Bezier and B-spline curves with knots in the complex plane

Article *in* Fractals · November 2011

DOI: 10.1142/S0218348X11005221

Bezier and B-spline Curves with Knots in the Complex Plane

KONSTANTINOS I. TSIANOS and RON GOLDMAN

Department of Computer Science

6100 Main Street, Houston, TX, 77005, USA

konstantinos.tsianos@gmail.com, rng@rice.edu

Abstract: We extend some well known algorithms for planar Bezier and B-spline curves, including the de Casteljau subdivision algorithm for Bezier curves and several standard knot insertion procedures (Boehm's algorithm, the Oslo algorithm, and Schaefer's algorithm) for B-splines, from the real numbers to the complex domain. We then show how to apply these polynomial and piecewise polynomial algorithms in a complex variable to generate many well known fractal shapes such as the Sierpinski gasket, the Koch curve, and the C-curve. Thus these fractals also have Bezier and B-spline representations, albeit in the complex domain. These representations allow us to change the shape of a fractal in a natural manner by adjusting their complex Bezier and B-spline control points. We also construct natural parameterizations for these fractal shapes from their Bezier and B-spline representations.

Keywords: Bezier Curves, B-Spline Curves, Fractals,Knot Insertion, Fractal, Subdivision

1 Introduction

Parametric polynomial (e.g., Bezier) and piecewise polynomial (e.g., B-spline) curves have quite simple properties. Polynomials and splines are one-dimensional curves; polynomials are infinitely differentiable everywhere and splines are infinitely differentiable almost everywhere. Therefore these curves can have only a finite number of inflections, cusps and self-intersection points (see Fig. 1). Thus polynomials and splines are not self-similar curves; they do not consist of several smaller copies of themselves each containing a like number of inflections, cusps and self-intersections. Moreover, polynomial and spline curves come with simple parameterizations which permit us to generate points on the curve by evaluating at the parameters. Finally, when written in Bezier or B-spline form, polynomials and splines have control points which allow us to change the shape of the curve in a natural manner by adjusting the location of the control points.

Fractal shapes appear to be very different from polynomial and piecewise polynomials curves (see Fig. 2). Typically, fractals are self-similar curves, made up out of several scaled down copies of themselves. Fractals can be continuous everywhere, yet differentiable nowhere. Fractal curves can also have non integral dimension i.e., fractional dimensions between one and two. Finally, fractals are often attractors, fixed points of iterated function systems.

Nevertheless, despite these apparent differences, the purpose of this paper is to show that many classical fractals such as the Sierpinski gasket, the Koch curve, and the C-curve (see Fig. 2) can be generated from the same fundamental algorithms that are commonly used to render polynomial and piecewise polynomial curves. In particular, we shall show how to apply the de Casteljau subdivision algorithm for Bezier curves as well as several standard knot insertion procedures (Boehm's algorithm, the Oslo algorithm, and Schaefer's algorithm) for B-splines to build fractal shapes. The trick is to extend these algorithms from the real numbers to the complex domain. This extension reveals the deep connections between fractals and splines.

This paper is organized in the following fashion. Section 2 surveys the related literature. In Sec. 3 we first give a brief introduction to Bezier curves and to standard algorithms for generating fractal shapes. We then extend the formulas and algorithms for Bezier curves from the real numbers to the complex domain, and we show how to use these extensions to generate fractal shapes using Bezier techniques. Section 4 contains the corresponding theory and extension for B-spline curves. The paper concludes in Sec. 5 with a brief summary of our work and some ideas for future research.

2 Related Literature

Connections between fractals and Bezier curves are studied in [8], where it is shown that every Bezier curve is an attractor. The author also shows how to construct for each Bezier curve an iterated function system (IFS) that when iterated on any compact set converges to the given Bezier curve. [17] generalizes Goldman's results by showing that curves and surfaces generated by many different subdivision schemes can also be built using iterated function systems. These papers are complementary to ours, since they concentrate on showing how to use fractal algorithms (i.e., iterated function systems) to generate Bezier and B-spline curves and surfaces. Here we study the converse direction, showing that many classical fractals can be generated from standard algorithms for polynomial and spline curves. We also show that fractals inherit several properties of Bezier and B-spline curves, such as control points and parameterizations.

Control points for fractals are also introduced in [17]. However adjusting these control points induces transformations that are affine (lines are mapped to lines) but not conformal (angles are not preserved). In contrast, adjusting the Bezier or B-spline control points introduced here for fractals, induces transformations that are conformal but not affine. Thus adjusting these new Bezier or B-spline control points of a fractal can transform the fractal in a non-linear manner while still maintaining the angles between adjacent segments of the fractal.

[13, 12] mimic subdivision and knot insertion algorithms by replacing traditional subdivision and knot insertion procedures with two generic matrix operators where the rows of the matrices sum to one – that is, by taking arbitrary affine combinations of the control points. The curves generated

this way are typically fractals, but their emphasis is on determining when their operators generate smooth curves. Moreover, their fractals have neither control points nor parameterizations. In contrast, our emphasis is on fractals. We show how to extend the standard subdivision and knot insertion algorithms using the expressive power of complex numbers to generate classical fractals with intuitive control points and natural parameterizations.

Finally, the term *complex B-splines* appears in the literature on statistical signal processing (see e.g. [6]), which treats complex exponents and is related to wavelets. These complex B-splines, however, are completely unrelated to the work presented in this paper.

3 Bezier Curves

3.1 Real Bezier Curves Generated by Subdivision

A planar Bezier curve is a parametric polynomial curve $B : I \subset \mathbb{R} \to \mathbb{R}^2$, where $B(t) = (x(t), y(t))$ and $x(t), y(t)$ are univariate polynomials in t. One common way to define Bezier curves is through a sequence of control points $P_0, \ldots, P_n$ which give rise to a degree n Bezier curve with a compact representation:

$$B(t) = \sum_{k=0}^{n} b_k^n(t) P_k, 0 \leq t \leq 1 \tag{1}$$

where $b_k^n(t) = \binom{n}{k} t^k (1-t)^{n-k}$ are the *Bernstein basis functions* [5]. If we connect control points with adjacent indices by straight lines, we get the *Bezier control polygon.* Bezier curves mimic the shape of their control polygons, so we can adjust the shape of a Bezier curve in a natural manner by changing the location of the control points (Fig. 3).

To evaluate a Bezier curve at any parameter t, we can also use the de Casteljau algorithm which is based on repeated linear interpolation and has a simple graphical interpretation (Fig. 4). Start by placing the control points in the nodes at the base of a triangle. The left child multiplied by $(1-t)$ is added to the right child multiplied by t to produce a new intermediate node. The value at the apex is the point on the Bezier curve at the parameter t.

Subdivision [5] is the key to the analysis of Bezier curves. Using subdivision, we can split a Bezier curve into two segments at any parameter r with $0 < r < 1$. These two segments are described by two sets of control points so that the concatenation of the two segments yields the original curve. Fortuitously, we can read the new control points for each segment off the de Casteljau pyramid as shown in Fig. 5. When subdivision is applied recursively, the control polygons converge to the Bezier curve defined by the original control points. Thus a Bezier curve can be viewed as a fixed point of this subdivision procedure. In practice, subdivision is often used for fast rendering and intersection computations [9].

3.2 Fractals as Fixed Points of Iterated Function Systems

Some classical fractals that we will encounter again later on can be seen in Fig. 2. Although these fractals may seem hard to render, in practice we can obtain these fractals together with many other famous fractals as fixed points of iterated function systems [3]. Let w be a function that maps points in the plane to points in the plane. Then we can extend w to a map on sets of points S by setting $w(S) = \{w(x) | x \in S\}$. An iterated function system is a collection of *contractive maps*[1] $W = \{w_1, \ldots, w_l\}$. We apply an iterated function system W to a set S by taking $W(S) = w_1(S) \cup \cdots \cup w_l(S)$. Fractals can be generated by iterating an iterated function system W, starting on any compact set S. That is, given an initial set $A_0 = B$, we can apply W *recursively* to create a sequence of sets $A_{n+1} = W(A_n)$. In the limit, A_n will converge to a fractal A, independent of the choice of A_0. For example, to generate the Sierpinski triangle, we can start with the outer triangle as the base case, and apply recursively an iterated function system consisting of three transformations, each scaling by 0.5 about one of the three vertices of the outer triangle (see Fig. 6).

[1] Given a metric space M equipped with a distance metric d, a function $f : M \to M$ is a contractive map if there exists a real number $0 < k < 1$ such that $\forall x, y \in M$, $d(f(x), f(y)) \leq kd(x, y)$.

3.3 Fractals from Recursive Turtle Programs

There is another simple yet powerful tool for drawing a large class of fractals: *turtle graphics* [1, 9]. A virtual turtle is located at some point $P = (x, y)$, heading in the direction of a vector $w = (u, v)$. There are four basic turtle commands: *Forward* D (move forward D steps of size $|w|$ in the direction w while drawing the traversed path), *Move* D (move forward D steps of size $|w|$ without drawing i.e., translate), *Turn* A (rotate w by A degrees), *Resize* S (scale the step size by a factor S i.e., scaling $|w|$ by S). Using these commands, we can write recursive turtle programs that generate fractal shapes. Algorithm 1 shows how to generate the Sierpinski triangle of Fig. 2 with a recursive turtle program. Note that since fractals are attractors, the base case (level 0) can be any curve (here the base case is a triangle). As long as the turtle returns to its initial state after drawing the base case, in the limit as the number of levels approaches infinity, this turtle program will generate the Sierpinski triangle regardless of exactly what shape is drawn as the base case. The turtle commands *Forward* (Move), *Turn* and *Resize*, correspond to the conformal (angle preserving) transformations translation, rotation and uniform scaling. Any fractal generated by a conformal iterated function system (an iterated function system constituted only of conformal transformations – that is, composites of translations, rotations and uniform scalings) can be reproduced by a recursive turtle program, and there is a straightforward algorithm for converting any conformal iterated function system into a recursive turtle program [10].

