High-order Solution Transfer between Curved Meshes and Ill-conditioned Bézier Curve Intersection

by

Daniel Jerome Hermes

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Applied Mathematics

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Per-Olof Persson, Chair
Professor John Strain
Professor Sanjay Govindjee

Summer 2018

High-order Solution Transfer between Curved Meshes and Ill-conditioned Bézier Curve Intersection

Abstract

High-order Solution Transfer between Curved Meshes and Ill-conditioned Bézier Curve Intersection

by

Daniel Jerome Hermes

Doctor of Philosophy in Applied Mathematics

University of California, Berkeley

Professor Per-Olof Persson, Chair

The problem of solution transfer between meshes arises frequently in computational physics, e.g. in Lagrangian methods where remeshing occurs. The interpolation process must be conservative, i.e. it must conserve physical properties, such as mass. We extend previous works — which described the solution transfer process for straight sided unstructured meshes — by considering high-order isoparametric meshes with curved elements. The implementation is highly reliant on accurate computational geometry routines for evaluating points on and intersecting Bézier curves and triangles.

Two ill-conditioned problems that occur evaluating points on and intersecting Bézier curves are then explored. This work presents a family of compensated algorithms to accurately evaluate a polynomial in Bernstein form with floating point coefficients. The principle is to apply error-free transformations to improve the traditional de Casteljau algorithm. The resulting output is as accurate as the de Casteljau algorithm performed in K times the working precision. After compensated evaluation is considered, a compensated Newton's method is described, both for root-finding for polynomials in the Bernstein basis and for Bézier curve intersection.

To my family of four that I gained during the PhD: my wife Sharona, who I proposed to and married during graduate school and our sons Jack and Max.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to acknowledge all my fellow graduate students with and from whom I have learned, especially Will Pazner, Chris Miller and Qiaochu Yuan. I am also thankful to all the mathematics faculty at UC Berkeley from whom I have learned so much.

I am especially grateful to my adviser, Per-Olof Persson, for guidance, support and ideas. I also would like to thank John Strain for many enlightening, enriching and entertaining conversations.

Chapter 1

Introduction

This is a work in two parts, each in a different subfield of mathematics. The first part is a general-purpose tool for computational physics problems. The tool enables solution transfer across two curved meshes. Since the tool requires a significant amount of computational geometry, the second half focuses on computational geometry. In particular, it considers cases where the geometric methods used have seriously degraded accuracy due to ill-conditioning.

1.1 Computational Physics

In computational physics, the problem of solution transfer between meshes occurs in several applications. For example, by allowing the underyling computational domain to change during a simulation, computational effort can be focused dynamically to resolve sensitive features of a numerical solution. Mesh adaptivity (see, for example, [BR78, PVMZ87, PUdOG01]), this in-flight change in the mesh, requires translating the numerical solution from the old mesh to the new, i.e. solution transfer. As another example, Lagrangian or particle-based methods treat each node in mesh as a particle and so with each timestep the mesh travels **with** the fluid (see, for example, [HAC74]). However, over (typically limited) time the mesh becomes distorted and suffers a loss in element quality which causes catastrophic loss in the accuracy of computation. To overcome this, the domain must be remeshed or rezoned and the solution must be transferred (remapped) onto the new mesh configuration.

When pointwise interpolation is used to transfer a solution, quantities with physical meaning (e.g. mass, concentration, energy) may not be conserved. To address this, there have been many explorations (for example, [JH04, FPP+09, FM11]) of **conservative interpolation** (typically using Galerkin or L_2-minimizing methods). In this work, the author introduces a conservative interpolation method for solution transfer between high-order meshes. These high-order meshes are typically curved, but not necessarily all elements or at all timesteps.

The existing work on solution transfer has considered straight-sided meshes, which use

shape functions that have degree $p = 1$ to represent solutions on each element or so-called superparametric elements (i.e. a linear mesh with degree $p > 1$ shape functions on a regular grid of points). However, both to allow for greater geometric flexibility and for high order of convergence, this work will consider the case of curved isoparametric[1] meshes. Allowing curved geometries is useful since many practical problems involve geometries that change over time, such as flapping flight or fluid-structure interactions. In addition, high-order CFD methods ([WFA+13]) have the ability to produce highly accurate solutions with low dissipation and low dispersion error.

1.2 Computational Geometry

For a function in Bernstein form, the condition number of evaluation becomes infinite as the input approaches a root. Similarly, as a (transversal) intersection of two Bézier curves approaches a point of tangency, the condition number of intersection becomes infinite. These breakdowns in accuracy cause problems when evaluating integrals on elements of a curved mesh or on the intersections of two elements. For example, consider the problem of solution transfer from one mesh to another. As both meshes are refined simultaneously, the probability of an "almost tangent" pair of curved edges increases towards unity. Tangent curves correspond to the case of a double root of a polynomial. Though they are unlikely for a random mesh pair "double roots, though rare, are overwhelmingly more common in practice than are roots of higher multiplicity" ([Kah72], page 6).

Two approaches will be described that can help recover this lost accuracy in the presence of ill-conditioning. The first allows for greater accuracy when performing the de Casteljau algorithm to evaluate a function in Bernstein form. This **compensated algorithm** (Chapter 5) produces results that are as accurate as if the computations were done in K times the working precision and then rounded back to the working precision. By just using a more precise evaluation of the residual function, [Tis01] showed that the accuracy of Newton's method can be improved. So as a natural extension, the second approach explores the improvement in Newton's method applied both to root-finding and Bézier curve intersection (Chapter 6).

1.3 Organization

This work is organized as follows. Chapter 2 establishes common notation and reviews basic results relevant to the topics at hand. Chapter 3 is an in-depth discussion of the computational geometry methods needed to implement to enable solution transfer. Chapter 4 describes the solution transfer process and gives results of some numerical experiments confirming the rate of convergence. Chapter 5 describes a compensated algorithm for evaluating

[1] I.e. the degree of the discrete field on the mesh is same as the degree of the shape functions that determine the mesh.

functions in Bernstein form (such as Bézier curves); this algorithm produces results that are as accurate as if the computations were done in K times the working precision and then rounded back to the working precision. Chapter 6 describes two modified Newton's methods which allow for greater accuracy in the presence of ill-conditioning; one is used for computing simple zeros of polynomials in Bernstein form and the other for computing Bézier curve intersections.

Chapter 2

Preliminaries

2.1 General Notation

We'll refer to $\mathbf{R}$ for the reals, $\mathcal{U}$ represents the unit triangle (or unit simplex) in $\mathbf{R}^2$: $\mathcal{U} = \{(s,t) \mid 0 \le s,t,s+t \le 1\}$. When dealing with sequences with multiple indices, e.g. $s_{m,n} = m+n$, we'll use bold symbols to represent a multi-index: $\boldsymbol{i} = (m,n)$. We'll use $|\boldsymbol{i}|$ to represent the sum of the components in a multi-index. The binomial coefficient $\binom{n}{k}$ is equal to $\frac{n!}{k!(n-k)!}$ and the trinomial coefficient $\binom{n}{i,j,k}$ is equal to $\frac{n!}{i!j!k!}$ (where $i+j+k=n$). The notation δ_{ij} represents the Kronecker delta, a value which is 1 when $i=j$ and 0 otherwise.

2.2 Floating Point and Forward Error Analysis

We assume all floating point operations obey

$$a \star b = \mathrm{fl}\,(a \circ b) = (a \circ b)(1+\delta_1) = (a \circ b)/(1+\delta_2) \tag{2.1}$$

where $\star \in \{\oplus, \ominus, \otimes, \oslash\}$, $\circ \in \{+,-,\times,\div\}$ and $|\delta_1|, |\delta_2| \le \mathbf{u}$. The symbol $\mathbf{u}$ is the unit round-off and $\star$ is a floating point operation, e.g. $a \oplus b = \mathrm{fl}\,(a+b)$. (For IEEE-754 floating point double precision, $\mathbf{u} = 2^{-53}$.) We denote the computed result of $\alpha \in \mathbf{R}$ in floating point arithmetic by $\widehat{\alpha}$ or $\mathrm{fl}\,(\alpha)$ and use $\mathbf{F}$ as the set of all floating point numbers (see [Hig02] for more details). Following [Hig02], we will use the following classic properties in error analysis.

1. If $\delta_i \le \mathbf{u}$, $\rho_i = \pm 1$, then $\prod_{i=1}^{n}(1+\delta_i)^{\rho_i} = 1+\theta_n$,
2. $|\theta_n| \le \gamma_n \coloneqq n\mathbf{u}/(1-n\mathbf{u})$,
3. $(1+\theta_k)(1+\theta_j) = 1+\theta_{k+j}$,
4. $\gamma_k + \gamma_j + \gamma_k\gamma_j \le \gamma_{k+j} \iff (1+\gamma_k)(1+\gamma_j) \le 1+\gamma_{k+j}$,
5. $(1+\mathbf{u})^j \le 1/(1-j\mathbf{u}) \iff (1+\mathbf{u})^j - 1 \le \gamma_j$.

2.3 Bézier Curves

A **Bézier curve** is a mapping from the unit interval that is determined by a set of control points $\{\boldsymbol{p}_j\}_{j=0}^n \subset \mathbf{R}^d$. For a parameter $s \in [0,1]$, there is a corresponding point on the curve:

$$b(s) = \sum_{j=0}^{n} \binom{n}{j} (1-s)^{n-j} s^j \boldsymbol{p}_j \in \mathbf{R}^d. \tag{2.2}$$

This is a combination of the control points weighted by each Bernstein basis function $B_{j,n}(s) = \binom{n}{j}(1-s)^{n-j}s^j$. Due to the binomial expansion $1 = (s + (1-s))^n = \sum_{j=0}^n B_{j,n}(s)$, a Bernstein basis function is in $[0,1]$ when s is as well. Due to this fact, the curve must be contained in the convex hull of it's control points.

2.3.1 de Casteljau Algorithm

Next, we recall[1] the de Casteljau algorithm:

Algorithm 2.1 *de Casteljau algorithm for polynomial evaluation.*

function `result` = `DeCasteljau`(b, s)
  $n =$ `length`$(b) - 1$
  $\widehat{r} = 1 \ominus s$

  for $j = 0, \ldots, n$ **do**
    $\widehat{b}_j^{(n)} = b_j$
  end for

  for $k = n-1, \ldots, 0$ **do**
    for $j = 0, \ldots, k$ **do**
      $\widehat{b}_j^{(k)} = \left(\widehat{r} \otimes \widehat{b}_j^{(k+1)}\right) \oplus \left(s \otimes \widehat{b}_{j+1}^{(k+1)}\right)$
    end for
  end for

  `result` $= \widehat{b}_0^{(0)}$
end function

[1] We have used slightly non-standard notation for the terms produced by the de Casteljau algorithm: we start the superscript at n and count down to 0 as is typically done when describing Horner's algorithm. For example, we use $b_j^{(n-2)}$ instead of $b_j^{(2)}$.

Theorem 2.1 ([MP99], Corollary 3.2)**.** If $p(s) = \sum_{j=0}^{n} b_j B_{j,n}(s)$ and $\texttt{DeCasteljau}(p, s)$ is the value computed by the de Casteljau algorithm then[2]

$$|p(s) - \texttt{DeCasteljau}(p, s)| \leq \gamma_{3n} \sum_{j=0}^{n} |b_j| \, B_{j,n}(s). \tag{2.3}$$

The relative condition number of the evaluation of $p(s) = \sum_{j=0}^{n} b_j B_{j,n}(s)$ in Bernstein form used in this work is (see [MP99, FR87]):

$$\text{cond}\,(p, s) = \frac{\widetilde{p}(s)}{|p(s)|}, \tag{2.4}$$

where $\widetilde{p}(s) \coloneqq \sum_{j=0}^{n} |b_j| \, B_{j,n}(s)$.

To be able to express the algorithm in matrix form, we define the vectors

$$b^{(k)} = \begin{bmatrix} b_0^{(k)} & \cdots & b_k^{(k)} \end{bmatrix}^T, \quad \widehat{b}^{(k)} = \begin{bmatrix} \widehat{b}_0^{(k)} & \cdots & \widehat{b}_k^{(k)} \end{bmatrix}^T \tag{2.5}$$

and the reduction matrices:

$$U_k = U_k(s) = \begin{bmatrix} 1-s & s & 0 & \cdots & \cdots & 0 \\ 0 & 1-s & s & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & 0 & 1-s & s \end{bmatrix} \in \mathbf{R}^{k \times (k+1)}. \tag{2.6}$$

With this, we can express ([MP99]) the de Casteljau algorithm as

$$b^{(k)} = U_{k+1} b^{(k+1)} \Longrightarrow b^{(0)} = U_1 \cdots U_n b^{(n)}. \tag{2.7}$$

In general, for a sequence $v_0, \ldots, v_n$ we'll refer to v as the vector containing all of the values: $v = \begin{bmatrix} v_0 & \cdots & v_n \end{bmatrix}^T$.

2.4 Bézier Triangles

A **Bézier triangle** ([Far01, Chapter 17]) is a mapping from the unit triangle $\mathcal{U}$ and is determined by a control net $\{\boldsymbol{p}_{i,j,k}\}_{i+j+k=n} \subset \mathbf{R}^d$. A Bézier triangle is a particular kind of Bézier surface, i.e. one in which there are two cartesian or three barycentric input parameters.

[2] In the original paper the factor on $\widetilde{p}(s)$ is γ_{2n}, but the authors did not consider round-off when computing $1 \ominus s$.

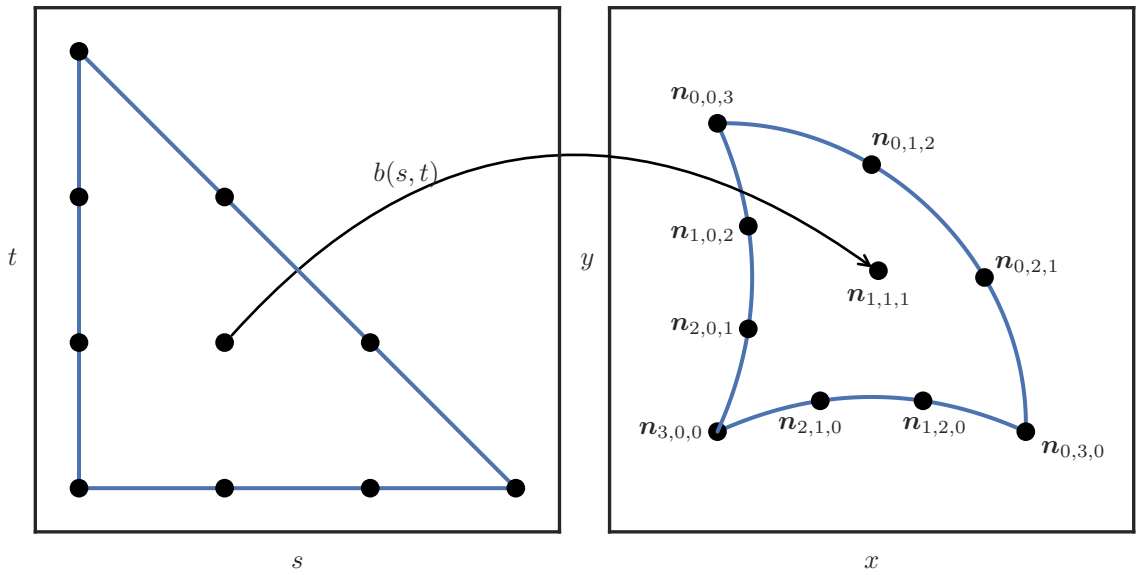

Figure 2.1: Cubic Bézier triangle

Often the term Bézier surface is used to refer to a tensor product or rectangular patch. For $(s,t) \in \mathcal{U}$ we can define barycentric weights $\lambda_1 = 1 - s - t, \lambda_2 = s, \lambda_3 = t$ so that

$$1 = (\lambda_1 + \lambda_2 + \lambda_3)^n = \sum_{\substack{i+j+k=n \\ i,j,k \geq 0}} \binom{n}{i,j,k} \lambda_1^i \lambda_2^j \lambda_3^k. \tag{2.8}$$

Using this we can similarly define a (triangular) Bernstein basis

$$B_{i,j,k}(s,t) = \binom{n}{i,j,k} (1-s-t)^i s^j t^k = \binom{n}{i,j,k} \lambda_1^i \lambda_2^j \lambda_3^k \tag{2.9}$$

that is in $[0,1]$ when (s,t) is in $\mathcal{U}$. Using this, we define points on the Bézier triangle as a convex combination of the control net:

$$b(s,t) = \sum_{i+j+k=n} \binom{n}{i,j,k} \lambda_1^i \lambda_2^j \lambda_3^k \boldsymbol{p}_{i,j,k} \in \mathbf{R}^d. \tag{2.10}$$

Rather than defining a Bézier triangle by the control net, it can also be uniquely determined by the image of a standard lattice of points in $\mathcal{U}$: $b\left(j/n, k/n\right) = \boldsymbol{n}_{i,j,k}$; we'll refer to these as **standard nodes**. Figure 2.1 shows these standard nodes for a cubic triangle in $\mathbf{R}^2$. To see the correspondence, when $p = 1$ the standard nodes **are** the control net

$$b(s,t) = \lambda_1 \boldsymbol{n}_{1,0,0} + \lambda_2 \boldsymbol{n}_{0,1,0} + \lambda_3 \boldsymbol{n}_{0,0,1} \tag{2.11}$$

and when $p = 2$

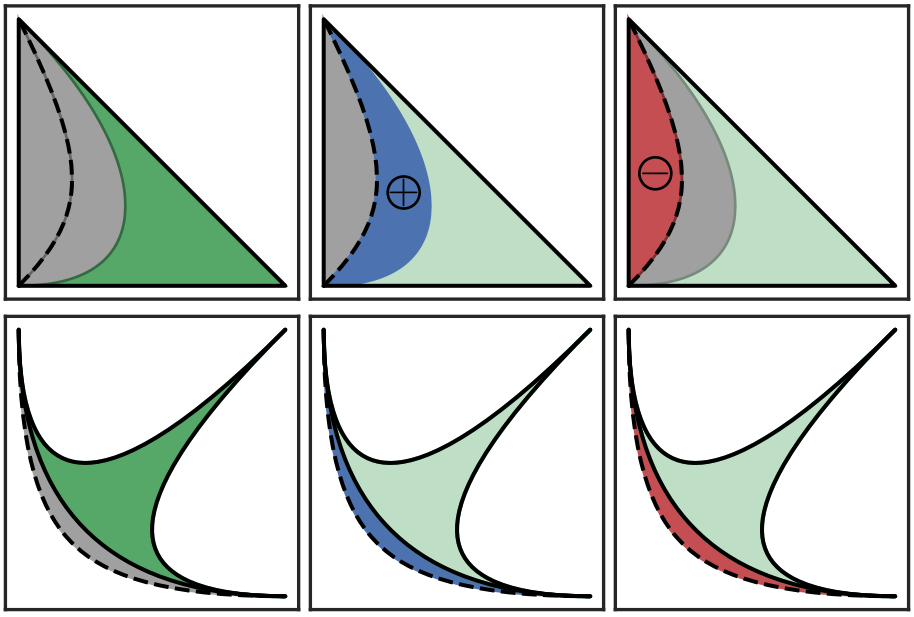

Figure 2.2: The Bézier triangle given by $b(s,t) = [(1-s-t)^2 + s^2 \;\; s^2 + t^2]^T$ produces an inverted element. It traces the same region twice, once with a positive Jacobian (the middle column) and once with a negative Jacobian (the right column).

$$b(s,t) = \lambda_1 (2\lambda_1 - 1)\, \boldsymbol{n}_{2,0,0} + \lambda_2 (2\lambda_2 - 1)\, \boldsymbol{n}_{0,2,0} + \lambda_3 (2\lambda_3 - 1)\, \boldsymbol{n}_{0,0,2} + 4\lambda_1\lambda_2 \boldsymbol{n}_{1,1,0} + 4\lambda_2\lambda_3 \boldsymbol{n}_{0,1,1} + 4\lambda_3\lambda_1 \boldsymbol{n}_{1,0,1}. \quad (2.12)$$

However, it's worth noting that the transformation between the control net and the standard nodes has condition number that grows exponentially with n (see [Far91], which is related but does not directly show this). This may make working with higher degree triangles prohibitively unstable.

A **valid** Bézier triangle is one which is diffeomorphic to $\mathcal{U}$, i.e. $b(s,t)$ is bijective and has an everywhere invertible Jacobian. We must also have the orientation preserved, i.e. the Jacobian must have positive determinant. For example, in Figure 2.2, the image of $\mathcal{U}$ under the map $b(s,t) = \begin{bmatrix} (1-s-t)^2 + s^2 & s^2 + t^2 \end{bmatrix}^T$ is not valid because the Jacobian is zero along the curve $s^2 - st - t^2 - s + t = 0$ (the dashed line). Elements that are not valid are called **inverted** because they have regions with "negative area". For the example, the image $b(\mathcal{U})$ leaves the boundary determined by the edge curves: $b(r,0)$, $b(1-r,r)$ and $b(0,1-r)$ when $r \in [0,1]$. This region outside the boundary is traced twice, once with a positive Jacobian and once with a negative Jacobian.

2.5 Curved Elements

We define a curved mesh element $\mathcal{T}$ of degree p to be a Bézier triangle in $\mathbf{R}^2$ of the same degree. We refer to the component functions of $b(s,t)$ (the map that gives $\mathcal{T} = b(\mathcal{U})$) as $x(s,t)$ and $y(s,t)$.

This fits a typical definition ([JM09, Chapter 12]) of a curved element, but gives a special meaning to the mapping from the reference triangle. Interpreting elements as Bézier triangles has been used for Lagrangian methods where mesh adaptivity is needed (e.g. [CMOP04]). Typically curved elements only have one curved side ([MM72]) since they are used to resolve geometric features of a boundary. See also [Zlá73, Zlá74]. Bézier curves and triangles have a number of mathematical properties (e.g. the convex hull property) that lead to elegant geometric descriptions and algorithms.

Note that a Bézier triangle can be determined from many different sources of data (for example the control net or the standard nodes). The choice of this data may be changed to suit the underlying physical problem without changing the actual mapping. Conversely, the data can be fixed (e.g. as the control net) to avoid costly basis conversion; once fixed, the equations of motion and other PDE terms can be recast relative to the new basis (for an example, see [PBP09], where the domain varies with time but the problem is reduced to solving a transformed conservation law in a fixed reference configuration).

2.5.1 Shape Functions

When defining shape functions (i.e. a basis with geometric meaning) on a curved element there are (at least) two choices. When the degree of the shape functions is the same as the degree of the function being represented on the Bézier triangle, we say the element $\mathcal{T}$ is **isoparametric**. For the multi-index $\boldsymbol{i} = (i,j,k)$, we define $\boldsymbol{u_i} = (j/n, k/n)$ and the corresponding standard node $\boldsymbol{n_i} = b(\boldsymbol{u_i})$. Given these points, two choices for shape functions present themselves:

- **Pre-Image Basis**: $\phi_{\boldsymbol{j}}(\boldsymbol{n_i}) = \widehat{\phi}_{\boldsymbol{j}}(\boldsymbol{u_i}) = \widehat{\phi}_{\boldsymbol{j}}(b^{-1}(\boldsymbol{n_i}))$ where $\widehat{\phi}_{\boldsymbol{j}}$ is a canonical basis function on $\mathcal{U}$, i.e. $\widehat{\phi}_{\boldsymbol{j}}$ a degree p bivariate polynomial and $\widehat{\phi}_{\boldsymbol{j}}(\boldsymbol{u_i}) = \delta_{\boldsymbol{ij}}$
- **Global Coordinates Basis**: $\phi_{\boldsymbol{j}}(\boldsymbol{n_i}) = \delta_{\boldsymbol{ij}}$, i.e. a canonical basis function on the standard nodes $\{\boldsymbol{n_i}\}$.

For example, consider a quadratic Bézier triangle:

$$b(s,t) = \begin{bmatrix} 4(st+s+t) & 4(st+t+1) \end{bmatrix}^T \tag{2.13}$$

$$\implies \begin{bmatrix} \boldsymbol{n}_{2,0,0} & \boldsymbol{n}_{1,1,0} & \boldsymbol{n}_{0,2,0} & \boldsymbol{n}_{1,0,1} & \boldsymbol{n}_{0,1,1} & \boldsymbol{n}_{0,0,2} \end{bmatrix} = \begin{bmatrix} 0 & 2 & 4 & 2 & 5 & 4 \\ 4 & 4 & 4 & 6 & 7 & 8 \end{bmatrix}. \tag{2.14}$$

In the **Global Coordinates Basis**, we have

$$\phi^G_{0,1,1}(x,y) = \frac{(y-4)(x-y+4)}{6}. \tag{2.15}$$

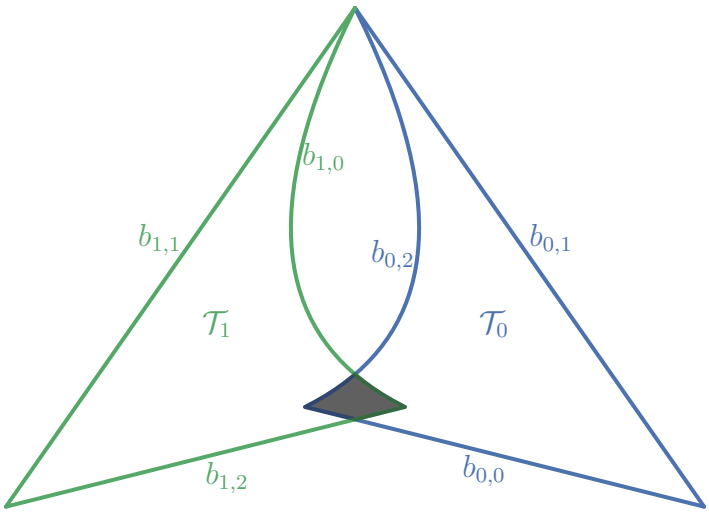

Figure 2.3: Intersection of Bézier triangles form a curved polygon.

For the **Pre-Image Basis**, we need the inverse and the canonical basis

$$b^{-1}(x, y) = \begin{bmatrix} \frac{x-y+4}{4} & \frac{y-4}{x-y+8} \end{bmatrix} \quad \text{and} \quad \widehat{\phi}_{0,1,1}(s, t) = 4st \tag{2.16}$$

and together they give

$$\phi^P_{0,1,1}(x, y) = \frac{(y-4)(x-y+4)}{x-y+8}. \tag{2.17}$$

In general $\phi^P_{\boldsymbol{j}}$ may not even be a rational bivariate function; due to composition with b^{-1} we can only guarantee that it is algebraic (i.e. it can be defined as the zero set of polynomials).

2.5.2 Curved Polygons

When intersecting two curved elements, the resulting surface(s) will be defined by the boundary, alternating between edges of each element. For example, in Figure 2.3, a "curved quadrilateral" is formed when two Bézier triangles $\mathcal{T}_0$ and $\mathcal{T}_1$ are intersected.

