Brigham Young University
BYU ScholarsArchive

All Theses and Dissertations

2015-06-01

Evaluation and Refinement of Generalized B-splines

Ian Daniel Henriksen
Brigham Young University - Provo

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Mathematics Commons

BYU ScholarsArchive Citation

Henriksen, Ian Daniel, "Evaluation and Refinement of Generalized B-splines" (2015). *All Theses and Dissertations*. 5887.
https://scholarsarchive.byu.edu/etd/5887

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in All Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.

Evaluation and Refinement of Generalized B-Splines

Ian Daniel Henriksen

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Emily J. Evans, Chair
Jeffrey Humpherys
Kevin Tew

Department of Mathematics

Brigham Young University

June 2015

Copyright © 2015 Ian Daniel Henriksen

All Rights Reserved

ABSTRACT

Evaluation and Refinement of Generalized B-Splines

Ian Daniel Henriksen
Department of Mathematics, BYU
Master of Science

In this thesis a method for direct evaluation of Generalized B-splines (GB-splines) via the representation of these curves as piecewise functions is presented. A local structure is introduced that makes the GB-spline curves more amenable to the integration used in constructing bases of higher degree. This basis is used to perform direct computation of piecewise representation of GB-spline bases and curves. Algorithms for refinement using these local structures are also developed.

Keywords: GB-splines, Trigonometric splines, Hyperbolic splines, Exponential splines, Curve Refinement, Knot Insertion, Degree Elevation

Contents

Contents **iii**

List of Tables **iv**

List of Figures **v**

1 Introduction **1**

1.1 Notation 3

2 Background **3**

2.1 Bernstein Polynomials 4

2.2 Bézier Curves 16

2.3 B-splines 20

3 Definition of GB-splines **27**

4 Evaluation of GB-splines **35**

5 Refinement operations on GB-splines **54**

5.1 Algorithms For Refinement 57

6 Stability of Evaluation **78**

7 Future Directions **82**

Appendices **85**

A Conventions and Notation for Algorithms **86**

B Code for Computation of Basis Coefficients **89**

Bibliography **100**

List of Tables

1.1 Notational conventions. 3

A.1 Notational conventions used in algorithms. 87

List of Figures

2.1 The Bernstein basis polynomials of degrees 2, 3, 4, and 5. 5

2.2 Polynomials in Bernstein form of degrees 2, 3, 4, and 5 (green) shown with their Bernstein coefficients (blue). 6

2.3 An illustration of the De Casteljau algorithm (Algorithm 1) for evaluating polynomials in Bernstein form. Here, parameter values $\frac{1}{3}$ and $\frac{2}{3}$ are used. . . 7

2.4 Bézier curves of degrees 3 and 4. 17

2.5 The De Casteljau algorithm (Algorithm 1) applied to a Bézier curve at parameter values $\frac{1}{3}$ and $\frac{2}{3}$. 18

2.6 B-spline bases of degrees 1, 2, 3, and 4. 21

2.7 A degree 4 B-spline curve. 22

2.8 B-spline bases of degrees 1, 2, 3, and 4. 23

2.9 The geometric representation of the De Boor recurrence (given in Definition 2.33) shown here at parameter values 0.2, 0.8, 1.2, 1.8, 2.2, and 5.8. 25

3.1 Bernstein-like GB-spline basis functions formed by using $\cos\left(\frac{\pi}{2}\left(t-t_i\right)\right)$ and $\sin\left(\frac{\pi}{2}\left(t-t_i\right)\right)$ as the knot functions. 29

3.2 B-spline-like GB-spline bases of degree 2 and 3 formed by using $\cos\left(\frac{\pi}{2}\left(t-t_i\right)\right)$ and $\sin\left(\frac{\pi}{2}\left(t-t_i\right)\right)$ as the knot functions. 30

3.3 A GB-spline curve equal to different circles on different parts of its domain. . 31

3.4 A GB-spline curve over a uniform knot vector that exactly represents a circle. 32

4.1 The local polynomial representation for a uniform B-spline basis function of degree 3 compared with the local representation for a uniform GB-spline function of degree 3 defined using trigonometric knot functions. 36

4.2 The local polynomial representation for a uniform B-spline basis function of degree 4 compared with the local representation for a uniform GB-spline function of degree 4 defined using trigonometric knot functions. 37

4.3 The local polynomial representation for a uniform B-spline basis function of degree 5 compared with the local representation for a uniform GB-spline function of degree 5 defined using trigonometric knot functions. 38

5.1 Various levels of degree elevation on a degree 2 GB-spline basis function from a Bernstein-like basis formed using the trigonometric functions $\cos\left(\frac{\pi}{2}t\right)$ and $\sin\left(\frac{\pi}{2}t\right)$ on the interval $[0,1]$. The smooth curve is the spline, and the polygonal curves are control meshes of sucessively higher degrees. 56

5.2 Successive degree elevations of each basis function in a Bernstein-like basis of degree 3 spanning trigonometric functions. 57

5.3 Various levels of degree elevation on a degree 4 GB-spline basis function from a Bernstein-like basis spanning trigonometric functions. 58

5.4 Various levels of degree elevation on a circle first represented as a degree 2 GB-spline. Again, trigonometric knot functions are used, and the polygonal curves show the control meshes of successively higher degrees. While the previous curves had the parameter domain along the horizontal axis, this shows a 2-dimensional spline curve. 59

5.5 Various levels of degree elevation on a degree 4 2-dimensional GB-spline curve defined for a Bernstein-like basis spanning trigonometric functions. 60

5.6 A circle represented with respect to GB-spline bases with trigonometric knot functions and different end conditions. 61

5.7 A 2-dimensional GB-spline curve represented with respect to GB-spline bases with different end conditions. 62

5.8 Degree elevations of a one dimensional GB-spline curve of degree 3 formed by a basis spanning trigonometric functions over the knot vector $[0,0,0,0,.5,1,1,1,1]$. 63

5.9 Degree elevations of a two dimensional GB-spline curve of degree 2 formed by a basis spanning trigonometric functions over the knot vector $[0,0,0,0,.5,1,1,1,1]$. 64

5.10 Knot insertion at .25 and .75 on a degree 4 GB-spline curve defined over $[0,0,0,0,0,.5,1,1,1,1,1]$. 65

5.11 Knot insertion at .25 and .75 on a degree 2 GB-spline curve defined over $[0,0,0,0,0,.5,1,1,1,1,1]$. 66

5.12 Insertion of 3 knots at .5 and then 3 knots at .25 and again at .75 on a degree 3 GB-spline curve defined using a Bernstein-like basis that spans the trigonometric functions $\cos\left(\frac{\pi}{2}t\right)$ and $\sin\left(\frac{\pi}{2}t\right)$. 67

5.13 Insertion of 5 knots at .5 and then 5 knots at .25 and again at .75 on a degree 5 GB-spline curve defined using a Bernstein-like basis that spans the trigonometric functions $\cos\left(\frac{\pi}{2}t\right)$ and $\sin\left(\frac{\pi}{2}t\right)$. 68

5.14 Insertion of 2 knots at .5 and then 2 knots at .25 and again at .75 on a degree 2 GB-spline curve that exactly represents a quarter circle and is defined using a Bernstein-like basis that spans the trigonometric functions $\cos\left(\frac{\pi}{2}t\right)$ and $\sin\left(\frac{\pi}{2}t\right)$. 69

5.15 Insertion of 5 knots at .5 and then 5 knots at .25 and again at .75 on a degree 5 GB-spline curve defined using a Bernstein-like basis that spans the trigonometric functions $\cos\left(\frac{\pi}{2}t\right)$ and $\sin\left(\frac{\pi}{2}t\right)$. 70

5.16 Insertion of 4 knots at .5 and then 4 knots at .25 and again at .75 on a degree 4 GB-spline curve defined using a Bernstein-like basis that spans the trigonometric functions $\cos\left(\frac{\pi}{2}t\right)$ and $\sin\left(\frac{\pi}{2}t\right)$. 71

6.1 The sizes of the transcendental terms and polynomial terms for the generalized Bernstein basis of degrees 12 and 13 with $u = \cos\left(\frac{\pi}{2}t\right)$ and $v = \sin\left(\frac{\pi}{2}t\right)$. The transcendental parts are shown in blue and the polynomial parts are shown in red. 81

6.2 Breakdown in computation of trigonometric basis functions for $p = 16$. . . . 81

6.3 Generalized Bernstein bases of degree 26 with $u = \cos\left(\frac{\pi}{2}t\right)$ and $v = \sin\left(\frac{\pi}{2}t\right)$ computed using the tails of the Taylor series for sin and cos. 82

Chapter 1. Introduction

Generalized B-splines (GB-splines) are a class of splines that have received greater attention in research in recent years. In 1999 Ksasov and Sattayatham [1] demonstrated a variety of the properties of GB-splines. In 2005 Costantini et al. [2] studied generalized Bernstein bases of this form in greater detail. In 2008, Wang et al. [3] introduced unified extended splines (UE-splines), a subclass of GB-splines and demonstrated that this new class of splines contains several other classes of generalized splines. In 2011, Manni et al. [4] proposed that GB-splines be used for isogeometric analysis. In [5], Romani successfully applied the techniques from [6] to form subdivision methods allow for the approximation of UE-splines via a limit of control meshes successively refined by a non-stationary subdivision scheme. Our purpose here is to provide direct methods for the evaluation and refinement of GB-splines.

In the latter part of the twentieth century B-splines have seen widespread use in Computer Aided Design (CAD). GB-splines are a promising generalization of B-splines that provide workarounds for some of the fundamental problems that arise in CAD when using surfaces defined by polynomial and rational functions. Rather than spanning the spaces of piecewise polynomials spanned by traditional B-spline curves, on each interval $[t_i, t_{i+1})$ in the given knot vector T, they span the spaces $\left\{1, t, \ldots, t^{p-2}, u_i^{[p-1]}, v_i^{[p-1]}\right\}$ where $u_i^{[p-1]}$ and $v_i^{[p-1]}$ are $p-1$'th integrals of arbitrary functions forming a Chebyshev space over $[t_i, t_{i+1}]$. Because of their ability to span more general classes of functions, GB-splines provide a way to exactly model geometric shapes like circles using control point representations that are intuitive and natural to designers. They are, however, defined in terms of a recursive integral process that makes them more difficult to use in computation. The primary purpose of this thesis is to present algorithms that make computation and design with GB-splines significantly easier in practice.

Section 1.1 will provide a basic introduction to the mathematical notation used in this thesis. Appendix A will provide the precise notation used in the algorithms in greater detail.

Chapter 2 is intended to be a concise and rigorous introduction to Bernstein polynomials.

Simple algorithms for a variety of common operations on Bezier curves are presented. B-splines are also introduced and several of their properties are presented. De Boor [7], Farin [8], Piegl and Tiller [9], and Schumaker [10] all provide a more lengthy introduction to these topics. A good short survey is also provided by Farouki [11].

Chapter 3 is dedicated primarily to the definition of GB-splines, and a discussion of their properties. Some relevant proofs are also presented. Here, the local structures used to perform evaluation and refinement will also be introduced.

Chapter 4 presents general algorithms to construct piecewise representations for a GB-spline basis. The primary driving algorithm there is Algorithm 9 which constructs the piecewise representation for a GB-spline basis. The central purpose of this chapter is to provide detailed descriptions of practical and efficient algorithms that can be used for the evaluation of GB-spline bases.

Chapter 5 presents a general algorithm for representing a given spline curve in terms of a GB-spline basis that contains the given curve in its span. Algorithm 17 provides a method that can be used for both degree elevation, knot insertion, and the modification of end conditions for a given spline curve. This algorithm is effective for both polynomial splines and for more general GB-spline curves. The approach in Chapter 5 of projecting to and from local representations of spline curves can be used to perform more general projections between spline bases. This technique is demonstrated for B-spline curves in [12].

Chapter 6 discusses various numerical concerns and discusses how instabilities can be safely avoided. The generality of the algorithms presented earlier on requires a discussion of when the computation of GB-spline bases may be ill conditioned. Workarounds are suggested for cases in which the algorithms presented here may be less accurate.

Chapter 7 provides a short discussion of future research that can build off of the approaches used here.

Appendix B provides Python routines that perform the algorithms documented in Chapters 4 and 5 as well as some useful auxiliary routines.

1.1 Notation

Throughout this document the following notation will be used.

$u^{(n)}$	The n'th derivative of the function u
$u^{[n]}$	The n'th indefinite integral of the function u (where $u^{[n]}(0) = \cdots = u^{[1]}(0) = 0$)
$u^{\{n\}}$	The Taylor series of $u^{[n]}$
χ_A	The indicator function of the set A

Table 1.1: Notational conventions.

The notation for algorithms is outlined more particularly in Appendix A. Roughly speaking, the notation follows the conventions used in NumPy. Numpy-style array slicing notation will be used (see Appendix A). All arithmetic operations on arrays should be considered elementwise unless otherwise noted. Matrix multiplication will be written by simply writing the variables next to one another. 0-based indexing for arrays is assumed throughout.

Broadcasting semantics for applying operations along various axes of different arrays are used extensively in the algorithms in this thesis. They are described in greater detail in Appendix A. Broadcasting of operations on arrays is particularly important to the material here because it provides a simple and concise way to express operations along different axes of an array without obscuring the primary meaning of an expression with lengthy subscripts and many nested loops.

Much of the syntax for the algorithms outlined in the appendix is borrowed from Python. In the face of any doubts, the reader is invited to consult the exact code for each algorithm provided in Appendix B.

Chapter 2. Background

The Bernstein polynomials were first introduced by Bernstein in [13] as a method for approximation by polynomials. They were used as an alternative method of proof for the Weierstrass Approximation Theorem, but saw little use otherwise. In the late 1950's and early 1960's, De Casteljau and Bezier both began work on using Bernstein polynomials for

computer aided design. De Casteljau and Bézier were both mathematicians working for car companies. Bézier worked at Renault and De Casteljau worked at Citroën. De Casteljau's work preceded that of Bézier, but, due to Citroën's more restrictive policies about publication of original research, his work was not widely known until several years later. Because of this, curves for CAD formed using Bernstein polynomials are called Bézier curves. The primary algorithm used to evaluate Bézier curves, however, is still known as De Casteljau's algorithm. A more complete discussion of the origins of Bernstein polynomials and Bézier curves is given in [11] as well as the first chapter of [8].

B-splines also have a long history. They were first introduced by Schoenberg for smoothing data in 1946 [14]. A brief discussion of the origins of B-splines can be found in the introduction of the chapter on B-splines in Farin's book on splines [8].

The purpose of this chapter is to provide a concise introduction to the properties of Bernstein polynomials and B-splines. Many of the properties here extend to GB-splines as well. This chapter also serves as a stand-alone reference showing many of the basic proofs and algorithms for working with Bernstein polynomials.

2.1 Bernstein Polynomials

Definition 2.1. For any degree p and for $i = 0, 1, \dots, p$, B_i^p, the i'th Bernstein basis polynomial of degree p, is defined as

$$\binom{p}{i} (1-t)^{p-i} t^i$$

A polynomial is said to be in Bernstein form if it is represented as a linear combination of Bernstein polynomials of a given degree. The coefficients for Bernstein basis function used in this linear combination are called the Bernstein coefficients.

Definition 2.2. We will refer to the usual polynomial basis $1, x, \dots, x^p$ as the power basis.

Figure 2.1: The Bernstein basis polynomials of degrees 2, 3, 4, and 5.

Though the definition of the Bernstein polynomials already provides a simple means for evaluation of polynomials in Bernstein form, they are seldom computed using the original definition. It is faster and more numerically stable to compute them using a system of successive averages of the Bernstein coefficients. This averaging process is shown in Algorithm 1. It is commonly known as the De Casteljau algorithm. The De Casteljau algorithm is illustrated for polynomials in Bernstein form in Figure 2.3.

Algorithm 1 The De Casteljau algorithm for evaluating polynomials in Bernstein form

procedure DECASTELJAU(a, t)
  ▷ Evaluate polynomial with coefficients a at parameter value t.
  $p = \deg(a)$
  for $i = 0$, $i < p$ **do**
    $a = (1 - t) * a[: p - i - 1] + t * a[1 :]$
  end for
  return $a[0]$
end procedure

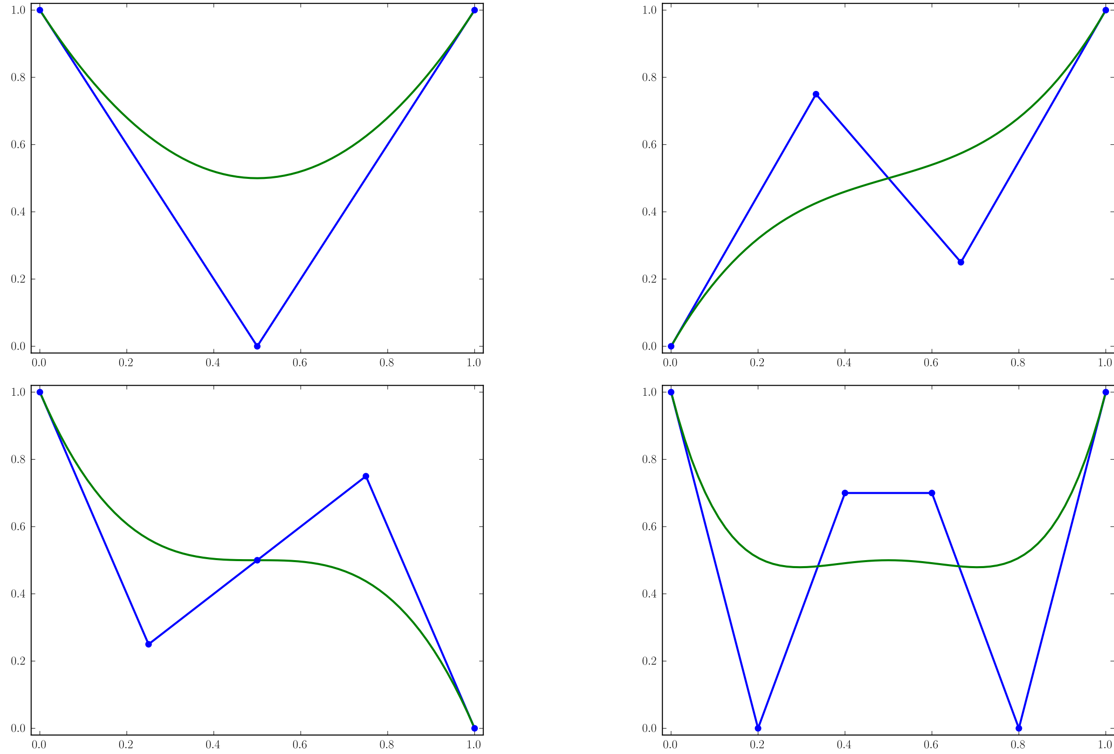

Figure 2.2: Polynomials in Bernstein form of degrees 2, 3, 4, and 5 (green) shown with their Bernstein coefficients (blue).

Theorem 2.3. *The De Casteljau algorithm for evaluating polynomials in Bernstein form is correct.*

Proof. Notice that

$$\binom{p}{i} = \binom{p-1}{i-1} + \binom{p-1}{i}.$$

Applying this to the definition of the Bernstein polynomials, we see that

$$\begin{aligned}
B_i^p(t) &= \binom{p}{i} t^i (1-t)^{p-i} \\
&= \left(\binom{p-1}{i-1} + \binom{p-1}{i} \right) t^i (1-t)^{p-i} \\
&= t \binom{p-1}{i-1} (1-t)^{p-1-(i-1)} t^{i-1} + (1-t) \binom{p-1}{i} (1-t)^{p-1-i} t^{i-1} \\
&= t B_{i-1}^{p-1}(t) + (1-t) B_i^{p-1}(t)
\end{aligned}$$

Consider some parameter value t. For a given Bernstein polynomial a with coefficients a_i,

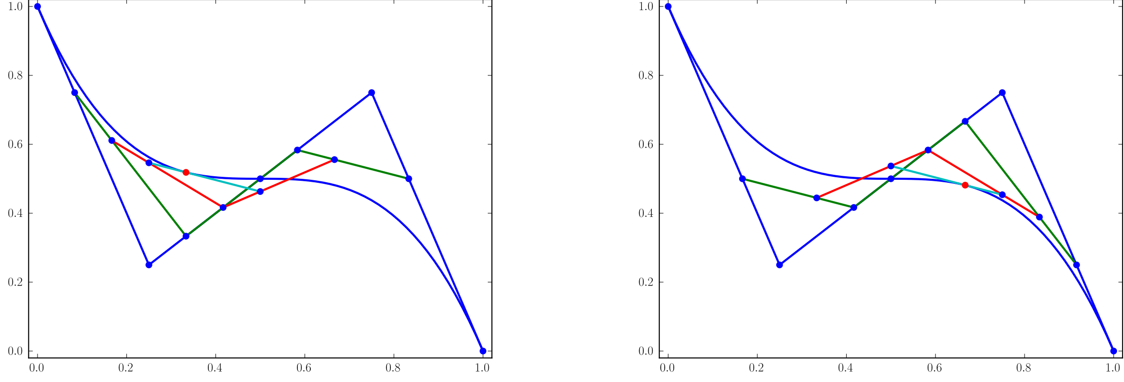

Figure 2.3: An illustration of the De Casteljau algorithm (Algorithm 1) for evaluating polynomials in Bernstein form. Here, parameter values $\frac{1}{3}$ and $\frac{2}{3}$ are used.

define $b_i^p = a_i$ for $i = 0, \ldots, p$, and $b_i^{q-1} = (1-t)\, b_i^q + t b_{i+1}^q$ for $0 \leq q < p$ and $0 \leq i \leq p - q$. Notice that, in Algorithm 1, b_i^q is the i'th entry of the array a after $p - q$ steps of the for loop. For brevity in notation, we assume all other b_i^q are 0 and $B_i^q = 0$ whenever $i < 0$ or $i > q$. Now notice that

$$
\begin{aligned}
a(t) &= \sum_{i=0}^{p} b_i^p B_i^p(t) \\
&= \sum_{i=0}^{p} \left(t b_i^p B_{i-1}^{p-1}(t) + (1-t)\, b_i^p B_i^{p-1}(t) \right) \\
&= \sum_{i=0}^{p-1} \left((1-t)\, b_i^p + t b_{i+1}^p \right) B_i^{p-1}(t) \\
&= \sum_{i=0}^{p-1} b_i^{p-1} B_i^{p-1}(t) \\
&\vdots \\
&= b_0^0 B_0^0(t) \\
&= b_0^0.
\end{aligned}
$$

□

Remark 2.4. The Bernstein basis is symmetric across the line $x = \frac{1}{2}$. In other words, $B_i^p(1-t) = B_{p-i}^p(t)$. This implies that a polynomial in Bernstein form can be reflected

across the line $x = \frac{1}{2}$ by reversing its coefficients.

Definition 2.5. A finite set of continuous functions over an interval is said to be a *partition of unity* if, at any point in that interval, the functions all sum to 1.

Theorem 2.6. *The Bernstein polynomials of a given degree form a partition of unity.*

Proof. Consider any degree $p \geq 0$. Note that, by the binomial theorem

$$1 = 1^p = (t + (1-t))^p = \sum_{i=0}^{p} \binom{p}{i} (1-t)^{p-i} t^i = \sum_{i=0}^{p} B_i^p(t).$$

$\square$

Remark 2.7. Each Bernstein basis polynomial takes values in $(0, 1)$ on $(0, 1)$. This implies that each basis function is nonnegative.

Remark 2.8. For any $p \geq 0$ and $0 < i < p$, $B_0^p(0) = B_p^p(1) = 1$ and $B_0^p(1) = B_i^p(0) = B_i^p(1) = B_p^p(0) = 0$.

Identities also exist for differentiation and integration of Bernstein polynomials

Theorem 2.9. *The derivative of B_i^p is $p\left(B_{i-1}^{p-1} - B_i^{p-1}\right)$ with $B_{-1}^p = B_{p+1}^p$ understood as 0.*

Proof. Assuming $i \notin \{0, p\}$,

$$\begin{aligned}
(B_i^p)'(t) &= \frac{p!}{i!(p-i)!} i (1-t)^{p-i} t^{i-1} - \frac{p!}{i!\,(p-i)!} (p-i)(1-t)^{p-i-1} t^i \\
&= \frac{(p-1)}{(i-1)!\,(p-1-(i-1))!} p (1-t)^{p-1-(i-1)} t^{i-1} - \frac{(p-1)!}{(i)!\,(p-1-i)!} p (1-t)^{p-1-i} t^i \\
&= p\left(\binom{p-1}{i-1} (1-t)^{p-1-(i-1)} t^{i-1} - \binom{p-1}{i} (1-t)^{p-1-i} t^i\right) \\
&= p\left(B_{i-1}^{p-1}(t) - B_{i-1}^p(t)\right).
\end{aligned}$$

For the case that $i \in \{0, p\}$, the proof is the same, but with the zero terms omitted. $\square$

Corollary 2.10. *Given a polynomial a with Bernstein coefficients a_i (where $\ldots, a_{-2}, a_{-1}$, and $a_{p+1}, a_{p+2}, \ldots$ are all considered to be 0). The n'th derivative of a is:*

$$a^{(n)} = \frac{p!}{(p-n)!} \sum_{i=0}^{p-n} \left(\sum_{j=0}^{n} \binom{n}{j} (-1)^{n-j} \, a_{i-j} \right) B_i^{p-n}.$$

In particular

$$a' = p \sum_{i=0}^{p-1} (a_{i-1} - a_i) \, B_i^{p-1}.$$

Theorem 2.11. *Each B_i^p has a unique local maximum of $\frac{i^i (p-i)^{p-i}}{p^p} \binom{p}{i}$ on the interval $[0, 1]$ at $\frac{i}{n}$.*

Proof. The derivative of B_i^p is $p \left(B_{i-1}^{p-1} - B_i^{p-1} \right)$. Evaluating this expression directly at $\frac{i}{n}$ we see

$$\begin{aligned} (B_i^p)' \left(\frac{i}{n} \right) &= p \left(\frac{(p-1)!}{(p-i)! \, (i-1)!} \left(\frac{p-i}{p} \right)^{p-i} \left(\frac{i}{p} \right)^{i-1} - \frac{(p-1)!}{(p-i-1)! i!} \left(\frac{p-i}{p} \right)^{p-i-1} \left(\frac{i}{p} \right)^{i-1} \right) \\ &= 0. \end{aligned}$$

Furthermore, B_{i-1}^{p-1} and B_i^{p-1} must have another $p-2$ factors that are either t or $1-t$, so the only possible maximum or minimum on $[0, 1]$ is at $\frac{i}{n}$. By Remarks 2.7 and 2.8, this point is a maximum on $[0, 1]$, so it is the unique maximum. Direct evaluation gives the desired value. □

Theorem 2.12. *The integral of B_i^p is*

$$\frac{1}{p+1} \sum_{j=i+1}^{p+1} B_j^{p+1}$$

and consequently,

$$\int_0^1 B_i^p = \frac{1}{p+1}.$$

Proof. Note that

$$\left(\frac{1}{p+1}\sum_{j=i+1}^{p+1} B_j^{p+1}\right)' = \sum_{j=i+1}^{p+1}\left(B_{j-1}^{p} - B_j^{p}\right) = B_i^{p}.$$

This means that this is a valid indefinite integral for the given Bernstein basis function. Since this expression is 0 at 0, this is exactly equal to the indefinite integral when it is taken with respect to the power basis. Now for the definite integral, note that, given the values of each basis function at 0 and 1,

$$\int_0^1 B_i^p(t)\,dt = \frac{1}{p+1}\sum_{j=i+1}^{p+1} B_j^{p+1}(1) - \frac{1}{p+1}\sum_{j=i+1}^{p+1} B_j^{p+1}(0) = \frac{1}{p+1}.$$

□

Corollary 2.13. *For a polynomial a with Bernstein coefficients a_i,*

$$\begin{aligned}\int a(t)\,dt &= \frac{1}{p+1}\sum_{i=1}^{p+1} a_i \sum_{j=i+1}^{p+1} B_j^{p+1}(t)\\ &= \sum_{i=1}^{p+1}\left(\frac{1}{p+1}\sum_{j=0}^{i-1} a_j\right) B_i^{p+1}(t).\end{aligned}$$

Multiplication of polynomials in Bernstein form can be performed by transforming to the basis $(1-t)^{p-i}t^i$, expanding in terms of $(1-t)$ and t, and then transforming back to the Bernstein basis of the appropriate degree (see [15]). This process is shown in Algorithm 2.

Algorithm 2 Multiplication of two polynomials in Bernstein form

---

procedure BernsteinMultiply(a, b)
 ▷ a and b are arrays containing the Bernstein coefficients for two polynomials.
 $c = (a * \text{bin}(\deg a)) \circledast (b * \text{bin}(\deg b))$
 return $c/\text{bin}(\deg a + \deg b)$
end procedure

Using a similar process as for multiplication, the following is also clear:

Remark 2.14. A Bernstein polynomial of order p is divisible by t if the first term in its Bernstein representation is 0 and divisible by $1-t$ if the last term in its Bernstein

representation is 0. This motivates Algorithm 4 for dividing a polynomial by t or $1 - t$.

Algorithm 3 Factor out a root at the left or right endpoint of a Bernstein polynomial.

---

procedure FactorLeft(a)
  ▷ a is an array containing the coefficients of a given Bernstein polynomial.
  $p = \deg a$
  return $(a * \text{bin}(p))[1:]/\text{bin}(p-1)$
end procedure
procedure FactorRight(a)
  ▷ a is an array containing the coefficients of a given Bernstein polynomial.
  $p = \deg a$
  return $(a * \text{bin}(p))[:p-1]/\text{bin}(p-1)$
end procedure

For polynomials of relatively small degree, the following algorithm can be used to perform polynomial division in the Bernstein basis. This algorithm works the same way as the

---

Algorithm 4 Factor out a root at the left or right endpoint of a Bernstein polynomial.

---

procedure BernsteinDivide(a, b)
  ▷ Perform polynomial division of a by b.
  $q, r = \text{dcv}(a * \text{bin}(\text{len}(a) - 1), b * \text{bin}(\text{len}(b) - 1))$
  $q \mathrel{/}= \text{bin}(\text{len}(q) - 1)$
  $r \mathrel{/}= \text{bin}(\text{len}(a) - 1)$
  return q, r
end procedure

---

multiplication algorithm. It changes the polynomial to a scaled version of the basis, performs polynomial division, then scales back to the Bernstein basis. One key difference between this algorithm and standard polynomial division is that the remainder r has the same degree as a. All but the last $\text{len}(b) - 1$ entries of r should always be close to 0. In practice, this algorithm is not always numerically stable. A more robust approach for a root between 0 and 1 is to subdivide the curve at the computed root. The process for subdividing a Bernstein polynomial into two separate Bernstein polynomials defined on the subintervals that lie on either side of the point of subdivision is shown in Algorithm 8. That approach has been applied successfully to develop root finding algorithms for Bernstein polynomials

in [16]. More stable methods for Bernstein polynomial division have also been developed ([17], [18], [19]).

Bernstein polynomials can also be represented as Bernstein polynomials of higher degree.

Theorem 2.15. *A polynomial in Bernstein form can be represented as a Bernstein polynomial of higher degree, in particular,*

$$\sum_{i=0}^{p} a_i B_i^p = \sum_{i=0}^{p+1} \left(\frac{i}{p+1} + \frac{p+1-i}{p+1} \right) a_i B_i^{p+1}$$

with $a_{-1} = a_{p+1} = 0$.

Proof. Note that

$$\frac{\binom{p}{i-1}}{\binom{p+1}{i}} = \frac{i}{p+1},$$

and

$$\frac{\binom{p}{i}}{\binom{p+1}{i}} = \frac{p+1-i}{p+1}.$$

So, letting a_{-1} and a_{p+1} take arbitrary finite values, we have

$$\begin{aligned}
\sum_{i=0}^{p} a_i B_i^p(t) &= ((1-t)+t) \sum_{i=0}^{p} a_i \binom{p}{i} (1-t)^{p-i} t^i \\
&= \sum_{i=0}^{p} a_i \binom{p}{i} \left((1-t)^{p+1-i} t^i + (1-t)^{p+1-(i+1)} t^{i+1} \right) \\
&= \sum_{i=1}^{p+1} a_{i-1} \frac{\binom{p}{i-1}}{\binom{p+1}{i}} B_i^{p+1}(t) + \sum_{i=0}^{p} a_i \frac{\binom{p}{i}}{\binom{p+1}{i}} B_i^{p+1}(t) \\
&= \sum_{i=0}^{p+1} \left(\frac{i}{p+1} a_{i-1} + \frac{p+1-i}{p+1} a_i \right) B_i^{p+1}(t).
\end{aligned}$$

□

Algorithm 5 shows one way to implement Theorem 2.15. The process of representing a Bernstein polynomial (or any sort of spline curve) as a Bernstein polynomial (or spline) of higher degree is known as degree elevation.

Algorithm 5 Degree elevation of a Bernstein polynomial

```
procedure DegreeElevate(a)
    ▷ a is an array containing the Bernstein coefficients for a polynomial in Bernstein form.
    p = deg(a)
    b = an empty array of length p + 1
    b(0) = 1
    for i = 1, i < p do
        b[i] = 1/(p+1) a[i − 1] + (p+1−i)/(p+1) a[i]
    end for
    b[p] = a[p − 1]
end procedure
```

Remark 2.16. Addition and subtraction of Bernstein polynomials of different degree can be performed by degree elevating the polynomial of lower degree until both polynomials have the same degree and then adding (or subtracting elementwise).

Algorithm 6 shows how to convert a polynomial from Bernstein form to power basis form.