Algorithm 1 Turtle Sierpinski Triangle

```
procedure DrawSierp($level$)
    if $level = 0$ then
        for $(rep = 0; rep < 3; rep++)$ do
            Forward 1
            Turn $\frac{2\pi}{3}$
        end for
    else
        for $(rep = 0; rep < 3; rep++)$ do
            Resize $\frac{1}{2}$
            DrawSierp($level - 1$)
            Resize 2
            Move 1
            Turn $\frac{2\pi}{3}$
        end for
    end if
end procedure
```

3.4 Complex Bezier Curves

From the discussion so far, although there are many fundamental differences, there is also a subtle connection between fractals and Bezier curves: fractals are fixed points of an iterated function system; Bezier curves are fixed points of a subdivision procedure. In this section we investigate this connection further and try to answer three basic questions:

- Can fractals be understood as some exotic forms of Bezier curves?
- Can we generate fractals with controls points?
- Can we parameterize fractals?

We shall start by extending the definition of a Bezier curve to the domain of complex numbers. This extension is motivated by the observation that every polynomial algorithm or identity for real numbers is also valid for complex numbers. We will look into the complex representation of Bezier curves using Bernstein basis functions and we will also discuss the properties of complex subdivision. By the end of this section it will become clear how these two approaches are related.

We define a complex Bezier curve as a function $B : S \subset \mathbb{C} \to \mathbb{C}$. To describe the curve, we are given controls points $w_0, \ldots, w_n$ that are represented as complex numbers i.e., $w_k = x_k + iy_k$. Using the Bernstein basis functions on the complex domain, we write:

$$B(z) = \sum_{k=0}^{n} \binom{n}{k} z^k (1-z)^{n-k} w_k, \quad z \in S, w_0, \ldots, w_n \in \mathbb{C} \tag{2}$$

Alternatively, we can try to generate complex Bezier curves using a complex parameter in the de Casteljau subdivision algorithm. In the case of real numbers, each step of subdivision amounts to a scaling operation, since at each step we compute $Q = (1-r)P_0 + rP_1 = P_0 + r(P_1 - P_0)$ as can be seen in Fig. 7 (left).

When, however, the subdivision parameter is allowed to be complex, we have more freedom, since we are now allowed both to scale and to rotate. If we subdivide at $re^{i\theta}$, then at one step of the de Casteljau algorithm we compute $w = (1 - re^{i\theta})w_0 + re^{i\theta}w_1 = w_0 + re^{i\theta}(w_1 - w_0)$. Multiplication by r causes a scaling, while multiplication by $e^{i\theta}$ induces a rotation. Alternatively, in cartesian coordinates, if the subdivision parameter is $x + iy$, we get $w = w_0 + x(w_1 - w_0) + iy(w_1 - w_0)$. Thus, the vector $w_1 - w_0$ scaled by x is added to the vector $w_1 - w_0$ rotated 90 degrees and then scaled by y. Hence subdividing at a complex number forces the intermediate point to leave the real line as shown in Fig. 7 (right).

To exhibit the effect of complex subdivision, let's start with the simplest Bezier curve, the straight line joining $P_0 = (0, 0)$ and $P_1 = (1, 0)$. If we apply de Casteljau subdivision at $r = 0.5 + i0.5$, we get two line segments that no longer lie on the real line. If we continue subdividing at $r = 0.5 + i0.5$, then as we see in Fig. 8 we generate the C curve!

We do not need to subdivide only at one point for each iteration of subdivision. Let us define two operators, $L(cp, r)$ and $R(cp, r)$. If we start with a control polygon cp and run the de Casteljau algorithm to subdivide at r, then L extracts the control polygon on the left side of the de Casteljau pyramid and R extracts the control polygon on the right side of the de Casteljau pyramid. Using these operators, we can perform multiple subdivision steps at each iteration. Another way of thinking about this approach is that instead of binary subdivision, we subdivide a curve into multiple segments in each iteration of the algorithm. For example, let‘s start with the same degree 1 Bezier curve as before with $cp = [P_0, P_1]$. Define one iteration of subdivision by

$$\begin{aligned} cp_{new} = [&L(cp, r_1)\cup \\ &L(L(R(cp, r_1), r_3), r_4]\cup \\ &L(R(L(R(cp, r_1), r_3), r_4), r_5)\cup \\ &R(cp, r_2)] \end{aligned}$$

where $r_1 = \frac{1}{3}, r_2 = \frac{2}{3}, r_3 = \cos\frac{\pi}{3} + i\sin\frac{\pi}{3}, r_4 = \frac{1}{2}, r_5 = \cos -\frac{2\pi}{3} + i\sin -\frac{2\pi}{3}$. Notice that the different r’s are selected so that the outcome of one iteration of subdivision generates a triangular pulse out of the straight line segment (see Fig. 9). After several iterations, we produce another famous fractal, the Koch curve.

The results shown so far are not a coincidence. Theorem 3.2 establishes a strong relationship between turtle fractals and Bezier subdivision. To establish this result, we first prove the following lemma:

Lemma 3.1. *The conformal transformations – translation, rotation, and uniform scaling – can be simulated with subdivision operations.*

Proof. Consider a control polygon $cp = [P_0, P_1]$. We show how to simulate the conformal transformations translation, rotation, and uniform scaling through subdivision.

- **(Uniform) Scaling:** We can scale by a using $L(cp, a)$
- **Rotation:** We can rotate by θ about P_0 using $L(cp, e^{i\theta})$
- **Translation:** We can achieve translation in two steps. Suppose that we need to to translate by d. First we scale by $d+1$ and keep the left segment. Then we scale by $\frac{d}{d+1}$ keeping the right segment. The total transformation is $R(L(cp, d+1), \frac{d}{d+1})$. To better understand these steps, see Fig. 10 for a graphical interpretation.

$\square$

We can use the proof of Lemma 3.1 to describe an algorithm that translates recursive turtle programs into subdivision procedures. A typical recursive turtle program will look like Algorithm 2. We can assume without loss of generality that both the base case and the recursive calls do not change the state of the turtle [9]. The base case ($level = 0$) amounts to drawing any basic shape, as long as the turtle returns to its initial state. For the main body, we have a sequence of turtle commands, whose only effect is to change the state of the turtle through conformal transformations, followed by recursive calls. The subdivision procedure takes as input an initial control polygon $cp = [0, 1]$ and computes a new control polygon cp_{new}. Initially we set

$cp_{new} = \emptyset$ and we introduce a helper variable $U = cp$. Then we use Lemma 3.1 to write a subdivision procedure of the form $U = \ldots L(R(U, r_1), r_2)\ldots$ that simulates all the state changing transformations up to the next recursive call. A recursive call ($TurtleFractal(level - 1)$) signifies a part of the fractal that needs to be saved. We simulate this part of the recursive turtle program by concatenating the current portion of the fractal (here U) with cp_{new}, i.e. we set $cp_{new} = cp_{new} \cup U$. We continue computing a new U from the old U and concatenating with the current value of cp_{new} until the last recursive call.

Algorithm 2 Abstract Turtle Program

```
1: procedure TurtleFractal($level$)
2:     if $level = 0$ then
3:         Draw base case
4:     else
5:         Change turtle state
6:         TurtleFractal($level - 1$)
7:         Change turtle state
8:         TurtleFractal($level - 1$)
9:         ...
10:    end if
11: end procedure
```

For an illustration, consider Algorithm 3 which shows one step of the recursive procedure that generates the Sierpinski triangle using subdivision. It is instructive to examine Algorithm 3 alongside Algorithm 1. Lines 1 and 2 initialize cp_{new} to the empty set and set U to the base case, which we always take to be the interval $[0, 1]$. Lines $4, 5$, and 6 simulate the loop in Algorithm 1. Line 4 scales by $\frac{1}{2}$. Line 5 concatenates the result of scaling to the new control polygon, since the initial scaling in Algorithm 1 is followed by a recursive call. In lines $6, 7, 8$ we store in the intermediate variable U the combined result of scaling by 2, translating by 1 and rotating by $\frac{2\pi}{3}$. Here we employ Lemma 3.1 to translate line for line the turtle commands in Algorithm 1.

Algorithm 3 Subdivision Sierpinski Triangle

```
1: $cp_{new} = \emptyset$
2: $U = [0, 1]$
3: **for** $(rep = 0; rep < 3; rep++)$ **do**
4:     $U = L(U, \frac{1}{2})$
5:     $cp_{new} = cp_{new} \cup U$
6:     $U = L(U, 2)$
7:     $U = R(L(U, 2), \frac{1}{2})$
8:     $U = L(U, e^{i\frac{2\pi}{3}})$
9: **end for**
```

Theorem 3.2. *Every fractal in the plane generated by a recursive turtle pro-*

gram can be generated by Bezier subdivision in the complex plane. Moreover, Bezier curves of degree 1 suffice.

Proof. We have just described an algorithm for simulating recursive turtle programs through recursive subdivision using Lemma 3.1 to infer the complex subdivision parameters required to simulate the conformal transformations in the recursive body of the turtle program. □

Corollary 3.3. *Every fractal in the plane generated by a conformal iterated function system can be reproduced by Bezier subdivision in the complex plane. Moreover, Bezier curves of degree* 1 *suffice.*

Proof. It is proved in [10] that any fractal in the plane generated by a conformal iterated function system can be reproduced by a recursive turtle program. Moreover, Theorem 3.2 shows that any recursive turtle program can be simulated via Bezier subdivision in the complex plane. □

Observe that Theorem 3.2 and Corollary 3.3 do not imply equivalence between conformal iterated function systems and Bezier subdivision. In fact, if we start with Bezier curves of degree greater than 1, then we can manipulate the control points to introduce non-linear transformations that are not possible with a conformal iterated function system. Thus, Bezier subdivision is richer and more expressive than iterated functions systems represented by conformal matrices. In Figs. 11, 12 we create a distorted Sierpinski triangle and a distorted Koch curve using (quadratic) Bezier subdivision by moving the interior control point off the real line. It is hard to come up with simple non-linear transformations that reproduce the same effects.