A **curved polygon** is defined by a collection of Bézier curves in $\mathbf{R}^2$ that determine the boundary. In order to be a valid polygon, none of the boundary curves may cross, the ends of consecutive edge curves must meet and the curves must be right-hand oriented. For our example in Figure 2.3, the triangles have boundaries formed by three Bézier curves: $\partial\mathcal{T}_0 = b_{0,0} \cup b_{0,1} \cup b_{0,2}$ and $\partial\mathcal{T}_1 = b_{1,0} \cup b_{1,1} \cup b_{1,2}$. The intersection $\mathcal{P}$ is defined by four boundary curves: $\partial\mathcal{P} = C_1 \cup C_2 \cup C_3 \cup C_4$. Each boundary curve is itself a Bézier curve[3]: $C_1 = b_{0,0}([0, 1/8])$, $C_2 = b_{1,2}([7/8, 1])$, $C_3 = b_{1,0}([0, 1/7])$ and $C_4 = b_{0,2}([6/7, 1])$.

[3] A specialization of a Bézier curve $b([a_1, a_2])$ is also a Bézier curve.

Though an intersection can be described in terms of the Bézier triangles, the structure of the control net will be lost. The region will not in general be able to be described by a mapping from a simple space like $\mathcal{U}$.

2.6 Error-Free Transformation

An error-free transformation is a computational method where both the computed result and the round-off error are returned. It is considered "free" of error if the round-off can be represented exactly as an element or elements of $\mathbf{F}$. The error-free transformations used in this work are the `TwoSum` algorithm by Knuth ([Knu97]) and `TwoProd` algorithm by Dekker ([Dek71], Section 5), respectively.

Theorem 2.1 ([ORO05], Theorem 3.4)**.** For $a, b \in \mathbf{F}$ and $P, \pi, S, \sigma \in \mathbf{F}$, `TwoSum` and `TwoProd` satisfy

$$[S, \sigma] = \texttt{TwoSum}(a, b), \quad S = \mathrm{fl}\left(a + b\right), S + \sigma = a + b, \sigma \leq \mathbf{u}\left|S\right|, \sigma \leq \mathbf{u}\left|a + b\right| \tag{2.18}$$

$$[P, \pi] = \texttt{TwoProd}(a, b), P = \mathrm{fl}\left(a \times b\right), P + \pi = a \times b, \pi \leq \mathbf{u}\left|P\right|, \pi \leq \mathbf{u}\left|a \times b\right|. \tag{2.19}$$

The letters σ and π are used to indicate that the errors came from sum and product, respectively. See Appendix A for implementation details.

Chapter 3

Bézier Intersection Problems

3.1 Intersecting Bézier Curves

The problem of intersecting two Bézier curves is a core building block for intersecting two Bézier triangles in $\mathbf{R}^2$ Since a curve is an algebraic variety of dimension one, the intersections will either be a curve segment common to both curves (if they coincide) or a finite set of points (i.e. dimension zero). Many algorithms have been described in the literature, both geometric ([SP86, SN90, KLS98]) and algebraic ([MD92]).

In the implementation for this work, the Bézier subdivision algorithm is used. In the case of a transversal intersection (i.e. one where the tangents to each curve are not parallel and both are non-zero), this algorithm performs very well. However, when curves are tangent, a large number of (false) candidate intersections are detected and convergence of Newton's method slows once in a neighborhood of an actual intersection. Non-transversal intersections have infinite condition number, but transversal intersections with very high condition number can also cause convergence problems.

In the Bézier subdivision algorithm, we first check if the bounding boxes for the curves are disjoint (Figure 3.1). We use the bounding boxes rather than the convex hulls since they are easier to compute and the intersections of boxes are easier to check. If they are disjoint, the pair can be rejected. If not, each curve $\mathcal{C} = b([0,1])$ is split into two halves by splitting the unit interval: $b\left(\left[0, \frac{1}{2}\right]\right)$ and $b\left(\left[\frac{1}{2}, 1\right]\right)$ (Figure 3.2).

As the subdivision continues, some pairs of curve segments may be kept around that won't lead to an intersection (Figure 3.3). Once the curve segments are close to linear within a given tolerance (Figure 3.4), the process terminates.

Once both curve segments are linear (to tolerance), the intersection is approximated by intersecting the lines connecting the endpoints of each curve segment. This approximation is used as a starting point for Newton's method, to find a root of $F(s,t) = b_0(s) - b_1(t)$. Since $b_0(s), b_1(t) \in \mathbf{R}^2$ we have Jacobian $J = \begin{bmatrix} b_0'(s) & -b_1'(t) \end{bmatrix}$. With these, Newton's method is

$$\begin{bmatrix} s_{n+1} & t_{n+1} \end{bmatrix}^T = \begin{bmatrix} s_n & t_n \end{bmatrix}^T - J_n^{-1} F_n. \tag{3.1}$$

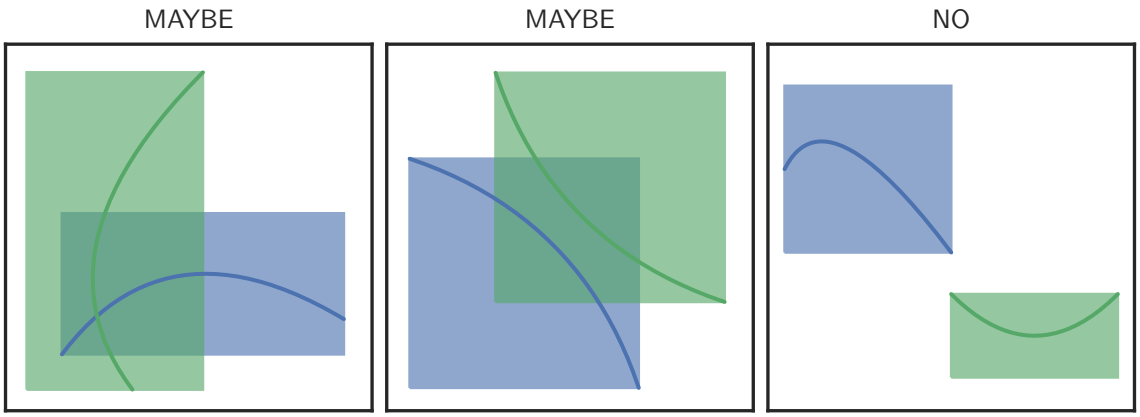

Figure 3.1: Bounding box intersection predicate. This is a cheap way to conclude that two curves don't intersect, though it inherently is susceptible to false positives.

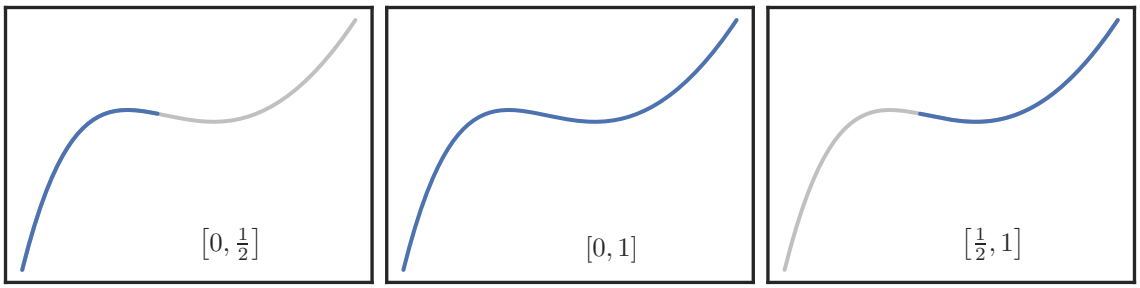

Figure 3.2: Bézier curve subdivision.

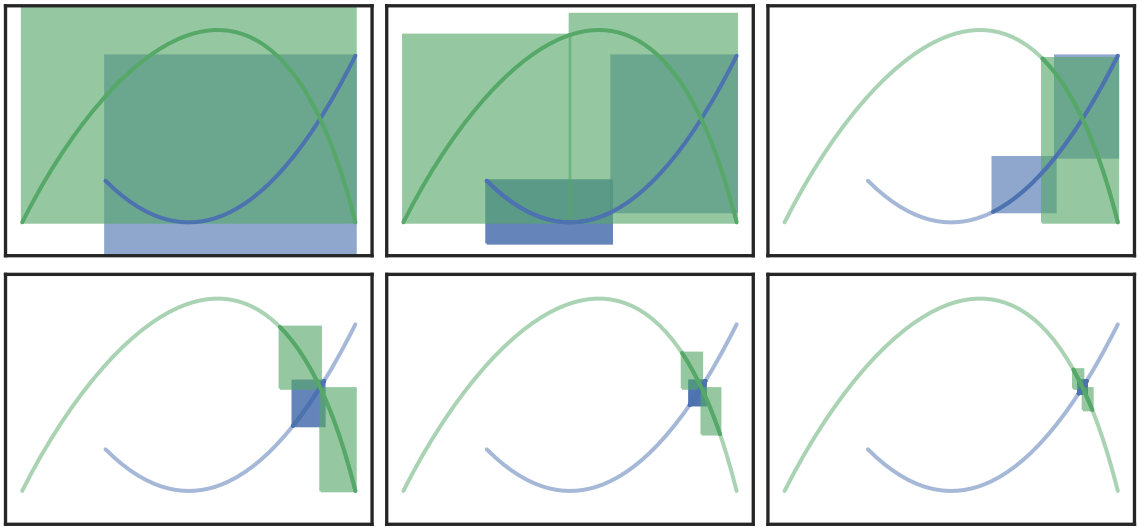

Figure 3.3: Bézier subdivision algorithm.

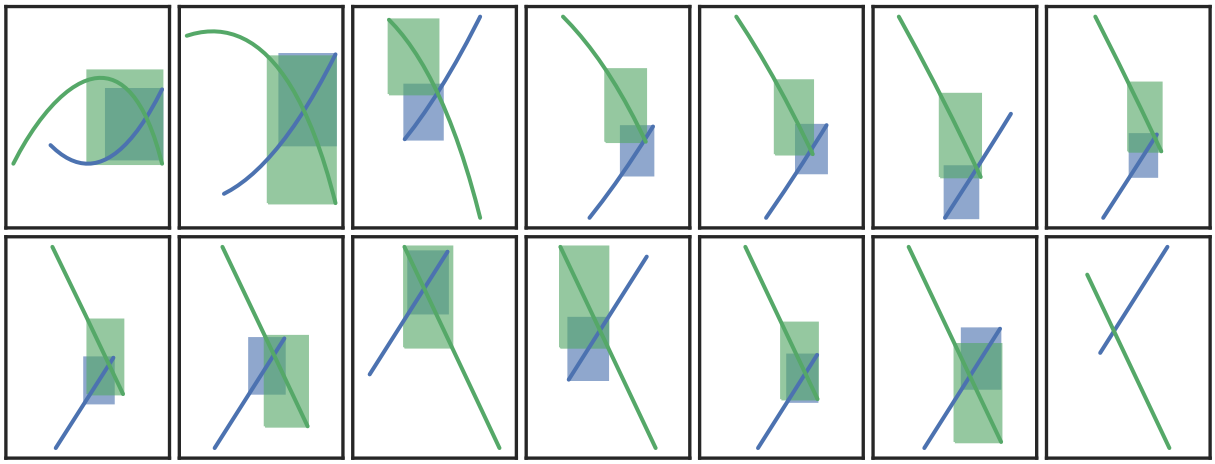

Figure 3.4: Subdividing until linear within tolerance.

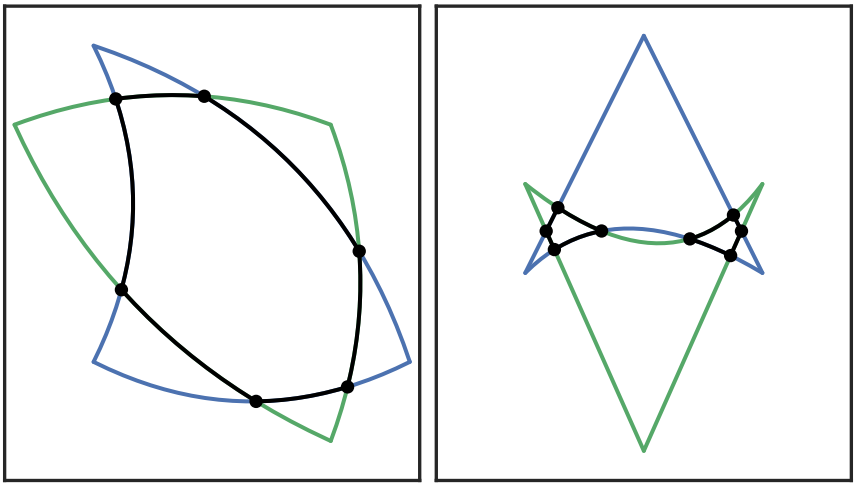

Figure 3.5: Edge intersections during Bézier triangle intersection.

This also gives an indication why convergence issues occur at non-transveral intersections: they are exactly the intersections where the Jacobian is singular.

3.2 Intersecting Bézier Triangles

The chief difficulty in intersecting two surfaces is intersecting their edges, which are Bézier curves. Though this is just a part of the overall algorithm, it proved to be the **most difficult** to implement ([Her17]). So the first part of the algorithm is to find all points where the edges intersect (Figure 3.5).

To determine the curve segments that bound the curved polygon region(s) (see Sec-

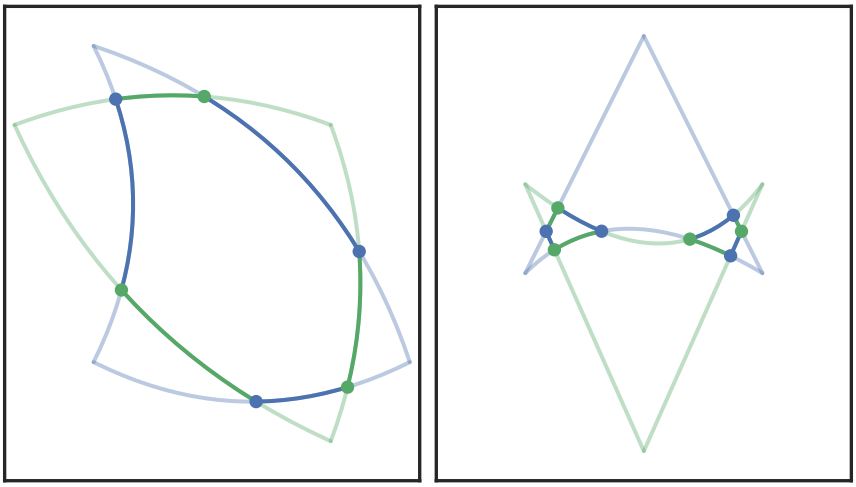

Figure 3.6: Classified intersections during Bézier triangle intersection.

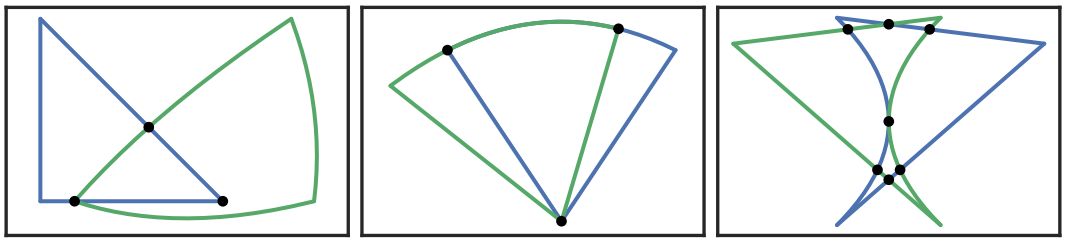

Figure 3.7: Bézier triangle intersection difficulties.

tion 2.5.2 for more about curved polygons) of intersection, we not only need to keep track of the coordinates of intersection, we also need to keep note of **which** edges the intersection occurred on and the parameters along each curve. With this information, we can classify each point of intersection according to which of the two curves forms the boundary of the curved polygon (Figure 3.6). Using the right-hand rule we can compare the tangent vectors on each curve to determine which one is on the interior.

This classification becomes more difficult when the curves are tangent at an intersection, when the intersection occurs at a corner of one of the surfaces or when two intersecting edges are coincident on the same algebraic curve (Figure 3.7).

In the case of tangency, the intersection is non-transversal, hence has infinite condition number. In the case of coincident curves, there are infinitely many intersections (along the segment when the curves coincide) so the subdivision process breaks down.

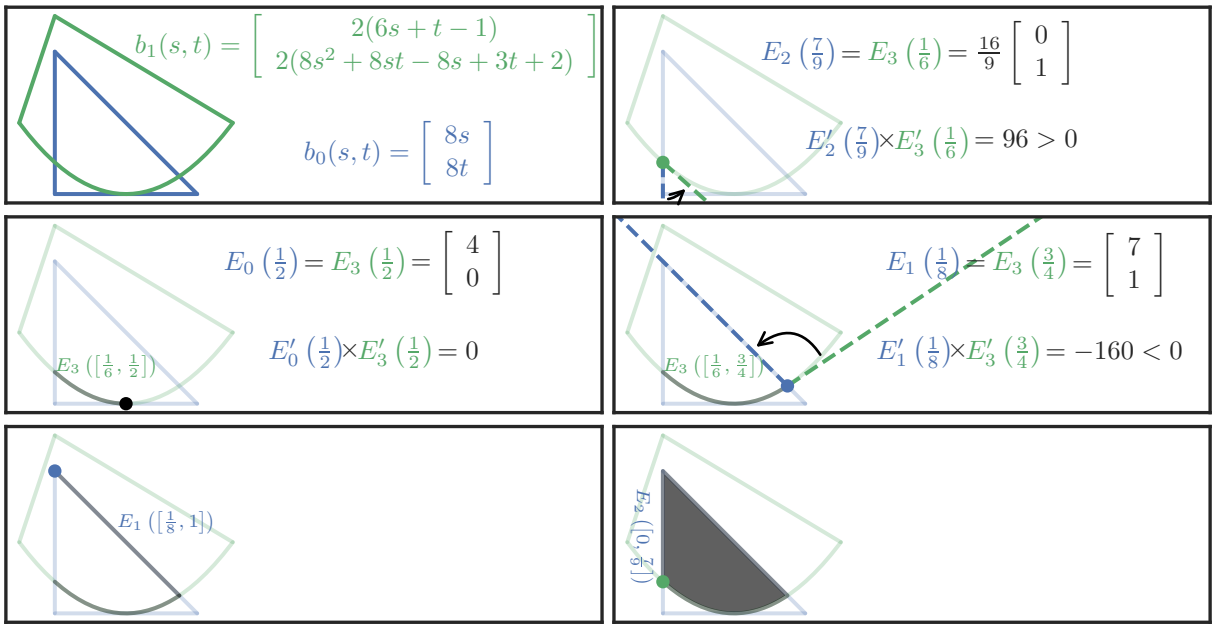

Figure 3.8: Surface Intersection Example

3.2.1 Example

Consider two Bézier surfaces (Figure 3.8)

$$b_0(s,t) = \begin{bmatrix} 8s \\ 8t \end{bmatrix} \qquad b_1(s,t) = \begin{bmatrix} 2(6s+t-1) \\ 2(8s^2+8st-8s+3t+2) \end{bmatrix} \tag{3.2}$$

In the **first step** we find all intersections of the edge curves

$$E_0(r) = \begin{bmatrix} 8r \\ 0 \end{bmatrix}, E_1(r) = \begin{bmatrix} 8(1-r) \\ 8r \end{bmatrix}, E_2(r) = \begin{bmatrix} 0 \\ 8(1-r) \end{bmatrix},$$
$$E_3(r) = \begin{bmatrix} 2(6r-1) \\ 4(2r-1)^2 \end{bmatrix}, E_4(r) = \begin{bmatrix} 10(1-r) \\ 2(3r+2) \end{bmatrix}, E_5(r) = \begin{bmatrix} -2r \\ 2(5-3r) \end{bmatrix}. \tag{3.3}$$

We find three intersections and we classify each of them by comparing the tangent vectors

$$I_1 : E_2\left(\frac{7}{9}\right) = E_3\left(\frac{1}{6}\right) = \frac{16}{9}\begin{bmatrix} 0 \\ 1 \end{bmatrix} \Longrightarrow E_2'\left(\frac{7}{9}\right) \times E_3'\left(\frac{1}{6}\right) = 96 \tag{3.4}$$

$$I_2 : E_0\left(\frac{1}{2}\right) = E_3\left(\frac{1}{2}\right) = \begin{bmatrix} 4 \\ 0 \end{bmatrix} \Longrightarrow E_0'\left(\frac{1}{2}\right) \times E_3'\left(\frac{1}{2}\right) = 0 \tag{3.5}$$

$$I_3 : E_1\left(\frac{1}{8}\right) = E_3\left(\frac{3}{4}\right) = \begin{bmatrix} 7 \\ 1 \end{bmatrix} \Longrightarrow E_1'\left(\frac{1}{8}\right) \times E_3'\left(\frac{3}{4}\right) = -160. \tag{3.6}$$

From here, we construct our curved polygon intersection by drawing from our list of intersections until none remain.

- First consider I_1. Since $E_2' \times E_3' > 0$ at this point, then we consider the curve E_3 to be *interior*.
- After classification, we move along E_3 until we encounter another intersection: I_2
- I_2 is a point of tangency since $E_0'\left(\frac{1}{2}\right) \times E_3'\left(\frac{1}{2}\right) = 0$. Since a tangency has no impact on the underlying intersection geometry, we ignore it and keep moving.
- Continuing to move along E_3, we encounter another intersection: I_3. Since $E_1' \times E_3' < 0$ at this point, we consider the curve E_1 to be *interior* at the intersection. Thus we stop moving along E_3 and we have our first curved segment: $E_3\left(\left[\frac{1}{6}, \frac{3}{4}\right]\right)$
- Finding no other intersections on E_1 we continue until the end of the edge. Now our (ordered) curved segments are:

$$E_3\left(\left[\frac{1}{6}, \frac{3}{4}\right]\right) \longrightarrow E_1\left(\left[\frac{1}{8}, 1\right]\right). \tag{3.7}$$

- Next we stay at the corner and switch to the next curve E_2, moving along that curve until we hit the next intersecton I_1. Now our (ordered) curved segments are:

$$E_3\left(\left[\frac{1}{6}, \frac{3}{4}\right]\right) \longrightarrow E_1\left(\left[\frac{1}{8}, 1\right]\right) \longrightarrow E_2\left(\left[0, \frac{7}{9}\right]\right). \tag{3.8}$$

  Since we are now back where we started (at I_1) the process stops

We represent the boundary of the curved polygon as Bézier curves, so to complete the process we reparameterize ([Far01, Ch. 5.4]) each curve onto the relevant interval. For example, E_3 has control points $p_0 = \begin{bmatrix} -2 \\ 4 \end{bmatrix}$, $p_1 = \begin{bmatrix} 4 \\ -4 \end{bmatrix}$, $p_2 = \begin{bmatrix} 10 \\ 4 \end{bmatrix}$ and we reparameterize on $\alpha = \frac{1}{6}, \beta = \frac{3}{4}$ to control points

$$q_0 = E_3\left(\frac{1}{6}\right) = \frac{16}{9}\begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{3.9}$$

$$q_1 = (1-\alpha)\left[(1-\beta)p_0 + \beta p_1\right] + \alpha\left[(1-\beta)p_1 + \beta p_2\right] = \frac{1}{6}\begin{bmatrix} 21 \\ -8 \end{bmatrix} \tag{3.10}$$

$$q_2 = E_3\left(\frac{3}{4}\right) = \begin{bmatrix} 7 \\ 1 \end{bmatrix}. \tag{3.11}$$

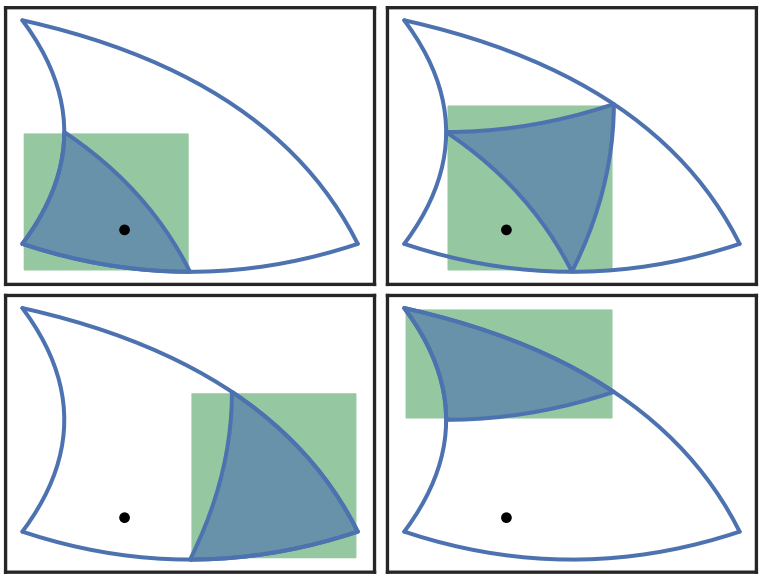

Figure 3.9: Checking for a point $\boldsymbol{p}$ in each of four subregions when subdividing a Bézier triangle.

3.3 Bézier Triangle Inverse

The problem of determining the parameters (s, t) given a point $\boldsymbol{p} = \begin{bmatrix} x & y \end{bmatrix}^T$ in a Bézier triangle can also be solved by using subdivision with a bounding box predicate and then Newton's method at the end.

For example, Figure 3.9 shows how regions of $\mathcal{U}$ can be discarded recursively until the suitable region for (s, t) has a sufficiently small area. At this point, we can apply Newton's method to the map $F(s, t) = b(s, t) - \boldsymbol{p}$. It's very helpful (for Newton's method) that $F : \mathbf{R}^2 \longrightarrow \mathbf{R}^2$ since the Jacobian will always be invertible when the Bézier triangle is valid. If $\boldsymbol{p} \in \mathbf{R}^3$ then the system would be underdetermined. Similarly, if $\boldsymbol{p} \in \mathbf{R}^2$ but $b(s)$ is a Bézier curve then the system would be overdetermined.

Chapter 4

Solution Transfer

4.1 Introduction

In this chapter, an algorithm for conservative solution transfer between curved meshes will be described. This has practical applications to many methods in computational physics. Solution transfer is needed when a solution (approximated by a discrete field) is known on a **donor** mesh and must be transferred to a **target** mesh. In many applications, the field must be conserved for physical reasons, e.g. mass or energy cannot leave or enter the system, hence the focus on **conservative** solution transfer. A few scenarios where solution transfer is necessary will be considered below to motivate the "black box" solution transfer algorithm.

Since solution transfer is so commonly needed in physical applications, this problem of conservative interpolation has been considered already for straight sided meshes. The **common refinement** approach in [JH04] is used to compare several methods for solution transfer across two meshes. However, the problem of constructing a common refinement is not discussed there. The problem of constructing such a refinement is considered in [FPP+09, FM11] (called a supermesh by the authors). However, the solution transfer becomes considerably more challenging for curved meshes. For a sense of the difference between the straight sided and curved cases, consider the problem of intersecting an element from the donor mesh with an element from the target mesh. If the elements are triangles, the intersection is either a convex polygon or has measure zero. If the elements are curved, the intersection can be non-convex and can even split into multiple disjoint regions.