Algorithm 6 Conversion of a polynomial in power basis form to Bernstein form.

```
procedure BernsteinToPowerBasis(a)
    ▷ Overwrite a with the polynomial coefficients in ascending order.
    p = deg(a)
    for i = 1, i < p do
        a[i :] −= a[i − 1 : p − 1]
    end for
    a *= bin(p)
end procedure
```

Theorem 2.17. *Algorithm 6 is a valid algorithm for converting from Bernstein form to power form.*

Proof. Consider a polynomial a in Bernstein form with coefficients a_i. Since the terms can be expanded, this polynomial can be represented in terms of the power basis with coefficients b_i. Since these two expressions are equal, their derivatives at 0 must be equal. Taking the i'th derivative of the power basis representation at 0, we see the i'th derivative is equal to $i!b_i$. Theorem 2.9 shows that the i'th derivative at 0 is just the i'th entry of the array a of Bernstein coefficients at the i'th step in the for-loop in Algorithm 6 times $p, p-1, \ldots, p-i+1$.

Since each term is left in the i'th entry of the array a, once the loop has finished,

$$a^{(i)} = \frac{p!}{(p-i)!} a[i] = i! b_i.$$

This implies that

$$b_i = \binom{p}{i} a[i].$$

□

Algorithm 7 shows how to convert a polynomial to Bernstein form.

Algorithm 7 Conversion of a polynomial from power basis form to Bernstein form

```
procedure PowerBasisToBernstein(a)
    ▷ a is an array of the polynomial coefficients in ascending order.
    ▷ Overwrite a with the coefficients for the Bernstein form.
    p = deg(a)
    a /= bin(p)
    for i = 1, i < p do
        a[i :] += a[i − 1 : p − 1]
    end for
end procedure
```

Theorem 2.18. *Algorithm 7 is a valid algorithm for converting from power form to Bernstein form.*

Proof. This amounts to showing that the two algorithms are inverses of one another. The power basis coefficients are multiplied by the binomial coefficients at the end of Algorithm 6 and divided by them at the beginning of Algorithm 7, so it is sufficient to prove that the loops in the algorithms are the inverses of one another. Given an array a of the Bernstein coefficients define a_j^i to be the j'th entry of the array a after i iterations of the loop in Algorithm 6. In this form the input values in a are $a_0^0, a_1^0, \ldots, a_p^0$ and the resulting values can be found in $a_0^0, a_1^1, \ldots a_p^p$. It is also true that for $i \leq j$, $i > 0$, $a_j^i = a_j^{i-1} - a_{j-1}^{i-1}$. This equality certainly also implies that $a_j^{i-1} = a_j^i + a_{j-1}^{i-1}$. This recursion has the form of the main loop in

Algorithm 7, so, given an initial array b containing $a_0^0, \ldots, a_p^p$, after the loop is finished, the resulting array contains $a_0^0, \ldots, a_p^p$, as desired. □

Remark 2.19. By Theorem 2.18, Algorithm 7 is correct for any polynomial represented in the power basis. Since the power basis is a linearly independent set of functions and basis conversion is a linear operator with an inverse (since Algorithm 6 is also correct), the Bernstein basis functions are all linearly independent.

Arithmetic operations with Bernstein polynomials are well studied. See [15] and [11] for more details.

Remark 2.20. Evaluation of Bernstein polynomials inside the domain $[0, 1]$ is known to be a very stable operation ([20], [21]). The algorithms for conversion to and from the power basis (Algorithms 6 and 7) are not numerically well-behaved ([22]), nor is the short algorithm for polynomial division (Algorithm 4).

Remark 2.21. Bernstein polynomials can be represented over intervals other than $[0, 1]$. The parameter values are just scaled linearly so that the new interval is mapped to $[0, 1]$ before the De Casteljau algorithm is applied. In finite element analysis, the convention of representing Bernstein polynomials over the interval $[-1, 1]$ is common.

The fact that the Bernstein basis is linearly independent implies that, at any given parameter value t (even outside $[0, 1]$) we may subdivide a polynomial a into two separate polynomials a_1 and a_2 such that if a_1 is represented in the Bernstein basis from 0 to t and a_2 is represented in the Bernstein basis from t to 1, then $a(t) = a_1(t)$ whenever t is between 0 and t and $a(t) = a_2(t)$ whenever t is between t and 1. Algorithm 8 shows how to compute the new control points the new curves a_1 and a_2. Theorem 2.22 shows the correctness of Algorithm 8.

Theorem 2.22. *Algorithm 8 is a valid algorithm for subdividing a polynomial in Bernstein form.*

Algorithm 8 Subdivision of a Bernstein polynomial

```
procedure SUBDIVIDE(a, t)
    ▷ a is an array with the Bernstein representation of a polynomial.
    ▷ t is the parameter value where the subdivision should take place.
    p = deg (a)
    a_1 = empty array of same shape and type as a
    a_2 = empty array of same shape and type as a
    a = (1 − t) ∗ a [: p − 1] + t ∗ a [1 :]
    a_1 [0] = a [0]
    a_2 [p − 1] = a [p − 1]
    for i = 1, i < p do
        a = (1 − t) ∗ a [: p − 1] + t ∗ a [1 :]
        a_1 [i] = a [i]
        a_3 [p − i − 1] = a [p − i − 1]
    end for
    return a_1, a_2
end procedure
```

Proof. This algorithm can be viewed as a way of storing intermediate values in the De Casteljau algorithm. It is discussed in greater detail in [8]. It is also shown in a different context as a method for representing Bézier curves as subdivision surfaces in [6]. □

2.2 Bézier Curves

Definition 2.23. Given a set of points $a_0, \dots, a_p$ in $\mathbb{R}^n$, the Bézier curve defined by those points is

$$a(t) = \sum_{i=0}^{p} a_i B_i^p (t)$$

The points $a_0, \dots a_p$ are called the control points of a.

Remark 2.24. Though a Bézier curve defines a curve for all parameter values, generally its domain is restricted to $[0, 1]$. Only some of the properties of Bézier curves extend to parameter values outside their primary interval of definition. For example, outside the interval $[0, 1]$, basis functions need not take values in $[0, 1]$ or even be positive. Algorithms for things like evaluation and subdivision will be correct, but their stability may degrade outside the primary interval of interest.

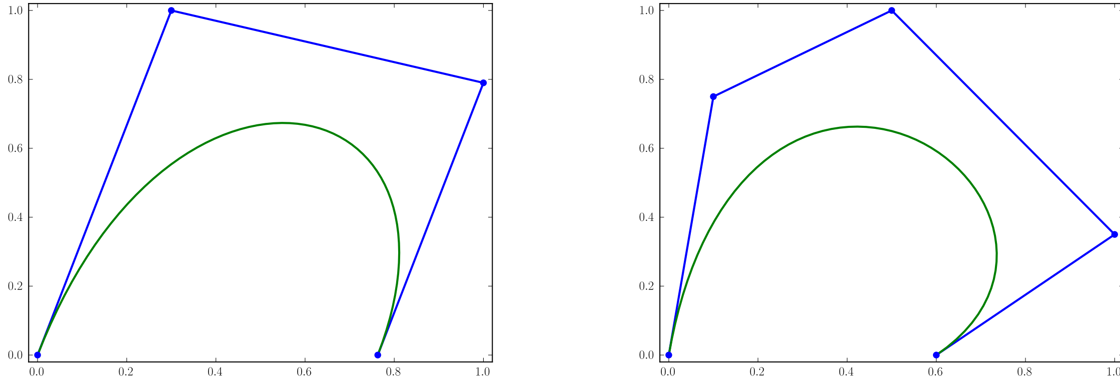

Figure 2.4: Bézier curves of degrees 3 and 4.

Remark 2.25. Many different types of curves can be represented in terms of basis functions and control points as in Definition 2.23. Bézier curves are defined to use the Bernstein basis, but other types of curves can be defined using different bases with similar properties. Functions used to represent a curve in this way are called blending functions. When referring to a curve, we will assume it is represented in terms of control points and blending functions from a given basis.

Definition 2.26. The control polygon of a Bézier curve is defined to be the piecewise linear interpolant of the control points.

Definition 2.27. A curve is variation diminishing if any line intersecting the curve intersects the curve's control polygon at least as many times as it intersects the curve.

Definition 2.28. A curve is said to have the convex hull property if it only takes values that lie in the convex hull of its control points (i.e., each point on the curve is a convex combination of the control points).

Definition 2.29. A set of basis functions is affine invariant or coordinate system independent if, for an affine transformation T and any set of control points a_i,

$$T\left(\sum_{i=0}^{n} a_i B_i^p(t)\right) = \sum_{i=0}^{n} T(a_i)\, B_i^p(t)\,.$$

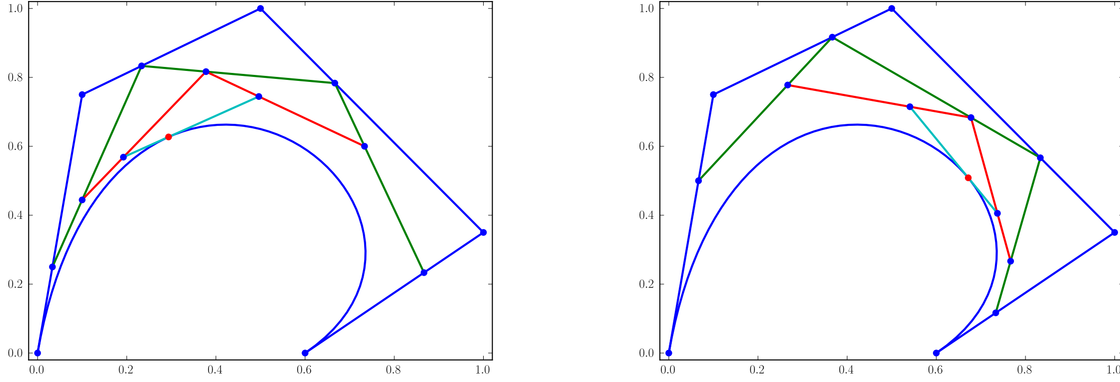

Figure 2.5: The De Casteljau algorithm (Algorithm 1) applied to a Bézier curve at parameter values $\frac{1}{3}$ and $\frac{2}{3}$.

In other words, a curve is coordinate system independent if linear transformations and translations of the curve can be performed on the entire curve by applying the given transformation to the curve's control points.

Theorem 2.30. *A curve is coordinate system independent if and only if its blending functions form a partition of unity.*

Proof. Here we follow the proof in [23]. Let T be an affine transformation with corresponding matrix representation $T(x) = Ax + b$. Then, for a curve with control points a_i in the domain of T defined in terms of a given set of basis functions, the following is true if and only if the basis functions form a partition of unity

$$
\begin{aligned}
T\left(\sum_{i=0}^{n} a_i B_i^p(t)\right) &= \sum_{i=0}^{n} B_i^p(t)\, A a_i + b \\
&= \sum_{i=0}^{n} B_i^p(t)\, A a_i + \sum_{i=0}^{n} B_i^p(t)\, b \\
&= \sum_{i=0}^{n} T(a_i)\, B_i^p(t)
\end{aligned}
$$

□

Theorem 2.31. *Any Bézier curve $a(t)$ of degree p with control points $a_0, \ldots, a_p$ has the following properties:*

(i) $a(t)$ interpolates its endpoints, i.e. $a(0) = a_0$ and $a(1) = a_p$,

(ii) $a(t)$ is variation diminishing,

(iii) $a(t)$ lies entirely within the convex hull of its control points,

(iv) $a(t)$ is coordinate system independent,

(v) $a'(t)$ and $\int a(t)dt$ can be obtained the same way as for Bernstein polynomials except that arithmetic involving control points is understood to be elementwise,

(vi) $a(t)$ can be evaluated by applying the De Casteljau Algorithm (Algorithm 1) to its control points,

(vii) $a(t)$ can be subdivided at any given parameter value by applying Algorithm 8 to its control points elementwise,

(viii) $a(t)$ can be represented as a Bezier curve of higher degree with the new control polygon computed the same way the coefficients for degree elevation were computed for Bernstein polynomials,

(ix) Algorithms 6 and 7 can be used to convert between the Bézier representation of a curve and a polynomial parameterization of each coordinate of the curve,

(x) Reversing the control points of a gives a backwards parameterization of a,

Proof. (i) is an immediate consequence of the end-values of the Bernstein basis functions.

(ii) is a special case of Theorem 3.10.

(iii) follows since, by Theorem 2.6, every point on a Bézier curve is a convex combination of its control points.

(iv) is an immediate consequence of Theorem 2.30.

(v) - (x) are direct analogues of some of the properties of Bernstein polynomials and are a result of elementwise arithmetic.

□

2.3 B-splines

B-splines are an important generalization of Bézier curves. B-splines have many of the same properties as Bézier curves, but they also address some key shortcomings. B-spline theory allows the creation of curves where each control point only has an effect on a small portion of the curve. It allows greater control of the parameter domain and of differing degrees of continuity at different parameter values. It also separates the number of control points from the degree of the curve to allow the definition of lower degree curves that have many control points.

In these definitions, we use the recursion introduced in [24] and [25] as the primary definition for B-splines. There was an earlier definition of these curves in terms of divided differences, but, for both simplicity and numerical stability, we will omit it here.

Definition 2.32. A knot vector is a nondecreasing vector of real numbers. Knot vectors are used to define the parameter domain for a B-spline. Each value in the knot vector is called a knot.

Definition 2.33. Given a specific knot vector $T = (t_0, \ldots, t_m)$ of length m, denote the i'th B-spline basis function B_i^p. (The difference between a B-spline basis function and a Bernstein basis function should always be clear from the context). Define (for $0 < i < m - 1$)

$$B_i^0(t) = \chi_{[t_i, t_{i+1})}(t)$$

and for $0 \leq i < m - 1 - p$ with $0 < p < m - 1$,

$$B_i^p(t) = \frac{t - t_i}{t_{i+p} - t_i} B_i^{p-1}(t) - \frac{t - t_{i+1}}{t_{i+p+1} - t_{i+1}} B_{i+1}^{p-1}(t)$$

p is called the degree of the B-spline. If either of the fractions in this definition have a 0 in the numerator, they should evaluate to 0, regardless of whether or not the denominator is 0. In addition, if $t_{m-p-1} = \cdots = t_{m-1}$ and $t_{m-p-2} \neq t_{m-p-1}$ (i.e., if the last p knots are

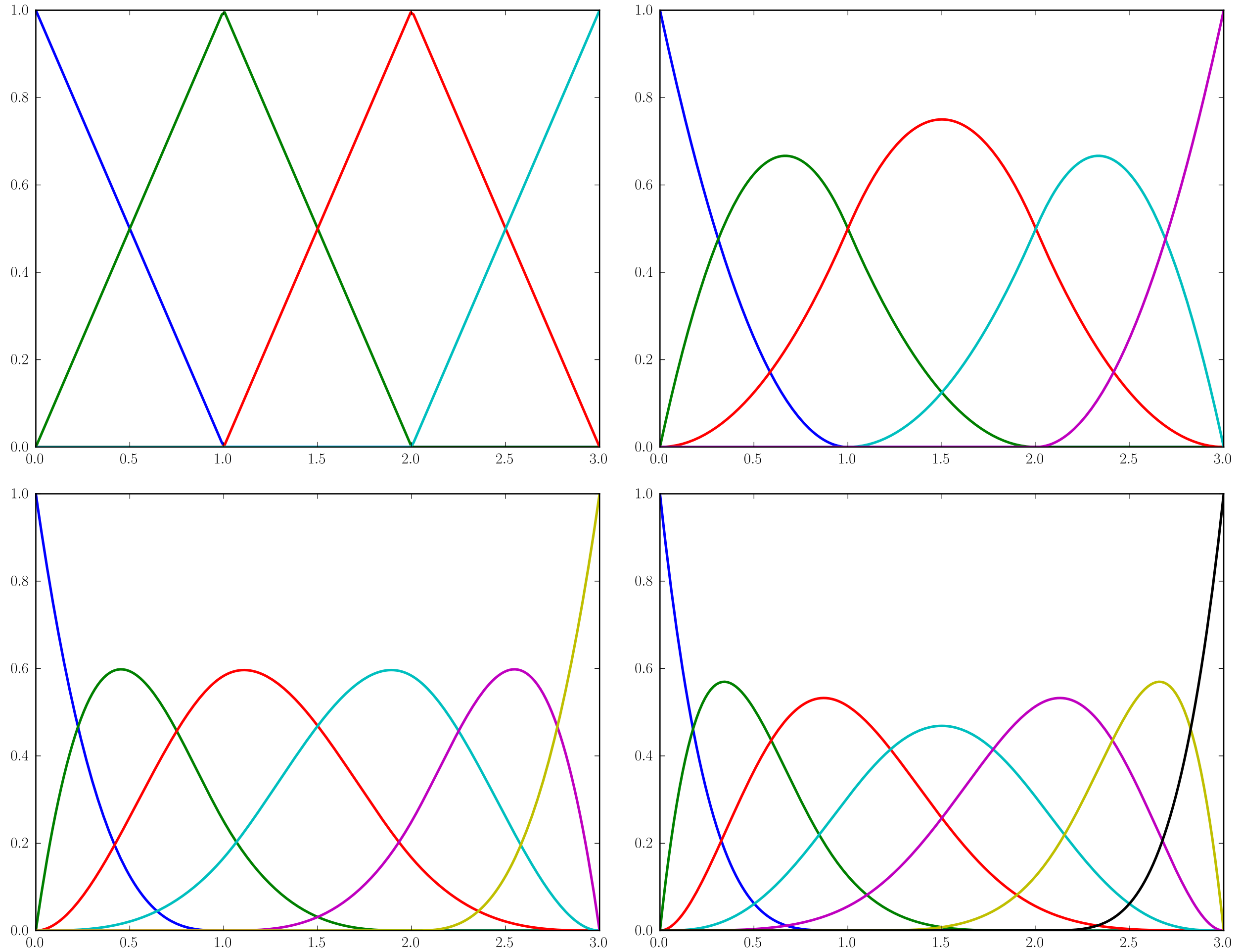

Figure 2.6: B-spline bases of degrees 1, 2, 3, and 4.

repeated, and the last basis function is nonzero), define $B^p_{m-p-2}(t_{m-1}) = 1$.

Definition 2.34. Given a spline degree p, the active region of a given knot vector T is the interval $[t_p, t_{m-p-1}]$.

Definition 2.35. Given a knot vector T of length m, a degree p, and $m - p - 1$ control points a_i, define the corresponding B-spline curve $a(t)$ as

$$a(t) = \sum_{i=0}^{m-p-2} a_i B^p_i(t)$$

for $t \in [t_p, t_{m-p-1}]$. $a(t)$ is not defined outside the active region of T. The knots $t_0, \ldots, t_{p-1}$ and $t_{m-p}, \ldots, t_{m-1}$ (those that can possibly lie outside the active region) are called end

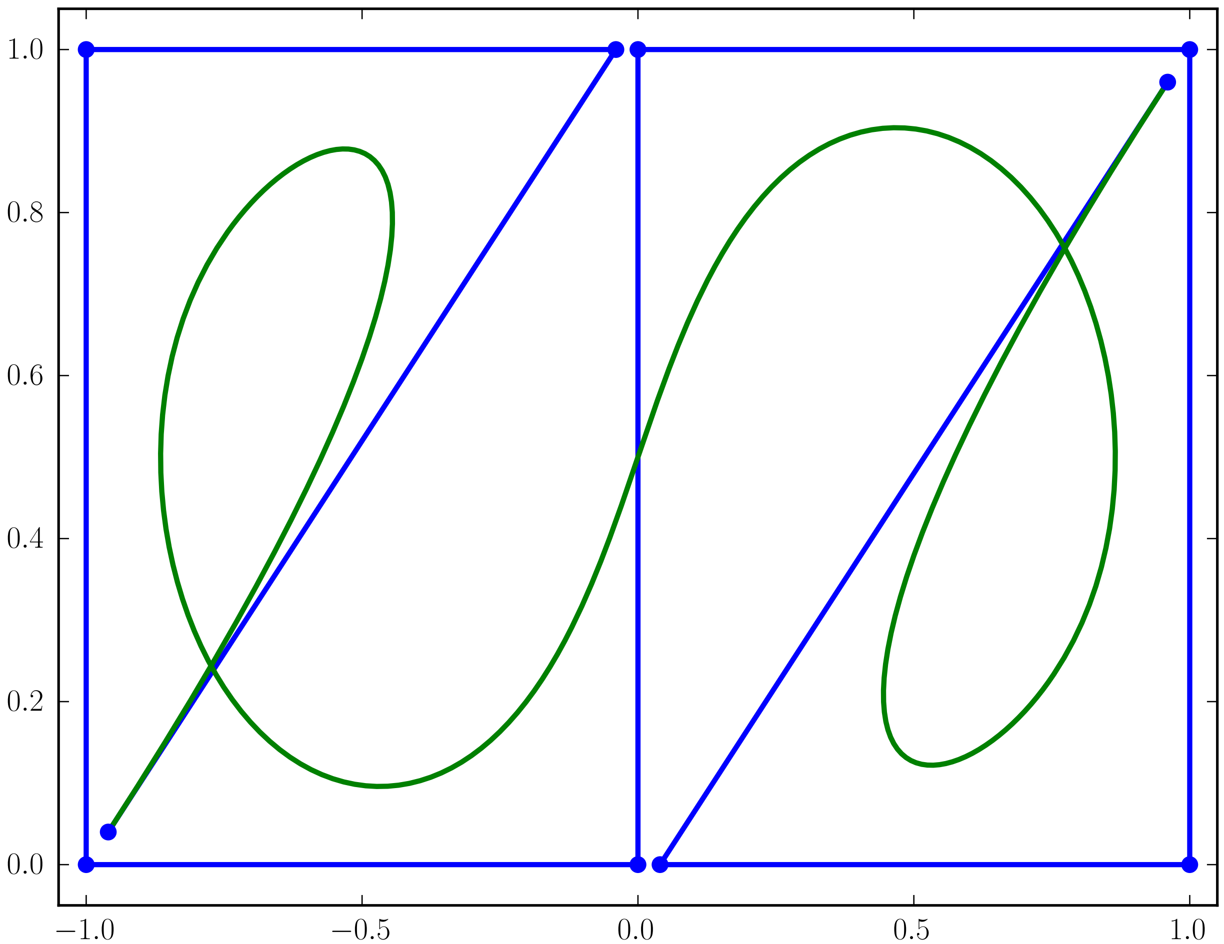

Figure 2.7: A degree 4 B-spline curve.

condition knots.

Remark 2.36. We will refer to the recursion in Definition 2.33 as the De Boor recursion.

Definition 2.37. Given a degree p, a knot vector T is said to be open if the first $p + 1$ knots are equal and the last $p+1$ knots are equal. This is equivalent to saying that the knot vector's span is equal to its active region.

Remark 2.38. In practice, it is common to say that a B-spline of degree p has order $p+1$. Throughout this thesis, degrees will be used.

Remark 2.39. The degree 0 basis functions are defined, loosely speaking as the indicator functions of the intervals between the knots in the knot vector, so for any given knot vector,

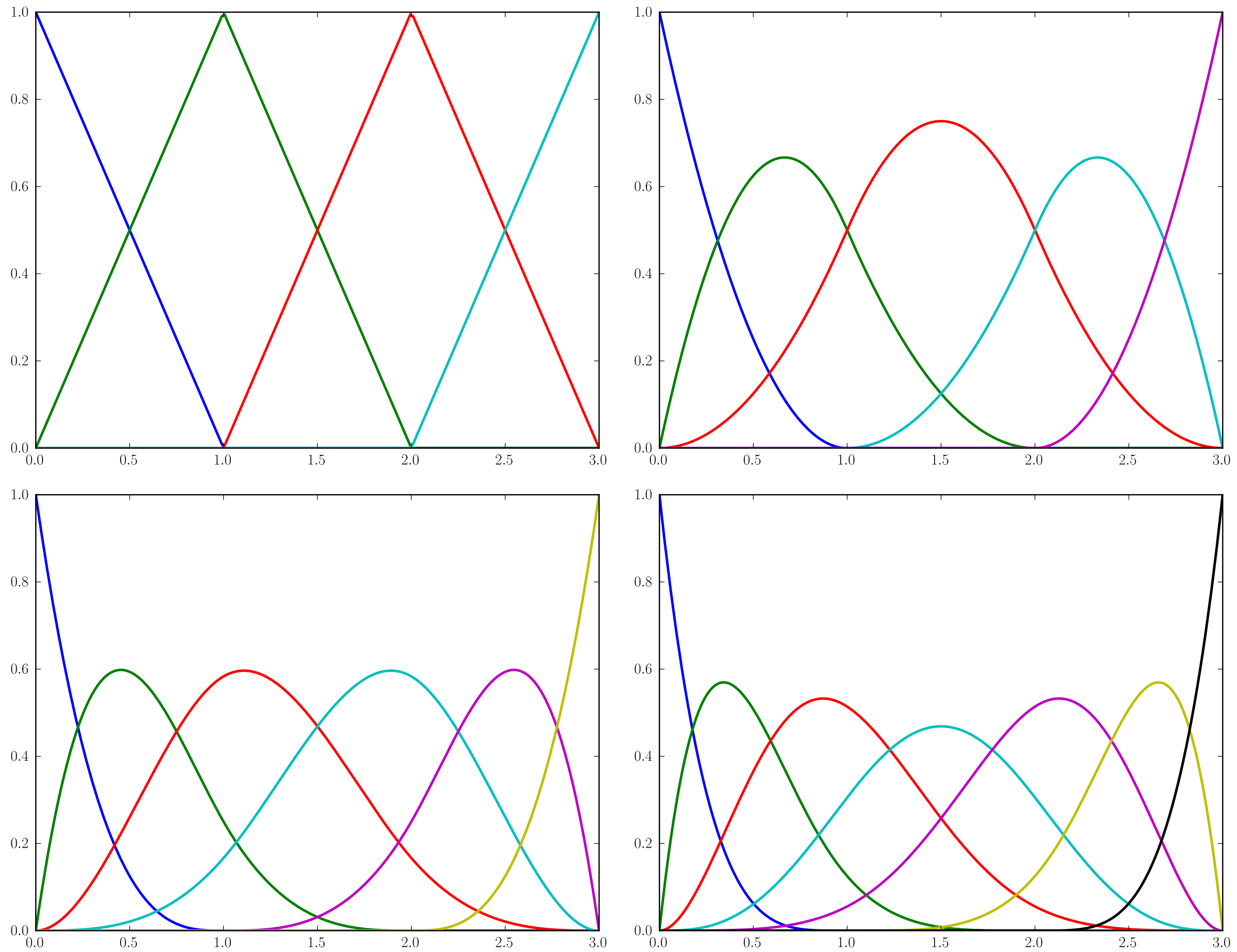

Figure 2.8: B-spline bases of degrees 1, 2, 3, and 4.

the maximum number of degree 0 basis functions have been defined. Furthermore, the basis functions B_i^p have been defined for all i and p for which B_i^{p-1} and B_{i+1}^{p-1} are also defined. In general, for a knot vector of length n, there are $n-p-1$ B-spline basis functions of degree p.

Remark 2.40. If $t_i < t_{i+1} < t_{i+2}$, the B-spline basis function B_i^1 is the unique piecewise linear function that is linear everywhere except at t_i, t_{i+1}, and t_{i+2}, takes a values of 0 at t_i, 1 at t_{i+1}, and 0 at t_{i+2}, and is 0 outside the interval (t_i, t_{i+2}). These are the same as the hat functions commonly used in finite element analysis.

Theorem 2.41. *Bézier curves are a special case of B-splines.*

Proof. Consider the Bernstein basis of degree p. We desire to show that this is a B-spline basis for a given degree. Consider the knot vector that consists of 0 repeated p times and then 1 repeated $p+1$ times. Then, the De Boor recursion can be rewritten as

$$B_i^p(t) = tB_i^{p-1}(t) + (1-t)\, B_{i+1}^{p-1}(t)\,.$$

This is the same recurrence as was used in the De Casteljau algorithm. Now noting Remark 2.40 the nonzero basis functions for this knot vector exactly coincide with the Bernstein basis polynomials of degree 1, so all nonzero basis functions are the same. Since the nonzero basis functions are the same for each degree up to and including p, the bases of degree p are certainly the same as well. □

Theorem 2.42. *B-splines have all of the following properties (where $a(t)$ is any given B-spline of degree p with knot vector T and control polygon a comprised of n control points):*

(i) *$a(t)$ interpolates its endpoints if T is open.*

(ii) *A B-spline basis of degree p over a knot vector T of length m forms a partition of unity over its active region.*

(iii) *$a(t)$ lies entirely within the convex hull of its control points.*

(iv) *$a(t)$ is variation diminishing.*

(v) *$a(t)$ is coordinate system independent.*

(vi) *$a(t)$ can be represented as a B-spline curve of higher degree over a new knot vector with the knots in T and one additional knot less than or equal to the first knot of T and one additional knot greater than or equal to the last knot of T.*

(vii) *Each B-spline basis function is supported on the interval $[T[i], T[i+p+1]]$. This means that each control point controls only a portion of the curve.*

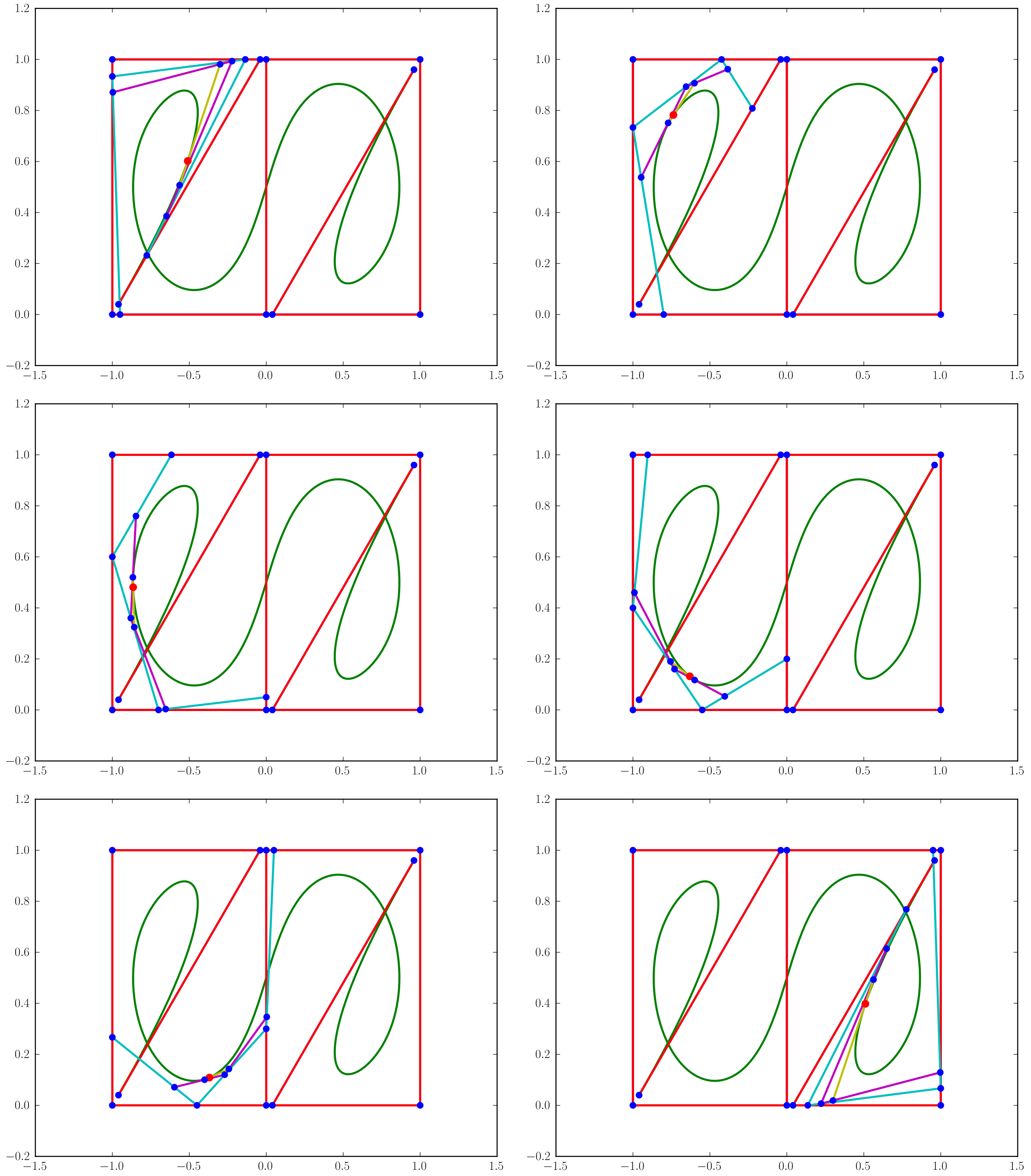

Figure 2.9: The geometric representation of the De Boor recurrence (given in Definition 2.33) shown here at parameter values 0.2, 0.8, 1.2, 1.8, 2.2, and 5.8.

(viii) $a(t)$ *is infinitely differentiable everywhere except at its knots, where it is* C^{p-j} *where* j *is the number of times the knot is present in the knot vector (with negative degrees of continuity taken to mean that no continuity is implied).*

(ix) *The* i*'th basis function of degree* p *is* 0 *everywhere if and only if* $T[i] = \cdots = T[i+p+1]$.

(x) *Each B-spline basis function is positive on the interior of its support.*

(xi) *All nonzero B-spline basis functions are linearly independent.*

(xii) $a(t)$ *can be represented as a degree* p *B-spline over a knot vector* T' *with the same active region where* T' *contains all the knots in* T*, in particular,* $a(t)$ *can be represented as a B-spline over a knot vector* T' *where* T' *is* T *with one additional knot (or one additional repetition of an existing knot) inside its active region. This operation is linear.*

(xiii) *The derivative of* $a(t)$ *is*

$$a'(t) = \sum_{i=0}^{n} p \frac{a_{i+1} - a_i}{t_{i+p} - t_i} B_i^{p-1}(t)$$

where the basis functions B_i^{p-1} *are over the knot vector formed by removing the first and last knot of* T.