Next we establish at what values we can subdivide and get the control polygons to converge.

Lemma 3.4. *The lengths of the control polygons generated by recursive subdivision at ρ for a Bezier curve of degree n converge to zero whenever:*

- $\rho = r \in \mathbb{R}$ *and* $0 < r < 1$
- $\rho = re^{i\theta} \in \mathbb{C}$ *and* $0 < (r+q)^{n-1} * \max(r, q) < 1$

where $q = |1-\rho|$. *In particular, Bezier curves of degree one converge whenever* $0 < \max(r, q) < 1 \Leftrightarrow 0 < r < \min(1, 2\cos(\theta))$.

Proof. We prove the complex case, since the real case is well-known and follows directly from the complex case. Consider a degree n Bezier curve with complex control points $w_0, \ldots, w_n$. Consulting Fig. 13 let $u_0, \ldots, u_n$ and $v_0, \ldots, v_n$ be the left and right complex control points generated by de Casteljau subdivision at $\rho = re^{i\theta}$. From the de Casteljau algorithm we have:

$$u_k = \sum_{j=0}^{k} B_j^k(\rho) w_j \quad \textit{and} \quad u_{k+1} = \sum_{j=0}^{k+1} B_j^{k+1}(\rho) w_j \tag{3}$$

We will bound the quantities $|u_{k+1} - u_k|$. First we degree elevate u_k using a well known identity [7]:

$$u_k = \sum_{j=0}^{k+1} B_j^{k+1}(\rho) \left(\frac{j}{k+1} w_{j-1} + \frac{k+1-j}{k+1} w_j \right) \tag{4}$$

Using (4) and the right hand side of (3), we have

$$\begin{aligned} u_{k+1} - u_k &= \sum_{j=0}^{k+1} B_j^{k+1}(\rho)\left(\frac{j}{k+1}(w_j - w_{j-1})\right) \\ &= \sum_{j=0}^{k+1} \frac{j}{k+1}\binom{k+1}{j}\rho^j(1-\rho)^{k+1-j}(w_j - w_{j-1}) \qquad (5) \\ &= \rho\sum_{j=0}^{k+1}\binom{k}{j-1}\rho^{j-1}(1-\rho)^{k+1-j}(w_j - w_{j-1}) \end{aligned}$$

Now let $d = \max_i |w_{i+1} - w_i|$ and $q = |1-\rho| = \sqrt{1 + r^2 - 2r\cos(\theta)}$. Then

$$\begin{aligned} |u_{k+1} - u_k| &= \left|\rho\sum_{j=0}^{k+1}\binom{k}{j-1}\rho^{j-1}(1-\rho)^{k+1-j}(w_j - w_{j-1})\right| \\ &\leq |\rho|\sum_{j=0}^{k+1}\binom{k}{j-1}|\rho|^{j-1}|1-\rho|^{k+1-j}|w_j - w_{j-1}| \qquad (6) \\ &\leq r(r+q)^k d \end{aligned}$$

By a similar computation we get $|v_{k+1} - v_k| \leq q(r+q)^k d$. Therefore if $\max(r,q) * (r+q)^{n-1} < 1$, then for all k, $|u_{k+1} - u_k| < d$ and $|v_{k+1} - v_k| < d$. Hence, after m iterations of subdivision, the distance between any two consecutive control points will be less than $f^m d$ where $0 < f = (r+q)^{n-1} * \max(r,q) < 1$. Therefore as m goes to infinity, the distance between the consecutive control points of each control polygon goes to zero and thus the length of each control polygon generated by recursive subdivision converges to zero. $\square$

Observations:

- In the real case, $r + q = 1$ and $\theta = 0$, so the conditions reduce to $\max(1-r, r) < 1 \Leftrightarrow 0 < r < 1$.
- In the complex case, for degree $n = 1$, the condition $0 < (r+q)^{n-1} * \max(r, q) < 1$ reduces to $0 < \max(r, q) < 1$. But $q = |1-\rho| = |1 - re^{i\theta}| = \sqrt{1 - 2r\cos(\theta) + r^2}$. Therefore $q < 1 \Leftrightarrow r < 2\cos(\theta)$. Hence $0 < \max(r, q) < 1 \Leftrightarrow 0 < r < \min(1, 2\cos(\theta))$.

After establishing conditions on the subdivision parameter in Lemma 3.4 which guarantee that the length of a Bezier control polygon converges to zero under recursive subdivision, the natural question to ask is whether the control polygons converge and if so, to what do they converge? The answer is given in the following theorems.

Consider the Bezier curve given by the straight line segment $I = [0, 1]$ in the complex plane. For this line segment, recursive subdivision at the complex parameter $w = re^{i\theta}$ generates the following points:

$$\text{Level } 1 \quad : \quad w_{\frac{1}{2}} = w$$

$$\text{Level } k+1 \quad : \quad \begin{array}{l} w_{\frac{2j}{2^{k+1}}} = w_{\frac{j}{2^k}} \text{ and} \\ w_{\frac{2j+1}{2^{k+1}}} = w_{\frac{j}{2^k}} + w(w_{\frac{j+1}{2^k}} - w_{\frac{j}{2^k}}) \end{array}$$

At the same time that we subdivide the line segment $I = [0, 1]$ in the complex plane, we shall also subdivide I along the reals at the parameter $\rho = r\cos\theta$. Recursive subdivision at the real parameter ρ generates the following real numbers:

$$\text{Level } 1 \quad : \quad t_{\frac{1}{2}} = \rho$$

$$\text{Level } k+1 \quad : \quad \begin{array}{l} t_{\frac{2j}{2^{k+1}}} = t_{\frac{j}{2^k}} \text{ and} \\ t_{\frac{2j+1}{2^{k+1}}} = t_{\frac{j}{2^k}} + \rho(t_{\frac{j+1}{2^k}} - t_{\frac{j}{2^k}}) \end{array}$$

Notice that as k goes to infinity the values $\{t_{\frac{i}{2^k}}\}, i = 0, \ldots, 2^k$ densely cover I.

Let $z_k(t)$ be the piecewise linear function that for $j = 0, 1, \ldots, 2^k$ interpolates the point $w_{\frac{j}{2^k}}$ at the parameter $t_{\frac{j}{2^k}}$. By construction, $z_k(t)$ is the control polygon generated at the *kth* level of recursive subdivision at the parameter w starting from line segment I.

Theorem 3.5. *Let $z_k(t)$ denote the control polygon generated after k levels of recursive subdivision at the complex parameter $w = re^{i\theta}$, starting from the line segment $I = [0,1]$. If $0 < r < 1$ and $r < 2\cos\theta$, then the functions $z_k(t)$ converge to a continuous function $z(t)$. Moreover, $z(t_{\frac{i}{2^k}}) = w_{\frac{j}{2^k}}$.*

Proof. To prove that the functions $z_k(t)$ converge to a continuous function $z(t)$, it is enough to show that the functions $\{z_k(t)\}$ converge uniformly. Now by construction,

$$\max_{t \in I} |z_{k+1}(t) - z_k(t)| \leq \max_j \{|w||w_{\frac{j+1}{2^k}} - w_{\frac{j}{2^k}}|, |1-w||w_{\frac{j+1}{2^k}} - w_{\frac{j}{2^k}}|\}$$

Let $s = \max\{|w|, |1-w|\}$. Since by assumption $0 < r < 1$ and $r < 2\cos\theta$, we have $|w| < r < 1$ and $|1-w| < 1$, so $0 < s < 1$. Hence it follows by induction on k that

$$\max |z_{k+1}(t) - z_k(t)| = s^k, \quad 0 < s < 1$$

so the functions $\{z_k(t)\}$ converge uniformly. Therefore these piecewise linear functions converge to a continuous limit function $z(t)$. Moreover, $z(t_{\frac{i}{2^k}}) = w_{\frac{j}{2^k}}$ because $z_i(t_{\frac{j}{2^k}}) = w_{\frac{j}{2^k}}$, *for all* $i \geq k$. □

For a graphic illustration of theorem 3.5 see figure 8.

Observation: Before stating the main theorem, observe that when we start with a Bezier curve having a control polygon $[w_0, \ldots, w_n]$ and we run the de Casteljau algorithm to subdivide at some complex value w, the point produced at the apex of the pyramid is $B(w) = \sum_{k=0}^{n} b_k^n(w) w_k$. Thus, we obtain two control polygons the first one interpolating w_0 and $B(w)$ and the second interpolating $B(w)$ and w_n. If we keep subdividing, each point appearing at the apex is added the set of control points and is never removed. Furthermore, by Lemma 3.4 we know conditions on w which guarantee that the first and last points of each sub-control polygon are coming closer and closer together.

Theorem 3.6. *The control polygons generated by recursive subdivision of a degree n Bezier curve at $w = re^{i\theta}$, converge to the curve*

$$Q(t) = \sum_{k=0}^{n} b_k^n(z(t)) w_k \tag{7}$$

where $z(t)$ is the curve generated by recursive subdivision at w on the line segment $I = [0,1]$ and w_k are the original control points, provided that $0 < (r+q)^{n-1} * \max(r,q) < 1$ where $q = |1-w|$. Therefore any curve generated by recursive subdivision at w is a deformation of the curve $z(t)$.