4.1.1 Lagrangian Methods

The method of characteristics helps transform partial differential equations into ordinary differential equations by dividing the physical domain into a family of curves. For example, the simple transport equation

$$u_t + cu_x = 0 \tag{4.1}$$

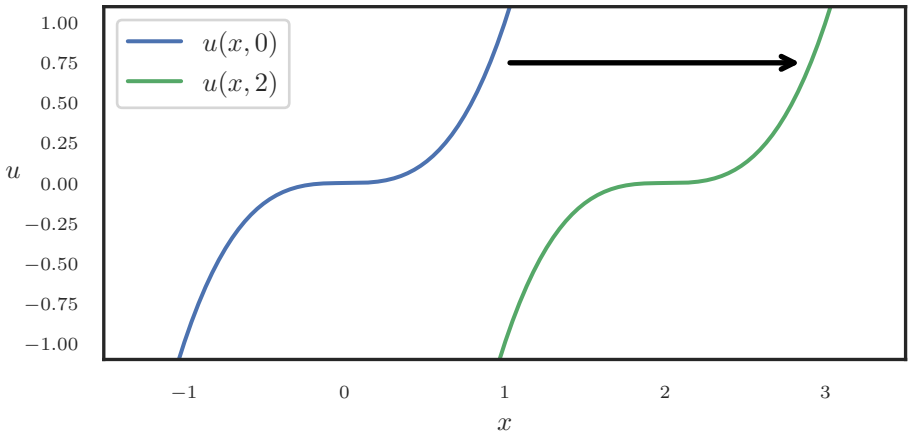

Figure 4.1: The solution to $u_t + u_x = 0,\ u(x,0) = x^3$ plotted in the xu-plane. Demonstrates simple transport of the solution.

can be transformed when restricting to the family of lines $x(t) = x_0 + ct$. On these lines $u(x(t), t)$ is constant, by construction, and so the solution is "transported" from $u(x_0, 0)$ along each characteristic line (Figure 4.1).

Motivated by this, **Lagrangian methods** treat each point in the physical domain as a "particle" which moves along a characteristic curve over time and then monitor values associated with the particle (heat / energy, velocity, pressure, density, concentration, etc.). They are an effective way to solve PDEs, even with higher order or non-linear terms. For example, if a viscosity term is added to (4.1)

$$u_t + cu_x - \varepsilon u_{xx} = 0 \tag{4.2}$$

then the same characteristics can be used, but the value along each characteristic is no longer constant; instead it satisfies the ODE $\frac{d}{dt}u(x(t), t) = \varepsilon u_{xx}$.

This approach transforms the numerical solution of PDEs into a family of numerical solutions to many independent ODEs. It allows the use of familiar and well understood ODE solvers. In addition, Lagrangian methods often have less restrictive conditions on time steps than Eulerian methods[1]. When solving PDEs on unstructured meshes with Lagrangian methods, the nodes move (since they are treated like particles) and the mesh "travels".

4.1.2 Remeshing and Adaptivity

A flow-based change to a mesh can cause problems if it causes the mesh to leave the domain being analyzed or if it distorts the mesh until the element quality is too low in some mesh elements. Over enough time, the mesh can even tangle (i.e. elements begin to overlap).

[1] In Eulerian methods, the mesh is fixed.

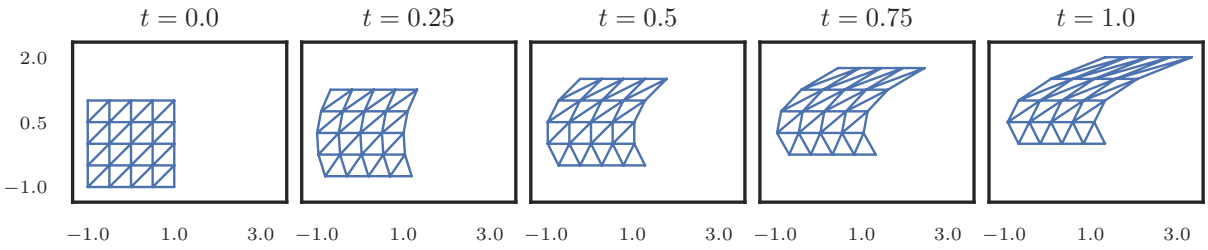

Figure 4.2: Distortion of a regular mesh caused by particle motion along the velocity field $[y^2\ 1]^T$ from $t = 0$ to $t = 1$ with $\Delta t = 1/4$.

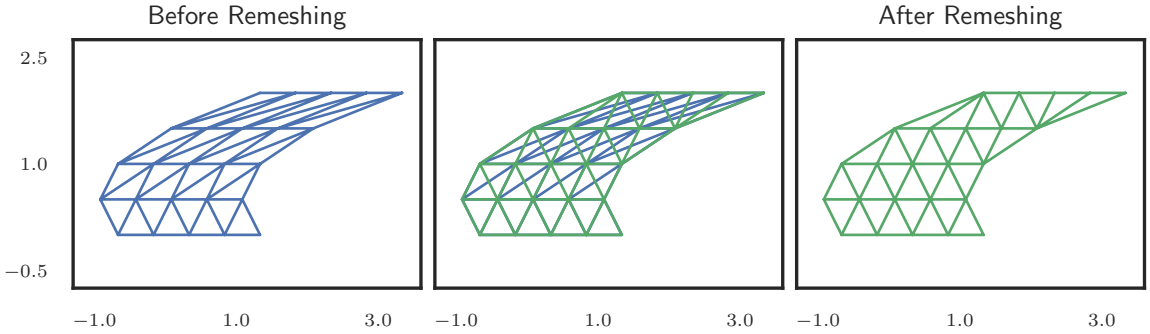

Figure 4.3: Remeshing a domain after distortion caused by particle motion along the velocity field $[y^2\ 1]^T$ from $t = 0$ to $t = 1$.

For an example of such distortion (Figure 4.2), consider a PDE of the form

$$u_t + \begin{bmatrix} y^2 \\ 1 \end{bmatrix} \cdot \nabla u + F\left(u, \nabla u\right) = 0. \tag{4.3}$$

The characteristics $y(t) = y_0 + t, x(t) = x_0 + \left(y(t)^3 - y_0^3\right)/3$ distort the mesh considerably after just one second.

To deal with distortion, one can allow the mesh to adapt in between time steps. For example, Figure 4.3 shows an example remeshing of the domain. In addition to improving mesh quality, mesh adaptivity can be used to dynamically focus computational effort to resolve sensitive features of a numerical solution. From [IK04]

> In order to balance the method's approximation quality and its computational costs effectively, adaptivity is an essential requirement, especially when modelling multiscale phenomena.

For more on mesh adaptivity, see [BR78, PVMZ87, PUdOG01].

In either case, the change in the mesh between time steps requires transferring a known solution on the discarded mesh to the mesh produced by the remeshing process. Without the

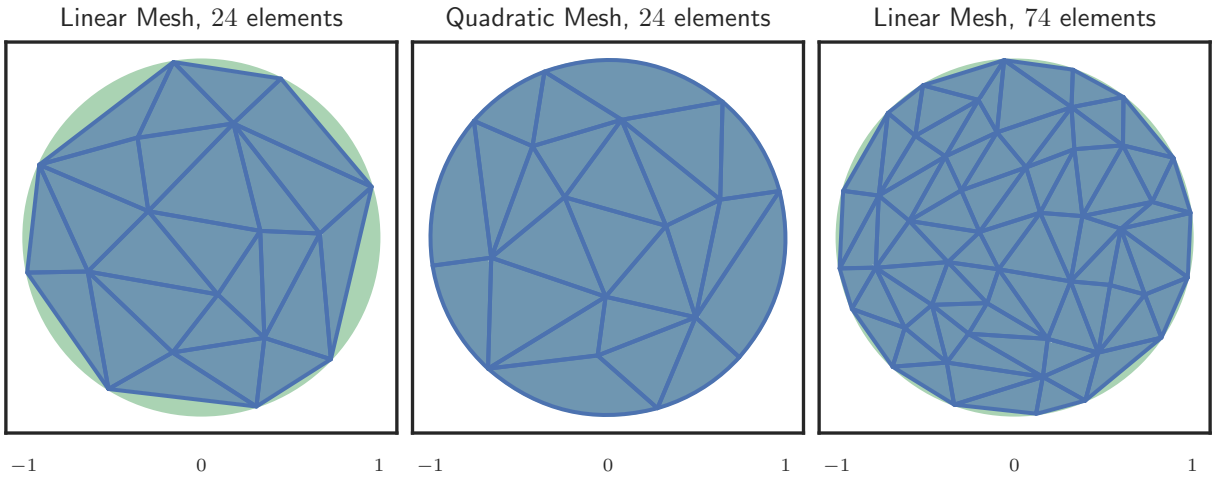

Figure 4.4: Comparing straight sided meshes to a curved mesh when approximating the unit disc in $\mathbf{R}^2$.

ability to change the mesh, Lagrangian methods (or, more generally, ALE [HAC74]) would not be useful, since after a limited time the mesh will distort.

4.1.3 High-order Meshes

To allow for greater geometric flexibility and for high order of convergence, curved mesh elements can be used in the finite element method. Though the complexity of a method can steeply rise when allowing curved elements, the trade for high-order convergence can be worth it. (See [WFA+13] for more on high-order CFD methods.) Curved meshes can typically represent a given geometry with far fewer elements than a straight sided mesh (for example, Figure 4.4). The increase in accuracy also allows for the use of fewer elements, which in turn can also facilitate a reduction in the overall computation time.

Even if the domain has no inherent curvature, high-order (degree p) shape functions allow for order $p+1$ convergence, which is desirable in it's own right. However, even in such cases, a Lagrangian method must either curve the mesh or information about the flow of the geometry will be lost. Figure 4.5 shows what happens to a given quadratic element as the nodes move along the characteristics from (4.3). This element uses the triangle vertices and edge midpoints to determine the shape functions. However, as the nodes move with the flow, the midpoints are no longer on the lines connecting the vertex nodes. To allow the mesh to more accurately represent the solution, the edges can instead curve so that the midpoint nodes remain halfway between (i.e. half of the parameter space) the vertex nodes along an edge.

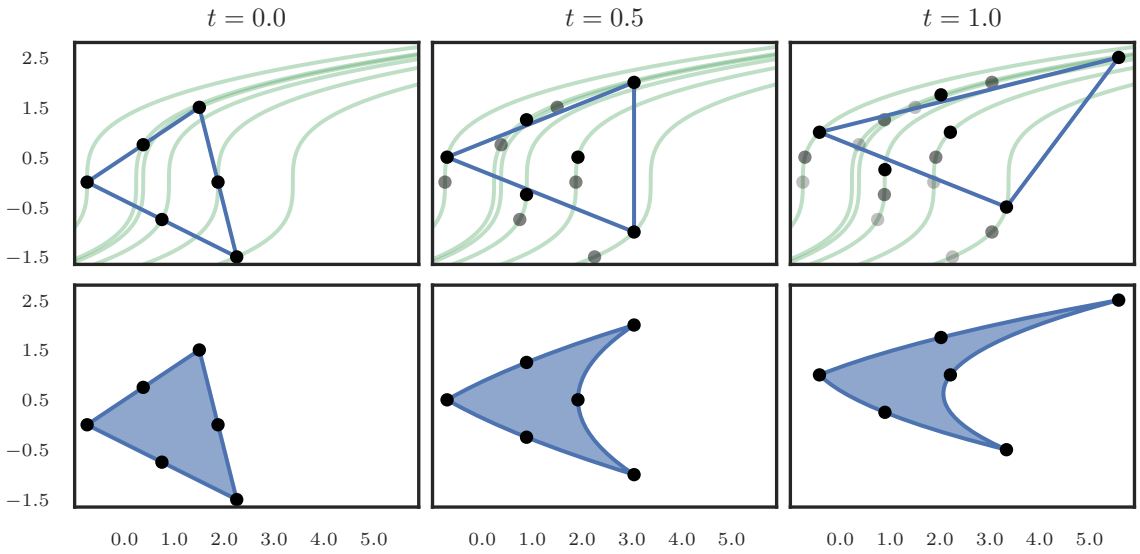

Figure 4.5: Movement of nodes in a quadratic element under distortion caused by particle motion along the velocity field $[y^2\ 1]^T$ from $t = 0$ to $t = 1$ with $\Delta t = 1/2$. The green curves represent the characteristics that each node travels along.

4.1.4 Multiphysics and Comparing Methods

In multiphysics simulations, a problem is partitioned into physical components. This partitioning can apply to both the physical domain (e.g. separating a solid and fluid at an interface) and the simulation solution itself (e.g. solving for pressure on one mesh and velocity on another). Each (multi)physics component is solved for on its own mesh. When the components interact, the simulation solution must be transferred between those meshes.

In a similar category of application, solution transfer enables the comparison of solutions defined on different meshes. For example, if a reference solution is known on a very fine special-purpose mesh, the error can be computed for a coarse mesh by transferring the solution from the fine mesh and taking the difference. Or, if the same method is used on different meshes of the same domain, the resulting computed solutions can be compared via solution transfer. Or, if two different methods use two different meshes of the same domain.

4.1.5 Local versus Global Transfer

Conservative solution transfer has been around since the advent of ALE, and as a result much of the existing literature focuses on mesh-mesh pairs that will occur during an ALE-based simulation. When flow-based mesh distortion occurs, elements are typically "flipped" (e.g. a diagonal is switched in a pair of elements) or elements are subdivided or combined. These operations are inherently local, hence the solution transfer can be done locally across known neighbors. Typically, this locality is crucial to solution transfer methods. In [MS03],

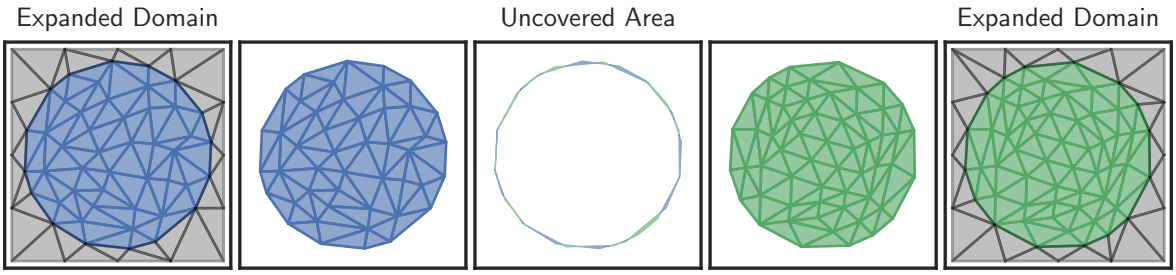

Figure 4.6: Partially overlapping meshes on a near identical domain. Both are linear meshes that approximate the unit disc in $\mathbf{R}^2$. The outermost columns show how the domain of each mesh can be expanded so they agree.

the transfer is based on partitioning elements of the updated mesh into components of elements from the old mesh and "swept regions" from neighbouring elements. In [KS08], the (locally) changing connectivity of the mesh is addressed. In [GKS07], the local transfer is done on polyhedral meshes.

Global solution transfer instead seeks to conserve the solution across the whole mesh. It makes no assumptions about the relationship between the donor and target meshes. The loss in local information makes the mesh intersection problem more computationally expensive, but the added flexibility reduces timestep restrictions since it allows remeshing to be done less often. In [Duk84, DK87], a global transfer is enabled by transforming volume integrals to surface integrals via the divergence theorem to reduce the complexity of the problem.

4.1.6 Limitations

The method described in this work only applies to meshes in $\mathbf{R}^2$. Application to meshes in $\mathbf{R}^3$ is a direction for future research, though the geometric kernels (see Chapter 3) become significantly more challenging to describe and implement. In addition, the method will assume that every element in the target mesh is contained in the donor mesh. This ensures that the solution transfer is *interpolation*. In the case where all target elements are partially covered, *extrapolation* could be used to extend a solution outside the domain, but for totally uncovered elements there is no clear correspondence to elements in the donor mesh.

The case of partially overlapping meshes can be addressed in particular cases (i.e. with more information). For example, consider a problem defined on $\Omega = \mathbf{R}^2$ and solution that tends towards zero as points tend to infinity. A typical approach may be to compute the solution on a circle of large enough radius and consider the numerical solution to be zero outside the circle. Figure 4.6 shows how solution transfer could be performed in such cases when the meshes partially overlap: construct a simple region containing both computational domains and then mesh the newly introduced area. However, the assumption that the numerical solution is zero in the newly introduced area is very specific and a similar approach may not apply in other cases of partial overlap.

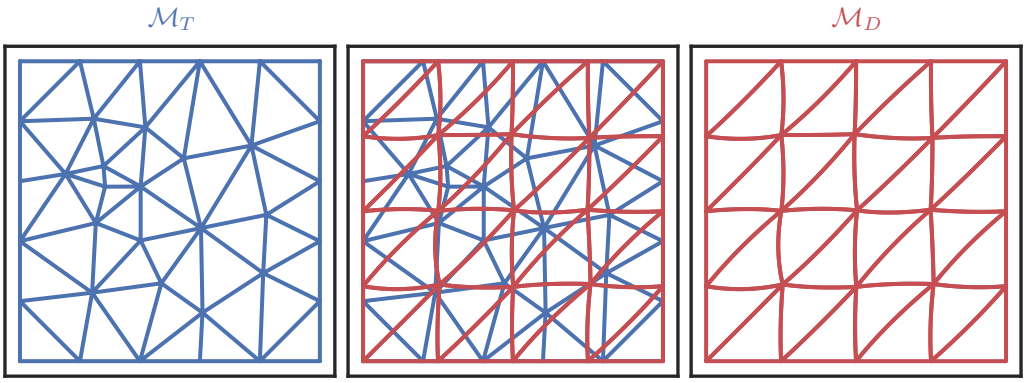

Figure 4.7: Mesh pair: donor mesh $\mathcal{M}_D$ and target mesh $\mathcal{M}_T$.

Some attempts ([Ber87, CH94, CDS99]) have been made to interpolate fluxes between overlapping meshes. These perform an interpolation on the region common to both meshes and then numerically solve the PDE to determine the values on the uncovered elements.

4.2 Galerkin Projection

Consider a donor mesh $\mathcal{M}_D$ with shape function basis $\phi_D^{(j)}$ and a known field $\boldsymbol{q}_D = \sum_j d_j \phi_D^{(j)}$ [2] and a target mesh $\mathcal{M}_T$ with shape function basis $\phi_T^{(j)}$ (Figure 4.7). Each shape function ϕ corresponds to a given isoparametric curved element (see Section 2.5) $\mathcal{T}$ in one of these meshes and has $\text{supp}(\phi) = \mathcal{T}$. Additionally, the shape functions are polynomial degree p (see Section 2.5.1 for a discussion of shape functions), but the degree of the donor mesh need not be the same as that of the target mesh. We assume that both meshes cover the same domain $\Omega \subset \mathbf{R}^2$, however we really only require the donor mesh to cover the target mesh.

We seek the L_2-optimal interpolant $\boldsymbol{q}_T = \sum_j t_j \phi_T^{(j)}$:

$$\|\boldsymbol{q}_T - \boldsymbol{q}_D\|_2 = \min_{\boldsymbol{q} \in \mathcal{V}_T} \|\boldsymbol{q} - \boldsymbol{q}_D\|_2 \tag{4.4}$$

where $\mathcal{V}_T = \text{Span}_j \left\{ \phi_T^{(j)} \right\}$ is the function space defined on the target mesh. Since this is optimal in the L_2 sense, by differentiating with respect to each t_j in $\boldsymbol{q}_T$ we find the weak form:

$$\int_\Omega \boldsymbol{q}_D \phi_T^{(j)} \, dV = \int_\Omega \boldsymbol{q}_T \phi_T^{(j)} \, dV, \qquad \text{for all} j. \tag{4.5}$$

[2] This is somewhat a simplification. In the CG case, some coefficients will be paired with multiple shape functions, such as the coefficient at a vertex node.

If the constant function 1 is contained in $\mathcal{V}_T$, conservation follows from the weak form and linearity of the integral

$$\int_\Omega \boldsymbol{q}_D \, dV = \int_\Omega \boldsymbol{q}_T \, dV. \tag{4.6}$$

Expanding $\boldsymbol{q}_D$ and $\boldsymbol{q}_T$ with respect to their coefficients $\boldsymbol{d}$ and $\boldsymbol{t}$, the weak form gives rise to a linear system

$$M_T \boldsymbol{t} = M_{TD} \boldsymbol{d}. \tag{4.7}$$

Here M_T is the mass matrix for $\mathcal{M}_T$ given by

$$(M_T)_{ij} = \int_\Omega \phi_T^{(i)} \phi_T^{(j)} \, dV. \tag{4.8}$$

In the discontinuous Galerkin case, M_T is block diagonal with blocks that correspond to each element, so (4.7) can be solved locally on each element $\mathcal{T}$ in the target mesh. By construction, M_T is symmetric and sparse since $(M_T)_{ij}$ will be 0 unless $\phi_T^{(i)}$ and $\phi_T^{(j)}$ are supported on the same element $\mathcal{T}$. In the continuous case, M_T is globally coupled since coefficients corresponding to boundary nodes interact with multiple elements. The matrix M_{TD} is a "mixed" mass matrix between the target and donor meshes:

$$(M_{TD})_{ij} = \int_\Omega \phi_T^{(i)} \phi_D^{(j)} \, dV. \tag{4.9}$$

Boundary conditions can be imposed on the system by fixing some coefficients, but that is equivalent to removing some of the basis functions which may in term make the projection non-conservative. This is because the removed basis functions may have been used in $1 = \sum_j u_j \phi_T^{(j)}$.

Computing M_T is fairly straightforward since the (bidirectional) mapping from elements $\mathcal{T}$ to basis functions $\phi_T^{(j)}$ supported on those elements is known. When using shape functions in the global coordinates basis (see Section 2.5.1), the integrand $F = \phi_T^{(i)} \phi_T^{(j)}$ will be a polynomial of degree $2p$ on $\mathbf{R}^2$. The domain of integration $\mathcal{T} = b(\mathcal{U})$ is the image of a (degree p) map $b(s,t)$ from the unit triangle. Using substitution

$$\int_{b(\mathcal{U})} F(x,y) \, dx \, dy = \int_{\mathcal{U}} \det(Db) F\left(x(s,t), y(s,t)\right) \, ds \, dt \tag{4.10}$$

(we know the map preserves orientation, i.e. $\det(Db)$ is positive). Once transformed this way, a quadrature rule on the unit triangle ([Dun85]) can be used.

On the other hand, computing M_{TD} is significantly more challenging. This requires solving both a geometric problem — finding the region to integrate over — and an analytic problem — computing the integrals. The integration can be done with a quadrature rule, though finding this region is significantly more difficult.

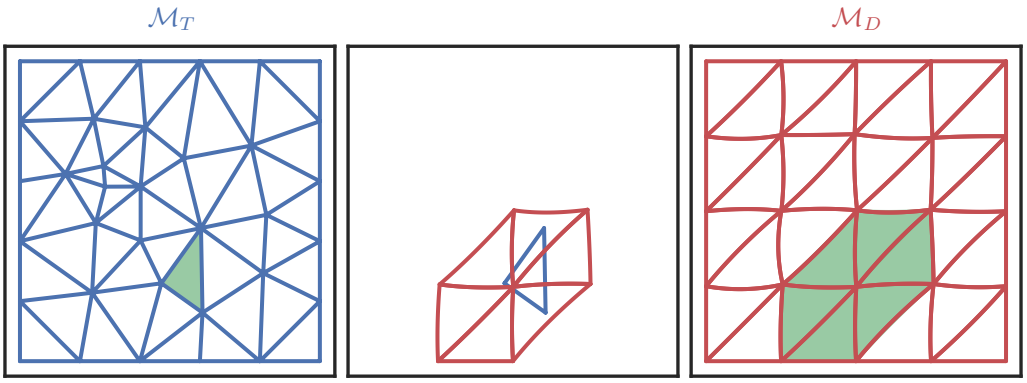

Figure 4.8: All donor elements that cover a target element

4.3 Common Refinement

Rather than computing M_{TD}, the right-hand side of (4.7) can be computed directly via

$$(M_{TD}\boldsymbol{d})_j = \int_\Omega \phi_T^{(j)} \boldsymbol{q}_D \, dV. \tag{4.11}$$

Any given ϕ is supported on an element $\mathcal{T}$ in the target mesh. Since $\boldsymbol{q}_D$ is piecewise defined over each element $\mathcal{T}'$ in the donor mesh, the integral (4.11) may be problematic. In the continuous Galerkin case, $\boldsymbol{q}_D$ need not be differentiable across $\mathcal{T}$ and in the discontinuous Galerkin case, $\boldsymbol{q}_D$ need not even be continuous. This necessitates a partitioning of the domain:

$$\int_\Omega \phi \, \boldsymbol{q}_D \, dV = \int_\mathcal{T} \phi \, \boldsymbol{q}_D \, dV = \sum_{\mathcal{T}' \in \mathcal{M}_D} \int_{\mathcal{T} \cap \mathcal{T}'} \phi \, \boldsymbol{q}_D|_{\mathcal{T}'} \, dV. \tag{4.12}$$

In other words, the integral over $\mathcal{T}$ splits into integrals over intersections $\mathcal{T} \cap \mathcal{T}'$ for all $\mathcal{T}'$ in the donor mesh that intersect $\mathcal{T}$ (Figure 4.8). Since both ϕ and $\boldsymbol{q}_D|_{\mathcal{T}'}$ are polynomials on $\mathcal{T} \cap \mathcal{T}'$, the integrals will be exact when using a quadrature scheme of an appropriate degree of accuracy. Without partitioning $\mathcal{T}$, the integrand is not a polynomial (in fact, possibly not smooth), so the quadrature cannot be exact.

In order to compute $M_{TD}\boldsymbol{d}$, we'll need to compute the **common refinement**, i.e. an intermediate mesh that contains both the donor and target meshes. This will consist of all non-empty $\mathcal{T} \cap \mathcal{T}'$ as $\mathcal{T}$ varies over elements of the target mesh and $\mathcal{T}'$ over elements of the donor mesh. This requires solving three specific subproblems:

- Forming the region(s) of intersection between two elements that are Bézier triangles.
- Finding all pairs of elements, one each from the target and donor mesh, that intersect.
- Numerically integrating over a region of intersection between two elements.

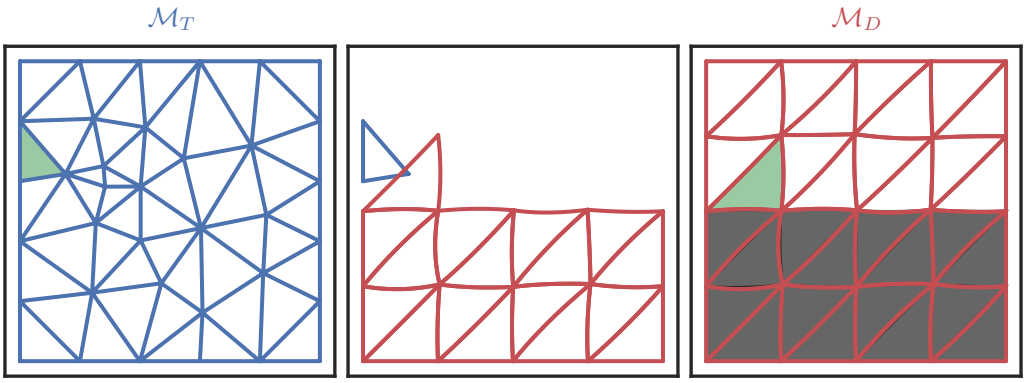

Figure 4.9: Brute force search for a donor element $\mathcal{T}'$ that matches a fixed target element $\mathcal{T}$.

The first subproblem has been addressed in Section 3.2 and the curved polygon region(s) of intersection have been described in Section 2.5.2. The second will be considered in Section 4.3.1 and the third in Section 4.3.2 below.