(xiv) *The indefinite integral of* B_i^p *from* t_i *to a given value in the corresponding active region of a knot vector* T *is given in terms of the basis functions of degree* $p+1$ *defined on a knot vector* T' *formed by adding an end condition knot to each end of* T*. It is*

$$\int B_i^p(s)\, ds = \frac{t_{i+p+1} - t_i}{p+1} \sum_{j=i}^{n+1} B_j^{p+1}(t)$$

(xv) *The definite integral of* B_i^p *over its support is* $\frac{t_{i+p+1} - t_i}{p+1}$.

Proof. Many of these facts will be established as special cases of more general theorems in Chapter 3. (i) is a consequence of Theorem (3.11). (ii) and (v) follow from Theorems 3.9 and 2.30. (iii) and (iv) follow immediately form Theorem 3.10. (vi) and (xii) are a consequence of the algorithm presented in Chapter 5. (xi) also follows from Theorem 3.10. (vii) is a consequence of Theorem 3.8. (xiii) and (xiv) are given in [7]. (xv) follows immediately from (xiv) by adding knots to the ends of the knot vector T so that the support of B_i^p lies entirely in the active region and then applying (vii), (ii), and (xiv). □

Chapter 3. Definition of GB-splines

Generalized B-splines (GB-splines) were introduced in [3]. They are a more general class of spline designed to retain most of the desirable properties of B-splines while also spanning spaces of the form $\{1, t, \ldots, t^{p-2}, u(t), v(t)\}$ for more general types of functions u and v over each interval in a knot vector. They were introduced to unify theories of a variety of other generalized spline curves that have been introduced in recent years.

Definition 3.1. A set of p linearly independent functions are said to form a Chebyshev space over an interval I if any nonzero function in their span has at most $p-1$ roots in that interval.

Remark 3.2. Any two functions each with $p-1$ zeros in a Chebyshev space whose zeros coincide must be scalar multiples of one another.

Theorem 3.3 provides some intuition into what kinds of functions can be used to form a Chebyshev space.

Theorem 3.3. *Two monotonic functions on an interval* $[a, b]$*, one strictly increasing and the other strictly decreasing that are either both nonpositive or both nonnegative form a Chebyshev space over* $[a, b]$*.*

Proof. Without loss of generality say that u is decreasing and v is increasing. Taking

$$\tilde{u}(t) = v(b)\, u(t) - u(b)\, v(t)$$

$$\tilde{v}(t) = u(a)\, v(t) - v(a)\, u(t)$$

and then scaling both by a constant so both are nonnegative we may also, without loss of generality say that $u(b) = v(a) = 0$ and $u(a) = v(b) = 1$.

Consider a linear combination (with at least one nonzero coefficient) of u and v that is equal to 0 somewhere in $[a, b]$. If the zero lies at an endpoint, one of the coefficients must be 0, so by strict monotonicity, the zero is unique. If the zero lies in the interior of $[a, b]$, then, since u and v are both positive on (a, b), the coefficients in the linear combination must have different signs, so the resulting linear combination is strictly monotonic, so the zero is unique. □

Definition 3.4. Given a knot vector T, and functions u_i and v_i forming a Chebyshev space on each $[t_i, t_{i+1}]$ of nonzero length such that $u(0) = v(1) = 1$ and $v(0) = u(1) = 0$, we will refer to the sets of functions u_i and v_i as the knot functions over T.

Definition 3.5. Let T be a knot vector with corresponding knot functions u_i and v_i and a degree p. Define the i'th GB-spline basis function of degree p (N_i^p) as follows:

$$N_i^1(t) = \begin{cases} u_i(t) & t \in [t_i, t_{i+1}) \\ v_{i+1}(t) & t \in [t_{i+1}, t_{i+2}] \\ 0 & \text{otherwise} \end{cases},$$

$$\delta_i^p = \int_{t_i}^{t_{i+p+1}} N_i^p(s)\, ds,$$

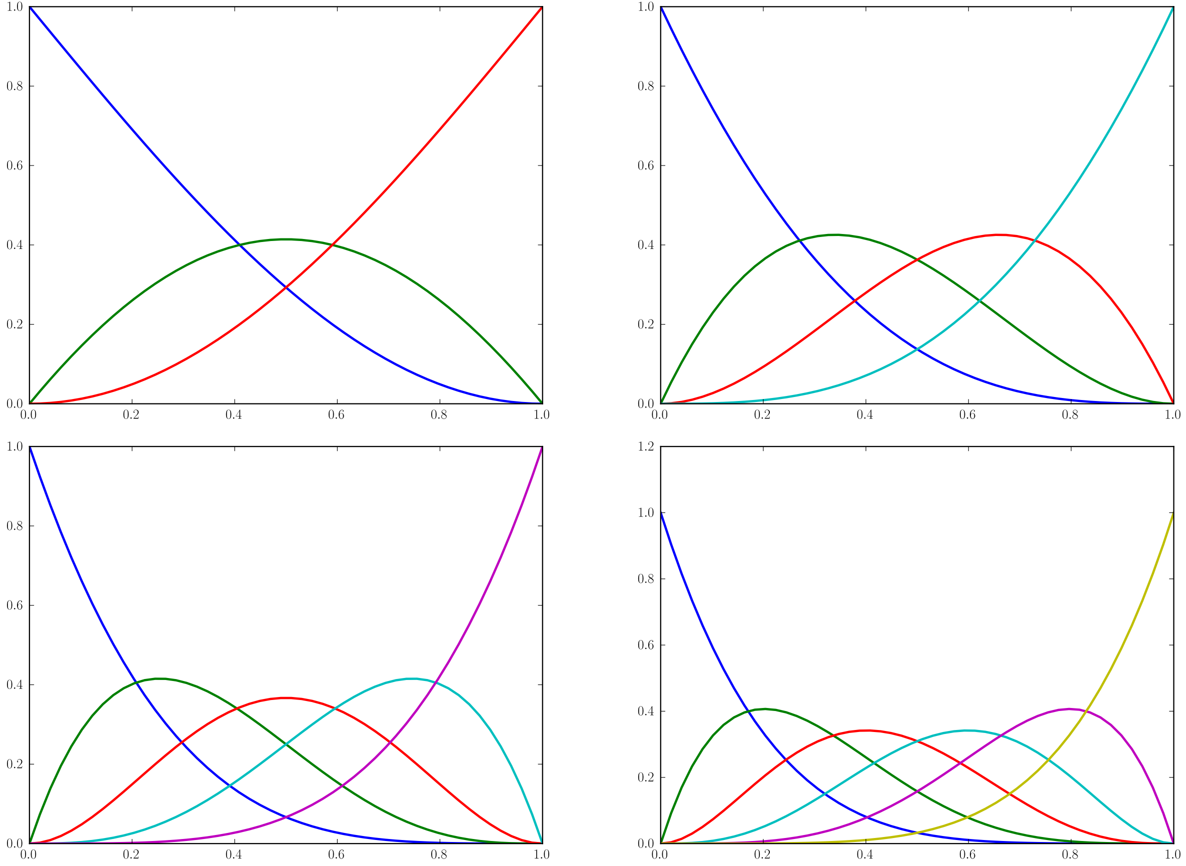

Figure 3.1: Bernstein-like GB-spline basis functions formed by using $\cos\left(\frac{\pi}{2}(t - t_i)\right)$ and $\sin\left(\frac{\pi}{2}(t - t_i)\right)$ as the knot functions.

and if $\delta_i^p = 0$ let

$$\Phi_i^p(t) = \begin{cases} 0 & t < t_{i+p+1} \\ 1 & t \geq t_{i+p+1} \end{cases}$$

and if $\delta_i^p \neq 0$

$$\Phi_i^p(t) = \frac{\int_{t_i}^{t} N_i^p(s)\, ds}{\delta_i^p}.$$

Now, for $p > 1$, define

$$N_i^p(t) = \Phi_i^{p-1}(t) - \Phi_{i+1}^{p-1}(t).$$

In addition, if $t_{m-p-1} = \cdots = t_{m-1}$ and $t_{m-p-2} \neq t_{m-p-1}$ (i.e. if the last p knots are repeated, and the last basis function is nonzero), define $B_{m-p-2}^p(t_{m-1}) = 1$.

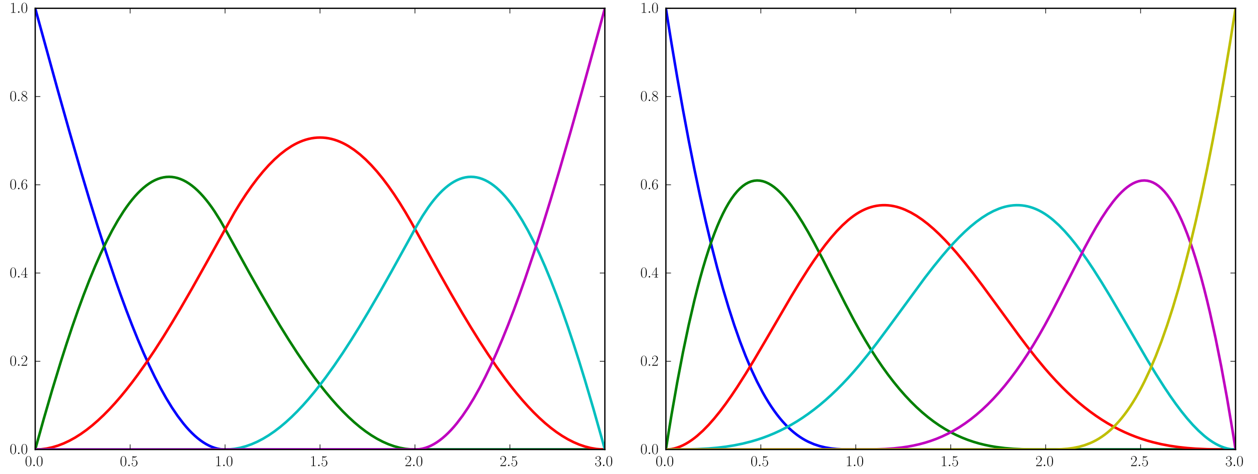

Figure 3.2: B-spline-like GB-spline bases of degree 2 and 3 formed by using $\cos\left(\frac{\pi}{2}(t-t_i)\right)$ and $\sin\left(\frac{\pi}{2}(t-t_i)\right)$ as the knot functions.

Definition 3.6. Given a knot vector T of length m with a corresponding set of knot functions, a degree $p > 1$, and $m-p-1$ control points a_i, define the corresponding GB-spline curve $a(t)$ as

$$a(t) = \sum_{i=0}^{m-p-2} a_i N_i^p(t)$$

for $t \in [t_p, t_{m-p-1}]$. $a(t)$ is not defined outside the active region of T.

Definition 3.7. Given a knot vector T with corresponding sets of knot functions u_i and v_i, define

$$V_i^p = \text{span}\left\{1, (t-t_i), \ldots, (t-t_i)^{p-2}, u^{[p-1]}(t), v^{[p-1]}(t)\right\}$$

Where $u^{[p-1]}$, and $v^{[p-1]}$ are the $(p-1)$'th indefinite integrals of u and v respectively.

Theorem 3.8. *Φ_i^p takes values of 0 for $t < t_i$ and 1 for $t \geq t_{i+p+1}$. Furthermore, N_i^p is zero outside the interval $[t_i, t_{i+p+1}]$.*

Proof. By definition, N_i^1 is 0 outside the interval $[t_i, t_{i+2}]$. If Φ_i^1 is a step function, it also satisfies the desired constraints. If Φ_i^1 is not a step function, then, because N_i^1 is 0 outside the interval $[t_i, t_{i+2}]$, $\Phi_i^1(t) = 0$ for $t < t_i$ and $\Phi_i^1(t) = \frac{\delta_i^1}{\delta_i^1} = 1$ for $t \geq t_{i+p+1}$. Now suppose for induction that Φ_i^p takes values of 0 for $t < t_i$ and 1 for $t \geq t_{i+p+1}$ and that N_i^p is zero

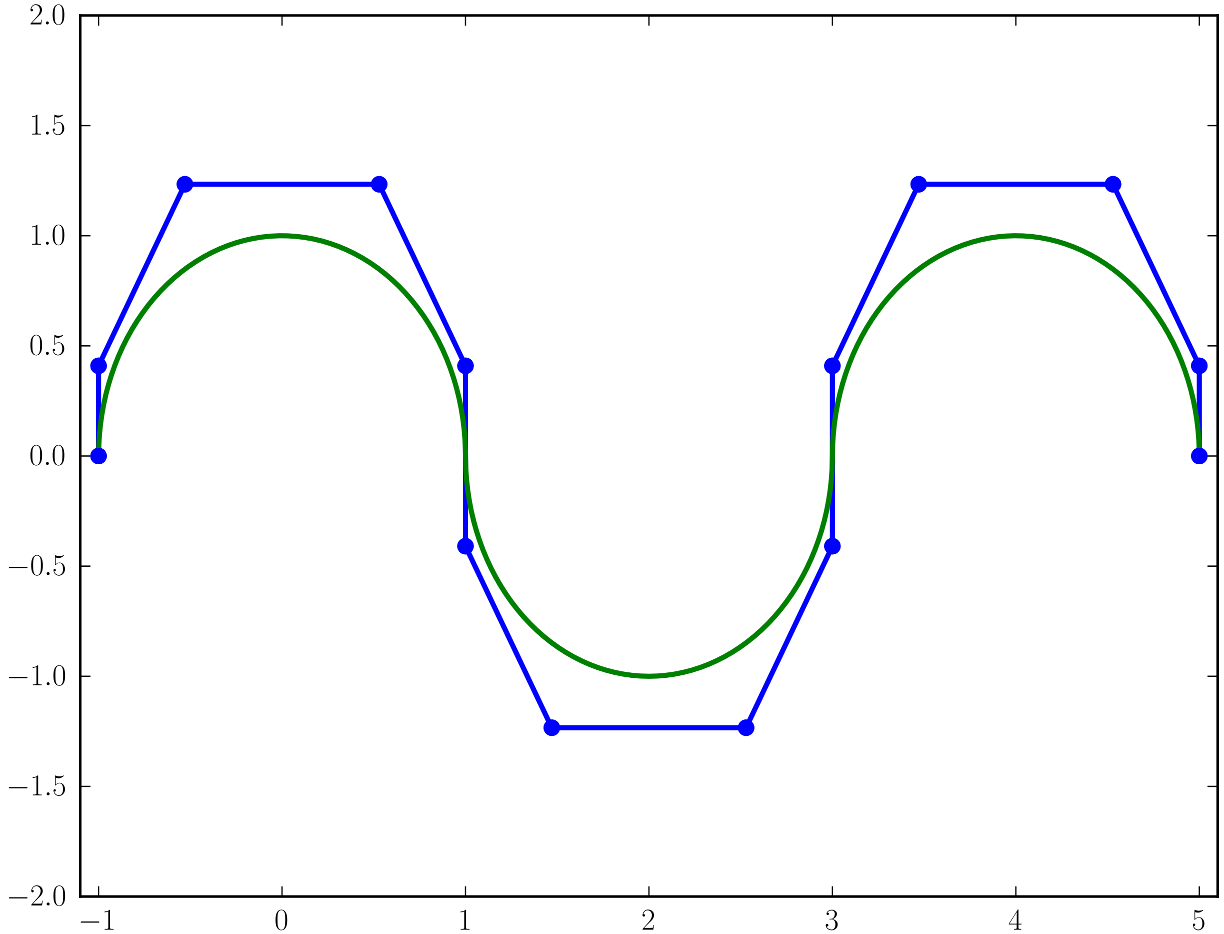

Figure 3.3: A GB-spline curve equal to different circles on different parts of its domain.

outside the interval $[t_i, t_{i+p+1}]$. This implies that, since $N_i^{p+1} = \Phi_i^p - \Phi_{i+1}^p$, it is 0 outside the set $[t_i, t_{i+p+1}] \cup [t_{i+1}, t_{i+p+2}] = [t_i, t_{i+p+2}]$. Again, if Φ_i^p is a step function, it satisfies the given constraints. If it is not, then, by the same argument as before, $\Phi_i^{p+1} = 0$ for $t < t_i$ and $\Phi_i^{p+1} = 1$ for $t \geq t_{i+p+2}$. □

Theorem 3.9. *The GB-spline basis sums to* 1 *inside the active region of a given knot vector.*

Proof. Let T be a knot vector. Consider a GB-spline basis of degree $p > 1$ over T with basis functions $N_0^p, \dots, N_n^p$. Then

$$\sum_{i=0}^{n} N_i^p(t) = \sum_{i=0}^{n} \Phi_i^{p-1}(t) - \Phi_{i+1}^{p-1}(t) = \Phi_0^{p-1}(t) - \Phi_{n+1}^{p-1}(t)$$

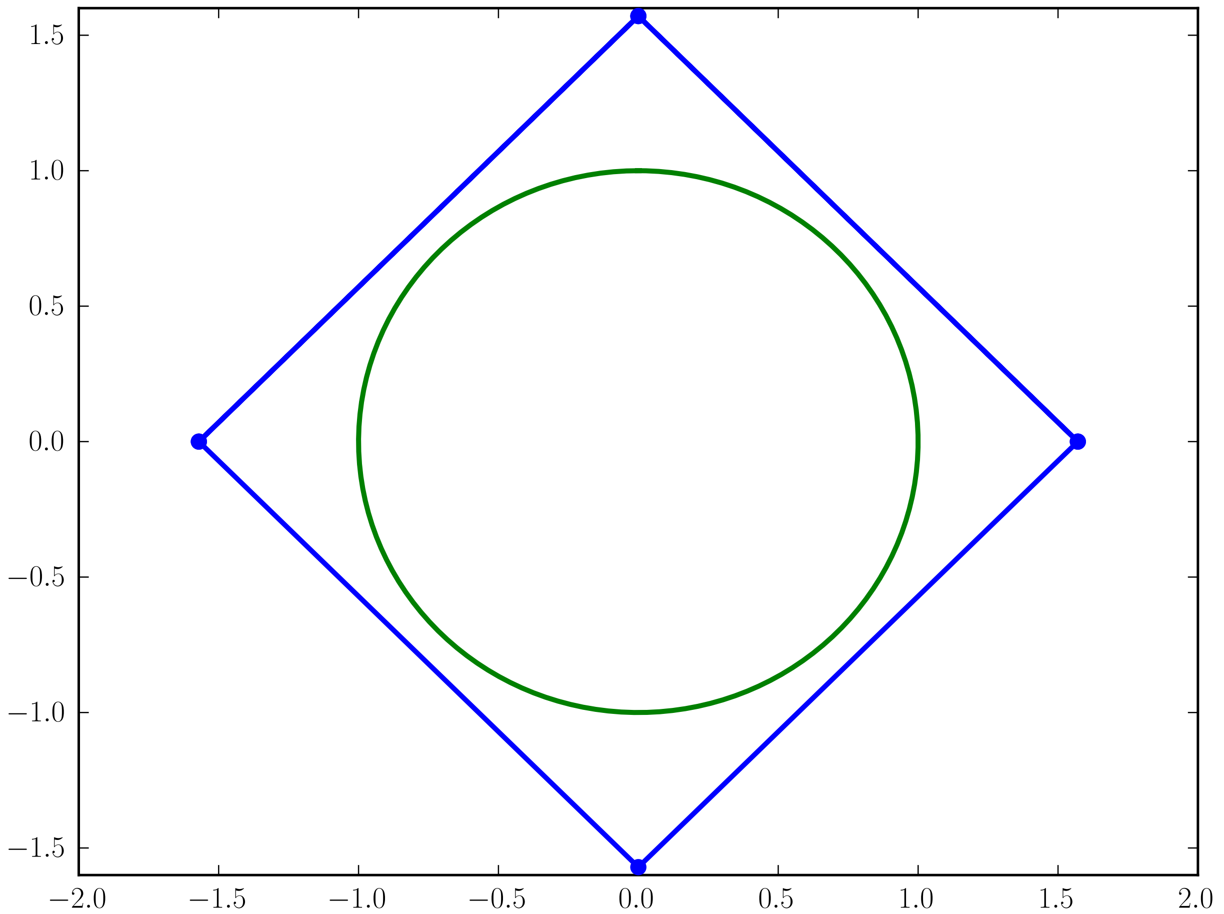

Figure 3.4: A GB-spline curve over a uniform knot vector that exactly represents a circle.

Now, by Theorem 3.8 Φ_0^{p-1} is 1 on the active region for degree p and Φ_{n+1}^{p-1} is 0, on the active region for degree p, this establishes the desired result. □

Theorem 3.10. *The nonzero GB-spline basis functions are linearly independent and positive on the interior of their supports. They span the space V_i^p on each interval inside the active region. GB-spline curves are also variation diminishing.*

Proof. These properties are demonstrated in [1] (see also [4]). In [1], GB-spline bases are defined for monotone knot functions, but the proofs there are also valid when the knot functions form a Chebyshev space. □

The primary advantages of GB-spline curves over traditional B-splines is that they allow for the exact representation of certain geometric curves and surfaces, like circles, hyperbolas,

spheres, and hyperboloids,that cannot be well-represented by polynomial splines. Figures 3.3 and 3.4 demonstrate this. See [3] for more examples.

Theorem 3.11. *A GB-spline over an open knot vector T with no degenerate basis functions in the corresponding spline basis interpolates its endpoints; in particular, if the spline is $a(t)$, the active region is the interval $[b, c]$, and the control points are $a_0, \ldots, a_n$, then $a(b) = a_0$ and $a(c) = a_n$.*

Proof. This is true by definition for $p = 1$, so say $p > 1$. Note that, since T is open with respect to the degree p of a, $t_0 = \cdots = t_p$, so Φ_0^{p-1} is an indicator function. Since N_0^p is not degenerate, this means that Φ_1^{p-1} is an integral term so it takes a value of 0 at b. Since $N_0^p = \Phi_0^{p-1} - \Phi_1^{p-1}$, $N_0^p(b) = 1$. Since the basis functions are nonnegative and sum to 1 on the active region, all the other basis functions are 0, so $a(b) = a_0$.

It has already been defined that $a(c) = a_n$, but we desire to show that the curve is continuous ☐

Theorem 3.12. *Given a GB-spline curve a of degree p over a knot vector T with knot functions u_i and v_i, all of the following are true:*

1. *The GB-spline basis restricted to each interval lies in V_i^p.*
2. *Each GB-spline is C^{p-k} at each of the knots in the knot interval where k is the number of times a knot is repeated.*
3. *Each GB-spline is at least C^{p+r-1} for each point t not in its knot vector, where r is the minimum continuity of the knot functions over the interval in the knot vector that contains t.*

Proof. For any $f \in V_i^{p-1}$, note that for

$$g(t) = \int_{t_i}^{t} f(s)\, ds$$

g must lie in V_i^p. Since each new basis function is a linear combination of functions of the same form as f, it must also lie in V_i^p, so the span of the basis functions lies in V_i^p. This completes the proof of 1.

3 is an immediate consequence of 1.

2 for the degree 1 basis functions follows from their definition.

We will first prove that a basis function N_i^p is discontinuous at a knot point t if and only if $t = t_i = \cdots = t_{i+p}$ or $t = t_{i+1} = \cdots = t_{i+p+1}$ but not both.

If $t_i = \cdots = t_{i+p+1}$, both N_i^{p-1} and N_{i+1}^{p-1} are identically 0, so $t_i = \cdots = t_{i+p+1}$, so N_i^p is also degenerate, so it is continuous. If neither equality holds, N_i^{p-1} nor N_{i+1}^{p-1} is identically 0, so N_i^p consists of two integral terms as in Definition 3.5, so it is continuous. This means that if N_i^p is discontinuous, either $t = t_i = \cdots = t_{i+p}$ or $t = t_{i+1} = \cdots = t_{i+p+1}$ but not both.

Conversely, say that $t = t_i = \cdots = t_{i+p}$ or $t_{i+1} = \ldots t_{i+p+1}$ but not both. Then exactly one of N_i^{p-1} and N_{i+1}^{p-1} is degenerate. This means that N_i^p consists of exactly one integral term as in Definition 3.5 and one step function, so it is discontinuous. In particular, it is discontinuous at the knot that is repeated $p-1$ times.

Since N_i^p is discontinuous at a knot point t_j if and only if $t_j = t_i = \cdots = t_{i+p}$ or $t_j = t_{i+1} = \cdots = t_{i+p+1}$ but not both, N_i^p is discontinuous at a knot point if and only if it is repeated $p+1$ times in the knots that define the support of N_i^p. Since each set of basis functions is defined in terms of integrals and step functions formed from the set of basis functions of next lowest degree, the degree of continuity increases by one at each knot point where no step function is defined. This means that, if a knot t is repeated n times, splines defined over the knot vector T of degree n will be C^0, splines of degree $n+1$ will be C^1, etc., as desired. □

Theorem 3.13. *Where it exists, the derivative of a GB-spline basis function is given by*

$$(N_i^p)'(t) = \frac{N_i^{p-1}(t)}{\delta_i^{p-1}} - \frac{N_{i+1}^{p-1}(t)}{\delta_{i+1}^{p-1}}.$$

Where it exists, the derivative of a GB-spline curve a with control points $a_0, \ldots, a_n$ is given

by

$$\sum_{i=0}^{n+1} a_i \frac{N_i^{p-1}(t)}{\delta_i^{p-1}} - a_{i+1} \frac{N_{i+1}^{p-1}(t)}{\delta_{i+1}^{p-1}}$$

with a_{-1} and a_{n+1} defined to be 0.

Proof. This follows immediately from the fundamental theorem of calculus applied to Definition 3.5. □

Theorem 3.14. *B-splines are GB-splines.*

Proof. This is shown in [3] for UE-splines, which are a subset of GB-splines defined by the same recurrence. □

Chapter 4. Evaluation of GB-splines

Though Definition 3.5 makes it easy to see why many of the properties of spline curves are true, it does not provide for a simple means of evaluation. As the definition stands, the only effective means of evaluation are either recursive numeric integration, and symbolic computation of indefinite integrals. Recursive numeric integration becomes very costly for all but the lowest degrees of splines. Symbolic computation is effective, though it can be unwieldy for numeric computation. In order to address this deficiency, we present a more direct method of computing values on these spline curves.

Given that each basis function lies in the space V_i^p, we may introduce a local representation of each basis function in terms of functions spanning the space V_i^p. Given that the recursive integrals must be computed, it would be ideal for these local representations to be more amenable to integration. This introduces a few possible choices for the local representations of the splines:

1. $u_i^{[p-1]}$, $v_i^{[p-1]}$, and an additional polynomial term of degree $p-2$

2. The remainder terms for the Taylor series centered at t_i of $u_i^{[p-1]}$, and $v_i^{[p-1]}$ with an additional polynomial term of degree $p-2$

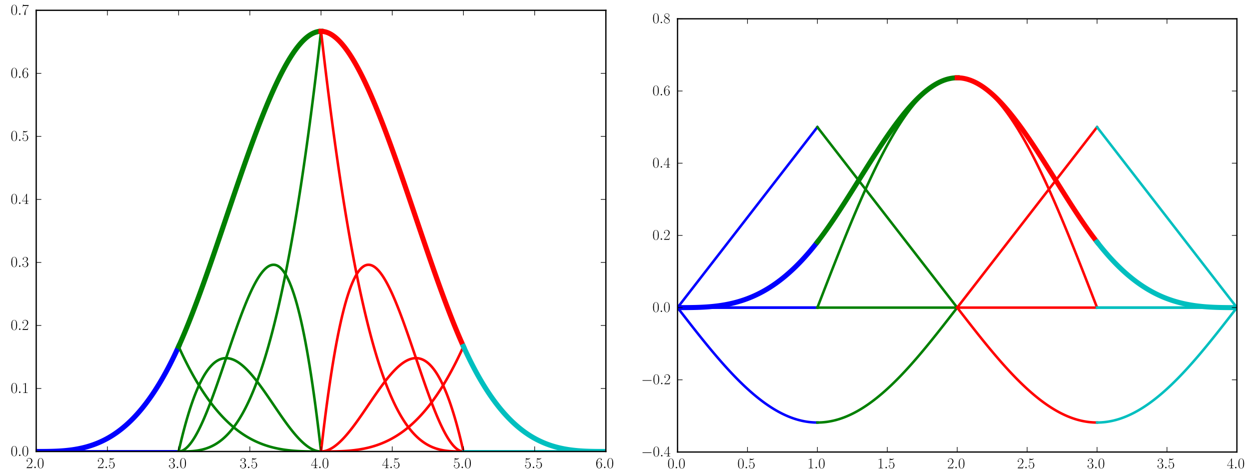

Figure 4.1: The local polynomial representation for a uniform B-spline basis function of degree 3 compared with the local representation for a uniform GB-spline function of degree 3 defined using trigonometric knot functions.

3. The remainder term for the Taylor series centered at t_i for v_i, the remainder term for the Taylor series centered at t_{i+1} for u_i and an additional polynomial term of degree $p-2$

By virtue of the linear independence of $u_i^{[p-1]}$ and $v_i^{[p-1]}$ from all other polynomial terms, each of these is a valid choice for a basis. Other combinations of the knot functions and the remainder terms for their Taylor series centered at either side of the interval $[t_i, t_{i+1}]$ will also provide usable bases, but we will confine our discussion here to the examples above. We will first consider the case without remainder terms for Taylor polynomials. Terms formed using Taylor series remainder terms will be discussed briefly in Chapter 6.

Local Representations: Knot Functions and Polynomials.

Definition 4.1. Since each basis function lies in the space V_i^p, we may say that the i'th basis function of degree p can be represented on the j'th interval in T as:

$$N_i^p(t) = P_{i,j}^p(t) + a_{i,j}^p u_j^{[p-1]}(t) + b_{i,j}^p v_j^{[p-1]}(t)$$

where $P_{i,j}^p$ is a polynomial term and $a_{i,j}^p$ and $b_{i,j}^p$ are constants. This means that the recurrence

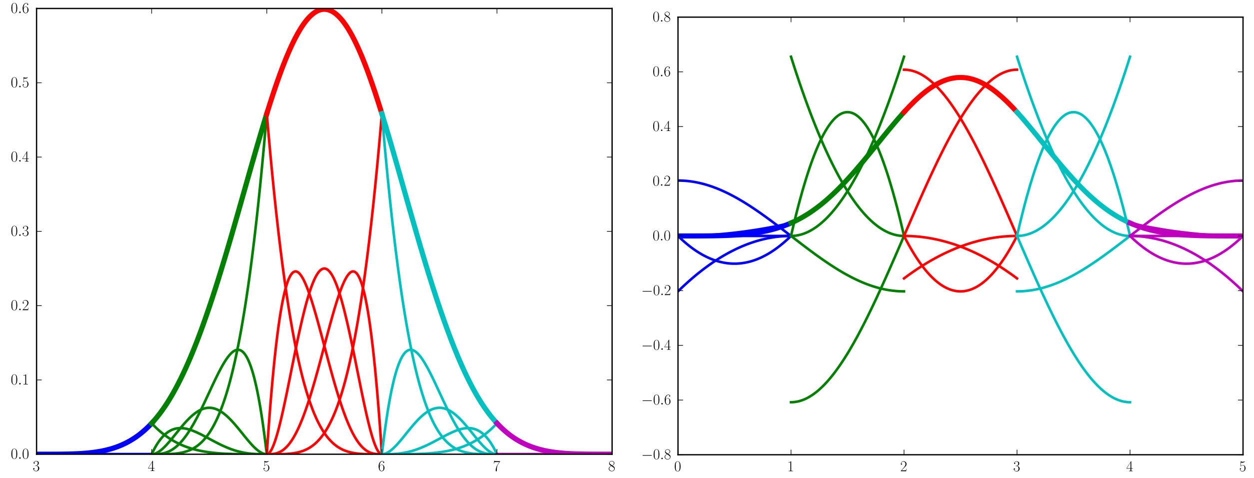

Figure 4.2: The local polynomial representation for a uniform B-spline basis function of degree 4 compared with the local representation for a uniform GB-spline function of degree 4 defined using trigonometric knot functions.

stated in Definition 3.5 can be written as

$$a_{i,j}^{p} = \frac{a_{i,j}^{p-1}}{\delta_i^{p-1}} - \frac{a_{i+1,j}^{p-1}}{\delta_{i+1}^{p-1}}$$

$$b_{i,j}^{p} = \frac{b_{i,j}^{p-1}}{\delta_i^{p-1}} - \frac{b_{i+1,j}^{p-1}}{\delta_{i+1}^{p-1}}$$

$$\begin{aligned} P_{i,j}^{p}(t) = N_i^p\left(t_j\right) & \\ &+ \frac{1}{\delta_i^{p-1}}\left(\int_{t_j}^{t} P_{i,j}^{p-1}(s)ds - a_{i,j}^{p-1}u_j^{[p-1]}(t_j) - b_{i,j}^{p-1}v_j^{[p-1]}(t_j)\right) \\ &- \frac{1}{\delta_{i+1}^{p-1}}\left(\int_{t_j}^{t} P_{i+1,j}^{p-1}(s)ds - a_{i+1,j}^{p-1}u_j^{[p-1]}(t_j) - b_{i+1,j}^{p-1}v_j^{[p-1]}(t_j)\right) \end{aligned}$$

With the additional stipulation that if N_i^{p-1} is identically zero, and the interval $[t_{i+p}, t_{i+p+1})$ is empty, that $P_{i,j}^{p-1}$ have an additional 1 added to it to account for the modified treatment of basis functions that are identically 0 in Definition 3.5. As before, we also require that, if the last basis function is discontinuous at the end of the active region, that it must take a value of 1 at the last point in the active region.