Proof. Consider an arbitrary initial control polygon $P = [w_0, \ldots, w_n]$ and curve $Q(t) = \sum_{k=0}^{n} b_k^n(z(t))w_k$ which interpolates the first and last control points of P. After one level of recursive subdivision we generate two polygons $P_0 = [w_0, \ldots, B(w_{\frac{1}{2}})]$ and $P_1 = [B(w_{\frac{1}{2}}), \ldots, w_n]$. After one more level of subdivision we get $P_{00} = [w_0, \ldots, B(w_{\frac{1}{4}})]$, $P_{01} = [B(w_{\frac{1}{4}}), \ldots, B(w_{\frac{1}{2}})]$, $P_{10} = [B(w_{\frac{1}{2}}), \ldots, B(w_{\frac{3}{4}})]$ and $P_{11} = [B(w_{\frac{3}{4}}), \ldots, B(w_n)]$. Notice that we index the sub-control polygons attaching a digit 0 or 1 to the indices of the previous subdivision step to indicate left or right subdivision. Fig. 14 illustrates this process.

After k levels of recursive subdivision, we reach the control polygons $P_{b_1 \ldots b_k}$. To simplify our notation, we set $i_k = b_1 \ldots b_k$. Let $t_\infty = \lim_{k\to\infty} t_{i_k}$ and $w_\infty = \lim_{k\to\infty} w_{i_k} = \lim_{k\to\infty} z(t_{i_k}) = z(t_\infty)$. Recall that by Lemma 3.4 the length of P_{i_k} goes to zero as k goes to infinity. Therefore the P_{i_k} must converge to a point on $Q(t)$ since

$$\begin{aligned}
\lim_{k\to\infty} P_{i_k} &= \lim_{k\to\infty} [B(w_{i_k}), \ldots, B(w_{i_k + \frac{1}{2^k}})] \\
&= [B(w_\infty), B(w_\infty), \ldots, B(w_\infty)] \qquad (8) \\
&= B(w_\infty) = B(z(t_\infty)) = Q(t_\infty)
\end{aligned}$$

Conversely, for any $t^* \in [0,1]$ we can find a sequence t_{i_k} converging to t^*. Therefore for every point $Q(t^*)$ we can find a sequence of control polygons $\{P_{i_k}\}$ converging to $Q(t^*)$. As a result, the control polygons generated by recursive subdivision converge to the continuous curve $Q(t) = \sum_{k=0}^{n} b_k^n(z(t))w_k,\ t \in [0,1]$ $\square$

Theorems 3.5 and 3.6 are presented for binary subdivision i.e., when we subdivide at only one point r at each step. However the same results are true when we subdivide at multiple points i.e., in one step we subdivide the Bezier curve into multiple segments. The proofs are similar, and the only subtle point is to make sure that the combination of different subdivision parameters constitutes a contractive map, i.e. if we subdivide at $r_1, r_2, \ldots, r_n$, the conditions of Lemma 3.4 must be satisfied for $r = r_1 * \ldots * r_n$.

When the control points are evenly spaced along the straight line segment $I = [0,1]$ – that is, when the control points are given by $w_k = \frac{k}{n},\ k = 0, \ldots, n$ – then by Theorem 3.6 the curve $Q(t)$ generated by recursive subdivision at the complex parameter w is the same as the curve z(t) generated by recursive subdivision of the line segment I at the complex parameter w, since

$$Q(t) = \sum_{k=0}^{n} \frac{k}{n} b_k^n(z(t)) = z(t) \tag{9}$$

Often $z(t)$ consists of a collection of straight line segments. When the control points $w_0, \ldots, w_n$ are moved off the real line, then the line segments of $z(t)$ are mapped to polynomial curves, since the Bernstein polynomial

$$P(z) = \sum_{k=0}^{n} b_k^n(z) w_k \tag{10}$$

maps straight lines into polynomials of degree (less than or equal to) n. For example, the warped Sierpinski triangle in Fig. 11 consists of parabolic segments, since in this example $P(z)$ has three control points and hence is a quadratic polynomial. Moreover, the parabolic segment at the base of the deformed Sierpinski triangle in Fig. 11 is the real Bezier curve for the given control points, since $P(z)$ maps the real line segment I into the real Bezier curve

$$P(t) = \sum_{k=0}^{n} b_k^n(t) w_k, \quad 0 \leq t \leq 1 \tag{11}$$

Notice too that the angles between the parabolic segments in the warped Sierpinski triangle of Fig. 11 are the same as the angles between the line segments of the original Sierpinski triangle in Fig. 2. In general, we have the following result.

Corollary 3.7. *Let $B(t)$ be the Bezier curve with control points $w_0, \ldots, w_n$ built by recursive subdivision at some complex parameter w. The deformations generated by manipulating the Bezier control points are conformal transformations i.e., they preserve the angles of $B(t)$.*

Proof. A well known result in complex analysis states that analytic functions are conformal transformations of the complex plane – that is, they preserve angles [2]. But Theorem 3.6 states that by subdivision we obtain the curve $P(t) = \sum_{k=0}^{n} b_k^n(z(t))w_k$ which is a transformation of the curve $z(t)$ through the polynomial $\sum_{k=0}^{n} b_k^n(z)w_k$. Since polynomials are analytic functions, the angles of $z(t)$ are preserved for any choice of control points. □

Fractal Parameterization An important consequence of generating fractals using Bezier subdivision is that we can create parameterizations for fractals in a natural manner, as explained in the proof of Theorem 3.6. For any point P^* on the fractal we can find a sequence P_{i_k} such that $P^* = \lim_{k\to\infty} P_{i_k} = B(z(t^*))$. We can now recover the parameter value t^* as $\lim_{k\to\infty} t_{i_k}$. In practice, after just a few levels of subdivision we can get an accurate approximation of the parameter as the control polygon and the curve become almost indistinguishable; see for example, figures 8 and 9. After some levels of subdivision the control polygon practically matches the fractal curve and we can retrieve the parameter value.

4 B-Spline Curves

4.1 Real B-Spline Curves

B-splines are piecewise polynomials. To describe a B-spline curve of degree n, we need to provide a sequence of real numbers $T = \{t_1, \ldots, t_{v+n}\}$ that represent the parameters of the connection points between the polynomials. The real numbers T are typically called *knots* in the technical literature on splines. We also need a set of control points $P = \{P_0, \ldots, P_v\}$ that define the shape of the curve. Each polynomial segment of the B-spline curve lies over a parameter interval $[t_k, t_{k+1}]$ and is affected by the $n+1$ control points $P_{k-n}, \ldots, P_k$, and the $2n$ knots $t_{k-n+1}, \ldots, t_{k+n}$. B-spline curves also admit a compact representation as:

$$B(t) = \sum_k N_k^n(t|t_k, \ldots, t_{k+n+1})P_k, \quad t_n \leq t \leq t_v \tag{12}$$

where the piecewise polynomial B-spline basis functions N_k^n are more complicated than the Bernstein basis functions and are described e.g., in [7]. To evaluate $B(t)$, we use the de Boor algorithm which shares certain similarities with the de Casteljau algorithm described in Sec. 3.1. After locating the interval $[t_k, t_{k+1}]$ in which t lies, we use the relevant knots and control points in a pyramid as shown in Fig. 15 to evaluate $B(t)$. This figure is based on properties of *blossoming* (see [14, 15]).

Blossoming is a powerful tool that significantly simplifies the analysis of B-spline algorithms. For any degree n polynomial $P(t)$ the *blossom* is the unique symmetric multiaffine function $p(u_1, ..., u_n)$ that reduces to $P(t)$ along the diagonal. That is, $p(u_1, ..., u_n)$ is the unique multivariate polynomial satisfying the following three axioms:

- *Symmetry*

$$p(u_1, ..., u_n) = p(u_{\sigma(1)}, ..., u_{\sigma(n)}) \quad for\ every\ permutation\ \sigma\ of\ 1, \ldots, n$$

- *Multiaffine*

$$p(u_1, ..., (1-a)u_k + v_k, ..., u_n) =$$
$$(1-a)p(u_1, ..., u_k, ..., u_n) + ap(u_1, ..., v_k, ..., u_n)$$

- *Diagonal*

$$p(t, ..., t) = P(t)$$

In the following discussion, we will often use the *dual function* property of the blossom which states that if $P(t)$ is one segment of a B-spline curve with knots $t_i, \ldots, t_{i+2n}$ and control points $P_j, \ldots, P_{j+n}$, then $P_k = p(t_{k+1}, \ldots, t_{k+n}),\ k = j, \ldots, j+n$. To simplify our notation, we shall often write $t_{k+1}t_{k+2}\ldots t_{k+n}$ in place of $p(t_{k+1}, \ldots, t_{k+n})$.

Analogous to subdivision procedures for Bezier curves, for B-spline curves we have *knot insertion* algorithms. In knot insertion we insert new knots to form a new knot sequence $K = \{\tau_1, \ldots, \tau_{\mu+n}\}, T \subset K$. The goal of knot insertion procedures is to compute a new set of control points $Q = \{Q_0, \ldots Q_\mu\}$ so that the B-spline curve for the new control points and new knots is identical to the B-spline curve for the original control points and original knots i.e., $B(t|T, P) = B(t|K, Q)$ (see Fig. 16).

There are many different knot insertion algorithms, including both local and global algorithms. Local algorithms insert new knots between one pair of consecutive knots, while global algorithms insert one new knot between every pair of consecutive knots. Boehm's algorithm is a popular

local knot insertion algorithm [4]. Suppose that we have the initial knot sequence $\ldots, t_{k-n+1}, \ldots, t_k, t_{k+1}, \ldots, t_{k+n}, \ldots$ for a degree n curve. We want to insert the knot u between t_k and t_{k+1} to obtain the new knot sequence $\ldots, t_{k-n+1}, \ldots, t_k, u, t_{k+1}, \ldots, t_{k+n}, \ldots$. We need to use the original control points $b(t_{k-n+1}, \ldots, t_k), \ldots, b(t_{k+1}, \ldots, t_{k+n})$ to compute the new control points $b(t_{k-n+1}, \ldots, t_k)$, $b(t_{k-n+2}, \ldots, t_k, u), \ldots,$
$b(u, t_{k+1}, \ldots, t_{k+n-1})$, $b(t_{k+1}, \ldots, t_{k+n})$. The computation of the new control points amounts simply to running only one layer of the de Boor algorithm and reading off the new control points at the spine of the pyramid. An example for a cubic spline is shown in Fig. 17. Later we will also use Schaefer‘s global knot insertion algorithm [16], because it is simple to implement and it allows a great deal of flexibility concerning where the new knots can be inserted (in contrast e.g., to the Lane-Riesenfeld algorithm [11]).