4.3.1 Advancing Front

We seek to identify all pairs $\mathcal{T}$ and $\mathcal{T}'$ of intersecting target and donor elements. The naïve approach just considers every pair of elements and takes $\mathcal{O}(|\mathcal{M}_D|\,|\mathcal{M}_T|)$ to complete.[3] Taking after [FM11], we can do much better than this quadratic time search. In fact, we can compute all integrals in $\mathcal{O}(|\mathcal{M}_D| + |\mathcal{M}_T|)$. First, we fix an element of the target mesh and perform a brute-force search to find an intersecting element in the donor mesh (Figure 4.9). This has worst-case time $\mathcal{O}(|\mathcal{M}_D|)$.

Once we have such a match, we use the connectivity graph of the donor mesh to perform a breadth first search for neighbors that also intersect the target element $\mathcal{T}$ (Figure 4.10). This search takes $\mathcal{O}(1)$ time. It's also worthwhile to keep the first layer of donor elements that don't intersect $\mathcal{T}$ because they are more likely to intersect the neighbors of $\mathcal{T}$[4]. Using the list of intersected elements, a neighbor of $\mathcal{T}$ can find a donor element it intersects with in $\mathcal{O}(1)$ time (Figure 4.11). As seen, after our $\mathcal{O}(|\mathcal{M}_D|)$ initial brute-force search, the localized intersections for each target element $\mathcal{T}$ take $\mathcal{O}(1)$ time. So together, the process takes $\mathcal{O}(|\mathcal{M}_D| + |\mathcal{M}_T|)$.

For special cases, e.g. ALE methods, the initial brute-force search can be reduced to $\mathcal{O}(1)$. This could be enabled by tracking the remeshing process so a correspondence already exists. If a full mapping from donor to target mesh exists, the process of computing M_T

[3] For a mesh $\mathcal{M}$, the expression $|\mathcal{M}|$ represents the number of elements in the mesh.

[4] In the very unlikely case that the boundary of $\mathcal{T}$ **exactly matches** the boundaries of the donor elements that cover it, **none** of the overlapping donor elements can intersect the neighbors of $\mathcal{T}$ so the first layer of non-intersected donor elements must be considered.

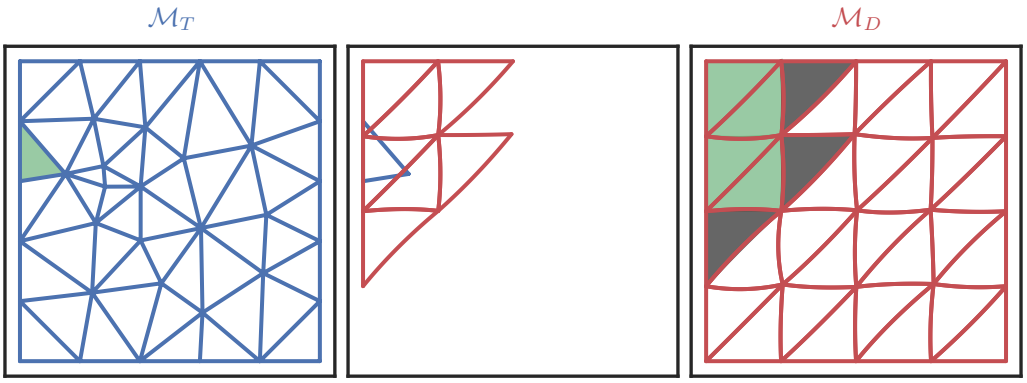

Figure 4.10: All donor elements $\mathcal{T}'$ that cover a target element $\mathcal{T}$, with an extra layer of neighbors in the donor mesh that **do not** intersect $\mathcal{T}$.

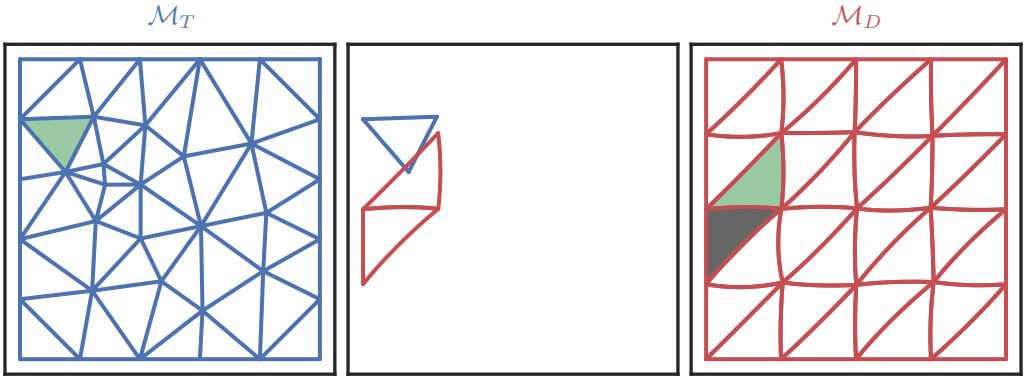

Figure 4.11: First match between a neighbor of the previously considered target element and all the donor elements that match the previously considered target element.

and $M_{TD}\boldsymbol{d}$ can be fully parallelized across elements of the target mesh, or even across shape functions $\phi_T^{(j)}$.

4.3.2 Integration over Curved Polygons

In order to numerically evaluate integrals of the form

$$\int_{\mathcal{T}_0 \cap \mathcal{T}_1} F(x, y)\, dV \tag{4.13}$$

we must have a quadrature rule on these curved polygon (Section 2.5.2) intersections $\mathcal{T}_0 \cap \mathcal{T}_1$. To do this, we transform the integral into several line integrals via Green's theorem and then use an exact Gaussian quadrature to evaluate them. Throughout this section we'll assume

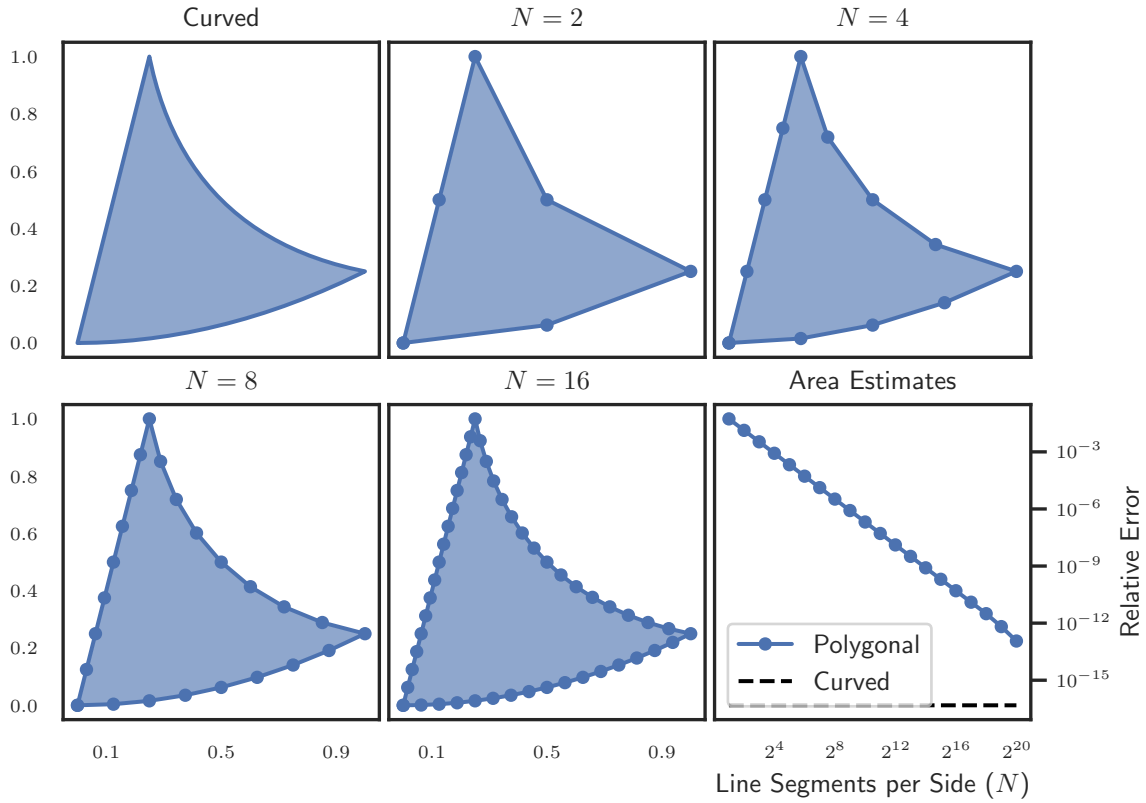

Figure 4.12: Comparing the relative error for the computed area of a quadratic Bézier triangle. In one method, the curved boundary is used and the result is correct to machine precision. In the other, the curved edges are approximated by polygonal paths.

the integrand is of the form $F = \phi_0\phi_1$ where each ϕ_j is a shape function on $\mathcal{T}_j$. In addition, we'll refer to the two Bézier maps that define the elements being intersected: $\mathcal{T}_0 = b_0(\mathcal{U})$ and $\mathcal{T}_1 = b_1(\mathcal{U})$.

First a discussion of a method not used. A somewhat simple approach would be be to use polygonal approximation. I.e. approximate the boundary of each Bézier triangle with line segments, intersect the resulting polygons, triangulate the intersection polygon(s) and then numerically integrate on each triangulated cell. However, this approach is prohibitively inefficient. For example, consider the curved polygon in Figure 4.12. Computing the area of $\mathcal{P}$ can be done via an integral: $\int_{\mathcal{P}} 1\,dV$. By using the actual curved boundary, this integral can be computed with relative error (the dashed line in the bottom right subplot) on the order of machine precision $\mathcal{O}(\mathbf{u})$. On the other hand, approximating each side of $\mathcal{P}$ with N line segments, the relative error is $\mathcal{O}(1/N^2)$. (For example, if $N = 2$ an edge curve $b(s, 0)$ would be replaced by segments connecting $b(0, 0), b(1/2, 0)$ and $b(1, 0)$, i.e. $N + 1$ equally spaced parameters.) This means that in order to perform as well as an **exact** quadrature

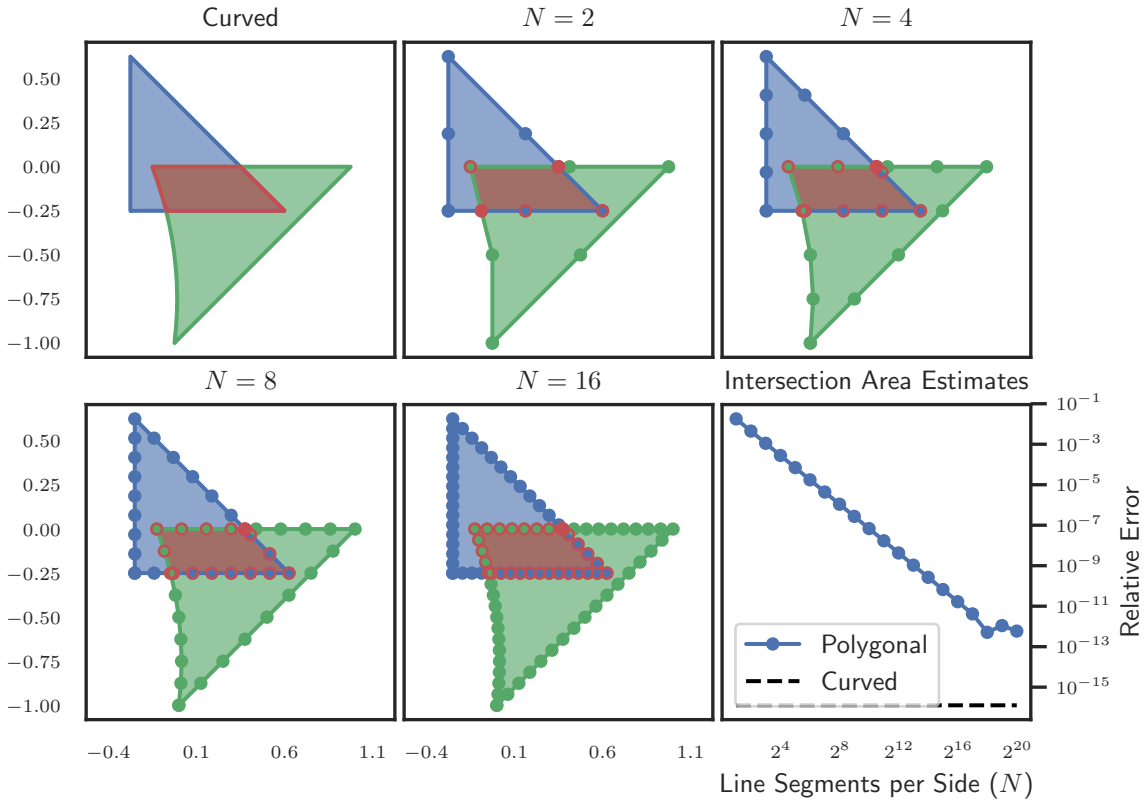

Figure 4.13: Comparing the relative error for the computed area of the intersection of two quadratic Bézier triangles. In one method, the intersection boundary is fully specified as the union of Bézier curve segments and the area is computed correctly to machine precision. In the other, the curved edges are approximated by polygonal paths and the intersection of the resulting polygons is computed.

used in the curved case, we'd need $N = \mathcal{O}\left(1/\sqrt{\mathbf{u}}\right)$. Compute the area of $\mathcal{P}$ in this way assumes that $\mathcal{P}$ is already known. If instead only the Bézier triangles in the intersection are known, as in Figure 4.13, the polygonal approach suffers from the same inefficiency.

Since polygonal approximation is prohibitively expensive, we work directly with curved edges and compute integrals on a regular domain via substitution. If two Bézier triangles intersect with positive measure, then the region of intersection is one or more disjoint curved polygons: $\mathcal{T}_0 \cap \mathcal{T}_1 = \mathcal{P}$ or $\mathcal{T}_0 \cap \mathcal{T}_1 = \mathcal{P} \cup \mathcal{P}' \cup \cdots$. The second case can be handled in the same way as the first by handling each disjoint region independently:

$$\int_{\mathcal{T}_0 \cap \mathcal{T}_1} F(x, y)\, dV = \int_{\mathcal{P}} F(x, y)\, dV + \int_{\mathcal{P}'} F(x, y)\, dV + \cdots . \tag{4.14}$$

Each curved polygon $\mathcal{P}$ is defined by its boundary, a piecewise smooth parametric curve:

$$\partial \mathcal{P} = C_1 \cup \cdots \cup C_n. \tag{4.15}$$

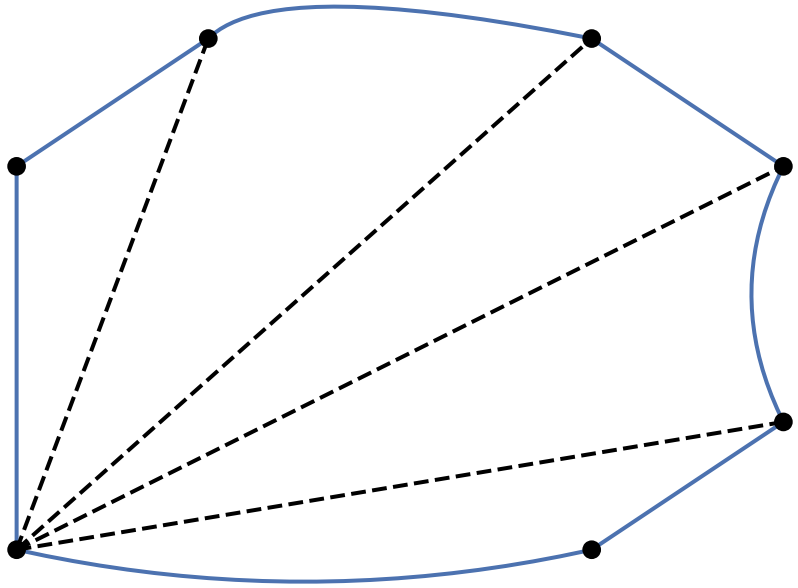

Figure 4.14: Curved polygon tessellation, done by introducing diagonals from a single vertex node.

This can be thought of as a polygon with n-sides that happens to have curved edges.

Quadrature rules for straight sided polygons have been studied ([MXS09]) though they are not in wide use. Even if a polygonal quadrature rule was to be employed, a map would need to be established from a reference polygon onto the curved edges. This map could then be used with a change of coordinates to move the integral from the curved polygon to the reference polygon. The problem of extending a mapping from a boundary to an entire domain has been studied as transfinite interpolation ([Che74, GT82, Per98]), barycentric coordinates ([Wac75]) or mean value coordinates ([Flo03]). However, these maps aren't typically suitable for numerical integration because they are either not bijective or increase the degree of the edges.

Since simple and well established quadrature rules do exist for triangles, a valid approach would be to tessellate a curved polygon into valid Bézier triangles (Figure 4.14) and then use substitution as in (4.10). However, tessellation is challenging for both theoretical and computational reasons. Theoretical: it's not even clear if an arbitrarily curved polygon **can** be tessellated into Bézier triangles. Computational: the placement of diagonals and potential introduction of interior nodes is very complex in cases where the curved polygon is nonconvex. What's more, the curved polygon is given only by the boundary, so higher degree triangles (i.e. cubic and above) introduced during tessellation would need to place interior control points without causing the triangle to invert. To get a sense for these challenges, note how the "simple" introduction of diagonals in Figure 4.15 leads to one inverted element (the gray element) and another element with area outside of the curved polygon (the yellow

element). Inverted Bézier triangles are problematic because the accompanying mapping leaves the boundary established by the edge curves. For example, if the tessellation of $\mathcal{P}$ contains an inverted Bézier triangle $\mathcal{T}_2 = b(\mathcal{U})$ then we'll need to numerically integrate

$$\int_{\mathcal{T}_2} \phi_0\phi_1 \, dx \, dy = \int_{\mathcal{U}} |\det(Db)| \, (\phi_0 \circ b) \, (\phi_1 \circ b) \, ds \, dt. \tag{4.16}$$

If the shape functions are from the pre-image basis (see Section 2.5.1), then ϕ_0 will not be defined at $b(s,t) \notin \mathcal{T}_0$ (similarly for ϕ_1). Additionally, the absolute value in $|\det(Db)|$ makes the integrand non-smooth since for inverted elements $\det(Db)$ takes both signs. If the shape functions are from the global coordinates basis, then tessellation can be used via an application of Lemma B.1, however this involves more computation than just applying Green's theorem along $\partial\mathcal{P}$. By applying the lemma, inverted elements may be used in a tessellation, for example by introducing artificial diagonals from a vertex as in Figure 4.14. In the global coordinates basis, $F = \phi_0\phi_1$ *can* be evaluated for points in an inverted element that leave $\mathcal{T}_0$ or $\mathcal{T}_1$.

Instead, we focus on a Green's theorem based approach. Define horizontal and vertical antiderivatives H, V of the integrand F such that $H_x = V_y = F$. We make these *unique* by imposing the extra condition that $H(0,y) \equiv V(x,0) \equiv 0$. This distinction is arbitrary, but in order to *evaluate* H and V, the univariate functions $H(0,y)$ and $V(x,0)$ must be specified. Green's theorem tells us that

$$\int_{\mathcal{P}} 2F \, dV = \int_{\mathcal{P}} H_x + V_y \, dV = \oint_{\partial\mathcal{P}} H \, dy - V \, dx = \sum_j \int_{C_j} H \, dy - V \, dx. \tag{4.17}$$

For a given curve C with components $x(r), y(r)$ defined on the unit interval, this amounts to having to integrate

$$G(r) = H(x(r), y(r))y'(r) - V(x(r), y(r))x'(r). \tag{4.18}$$

To do this, we'll use Gaussian quadrature with degree of accuracy sufficient to cover the degree of $G(r)$.

If the shape functions are in the global coordinates basis (Section 2.5.1), then G will also be a polynomial. This is because for these shape functions $F = \phi_0\phi_1$ will be polynomial on $\mathbf{R}^2$ and so will H and V and since each curve C is a Bézier curve segment, the components are also polynomials. If the shape functions are in the pre-image basis, then F won't in general be polynomial, hence the quadrature rules can only be approximate.

To evaluate G, we must also evaluate H and V numerically. For example, since $H(0,y) \equiv 0$, the fundamental theorem of calculus tells us that $H(\alpha,\beta) = \int_0^\alpha F(x,\beta) \, dx$. To compute this integral via Gaussian quadrature

$$H(\alpha,\beta) \approx \frac{\alpha}{2} \sum_j w_j F\left(\frac{\alpha}{2}(x_j + 1), \beta\right) \tag{4.19}$$

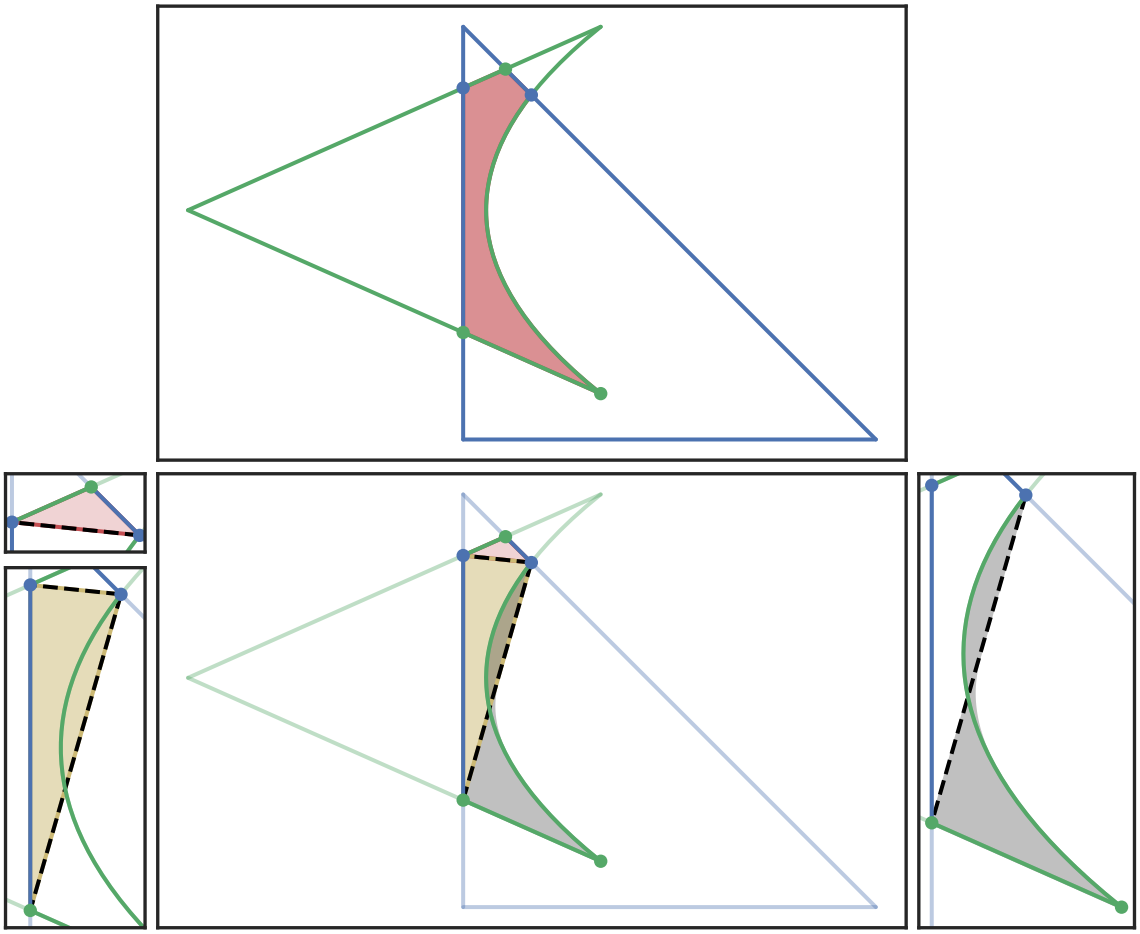

Figure 4.15: Curved polygon intersection that can't be tessellated (with valid Bézier triangles) by introducing diagonals.

we must be able to evaluate F for points on the line $y = \beta$ for x between 0 and α. If the shape functions are from the pre-image basis, it may not even be possible to evaluate F for such points. Since $\begin{bmatrix} \alpha & \beta \end{bmatrix}^T$ is on the boundary of $\mathcal{P}$, WLOG assume it is on the boundary of $\mathcal{T}_0$. Thus, for some elements (e.g. if the point is on the bottom of the element), points $\begin{bmatrix} \nu & \beta \end{bmatrix}^T$ may not be in $\mathcal{T}_0$. Since $\operatorname{supp}(\phi_0) = \mathcal{T}_0$, we could take $\phi_0(\nu, \beta) = 0$, but this would make the integrand non-smooth and so the accuracy of the *exact* quadrature would be lost. But extending $\phi_0 = \widehat{\phi}_0 \circ b_0^{-1}$ outside of $\mathcal{T}_0$ may be impossible: even though b_0 is bijective on $\mathcal{U}$ it may be many-to-one elsewhere hence b_0^{-1} can't be reliably extended outside of $\mathcal{T}_0$.

Even if the shape functions are from the global coordinates basis, setting $H(0, y) \equiv 0$ may introduce quadrature points that are very far from $\mathcal{P}$. This can be somewhat addressed by using $H(m, y) \equiv 0$ for a suitably chosen m (e.g. the minimum x-value in $\mathcal{P}$). Then we have $H(\alpha, \beta) = \int_m^\alpha F(x, \beta)\, dx$.

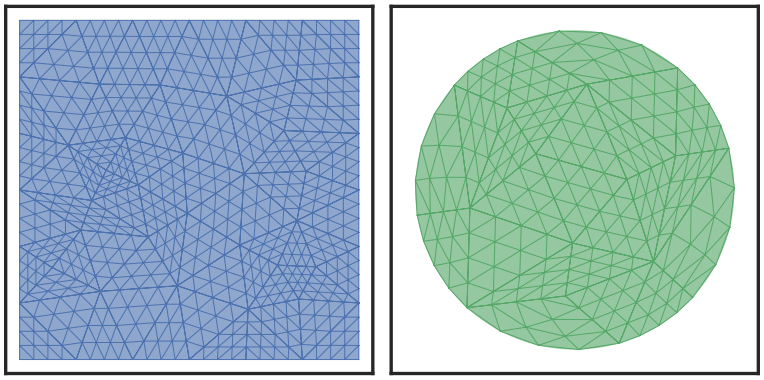

Figure 4.16: Some example meshes used during the numerical experiments; the donor meshs is on the left / blue and the target mesh is on the right / green. These meshes are the cubic approximations of each domain and have been refined twice.

4.4 Numerical Experiments

A numerical experiment was performed to investigate the observed order of convergence. Three pairs of random meshes were generated, the donor on a square of width 17/8 centered at the origin and the target on the unit disc. These domains were chosen intentionally so that the target mesh was completely covered by the donor mesh and the boundaries did not accidentally introduce ill-conditioned intersection between elements. The pairs were linear, quadratic and cubic approximations of the domains. The convergence test was done by refining each pair of meshes four times and performing solution transfer at each level. Figure 4.16 shows the cubic pair of meshes after two refinements.