The recurrence relation outlined in Definition 4.1 is not as easy to implement as De Boor's

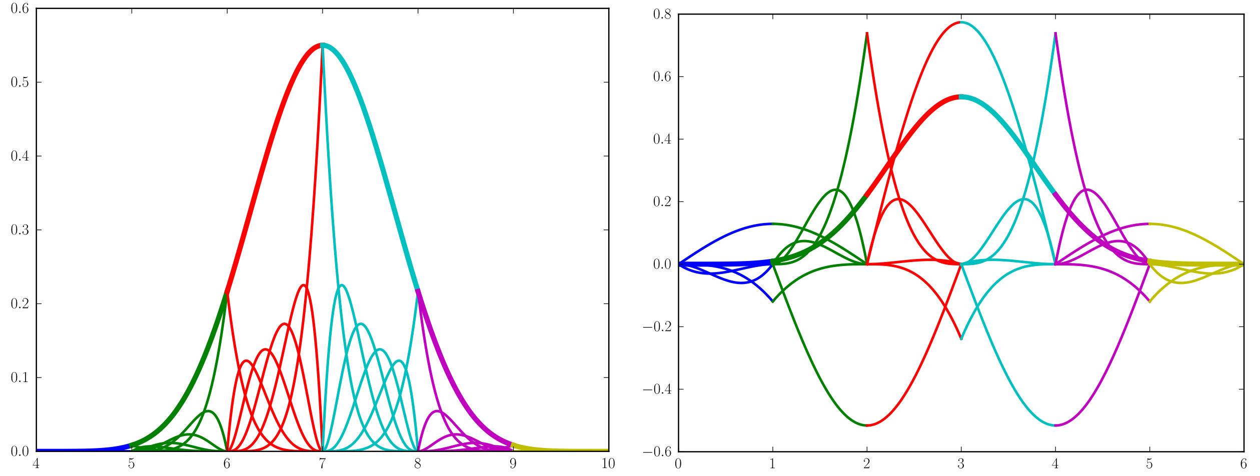

Figure 4.3: The local polynomial representation for a uniform B-spline basis function of degree 5 compared with the local representation for a uniform GB-spline function of degree 5 defined using trigonometric knot functions.

recurrence (see Definition 2.33), however it does make it so that the evaluation of GB-spline curves is no longer tied to symbolic integrals or recursive quadrature. It makes it clear that the values of N_i^p on the interval $[t_j, t_{j+1})$ depend only on $N_i^p(t_j)$, the values of $u_j^{[p-1]}$ and $v_j^{[p-1]}$ at t and t_j, and the full set of coefficients for N_i^{p-1} and N_{i+1}^{p-1}. These dependencies can be stated more explicitly. To evaluate N_i^p at time $t \in [t_j, t_{j+1})$, it is necessary to know the values of the following function values:

- $u_j^{[p-1]}$ and $v_j^{[p-1]}$ at t
- the values of all the different $u_j, u_j^{[1]}, \ldots, u_j^{[p-2]}$ and $v_j, v_j^{[1]}, \ldots, v_j^{[p-2]}$ at t_j and $t_j + 1$ for each index j corresponding to an interval in the support of N_i^p
- The values of $u_j^{[p-1]}$ and $v_j^{[p-1]}$ if j is an index corresponding to an interval in the support of N_i^p at t_j if $t_j < t$ and at t_{j+1} when $t_{j+1} < t$

To construct a direct algorithm for the evaluation of arbitrary GB-spline curves, we must first determine how to best to trace through the dependencies between the intervals. Given that the representation of N_i^p on $[t_j, t_{j+1})$ depends on the representations of N_i^{p-1} and N_{i+1}^{p-1} over their supports and the representations of N_i^p over the intervals of its support that lie

to the left of $[t_j, t_{j+1})$, it is natural to construct the set of basis functions of each degree using the set of basis functions of the degree one less than the one being computed. The computation most naturally runs from left to right along each basis function. Given this structure, the algorithm should operate roughly as follows:

- Initialize a list of basis functions using the known values for the degree 1 case.
- For each degree from 2 to the desired degree p, do the following:
  - Integrate each polynomial term in the basis.
  - Use the polynomial terms, the transcendental coefficients, and the values of the indefinite integrals of the knot functions at the points in the knot vector to compute the definite integral of each basis function over its support.
  - Divide the indefinite integrals of the polynomial terms by the definite integral of the basis function they represent.
  - Divide the transcendental coefficients by the definite integral of the basis function they represent.
  - Compute the differences between the scaled transcendental coefficients for basis functions whose indices differ by 1.
  - Compute the differences between the scaled polynomials for basis functions whose indices differ by 1, adding the constant terms from the transcendental part to the polynomial.
  - Use the values of these differences to add the value of each basis function over each interval to its polynomial term over each interval.
  - Store these differences between the polynomial and transcendental terms as the new set of basis functions.

In practice, the functions that we desire to include in the span of the spline basis may not always satisfy the constraints on the values of the knot functions at the endpoints of

each interval. This can be resolved by taking linear combinations of the original functions on each interval so that the value constraints are satisfied. This can be taken care of as a part of the algorithm for constructing a basis by taking the needed linear combinations of the integral terms given as input and then, once the local representations of the spline basis have been computed, changing the representations so that they are given with respect to the original functions rather than the computed linear combinations. In order to ensure the desired properties of a spline basis, the functions used to create the knot functions must still form a Chebyshev space over each corresponding nondegenerate interval in the knot vector. The matrix

$$\begin{bmatrix} u_i\left(t_i\right) & v_i\left(t_i\right) \\ u_i\left(t_{i+1}\right) & v_i\left(t_{i+1}\right) \end{bmatrix}$$

must also be invertible (and sufficiently well-conditioned) so that the needed linear combinations can be computed.

In addition, the spline basis constructed will span $u^{[p-1]}$ and $v^{[p-1]}$, not u and v. It is often desirable to construct a basis that spans C^{p-1} functions $\tilde{u}$ and $\tilde{v}$ instead. To handle that properly, we need only begin the iteration with the knot functions $\tilde{u}^{(p-1)}$ and $\tilde{v}^{(p-1)}$, noting that, after the corresponding numbers of integrals have all been taken, the spline basis will span the desired functions. To use this approach, it is necessary that there exist linear combinations of $\tilde{u}^{(p-1)}$ and $\tilde{v}^{(p-1)}$ that satisfy the constraints that would normally be imposed on u and v.

The algorithms here will be presented in a form that is independent of the polynomial basis used. Power basis polynomials provide a simple base case, but known identities for operations on other polynomial bases can be used to perform these same operations with respect to different polynomial bases. Here, operations on polynomial terms will be performed by auxiliary routines with specific and well documented purposes that can be implemented in different ways for different polynomial representations.

For practical use it is also helpful to follow the convention that the knot functions and

polynomials corresponding to each interval are defined on the interval $[0, t_{i+1} - t_i]$ and that $t - t_i$ is used as an argument rather than t itself. This makes it so that, for any given polynomial representation requiring boundaries of definition (Bernstein polynomials, Chebyshev polynomials, etc.), only the lengths of each interval must be passed to the polynomial integration and evaluation routines.

The recurrence outlined in Definition 4.1 is restated as an algorithm in Algorithm 9. In the computation here, we will store each basis as two arrays, the first containing the polynomial terms corresponding to each interval within the support of each basis function and the second containing the corresponding transcendental terms. Given that the support of each basis function is known, we only include the representation of each basis function on an interval where it will be nonzero. This shifts the indices for the representation of each basis function, but the structure of the iteration is essentially the same.

An important consequence of using the piecewise representations for these basis functions is that, once a piecewise representation for a spline curve is created, the only remaining costs of evaluating the function at any given point come from identifying which interval in the knot vector contains the given parameter value, evaluating a polynomial term, and evaluating the terms $u_i^{[p-1]}$ and $v_i^{[p-1]}$. No further recursion or integration is needed.

The algorithms will be presented in vectorized form with a particular emphasis on clarity. A variety of other small optimizations could be added to further remove redundant computations; however the presentation here is meant primarily to provide a clear explanation of the algorithm. It presents a relatively efficient version of the algorithm, but, for simplicity, redundant computations have not been completely removed.

For clarity within the algorithm and its helper routines, we will now introduce the many variables used throughout this algorithm and its corresponding helper routines. Throughout the code for this algorithm, the following variables will be used:

- T will be a 1-dimensional array containing the knot vector.
- *Tlens* will be the lengths of the intervals between the knots values in T.

- *Tvals* will be an array shaped like *Tlens* containing the lengths of each interval. Rather than be indexed according to interval, *Tvals* will be indexed first by basis function, then by interval within the support of a given basis function.

- p will be the degree of the desired basis, or of the spline to be evaluated.

- d will be a looping variable used in the loop that constructs the basis of each degree from the basis of previous degree. Here d will be the degree of the basis being constructed. *dmin* will be equal to $d - 1$.

- n will be the number of basis functions in a given basis.

- *ints* will be a 4-dimensional array of shape $(p, \text{len}(t) - 1, 2, 2)$ containing the values of the indefinite integrals of the knot functions at the endpoints of each interval. The p'th integrals should be indexed in ascending order along the first axis. The different intervals within the knot vector should be indexed along the second axis. The different endpoints of each interval should be indexed along the third axis. The different knot functions (u and v) will be indexed along the last axis with u first.

  wints will be a 3-dimensional view into *ints* corresponding to the integral values of a given degree, indexed first by basis function, then by interval within the support of each basis function, then by endpoint, then by the different knot functions.

- *polys* will be an array containing the coefficients for the polynomial parts of the basis functions in a given basis. Basis functions should be indexed along the first axis and intervals within the support of each basis function should be indexed along the second axis. Here we will assume that the polynomial term over the i'th interval is stored in the form $p(t - t_i)$, i.e. that the polynomial terms are translated so that the first value taken by the polynomial in each interval is the value of the polynomial at 0. This algorithm does not depend on the representation used to store the polynomial terms, but in most cases an array of shape $(n, p + 1, p - 1)$ containing only the necessary coefficients should suffice.

- *pints* will be an array containing the integrals of all the polynomial terms in a given array *polys*. All the axes will be indexed the same as the axes in *polys*. The shape will be the same except that the last axis will be one index longer than the last axis of *polys*.
- *trans* will be an array containing the coefficients for the transcendental terms of the basis functions in a given basis. Basis functions should be indexed along the first axis, intervals within the support of each basis function along the second axis, and the two transcendental coefficients along the third (with u first, then v). This array will have a shape of $(n, p+1, 2)$.
- *scal* will be an array containing the necessary scaling matrices needed to scale *ints* to represent the scaled versions of u and v that satisfy the value constraints at their endpoints and also needed to scale the coefficients in *trans* so that they represent the basis functions in terms of the original u and v.
- *pos* will be an array of boolean values. The i'th entry of *pos* will be true if $\delta_i^{d-1} \neq 0$.
- *deltas* will be an array containing the indefinite integrals of each basis function over its support.
- *consts* will be an array containing the constant terms to be added to the polynomial terms on each interval. It will be indexed first by basis function, then by interval within the support of each basis function. In the recurrence in Definition 4.1, these are the terms

$$-a_{i,j}^{d-2} u_j^{[d-2]}(t_j) - b_{i,j}^{d-2} v_j^{[d-2]}(t_j)$$

- *vals* will be a temporary array used to store outputs of various functions.

Algorithm 9 shows the primary routine used to compute the local representations of a given basis function. It contains calls to a variety of auxiliary routines, all of which will be explained here. Here we include the primary algorithm first so that the reader may

understand the general flow of the algorithm and the proper place for each auxiliary routine before handling the many details that are taken care of in the auxiliary routines.

Algorithm 9 uses the following auxiliary routines:

- *Wrap*: A utility function used to convert between indexing by interval to indexing by basis function, then by interval within the support of each basis function.

- *MatMul*: A utility function that performs matrix multiplication following certain broadcasting semantics.

- *MakeDegreeOne*: A function that initializes the coefficient arrays for a basis of degree 1.

- *ScaleKnotFuncs*: A function that computes the scaled *ints* and the corresponding array *invs* containing the scalings. This function is what changes all the basis functions to be represented in terms of the linear combinations of the functions used to create *invs* that satisfy the required constraints to be knot functions.

- *ScaleTransCoefs*: A function that modifies *trans* in-place to change the coefficients to represent the basis functions in terms of the functions used to create *invs*.

- *PolyInt*: A function that, given an array of polynomial coefficients with the coefficients indexed along the last axis and another array containing the lengths of the intervals corresponding to each polynomial term, computes the indefinite integrals of all the polynomials.

- *PolyVal*: A function that, given an array of polynomial coefficients with the coefficients indexed along the last axis, and an array containing the lengths of the intervals corresponding to each polynomial term, evaluates each polynomial at the corresponding term in an array *vals*. This function is used only within other auxiliary routines.

- *IntegrateSupports*: A function that computes the integral of each basis function over its support

Algorithm 9 Computing the local coefficients for a GB-spline basis

1: **procedure** BasisCoefs($T, ints, tol = 10^{-8}$)
2: ▷ Initialize p, $Tlens$, and $Tvals$ and coefficient arrays for a basis of degree 1.
3: $p = \text{shape}(ints)[0]$
4: $Tlens = T[1:] - T[:-1]$
5: $Tvals = \text{Wrap}(Tlens, 2)$
6: $polys, trans = \text{MakeDegreeOne}(\text{shape}(T)[0] - 2)$
7: ▷ Take linear combinations of the functions with integrals in $ints$ so that
8: ▷ the resulting linear combinations satisfy constraints on knot functions.
9: $ints, scal = \text{ScaleKnotFuncs}(ints, Tlens, tol)$
10: ▷ Construct each successive set of local coefficients.
11: **for** $d = 2$, $d \leq p$ **do**
12: ▷ Compute the indefinite integrals of all polynomial terms
13: $pints = \text{PolyInt}(polys, Tvals)$
14: ▷ Construct $wints$ by wrapping the first axis of $ints$ into two new axes.
15: $wints = \text{Wrap}(ints[d-1], d)$
16: ▷ Integrate the current set of basis functions over their supports.
17: $deltas, consts = \text{IntegrateSupports}(Tvals, pints, trans, wints)$
18: ▷ Add constant terms from the transcendental integrals to the $pints$.
19: $\text{OffsetConstants}(pints, consts)$
20: ▷ Compute the indices of the basis functions that are identically 0.
21: $pos = (T[d:] - T[:-d]) > tol$
22: ▷ Take the deltas corresponding to positive basis functions.
23: ▷ Also reshape the deltas for broadcasting with $pints$ and $trans$.
24: $deltas = deltas[pos, None, None]$
25: ▷ Divide the terms in $pints$ and $trans$ by their corresponding entries in $deltas$.
26: $pints[pos] \mathrel{/}= deltas$
27: $trans[pos] \mathrel{/}= deltas$
28: ▷ Take the differences between neighboring terms in $pints$ and $trans$.
29: $polys, trans = \text{OffsetDifferences}(pints, trans)$
30: ▷ Add ones where needed to account for the integral terms of 0-valued
31: ▷ basis functions after the knot value where their support would end.
32: $\text{AddOnes}(polys, pos)$
33: ▷ Compute the set of Tvals for the next basis.
34: $Tvals = \text{Wrap}(Tlens, d + 1)$
35: ▷ Add in the constant terms that come from evaluating each basis function
36: ▷ at the end of each interval within the knot vector.
37: $\text{ConnectBoundaries}(polys, trans, wints, Tvals)$
38: **end for**
39: ▷ Scale the coefficients in $trans$ so that the basis functions are represented
40: ▷ in terms of the transcendental terms originally represented in $ints$.
41: $\text{ScaleTransCoefs}(\text{Wrap}(scal, p + 1), trans)$
42: **return** $polys, trans$
43: **end procedure**

- *TransInts*: A helper function called within *IntegrateSupports*. It computes the portion of the integral of each basis function that comes from the knot functions over each interval in its support.

- *OffsetConstants*: A function that modifies *pints* in-place to add in the constant terms that come from the transcendental integral. In the polynomial part of the recurrence from Definition 4.1, this accounts for the terms

$$-a_{i,j}^{p-1} u_j^{[p-1]}(t_j) - b_{i,j}^{p-1} v_j^{[p-1]}(t_j)$$

  and

$$a_{i+1,j}^{p-1} u_j^{[p-1]}(t_j) - b_{i+1,j}^{p-1} v_j^{[p-1]}(t_j)$$

- *OffsetDifferences*: A function that takes the *pints* and *trans* (after each term has been divided by the corresponding δ_i terms) that correspond to a given basis and computes the differences between consecutive terms. This returns the differences between both the polynomial and transcendental terms. These terms account for all terms in the recurrence from Definition 4.1 with the exception of $N_i^p(t_j)$.

- *AddOnes*: A function that adds the one terms to *polys* that come from the handling of the integral terms from basis functions that are identically 0 as defined in Definition 3.5.

- *ConnectBoundaries*: A function that computes the term $N_i^p(t_j)$ for each interval where it is needed and adds it in place to *polys*.

We will now discuss the implementations of these different functions in greater detail.

The helper routine *Wrap* is used to expand a given axis into two axes where each index of the first of the two new axes provides a moving window of the given width along the original axis that is being expanded. Within this algorithm, this function is used to convert

between data that is indexed by interval within the knot vector to data that is indexed by basis function, then by interval within the support of each basis function.

The implementation for the routine *Wrap* will depend on the language used. A complete discussion of its implementation is not crucial to the algorithm itself, but we will discuss it briefly here to aid anyone who wishes to implement this algorithm. It can be implemented easily by allocating a new array and looping through it, filling it with the appropriate values. It can also be implemented to work without consuming any extra memory using NumPy's support for arbitrary strided arrays. In Python it is easy to perform this operation along any given axis, but whenever this function is used here it will always be used to expand along the first axis. In Python the function will look like this:

```
def Wrap(a, width, axis=0):
    """
    Use stride tricks to make a new array that views 'axis' of 'a'
    as two new axes where an entry of the first of the two new axes
    is a rolling window of 'width' length of the entries indexed
    along 'axis' of 'a'.
    """
    axis = axis % a.ndim
    # Compute the shape of the desired output array.
    shape = (a.shape[:axis] + (a.shape[axis]-width+1,width) + a.shape[axis+1:])
    # Make the stride of the new array the same along both expanded axes.
    strides = a.strides[:axis] + (a.strides[axis],) + a.strides[axis:]
    # Return an array of the given shapes and strides viewing the original array.
    return np.lib.stride_tricks.as_strided(a, shape, strides)
```

The implementation for the routine *MatMul* also depends heavily on the language used. It can be understood most directly as a matrix multiplication routine following the same array broadcasting semantics used for the other arithmetic operators. The only difference is that matrix multiplication is, fundamentally, an operation between 2-dimensional arrays, so the last two axes of the operand arrays must match as would be expected for matrix multiplication. The broadcasting occurs along all other axes. The matrix multiplication is done for each specific index for all the leading axes, operating on slices along both of the last two axes of each array.

The function *MatMul* can be implemented in different ways depending on the language. In Python this can be done like this:

```
def MatMul(a, b):
    """
    Perform matrix multiplication between the arrays 'a' and 'b'
    following normal gufunc broadcasting rules.
    """
    return np.einsum('...ij,...jk', a, b)
```

The implementation for *MakeDegreeOne* should also be relatively simple. Recall that each degree 1 basis function has the form

$$\begin{cases} v_i(t) & x \in [t_i, t_{i+1}) \\ u_i(t) & t \in [t_{i+1}, t_{i+2}) \end{cases}$$

This means that this function should allocate *polys* as an empty array of shape $(n, 2, 0)$ and allocate *trans* as an array of 0's of shape $(n, 2, 2)$. Then it should fill *trans* with values such that *trans* $[i, j, k]$ is equal to 0 when $j = k$ and 1 otherwise. It should then return *polys* and *trans*. In Python, this can be done as follows:

```
def MakeDegreeOne(n):
    """
    Generate the polynomial and transcendental terms
    for a degree 1 basis with 'n' basis functions.
    """
    trans = np.zeros((n, 2, 2))
    trans.reshape((n, 4))[:,1:3] = 1
    return np.empty((n, 2, 0)), trans
```

ScaleKnotFuncs allows derivatives from functions that do not necessarily satisfy the constraints $u_i(t_i) = v_i(t_{i+1}) = 1$ and $u_i(t_{i+1}) = v_i(t_i) = 0$ for the integral values stored in *ints*. To do this, it must require that, for each nonempty interval $[t_i, t_{i+1})$, the matrix

$$A_i = \begin{bmatrix} u_i(t_i) & v_i(t_i) \\ u_i(t_{i+1}) & v_i(t_{i+1}) \end{bmatrix}$$

be invertible and reasonably well-conditioned. It is still required that u_i and v_i form a Chebyshev space.

Since the algorithm for constructing the basis coefficients relies on each u_i and v_i satis-

fying the constraints on its values at the endpoints of each interval in the knot vector, we must compute the linear combinations of u and v that satisfy the value constraints at each endpoint. Since matrix multiplication can be seen as using the columns of the matrix on the right as coefficients for linear combinations of the columns of the matrix on the left, we see that the matrix B_i with the desired coefficients for the linear combinations must satisfy the equation $A_i B_i = I$, so $B_i = A_i^{-1}$. Since it is only necessary to invert matrices of size 2×2, for simplicity we will content ourselves with using a direct matrix inverse to compute the new derivatives, though other methods could be used to compute the desired derivative and integral values.

Now, given the matrices B_i, we must now use the coefficients for the desired linear combinations stored as columns of B_i to compute the corresponding integral terms. Using similar reasoning as before, taking the needed linear combinations of the integral terms stored in *ints* corresponds to right-multiplication of each 2×2 matrix corresponding to a given degree and interval by the matrix B_i corresponding to that interval. This routine must return both the new scaled version of *ints* and the corresponding matrices B_i (these are the i'th entry along the first axis of *ints*) because the matrices B_i must be used again later to write the computed coefficients for the transcendental functions in terms of the original u and v rather than their scaled linear combinations. The internal workings of this auxiliary routine are outlined in Algorithm 10.

Once the main loop in Algorithm 9 is finished, the computed transcendental coefficients must be changed to represent each basis function in terms of the original knot functions rather than the chosen linear combinations of them. This is equivalent to left-multiplying the set of coefficients for each interval by the matrix B_i (as in the explanation for *ScaleKnotFuncs*). This process is shown in Algorithm 11.

The auxiliary routine *PolyInt* is dependent on the polynomial representation used. The array *Tvals* is used as an argument because the polynomial basis used could be defined over some given interval (as are the Bernstein, Chebyshev, and Legendre polynomials). For the

Algorithm 10 Take linear combinations of the input knot functions such that the desired linear combinations will satisfy the constraints $u_i(t_i) = v_i(t_{i+1}) = 1$ and $u_i(t_{i+1}) = v_i(t_i) = 0$. Return the corresponding integral terms of these linear combinations and the coefficients for the linear combinations over each interval.

1: **procedure** ScaleKnotFuncs($ints, Tlens, tol = 1E-8$)
2: ▷ Copy *ints* so it can be modified in-place without modifying the input array.
3: $ints = \text{copy}(ints)$
4: ▷ Get a boolean array showing where the the lengths of the intervals are nonzero.
5: $pos = Tlens > tol$
6: ▷ Compute the coefficients of the needed linear combinations.
7: $invs =$ array of 0's of shape $\text{shape}(ints)[1:]$
8: $invs[pos] = \text{inv}(ints[0, pos])$
9: ▷ Perform matrix multiplication of each set of coefficients for each interval and degree
10: ▷ by the corresponding scaling matrix for each interval.
11: $ints[:] = \text{MatMul}(ints, invs)$
12: **return** $ints, invs$
13: **end procedure**

Algorithm 11 Perform a change of basis on the transcendental coefficients so that the transcendental coefficients used to represent the basis correspond to the functions originally used to form the array *ints* of integral values.

1: **procedure** ScaleTransCoefs($invs, trans$)
2: **return** $\text{MatMul}(invs, trans[\ldots, None])[\ldots, 0]$
3: **end procedure**

power basis polynomials, that argument is not needed. As has already been mentioned, the interval lengths are all that is necessary since the polynomial and transcendental terms are all assumed to be shifted to be defined over an interval starting at 0.

The auxiliary routine *PolyVal* should be handled similarly as *PolyInt*. This routine is also dependent on the polynomial representation and is easily defined as using Horner's algorithm, the De Casteljau algorithm, Clenshaw's algorithm, or some other polynomial evaluation algorithm.

$TrigInts$ is a function to compute the transcendental integrals

$$\int_{t_i}^{t_{i+1}} \left(a_{i,j}^{d-2} u_{i,j}^{[d-2]}(s) + b_{i,j}^{d-2} v_{i,j}^{[d-2]}(s) \right) ds$$

with the corresponding constant terms

$$-a_{i,j}^{d-2} u_j^{[d-2]}(t_j) - b_{i,j}^{d-2} v_j^{[d-2]}(t_j)$$

from the left endpoint of the integral. It is dependent on the representation used for the knot functions (we've only introduced using the knot functions themselves thus far). In the case that the knot functions themselves are used, Algorithm 12 shows how this can be done.

Algorithm 12 Integrate the transcendental part of each basis function over each interval in the support of that basis function.

1: **procedure** TRANSINT($trans, wints$)
2: $\quad consts = -\text{sum}\left(trans * wints\left[:, :, 0\right], axis = -1\right)$
3: $\quad vals = \text{sum}\left(trans * wints\left[:, :, 1\right], axis = -1\right) + consts$
4: $\quad$ **return** $vals, consts$
5: **end procedure**

IntegrateSupports is a function that evaluates the integrals of each basis function over its corresponding support. It should return both the desired indefinite integrals and the constant terms (stored in variable *consts*) that come from the left bounds of each integral. This function is outlined in Algorithm 13.

Algorithm 13 Compute the definite integrals of each basis function over its support.

```
1: procedure IntegrateSupports(Tvals, pints, trans, wints)
2:     ▷ wints and Tvals line the integral terms and the interval lengths up
3:     ▷ with their corresponding interval in each basis function.
4:     vals, consts = TransInt (trans, wints)
5:     vals+ = PolyVal (pints, Tvals, Tvals)
6:     return sum (vals, axis = −1) , consts
7: end procedure
```

OffsetConstants is another helper routine that interfaces with the polynomials and is dependent on how the polynomials are represented. It adds the terms stored in *consts* to the corresponding terms in *pints*. In the case of the power basis, Chebyshev basis, or Legendre basis, this can be done by adding each constant term to the term in the polynomial representation that represents constants. In the case of the Bernstein polynomials, since all the coefficients sum to 1, adding a constant is the same as adding a constant to each coefficient, so this operation can be performed by adding the constant term for each polynomial to all the coefficients for that polynomial.

OffsetDifferences is an auxiliary routine that takes care of differencing between the integrated terms from the previous basis function to form the differences over each interval that are needed to form the new basis. Once this function has been applied, the terms account for everything included in the recurrence in Definition 4.1 with the exception of the constant term for each interval that comes from evaluating each basis function on the right endpoint of the interval to the left of the current interval. This function also does not account for adding the ones to handle basis functions that are identically 0. The pseudocode for this algorithm is outlined in Algorithm 14.

AddOnes, as has already been mentioned, is used to add the ones that come from the integral terms from Definition 3.5 that correspond to basis functions that are identically 0. We have separated it as an auxiliary routine because it both depends on the polynomial basis used and because this allows a more focused description of the operation itself. The 1's must be added only to the last interval of basis functions for which the first term of the

Algorithm 14 Take the differences between the integral terms for the previous set of basis functions to start forming the new set of basis functions.

```
1:  procedure OffsetDifferences(pints, trans)
2:      n = shape(pints)[0]
3:      nints = shape(pints)[1]
4:      ▷ Allocate the arrays needed to store the coefficients for the new basis.
5:      npolys = new array of 0's of shape (n − 1, nints + 1, dmin)
6:      ntrans = new array of 0's of shape (n − 1, nints + 1, 2)
7:      ▷ Take the needed differences between the corresponding terms.
8:      npolys[:, : −1] += pints[: −1]
9:      npolys[:, 1 :] −= pints[1 :]
10:     ntrans[:, : −1] += trans[: −1]
11:     ntrans[:, 1 :] −= trans[1 :]
12:     return npolys, ntrans
13: end procedure
```

recurrence from Definition 3.5 corresponds to a basis function that is identically 0. This is because, when constructing the basis functions of the next highest degree, the integral term corresponding to a basis function with index i appears only in the expressions for the basis functions at index $i-1$ and i. Of those two basis functions, only the basis function at index i takes nonzero values on an interval that lies to the right of the support of the basis function that is identically 0. Once understood, this operation is very simple to perform, as can be seen in Algorithm 15, which shows how this auxiliary routine would be implemented for polynomials represented in the power basis. Though this routine depends on the polynomial representation used, it is not necessary to pass *Tvals* since a constant terms is the same for a polynomial represented over any interval.

Algorithm 15 Add ones where needed to account for the integral terms in Definition 3.5 that correspond to basis functions that are identically 0.

```
1: procedure AddOnes(polys, pos)
2:     ▷ Where the integral of the basis function of previous degree at the same
3:     ▷ index was 0, add 1 to the constant term of the last polynomial term.
4:     polys[∼ pos[: −1], −1, −1] += 1
5: end procedure
```

ConnectBoundaries is the last auxiliary routine needed to construct the new basis functions from the previous ones. In the recurrence in Definition 4.1, this function adds in the

terms $N_i^p(t_j)$. This function effectively starts at the leftmost interval in the support of each basis function, computes the value of the basis function at the end of that interval, adds that constant term to the polynomial term of the basis function on the next interval, and continues until it has added the needed constant terms to every interval in the support of that basis function. This is done in a vectorized manner in Algorithm 16.

Algorithm 16 For each basis funciton, add in the constant terms $N_i^p(t_j)$ to each interval where they are needed.

1: **procedure** ConnectBoundaries($polys, trans, wints, Tvals$)
2: ▷ For each basis function, evaluate all but the leftmost polynomial term at the end
3: ▷ of the interval where it is defined.
4: $vals = \text{PolyVal}\left(polys\left[:,:-1\right], Tvals, Tvals\left[:,:-1\right]\right)$
5: ▷ Add in the corresponding values of the transcendental functions.
6: $vals \mathrel{+}= \text{sum}\left(trans\left[:,:-1\right] * wints\left[:-1,:,1\right], axis=-1\right)$
7: ▷ Add the needed constants to their corresponding polynomial terms.
8: $\text{OffsetConstants}\left(polys\left[:,1:\right], \text{cumsum}\left(vals, axis=1\right)\right)$
9: **end procedure**

Chapter 5. Refinement operations on GB-splines

In design, it is often desirable to represent a spline curve of degree p over a given knot vector T_0 as a different spline curve of degree $q \geq p$ over a different knot vector T_1 where the active regions for T_0 and T_1 coincide and the spline basis B_0 over T_0 is contained in the span of the spline basis B_1 on T_1. This process is called refinement. The introduction of the local bases V_i^p makes it so that any refinement operation of this form can be performed by computing the local representation for a spline over T_0 and then projecting it onto the desired basis functions over T_1. For B-splines, refinement algorithms are well-studied. Here we provide an algorithm for the refinement of GB-splines.

In practice, knot insertion and degree elevation are the most common types of refinement operations. In the case of knot insertion, T_1 is the same as T_0 except that an additional knot has been inserted in the interior of the active region of T_0. In the case of degree elevation, T_1 is formed from T_0 by inserting knots at each knot from T_0 that lies on the interior of the

active region of T_0, adding an additional end condition knot on each side of the active region, and increasing the degree of the spline functions by 1. The active regions are maintained from T_0 to T_1 by the addition of an end condition knot at either end of the knot vector. The knots in the interior of the active region are repeated so that the continuity at the points in the knot vector is maintained in spite of the increase in the degree of the spline. Refinement by projecting to and from local representations also provides a natural way to represent a spline over T_0 in terms of a knot vector T_1 with different end condition knots. This is shown in Figures 5.6 and 5.7. Projecting in this way also makes it so that any combination of these operations can be performed simultaneously.

There are a variety of existing algorithms for performing refinement of B-splines. Knot insertion is well-discussed in most standard texts on B-splines. See [26] for an efficient degree elevation algorithm for B-spline curves and references to other existing algorithms. The algorithm outlined here can be used for both knot insertion and degree elevation and follows a technique similar to the one used in [12].

In the case of GB-splines, a method for knot insertion using recursive integrals is provided in [3]. As with the evaluation of GB-splines, the local piecewise representation of GB-splines can be used to avoid computing these recursive integrals. In the case of refinement, projection to and from the local bases can also be used to perform degree elevation and change the placement of end condition knots. The existence of a local basis also makes it so that coarsening operations like those performed on B-splines via Bézier projection in [12] can be naturally extended to GB-splines. For simplicity, here we will focus primarily on refinement operations.