If we insert more and more knots so that the knot spacing approaches zero, the control polygons converge to the B-spline curve for the original control polygon. Thus, knot insertion for B-spline curves behaves a lot like subdivision for Bezier curves. In the next section we explore how we can use knot insertion to create B-spline curves with knots in the complex plane.

4.2 Complex B-spline Curves

To extend B-splines to the complex domain, we treat the control points $\{w_i\}$ as complex numbers and we also allow the knots $\{z_j\}$ to take on complex values. Now for each consecutive pair of knots, we define a polynomial by setting

$$P_{I(z_v,z_{v+1})}(z) = \sum_k N_k^n(z|z_k,\ldots,z_{k+n+1})_{I(z_v,z_{v+1})} w_k, \quad z, z_k, \ldots, z_{k+n+1} \in \mathbb{C} \tag{13}$$

We think of the complex B-spline curve as the collection of all the complex polynomials $P_{I(z_v,z_{v+1})}$. Notice that one needs to be more careful in this extension than in the extension of Bezier curves to the complex domain. First of all, notice that both B and N_k^n have an indicator function $I(z_v, z_{v+1})$ as a subscript. In the real setting, to evaluate a B-spline at some parameter t, we first find the pair of knots between which t lies, and then run the de Boor algorithm [7] using the knots and control points that affect this particular interval. In the complex domain however, given a complex point z it is unclear in which knot interval the evaluation should take place. Therefore, the indicator function I is required.

Another critical point is that the knots of a B-spline must always form a *progressive sequence*. The condition for a sequence of knots to be progressive is that $z_{j+n} \neq z_i, 1 \leq j \leq i \leq n$, where n is the degree of the B-spline curve. This condition is necessary to avoid zero denominators in the de Boor algorithm. In the case of real B-splines one does not have to worry about this constraint, since the knots are assumed to be in increasing order. Such an ordering no longer makes sense in the complex plane and care must be taken not to violate the progressive sequence condition.

As an example, let us try to generate the C-curve using Boehm's knot insertion algorithm. We can use a quadratic spline for which we need three control points P_0, P_1, P_2 and four knots which we take to be $t_i = i,\ i = 1, 2, 3, 4$. To create the C-curve, we try to insert knots *between* t_2 and t_3 so that the knots themselves form a C-curve. After one step of Boehm's algorithm, the knot $k_1 = t_2 + rt_3 = 3.5 + i1.5$ is inserted and the knot sequence is $K = t_1, t_2, k_1, t_3, t_4$. Next, we insert $k_2 = t_2 + rk_1 = 3 + i2.5$ and $k_3 = k_2 + rk_3 = 4.5 + i4$. Now $K = t_1, t_2, k_2, k_1, k_3, t_3, t_4$. Without showing all the intermediate computations, assuming that we continue inserting knots in the same fashion, the knot sequence will first be $K = \ldots, k_2, k_4, k_1, k_5, k_3, \ldots$ and then $K = \ldots, k_4, k_6, k_1, k_7, k_5, \ldots$ where $k_6 = k_7$ which violates the progressive knot sequence condition! If we try to continue the computation at this point, our knot insertion algorithm will have to do divisions by zero and fail. See Fig. 18 for an illustration of this phenomenon.

Nevertheless, inserting complex knots is a powerful and natural extention, which allows us to generate fractals using knot insertion algorithms (see e.g., Fig. 21). Before proceeding, however, we need to generalize the notion of a knot line.

Consider a sequence of complex knots $z_0, \ldots, z_K$. Assuming some interpolation method between each pair, we can form a continuous curve $z(t)$. Each knot z_j in the knot sequence lies on the curve $z(t)$ and has a corresponding parameter value t_j. Thus, if we need to evaluate a complex B-spline curve at $z(t^*)$ and $t_j \leq t^* \leq t_{j+1}$, then we also know that the complex knot interval on the curve $z(t)$ where $z(t^*)$ lies is $[z(t_j), z(t_{j+1})]$. Thus the curve $z(t)$ replaces the knot line. If we now substitute $z(t)$ for the complex variable z in our definition of complex B-splines we obtain a curve defined over t which is the transformation of $z(t)$ through the B-spline basis functions N_k^n :

$$B(t) = P(z(t)) = \sum_k N_k^n(z(t)|z_k, \ldots, z_{k+n+1})w_k \tag{14}$$

Notice that the knot curve $z(t)$ eliminates the need for the indicator functions $I(z_v, z_{v+1})$.

Theorem 4.1. *Convergence: As the distance between the knots approaches zero, the control polygon converges to the curve:*

$$B(t) = \sum_k N_k^n(z(t)|z_k, \ldots, z_{k+n+1})w_k \tag{15}$$

where $z(t)$ is the knot curve, z_k are the original knots and w_k are the original control points.

Proof. After j knot insertion steps, let the knot sequence be $Z^j = \{z_1^j, \ldots, z_M^j\}$ and the control points be $W^j = \{w_1^j, \ldots, w_K^j\}$. Z^j, W^j define a set of polynomials $P_{I(z_v,z_{v+1})}(z) : \mathbb{C} \to \mathbb{C}$, whose blossoms [14, 15] we denote by $p_{I(z_v,z_{v+1})}$. After one knot insertion step, Z^{j+1} has a few more knots than Z^j, and W^{j+1} changes shape, but the polynomials described by Z^{j+1}, W^{j+1} are still $P_{I(z_v,z_{v+1})}(z)$ (see Fig. 19).

Let $I(z_n^j, z_{n+1}^j)$ be a sequence of subintervals of $I(z_v, z_{v+1})$ along the knot curve $z(t)$, converging to a point z_c in the interval $I(z_v, z_{v+1})$. Let $P_v = P_{I(z_v,z_{v+1})}$ and p_v be the blossom of P_v. Since p_v is a continuous function, as j goes to ∞, it follows by the dual functional and diagonal properties of the blossom that $w_k^{j+1} = p_v(z_{k+1}^{j+1}, \ldots, z_{k+n}^{j+1}) \to p_v(z_c, \ldots, z_c) = P(z_c)$. Thus the control points converge to the points along the curve $P(z(t))$. Therefore:

$$B(t) = P(z(t)) = \sum_k N_k^n(z(t)|z_k, \ldots, z_{k+n})w_k \tag{16}$$

Notice in particular that by restricting ourselves to the slice of the complex plane described by the curve $z(t)$, we no longer need the indicator function, since t is enough to identify the knot interval that must be used to evaluate $P(z(t))$. $\square$

Since Bezier curves are a special case of B-spline curves, Theorem 3.6 is just a special case of Theorem 4.1. Figures 20 - 23 illustrate the convergence discussed in Theorem 4.1.

One way for the knot distances to converge to zero is to populate the line intervals between the original knots without changing the shape of the knot curve $z(t)$. This approach leaves the piecewise linear knot curve generated by the knots unchanged. A more general case is to allow the piecewise linear knot curve to change and converge, for example, to a fractal. In this case, knot insertion alters the knot curve, but once again the control polygon eventually converges.

Theorem 4.2. *Consider a B-spline curve $B(t)$ of degree n with knot curve $z(t)$ in C^k. Then $B(t)$ is in C^l where $l = \min(k, n-1)$. Moreover, if $n \geq 2$, the deformations generated by manipulating the B-spline control points are conformal transformations i.e., they preserve the angles of $B(t)$.*

Proof. By the chain rule: $B'(t) = P'(z(t))z'(t)$. Therefore $B(t)$ is in C^l where $l = \min(k, n-1)$. Furthermore, if the B-splines have degree 2 or higher, this formula guarantees the differentiability of $B(t)$ at the knots. Together with the fact that $P(z)$ is analytic (a polynomial), this differentiability implies that $B(t)$ is a conformal transformation of the knot curve $z(t)$ for any choice of control points (see also [2]). $\square$

Figure 22 shows an example of knots that converge to a smooth curve. The control polygon also converges to a smooth curve. In Fig. 23 we show an example where the knots converge to a closed curve (unit circle), but the corresponding B-spline curve is not closed.

To render a B-spline curve $P(z(t))$ with complex knots, we use the de Boor algorithm. In particular, for each complex number z^* we run the de Boor algorithm to compute $P(z^*)$ just as in the real case. To render the B-spline curve, we need to compute a point on the curve for each point $z(t)$ on the knot curve. In practice, when we move from knot to knot, we use linear interpolation, i.e. $z(t) = z_k + t(z_{k+1} - z_k),\ t \in [0,1], k = 1, \ldots, M-1$ and run the de Boor algorithm for all those intermediate points.

5 Summary, Extensions and Future Work

We have extended the formulation of Bezier and B-spline curves from the real numbers to the complex domain. We showed that the de Casteljau subdivision algorithm for Bezier curves and standard knot insertion algorithms for B-splines still converge when the knot spacing approaches zero, even for complex knots. By taking advantage of the power of complex multiplication, we are able to generate many classical fractals such as the Sierpinski triangle and the Koch curve, using standard Bezier subdivision and B-spline knot insertion procedures. This approach allows us to control the shape of fractals using control points, and by manipulating these control points to generate warped but conformal versions of these fractal shapes that would otherwise be difficult to produce. Moreover, we are able to introduce intuitive parameterizations for fractals; a feature that was not previously available.