Taking after [FM11], we transfer three discrete fields in the discontinuous Galerkin (DG) basis. Each field is derived from one of the smooth scalar functions

$$\zeta_1(x, y) = 5y^3 + x^2 + 2y + 3 \tag{4.20}$$
$$\zeta_2(x, y) = \exp\left(x^2\right) + 2y \tag{4.21}$$
$$\zeta_3(x, y) = \sin(x) + \cos(y). \tag{4.22}$$

For a given mesh size h, we expect that on a degree p isoparametric mesh our solution transfer will have $\mathcal{O}\left(h^{p+1}\right)$ errors. To measure the rate of convergence:

- Choose a mesh pair $\mathcal{M}_D$, $\mathcal{M}_T$ and a known function $\zeta(x, y)$.
- Refine the meshes recursively, starting with $\mathcal{M}_D^{(0)} = \mathcal{M}_D$, $\mathcal{M}_T^{(0)} = \mathcal{M}_T$.
- Create meshes $\mathcal{M}_D^{(j)}$ and $\mathcal{M}_T^{(j)}$ from $\mathcal{M}_D^{(j-1)}$ and $\mathcal{M}_T^{(j-1)}$ by subdividing each curved element into four elements.
- Approximate ζ by a discrete field: the nodal interpolant on the donor mesh $\mathcal{M}_D^{(j)}$. The nodal interpolant is constructed by evaluating ζ at the nodes $\boldsymbol{n}_j$ corresponding to each

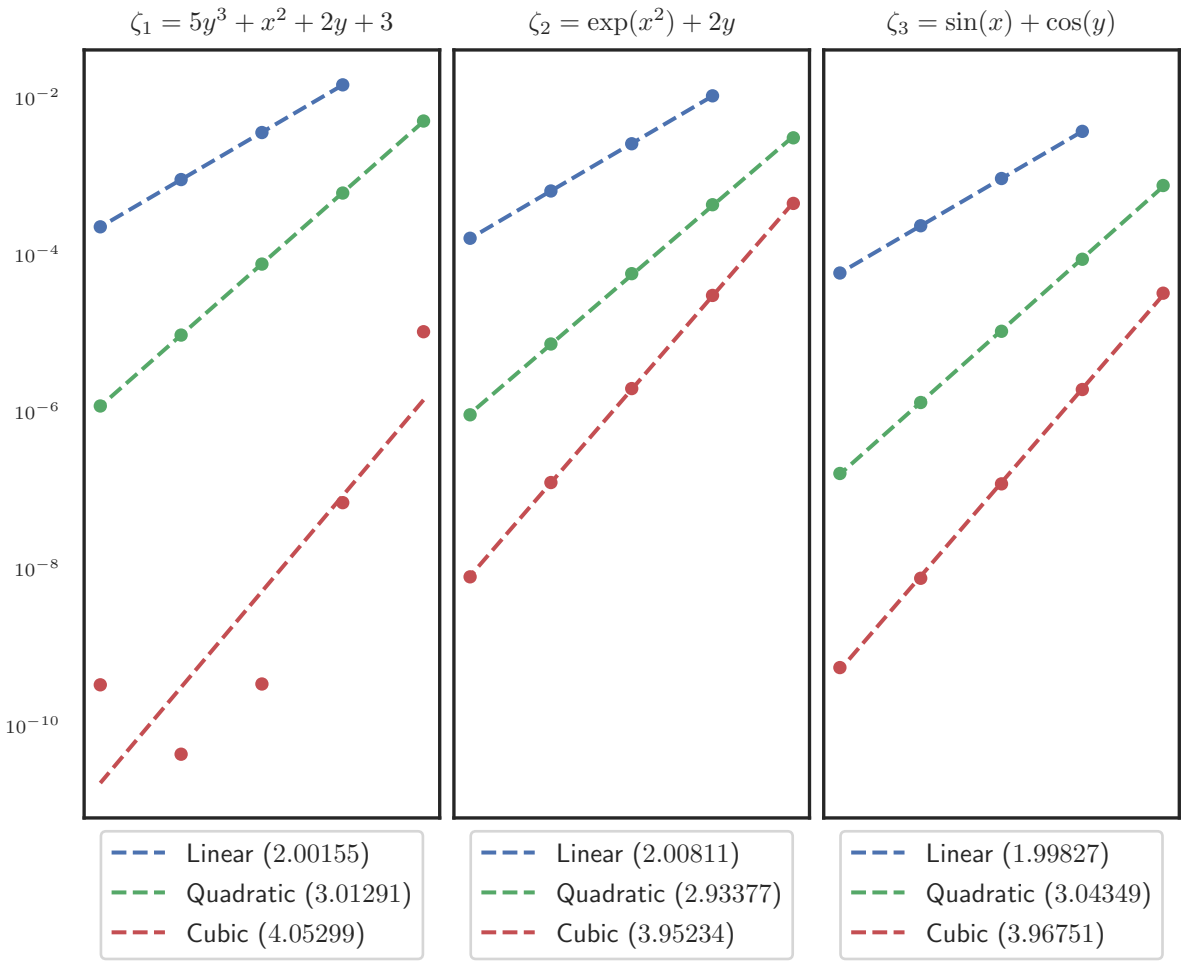

Figure 4.17: Convergence results for scalar fields on three pairs of related meshes: a linear, quadratic and cubic mesh of the same domain.

shape function:

$$\boldsymbol{f}_j = \sum_j \zeta\left(\boldsymbol{n}_j\right) \phi_D^{(j)}. \tag{4.23}$$

- Transfer $\boldsymbol{f}_j$ to the discrete field $\boldsymbol{g}_j$ on the target mesh $\mathcal{M}_T^{(j)}$.
- Compute the relative error on $\mathcal{M}_T^{(j)}$: $E_j = \|\boldsymbol{g}_j - \zeta\|_2 / \|\zeta\|_2$ (here $\|\cdot\|_2$ is the L_2 norm on the target mesh).

  We should instead be measuring $\|\boldsymbol{g}_j - \boldsymbol{f}_j\|_2 / \|\boldsymbol{f}_j\|_2$, but E_j is much easier to compute for $\zeta(x, y)$ that are straightforward to evaluate. Due to the triangle inequality E_j can be a reliable proxy for the actual projection error, though when $\|\boldsymbol{f}_j - \zeta\|_2$ becomes too large it will dominate the error and no convergence will be observed.

Convergence results are shown in Figure 4.17 and confirm the expected orders. Since ζ_1 is a cubic polynomial, the solution transfer on the cubic mesh should be **exact**, so there is a certain minimum threshold that can be reached.

Chapter 5

K-Compensated de Casteljau

5.1 Introduction

In computer aided geometric design, polynomials are usually expressed in Bernstein form. Polynomials in this form are usually evaluated by the de Casteljau algorithm. This algorithm has a round-off error bound which grows only linearly with degree, even though the number of arithmetic operations grows quadratically. The Bernstein basis is optimally suited ([FR87, DP15, MP05]) for polynomial evaluation; it is typically more accurate than the monomial basis, for example in Figure 5.1 evaluation via Horner's method produces a jagged curve for points near a triple root, but the de Casteljau algorithm produces a smooth curve. Nevertheless the de Casteljau algorithm returns results arbitrarily less accurate than the working precision $\mathbf{u}$ when evaluating $p(s)$ is ill-conditioned. The relative accuracy of the computed evaluation with the de Casteljau algorithm (`DeCasteljau`) satisfies ([MP99]) the following a priori bound:

$$\frac{|p(s) - \texttt{DeCasteljau}(p, s)|}{|p(s)|} \leq \operatorname{cond}(p, s) \times \mathcal{O}(\mathbf{u}). \tag{5.1}$$

In the right-hand side of this inequality, $\mathbf{u}$ is the computing precision and the condition number $\operatorname{cond}(p, s) \geq 1$ only depends on s and the Bernstein coefficients of p — its expression will be given further.

For ill-conditioned problems, such as evaluating $p(s)$ near a multiple root, the condition number may be arbitrarily large, i.e. $\operatorname{cond}(p, s) > 1/\mathbf{u}$, in which case most or all of the computed digits will be incorrect. In some cases, even the order of magnitude of the computed value of $p(s)$ can be incorrect.

To address ill-conditioned problems, error-free transformations (EFT) can be applied in *compensated algorithms* to account for round-off. Error-free transformations were studied in great detail in [ORO05] and open a large number of applications. In [LGL06], a compensated Horner's algorithm was described to evaluate a polynomial in the monomial basis. In [JLCS10], a similar method was described to perform a compensated version of the de Casteljau algorithm. In both cases, the $\operatorname{cond}(p, s)$ factor is moved from $\mathbf{u}$ to $\mathbf{u}^2$ and the

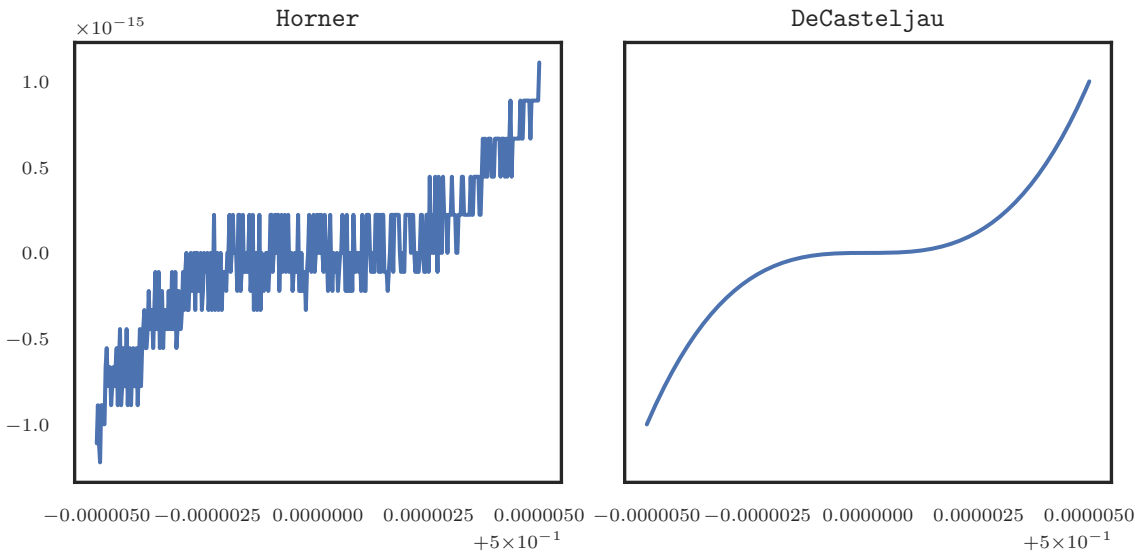

Figure 5.1: Comparing Horner's method to the de Casteljau method for evaluating $p(s) = (2s - 1)^3$ in the neighborhood of its multiple root $1/2$.

computed value is as accurate as if the computations were done in twice the working precision. For example, the compensated de Casteljau algorithm (`CompDeCasteljau`) satisfies

$$\frac{|p(s) - \texttt{CompDeCasteljau}(p, s)|}{|p(s)|} \leq \mathbf{u} + \operatorname{cond}(p, s) \times \mathcal{O}\left(\mathbf{u}^2\right). \tag{5.2}$$

For problems with $\operatorname{cond}(p, s) < 1/\mathbf{u}^2$, the relative error is $\mathbf{u}$, i.e. accurate to full precision, aside from rounding to the nearest floating point number. Figure 5.2 shows this shift in relative error from `DeCasteljau` to `CompDeCasteljau`.

In [GLL09], the authors generalized the compensated Horner's algorithm to produce a method for evaluating a polynomial as if the computations were done in K times the working precision for any $K \geq 2$. This result motivates this chapter, though the approach there is somewhat different than ours. They perform each computation with error-free transformations and interpret the errors as coefficients of new polynomials. They then evaluate the error polynomials, which (recursively) generate second order error polynomials and so on. This recursive property causes the number of operations to grow exponentially in K. Here, we instead have a fixed number of error groups, each corresponding to round-off from the group above it. For example, when $(1-s)b_j^{(n)} + sb_{j+1}^{(n)}$ is computed in floating point, any error is filtered down to the error group below it.

As in (5.1), the accuracy of the compensated result (5.2) may be arbitrarily bad for ill-conditioned polynomial evaluations. For example, as the condition number grows in Figure 5.2, some points have relative error exactly equal to 1; this indicates that `CompDeCasteljau`$(p, s) = 0$, which is a complete failure to evaluate the order of magnitude

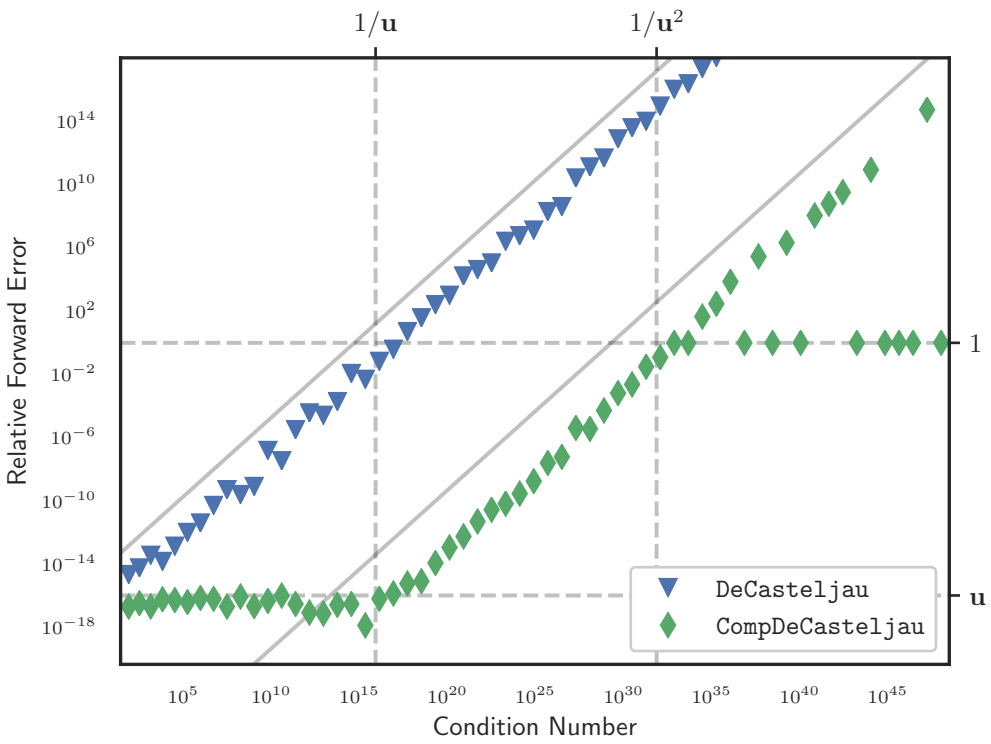

Figure 5.2: Evaluation of $p(s) = (s-1)(s-3/4)^7$ represented in Bernstein form.

of $p(s)$. For root-finding problems $\texttt{CompDeCasteljau}(p,s) = 0$ when $p(s) \neq 0$ can cause premature convergence and incorrect results. We describe how to defer rounding into progressively smaller error groups and improve the accuracy of the computed result by a factor of $\mathbf{u}$ for every error group added. So we derive `CompDeCasteljauK`, a K-fold compensated de Casteljau algorithm that satisfies the following a priori bound for any arbitrary integer K:

$$\frac{|p(s) - \texttt{CompDeCasteljauK}(p,s,K)|}{|p(s)|} \leq \mathbf{u} + \text{cond}\,(p,s) \times \mathcal{O}\left(\mathbf{u}^K\right). \tag{5.3}$$

This means that the computed value with `CompDeCasteljauK` is now as accurate as the result of the de Casteljau algorithm performed in K times the working precision with a final rounding back to the working precision.

The chapter is organized as follows. In Section 5.2, the compensated algorithm for polynomial evaluation from [JLCS10] is reviewed and notation is established for the expansion. In Section 5.3, the K-compensated algorithm is provided and a forward error analysis is performed. Finally, in Section 5.4 we perform two numerical experiments to give practical examples of the theoretical error bounds. (See Chapter 2 to review notation for error analysis with floating point operations, review results about error-free transformations or to review the de Casteljau algorithm.)

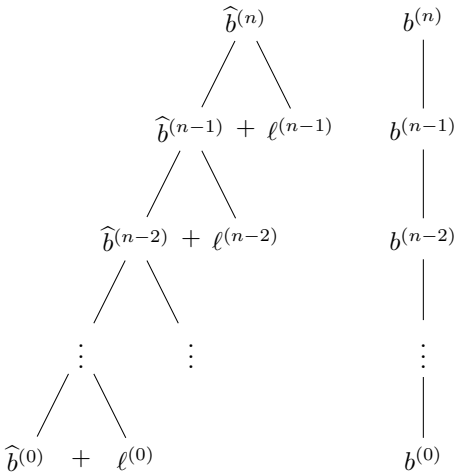

Figure 5.3: Local round-off errors

5.2 Compensated de Casteljau

In this section we review the compensated de Casteljau algorithm from [JLCS10]. In order to track the local errors at each update step, we use four EFTs:

$$[\widehat{r}, \rho] = \texttt{TwoSum}(1, -s) \tag{5.4}$$

$$[P_1, \pi_1] = \texttt{TwoProd}\left(\widehat{r}, \widehat{b}_j^{(k+1)}\right) \tag{5.5}$$

$$[P_2, \pi_2] = \texttt{TwoProd}\left(s, \widehat{b}_{j+1}^{(k+1)}\right) \tag{5.6}$$

$$\left[\widehat{b}_j^{(k)}, \sigma_3\right] = \texttt{TwoSum}(P_1, P_2) \tag{5.7}$$

With these, we can exactly describe the local error between the exact update and computed update:

$$\ell_{1,j}^{(k)} = \pi_1 + \pi_2 + \sigma_3 + \rho \cdot \widehat{b}_j^{(k+1)} \tag{5.8}$$

$$(1-s) \cdot \widehat{b}_j^{(k+1)} + s \cdot \widehat{b}_{j+1}^{(k+1)} = \widehat{b}_j^{(k)} + \ell_{1,j}^{(k)}. \tag{5.9}$$

By defining the global errors at each step

$$\partial b_j^{(k)} = b_j^{(k)} - \widehat{b}_j^{(k)} \tag{5.10}$$

we can see (Figure 5.3) that the local errors accumulate in $\partial b^{(k)}$:

$$\partial b_j^{(k)} = (1-s) \cdot \partial b_j^{(k+1)} + s \cdot \partial b_{j+1}^{(k+1)} + \ell_{1,j}^{(k)}. \tag{5.11}$$

When computed in exact arithmetic

$$p(s) = \widehat{b}_0^{(0)} + \partial b_0^{(0)} \tag{5.12}$$

and by using (5.11), we can continue to compute approximations of $\partial b_j^{(k)}$. The idea behind the compensated de Casteljau algorithm is to compute both the local error and the updates of the global error with floating point operations:

Algorithm 5.1 *Compensated de Casteljau algorithm for polynomial evaluation.*

function result = CompDeCasteljau(b, s)
  $n = \texttt{length}(b) - 1$
  $[\widehat{r}, \rho] = \texttt{TwoSum}(1, -s)$

  for $j = 0, \ldots, n$ **do**
    $\widehat{b}_j^{(n)} = b_j$
    $\widehat{\partial b}_j^{(n)} = 0$
  end for

  for $k = n-1, \ldots, 0$ **do**
    for $j = 0, \ldots, k$ **do**
      $[P_1, \pi_1] = \texttt{TwoProd}\left(\widehat{r}, \widehat{b}_j^{(k+1)}\right)$
      $[P_2, \pi_2] = \texttt{TwoProd}\left(s, \widehat{b}_{j+1}^{(k+1)}\right)$
      $\left[\widehat{b}_j^{(k)}, \sigma_3\right] = \texttt{TwoSum}(P_1, P_2)$
      $\widehat{\ell}_{1,j}^{(k)} = \pi_1 \oplus \pi_2 \oplus \sigma_3 \oplus \left(\rho \otimes \widehat{b}_j^{(k+1)}\right)$
      $\widehat{\partial b}_j^{(k)} = \widehat{\ell}_{1,j}^{(k)} \oplus \left(s \otimes \widehat{\partial b}_{j+1}^{(k+1)}\right) \oplus \left(\widehat{r} \otimes \widehat{\partial b}_j^{(k+1)}\right)$
    end for
  end for

  result $= \widehat{b}_0^{(0)} \oplus \widehat{\partial b}_0^{(0)}$
end function

When comparing this computed error to the exact error, the difference depends only on s and the Bernstein coefficients of p. Using a bound (Lemma 5.1) on the round-off error when computing $\partial b^{(0)}$, the algorithm can be shown to be as accurate as if the computations were done in twice the working precision:

Theorem 5.1 ([JLCS10], Theorem 5)**.** If no underflow occurs, $n \geq 2$ and $s \in [0, 1]$

$$\frac{|p(s) - \texttt{CompDeCasteljau}(p, s)|}{|p(s)|} \leq \mathbf{u} + 2\gamma_{3n}^2 \operatorname{cond}(p, s)\,. \tag{5.13}$$

Unfortunately, Figure 5.4 shows how CompDeCasteljau starts to break down in a region of high condition number (caused by a multiple root with multiplicity higher than two). For example, the point $s = \frac{1}{2} + 1001\mathbf{u}$ — which is in the plotted region $\left|s - \frac{1}{2}\right| \leq \frac{3}{2} \cdot 10^{-11}$ —

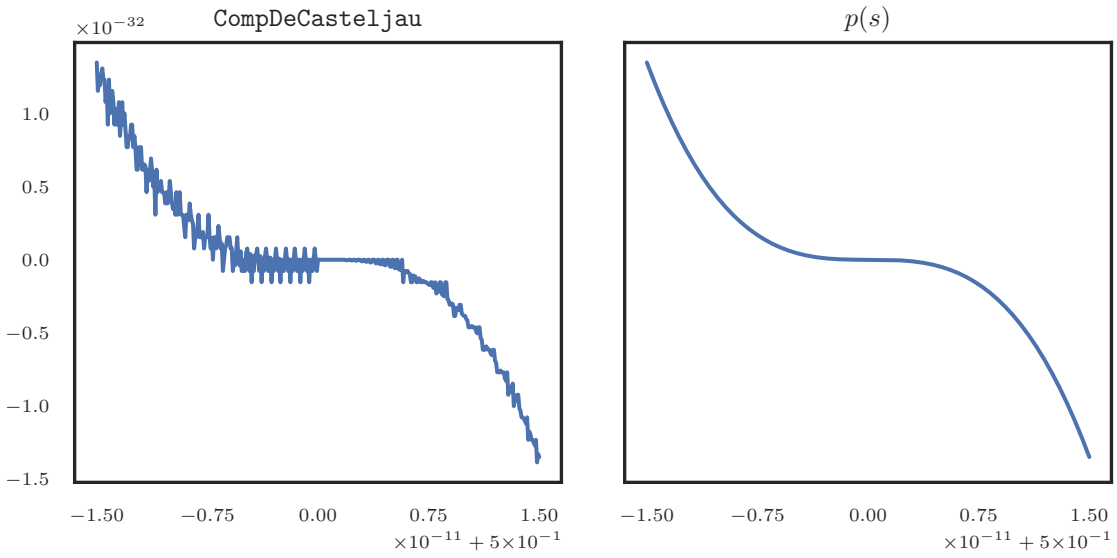

Figure 5.4: The compensated de Casteljau method starts to lose accuracy for $p(s) = (2s-1)^3(s-1)$ in the neighborhood of its multiple root $1/2$.

k	j	$\widehat{b}_j^{(k)}$	$\widehat{\partial b}_j^{(k)}$	$\partial b_j^{(k)} - \widehat{\partial b}_j^{(k)}$
3	0	$0.125 - 1.75(1001\mathbf{u}) - 0.25\mathbf{u}$	$0.25\mathbf{u}$	0
3	1	$-0.125 + 1.25(1001\mathbf{u}) + 0.25\mathbf{u}$	$-0.25\mathbf{u}$	0
3	2	$0.125 - 0.75(1001\mathbf{u})$	0	0
3	3	$-0.125 + 0.25(1001\mathbf{u})$	0	0
2	0	$-0.5(1001\mathbf{u})$	$3(1001\mathbf{u})^2$	0
2	1	$0.5(1001\mathbf{u}) + 0.125\mathbf{u}$	$-0.125\mathbf{u} - 2(1001\mathbf{u})^2$	0
2	2	$-0.5(1001\mathbf{u})$	$(1001\mathbf{u})^2$	0
1	0	$0.0625\mathbf{u} + (1001\mathbf{u})^2 + 239\mathbf{u}^2$	$-0.0625\mathbf{u} + 0.5(1001\mathbf{u})^2 - 239\mathbf{u}^2$	$-5(1001\mathbf{u})^3$
1	1	$0.0625\mathbf{u} - (1001\mathbf{u})^2 - 239\mathbf{u}^2$	$-0.0625\mathbf{u} - 0.5(1001\mathbf{u})^2 + 239\mathbf{u}^2$	$3(1001\mathbf{u})^3$
0	0	$0.0625\mathbf{u}$	$-0.0625\mathbf{u}$	$-4(1001\mathbf{u})^3 + 8(1001\mathbf{u})^4$

Table 5.1: Terms computed by `CompDeCasteljau` when evaluating $p(s) = (2s-1)^3(s-1)$ at the point $s = \frac{1}{2} + 1001\mathbf{u}$

evaluates to exactly 0 when it should be $\mathcal{O}(\mathbf{u}^3)$. As shown in Table 5.1, the breakdown occurs because $\widehat{b}_0^{(0)} = -\widehat{\partial b}_0^{(0)} = \mathbf{u}/16$.

5.3 K-Compensated de Casteljau

5.3.1 Algorithm Specified

In order to raise from twice the working precision to K times the working precision, we continue using EFTs when computing $\widehat{\partial b}^{(k)}$. By tracking the round-off from each floating point evaluation via an EFT, we can form a cascade of global errors:

$$b_j^{(k)} = \widehat{b}_j^{(k)} + \partial b_j^{(k)} \tag{5.14}$$

$$\partial b_j^{(k)} = \widehat{\partial b}_j^{(k)} + \partial^2 b_j^{(k)} \tag{5.15}$$

$$\partial^2 b_j^{(k)} = \widehat{\partial^2 b}_j^{(k)} + \partial^3 b_j^{(k)} \tag{5.16}$$

$$\vdots$$

In the same way local error can be tracked when updating $\widehat{b}_j^{(k)}$, it can be tracked for updates that happen down the cascade:

$$(1-s)\cdot \widehat{b}_j^{(k+1)} + s\cdot \widehat{b}_{j+1}^{(k+1)} = \widehat{b}_j^{(k)} + \ell_{1,j}^{(k)} \tag{5.17}$$

$$(1-s)\cdot \widehat{\partial b}_j^{(k+1)} + s\cdot \widehat{\partial b}_{j+1}^{(k+1)} + \ell_{1,j}^{(k)} = \widehat{\partial b}_j^{(k)} + \ell_{2,j}^{(k)} \tag{5.18}$$

$$(1-s)\cdot \widehat{\partial^2 b}_j^{(k+1)} + s\cdot \widehat{\partial^2 b}_{j+1}^{(k+1)} + \ell_{2,j}^{(k)} = \widehat{\partial^2 b}_j^{(k)} + \ell_{3,j}^{(k)} \tag{5.19}$$

$$\vdots$$

In `CompDeCasteljau` (Algorithm 5.1), after a single stage of error filtering we "give up" and use $\widehat{\partial b}$ instead of ∂b (without keeping around any information about the round-off error). In order to obtain results that are as accurate as if computed in K times the working precision, we must continue filtering (see Figure 5.5) errors down $(K-1)$ times, and only at the final level do we accept the rounded $\widehat{\partial^{K-1} b}$ in place of the exact $\partial^{K-1} b$.