When refining GB-splines, we must also be certain that the integrals of the knot functions that are present in the space V_i^p are spanned by the basis functions over the target knot vector. For example, when performing degree elevation by a single degree, the knot functions $\tilde{u}$ and $\tilde{v}$ chosen over an interval I_1 in T_1 must be equal to a linear combination of the derivatives of the knot functions u and v over the interval I_0 in T_0 that contains I_1. In

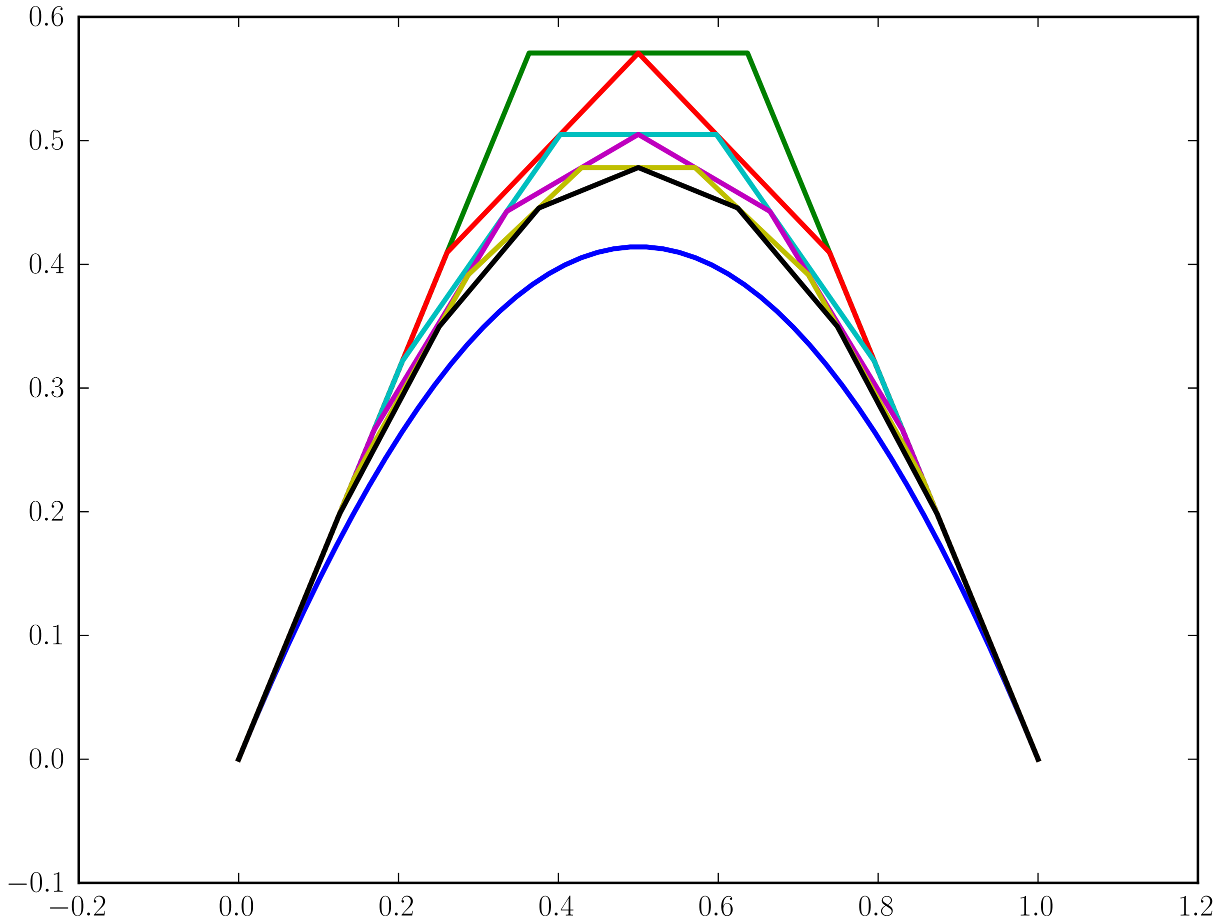

Figure 5.1: Various levels of degree elevation on a degree 2 GB-spline basis function from a Bernstein-like basis formed using the trigonometric functions $\cos\left(\frac{\pi}{2}t\right)$ and $\sin\left(\frac{\pi}{2}t\right)$ on the interval $[0, 1]$. The smooth curve is the spline, and the polygonal curves are control meshes of sucessively higher degrees.

general, it is sufficient for the knot functions $\tilde{u}$ and $\tilde{v}$ for an interval I_1 in T_1 used to form the target basis of degree q over T_1 to be linear combinations of the $(q-p)$'th derivatives of the knot functions u and v over the interval I_0 in T_0 containing I_1. These derivatives may not form a Chebyshev space over I_1, but this is sufficient. For example, in the case of polynomial terms, any additional derivatives of the original u and v are no longer linearly independent of one another, and construction of the new basis fails. On the other hand, as can be seen in [12], in the polynomial case, the polynomial terms over each basis provide a local basis that works perfectly well for refinement anyway. Examples of when successive derivatives do yield a Chebyshev space include trigonometric and exponential splines.

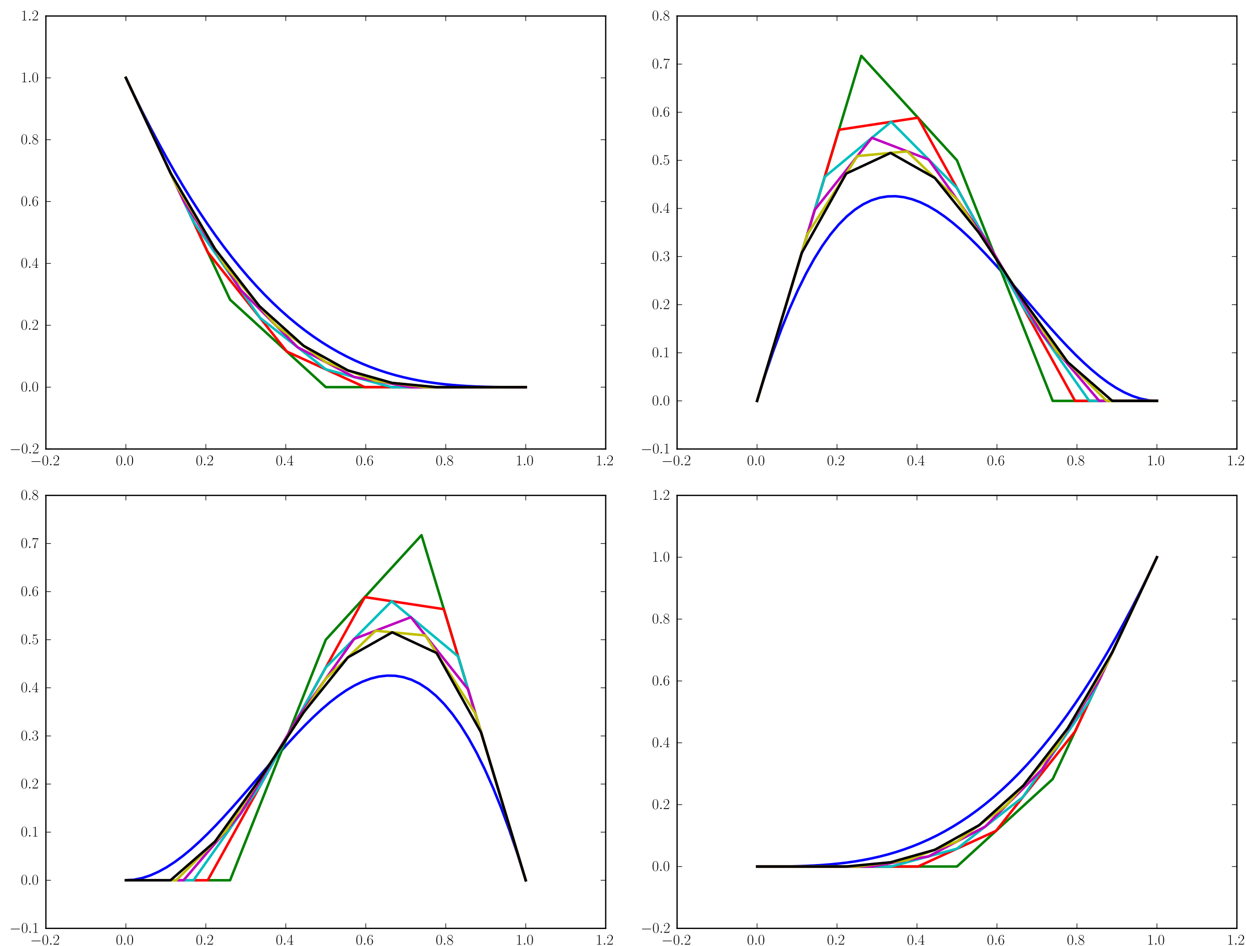

Figure 5.2: Successive degree elevations of each basis function in a Bernstein-like basis of degree 3 spanning trigonometric functions.

When the derivatives of the knot functions cease to form a Chebyshev space, refinement is still possible as long as the span of B_0 is contained in the span of B_1. All that is needed is a method for representing the integrals of the knot functions used in the local representations of B_0 in terms of the local bases used for B_1.

5.1 Algorithms For Refinement

Here we will present an algorithm that can be used to project a piecewise function in the span of a given spline basis B_0 defined on the active region of the knot vector T_0 corresponding to B_0 onto a different spline basis B_1, containing B_0, that is defined over a different knot vector

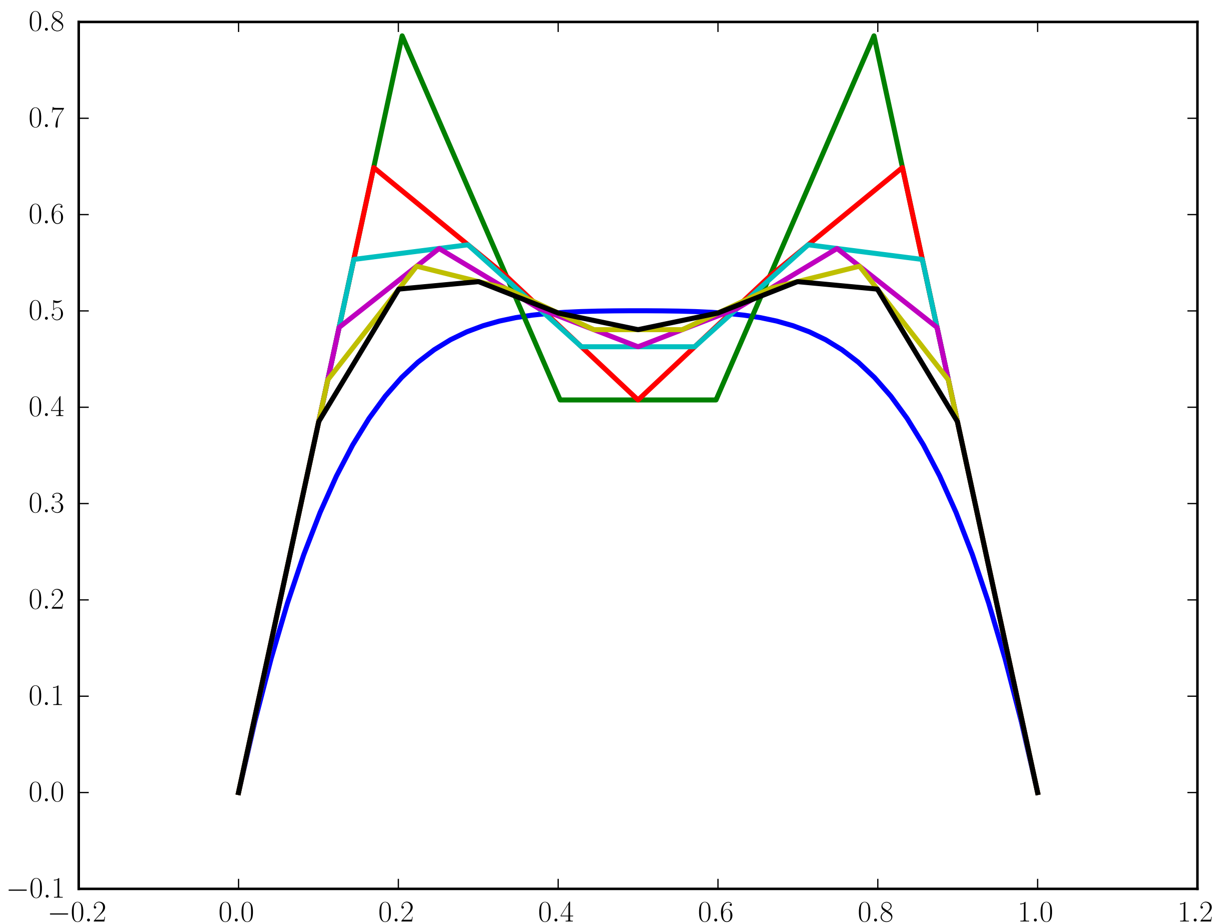

Figure 5.3: Various levels of degree elevation on a degree 4 GB-spline basis function from a Bernstein-like basis spanning trigonometric functions.

T_1. An additional routine will be provided to compute a piecewise representation from a set of control points and a set of local representations for a corresponding basis.

The main refinement routine can be outlined as follows:

- Write the piecewise function in terms of the local basis used to construct the basis B_1.
  - Subdivide the polynomial terms that correspond to intervals that have been divided into new intervals.
  - Degree-elevate the polynomial terms as many times as is necessary so that they are the same degree as the polynomial terms used for the target basis.

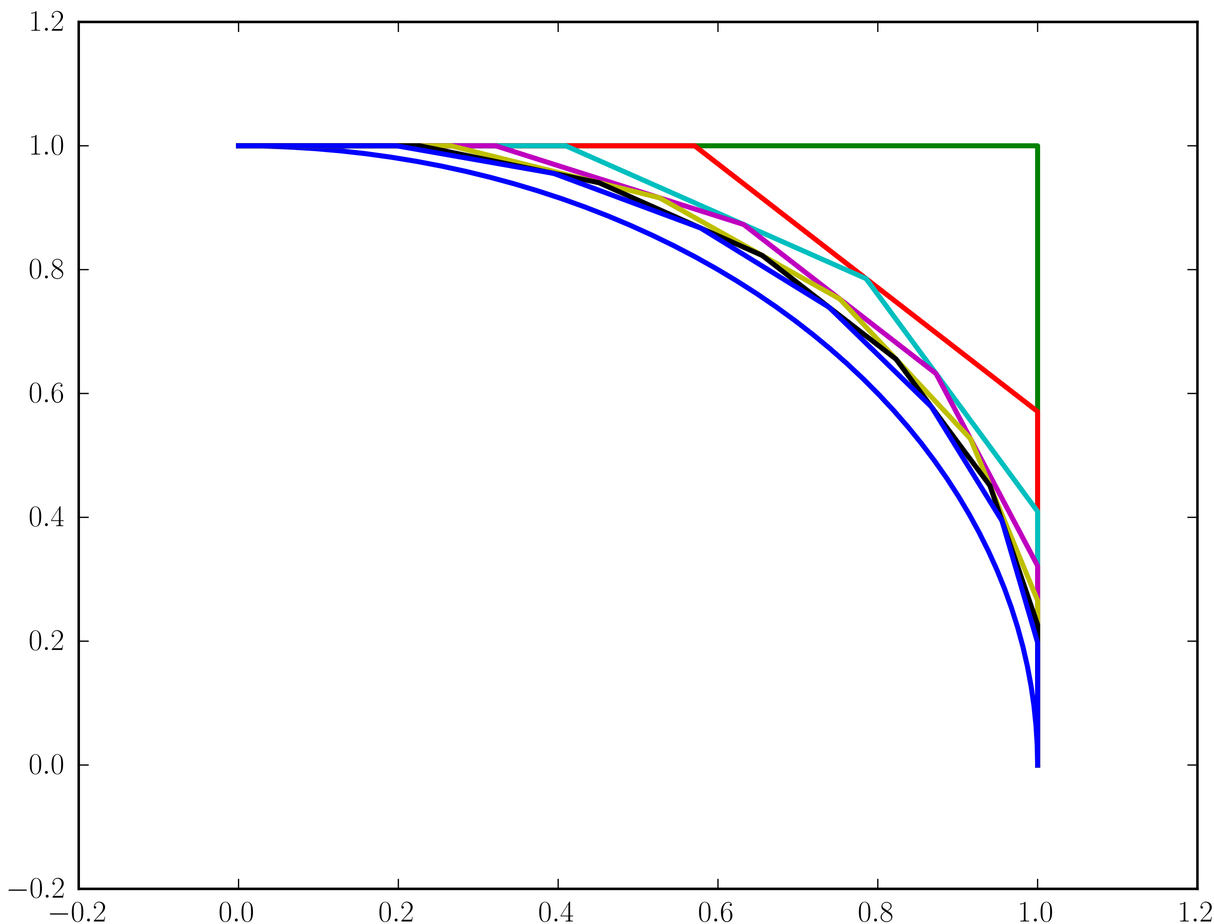

Figure 5.4: Various levels of degree elevation on a circle first represented as a degree 2 GB-spline. Again, trigonometric knot functions are used, and the polygonal curves show the control meshes of successively higher degrees. While the previous curves had the parameter domain along the horizontal axis, this shows a 2-dimensional spline curve.

  - Use the known values of the successive integrals of the knot functions to represent the knot functions over T_1 in terms of the local bases used for T_2.
  - Add the polynomial terms from the transcendental terms into the values for the degree-elevated polynomials.

- Solve the linear systems corresponding to nondegenerate intervals to compute the coefficients for each basis function
  - Here, for simplicity, aggregate the different computed coefficients for a given basis function by averaging them, raising an error if any value for an interval of non-

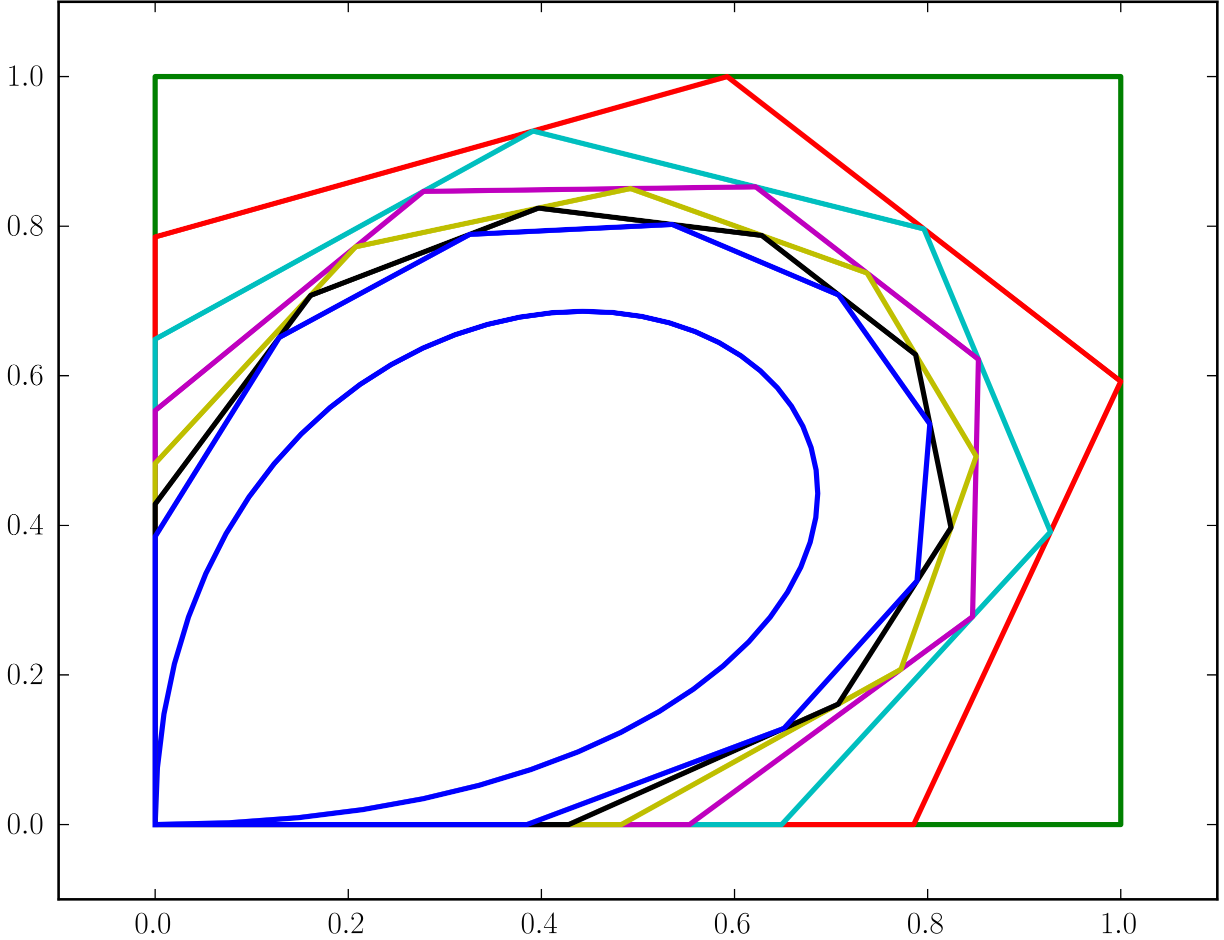

Figure 5.5: Various levels of degree elevation on a degree 4 2-dimensional GB-spline curve defined for a Bernstein-like basis spanning trigonometric functions.

negligible length is significantly different from the computed average.

This process is shown in greater detail in Algorithm 17

The end condition knots are only needed for computing the local representations of either set of basis functions. In addition, in order to compute the representation of the knot functions over T_0 in terms of the local bases for the intervals in T_1, the derivatives of the knot functions for T_0 and T_1 must be known and given *at the endpoints of all the intervals in the active region of* T_1.

Though they are not used directly in the algorithm, for the sake of explanation, we will let p be the degree of the spline given in piecewise form and q will be the degree of the target

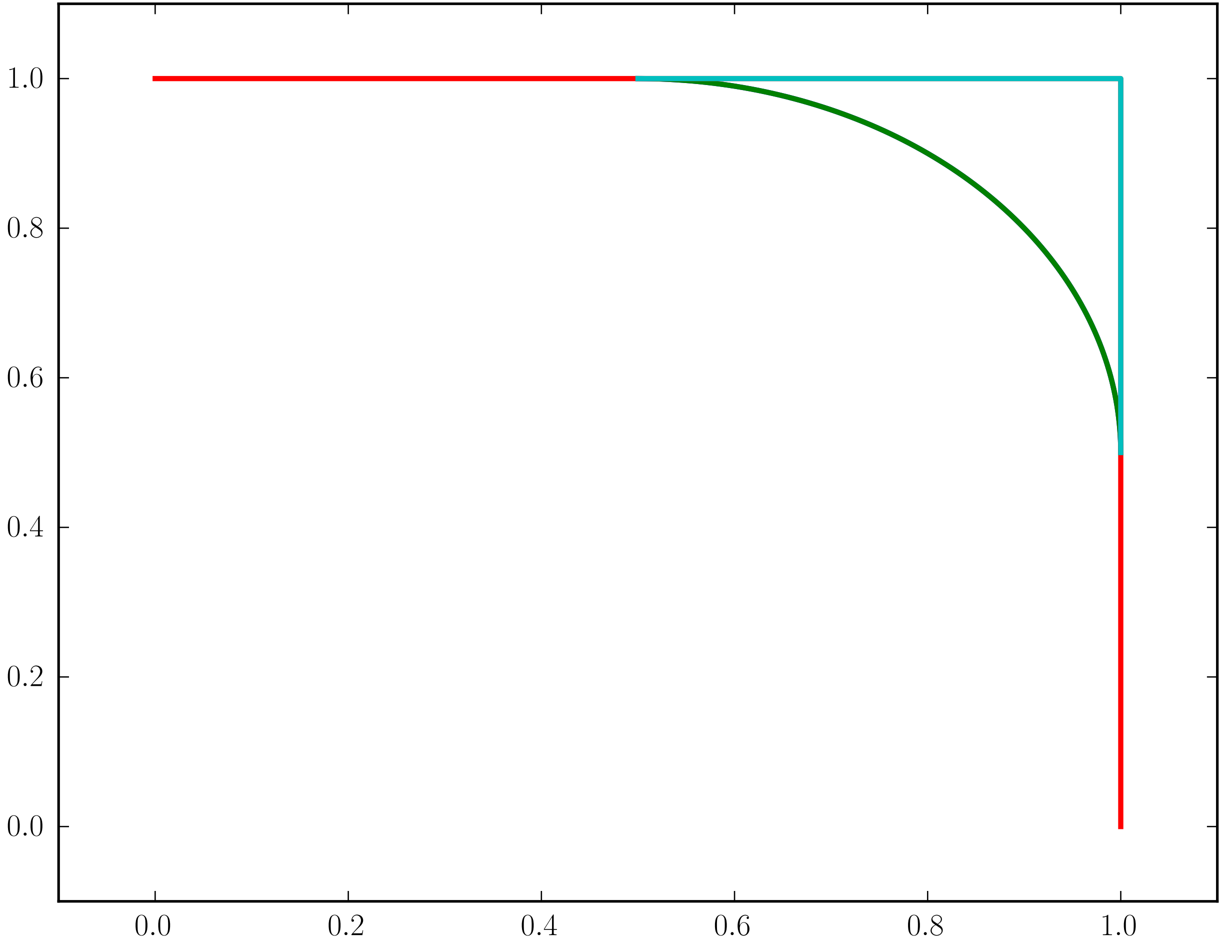

Figure 5.6: A circle represented with respect to GB-spline bases with trigonometric knot functions and different end conditions.

basis.

Throughout the descriptions of these algorithms the following variable naming conventions will be used

- *polys0* will be the polynomial terms for the piecewise function in the span of the first spline basis. It will be indexed first by interval within the active region of T_0, then by polynomial coefficient. In this algorithm it will be modified to contain the polynomial terms from the piecewise function represented over T_1.

- *polys1* will be the polynomial terms for the basis functions in target basis. It will be indexed the same as the variable *polys* is in the Algorithm 9.

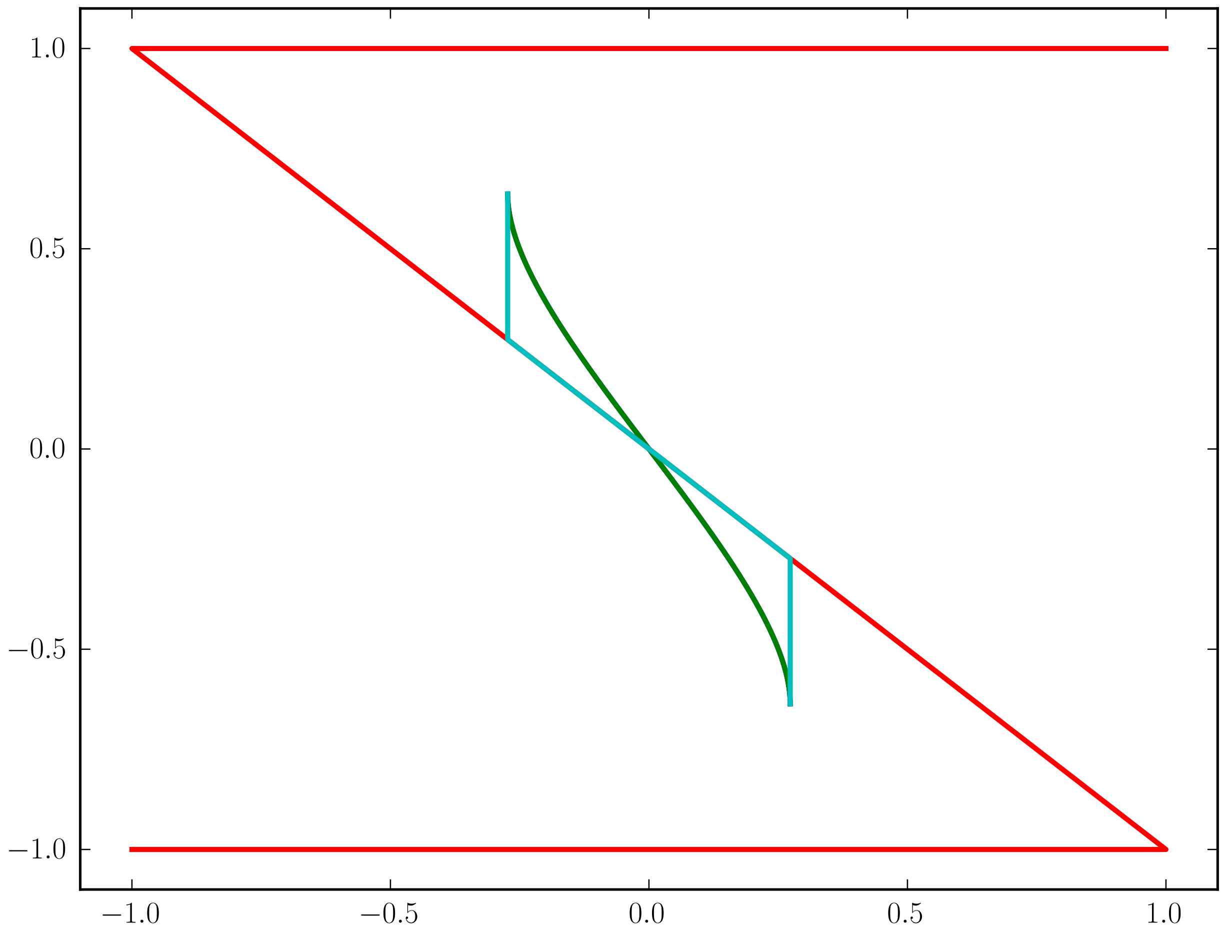

Figure 5.7: A 2-dimensional GB-spline curve represented with respect to GB-spline bases with different end conditions.

- *trans0* will be the coefficients for the integral terms of the knot functions corresponding to T_0. It will be indexed first by interval within the active region of T_0, then by the different integral terms. The term for u will be indexed at 0 and the term for v will be indexed at 1. This will be modified so that it will the same integral terms in the piecewise function in terms of the knot functions used to form the target basis.

- *trans1* will be the coefficients for the integral terms of the knot functions over T_1 in the piecewise representation of the target basis. It will be indexed the same as the variable *trans* is in Algorithm 9.

- *reg0* will be the knots that lie inside active region of T_0. Regardless of whether or not

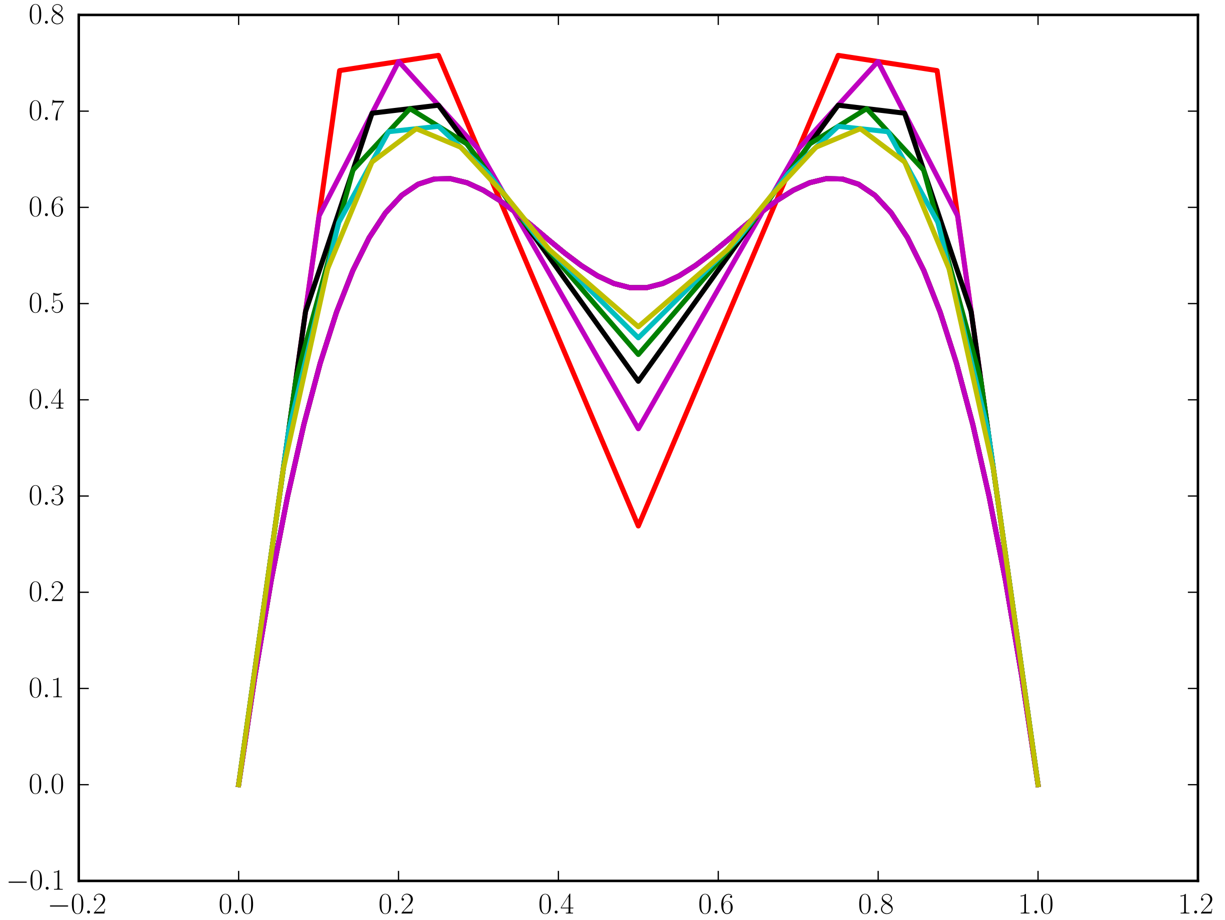

Figure 5.8: Degree elevations of a one dimensional GB-spline curve of degree 3 formed by a basis spanning trigonometric functions over the knot vector $[0, 0, 0, 0, .5, 1, 1, 1, 1]$.

the endpoints of the knot vector are repeated, if the piecewise term is a spline of degree p, this should be an array equivalent to $T0\,[p : -p]$.

- *reg1* will be the knots that lie in the active region of T_1, with indexing and handling of knot repetition, except that T_1 and the corresponding degree are used.