Other extensions are possible. For example, we could also extend rational Bezier and rational B-spline curves to the complex domain by associating a complex weight with each complex control point. Thus the control structure for a complex rational Bezier or complex rational B-spline curve would consist of pairs $(m_0w_0, m_0), \ldots, (m_kw_k, m_k)$, where $m_i, w_i,\ i = 0, \ldots, k$, are complex numbers. The subdivision and knot insertion procedures proceed component wise, but are otherwise the same as in the integral case. However when we want to view the new control polygons, we must divide the first component of each mass-point by the second component (the complex mass) and then display the first component as a point in the complex plane. Our theorems concerning convergence and conformality still hold in the rational setting, and the proofs are much the same. One additional advantage, however, of complex rational Bezier and complex rational B-spline schemes is that they would allow us to represent rational transformations of the plane, which include inversions of the plane that are not incorporated in polynomial or piecewise polynomial schemes. Since complex rational linear transformations can map lines into circles [2], with the right choice of complex control points and complex weights this approach would allow us to generate circular arcs in the complex plane simple by inserting a dense sequence of knots along subintervals of the real line.

In the future we would like to investigate whether Bezier subdivision algorithms and B-spline knot insertion procedures can also be extended to generate quaternion curves, a natural generalization of the complex numbers to 4-dimensions. Here instead of quaternion multiplication, the basic operation might be sandwiching, since sandwiching a vector between unit quaternions can be used to generate conformal transformations on vectors in 3-dimensions.

Finally, we hope to investigate whether we can use the same or similar algorithms to obtain control points for fractals such as the Julia sets which are attractors of an IFS. Such algorithms might allow us to alter the shape of these fractals in a natural way by adjusting their associated Bezier or B-spline control points.

Acknowledgements

We would to thank Professor David Levin for his insightful comments and especially his observation that moving the control points associated with the complex Bezier and B-spline curves introduced here induces conformal transformations on the associated fractal curves. We would also like to thank Thomas Kloecker for carefully proofreading the original manuscript and pointing out an error in the original statement and proof of Lemma 3.4.

References

[1] H. Abelson and A. DiSessa. *Turtle Geometry: The Computer as a Medium for Exploring Mathematics.* MIT Press, 1986.

[2] Lars V. Ahlfors. *Complex Analysis.* McGraw-Hill, 1979.

[3] M. F. Barnsley. *Fractals Everywhere, Second Edition.* Academic Press, 1993.

[4] W. Boehm. Inserting new knots into B-spline curves. *Computer Aided Design*, 12:99–101, 1980.

[5] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide.* Academic Press, 2002.

[6] B. Foster, T. Blu, and M. Unser. Complex B-splines. *Appl. Comp. Harmon. Anal.*, 20:281–282, 2006.

[7] Ron Goldman. *Pyramid Algorithms: A Dynamic Programming Approach to Curves and Surfaces for Geometric Modeling.* Morgan Kaufmann, 2002.

[8] Ron Goldman. The fractal nature of Bezier curves. In *Proceedings of Geometric Modeling and Processing : Theory and Applications*, pages 3–11, 2004.

[9] Ron Goldman. *An Integrated Approach to Computer Graphics and Geometric Modeling.* Francis and Taylor, 2009.

[10] T. Ju, Scott Schaefer, and Ron Goldman. Recursive turtle programs and iterated affine transformations. *Computers and Graphics*, 28:991–1004, 2004.

[11] J. Lane and R. Riesenfeld. A theoretical development for the computer generation and display of piecewise polynomial surfaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2:35–46, 1980.

[12] Charles A. Micchelli and Hartmut Prautzsch. Computing surfaces invariant under subdivision. *Computer Aided Geometric Design*, 4:321–328, 1987.

[13] Hartmut Prautzsch and Charles A. Micchelli. Computing curves invariant under halving. *Computer Aided Geometric Design*, 4:133–140, 1987.

[14] L. Ramshaw. Beziers and B-splines as multiaffine maps. In R.A. Earnshaw, editor, *NATO ASI*, volume 40, pages 757–776. Springer-Verlag, 1988.

[15] L. Ramshaw. Blossoms are polar forms. *Computer Aided Geometric Design*, 6:323–358, 1989.

[16] Scott Schaefer and Ron Goldman. Non-uniform subdivision for B-splines of arbitrary degree. *Computer Aided Geometric Design*, 26:75–

81, 2009.

[17] Scott Schaefer, Ron Goldman, and D. Levin. Subdivision schemes and attractors. In *Proceedings of Symposium on Geometric Processing*, pages 171–180, 2005.

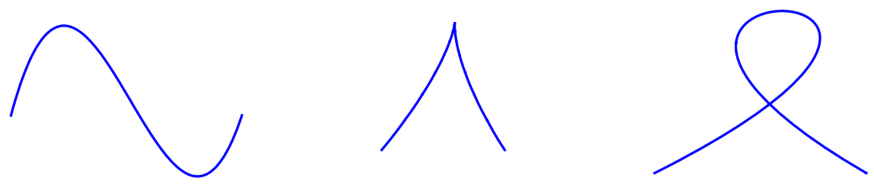

Figure 1: Polynomial and spline curves can have only a finite number of inflections (left), cusps (middle) and self-inflection points (right).

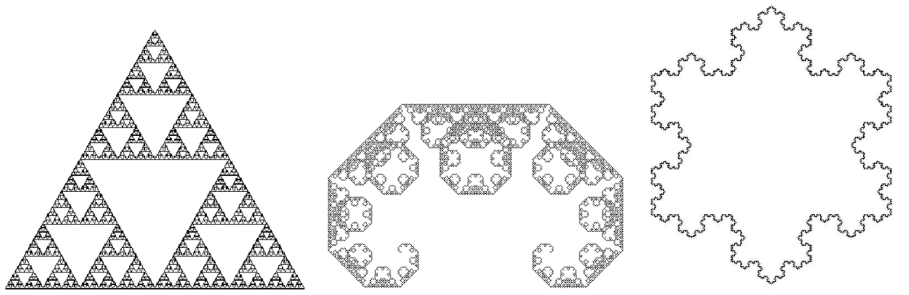

Figure 2: Three famous fractals: The Sierpinski triangle (left), the C curve (middle) and the Koch Snowflake (right – composed of six Koch curves). Fractals can have exotic properties. The Sierpinski gasket has a (fractal) dimension given by $d = \frac{log(3)}{log(2)} \approx 1.585$, while the Koch snowflake is continuous everywhere but differentiable nowhere.

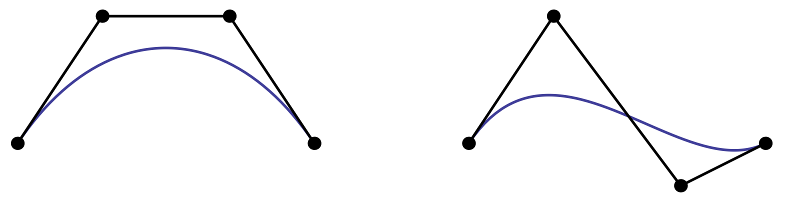

Figure 3: A simple cubic Bezier curve together with its initial control polygon (left). After we perturb the control points, we change the control polygon and the curve also changes accordingly (right).

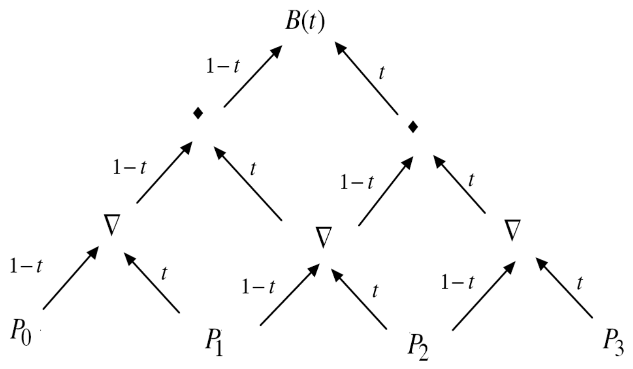

Figure 4: De Casteljau algorithm to evaluate points on a cubic Bezier curve.

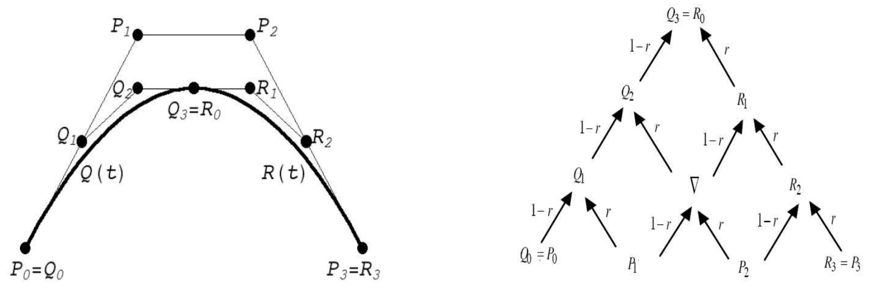

Figure 5: De Casteljau algorithm used to subdivide a curve at the parameter r. Off the left and right sides of the pyramid (right), we can read the control points $Q_0, \ldots, Q_3$ and $R_0, \ldots, R_3$ for the new control polygons (left).

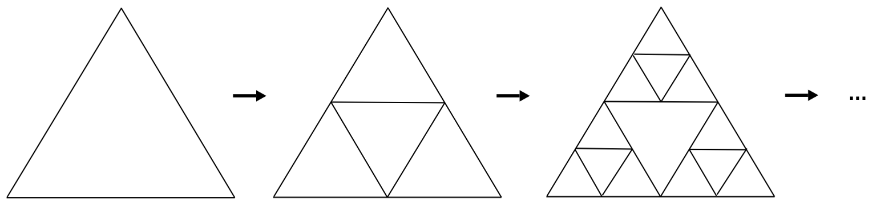

Figure 6: Starting from a triangle, we can apply an iterated function system consisting of scaling by 0.5 about each of the three vertices of the triangle to generate the Sierpinski triangle.