When computing $\widehat{\partial^F b}$ (i.e. the error after F stages of filtering) there will be several sources of round-off. In particular, there will be

- errors when computing $\widehat{\ell}_{F,j}^{(k)}$ from the terms in $\ell_{F,j}^{(k)}$
- an error for the "missing" $\rho \cdot \widehat{\partial^F b}_j^{(k+1)}$ in $(1-s)\cdot \widehat{\partial^F b}_j^{(k+1)}$
- an error from the product $\widehat{r} \otimes \widehat{\partial^F b}_j^{(k+1)}$
- an error from the product $s \otimes \widehat{\partial^F b}_{j+1}^{(k+1)}$
- two errors from the two $\oplus$ when combining the three terms in $\widehat{\ell}_{F,j}^{(k)} \oplus \left(s \otimes \widehat{\partial^F b}_{j+1}^{(k+1)} \right) \oplus \left(\widehat{r} \otimes \widehat{\partial^F b}_j^{(k+1)} \right)$

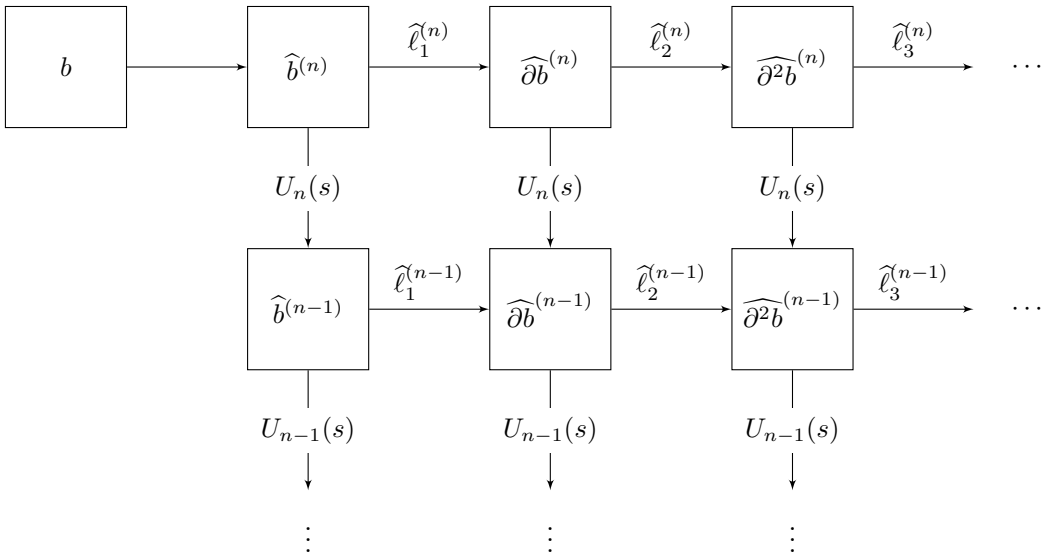

Figure 5.5: Filtering errors

For example, in (5.8):

$$\ell_{1,j}^{(k)} = \underbrace{\pi_1}_{P_1=\widehat{r}\otimes\widehat{b}_j^{(k+1)}} + \underbrace{\pi_2}_{P_2=s\otimes\widehat{b}_{j+1}^{(k+1)}} + \underbrace{\sigma_3}_{P_1\oplus P_2} + \underbrace{\rho\cdot\widehat{b}_j^{(k+1)}}_{(1-s)\widehat{b}_j^{(k+1)}} \tag{5.20}$$

After each stage, we'll always have

$$\ell_{F,j}^{(k)} = e_1 + \cdots + e_{5F-2} + \rho\cdot\widehat{\partial^{F-1}b}_j^{(k+1)} \tag{5.21}$$

where the terms $e_1,\ldots,e_{5F-2}$ come from using `TwoSum` and `TwoProd` when computing $\widehat{\partial^{F-1}b}_j^{(k)}$ and the ρ term comes from the round-off in $1\ominus s$ when multiplying $(1-s)$ by $\widehat{\partial^{F-1}b}_j^{(k+1)}$. With this in mind, we can define an EFT (`LocalErrorEFT`) that computes $\widehat{\ell}$ and tracks all round-off errors generated in the process:

Algorithm 5.2 *EFT for computing the local error.*

function $\left[\eta,\widehat{\ell}\right] = \texttt{LocalErrorEFT}(e,\rho,\delta b)$
$L = \texttt{length}(e)$

$\left[\widehat{\ell},\eta_1\right] = \texttt{TwoSum}(e_1,e_2)$
for $j = 3,\ldots,L$ **do**
$\left[\widehat{\ell},\eta_{j-1}\right] = \texttt{TwoSum}\left(\widehat{\ell},e_j\right)$

end for

$[P, \eta_L] = \texttt{TwoProd}\,(\rho, \delta b)$
$\left[\widehat{\ell}, \eta_{L+1}\right] = \texttt{TwoSum}\left(\widehat{\ell}, P\right)$
end function

With this EFT in place[1], we can perform $(K-1)$ error filtrations. Once we've computed the K stages of global errors, they can be combined with `SumK` (Algorithm A.6) to produce a sum that is as accurate as if computed in K times the working precision.

Algorithm 5.3 *K-compensated de Casteljau algorithm.*

function $\texttt{result} = \texttt{CompDeCasteljauK}(b, s, K)$
$n = \texttt{length}(b) - 1$
$[\widehat{r}, \rho] = \texttt{TwoSum}(1, -s)$

for $j = 0, \ldots, n$ **do**
$\widehat{b}_j^{(n)} = b_j$
for $F = 1, \ldots, K-1$ **do**
$\widehat{\partial^F b_j}^{(n)} = 0$
end for
end for

for $k = n-1, \ldots, 0$ **do**
for $j = 0, \ldots, k$ **do**
$[P_1, \pi_1] = \texttt{TwoProd}\left(\widehat{r}, \widehat{b}_j^{(k+1)}\right)$
$[P_2, \pi_2] = \texttt{TwoProd}\left(s, \widehat{b}_{j+1}^{(k+1)}\right)$
$\left[\widehat{b}_j^{(k)}, \sigma_3\right] = \texttt{TwoSum}(P_1, P_2)$

$e = [\pi_1, \pi_2, \sigma_3]$
$\delta b = \widehat{b}_j^{(k+1)}$

for $F = 1, \ldots, K-2$ **do**
$\left[\eta, \widehat{\ell}\right] = \texttt{LocalErrorEFT}(e, \rho, \delta b)$
$L = \texttt{length}(\eta)$

$[P_1, \eta_{L+1}] = \texttt{TwoProd}\left(s, \widehat{\partial^F b}_{j+1}^{(k+1)}\right)$
$[S_2, \eta_{L+2}] = \texttt{TwoSum}\left(\widehat{\ell}, P_1\right)$

[1] And the related `LocalError` in Algorithm A.7

$$[P_3, \eta_{L+3}] = \texttt{TwoProd}\left(\widehat{r}, \widehat{\partial^F b_j}^{(k+1)}\right)$$
$$\left[\widehat{\partial^F b_j}^{(k)}, \eta_{L+4}\right] = \texttt{TwoSum}\left(S_2, P_3\right)$$
$$e = \eta$$
$$\delta b = \widehat{\partial^F b_j}^{(k+1)}$$

end for

$$\widehat{\ell} = \texttt{LocalError}(e, \rho, \delta b)$$
$$\widehat{\partial^{K-1} b_j}^{(k)} = \widehat{\ell} \oplus \left(s \otimes \widehat{\partial^{K-1} b_{j+1}}^{(k+1)}\right) \oplus \left(\widehat{r} \otimes \widehat{\partial^{K-1} b_j}^{(k+1)}\right)$$

end for

end for

$$\texttt{result} = \texttt{SumK}\left(\left[\widehat{b}_0^{(0)}, \ldots, \widehat{\partial^{K-1} b_0}^{(0)}\right], K\right)$$

end function

Noting that $\ell_{F,j}$ contains $5F-1$ terms, one can show that `CompDeCasteljauK` (Algorithm 5.3) requires

$$(15K^2 + 11K - 34)T_n + 6K^2 - 11K + 11 = \mathcal{O}\left(n^2K^2\right) \tag{5.22}$$

flops to evaluate a degree n polynomial, where T_n is the nth triangular number. As a comparison, the non-compensated form of de Casteljau requires $3T_n + 1$ flops. In total this will require $(3K-4)T_n$ uses of `TwoProd`. On hardware that supports FMA, `TwoProdFMA` (Algorithm A.4) can be used instead, lowering the flop count by $15(3K-4)T_n$. Another way to lower the total flop count is to just use $\widehat{b}_0^{(0)} \oplus \cdots \oplus \widehat{\partial^{K-1} b_0}^{(0)}$ instead of `SumK`; this will reduce the total by $6(K-1)^2$ flops. When using a standard sum, the results produced are (empirically) identical to those with `SumK`. This makes sense: the whole point of `SumK` is to filter errors in a summation so that the final operation produces a sum of the form $v_1 \oplus \cdots \oplus v_K$ where each term is smaller than the previous by a factor of $\mathbf{u}$. This property is already satisfied for the $\widehat{\partial^F b_0}^{(0)}$ so in practice the K-compensated summation is likely not needed.

5.3.2 Error bound for polynomial evaluation

Theorem 5.1 ([ORO05], Proposition 4.10)**.** A summation can be computed (`SumK`, Algorithm A.6) with results that are as accurate as if computed in K times the working precision. When computed this way, the result satisfies:

$$\left|\texttt{SumK}(v, K) - \sum_{j=1}^{n} v_j\right| \leq \left(\mathbf{u} + 3\gamma_{n-1}^2\right)\left|\sum_{j=1}^{n} v_j\right| + \gamma_{2n-2}^K \sum_{j=1}^{n} |v_j|\,. \tag{5.23}$$

Lemma 5.1 ([JLCS10], Theorem 4)**.** The second order error $\partial^2 b_0^{(0)}$ satisfies[2]

$$\left|\partial b_0^{(0)} - \widehat{\partial b}_0^{(0)}\right| = \left|\partial^2 b_0^{(0)}\right| \leq 2\gamma_{3n+2}\gamma_{3(n-1)}\widetilde{p}(s). \tag{5.24}$$

To enable a bound on the K order error $\partial^K b_0^{(0)}$, it's necessary to understand the difference between the exact local errors $\ell_{F,j}$ and the computed equivalents $\widehat{\ell}_{F,j}$. To do this, we define

$$\widetilde{\ell}_{F,j} \coloneqq |e_1| + \cdots + |e_{5F-2}| + \left|\rho \cdot \widehat{\partial^{F-1} b_j}^{(k+1)}\right|. \tag{5.25}$$

Lemma 5.2. The local error bounds $\widetilde{\ell}_{F,j}$ satisfy:

$$\widetilde{\ell}_{1,j}^{(k)} \leq \gamma_3 \left((1-s)\left|\widehat{b}_j^{(k+1)}\right| + s\left|\widehat{b}_{j+1}^{(k+1)}\right|\right) \tag{5.26}$$

$$\widetilde{\ell}_{F+1,j}^{(k)} \leq \gamma_3 \left((1-s)\left|\widehat{\partial^F b_j}^{(k+1)}\right| + s\left|\widehat{\partial^F b_{j+1}}^{(k+1)}\right|\right) + \gamma_{5F} \cdot \widetilde{\ell}_{F,j}^{(k)} \text{ for } F \geq 1. \tag{5.27}$$

Proof. See proof in Appendix B. ■

As we'll see soon (Lemma 5.4), putting a bound on sums of the form $\sum_{j=0}^{k} \ell_{F,j}^{(k)} B_{j,k}(s)$ will be useful to get an overall bound on the relative error for `CompDeCasteljauK`, so we define $L_{F,k} \coloneqq \sum_{j=0}^{k} \ell_{F,j}^{(k)} B_{j,k}(s)$.

Lemma 5.3. For $s \in [0,1]$, the Bernstein-type error sum defined above satisfies the following bounds:

$$L_{F,n-k} \leq \left[\left(3^F \binom{k}{F-1} + \mathcal{O}\left(k^{F-1}\right)\right)\mathbf{u}^F + \mathcal{O}\left(\mathbf{u}^{F+1}\right)\right] \cdot \widetilde{p}(s) \tag{5.28}$$

$$\sum_{k=0}^{n-1} \gamma_{3k+5F} L_{F,k} \leq \left[\left(3^{F+1}\binom{n}{F+1} + \mathcal{O}\left(n^F\right)\right)\mathbf{u}^{F+1} + \mathcal{O}\left(\mathbf{u}^{F+2}\right)\right] \cdot \widetilde{p}(s). \tag{5.29}$$

In particular, this means that $\sum_{k=0}^{n-1} \gamma_{3k+5F} L_{F,k} = \mathcal{O}\left((3n\mathbf{u})^{F+1}\right) \cdot \widetilde{p}(s)$.

Proof. See proof in Appendix B. ■

Lemma 5.4. The K order error $\partial^K b_0^{(0)}$ satisfies

$$\left|\partial^{K-1} b_0^{(0)} - \widehat{\partial^{K-1} b_0}^{(0)}\right| = \left|\partial^K b_0^{(0)}\right| \leq \left[\left(3^K \binom{n}{K} + \mathcal{O}\left(n^{K-1}\right)\right)\mathbf{u}^K + \mathcal{O}\left(\mathbf{u}^{K+1}\right)\right] \cdot \widetilde{p}(s). \tag{5.30}$$

Proof. See proof in Appendix B. ■

[2] The authors missed one round-off error so used γ_{3n+1} where γ_{3n+2} would have followed from their arguments.

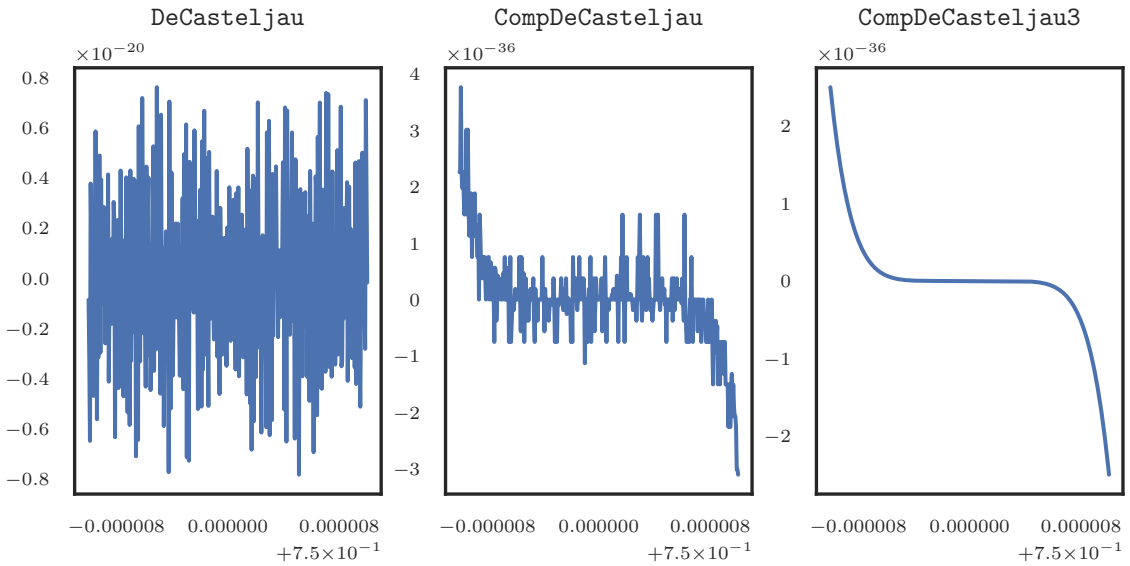

Figure 5.6: Evaluation of $p(s) = (s-1)(s-3/4)^7$ in the neighborhood of its multiple root $3/4$.

Theorem 5.2. If no underflow occurs, $n \geq 2$ and $s \in [0,1]$

$$\frac{|p(s) - \texttt{CompDeCasteljau}(p,s,K)|}{|p(s)|} \leq \left[\mathbf{u} + \mathcal{O}\left(\mathbf{u}^2\right)\right] + \left[\left(3^K \binom{n}{K} + \mathcal{O}\left(n^{K-1}\right)\right)\mathbf{u}^K + \mathcal{O}\left(\mathbf{u}^{K+1}\right)\right] \operatorname{cond}(p,s). \quad (5.31)$$

Proof. See proof in Appendix B. ■

For the first few values of K the coefficient of $\operatorname{cond}(p,s)$ in the bound is

K	Method	Multiplier
1	`DeCasteljau`	$3\binom{n}{1}\mathbf{u} = 3n\mathbf{u} \approx \gamma_{3n}$
2	`CompDeCasteljau`	$\left[9\binom{n}{2} + 15\binom{n}{1}\right]\mathbf{u}^2 = \frac{3n(3n+7)}{2}\mathbf{u}^2 \approx \frac{1}{4}\cdot 2\gamma_{3n}^2$
3	`CompDeCasteljau3`	$\left[27\binom{n}{3} + 135\binom{n}{2} + 150\binom{n}{1}\right]\mathbf{u}^3 = \frac{3n(3n^2+36n+61)}{2}\mathbf{u}^3$
4	`CompDeCasteljau4`	$\left[81\binom{n}{4} + 810\binom{n}{3} + 2475\binom{n}{2} + 2250\binom{n}{1}\right]\mathbf{u}^4$

See the proof of Lemma 5.3 for more details on where these polynomials come from.

5.4 Numerical experiments

All experiments were performed in IEEE-754 double precision. As in [JLCS10], we consider the evaluation in the neighborhood of the multiple root of $p(s) = (s-1)(s-3/4)^7$,

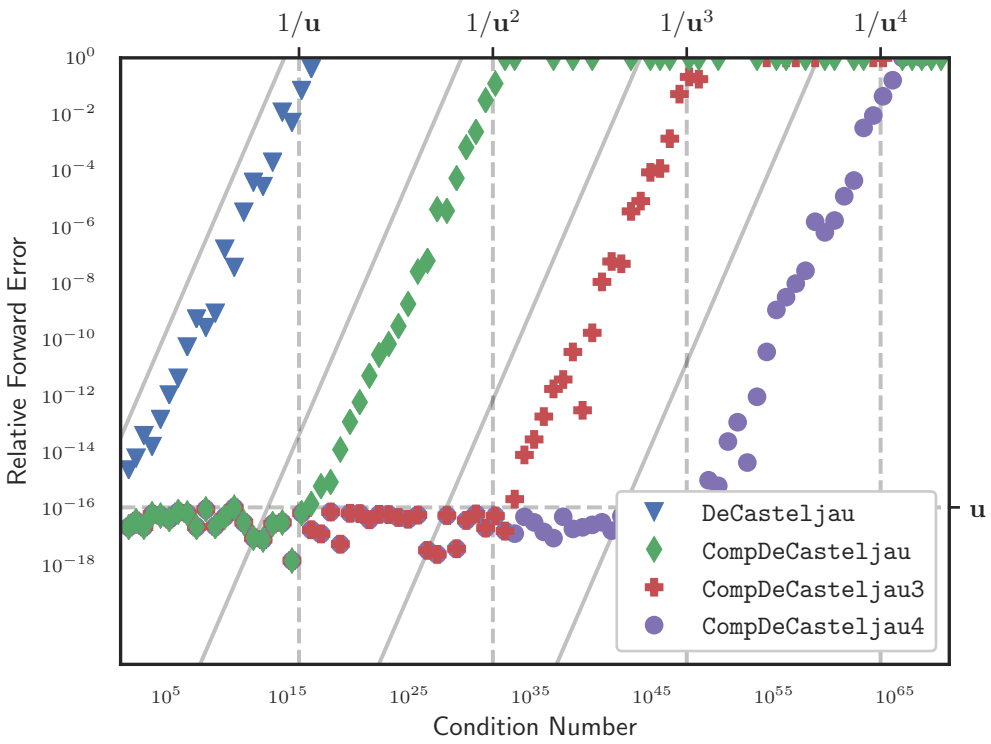

Figure 5.7: Accuracy of evaluation of $p(s) = (s-1)(s-3/4)^7$ represented in Bernstein form.

written in Bernstein form. Figure 5.6 shows the evaluation of $p(s)$ at the 401 equally spaced[3] points $\left\{\frac{3}{4} + j\frac{10^{-7}}{2}\right\}_{j=-200}^{200}$ with `DeCasteljau` (Algorithm 2.1), `CompDeCasteljau` (Algorithm 5.1) and `CompDeCasteljau3` (Algorithm 5.3 with $K = 3$). We see that `DeCasteljau` fails to get the magnitude correct, `CompDeCasteljau` has the right shape but lots of noise and `CompDeCasteljau3` is able to smoothly evaluate the function. This is in contrast to a similar figure in [JLCS10], where the plot was smooth for the 400 equally spaced points $\left\{\frac{3}{4} + \frac{10^{-4}}{2}\frac{2j-399}{399}\right\}_{j=0}^{399}$. The primary difference is that as the interval shrinks by a factor of $\approx \frac{10^{-4}}{10^{-7}} = 10^3$, the condition number goes up by $\approx 10^{21}$ and `CompDeCasteljau` is no longer accurate.

Figure 5.7 shows the relative forward errors compared against the condition number. To compute relative errors, each input and coefficient is converted to a fraction (i.e. infinite precision) and $p(s)$ is computed exactly as a fraction, then compared to the corresponding computed values. Similar tools are used to **exactly** compute the condition number,

[3]It's worth noting that 0.1 cannot be represented exactly in IEEE-754 double precision (or any binary arithmetic for that matter). Hence (most of) the points of the form $a + b \cdot 10^{-c}$ can only be approximately represented.

though here we can rely on the fact that $\widetilde{p}(s) = (s-1)(s/2-3/4)^7$. Once the relative errors and condition numbers are computed as fractions, they are rounded to the nearest IEEE-754 double precision value. As in [JLCS10], we use values $\left\{\frac{3}{4} - (1.3)^j\right\}_{j=-5}^{-90}$ [4]. The curves for `DeCasteljau` and `CompDeCasteljau` trace the same paths seen in [JLCS10]. In particular, `CompDeCasteljau` has a relative error that is $\mathcal{O}(\mathbf{u})$ until $\operatorname{cond}(p, s)$ reaches $1/\mathbf{u}$, at which point the relative error increases linearly with the condition number until it becomes $\mathcal{O}(1)$ when $\operatorname{cond}(p, s)$ reaches $1/\mathbf{u}^2$. Similarly, the relative error in `CompDeCasteljau3` (Algorithm 5.3 with $K = 3$) is $\mathcal{O}(\mathbf{u})$ until $\operatorname{cond}(p, s)$ reaches $1/\mathbf{u}^2$ at which point the relative error increases linearly to $\mathcal{O}(1)$ when $\operatorname{cond}(p, s)$ reaches $1/\mathbf{u}^3$ and the relative error in `CompDeCasteljau4` (Algorithm 5.3 with $K = 4$) is $\mathcal{O}(\mathbf{u})$ until $\operatorname{cond}(p, s)$ reaches $1/\mathbf{u}^3$ at which point the relative error increases linearly to $\mathcal{O}(1)$ when $\operatorname{cond}(p, s)$ reaches $1/\mathbf{u}^4$.

[4] As with 0.1, it's worth noting that $(1.3)^j$ can't be represented exactly in IEEE-754 double precision. However, this geometric series still serves a useful purpose since it continues to raise $\operatorname{cond}(p, s)$ as j decreases away from 0 and because it results in "random" changes in the bits of 0.75 that are impacted by subtracting $(1.3)^j$.

Chapter 6

Accurate Newton's Method for Bézier Curve Intersection

6.1 Introduction

When using Newton's method to find the root of a function via

$$G(\boldsymbol{x}) = \boldsymbol{x} - J^{-1}F(\boldsymbol{x}) \tag{6.1}$$

there are three computations performed that can introduce instability: evaluation of the residual function $F(\boldsymbol{x})$, evaluation of the Jacobian J and solution of the linear system $J\boldsymbol{y} = F(\boldsymbol{x})$. In [Tis01], the author showed that by just using a more precise evaluation of the residual function, the accuracy of Newton's method can be improved.

This chapter considers Newton's method applied to two problems: root-finding for polynomials expressed in Bernstein form and intersection of two Bézier curves in $\mathbf{R}^2$. In both problems, the compensated de Casteljau method (see Chapter 5) is used for evaluation of the residual. When evaluating a polynomial $p(s)$ this is straightforward, but when evaluating the difference $b_1(s) - b_2(t)$ between two curves special care must be taken.

In [Gra08], the problem of finding simple roots α of polynomials $p(s)$ expressed in the monomial basis is considered. A standard Newton's method (`HNewtonBasic`) that evaluates $p(s)$ and $p'(s)$ using Horner's method is compared to a modified Newton's method (`HNewtonAccurate`) that evaluates $p(s)$ with a compensated Horner's method. This proceeds as in [Tis01]: the evaluation of the residual is done with greater accuracy but the rest of the process is the same. When computing a root α, the standard Newton's method has a relative error that grows linearly with the condition number of the root (which will be defined in Section 6.2). The modified Newton's method is fully accurate to machine precision (i.e. the relative error is $\mathcal{O}(\mathbf{u})$) until $\operatorname{cond}(\alpha)$ reaches $1/\mathbf{u}$, as seen in Figure 6.1.