- *rints0* will be the 0 through $(q-1)$th integral terms for the $(q-p)$'th derivatives of the knot functions over T_0 evaluated at the intervals in *reg1*. Notice that the number of integrals and the choice of integrals is all from the basis over the second knot vector, but the knot functions are still chosen from the first knot vector. This variable is indexed in the same manner as the variable *ints* in Algorithm 9.

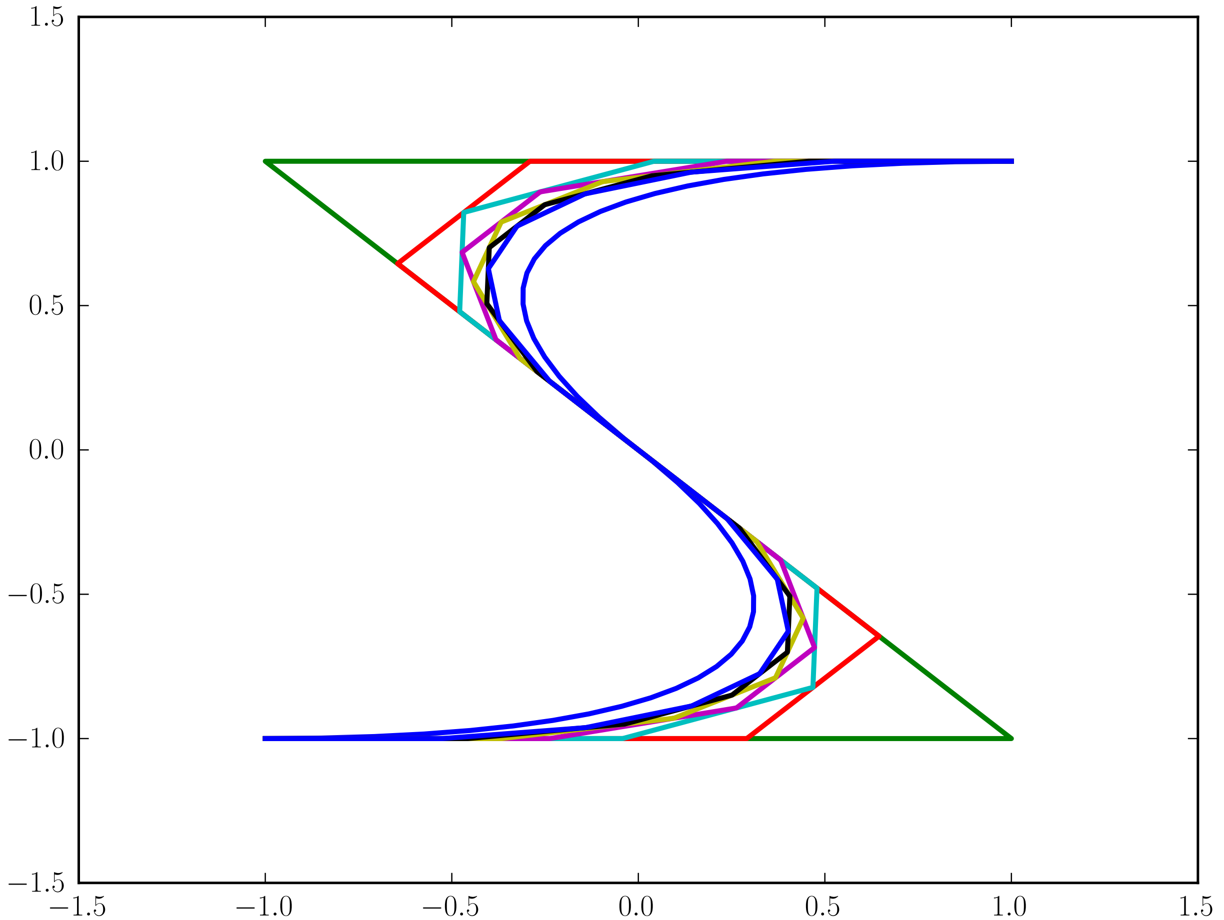

Figure 5.9: Degree elevations of a two dimensional GB-spline curve of degree 2 formed by a basis spanning trigonometric functions over the knot vector $[0, 0, 0, 0, .5, 1, 1, 1, 1]$.

- *rints1* will be the 0 through $(q-1)$th integral terms for the knot functions over T_1 evaluated at the intervals in *reg1*. If *ints* is the array of integral terms used to compute the local representations for the target basis, this is equivalent to $ints\,[:, q : -q]$.
- *npolys* will be a temporary array used to restructure *trans1*.
- *ntrans* will be a temporary array used to restructure *trans0*.
- *reg1lens* is the widths of the intervals in *reg1*.
- *pos* a boolean array where each entry is true if the corresponding entry in *reg1lens* is above a given tolerance.

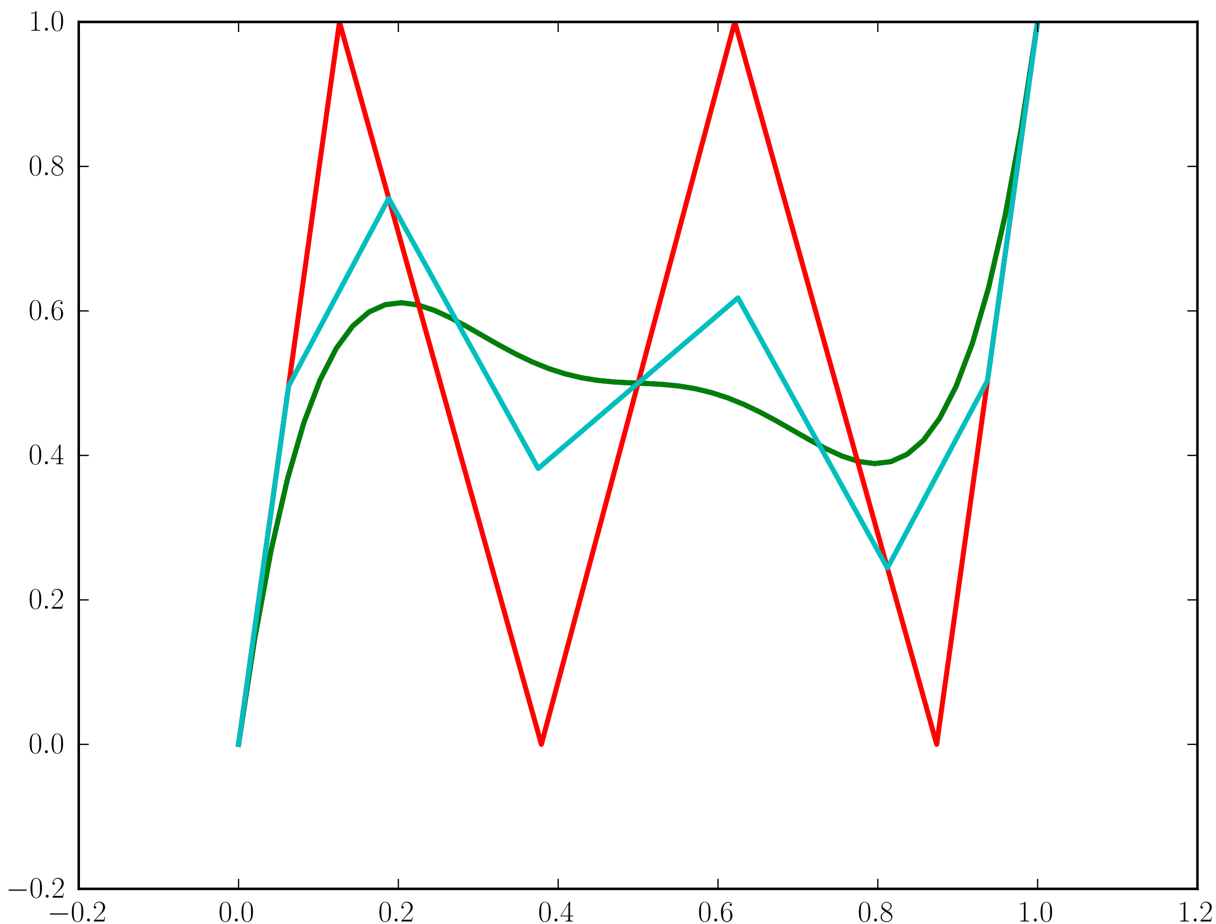

Figure 5.10: Knot insertion at .25 and .75 on a degree 4 GB-spline curve defined over $[0, 0, 0, 0, 0, .5, 1, 1, 1, 1, 1]$.

- *offset* is a temporary array of polynomial terms of the same shape as *polys0* once it is represented over the intervals in $regs1$.
- *offset1* and *offset2* will be arrays the same shape as *offset* that are used to compute *offset*.
- *ivals* and *nints* will be temporary arrays shaped like *rints0*. $diffs$ will be a temporary array with one less entry along the first axis.
- *lbases* is an array containing the local representations of the spline basis functions in the target basis for each interval in the active region of T_1. It is indexed first by interval in the active region, then by basis function index (indexing from 0 to q only

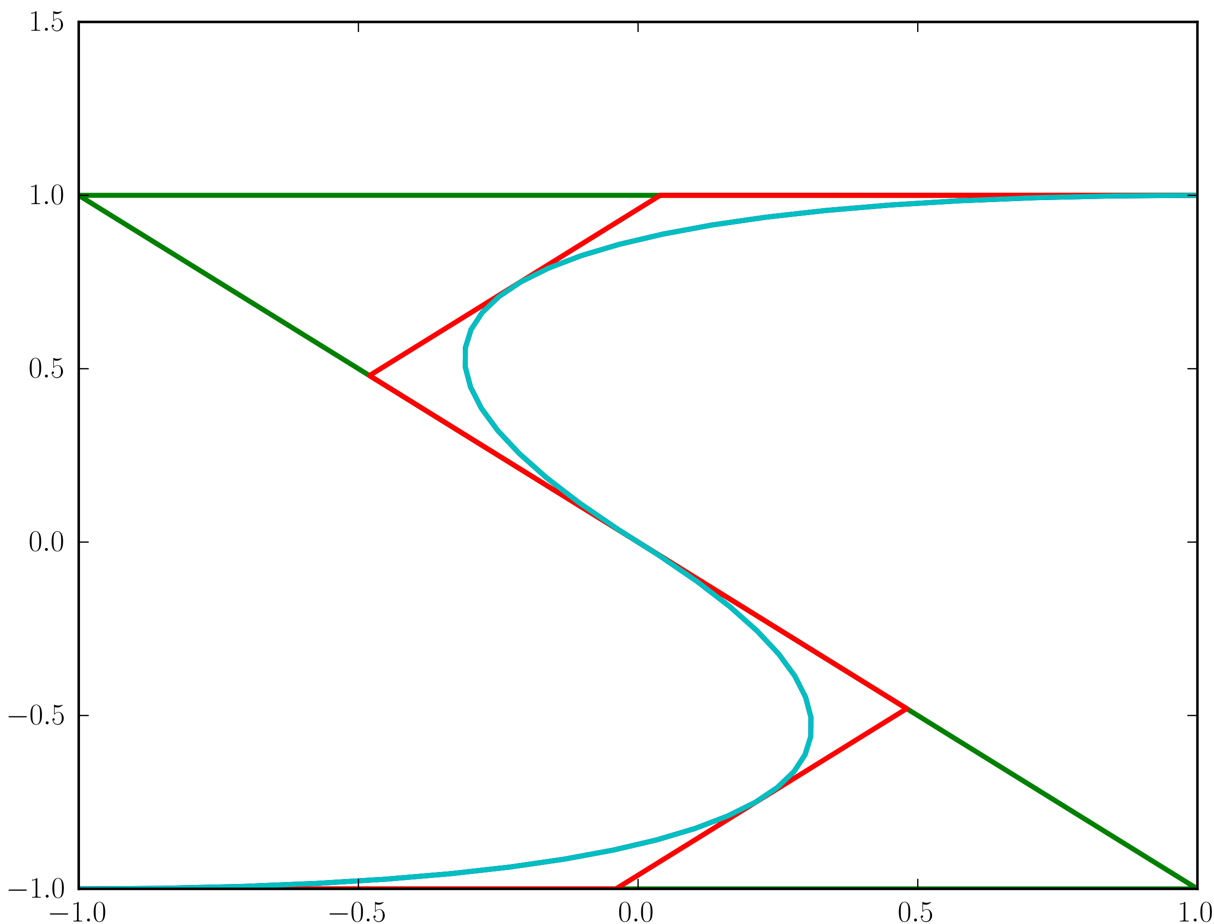

Figure 5.11: Knot insertion at .25 and .75 on a degree 2 GB-spline curve defined over $[0, 0, 0, 0, 0, .5, 1, 1, 1, 1, 1]$.

along the basis functions in that are nonzero over that interval), then by term in the local representation. The terms are ordered first by the polynomial terms with the last two entries corresponding to the coefficients for the integral terms from the knot functions.

- *lfunc* is an array containing the local representation of the piecewise function that we are representing as a spline with respect to the given basis. It is indexed first by basis function, then by term within the local representation with the terms in the local representation ordered the same way they are for *lbases*.

- *coefs* is an array containing the computed coefficients for the spline basis functions as

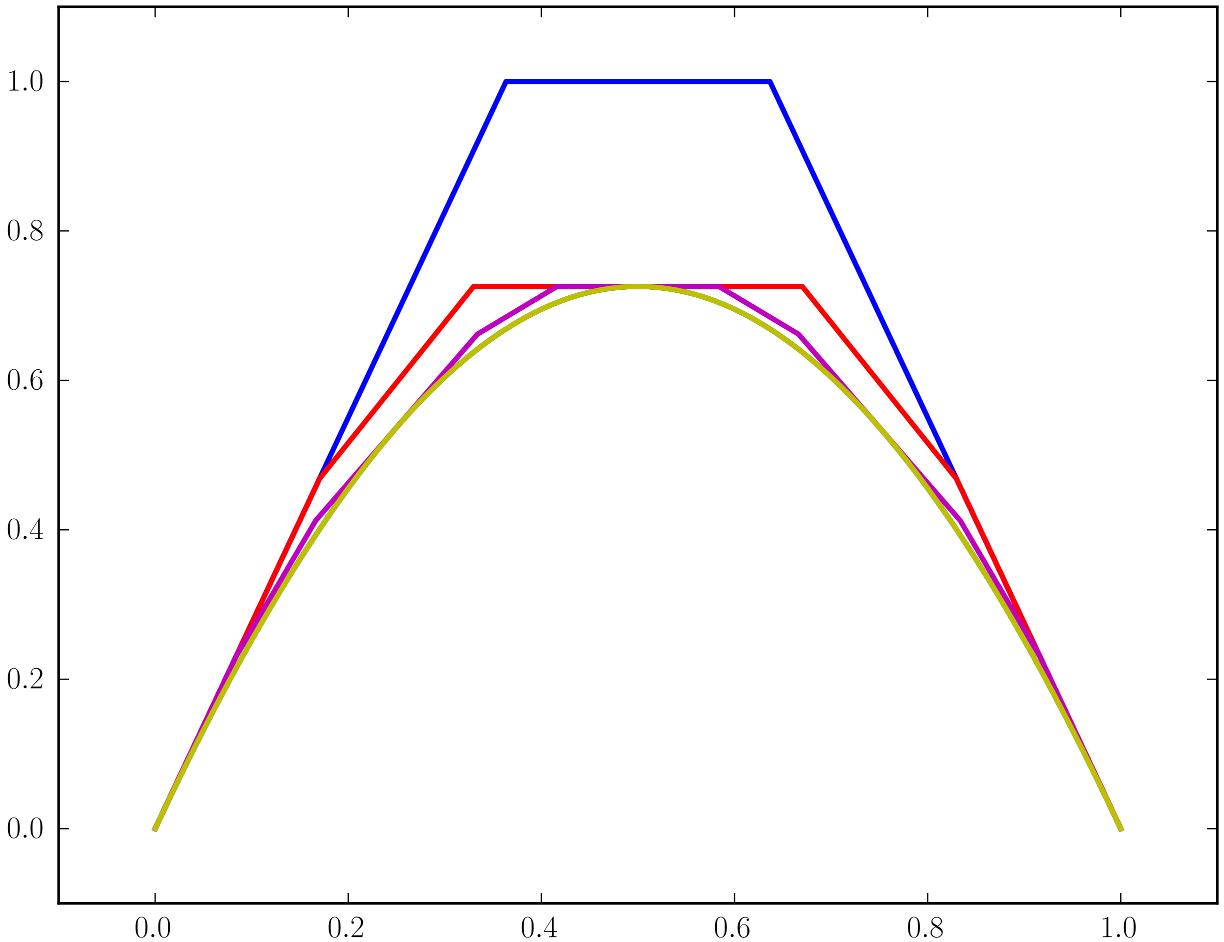

Figure 5.12: Insertion of 3 knots at .5 and then 3 knots at .25 and again at .75 on a degree 3 GB-spline curve defined using a Bernstein-like basis that spans the trigonometric functions $\cos\left(\frac{\pi}{2}t\right)$ and $\sin\left(\frac{\pi}{2}t\right)$.

they are computed over each interval. Rows in *coefs* corresponding to near 0 length intervals in *reg1* are set to NaN.

- *tol* will be a tolerance used to determine when intervals in a knot vector are short enough to be considered to have length 0. It will also be used to ensure the computed basis function coefficients do not differ significantly for the same basis function over different intervals.

- i and j will be indices use to loop through *reg0* and *reg1* respectively.

- *jidxs* will be a list of indices for *reg1* corresponding to the left endpoints of intervals of

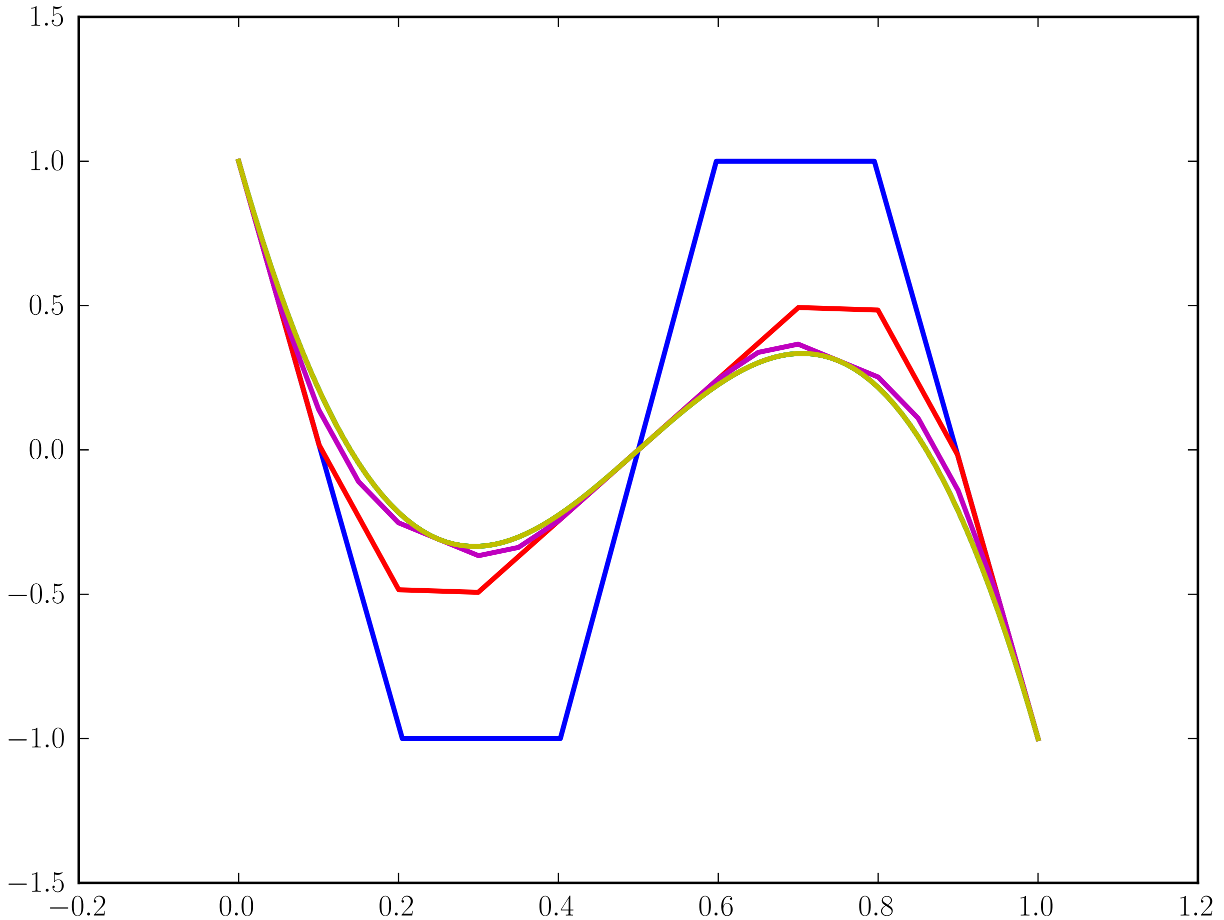

Figure 5.13: Insertion of 5 knots at .5 and then 5 knots at .25 and again at .75 on a degree 5 GB-spline curve defined using a Bernstein-like basis that spans the trigonometric functions $\cos\left(\frac{\pi}{2}t\right)$ and $\sin\left(\frac{\pi}{2}t\right)$.

nonzero length that all lie within a given interval in *reg0*.

- *endpts* will be a list of entries from *reg1* corresponding to all endpoints of the intervals that have an interval indexed by an entry in *jidxs*. Shared endpoints between intervals are not repeated here.

In addition, when performing refinement, the following routines will be used:

- *RefineLocal*: a routine that subdivides the polynomial terms in *polys0* so that they are represented over the intervals of *reg1*. It also modifies *trans1* so that it contains the coefficients for the integrals of the knot functions indexed by the intervals in *reg1*. The coefficients in *trans1* are not refined by this routine, they are only copied into

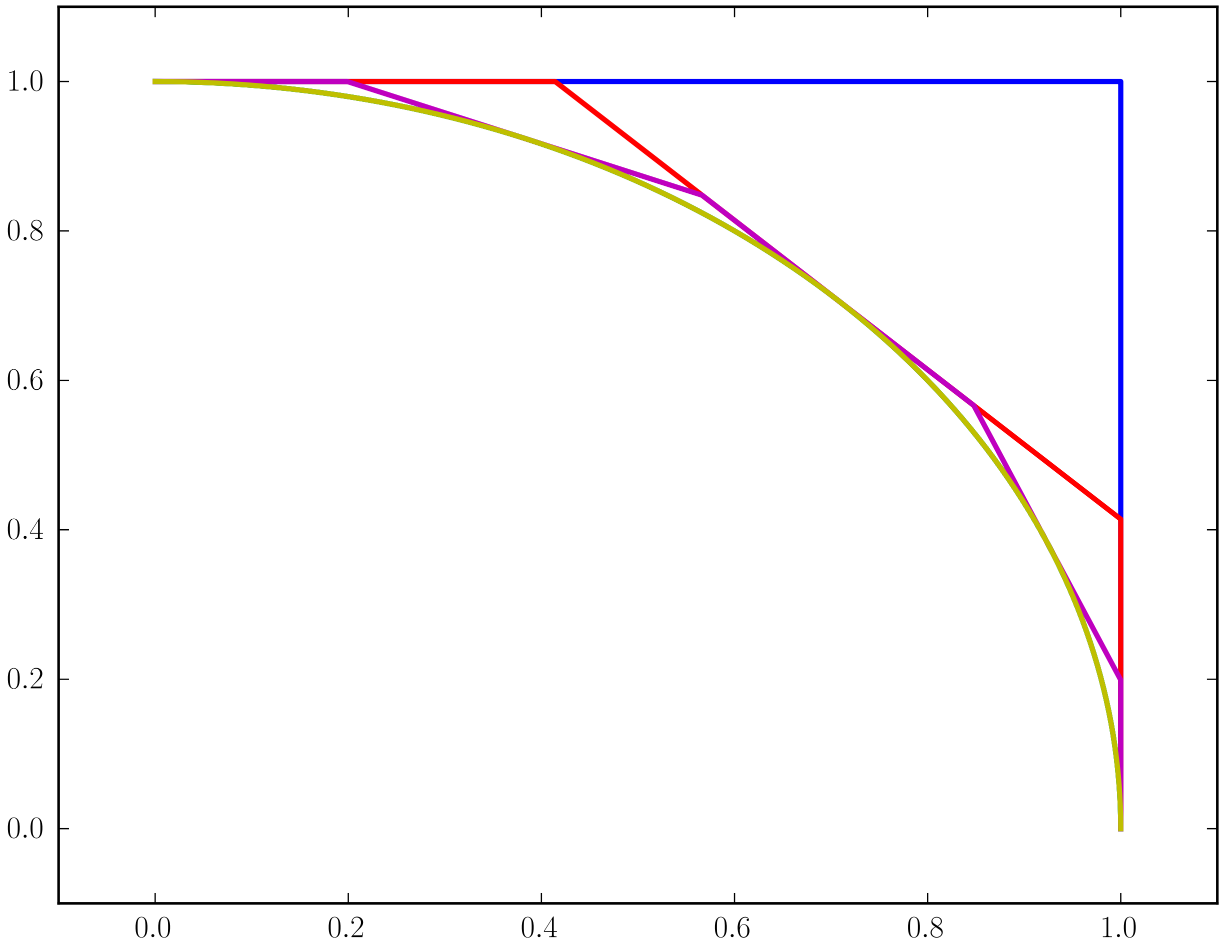

Figure 5.14: Insertion of 2 knots at .5 and then 2 knots at .25 and again at .75 on a degree 2 GB-spline curve that exactly represents a quarter circle and is defined using a Bernstein-like basis that spans the trigonometric functions $\cos\left(\frac{\pi}{2}t\right)$ and $\sin\left(\frac{\pi}{2}t\right)$.

a new array that is indexed by the intervals of *reg1* rather than by the intervals of *reg0*. The coefficients corresponding to an interval in *reg1* are simple the coefficients corresponding to the interval in *reg0* that contains it.

- *RestrictPoly*: a routine that restricts a polynomial over a given interval to a set of intervals and returns the new set of coefficients as an array.

- *ElevatePolys*: a routine to degree elevate an array of polynomial terms so they have a given degree.

- *LeftTaylorSeries*: a routine that computes the Taylor polynomial at the left endpoint

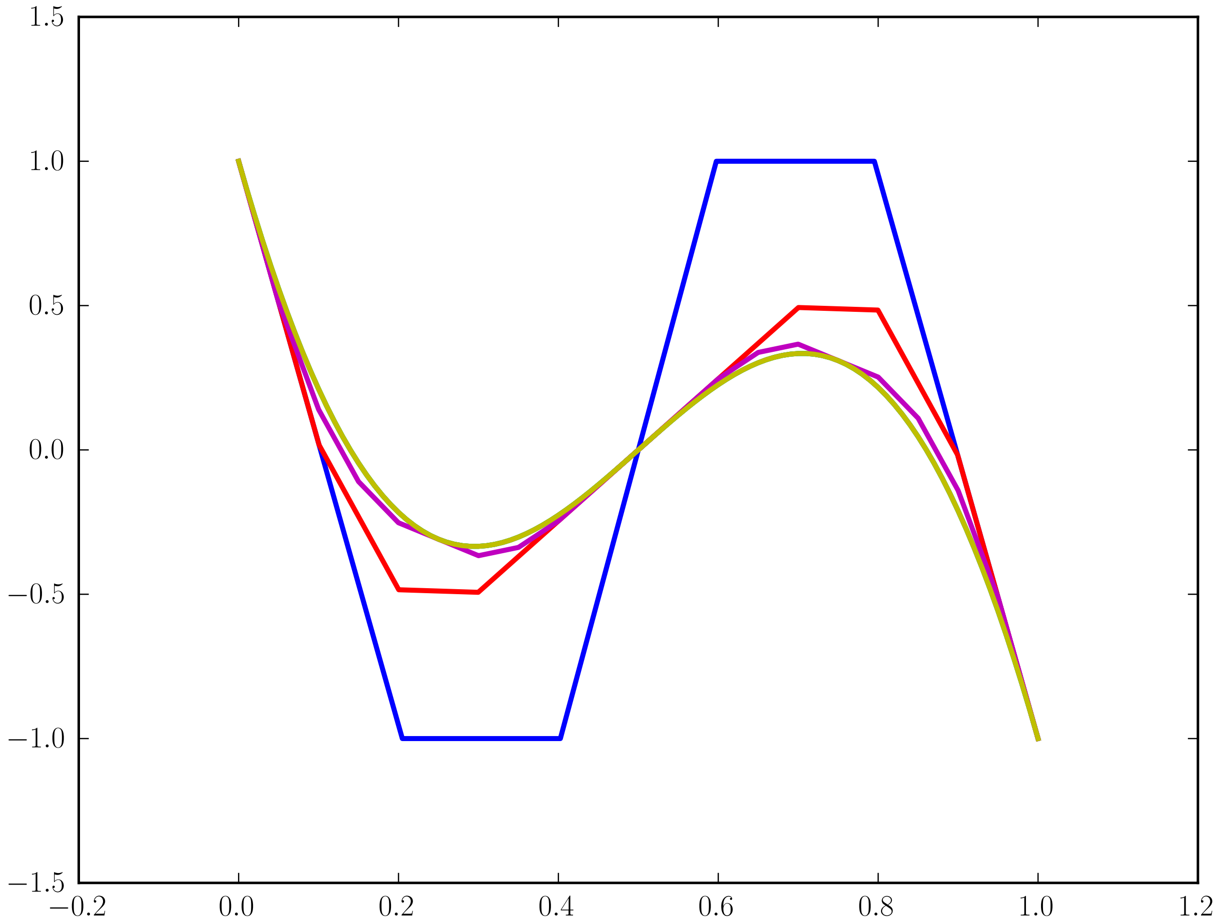

Figure 5.15: Insertion of 5 knots at .5 and then 5 knots at .25 and again at .75 on a degree 5 GB-spline curve defined using a Bernstein-like basis that spans the trigonometric functions $\cos\left(\frac{\pi}{2}t\right)$ and $\sin\left(\frac{\pi}{2}t\right)$.

of a given interval satisfying a given set of derivative values at that point.

- *RightTaylorSeries*: a routine that computes the Taylor polynomial at the right endpoint of a given interval satisfying a given set of derivative values at that point.

- *RepresentKnotFuncs*: a routine to change the coefficients in *trans0* so that they represent the terms from the knot functions in the given piecewise function in terms of the knot functions over *reg1* rather than the knot functions over *reg0*. This function also returns an array of polynomial terms to be added to *polys1* to compensate for different allowable choices for integral values for the given functions.

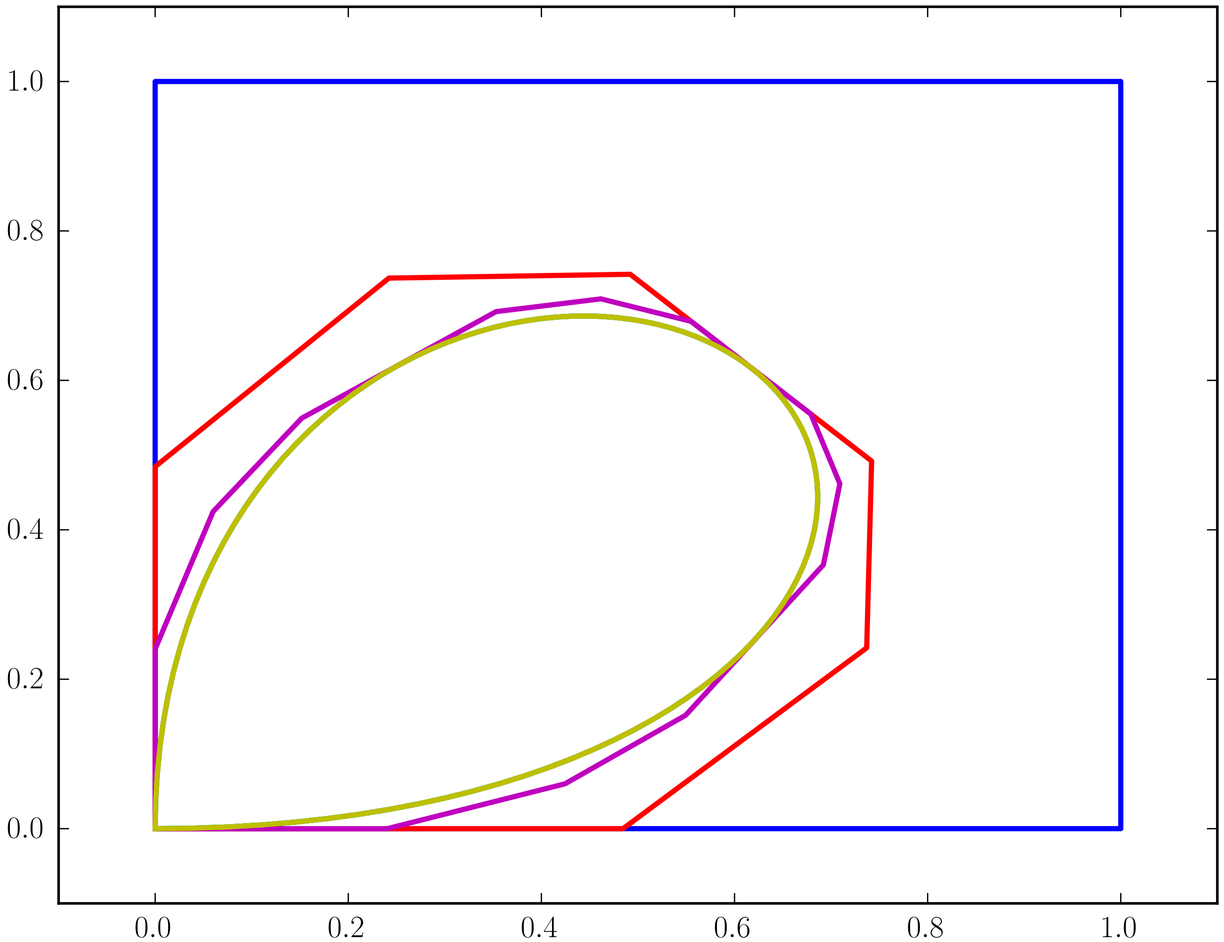

Figure 5.16: Insertion of 4 knots at .5 and then 4 knots at .25 and again at .75 on a degree 4 GB-spline curve defined using a Bernstein-like basis that spans the trigonometric functions $\cos\left(\frac{\pi}{2}t\right)$ and $\sin\left(\frac{\pi}{2}t\right)$.