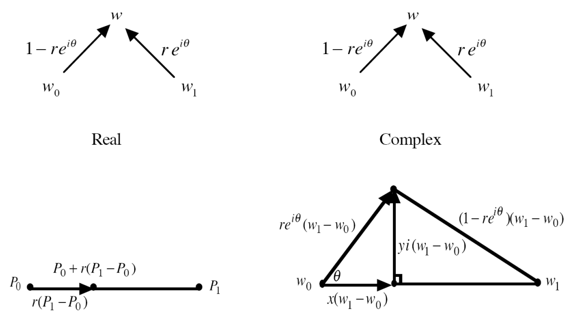

Figure 7: One step of subdivision in the real (left) and complex (right) domains.

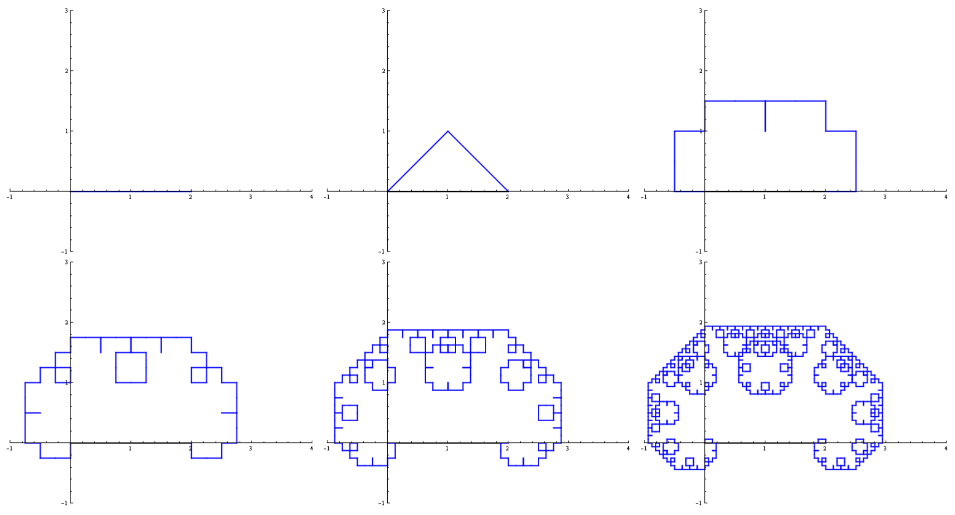

Figure 8: Recursive subdivision of a Bezier curve (straight line) at $r = 0.5 + i0.5$ generates the C curve (subdivision levels $0, 1, 4, 6, 8, 10$ are illustrated here).

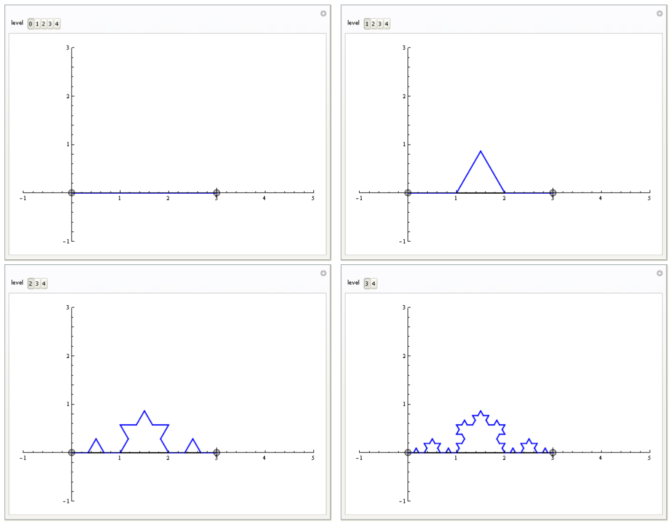

Figure 9: Recursive subdivision of a Bezier curve at $r_1, r_2, \ldots$ generates the Koch curve (subdivision levels $0, 1, 2, 3$ are illustrated here).

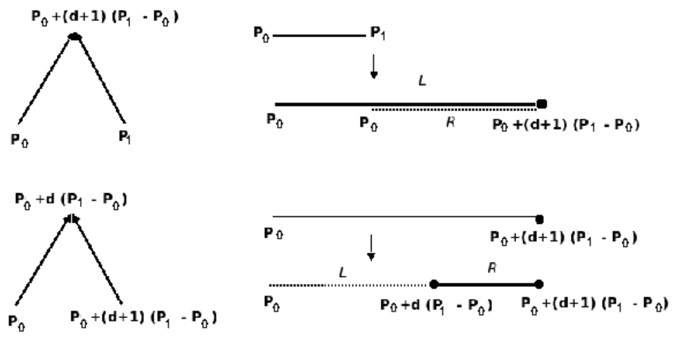

Figure 10: We can achieve translation through subdivision by two consecutive scalings, selecting first the left and then the right segment.

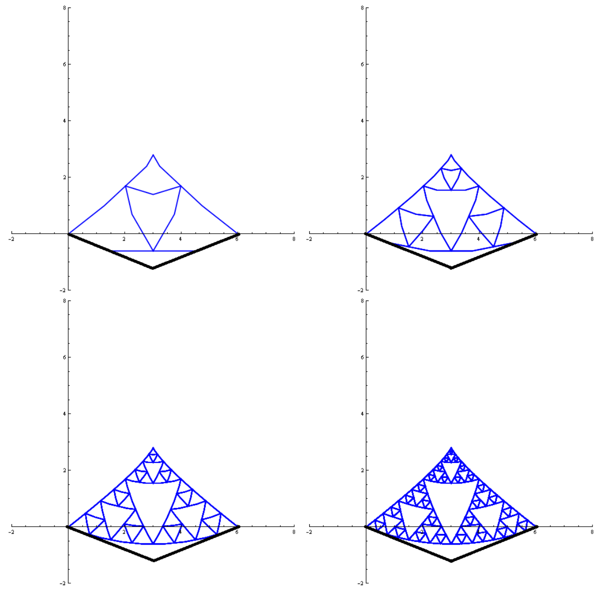
Figure 11: A warped Sierpinski triangle, generated by recursive subdivision with complex subdivision parameters applied to a quadratic Bezier control polygon (subdivision levels $1, 2, 3, 4$ are illustrated here). The bold lines show the Bezier control polygon.

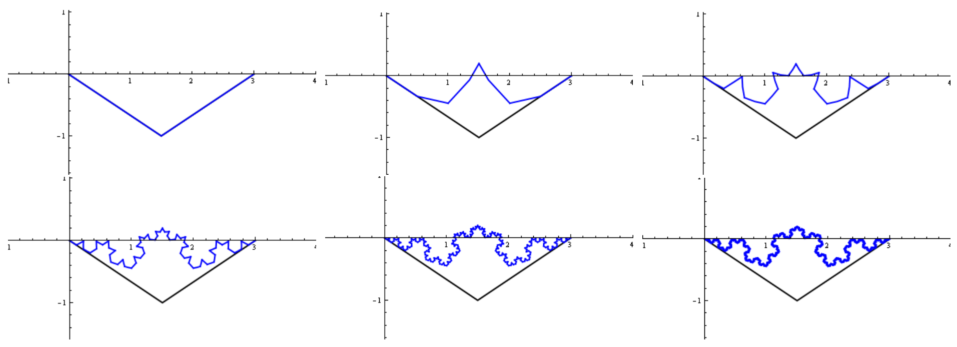

Figure 12: A warped Koch curve, generated by recursive subdivision with complex subdivision parameters applied to a quadratic Bezier control polygon (subdivision levels $0, 1, 2, 3, 4, 5$ are illustrated here). Again the bold lines show the Bezier control polygon.

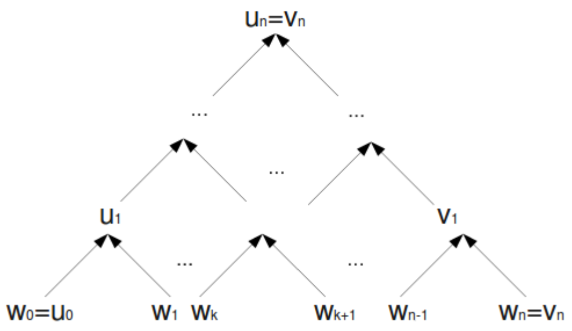

Figure 13: Pyramid created by the de Casteljau algorithm during one subdivision step.

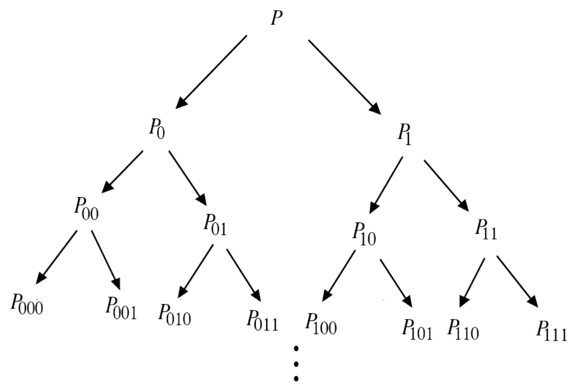

Figure 14: Recursive subdivision of a Bezier control polygon. Appending the index 0 to the left subpolygons and the index 1 to the right subpolygons yields indices that can be viewed as binary fractions.

$$\begin{tikzcd}
 & & & ttt & & & \\
 & & t_4tt \arrow[ur, "t_5 - t"] & & t_5tt \arrow[ul, "t - t_4"'] & & \\
 & t_3t_4t \arrow[ur, "t_5 - t"] & & t_4t_5t \arrow[ul, "t - t_3"'] \arrow[ur, "t_6 - t"] & & t_5t_6t \arrow[ul, "t - t_4"'] & \\
t_2t_3t_4 \arrow[ur, "t_5 - t"] & & t_3t_4t_5 \arrow[ul, "t - t_2"'] \arrow[ur, "t_6 - t"] & & t_4t_5t_6 \arrow[ul, "t - t_3"'] \arrow[ur, "t_7 - t"] & & t_5t_6t_7 \arrow[ul, "t - t_4"']
\end{tikzcd}$$

Figure 15: De Boor algorithm to evaluate a cubic B-spline curve in the segment $[t_4, t_5]$ using knots $t_2 \leq \ldots \leq t_7$. The value $B(t) = b(t, t, t)$ emerges at the apex. The coefficients appearing along the arrows must be normalized by denominators which are not shown. The denominators can easily be recovered as the sum of the coefficients of the arrows entering each node e.g. $t_5 - t$ and $t - t_2$ must be divided by $t_5 - t + t - t_2 = t_5 - t_2$.