After the point where `HNewtonAccurate` loses accuracy, we'd expect a linear increase in relative error based on a compensated rule of thumb:

$$\frac{|\widehat{\alpha} - \alpha|}{|\alpha|} \leq c_1\mathbf{u} + c_2 \operatorname{cond}(\alpha)\,\mathbf{u}^2 \tag{6.2}$$

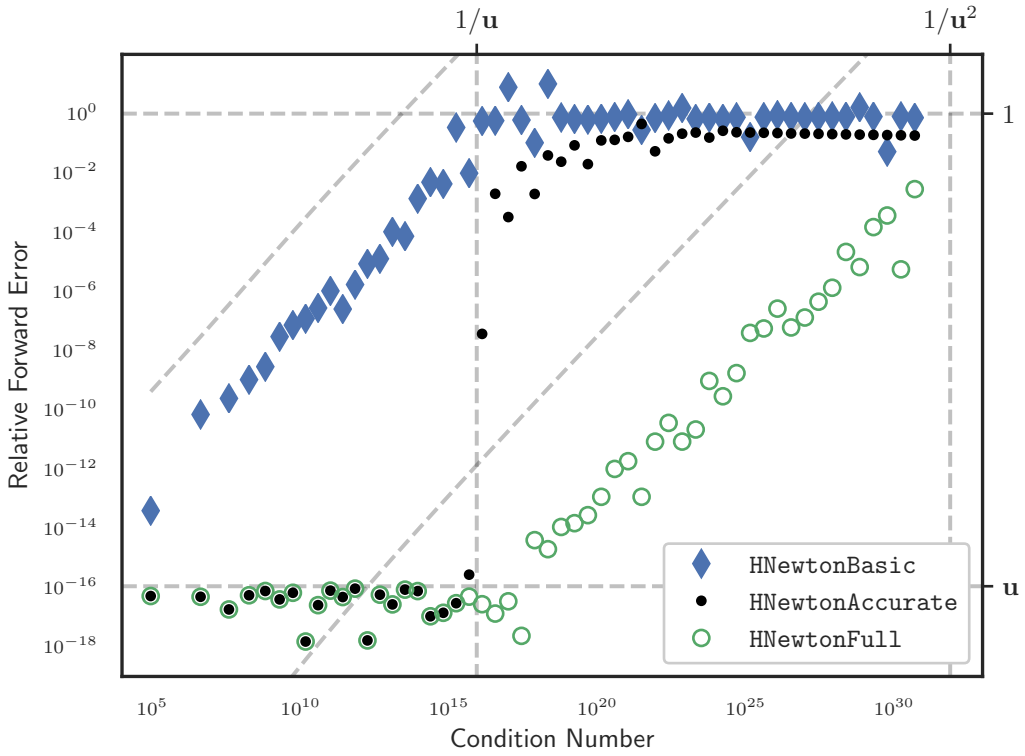

Figure 6.1: Comparing relative error to condition number when using Newton's method to find a root of $p(s) = (s-1)^n - 2^{-31}$, where polynomial evaluation occurs via Horner's method.

where $c_1\mathbf{u}$ corresponds to rounding into the given precision and $c_2 \operatorname{cond}(\alpha)\,\mathbf{u}^2$ reflects the typical error but from computations done with working precision $\overline{\mathbf{u}} = \mathbf{u}^2$. The point where the condition number exceeds $1/\mathbf{u}$ should correspond to the point where the second term is larger that the first term. However, this is not possible unless the Jacobian (i.e. $p'(s)$) is also evaluated with a compensated method. A second modified Newton's method (`HNewtonFull`) is introduced in [JGH+13, Section 8] and the author shows that this second modified Newton's method does indeed follow a compensated rule of thumb under appropriate conditions. We see in Figure 6.1 that `HNewtonFull` enables $\mathcal{O}(\mathbf{u})$ relative errors until the condition number reaches $1/\mathbf{u}$ and then a linear increase in error as the condition number grows from $1/\mathbf{u}$ to $1/\mathbf{u}^2$.

We'll proceed similarly for polynomials in Bernstein form. Since this is a one dimensional Newton's method, improving the evaluation of the Jacobian is straightforward. The results agree with what has been observed when using Horner's method for evaluation.

Computing the intersection(s) of two parametric plane curves is a common task in computational geometry and has many uses, e.g. in finite element methods that use overlapping curved meshes. Many methods have been described in the literature to solve this problem.

Algebraic methods such as implicitization and eigenvalue-based methods (e.g. [MD92]) suffer from accuracy issues for moderately high degrees and can often be very complex to implement. Some ([BHSW08]) even rely on symbolic algebraic manipulations, which can be quite costly since it requires arbitrary precision. Geometric methods (e.g. [SP86, SN90, KLS98]) typically use a form of domain splitting to focus on subproblems and eliminate parts of the domain where an intersection is guaranteed not to occur. After a domain has been sufficiently reduced, Newton's method is used for the last few bits of accuracy.

We'll focus on transversal curve intersections that are ill-conditioned. Transversal intersections are an extension of the concept of a simple root. A non-transversal intersection occurs when the curves or tangent at the point of intersection or when one of the curves has a zero tangent vector at that point, either due to an improper parameterization (e.g. $x(s) = s^2, y(s) = s^2 + 1$) or a cusp. In many cases, transversal intersections that are "almost tangent" have very high condition numbers.

The chapter is organized as follows. In Section 6.2 we define and discuss the conditioning of both a simple root and a transversal intersection. In Section 6.3 we describe two compensated Newton's methods for finding simple roots and perform a numerical experiment verifying the expected behavior. In Section 6.4 we describe a compensated Newton's method for Bézier curve intersection and perform sever numerical experiements to verify the expected behavior on both transversal intersections and tangent intersections (i.e. intersections with infinite condition number). Section 6.5 acts as a coda: it describes some failed attempts at constructing numerical examples. This section provides an in-depth discussion of a particular family of polynomials that has much better than expected conditioning when written in the Bernstein basis.

6.2 Problem conditioning

Consider a smooth function $F : \mathbf{R}^n \longrightarrow \mathbf{R}^n$ with Jacobian $F_{\boldsymbol{x}} = J$. We want to consider a special class of functions of the form $F(\boldsymbol{x}) = \sum_j c_j \phi_j(\boldsymbol{x})$ where the basis functions ϕ_j are also smooth functions on $\mathbf{R}^n$ and each $c_j \in \mathbf{R}$. We want to consider the effects on a root $\boldsymbol{\alpha} \in \mathbf{R}^n$ of a perturbation in one of the coefficients c_j. We examine the perturbed functions

$$G(x, \delta) = F(\boldsymbol{x}) + \delta \phi_j(\boldsymbol{x}). \tag{6.3}$$

Since $G(\boldsymbol{\alpha}, 0) = \mathbf{0}$, if J^{-1} exists at $\boldsymbol{x} = \boldsymbol{\alpha}$ then the implicit function theorem tells us that we can define $\boldsymbol{x}$ via

$$G(\boldsymbol{x}(\delta), \delta) = \mathbf{0}. \tag{6.4}$$

Taking the derivative with respect to δ we find that $\mathbf{0} = G_{\boldsymbol{x}} \boldsymbol{x}_\delta + G_\delta$. Plugging in $\delta = 0$ we find that $0 = J(\boldsymbol{\alpha}) \boldsymbol{x}_\delta + \phi_j(\boldsymbol{\alpha})$, hence we conclude that

$$\boldsymbol{x}(\delta) = \boldsymbol{\alpha} - J(\boldsymbol{\alpha})^{-1} \phi_j(\boldsymbol{\alpha}) \delta + \mathcal{O}(\delta^2). \tag{6.5}$$

This gives a relative condition number (for the root) of

$$\frac{\left\| J\left(\boldsymbol{\alpha}\right)^{-1} \phi_j\left(\boldsymbol{\alpha}\right)\right\|}{\|\boldsymbol{\alpha}\|}. \tag{6.6}$$

By considering perturbations in **all** of the coefficients: $|\delta_j| \leq \varepsilon |c_j|$, a similar analysis gives a root function

$$\boldsymbol{x}\left(\delta_0, \ldots, \delta_n\right) = \boldsymbol{\alpha} - J\left(\boldsymbol{\alpha}\right)^{-1} \sum_{j=0}^{n} \delta_j \phi_j\left(\boldsymbol{\alpha}\right) + \mathcal{O}\left(\varepsilon^2\right). \tag{6.7}$$

With this, we can define a root condition number

$$\kappa_{\boldsymbol{\alpha}} = \lim_{\varepsilon \to 0} \left(\sup \frac{\|\delta\boldsymbol{\alpha}\| / \varepsilon}{\|\boldsymbol{\alpha}\|} \right) = \lim_{\varepsilon \to 0} \left(\sup \frac{\left\| J\left(\boldsymbol{\alpha}\right)^{-1} \sum_j \delta_j \phi_j\left(\boldsymbol{\alpha}\right)\right\| / \varepsilon}{\|\boldsymbol{\alpha}\|} \right). \tag{6.8}$$

When $n = 1$, J^{-1} is simply $1/F'$ and we find

$$\kappa_\alpha = \frac{1}{|\alpha F'(\alpha)|} \sum_{j=0}^{n} \left| c_j \phi_j(\alpha) \right|. \tag{6.9}$$

This value is given by the triangle inequality applied to $\delta\alpha$ and equality can be attained since the sign of each $\delta_j = \pm c_j \varepsilon$ can be modified at will to make $\phi_j(\alpha)\delta_j = \left|\phi_j(\alpha) c_j\right| \varepsilon$.

When $n > 1$, the triangle inequality tells us that

$$\kappa_{\boldsymbol{\alpha}} = \lim_{\varepsilon \to 0} \left(\sup \frac{\|\delta\boldsymbol{\alpha}/\varepsilon\|}{\|\boldsymbol{\alpha}\|} \right) \leq \frac{1}{\|\boldsymbol{\alpha}\|} \sum_{j=0}^{n} |c_j| \left\| J\left(\boldsymbol{\alpha}\right)^{-1} \phi_j(\boldsymbol{\alpha}) \right\|. \tag{6.10}$$

However, this bound is only attainable if all $\phi_j(\boldsymbol{\alpha})$ are parallel. However, we'll seldom need to compute the exact condition number and are instead typically interested in the order of magnitude. In this case a lower bound

$$\frac{1}{\|\boldsymbol{\alpha}\|} \max_j |c_j| \left\| J\left(\boldsymbol{\alpha}\right)^{-1} \phi_j(\boldsymbol{\alpha}) \right\| \tag{6.11}$$

for $\kappa_{\boldsymbol{\alpha}}$ will suffice as an approximate condition number.

For an example, consider

$$\phi_0 = \begin{bmatrix} x_0 \\ 2 \\ 0 \end{bmatrix}, \phi_1 = \begin{bmatrix} 0 \\ x_1 \\ 3 \end{bmatrix}, \phi_2 = \begin{bmatrix} 2 \\ 0 \\ x_2 \end{bmatrix}, F = \phi_0 + 2\phi_1 + 3\phi_2, \boldsymbol{\alpha} = \begin{bmatrix} -6 \\ -1 \\ -2 \end{bmatrix}. \tag{6.12}$$

For a given ε, the maximum root perturbation occurs when $\delta_0 = \varepsilon, \delta_1 = 2\varepsilon, \delta_2 = -3\varepsilon$ and gives $\left\| J\left(\boldsymbol{\alpha}\right)^{-1} \sum_j \delta_j \phi_j\left(\boldsymbol{\alpha}\right)\right\| = 4\sqrt{10}\varepsilon \approx 12.65\varepsilon$. The pessimistic triangle inequality

bound gives $\sum_j |c_j| \left\| J(\boldsymbol{\alpha})^{-1} \phi_j(\boldsymbol{\alpha}) \right\| \approx 14.64\varepsilon$ and the maximum individual perturbation is $2\sqrt{10}\varepsilon \approx 6.325\varepsilon$ (this occurs when $\delta_0 = \delta_1 = 0, \delta_2 = \pm 3\varepsilon$).

In this general framework, we can define a condition number both for a simple root of a polynomial in Bernstein form and for the intersection of two planar Bézier curves. For the first, $\phi_j(s) = \binom{n}{j}(1-s)^{n-j}s^j$ the Bernstein basis functions, a polynomial $p(s) = \sum_j b_j \phi_j(s)$ with a simple root $\alpha \in (0, 1]$ has root condition number

$$\kappa_\alpha = \frac{1}{\alpha \left| p'(\alpha) \right|} \sum_{j=0}^{n} \left| b_j \phi_j(\alpha) \right| = \frac{\widetilde{p}(\alpha)}{\alpha \left| p'(\alpha) \right|}. \tag{6.13}$$

For the intersection of a degree m curve $b_1(s)$ and a degree n curve $b_2(t)$, we have basis functions

$$\begin{aligned}
\phi_{0,-1,1} = \begin{bmatrix} B_{0,m}(s) \\ 0 \end{bmatrix}, \phi_{0,-1,2} = \begin{bmatrix} 0 \\ B_{0,m}(s) \end{bmatrix}, \cdots, \\
\phi_{m,-1,1} = \begin{bmatrix} B_{m,m}(s) \\ 0 \end{bmatrix}, \phi_{m,-1,2} = \begin{bmatrix} 0 \\ B_{m,m}(s) \end{bmatrix}, \\
\phi_{-1,0,1} = \begin{bmatrix} -B_{0,n}(t) \\ 0 \end{bmatrix}, \phi_{-1,0,2} = \begin{bmatrix} 0 \\ -B_{0,n}(t) \end{bmatrix}, \cdots, \\
\phi_{-1,n,1} = \begin{bmatrix} -B_{n,n}(t) \\ 0 \end{bmatrix}, \phi_{-1,n,2} = \begin{bmatrix} 0 \\ -B_{n,n}(t) \end{bmatrix}.
\end{aligned} \tag{6.14}$$

Since $F(s,t) = b_1(s) - b_2(t)$ we have Jacobian $J(s,t) = \begin{bmatrix} b_1'(s) & -b_2'(t) \end{bmatrix}$. We'll consider a transversal intersection $F(\alpha, \beta) = \mathbf{0}$ with $\det J(\alpha, \beta) \neq 0$. Since each of the ϕ_j is just a scalar multiple of the standard basis vectors, writing $J^{-1} = \begin{bmatrix} \boldsymbol{v}_1 & \boldsymbol{v}_2 \end{bmatrix}$, we have

$$\begin{aligned}
J(\alpha,\beta)^{-1} \sum_{\boldsymbol{j}} \delta_{\boldsymbol{j}} \phi_{\boldsymbol{j}}(\alpha,\beta) = \left[\sum_{i=0}^{m} \delta_{i,-1,1} B_{i,m}(\alpha) + \sum_{j=0}^{n} \delta_{-1,j,1} B_{j,n}(\beta) \right] \boldsymbol{v}_1 \\
+ \left[\sum_{i=0}^{m} \delta_{i,-1,2} B_{i,m}(\alpha) + \sum_{j=0}^{n} \delta_{-1,j,2} B_{j,n}(\beta) \right] \boldsymbol{v}_2 = \nu_1 \boldsymbol{v}_1 + \nu_2 \boldsymbol{v}_2.
\end{aligned} \tag{6.15}$$

where

$$|\nu_k| / \varepsilon \leq \sum_{i=0}^{m} |c_{i,-1,k}| B_{i,m}(\alpha) + \sum_{j=0}^{n} |c_{-1,j,k}| B_{j,n}(\beta) = \mu_k \tag{6.16}$$

and the bound can be attained for both $k = 1, 2$ by making the signs of the $\delta_{\boldsymbol{j}}$ agree. If we name the components of each curve via $b_1(s) = \begin{bmatrix} x_1(s) & y_1(s) \end{bmatrix}^T$ and $b_2(t) = \begin{bmatrix} x_2(t) & y_2(t) \end{bmatrix}^T$ then we see that $\mu_1 = \widetilde{x}_1(\alpha) + \widetilde{x}_2(\beta)$ and $\mu_2 = \widetilde{y}_1(\alpha) + \widetilde{y}_2(\beta)$. Thus we have condition number

$$\kappa_{\alpha,\beta} = \frac{1}{\sqrt{\alpha^2 + \beta^2}} \max_{|\nu_k| \leq \mu_k} \left\| \nu_1 \boldsymbol{v}_1 + \nu_2 \boldsymbol{v}_2 \right\|_2 \tag{6.17}$$

$$= \sqrt{\frac{\max_{|\nu_k| \leq \mu_k} \nu_1^2 (\boldsymbol{v}_1 \cdot \boldsymbol{v}_1) + 2\nu_1\nu_2 (\boldsymbol{v}_1 \cdot \boldsymbol{v}_2) + \nu_2^2 (\boldsymbol{v}_2 \cdot \boldsymbol{v}_2)}{\alpha^2 + \beta^2}}. \tag{6.18}$$

Since J^{-1} is invertible, we know $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ are not parallel which can be used to show that the only internal critical point of the function to be maximimized in (6.18) is $\nu_1 = \nu_2 = 0$, which is the global minimum. Along the boundary of the rectangle $[-\mu_1, \mu_1] \times [-\mu_2, \mu_2]$, we fix one of ν_1 or ν_2 and the resulting univariate function is an up-opening parabola, hence any critical point must be a local minimum. Thus we know the maximum occurs at two of the four corners of the rectangle:

$$\kappa_{\alpha,\beta} = \sqrt{\frac{\mu_1^2 (\boldsymbol{v}_1 \cdot \boldsymbol{v}_1) + 2\mu_1\mu_2 |\boldsymbol{v}_1 \cdot \boldsymbol{v}_2| + \mu_2^2 (\boldsymbol{v}_2 \cdot \boldsymbol{v}_2)}{\alpha^2 + \beta^2}}. \tag{6.19}$$

As far as the author can tell, a condition number for the intersection of two planar Bézier curves has not been described in the Computer Aided Geometric Design (CAGD) literature. In [Hig02, Chapter 25, Equation 25.11] a more generic condition number is defined for the root of a nonlinear algebraic system that is similar to the definition above.

For an example, consider the line $b_1(s) = \begin{bmatrix} 2s & 2s \end{bmatrix}^T$ and improperly parameterized line $b_2(t) = \begin{bmatrix} 4t^2 & 2 - 4t^2 \end{bmatrix}^T$ which intersect at $\alpha = \beta = 1/2$. At the intersection we have $J^{-1} = \frac{1}{8}\begin{bmatrix} 2 & 2 \\ -1 & 1 \end{bmatrix}$, so that $\boldsymbol{v}_1 \cdot \boldsymbol{v}_1 = \boldsymbol{v}_2 \cdot \boldsymbol{v}_2 = 5/64$ and $\boldsymbol{v}_1 \cdot \boldsymbol{v}_2 = 3/64$. Since the x-component of $F(s,t)$ can be written as $2s - 4t^2 = 2B_{1,1}(s) - 4B_{2,2}(t)$ and the y-component as $2s + 4t^2 - 2 = 2B_{1,1}(s) - 2B_{0,2}(t) - 2B_{1,2}(t) + 2B_{2,2}(t)$ we have

$$\mu_1 = 2B_{1,1}(\alpha) \qquad + 4B_{2,2}(\beta) = 2 \tag{6.20}$$

$$\mu_2 = 2B_{1,1}(\alpha) + 2B_{0,2}(\beta) + 2B_{1,2}(\beta) + 2B_{2,2}(\beta) = 3. \tag{6.21}$$

Following (6.19), this gives $\kappa_{\alpha,\beta} = \sqrt{202}/8 \approx 1.78$.

6.3 Simple polynomial roots

Similar to [Gra08] we define an algorithm to perform standard Newton's method (`DNewtonBasic`) for polynomials in Bernstein form as well as a modified Newton's (`DNewtonAccurate`) that uses compensated de Casteljau (Algorithm 5.1) to evaluate the residual. Via the `NewtonGeneric` helper (Algorithm A.8), we need only define the callable that computes $p(s)/p'(s)$. (Such callables will have a single input s but will also be parameterized by the coefficients of p. This is often referred to as a closure in programming languages.)

Algorithm 6.1 *Newton's method for polynomial in Bernstein form.*

function $s_* =$ `DNewtonBasic`$(b, s_0,$ `tol`, `max_iter`$)$

$n = \texttt{length}(b) - 1$
for $j = 0, \ldots, n - 1$ **do**
$\Delta b_j = b_{j+1} - b_j$
end for

function `update` $=$ `update_fn`(s)
$\widehat{b} = \texttt{DeCasteljau}(b, s)$
$\widehat{b'} = n \otimes \texttt{DeCasteljau}(\Delta b, s)$
`update` $= \widehat{b} \oslash \widehat{b'}$
end function

$s_* = \texttt{NewtonGeneric}\,(\texttt{update\_fn}, s_0, \texttt{tol}, \texttt{max\_iter})$
end function

Algorithm 6.2 *Modified Newton's method for polynomial in Bernstein form.*

function $s_* = \texttt{DNewtonAccurate}(b, s_0, \texttt{tol}, \texttt{max\_iter})$
$n = \texttt{length}(b) - 1$
for $j = 0, \ldots, n - 1$ **do**
$\Delta b_j = b_{j+1} - b_j$
end for

function `update` $=$ `update_fn`(s)
$\widehat{b} = \texttt{CompDeCasteljau}(b, s)$
$\widehat{b'} = n \otimes \texttt{DeCasteljau}(\Delta b, s)$
`update` $= \widehat{b} \oslash \widehat{b'}$
end function

$s_* = \texttt{NewtonGeneric}\,(\texttt{update\_fn}, s_0, \texttt{tol}, \texttt{max\_iter})$
end function

As seen in [JGH+13, Section 8], by using greater accuracy when computing the Jacobian $p'(s)$, we can mitigate the drastic increase in error as the condition number rises. By using `CompDeCasteljauDer` (Algorithm A.9) to compute $p'(s)$ we define a "fully accurate" modified Newton's method (`DNewtonFull`).

Algorithm 6.3 *Fully Accurate modified Newton's method for polynomial in Bernstein form.*

function $s_* = \texttt{DNewtonFull}(b, s_0, \texttt{tol}, \texttt{max\_iter})$
function `update` $=$ `update_fn`(s)
$\widehat{b} = \texttt{CompDeCasteljau}(b, s)$
$\widehat{b'} = \texttt{CompDeCasteljauDer}(b, s)$

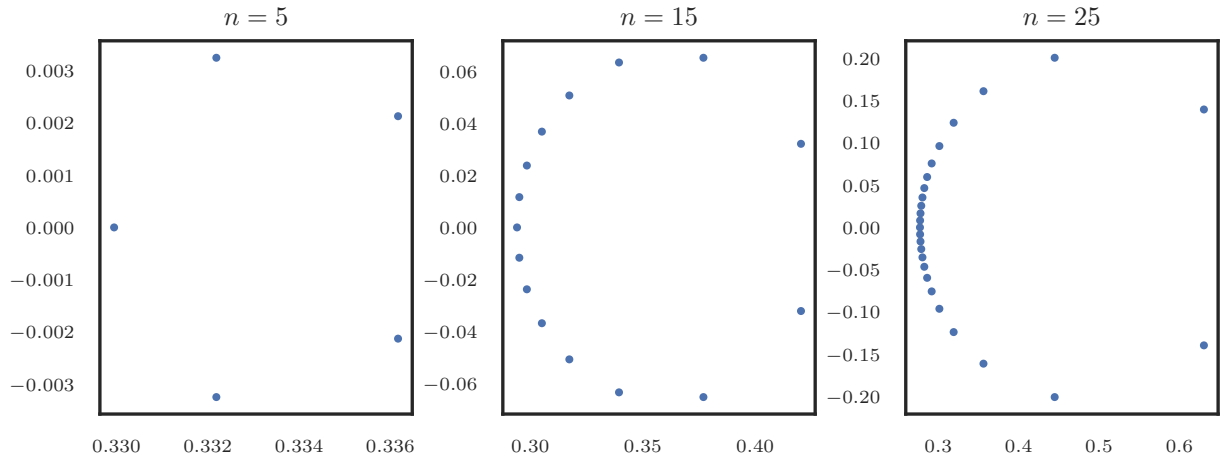

Figure 6.2: Roots of $p(s) = (1 - 5s)^n + 2^{30}(1 - 3s)^n$ for $n = 5, 15$ and 25.

$\texttt{update} = \widehat{b} \oslash \widehat{b'}$
end function

$s_* = \texttt{NewtonGeneric}\left(\texttt{update\_fn}, s_0, \texttt{tol}, \texttt{max\_iter}\right)$
end function

In order to verify the accuracy of the three proposed methods, we consider the family of polynomials

$$p(s) = (1 - 5s)^n + 2^d(1 - 3s)^n = \sum_{j=0}^{n} \left[(-4)^j + 2^d(-2)^j\right] B_{j,n}(s). \tag{6.22}$$

The coefficients can be represented exactly when $|(2j) - (j + d)| \leq 52$, so we specialize to $d = 30$ to get a suitable upper bound for n. We restrict to n odd so that $(1-5s)^n = 2^d(3s-1)^n$ has n distinct roots, only one of which is real. The distribution of the roots can be seen in Figure 6.2 for $n = 5, 15$ and 25.

To discuss the lone real root, we define $\omega \in \mathbf{R}$ such that $(1 + \omega)^n = 2^{30}$ and $\omega \to 0^+$. Solving $(1 - 5s) = (1 + \omega)(3s - 1)$ gives the root

$$\alpha = \frac{2 + \omega}{8 + 3\omega} \in \left[\frac{1}{4}, \frac{1}{3}\right]. \tag{6.23}$$

with condition number

$$\kappa_\alpha = \frac{\widetilde{p}(\alpha)}{\alpha\,|p'(\alpha)|} \sim \frac{7^n}{2^{20}\sqrt[7]{2}\;n}. \tag{6.24}$$

We use Newton's method with an absolute error tolerance of 10^{-15} and a maximum of 100 iterations. Since we know $\alpha \in [1/4, 1/3]$ we use $s_0 = 1/2$ as a starting point. Figure 6.3

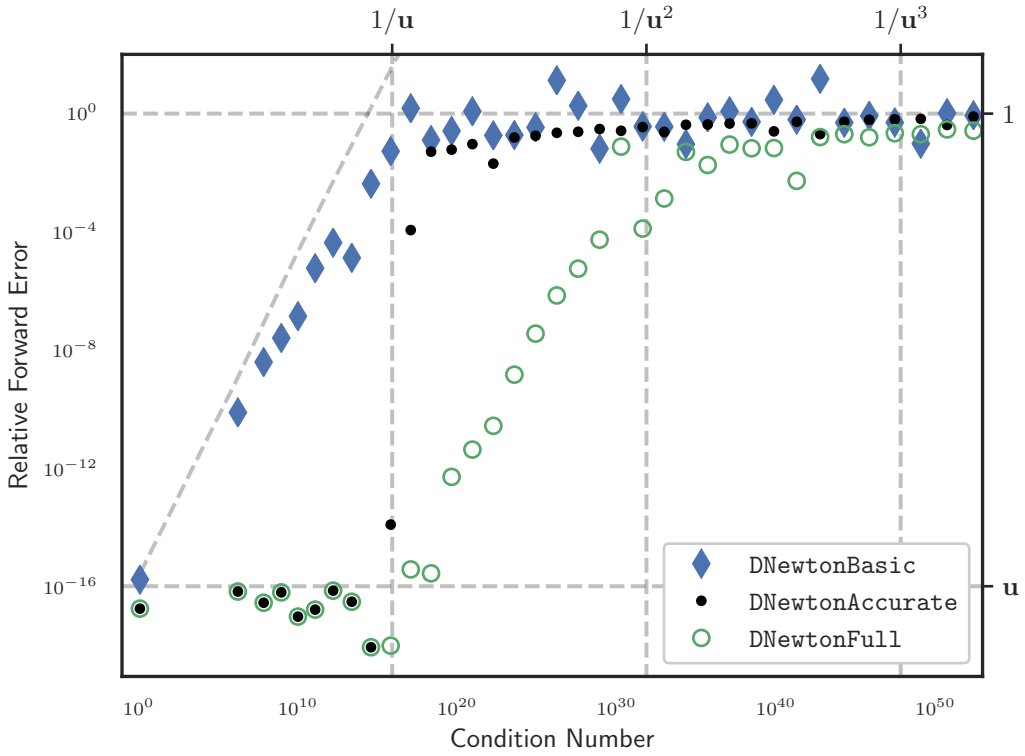

Figure 6.3: Comparing relative error to condition number when using Newton's method to find a root of $p(s) = (1-5s)^n + 2^{30}(1-3s)^n$ for n odd.

plots the relative error $|\alpha - \widehat{\alpha}| / \alpha$ against the condition number κ_α. Both the root α and the relative error are computed using a 500-bit extended precision arithmetic. As can be seen in the Figure, the standard Newton's method (`DNewtonBasic`) has relative error that is linear in κ_α. Both the first modified Newton's (`DNewtonAccurate`) and the the "fully accurate" modified Newton's (`DNewtonFull`) have $\mathcal{O}(\mathbf{u})$ errors until κ_α reaches $1/\mathbf{u}$. At this point, the accuracy in `DNewtonAccurate` totally collapses to $\mathcal{O}(1)$ while `DNewtonFull` follows a compensated rule of thumb (6.2). As expected, the relative error for `DNewtonFull` linearly increases from $\mathcal{O}(\mathbf{u})$ to $\mathcal{O}(1)$ as κ_α increases from $1/\mathbf{u}$ to $1/\mathbf{u}^2$, at which point the condition number overwhelms.