- *FullReverseDiagonals*: a routine that takes a given array and, along the first two axes, forms a new array whose rows consist of the entries of all the reverse diagonals of full width. Here it will always be called on arrays that have more rows than columns, so each row of the output array will have the same shape as a row of the original array. This functions effectively as a method of translating between the piecewise representations of the basis functions and the variable *lbases*.

- *ReverseDiagonalAverages*: a routine that takes the averages along all reverse diagonals of a given 2 dimensional array, ignoring NaN entries.

The refinement algorithm is outlined in further detail in Algorithm 17.

Algorithm 17 Computing the control point representation for a given spline curve represented in piecewise form.

1: **procedure** RefineCurve($polys0, trans0, treg0, rints0, polys1, trans1, reg1, rints1, tol$)
2: ▷ Change the indexing of *polys0* and *trans0* to be indexed by the intervals in $regs1$.
3: ▷ Subdivide the polynomial terms so they are represented over the intervals in $regs1$.
4: $polys0, trans0 = \text{RefineLocal}\left(polys0, trans0, reg0, reg1, tol\right)$
5: ▷ Degree elevate the polynomials in *polys0* so that
6: ▷ they are the same degree as the polynomials in *polys1*.
7: $polys0 = \text{ElevatePolys}\left(polys0, \text{shape}\left(poys1\right)[-1] - 1\right)$
8: ▷ Compute the lengths of the intervals in $regs1$.
9: $reg1lens = reg1\left[1 :\right] - reg1\left[: -1\right]$
10: ▷ Find the intervals in *reg1* that have positive length
11: $pos = \left(reg1lens > tol\right)$
12: ▷ Represent the terms in *trans0* in terms of the local basis on *reg1*.
13: $trans0, \mathit{offset} = \text{RepresentKnotFuncs}\left(trans0, rints0, rints1, reg1lens, pos, tol\right)$
14: ▷ Add the polynomial part from the terms in *trans0* to *polys0*.
15: ▷ This makes it so that the function represented by *polys0* and *trans0* is now
16: ▷ completely rewritten in terms of the local bases used in *polys1* and *trans1*.
17: $polys0 \mathrel{+}= \mathit{offset}$
18: ▷ Construct the local representations of the positive basis functions
19: ▷ for each interval in *reg1*.
20: $lbases = \text{FullReverseDiagonals}\left(\text{concat}\left(polys1, trans1, -1\right)\right)$
21: ▷ Combine the local representations in *polys0* and *trans0* into a single array.
22: $lfunc = \text{concat}\left(polys0, trans0, -1\right)$
23: ▷ Allocate *coefs*.
24: $coefs$ = empty array of NaN's of the same shape as *lfunc*
25: ▷ Find the coefficients that represent each piece of the piecewise function
26: ▷ as a linear combination of the basis functions in the target basis.
27: $coefs\left[pos\right] = \text{solve}\left(\text{reindex}\left(lbases\left[pos\right], \left(0,2,1\right)\right), lfunc\left[pos, \ldots, None\right]\right)\left[\ldots, 0\right]$
28: ▷ Combine the local results for each interval to compute the desired basis coefficents.
29: ▷ Return the result.
30: **return** $\text{ReverseDiagonalAverages}\left(coefs, tol\right)$
31: **end procedure**

The auxiliary routine *RefineLocal* begins the translation of the local representations over *reg0* into local representations over *reg1*. It represents the polynomial terms over the new intervals. It also restructures the coefficients in *trans0* so that each row in *trans0* corresponds to an interval in *reg1*. The coefficients stored in the row corresponding to a given interval I_1 are the coefficients that were originally stored for the interval I_0 in *reg0* that contains I_1. *RefineLocal* performs these operations by iterating through the points in *reg0* and *reg1* and constructing the new arrays *npolys* and *ntrans* that are going to replace *polys0* and *trans0*. This auxiliary routine is shown in greater detail in Algorithm 18.

The auxiliary routine *RestrictPoly* is dependent on the polynomial representation used. It accepts a polynomial term, the endpoints for the interval over which it is currently defined, and the endpoints for the intervals to subdivide the polynomial term onto. It returns an array of polynomial terms containing the coefficients for the polynomial represented over each of the new intervals. If the polynomial basis used is the Bernstein basis, this corresponds to subdividing a Bernstein polynomial. If the polynomial basis used is the power basis shifted so that the starting point of each interval corresponds to 0 in the polynomial term, this operation corresponds to a left shift operation.

The routine *ElevatePolys* is also dependent on the polynomial representation used. For the case of polynomials in Bernstein form, a simple algorithm follows easily from Theorem 2.15. For power basis polynomials, this corresponds to appending 0's to the left of each set of coefficients.

The routine *RepresentKnotFuncs* provides a way to transition between different choices that could possibly be made for the knot functions and their corresponding integral values. First, since indefinite integrals are only well defined up to constant shifts, a method is needed to account for different choices for the integral terms. Polynomial terms can be used to account for these differences in integral values. In addition, if the i'th set of knot functions are $\cos\left(\frac{\pi}{2}t\right)$ and $\sin\left(\frac{\pi}{2}t\right)$ over the interval $[0, 1]$ and the target intervals are $\left[0, \frac{1}{2}\right]$ and $\left[\frac{1}{2}, 1\right]$ with knot functions $\cos\left(\frac{\pi}{2}t\right)$ and $\sin\left(\frac{\pi}{2}t\right)$, and $\cos\left(\frac{\pi}{2}\left(t - \frac{1}{2}\right)\right)$ and $\sin\left(\frac{\pi}{2}\left(t - \frac{1}{2}\right)\right)$

Algorithm 18 Restructuring *polys0* and *trans0* so they are indexed by interval in *reg1*. Subdivide the polynomial terms to represent them over the new intervals.

```
1: procedure RefineLocal(polys0, trans0, reg0, reg1)
2:     npolys = array of zeros of shape (shape(reg1)[0] − 1, shape(polys)[1])
3:     ntrans = array of zeros of shape (shape(reg1)[0] − 1, 2)
4:     i = 0
5:     j = 0
6:     ▷ Iterate through the intervals in reg0 and reg1.
7:     while i < shape(reg0)[0] do
8:         ▷ Skip the empty intervals in reg0.
9:         while reg0[i] − reg1[i − 1] < tol do
10:            i += 1
11:        end while
12:        ▷ Stop if iteration has already reached the end of reg0.
13:        if i = shape(reg0)[0] then
14:            break
15:        end if
16:        jidxs = empty list
17:        endpts = empty list
18:        ▷ Iterate through the intervals in reg1 that lie in the current interval in reg0.
19:        ▷ Add the corresponding indices and endpoints to jidxs and endpts
20:        ▷ if the interval has positive length.
21:        while reg0[i] − reg0[i − 1] < tol do
22:            if reg1[j + 1] − reg1[j] > tol then
23:                append j to jidxs
24:                append reg1[j] to endpts
25:            end if
26:            j += 1
27:        end while
28:        append reg1[j] to endpts
29:        ▷ Subdivide the polynomial term on the current interval onto the corresponding
30:        ▷ intervals in reg1.
31:        npolys[jidxs] = RestrictPoly(polys0[i − 1], reg0[i − 1 : i + 1], endpts)
32:        ▷ Copy the coefficients from trans0 into all corresponding rows of ntrans.
33:        ntrans[jidxs] = trans0[i − 1]
34:        i += 1
35:    end while
36:    return npolys, ntrans
37: end procedure
```

respectively (taking linear combinations to satisfy the needed endpoint values), on $\left[\frac{1}{2},1\right]$, the knot functions over $[0,1]$ are only equal to linear combinations of the knot functions on the interval $\left[\frac{1}{2},1\right]$.

On the other hand, since the polynomial terms used in the representation of a spline of degree q have only degree $q-2$, any function in the span of B_1 is uniquely determined by its 0 through $(q-1)$'th derivatives at each knot in the active region. To see this, observe that, over a given interval I_1 with endpoints in *reg1* and corresponding knot functions $\tilde{u}$ and $\tilde{v}$, any function f on I_1 within the span of the basis functions can be represented uniquely in the form $a\tilde{u}^{[q-1]}(t) + b\tilde{v}^{[q-1]}(t) + P(t)$. Since the polynomial term $P(t)$ has at most a degree of $q-1$, $f^{(q-1)} = a\tilde{u} + b\tilde{v}$. Since $\tilde{u}$ and $\tilde{v}$ form a Chebyshev space over I_1, they are linearly independent, so the coefficients a and b are uniquely determined. Now observe that $P(t) = f(t) - a\tilde{u}(t) - b\tilde{v}(t)$, so the 0 through $(q-1)$'th derivatives of f at the endpoints of I_1 uniquely determine the 0 through $(q-1)$'th derivatives of P. Since 0 through $(p-2)$'th derivatives at any given point uniquely determine a polynomial of degree $p-2$, P is uniquely determined by its derivatives at either endpoint of I_1, so it is certainly uniquely determined by the 0 through $(q-1)$'th derivatives at both endpoints.

The observation that the derivatives of a function at the endpoints of each interval uniquely define the local representation of the function also suggests a method for computing the needed representations of the integral terms in the local representations of the original piecewise function. Here, let u_i and v_i be the knot functions corresponding to the interval I_0 from *reg0* that contains I_1. Let $\tilde{u}_j$ and $\tilde{v}_j$ be the knot functions on I_1. Also let f_j be the restriction of the piecewise function being represented as a spline curve to the interval I_1. It is already known that $f_j = g_j + P_j$ where P_j is a polynomial term of degree $p-2$ and g_j is a linear combination of $u_i^{[p-1]}$ and $v_i^{[p-1]}$.

- Compute the 0 through $(q-1)$'th derivatives of g_j at the endpoints of each interval with endpoints *reg1*.
- Represent $g_j^{(q-1)}$ as $g_j^{(q-1)} = a_j\tilde{u}_j + b_j\tilde{v}_j$.

- Compute the 0 through $(q-2)$'th derivatives of the integral terms $a_j \tilde{u}_j^{[q-1]} + b_j \tilde{v}_j^{[q-1]}$ at the endpoints of each interval in *reg1*.
- Use the computed derivative terms to find the 0 through $(q-2)$'th derivatives of $g_j - a_j \tilde{u}_j^{[q-1]} - b_j \tilde{v}_j^{[q-1]}$.
- Use the computed derivatives for the difference term to compute the polynomial part in the representation of g_j in terms of the local bases used for B_1.

This process is shown in Algorithm 19.

Algorithm 19 Restructuring *polys0* and *trans0* so they are indexed by interval in *reg1*. Subdivide the polynomial terms to represent them over the new intervals.

1: **procedure** RepresentKnotFuncs$(trans0, rints0, rints1, reg1lens, pos, tol)$
2: ▷ Find the values of the 0 through $(q-1)$'th derivatives of each g_j.
3: $ivals = \text{MatMul}\left(rints0, trans0\left[\ldots, None\right]\right)$
4: ▷ Compute the representations of the $(q-1)$'th derivatives of g_j
5: ▷ in terms of the knot functions on the intervals in *reg1*.
6: $ntrans =$ an array of zeros the same shape as $trans0\left[\ldots, None\right]$
7: $ntrans\left[pos\right] = \text{solve}\left(rints1\left[0, pos\right], ivals\left[0, pos\right]\right)$
8: ▷ Compute the 0 through $p-1$'th derivatives of the integral terms
9: ▷ $a_j \tilde{u}_j^{[q-1]} + b_j \tilde{v}_j^{[q-1]}$.
10: $nints = \text{MatMul}\left(rints\left[1:\right], ntrans\right)\left[\ldots, 0\right]$
11: ▷ Take the difference between corresponding derivative terms.
12: $diffs = ivals\left[1:, \ldots, 0\right] - nints$
13: ▷ Compute the polynomial term that satisfies the given derivative constraints.
14: ▷ Average the Taylor series at both endpoints to combine the results nicely.
15: $offset0 = \text{LeftTaylorSeries}\left(\text{reindex}\left(diffs\left[:: -1, :, 0\right], (1, 0)\right), reg1lens\right)$
16: $offset1 = \text{RightTaylorSeries}\left(\text{reindex}\left(diffs\left[:: -1, :, 1\right], (1, 0)\right), reg1lens\right)$
17: $offset = .5 * \left(offset0 + offset1\right)$
18: **return** $ntrans\left[\ldots, 0\right], offset$
19: **end procedure**

The routines *LeftTaylorSeries* and *RightTaylorSeries* both accept an array of derivative values, indexed first by interval, then by derivative degree from 0 to $q-2$ and return a polynomial with the desired derivative values at the left and right endpoints of an interval of the given length with left endpoint at 0.

The routine *ReverseDiagonalAverages* is used to aggregate the results from the local computations over each interval in *reg1*. It takes a 2D array and computes the sum along

each reverse diagonal ignoring entries of NaN. Since it should be aggregating terms that will be fairly close together, it can also be used to raise an error if any real entry differs from the average by more than a negligible amount. When it is called in Algorithm 17, it is passed a tolerance for that purpose.

The helper routine *FullReverseDiagonals* depends on the array library used. Here it is only called on arrays with at least as many rows as columns, so it is sufficient to consider the cases where the length of a given reverse diagonal is equal to the length of a row of the original array. This function will also only operate on the first two axes of a given input array. In Python, it can be implemented easily using NumPy.

```python
def FullReverseDiagonals(a):
    """
    Get a view of all reverse diagonals that have the same number
    of entries as a row from the array 'a'.
    This routine preserves the order of entries row by row and reverses
    the order of the columns.
    """
    strides = (a.strides[0], a.strides[0]-a.strides[1]) + a.strides[2:]
    shape = (a.shape[0]-(a.shape[1]-1), a.shape[1]) + a.shape[2:]
    return np.lib.stride_tricks.as_strided(a[0,-1:], shape, strides)
```

To see how this indexing transformation works, let

$$a = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \\ a_{3,0} & a_{3,1} & a_{3,2} \\ a_{4,0} & a_{4,1} & a_{4,2} \\ a_{5,0} & a_{5,1} & a_{5,2} \end{bmatrix}$$

so

$$\text{FullReverseDiagonals}\,(A) = \begin{bmatrix} a_{0,2} & a_{1,1} & a_{2,0} \\ a_{1,2} & a_{2,1} & a_{3,0} \\ a_{2,2} & a_{3,1} & a_{4,0} \\ a_{3,2} & a_{4,1} & a_{5,0} \end{bmatrix}$$

For higher dimensional arrays, this indexing transformation is performed along the first two axes. Conceptually, this can be thought of the same as the operation on a, except that the entries shown above are subarrays resulting from fixing two indices rather than single entries of a two dimensional array. This indexing operation is used to transition between indexing local representations of basis functions first by basis function, then by interval within the support of each basis function, then by term in the representation, and indexing local representations first by interval in the active region of a spline, then by positive basis functions over that interval, then by term in the representation.

An additional benefit of the refinement algorithm presented here is that it provides a method for computing the Greville abscissae associated with a given spline basis. The Greville abscissae are the coefficients for the linear combination of the spline basis functions that correspond to the linear function $y = x$. These points are used to accurately represent 1-dimensional spline curves as 2-dimensional splines that are linear in one dimension. This representation is commonly used to choose the x-axis values at which to plot the control points of a given 1-dimensional spline curve. In the case of $p = 2$, these coefficients may not exist; however, if the spline basis spans degree 1 polynomial terms, Algorithm 17 provides a method to compute them.

When performing degree elevation and knot insertion, it may also be necessary to rewrite a curve in piecewise form before passing it to the routine defined in Algorithm 17. This can be done very easily using the routine *FullReverseDiagonals*, since it is essentially a transformation from indexing by basis function to indexing by interval. The version presented here does involve some redundant computation, but, for the sake of simplicity, we will not optimize it further. The process of forming a piecewise representation is shown in Algorithm 20.

Algorithm 20 Construct the piecewise representation of a spline curve given a set of control points and the corresponding set of basis functions.

1: **procedure** FORMPIECEWISE($cpts, oplys, trans$)
2: ▷ Construct the coefficients by multiplying the coefficients for each basis function by
3: ▷ the corresponding control points, reindexing so the coefficients are accessed by
4: ▷ interval, then summing the coefficients from the different basis functions.
5: $npolys = \text{sum}\left(\text{FullReverseDiagonals}\left(polys * cpts\left[:, None, None\right]\right), 1\right)$
6: $ntrans = \text{sum}\left(\text{FullReverseDiagonals}\left(trans * cpts\left[:, None, None\right]\right), 1\right)$
7: **return** $npolys, ntrans$
8: **end procedure**

CHAPTER 6. STABILITY OF EVALUATION

In Chapter 4 we introduced a local representation for GB-splines. In Chapter 5 we demonstrated how this local representation can be used to perform various refinement operations on GB-splines. Here we will address corresponding numerical concerns. The primary idea regarding the stability of the algorithms presented earlier is that they are least stable precisely in the cases where they are least useful. GB-splines are used to model functions that are not easily approximated by the usual polynomial basis used for spline curves. Generally, if a spline of high degree is needed, polynomial terms are sufficient to closely approximate additional functions that may need to be included in the basis.

The first important observation to be made is that the choices for the indefinite integrals of the knot functions are only unique up to a constant shift. If large constants are added to the integral terms, the information given by the values of the integral terms at the endpoints is obscured by floating point inaccuracies. The recursion in Definition 3.5 can easily fall prey to catastrophic cancellation.

Another important consideration is that, if the knot functions given as input to Algorithm 9 do not already satisfy the value constraints for every interval of nonnegligible length, the matrix

$$\begin{bmatrix} u_i\left(t_i\right) & v_i\left(t_i\right) \\ u_i\left(t_{i+1}\right) & v_i\left(t_{i+1}\right) \end{bmatrix}$$

must be relatively well-conditioned so that linear combinations of the knot functions that

satisfy the needed constraints can be computed.

Beyond these concerns, however is the fact that, for a spline basis of degree p on a given interval of positive length, if either function $u_i^{[p-1]}$ or $v_i^{[p-1]}$ can be well-aproximated by polynomials of degree $p-2$, the problem of representing a function in terms of the local basis $\left\{1, t, \ldots, t^{p-2}, u_i^{[p-1]}, v_i^{[p-1]}\right\}$ is inherently poorly conditioned.

For example, Consider the basis functions of a given degree p formed by the open knot vector formed by repeating 0 exactly $p+1$ times and then 1 exactly $p+1$ times. Then, following the recurrence in Definition 3.5, on $[0,1]$,

$$N_p^p = \frac{\int_0^t N_{p-1}^{p-1}(s)\, ds}{\int_0^1 N_{p-1}^{p-1}(s)\, ds}$$

Following this simpler recurrence for the last basis function, it is easily shown that

$$N_p^p = \frac{v^{[p-1]}(t) - \sum_{r=0}^{p-2} v^{[p-1-r]}(0) \frac{t^r}{r!}}{v^{[p-1]}(1) - \sum_{r=0}^{p-2} v^{[p-1-r]}(0) \frac{1}{r!}}$$

but this is precisely the Taylor series remainder term for the function $v^{[p-1]}$ taken at 0 normalized to take a value of 1 at 1. In the computation, as it has been presented here, however, the remainder term and $v^{[p-1]}(t)$ are computed separately and then the difference is taken between them. This can cause inaccuracies in floating point arithmetic if the size of the remainder term for the Taylor series is very small relative to the value of $v^{[p-1]}(t)$. Since, as the Taylor series comes closer to the values of $v^{[p-1]}(t)$, the value of the denominator

$$v^{[p-1]}(1) - \sum_{r=0}^{p-2} v^{[p-1-r]}(0) \frac{1}{r!}$$

also becomes very small, this makes it so that the coefficients for the polynomial part of the representation of N_p^p and the coefficient corresponding to $v^{[p-1]}(t)$ become very large. This effect on the values of the real and transcendental parts of is easily seen in practice and is shown in Figure 6.1. The effects of catastrophic cancellation are shown in Figure 6.2

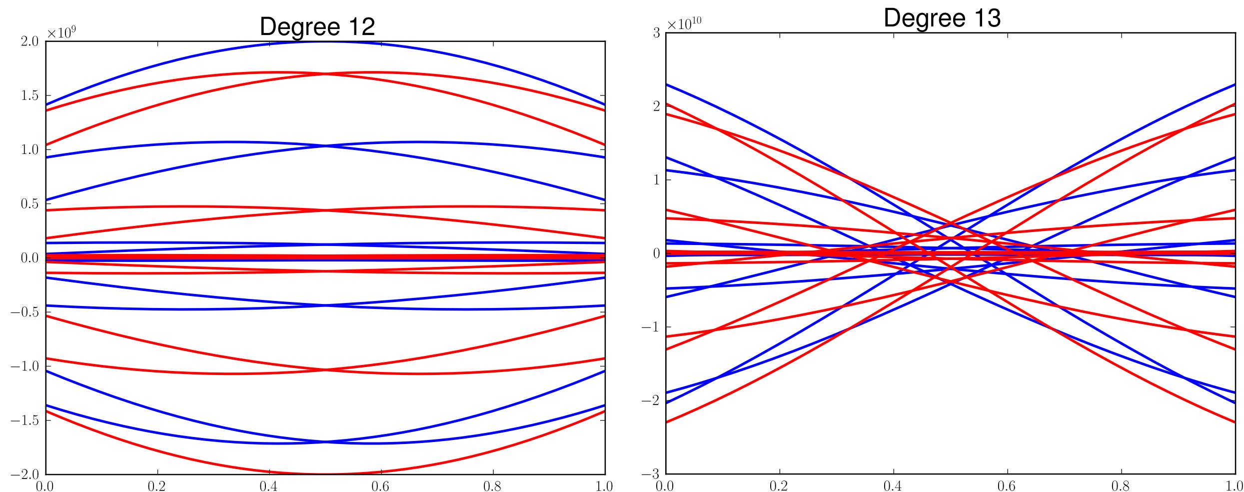

Figure 6.1: The sizes of the transcendental terms and polynomial terms for the generalized Bernstein basis of degrees 12 and 13 with $u = \cos\left(\frac{\pi}{2}t\right)$ and $v = \sin\left(\frac{\pi}{2}t\right)$. The transcendental parts are shown in blue and the polynomial parts are shown in red.

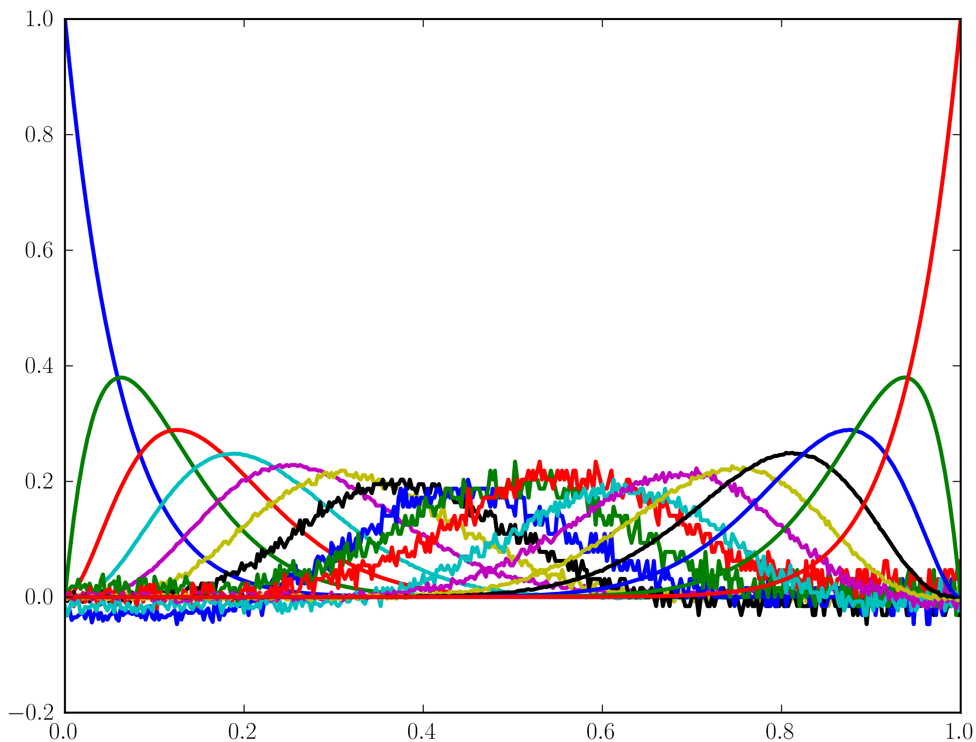

Figure 6.2: Breakdown in computation of trigonometric basis functions for $p = 16$.

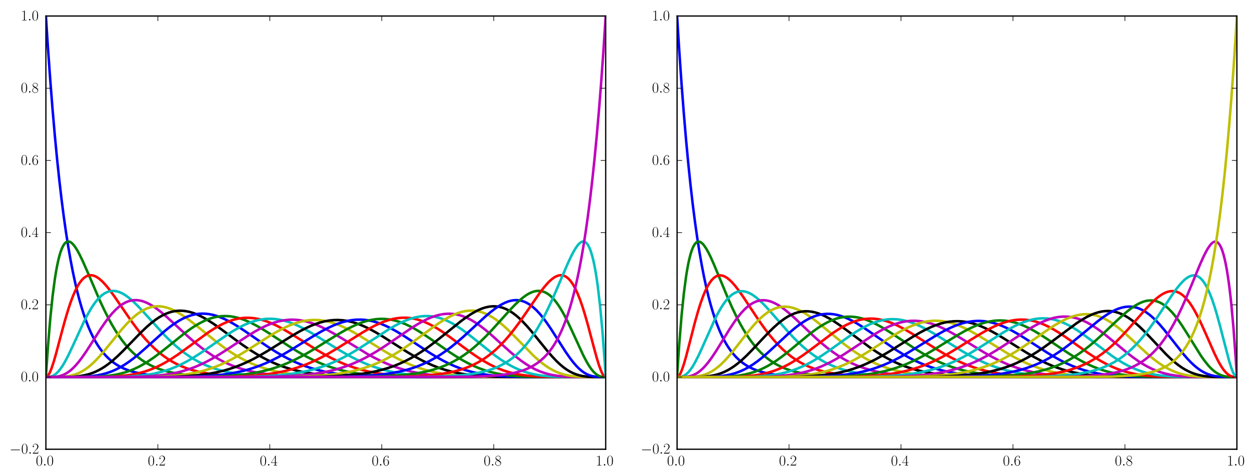

Figure 6.3: Generalized Bernstein bases of degree 26 with $u = \cos\left(\frac{\pi}{2}t\right)$ and $v = \sin\left(\frac{\pi}{2}t\right)$ computed using the tails of the Taylor series for sin and cos.

For the common cases of approximating transcendental functions, a more judicious choice for values of the constants of integration can mitigate this effect. For example, the constants of integration can be chosen such that $v_i^{[q]}(t_i) = u_i^{[q]}(t_{i+1}) = 0$ for every $q > 0$. If u and v were chosen as the $(p-1)$'th derivatives of specific functions, the terms $u_i^{[p-1]}$ and $v_i^{[p-1]}$ will be the values of the Taylor series remainder term for the series of degree $p-2$ centered at t_{i+1} for u and the series of degree $p-2$ centered at t_i for p. If the original function can be represented as a Taylor series, these remainder terms can be computed by estimating the tails of an infinite series rather than having to subtract two floating point numbers that are very close together. In practice, this approach does mitigate at least some of the inaccuracies of the basis construction, as can be seen in figure 6.3 which shows higher degree bases computed using Taylor series remainder terms for series that were both centered at 0.

The instabilities that arise from attempting to represent a function that is closely approximated by polynomial terms can also be seen when knots are very close together. In that case, the knot functions over the given interval may be so close to linear that using a linear knot functions to generate polynomial splines for that interval should be sufficient.

Chapter 7. Future Directions

Here we have presented algorithms for the evaluation and refinement of GB-spline curves via their piecewise representation. These algorithms provide a much more direct way of evaluating these spline curves than the recursive integral process given in the definition for GB-splines. They make practical computation with these curves vastly simpler in practice. They also make additional refinement operations easier to understand and practical to perform. In addition, these piecewise representations make it so that the cost of evaluating a given spline curve is bound primarily by the costs of finding the portion of the knot vector in which a given point lies and evaluating the functions spanned by the spline basis. Though these piecewise representations may suffer from some instabilities for cases where the local basis used is poorly conditioned, in general, the local bases introduced here offer a practical means for computation with GB-splines that is accurate precisely when these splines are needed to address the limitations of standard B-splines. These practical algorithms provide a means for CAD systems to exactly represent simple geometric objects like circles, spheres, hyperbolas, and hyperboloids in a way that can be further modified by designers.

There are a variety of ways in which the algorithms and structures here can be used for further research. The local bases on each interval used to construct each spline curve share some of the useful properties of the bases used in [27]. There, the process of inserting knots to represent a given B-spline curve as a piecewise polynomial was used to develop a local element structure that can, in turn, be used in Isogeometric Analysis (Finite Element Analysis on spline curves and surfaces, introduced in [28]). The local representations here can be used in a similar manner to provide element structures for Isogeometric Analysis.

Detailed analysis of the numerical stability of the algorithms presented here is also a topic where further work has yet to be done. Here we have outlined some practical ways to avoid instabilities that arise when working with a basis that is poorly conditioned. A more thorough analysis of these methods may provide greater insight into the structure of GB-spline curves and provide even more resilient versions of these algorithms.

The computational routines here can also be used to work toward developing more efficient methods for the evaluation of specific classes of GB-spline curves. They provide working examples that can be used to further study possible ways to provide better evaluation routines or subdivision methods for specific classes of GB-spline curves.

Methods for efficient evaluation for specific types of GB-spline curves may also be useful in building better root finding algorithms for different classes of transcendental curves.

The local structures used in this thesis also provide insight into possible generalizations of the approach used in [12] to local bases that are not generalized versions of the Bernstein basis. Further generalizations of the approach used there would provide insights into possible structures that could be used for adaptive refinement of finite element meshes that satisfy given smoothness constraints on their boundaries.

In practice, GB-spline bases look increasingly similar to B-spline bases for successively higher degrees of spline bases. A meaningful area for further work may come in explaining how, when, and how quickly this convergence occurs. Providing meaningful bounds on this convergence would also make it much easier to determine when polynomial basis functions can be used as a replacement for GB-spline basis functions.

In practice, the assumption that the knot functions form a Chebyshev space can also be loosened slightly. Higher degree splines may still satisfy many of the properties of GB-splines even if the knot functions do not form a Chebyshev space. For example, the basis on the knot vector $[0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1,]$ formed using the knot functions $\cos\left(\frac{7\pi}{2}t\right)$ and $\sin\left(\frac{7\pi}{2}t\right)$ on $[0, 1]$ appears to have many of the desirable properties needed for design for a Bernstein-like basis of degree 7.

Appendices

Appendix A. Conventions and Notation for Algorithms

Throughout the algorithms presented in this thesis, the following conventions and notation is used:

0-based indexing for arrays is assumed throughout.

Overwriting of function arguments is also allowed. If a variable is assigned to in a function, it is assumed that the original variable is only overwritten within the scope of the function. Array slices (exact slicing semantics are discussed later in this section) will be used for in-place assignment of values within arrays. In-place operators are also still assumed to modify the original object. When performing assignment, the right hand side of any expression is evaluated before assignment is performed. For example, $a = \frac{1}{2}(a + b)$ will compute the expression on the right hand side, then change the variable a to reference the result of the computation rather than it's previous values.

Arithmetic operations are assumed to broadcast (the same as NumPy). For arrays of the same shape, this is the same as performing the operations elementwise. Arrays are allowed to broadcast if corresponding entries are either equal or one of the two corresponding entries is 1. They are also allowed to broadcast if the shape of one array can be broadcast with the shape of the other by prepending ones to it. Arithmetic that is broadcast along an axis operates with the axis being broadcast taking constant values for all the indices along that axis. For example, arrays of shapes $(1, 3)$ and $(3, 1)$ would broadcast together to form an array of shape $(3, 3)$ containing the result of the arithmetic operation between every entry of the first array and every entry of the second array.

Broadcasting for solving linear systems will function in a similar fashion, operating on the last two axes of both arrays given. Solving the set of systems $Ax = b$ for x will be written as $\text{solve}(A, b)$. Matrix inversion will operate on the last two axes of a given array and broadcast along all other axes. Routines following proper broadcasting rules are available in NumPy as of version 1.8. A routine performing broadcasting matrix multiplication via numpy's einsum will be provided.

$+, -, *, /, \hat{}$	Elementwise arithmetic operators for scalars or arrays
$<, >, \leq, \geq, ==, !=$	Elementwise comparison operators on scalars or arrays
$+=, -=, *=, /=, \hat{}=$	In-place arithmetic operators for scalars or arrays
len	A function that returns the length of an array or list
deg	Function that takes the degree of a polynomial or spline
shape	Function returning the shape of a given array
concat	Concatenate two arrays along a given axis
sum	Sum an array along a given axis
solve	Solve a given linear system
inv	Compute a matrix inverse
reindex	Reorder the axes of a given array
bin	Function for the binomial coefficients of a given order
$\otimes$	Outer product of two vectors
$\langle \cdot, \cdot \rangle$	Inner product of two vectors
$\circledast$	Discrete convolution of two vectors
dcv	Discrete deconvolution of two vectors

Table A.1: Notational conventions used in algorithms.

