *The Java Tutorials: Geometric Primitives.* 2019. URL: https://docs.oracle.com/javase/tutorial/2d/overview/primitives.html (visited on 12/23/2019) (cit. on p. 20).

[Rap91] Rappoport, A. “Rendering Curves and Surfaces with Hybrid Subdivision and Forward Differencing”. In: *ACM Trans. Graph.* Vol. 10, no. 4 (Oct. 1991), pp. 323–341 (cit. on p. 28).

[Rie75] Riesenfeld, R. “On Chaikin’s algorithm”. In: *Computer Graphics and Image Processing* vol. 4, no. 3 (1975), pp. 304–310 (cit. on p. 24).

[Sch07] Schumaker, L. *Spline Functions: Basic Theory.* Cambridge Mathematical Library. Cambridge University Press, 2007 (cit. on pp. 3, 6).

[Shr+13] Shreiner, D. et al. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3, Eighth Edition.* 8th ed. Addison-Wesley Professional, 2013 (cit. on pp. 12, 20).

[SSF94] Sankar, P. et al. "Curve and Surface Generation and Refinement Based on a High Speed Derivative Algorithm". In: *CVGIP: Graphical Models and Image Processing* vol. 56, no. 1 (1994), pp. 94–101 (cit. on p. 22).

[Str99] Strebel, R. *Pieces of software for the Coulombic m body problem.* 1999 (cit. on p. 30).

[Tom93] Tom Lyche, K. M. *Knot insertion and deletion algorithms for B-spline curves and surfaces.* Place of publication not identified, 1993 (cit. on p. 36).

[Vuo+11] Vuong, A.-V. et al. "A hierarchical approach to adaptive local refinement in isogeometric analysis". In: *Computer Methods in Applied Mechanics and Engineering* vol. 200, no. 49-52 (2011), pp. 3554–3567 (cit. on p. 43).

[Wol19a] Wolfram Research. *Wolfram Documentation: Plot Points.* 2019. URL: https://reference.wolfram.com/language/ref/PlotPoints.html (visited on 12/23/2019) (cit. on p. 20).

[Wol19b] Wolfram Support Center. *Private communications.* Sept. 18, 2019 (cit. on p. 20).