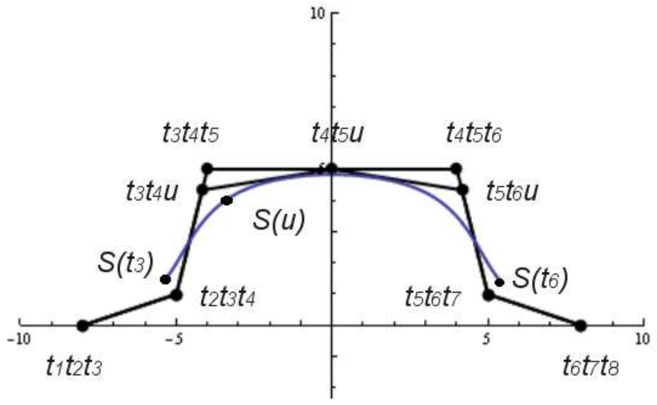

Figure 16: Inserting a knot u between t_4 and t_5 changes the knot sequence and control polygon but not the curve.

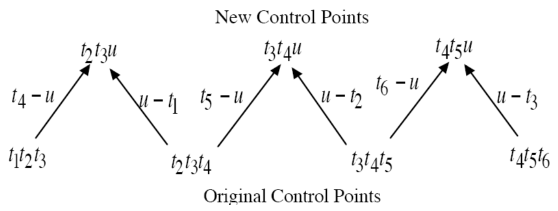

Figure 17: Boehm's local knot insertion algorithm for a cubic spline. Running one layer of the de Boor algorithm yields the new control points at the top.

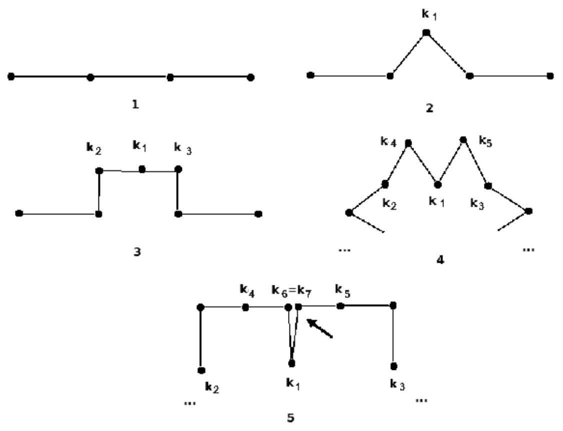

Figure 18: Inserting complex knots that form a C-curve in knot space ends up violating the progressive sequence constraint. For degree $n = 2$ we end up with knots k_6 and k_7 which are distance two apart in the knot sequence and are equal in the complex plane.

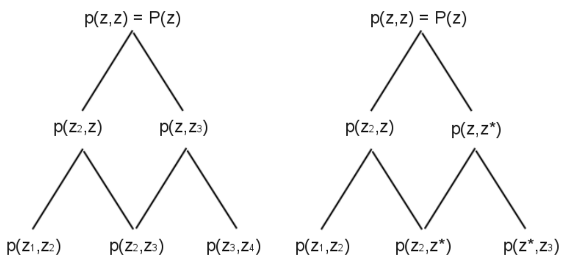

Figure 19: Knot insertion does not change the B-spline polynomials defined by the control points and knots. For example, on the left we have the de Boor algorithm for a quadratic curve with just three control points described by their blossoms $p(z_1, z_2), p(z_2, z_3), p(z_3, z_4)$. At the apex we compute the value of the polynomial $P(z)$ at the point z. If we insert a knot $z*$ between z_2 and z_3, then on the right hand side, we evaluate at the point z and again get $P(z)$ this time using the new control points $p(z_1, z_2), p(z_2, z*), p(z*, z_3)$.

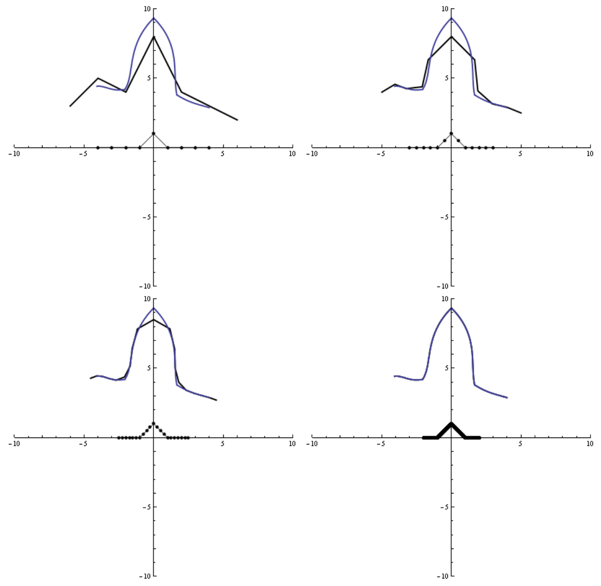

Figure 20: Inserting complex knots that populate the line intervals between knots without changing the shape of the knot curve. The picture shows the evolution of the control polygon along with the knot curve directly beneath the control polygon.

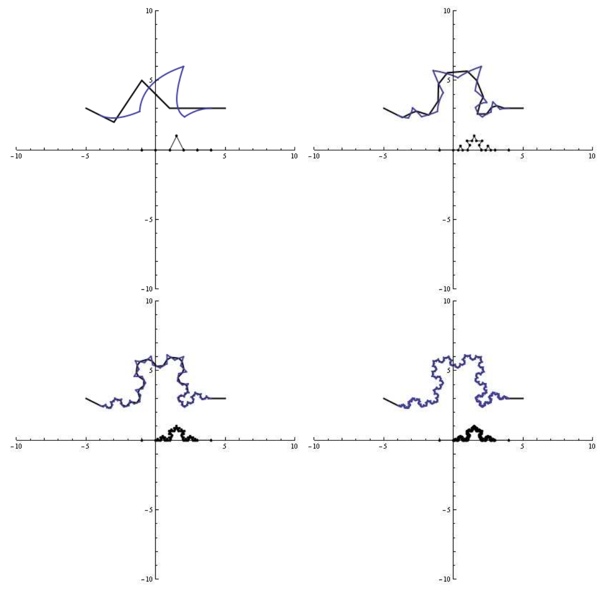

Figure 21: Inserting complex knots that converge to a continuous curve c (here the fractal Koch curve), forces the B-spline curve and the control polygon to converge to a warped version of c. The picture shows the evolution of the control polygon and the B-spline curve along with the knot curve directly below the other two curves.

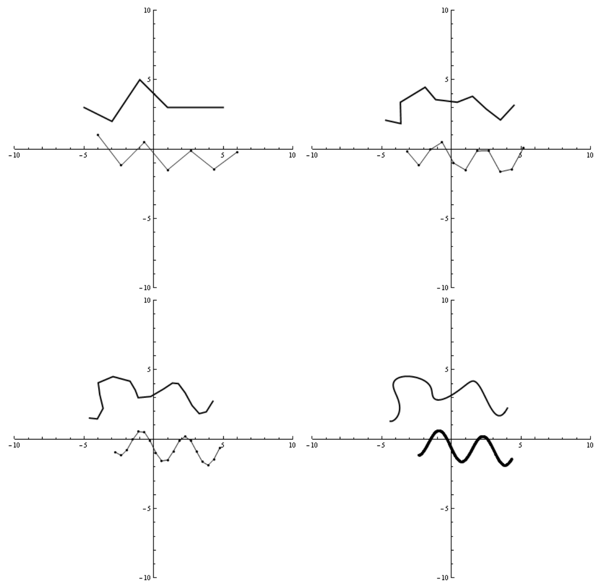

Figure 22: Inserting complex knots that form a smooth curve generates a smooth B-spline curve. The picture shows the evolution of the control polygon along with the knot curve directly below the control polygon.

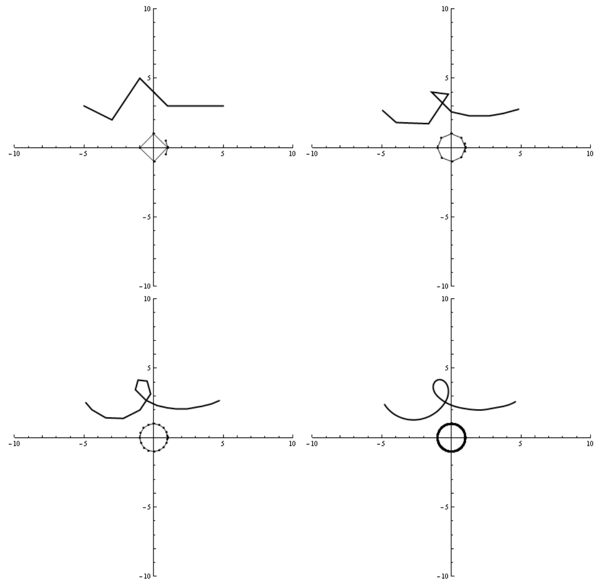

Figure 23: Inserting complex knots that form a circle generates a smooth B-spline curve that is not necessarily a closed curve. The picture shows the evolution of the control polygon along with the knot curve directly below the control polygon.