Numpy-style array slicing notation will be used. Shapes, axes, and indices for multi-dimensional arrays will always be written row-first. This format is, for an array a with n dimensions, $a\,[start_1 : stop_1 : step_1, start_2 : stop_2 : step_2, \ldots, start_n : stop_n : step_n]$. This format means, along the i'th axis, take every $step_i$ elements starting from $start_i$ up to (but not including) $stop_i$. Along each axis, the notations $a[:]$, $a[start]$, $a[start : stop]$, $a[start :]$, $a[: stop]$, $a[start :: step]$, $a[: stop : step]$, and $a[:: step]$ are also allowed. When a value is excluded in this manner, $start$ is assumed to take a value of 0, $stop$ is assumed to take the value of an index one after the length of the axis of the array, and $step$ is assumed to take a value of 1. A single ":" applied to a given axis means to take the whole axis and is used only as a placeholder between other axes. A single number means to take only elements of the array that have that index on that axis (so, for a 2×3 array, $a[0, :]$ would take the 0'th row). If there is a single number corresponding to every axis, a scalar is returned. Negative values are taken to mean values counting back from the end of the axis ($a[-1]$ means the last element in the axis, $a[-2]$ the second to the last, etc.).

The slice [:] serves only as a placeholder when performing assignment to indicate whether the variable itself is being changed to reference a different array or the array is being modified

in place. For example $a[:] = 0$ writes 0 to all values of the given array while $a = b$ makes it so that the variable a references the same array originally referenced by b. If b is replaced by some expression, the expression is evaluated, a new array is created by the expression and the variable a is made to reference the new array.

In addition, indices for various axes may be omitted using the notations $a[i], a[..., i]$. All sets of indices for axes starting from the left are assumed to be addressing the 0'th, first, second, etc. axes. All sets of indices for axes following the ellipsis, starting from the right are assumed to be addressing the last, second to the last, third to the last, etc. axes.

New axes of length 1 may be added via slicing by using `None` in the indexing notation. This will add a new axis wherever the `None` appears in the indexing expression. For example, `a[None]` will add a new axis of length 1 at the beginning of the shape of a. `a[...,None]` will add a new axis of size 1 at the end of the shape of a.

Boolean arrays may be used as indices along all axes or along a single axis. If used along all axes, the shape of the boolean array must be the same as the shape of the original array. The resulting array will be a 1D array containing the values where the array is `True`. If used along a single axis, the resulting array will be the array of slices along the given axis where the boolean array is `True`. Inplace operators, when applied directly to these sorts of indexing operators are assumed to operate on the entries of the array being indexed.

When an array is reindexed, the ordering of the axes is changed according to some given permutation of the axes. For example, if a has three dimensions, $\text{reindex}(a, (1, 2, 0))$, would return a new array where $\text{reindex}(a, (1, 2, 0))[j, k, i] = a[i, j, k]$ for valid indices i, j, and k. In NumPy, reindexing is easily done via the *transpose* function.

Following Python's convention, functions are allowed to return multiple values or arrays. When a function returns multiple values, it will return them as a comma separated list. The returned object is assumed to be a tuple containing all values. When retrieving both values from a function, multiple assignment may be used, as in $a, b = f(c, d)$. For brevity, multiple assignment as in $a, b = c, d$ may also be used. It is assumed that $a, b = c, d$ is the same as

evaluating the expressions c and d, then assigning them to a and b respectively. For example, $a, b = b, a$ would swap the values of a and b.

The convention used here for convolutions is that entries are not wrapped around and indices where the arrays only partially overlap are also included. This is the convention followed by default in both NumPy and Matlab. For example, for arrays a and b of lengths m and n respectively, the first entry of $a \circledast b$ is $a[0] * b[0]$, the second entry is $a[0] * b[1] + a[1] * b[0]$, etc. The resulting array has length $m + n - 1$. This operation is equivalent to performing polynomial multiplication of polynomials with the coefficients from a and b and returning an array containing the coefficients of the resulting polynomial. It is also equivalent to taking the outer product of a and b, reversing the order of the rows, then taking the sum along each diagonal.

The convention for deconvolutions is also the same as is followed by NumPy and Matlab. The deconvolution of two arrays a and b is expected to return 2 arrays, q and r such that $b \circledast q + r = a$. r is expected to be an array of the same size as a, however, the first $\text{len}(a) - \text{len}(b) + 1$ entries of r are only in place to account for errors in floating point arithmetic. They should, in fact, be very close to 0. This operation is equivalent to polynomial division.

Appendix B. Code for Computation of Basis Coefficients

All code included here will work with NumPy 1.10 on Python 2.7 or 3.4. Other versions may work as well.

This code is intended primarily as a reference implementation of the new algorithms discussed in this thesis. It is designed primarily to match the original algorithms well. Error handling and argument checking has been mostly omitted, and the API consists only of functions that implement specific algorithms. For simplicity, we have used power basis polynomials in this section of the code, though it is relatively easy to use polynomials in Bernstein form as well.

```
# Copyright (c) 2015, Ian Daniel Henriksen
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
# are met:
#
# 1. Redistributions of source code must retain the above copyright
#    notice, this list of conditions and the following disclaimer.
#
# 2. Redistributions in binary form must reproduce the above copyright
#    notice, this list of conditions and the following disclaimer in the
#    documentation and/or other materials provided with the distribution.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
# A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
# HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
# LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
# DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
# THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
# (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
# OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

import numpy as np
from scipy.misc import factorial

# Integral axes are: (degree, interval_index, endpoint, u or v).
# Coefficient axes are: (basis_function, interval_index, u or v).
# For coefficients and integrals, u is indexed before v.
# Polynomial axes: (basis_function, interval_index, poly_coefs).
# TDiffs axes: (basis_function, interval_index)
# the variable 'd' in the loop is the degree of the basis being formed,
# not that of the previous basis.

# Wrap an axis into two new axes where the rows are just shifted windows
# of the original axis.
def Wrap(a, width, axis=0):
    """
    Use stride tricks to make a new array that views 'axis' of 'a'
    as two new axes where an entry of the first of the two new axes
    is a rolling window of 'width' length of the entries indexed
    along 'axis' of 'a'.
    """
    if (axis < -a.ndim) or (a.ndim <= axis):
        raise ValueError("'axis' entry out of bounds.")
    # Compute the shape of the desired output array.
    axis = axis % a.ndim
    shape = (a.shape[:axis] + (a.shape[axis]-width+1,width) + a.shape[axis+1:])
    # Make the strides of the new array the same along both expanded axes.
    strides = a.strides[:axis] + (a.strides[axis],) + a.strides[axis:]
    # Return an array of the given shapes and strides
    # viewing the memory of the original array.
    return np.lib.stride_tricks.as_strided(a, shape, strides)

def MatMul(a, b):
```

```
    """
60    Perform matrix multiplication between the arrays 'a' and 'b'
    following normal gufunc broadcasting rules.
    """
    return np.einsum('...ij,...jk', a, b)

65 def MakeDegreeOne(n):
    """
    Generate the polynomial and transcendental terms
    for a degree 1 basis with 'n' basis functions.
    """
70    trans = np.zeros((n, 2, 2))
    trans.reshape((n, 4))[:,1:3] = 1
    return np.empty((n, 2, 0)), trans

def ScaleKnotFuncs(ints, Tlens, tol=1E-8):
75    """
    Compute the linear combinations of the functions with
    indefinite integrals given in 'ints' such that the
    linear combinations satisfy the value constraints
    for each knot function. Return the rescaled version
80    of 'ints' and the matrices corresponding to the change
    of basis on each interval.
    """
    ints = ints.copy()
    pos = Tlens > tol
85    invs = np.zeros(ints.shape[1:])
    invs[pos] = np.linalg.inv(ints[0,pos])
    ints[:] = MatMul(ints, invs)
    return ints, invs

90 def ScaleTransCoefs(invs, trans):
    """
    Use the scaling matrices returned by 'ScaleKnotFuncs'
    to scale the coefficients for a given basis so that they
    are represented in terms of the original knot functions
95    rather than in terms of the scaled versions that satisfy
    the constraints over each interval in the knot vector.
    """
    trans[:] = MatMul(invs, trans[...,None])[...,0]

100 def PolyInt(polys, Tvals):
    """
    Integrate an array of polynomials represented in the power basis.
    The coefficients are assumed to be stored along the last axis.
    The coefficients for the indefinite integrals of the polynomial
105    terms are returned.
    Each polynomial is assumed to be defined on an integral of the length
    stored at the corresponding index in 'Tvals'.
    In the case of the power basis, the array 'Tvals' is not used.
    """
110    # Integrate a polynomial represented in the Power basis.
    pints = np.empty(polys.shape[:-1] + (polys.shape[-1]+1,))
    pints[...,:-1] = polys / np.arange(polys.shape[-1], 0, -1)
    pints[...,-1] = 0.
    return pints
115
def PolyVal(polys, Tvals, vals):
    """
```

```
    Evaluate each polynomial with coefficients indexed along the last
    axis of 'polys' defined over intervals of the lengths stored in
    'Tvals' at each corresponding value stored in vals.
    'Tvals' and 'vals' are expected to have a shape of 'polys.shape[:-1]'.
    """
    # Evaluate polys at vals, matching all the first dimensions
    pvals = polys[...,0].copy()
    for i in xrange(1, polys.shape[-1]):
        pvals *= vals
        pvals += polys[...,i]
    return pvals

def TransInt(trans, wints):
    """
    Given the transcendental coefficients from a given set of basis functions,
    compute the indefinite integral of the transcendental terms.
    Also return the constant terms that are computed from the left boundary
    of each interval.
    """
    consts = -(trans * wints[:,:,0]).sum(axis=-1)
    vals = (trans * wints[:,:,1]).sum(axis=-1) + consts
    return vals, consts

def IntegrateSupports(Tvals, pints, trans, wints):
    """
    Integrate each basis function over its support.
    Also return the constant terms from the left endpoints
    of each transcendental integral so they can be used
    when computing the polynomial terms for the new basis.
    """
    vals, consts = TransInt(trans, wints)
    # Tvals lines values up with first two axes.
    # Need to get the values from the polynomials along the last axis.
    vals += PolyVal(pints, Tvals, Tvals)
    # Return the final sum, evaluated over all
    # the intervals for each basis function.
    # Line containing einsum equivalent to:
    return vals.sum(axis=-1), consts

def OffsetDifferences(pints, trans):
    """
    Given the integrated and normalized values
    of the indefinite integrals of each basis function
    over each interval, take the differences
    between these intermediate terms to construct a
    new basis. This function does not account for the
    constant offsets in the polynomial term, nor does it
    account for the constant terms that come from evaluating
    the new basis function on each previous interval.
    """
    n, nints, dmin = pints.shape
    npolys = np.zeros((n-1, nints+1, dmin))
    ntrans = np.zeros((n-1, nints+1, 2))
    npolys[:,:-1] += pints[:-1]
    npolys[:,1:] -= pints[1:]
    ntrans[:,:-1] += trans[:-1]
    ntrans[:,1:] -= trans[1:]
    return npolys, ntrans
```

```
def OffsetConstants(polys, consts):
    """
    Apply a constant offset inplace to the polynomials
    stored in polys.
    This depends on polynomial representation, but is
    independent of the length of the interval used in
    representing each polynomial.
    """
    # This depends on polynomial representation.
    # Here, assume power basis representation with higher
    # degree terms at the beginning.
    polys[...,-1] += consts

def AddOnes(polys, pos):
    """
    This function adds ones to the last interval of some
    of the basis functions to account for the case that
    one of the basis functions used in the computation
    of the new basis function is equal to 0.
    """
    # Add 1 to the polynomial part of each basis function
    # that has an index that corresponded to a degenerate
    # basis function for the previous degree.
    # Only do this after the support of the degenerate
    # basis function should have ended, i.e. on the
    # last interval where the current basis function is nonzero.
    # In the Power polynomial case, this means just adding 1 to
    # the constant term of the polynomial part over that interval.
    polys[~pos[:-1],-1,-1] += 1

def ConnectBoundaries(polys, trans, wints, Tvals):
    """
    This function adds in the constant terms that come
    from evaluating each basis function at the endpoint
    of each interval.
    """
    vals = PolyVal(polys[:,:-1], Tvals, Tvals[:,:-1])
    vals += (trans[:,:-1] * wints[:-1,:,1]).sum(axis=-1)
    OffsetConstants(polys[:,1:], vals.cumsum(axis=1))

def BasisCoefs(T, ints, tol=1E-8):
    """
    Construct the local representation of a GB-spline
    basis in terms of a polynomial term with two transcendental
    terms on each interval.
    """
    p = ints.shape[0]
    Tlens = T[1:] - T[:-1]
    Tvals = Wrap(Tlens, 2)
    ints, scal = ScaleKnotFuncs(ints, Tlens, tol=tol)
    polys, trans = MakeDegreeOne(T.size - 2)
    for d in xrange(2, p+1):
        pints = PolyInt(polys, Tvals)
        wints = Wrap(ints[d-1], d)
        deltas, consts = IntegrateSupports(Tvals, pints, trans, wints)
        OffsetConstants(pints, consts)
        pos = (T[d:] - T[:-d]) > tol
        deltas = deltas[pos,None,None]
        pints[pos] /= deltas
```

```
        trans[pos] /= deltas
        polys, trans = OffsetDifferences(pints, trans)
        AddOnes(polys, pos)
        Tvals = Wrap(Tlens, d+1)
240     ConnectBoundaries(polys, trans, wints, Tvals)
    ScaleTransCoefs(Wrap(scal, p+1), trans)
    return polys, trans

def expand_comp(a):
245 """
    Expand a complex array into a floating point array
    with an additional last axis used to index the real
    and complex parts of the original array.
    """
250 assert a.dtype == np.complex128
    return a.view(dtype='d').reshape(a.shape[:-1] + (-1,2))

def trig_ders(p, T, c=np.pi*.5, tol=1E-11):
    """
255 Compute the derivatives of the trigonometric
    functions cos(c*t) and sin(c*t).
    'T' is the knot vector over which to compute the derivatives.
    'p' is the degree of the basis to be constructed.
    'p' is also equal to one more than the number of derivatives needed.
260 Derivatives are returned the format expected to be used for
    the variable 'ints' later in the basis construction algorithm;
    i.e., the various derivatives should be indexed from highest
    to lowest along the first axis, the different intervals in
    the knot vector should be indexed algong the second axis,
265 the first and second endpoints of each interval along the third axis,
    and the function being differentiated along the last axis
    with cosine coming before sine.
    These derivatives are represented in shifted form so that
    the input values from each interval are assumed to begin at 0.
270 In other words, on the i'th interval in T, the knot functions u and v
    are equal to cos(c * (t-t_i)) and sin(c * (t - t_i)) respectively.
    """
    # This computation computes the derivatives as various phase shifts
    # of the original sin and cos functions.
275 # The complex exponential function is used to compute both
    # the sine and cosine terms simultaneously.
    # First allocate the output array.
    ders = np.empty((p, T.size-1, 2, 2))
    # Compute the lengths of the different intervals.
280 Tlens = T[1:] - T[:-1]
    # The phase shifts used to get the different derivatives.
    angles = np.arange(p-1, -1, -1)
    # The constant coefficients that come from c.
    coefs = (c)**(angles)
285 ders[...,0,:] = expand_comp(coefs*np.exp(.5j*np.pi*angles))[:,None]
    ders[...,1,:] = expand_comp(coefs[:,None]*np.exp(c*1.0j*Tlens[None]+
                                                   .5j*np.pi*angles[:,None]))
    ders[np.absolute(ders)<tol] = 0
    return ders
290
def trig_ders_plain(p, T, c=np.pi*.5):
    """
    Compute the derivatives of  up to degree p-1 at the points in T
    and return them sorted first by degree, then by point in T, then by
```

```
    function (cosine first, then sine).
    """
    # The phase shifts used to get the different derivatives.
    angles = np.arange(p-1, -1, -1)
    # The constant coefficients that come from c.
    coefs = (c)**(angles)
    return expand_comp(coefs[:,None]*np.exp(c*1.0j*T[None]+
                                           .5j*np.pi*angles[:,None]))

def ReverseDiagonalView(a, i):
    """
    Take the 'i'th reverse diagonal.
    This function is useful when evaluating basis functions
    based on the knot vectors.
    """
    # This may return a copy, read-only view,
    # or a view, depending on the version of numpy,
    # but, as far as the correctness of this algorithm
    # is concerned, The distinction does not matter
    # in this case.
    # A view is preferable only because no copy occurs.
    return np.rollaxis(a[:,::-1].diagonal(a.shape[1] - i - 1), -1)

def eval_basis(polys, trans, T, knot_funcs, t_vals):
    """
    Evaluate a basis given the polynomial terms 'polys', the
    transcendental terms 'trans', the knot vector 'T',
    and the knot functions 'knot_funcs' at the parameter
    values 't_vals'.
    """
    assert polys.shape[0] == trans.shape[0]
    p = polys.shape[-1] + 1
    assert len(T) - p - 1 == polys.shape[0]
    assert len(T) == len(knot_funcs) - 1
    bins = np.digitize(t_vals, T) - 1
    out = np.zeros((polys.shape[0], t_vals.size), order='F')
    for j, (i, t) in enumerate(zip(bins, t_vals)):
        if i == -1:
            continue
        if i == len(T) - 1:
            if T[-1] == T[-p-1]:
                out[-1,j] = 1
            continue
        loc_polys = ReverseDiagonalView(polys, i)
        loc_trans = ReverseDiagonalView(trans, i)
        f, l = max(0, i+1-trans.shape[1]), min(trans.shape[0], i)
        out[f:l+1,j] = PolyVal(loc_polys, None, t - T[i])
        out[f:l+1,j] += knot_funcs[i][0](t-T[i]) * loc_trans[:,0]
        out[f:l+1,j] += knot_funcs[i][1](t-T[i]) * loc_trans[:,1]
    return out

def eval_trig_basis(polys, trans, T, t_vals, c=.5*np.pi):
    """
    Evaluate a basis given the polynomial terms 'polys', the
    transcendental terms 'trans', and the knot vector 'T',
    at the parameter values 't_vals' using 'cos(c*t)' and
    'sin(c*t)' as the knot functions over each interval.
    """
    assert polys.shape[0] == trans.shape[0]
```

```
    p = polys.shape[-1] + 1
    assert len(T) - p - 1 == polys.shape[0]
    bins = np.digitize(t_vals, T) - 1
    out = np.zeros((polys.shape[0], t_vals.size), order='F')
    for j, (i, t) in enumerate(zip(bins, t_vals)):
        if i == -1:
            continue
        if i == len(T) - 1:
            if T[-1] == T[-p-1]:
                out[-1,j] = 1
            continue
        loc_polys = ReverseDiagonalView(polys, i)
        loc_trans = ReverseDiagonalView(trans, i)
        f, l = max(0, i+1-trans.shape[1]), min(trans.shape[0], i)
        out[f:l+1,j] = PolyVal(loc_polys, None, t - T[i])
        out[f:l+1,j] += np.cos(c*(t-T[i])) * loc_trans[:,0]
        out[f:l+1,j] += np.sin(c*(t-T[i])) * loc_trans[:,1]
    return out

def eval_trig_spline(cpts, t_vals, polys, trans, T, c=.5*np.pi):
    """
    Convenience function for evaluating trigonometric spline.
    """
    basis = eval_trig_basis(polys, trans, T, t_vals, c=c)
    return MatMul(cpts.T[...,None,:], basis)[...,0,:].T

def FullReverseDiagonals(a):
    """
    Get a view of all reverse diagonals that have the same number
    of entries as a row from the array 'a'.
    This routine preserves the order of entries row by row and reverses
    the order of the columns.
    """
    assert a.shape[0] >= a.shape[1]
    strides = (a.strides[0], a.strides[0]-a.strides[1]) + a.strides[2:]
    shape = (a.shape[0]-(a.shape[1]-1), a.shape[1]) + a.shape[2:]
    return np.lib.stride_tricks.as_strided(a[0,-1:], shape, strides)

def ReverseDiagonalAverages(a, tol=1E-8):
    """
    Take the average along the reverse diagonals of the given array.
    Ignore NaN values.
    """
    n = a.shape[0]+a.shape[1]-1
    avgs = np.empty(n)
    for i in xrange(n):
        coefs = ReverseDiagonalView(a, i)
        coefs = coefs[~np.isnan(coefs)]
        assert coefs.size > 0
        avgs[i] = coefs.mean()
        assert (np.absolute(coefs - avgs[i]) < tol).all()
    return avgs

def ShiftPolynomials(polys, consts):
    """
    Shift the polynomial terms in polys to the left by
    the corresponding constants in consts.
    """
    npolys = np.zeros_like(polys)
```

```
    npolys[...,-1] = polys[...,-1]
    pws = (consts[...,None])**np.arange(polys.shape[-1])
    bins = np.zeros(polys.shape[-1])
    bins[0] = 1
    for i in xrange(1, polys.shape[-1]):
        bins[1:i+1] += bins[:i].copy()
        npolys[...,-i-1:] += polys[...,-i-1,None] * (pws[:,:i+1] * bins[:i+1])
    return npolys

def LeftTaylorSeries(ders, lens):
    return (ders / factorial(np.arange(ders.shape[-1])))[::-1]

def RightTaylorSeries(ders, lens):
    return ShiftPolynomials(LeftTaylorSeries(ders, lens), lens)

def Repeat(a, n, axis=0):
    """
    Use stride tricks to repeat an array along a given axis of a new array.
    """
    shape = a.shape[:axis] + (n,) + a.shape[axis:]
    strides = a.strides[:axis] + (0,) + a.strides[axis:]
    return np.lib.stride_tricks.as_strided(a, shape=shape, strides=strides)

def RestrictPoly(poly, int_, oints):
    """
    Restrict the domain of a given polynomial to a given set of intervals.
    """
    oints = np.array(oints)
    poly = Repeat(poly, len(oints)-1)
    return ShiftPolynomials(poly, oints[:-1] - oints[:1])

def RefineLocal(polys0, trans0, reg0, reg1, tol=1E-8):
    """
    Represent the piecewise polynomial over the intervals
    in the active region reg0 with the piecewise polynomial
    terms stored as rows in 'polys' as a piecewise polynomial
    term over the refined (or identical) active region reg1.
    """
    # Extending the polynomial outside the domain is not well-defined,
    # so this must be done only when the endpoints coincide.
    assert abs(reg1[0] - reg0[0]) < tol
    assert abs(reg1[-1] - reg0[-1]) < tol
    npolys = np.zeros((len(reg1)-1, polys0.shape[1]))
    ntrans = np.zeros((len(reg1)-1, 2))
    # Iterate through intervals in reg0 by right endpoint.
    # Iterate through intervals in reg1 by left endpoint.
    # Index within reg1 for loop over reg0.
    j = 0
    i = 1
    while i < len(reg0):
        while reg0[i] - reg0[i-1] < tol:
            i += 1
        if i == len(reg0):
            break
        jidxs = []
        endpts = []
        while reg0[i] - reg1[j] > tol:
            if reg1[j+1] - reg1[j] > tol:
                jidxs.append(j)
```

```
                endpts.append(reg1[j])
            j += 1
        endpts.append(reg1[j])
475     assert len(jidxs) >= 1
        npolys[jidxs] = RestrictPoly(polys0[i-1], reg0[i-1:i+1], endpts)
        ntrans[jidxs] = trans0[i-1]
        i += 1
    return npolys, ntrans
480
def FormPiecewise(cpts, polys, trans):
    """
    Compute the coefficients for the piecewise representation of a spline
    curve given the control points and the piecewise representations of
485 the corresponding set of basis functions.
    """
    cpts = cpts[:,None,None]
    npolys = FullReverseDiagonals(polys * cpts).sum(axis=1)
    ntrans = FullReverseDiagonals(trans * cpts).sum(axis=1)
490 return npolys, ntrans

def RepresentKnotFuncs(trans0, rints0, rints1, reg1lens, pos, tol=1E-8):
    """
    Represent the knot functions from the piecewise function in
495 terms of the local bases used to construct the target basis.
    This only works if the local bases for the target basis
    contain the knot functions from the piecewise function in their span.
    """
    ivals = MatMul(rints0, trans0[...,None])
500 ntrans = np.zeros_like(trans0[...,None])
    ntrans[pos] = np.linalg.solve(rints1[0,pos], ivals[0,pos])
    nints = MatMul(rints1[1:], ntrans)[...,0]
    diffs = ivals[1:,...,0] - nints
    offset0 = LeftTaylorSeries(diffs[::-1,:,0].T, reg1lens)
505 offset1 = RightTaylorSeries(diffs[::-1,:,1].T, reg1lens)
    assert (np.absolute(offset0 - offset1) < tol).all()
    offset = .5 * (offset0 + offset1)
    return ntrans[...,0], offset

510 def ElevatePolys(polys, deg):
    """
    Degree Elevate an array of polynomial coefficients.
    """
    dif = deg + 1 - polys.shape[-1]
515 if dif:
        return np.append(np.zeros(polys.shape[:-1]+(dif,)), polys, -1)
    return polys

def RefineCurve(polys0, trans0, reg0, rints0,
520             polys1, trans1, reg1, rints1, tol=1E-6):
    """
    Given a spline represented in piecewise form, compute its control
    point representation with respect to a basis that spans it.
    'polys0' and 'trans0' are the coefficients for the piecewise
525 parts of the piecewise function.
    'reg0' is the portion of the knot vector corresponding to the active region
    for the initial spline curve now represented as a piecewise function.
    'polys1' and 'trans1' are the polynomial and integral terms for the
    target basis.
530 'reg1' is the portion of the knot vector corresponding to the active
```

```
    region for the target basis.
    'rints1' contains the integral terms for the knot functions used to
    form the target basis evaluated at the endpoints of each interval in
    'reg1'.
535 'rints0' contains the integral terms for the knot functions used to form
    the original spline curve evaluated at the endpoints of each interval in
    'reg1'.
    Both 'rints0' and 'rints1' must contain the 0 through (p-1)'th integral
    terms where p is the degree of the target basis.
540 """
    polys0, trans0 = RefineLocal(polys0, trans0, reg0, reg1, tol=1E-8)
    # Degree elevation is an operation that is independent of
    # the interval of definition for all standard polynomial bases,
    # so it isn't necessary to pass in the interval widths.
545 polys0 = ElevatePolys(polys0, polys1.shape[-1]-1)
    # It isn't always true that the transcendental terms carry
    # straight across between refined bases.
    reg1lens = reg1[1:] - reg1[:-1]
    pos = reg1lens > tol
550 trans0, offset = RepresentKnotFuncs(trans0, rints0, rints1, reg1lens,
                                        pos, tol=tol)
    polys0 += offset
    # A spline of degree p is only defined over the intervals where
    # p+1 basis functions are nonzero.
555 # This corresponds to the polynomial and transcendental terms
    # that lie in a reverse diagonal that has the same length as
    # a given row of coefficients.
    # Here we preserve the order of the basis functions.
    lbases = FullReverseDiagonals(np.concatenate((polys1, trans1), -1))
560 lfunc = np.concatenate((polys0, trans0), -1)
    # The averaging process assumes that all the coefficients are relatively
    # close to one another. If the piecewise function given doesn't actually
    # lie in the space spanned by the splines, this is where the computation
    # will go awry.
565 coefs = np.empty_like(lfunc)
    coefs[:] = np.nan
    coefs[pos] = np.linalg.solve(np.transpose(lbases[pos], (0, 2, 1)),
                                 lfunc[pos,...,None])[...,0]
    # Thros an error if there is a mismatch in the averages.
570 return ReverseDiagonalAverages(coefs, tol=tol)
```

Bibliography

[1] B. I. Ksasov, P. Sattayatham, GB-splines of arbitrary order, Journal of Computational and Applied Mathematics 104 (1) (1999) 63 – 88. doi:10.1016/S0377-0427(98)00265-9.

[2] P. Costantini, T. Lyche, C. Manni, On a class of weak Tchebycheff systems, Numerische Mathematik 101 (2) (2005) 333–354. doi:10.1007/s00211-005-0613-6.

[3] G. Wang, M. Fang, Unified and extended form of three types of splines, Journal of Computational and Applied Mathematics 216 (2) (2008) 498 – 508. doi:10.1016/j.cam.2007.05.031.

[4] C. Manni, F. Pelosi, M. L. Sampoli, Generalized B-splines as a tool in isogeometric analysis, Computer Methods in Applied Mechanics and Engineering 200 (58) (2011) 867 – 881. doi:10.1016/j.cma.2010.10.010.

[5] L. Romani, From approximating subdivision schemes for exponential splines to high-performance interpolating algorithms, Journal of Computational and Applied Mathematics 224 (1) (2009) 383 – 396. doi:10.1016/j.cam.2008.05.013.

[6] J. Warren, H. Weimer, Subdivision Methods for Geometric Design: A Constructive Approach, Morgan Kaufmann series in computer graphics and geometric modeling, Morgan Kaufmann, 2002.

[7] C. de Boor, A Practical Guide to Splines, Applied Mathematical Sciences, Springer New York, 2001.

[8] G. Farin, Curves and Surfaces for CAGD: A Practical Guide, Computer Graphics and Geometric Modeling, Morgan Kaufmann, 2002.

[9] L. Piegl, W. Tiller, The NURBS Book, Monographs in Visual Communication, U.S. Government Printing Office, 1997.

[10] L. Schumaker, Spline Functions: Basic Theory, Cambridge Mathematical Library, Cambridge University Press, 2007.

[11] R. T. Farouki, The Bernstein polynomial basis: A centennial retrospective, Computer Aided Geometric Design 29 (6) (2012) 379 – 419. doi:10.1016/j.cagd.2012.03.001.

[12] D. Thomas, M. Scott, J. Evans, K. Tew, E. Evans, Bézier projection: A unified approach for local projection and quadrature-free refinement and coarsening of NURBS and t-splines with particular application to isogeometric design and analysis, Computer Methods in Applied Mechanics and Engineering 284 (0) (2015) 55 – 105, isogeometric Analysis Special Issue. doi:http://dx.doi.org/10.1016/j.cma.2014.07.014.

[13] S. N. Bernstein, Démonstration du théorème de weierstrass fondeé sur le calcul des probabilités, Communiactions de la Société Mathématique de Kharkov 2.

[14] I. J. Schoenberg, Contributions to the problem of approximation of equidistant data by analytic functions, Part A: On the problem of smoothing of graduation, a first class of analytic approximation, Quarterly of Applied Mathematics 4 (1946) 45–88.

[15] J. Sánchez-Reyes, Algebraic manipulation in the Bernstein form made simple via convolutions, Computer-Aided Design 35 (10) (2003) 959 – 967. doi:10.1016/S0010-4485(03)00021-6.

[16] M. R. Spencer, Polynomial real root finding in Bernstein form, Ph.D. thesis, Brigham Young University, Provo, UT, USA, uMI Order No. GAX94-23360 (1994).

[17] N. Yang, Structured matrix methods for computations on Bernstein basis polynomials, Ph.D. thesis, Sheffield University (2013).

[18] L. Busé, R. Goldman, Division algorithms for Bernstein polynomials, Computer Aided Geometric Design 25 (9) (2008) 850 – 865, Classical Techniques for Applied Geometry. doi:10.1016/j.cagd.2007.10.003.

[19] M. Minimair, Basis-independent polynomial division algorithm applied to division in lagrange and Bernstein basis, in: Computer Mathematics, Vol. 5081 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008, pp. 72–86. doi:10.1007/978-3-540-87827-8_6.

[20] R. Farouki, V. Rajan, On the numerical condition of polynomials in Bernstein form, Computer Aided Geometric Design 4 (3) (1987) 191 – 216. doi:10.1016/0167-8396(87)90012-4.

[21] E. Mainar, J. Peña, Error analysis of corner cutting algorithms, Numerical Algorithms 22 (1) (1999) 41–52. doi:10.1023/A:1019190220312.

[22] R. Farouki, On the stability of transformations between power and Bernstein polynomial forms, Computer Aided Geometric Design 8 (1) (1991) 29 – 36. doi:10.1016/0167-8396(91)90047-F.

[23] A. Hardy, W. Steeb, Mathematical Tools in Computer Graphics with C# Implementations, World Scientific, 2008.

[24] C. de Boor, On calculating with B-splines, Journal of Approximation Theory 6 (1) (1972) 50 – 62. doi:10.1016/0021-9045(72)90080-9.

[25] M. G. Cox, The numerical evaluation of B-splines, IMA Journal of Applied Mathematics 10 (2) (1972) 134–149. doi:10.1093/imamat/10.2.134.

[26] Q.-X. Huang, S.-M. Hu, R. R. Martin, Fast degree elevation and knot insertion for b-spline curves, Computer Aided Geometric Design 22 (2) (2005) 183 – 197. doi:10.1016/j.cagd.2004.11.001.

[27] M. J. Borden, M. A. Scott, J. A. Evans, T. J. R. Hughes, Isogeometric finite element data structures based on Bézier extraction of NURBS, International Journal for Numerical Methods in Engineering 87 (1-5) (2011) 15–47. doi:10.1002/nme.2968.

[28] T. Hughes, J. Cottrell, Y. Bazilevs, Isogeometric analysis: Cad, finite elements, NURBS, exact geometry and mesh refinement, Computer Methods in Applied Mechanics and Engineering 194 (3941) (2005) 4135 – 4195. doi:10.1016/j.cma.2004.10.008.