6.4 Bézier curve intersection

When intersecting curves, our residual $F(s,t) = b_1(s) - b_2(t)$ involved the evaluation of four polynomials in Bernstein form: $x_1(s), y_1(s), x_2(t), y_2(t)$. In order to compute a more accurate residual, the straightforward approach would just use `CompDeCasteljau` (Algo-

rithm 5.1) to compute each polynomial:

$$\widehat{F} = \begin{bmatrix} \texttt{CompDeCasteljau}(x_1, s) \ominus \texttt{CompDeCasteljau}(x_2, t) \\ \texttt{CompDeCasteljau}(y_1, s) \ominus \texttt{CompDeCasteljau}(y_2, t) \end{bmatrix}. \tag{6.25}$$

However, when the common value (e.g. $x_1(\alpha) = x_2(\beta)$) at an intersection is significantly larger than zero, this may not be helpful. When computing $\widehat{b} + \widehat{\partial b}$, the values may be so far apart in magnitude that the compensated value is the same as the non-compensated value. For example, consider the equation

$$1088 = s^2 + 60s + 1076 = 1076(1-s)^2 + 1106 \cdot 2(1-s)s + 1137s^2. \tag{6.26}$$

This has one root in the unit interval $\alpha = 4\sqrt{57} - 30 \approx 0.1993$. For a nearby value like $s = \frac{51}{256} + \frac{1}{2^{22}} \approx 0.1992$ we compute $\widehat{\partial b} = 512\mathbf{u}$, but this value is too small relative to 1088 so $\widehat{b} \oplus \widehat{\partial b} = \widehat{b}$.

At intersections, we expect $x_1(\alpha) - x_2(\beta)$ to be small but don't necessarily expect the common value to be small (similarly for the y-component). Hence we use `DeCasteljauEFT` (Algorithm A.10, the EFT associated with `CompDeCasteljau`) to separate the "large part" (computed value) from the "small part" (compensation term). When subtracting $x_2(t) = \widehat{x_2} + \partial x_2$ from $x_1(s) = \widehat{x_1} + \partial x_1$ at an intersection, the values $\widehat{x_j}$ may be large but their difference should be small and only that difference is suited to interact with the compensation terms: $[D, \sigma] = \texttt{TwoSum}(\widehat{x_1}, -\widehat{x_2})$. Then the computation becomes

$$[\widehat{x_1} + \partial x_1] - [\widehat{x_2} + \partial x_2] = D + (\partial x_1 - \partial x_2 + \sigma). \tag{6.27}$$

Unfortunately the compensation term $\tau = \partial x_1 - \partial x_2 + \sigma$ can't be computed without rounding, but $\widehat{\tau} = \left[\widehat{\partial x_1} \ominus \widehat{\partial x_2}\right] \oplus \sigma$ can be used instead.

With this, we specify a compensated algorithm (`CompCurveResidual`) for evaluating the residual $F(s, t)$:

Algorithm 6.4 *Compensated method for residual of Bézier curve intersection.*

function $\widehat{F} = \texttt{CompCurveResidual}(x_1, y_1, s, x_2, y_2, t)$

$\left[\widehat{x_1}, \widehat{\partial x_1}\right] = \texttt{DeCasteljauEFT}(x_1, s)$

$\left[\widehat{x_2}, \widehat{\partial x_2}\right] = \texttt{DeCasteljauEFT}(x_2, t)$

$[D, \sigma] = \texttt{TwoSum}(\widehat{x_1}, -\widehat{x_2})$

$\widehat{\tau} = \left[\widehat{\partial x_1} \ominus \widehat{\partial x_2}\right] \oplus \sigma$

$\widehat{F_0} = D \oplus \widehat{\tau}$

$\left[\widehat{y_1}, \widehat{\partial y_1}\right] = \texttt{DeCasteljauEFT}(y_1, s)$

$\left[\widehat{y_2}, \widehat{\partial y_2}\right] = \texttt{DeCasteljauEFT}(y_2, t)$

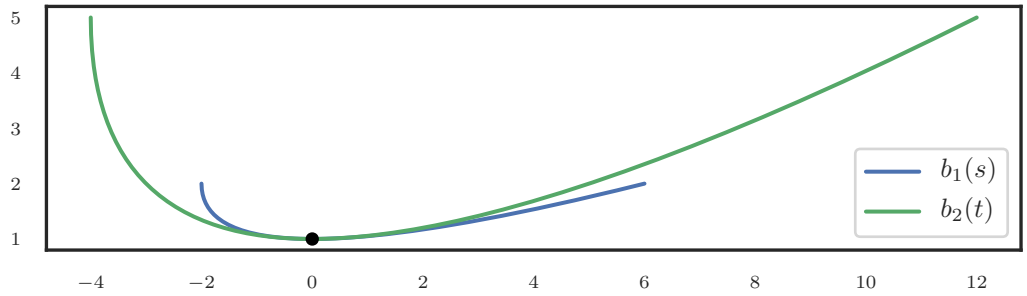

Figure 6.4: Intersection of two Bézier curves that are tangent and have the same curvature at the point of tangency.

$[D, \sigma] = \texttt{TwoSum}(\widehat{y_1}, -\widehat{y_2})$
$\widehat{\tau} = \left[\widehat{\partial y_1} \ominus \widehat{\partial y_2}\right] \oplus \sigma$
$\widehat{F_1} = D \oplus \widehat{\tau}$
end function

In order to compare Newton's method with the compensated residual to standard Newton's, we consider a few examples. The first is on an intersection (Figure 6.4) with infinite condition number:

$$F(s,t) = \begin{bmatrix} 2(4s^2 - 1) \\ (2s-1)^2 + 1 \end{bmatrix} - \begin{bmatrix} 4(4t^2 - 1) \\ 4(2t-1)^2 + 1 \end{bmatrix}. \tag{6.28}$$

These curves are tangent and have the same curvature when $\alpha = \beta = 1/2$; this is equivalent to a polynomial with a triple root.

When using Newton's method to find non-transveral intersections (i.e. roots where the Jacobian is singular), it is well-known that convergence will be linear rather than quadratic. However, it is less well known that convergence will typically stop prematurely due to loss of accuracy when computing the residual. For our example, we start with $s_0 = 1 - 2^{-40}$ and $t = 3/4 + 2^{-20}$; these slight perturbations are to avoid a "direct" path to the intersection. We stop converging once the length of the update vector is below 10^{-15} or after 50 iterations, whichever comes first. In Figure 6.5 we see that the standard Newton's method stops converging when the relative error is around $\sqrt[3]{\mathbf{u}}$, both for the s and t component of the intersection. This is because the cubic-like behavior of the intersection makes $\left(\sqrt[3]{\mathbf{u}}\right)^3$ resemble zero and falsely produces a zero residual (which results in a zero Newton update). The modified Newton's continues for much longer, until the relative error reaches $\sqrt[3]{\mathbf{u}^2}$.

It is from this example that we base our next numerical experiment. By introducing a small perturbation r to the x-component of one of the curves, we can make the tangent intersection split into three distinct intersections. Similarly, a large perturbation $1/r$ added to both y-components will leave the solution intact but will make the condition number

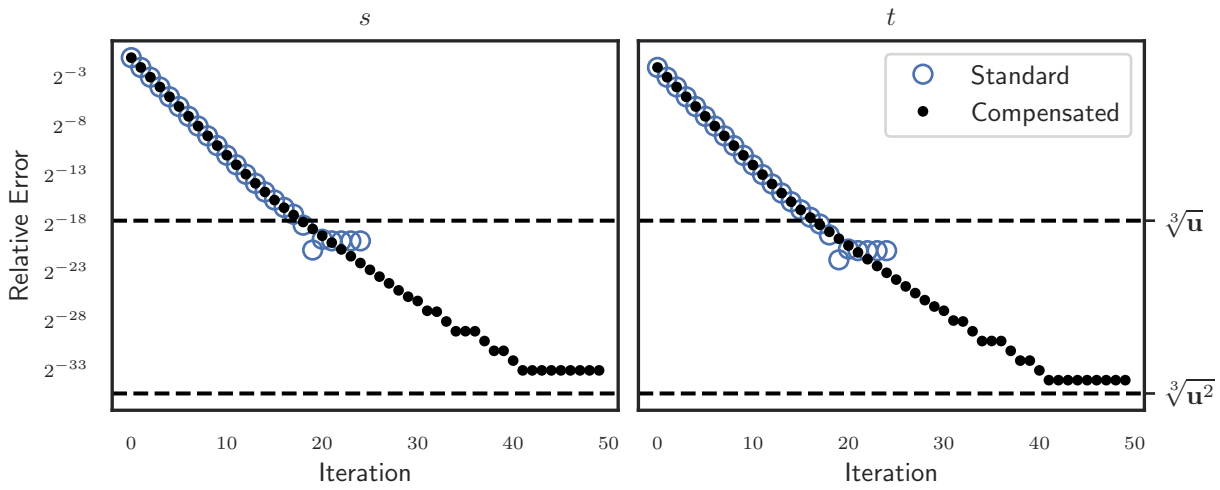

Figure 6.5: Relative error plots for the computed intersection $\alpha = \beta = 1/2$ when using standard Newton's method and a modified Newton's method that uses a compensated residual.

increase. This gives a family (parameterized by r):

$$F(s,t) = \begin{bmatrix} 2(4s^2-1) - r \\ (2s-1)^2 + 1 + 1/r \end{bmatrix} - \begin{bmatrix} 4(4t^2-1) \\ 4(2t-1)^2 + 1 + 1/r \end{bmatrix}. \tag{6.29}$$

We'll focus on $r = 2^{-n}$ for $2 \leq n \leq 50$, since for these values the coefficients of F can be represented exactly in $\mathbf{F}$. These curves intersect at

$$(\alpha_1, \beta_1) = \left(\frac{1+\sqrt{r}}{2}, \frac{2+\sqrt{r}}{4} \right), \tag{6.30}$$

$$(\alpha_2, \beta_2) = \left(\frac{1-\sqrt{r}}{2}, \frac{2-\sqrt{r}}{4} \right), \tag{6.31}$$

$$(\alpha_3, \beta_3) = \left(\frac{-3+\sqrt{16+r}}{2}, \frac{6-\sqrt{16+r}}{4} \right). \tag{6.32}$$

When starting Newton's method from the $s_0 = t_0 = 1$ (the "top-right" of the unit square), we'll converge to the first intersection (α_1, β_1). For our chosen values of r (which will determine which coefficients are positive, e.g. in $\widetilde{y}_2$), one can show that

$$\kappa_{\alpha_1,\beta_1} = \frac{\sqrt{10}}{2r^2} - \frac{3\sqrt{10}}{40r\sqrt{r}} + \mathcal{O}\left(\frac{1}{r}\right). \tag{6.33}$$

We compare the standard Newton's to the modified Newton's with a compensated residual as $r \to 0^+$. We use $s_0 = t_0 = 1$ as mentioned above and the same stopping criterion from the first example: stop once the length of the update vector is below 10^{-15} or after

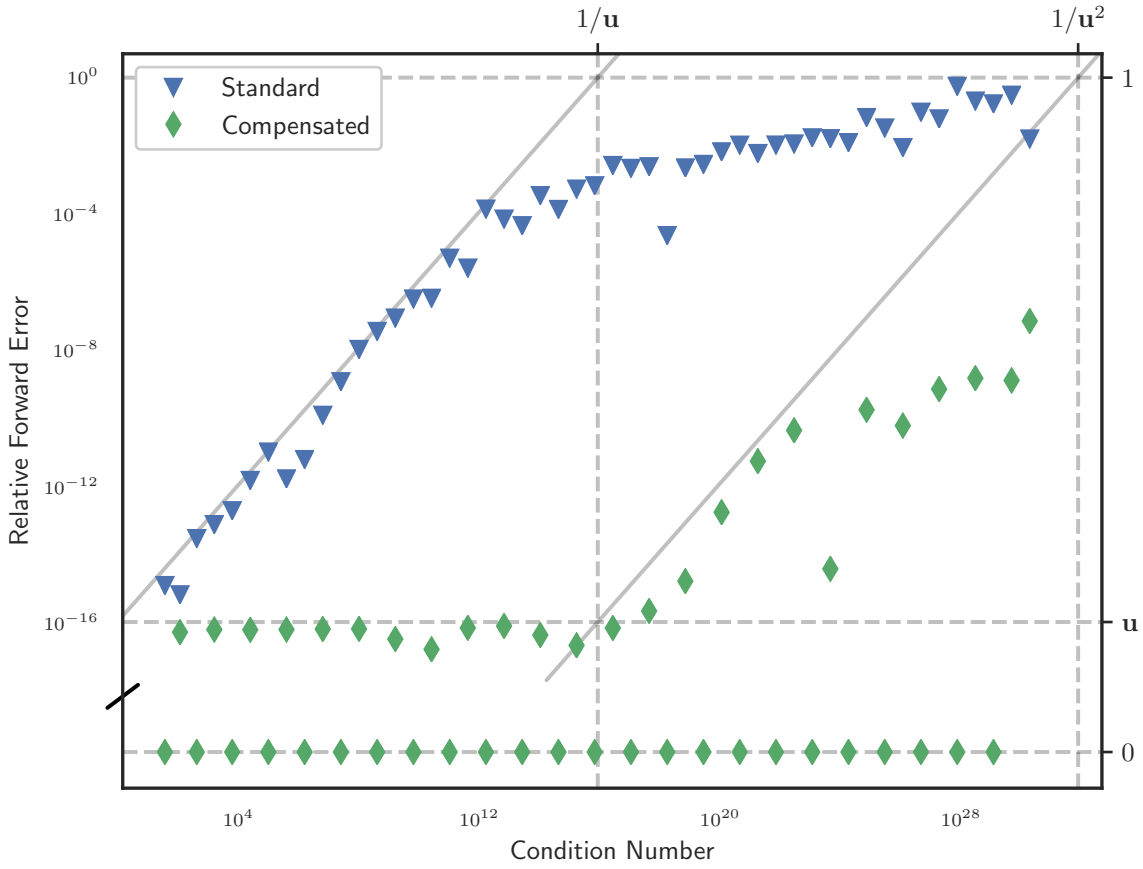

Figure 6.6: Relative error plots for the computed intersection $\alpha = \beta = 1/2$ when using standard Newton's method and a modified Newton's method that uses a compensated residual.

50 iterations, whichever comes first. As seen in Figure 6.6, the standard Newton's method has relative error that increases linearly with the condition number, i.e. $\mathcal{O}(\mathbf{u})\,\kappa_{\alpha_1,\beta_1}$, until it reaches $\mathcal{O}(1)$. The modified Newton's method satisfies a compensated rule of thumb (6.2): the relative error is $\mathcal{O}(\mathbf{u})$ (i.e. fully accurate up to rounding) until $\kappa_{\alpha_1,\beta_1}$ reaches $1/\mathbf{u}$ at which point the relative error resembles $\mathcal{O}(\mathbf{u}^2)\,\kappa_{\alpha_1,\beta_1}$. What's more, when $r = 2^{-2m}$ has a rational square root, the intersection can be represented exactly in $\mathbf{F}$ and the compensated method computes this intersection without any error.

6.5 False starts

It was quite challenging to generate families of polynomials with known ill-conditioned simple roots and families of Bézier curve pairs with known ill-conditioned transversal intersections. The issue was not in finding a family with moderately high condition number, but

in finding one where the condition number would grow past $1/\mathbf{u}$ and even as far as $1/\mathbf{u}^2$. In many failed attempts, as the condition increased in a given family, it became impossible to represent the coefficients exactly in $\mathbf{F}$. Once the coefficients are rounded, the roots or intertesections themselves will likely move erratically. This erratic behavior is the goal when generating ill-conditioned problems, but makes the values themselves less useful.

To give a sense of the nontrivial effort involved in finding a family, consider the following families of polynomials that **failed**:

- $p(s) = (1-s)^n - 2^{-d}$ at the simple root $s_* = 1 - 2^{-d/n}$. In the monomial basis, the condition number of s_* grows exponentially and was used as the example in [Gra08] and [JGH+13, Section 8]. However in the Bernstein basis, the root condition number is always less than 1.
- $p(s) = (as-1)^n - 2^{-d}(1-s)^n$ at the simple root $s_* = \frac{1+2^{-d/n}}{a+2^{-d/n}}$. This is problematic because when $a-1$ is not a power of 2, the coefficients in $(as-1)^n = [-(1-s) + (a-1)s]^n$ cannot be represented exactly in $\mathbf{F}$ for high enough degree. The rounded coefficients result in sporadic behavior for the resulting polynomials. When $a - 1 = 2^d$, Newton's method converges with much better than expected accuracy due to a unique quirk of the de Casteljau algorithm. This case is interesting enough to consider in greater detail in [Her18].
- $p(s) = (2s-1)^n - 2^{-52}$ at the simple root $s_* = \frac{1+2^{-52/n}}{2}$. The coefficients $b_j = (-1)^{n-j} - 2^{-52}$ can always be represented in $\mathbf{F}$, but as $n \to \infty$ we have $s_* \to 1$. A root exactly equal to 1 is perfectly conditioned, so unsurprisingly $\kappa \to 0$ after reaching a maximum value of $\approx 3 \cdot 10^{13}$ when $n = 28$.
- $p(s) = (3s-1)^n - 2^{-d}$ at the simple root $s_* = \frac{1+2^{-d/n}}{3}$. The coefficients $b_j = (-1)^{n-j}2^j - 2^{-d}$ can be represented exactly in $\mathbf{F}$ for $|j+d| \leq 52$ (or 53, depending on n). This limitation stops the condition number from growing large enough. For example, when $d = 10$, the largest term that can be represented exactly is $\kappa_{43} \approx 1.03 \cdot 10^{10}$, when $d = 20$, the largest term that can be represented exactly is $\kappa_{33} \approx 2.53 \cdot 10^{10}$ and when $d = 30$, the largest term that can be represented exactly is $\kappa_{23} \approx 9.24 \cdot 10^{10}$.

In addition, several false starts were made when generating a family of Bézier curve pairs with ill-conditioned intersection. In the case of polynomials we increased the condition number via the degree, but in the case of curve pairs it was increased by reducing a continuous parameter towards zero at which point the pair becomes tangent. The most problematic case came from

$$F(s,t;r) = \begin{bmatrix} 2(4s^2-1)+r \\ (2s-1)^2 \end{bmatrix} - \begin{bmatrix} 4(4t^2-1) \\ 4(2t-1)^2 \end{bmatrix}. \tag{6.34}$$

In this case, the polynomials $(2s-1)^2$ and $4(2t-1)^2$ come from a very special class of polynomials described in [Her18] where the de Casteljau algorithm is very accurate (much better than the a priori upper bound). As a result, the y-contributions to the relative error were essentially zero. This can be thought of as setting $\mu_2 = 0$ when computing the

condition number. However, for this family of curves the x-components contribute $\mathcal{O}(1)$ to the condition number while the y-components *should* contribute $\mathcal{O}(1/r)$ but instead contribution nothing. Hence the solution can always be computed by Newton's method with $\mathcal{O}(\mathbf{u})$ relative error.

Bibliography

[Ber87] Marsha J. Berger. On Conservation at Grid Interfaces. *SIAM Journal on Numerical Analysis*, 24(5):967–984, Oct 1987.

[BHSW08] Daniel J. Bates, Jonathan D. Hauenstein, Andrew J. Sommese, and Charles W. Wampler. Adaptive Multiprecision Path Tracking. *SIAM Journal on Numerical Analysis*, 46(2):722–746, Jan 2008.

[BR78] I. Babuška and W. C. Rheinboldt. Error Estimates for Adaptive Finite Element Computations. *SIAM Journal on Numerical Analysis*, 15(4):736–754, 1978.

[CDS99] Xiao-Chuan Cai, Maksymilian Dryja, and Marcus Sarkis. Overlapping Nonmatching Grid Mortar Element Methods for Elliptic Problems. *SIAM Journal on Numerical Analysis*, 36(2):581–606, Jan 1999.

[CH94] G. Chesshire and W. D. Henshaw. A Scheme for Conservative Interpolation on Overlapping Grids. *SIAM Journal on Scientific Computing*, 15(4):819–845, Jul 1994.

[Che74] Patrick Chenin. *Quelques problèmes d'interpolation à plusieurs variables*. Theses, Institut National Polytechnique de Grenoble - INPG ; Université Joseph-Fourier - Grenoble I, June 1974. Université : Université scientifique et médicale de Grenoble et Institut National Polytechnique.

[CMOP04] David E. Cardoze, Gary L. Miller, Mark Olah, and Todd Phillips. A Bézier-Based Moving Mesh Framework for Simulation with Elastic Membranes. In *Proceedings of the 13th International Meshing Roundtable, IMR 2004, Williamsburg, Virginia, USA, September 19-22, 2004*, pages 71–80, 2004.

[Dek71] T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, Jun 1971.

[DK87] John K. Dukowicz and John W. Kodis. Accurate Conservative Remapping (Rezoning) for Arbitrary Lagrangian-Eulerian Computations. *SIAM Journal on Scientific and Statistical Computing*, 8(3):305–321, May 1987.

[DP15] Jorge Delgado and J.M. Peña. Accurate evaluation of Bézier curves and surfaces and the Bernstein-Fourier algorithm. *Applied Mathematics and Computation*, 271:113–122, Nov 2015.

[Duk84] John K Dukowicz. Conservative rezoning (remapping) for general quadrilateral meshes. *Journal of Computational Physics*, 54(3):411–424, Jun 1984.

[Dun85] D. A. Dunavant. High degree efficient symmetrical Gaussian quadrature rules for the triangle. *International Journal for Numerical Methods in Engineering*, 21(6):1129–1148, Jun 1985.

[Far91] R.T. Farouki. On the stability of transformations between power and Bernstein polynomial forms. *Computer Aided Geometric Design*, 8(1):29–36, Feb 1991.

[Far01] Gerald Farin. *Curves and Surfaces for CAGD, Fifth Edition: A Practical Guide (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann, 2001.

[Flo03] Michael S. Floater. Mean value coordinates. *Computer Aided Geometric Design*, 20(1):19–27, Mar 2003.

[FM11] P.E. Farrell and J.R. Maddison. Conservative interpolation between volume meshes by local Galerkin projection. *Computer Methods in Applied Mechanics and Engineering*, 200(1-4):89–100, Jan 2011.

[FPP+09] P.E. Farrell, M.D. Piggott, C.C. Pain, G.J. Gorman, and C.R. Wilson. Conservative interpolation between unstructured meshes via supermesh construction. *Computer Methods in Applied Mechanics and Engineering*, 198(33-36):2632–2642, Jul 2009.

[FR87] R.T. Farouki and V.T. Rajan. On the numerical condition of polynomials in Bernstein form. *Computer Aided Geometric Design*, 4(3):191–216, Nov 1987.

[GKS07] Rao Garimella, Milan Kucharik, and Mikhail Shashkov. An efficient linearity and bound preserving conservative interpolation (remapping) on polyhedral meshes. *Computers & Fluids*, 36(2):224–237, Feb 2007.

[GLL09] Stef Graillat, Philippe Langlois, and Nicolas Louvet. Algorithms for accurate, validated and fast polynomial evaluation. *Japan Journal of Industrial and Applied Mathematics*, 26(2-3):191–214, Oct 2009.

[Gra08] Stef Graillat. Accurate simple zeros of polynomials in floating point arithmetic. *Computers & Mathematics with Applications*, 56(4):1114–1120, Aug 2008.

[GT82] William J. Gordon and Linda C. Thiel. Transfinite mappings and their application to grid generation. *Applied Mathematics and Computation*, 10-11:171–233, Jan 1982.

[HAC74] C.W Hirt, A.A Amsden, and J.L Cook. An arbitrary Lagrangian-Eulerian computing method for all flow speeds. *Journal of Computational Physics*, 14(3):227–253, Mar 1974.

[Her17] Danny Hermes. Helper for Bézier Curves, Triangles, and Higher Order Objects. *The Journal of Open Source Software*, 2(16):267, Aug 2017.

[Her18] Danny Hermes. A Curious Case of Curbed Condition. *ArXiv e-prints*, Jun 2018.

[Hig02] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Jan 2002.

[IK04] Armin Iske and Martin Käser. Conservative semi-Lagrangian advection on adaptive unstructured meshes. *Numerical Methods for Partial Differential Equations*, 20(3):388–411, Feb 2004.

[JGH+13] Hao Jiang, Stef Graillat, Canbin Hu, Shengguo Li, Xiangke Liao, Lizhi Cheng, and Fang Su. Accurate evaluation of the k-th derivative of a polynomial and its application. *Journal of Computational and Applied Mathematics*, 243:28–47, May 2013.

[JH04] Xiangmin Jiao and Michael T. Heath. Common-refinement-based data transfer between non-matching meshes in multiphysics simulations. *International Journal for Numerical Methods in Engineering*, 61(14):2402–2427, 2004.

[JLCS10] Hao Jiang, Shengguo Li, Lizhi Cheng, and Fang Su. Accurate evaluation of a polynomial and its derivative in Bernstein form. *Computers & Mathematics with Applications*, 60(3):744–755, Aug 2010.

[JM09] Claes Johnson and Mathematics. *Numerical Solution of Partial Differential Equations by the Finite Element Method (Dover Books on Mathematics)*. Dover Publications, 2009.

[Kah72] William Kahan. Conserving confluence curbs ill-condition. Technical report, UC Berkeley Department of Computer Science, Aug 1972.

[KLS98] Deok-Soo Kim, Soon-Woong Lee, and Hayong Shin. A cocktail algorithm for planar Bézier curve intersections. *Computer-Aided Design*, 30(13):1047–1051, Nov 1998.

[Knu97] Donald E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition)*. Addison-Wesley Professional, 1997.

[KS08] M. Kucharik and M. Shashkov. Extension of efficient, swept-integration-based conservative remapping method for meshes with changing connectivity. *International Journal for Numerical Methods in Fluids*, 56(8):1359–1365, 2008.

[LGL06] Philippe Langlois, Stef Graillat, and Nicolas Louvet. Compensated Horner Scheme. In Bruno Buchberger, Shin'ichi Oishi, Michael Plum, and Sigfried M. Rump, editors, *Algebraic and Numerical Algorithms and Computer-assisted Proofs*, number 05391 in Dagstuhl Seminar Proceedings, pages 1–29, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[MD92] Dinesh Manocha and James W. Demmel. Algorithms for Intersecting Parametric and Algebraic Curves. Technical Report UCB/CSD-92-698, EECS Department, University of California, Berkeley, Aug 1992.

[MM72] R. McLeod and A. R. Mitchell. The Construction of Basis Functions for Curved Elements in the Finite Element Method. *IMA Journal of Applied Mathematics*, 10(3):382–393, 1972.

[MP99] E. Mainar and J.M. Peña. Error analysis of corner cutting algorithms. *Numerical Algorithms*, 22(1):41–52, 1999.

[MP05] E. Mainar and J. M. Peña. Running Error Analysis of Evaluation Algorithms for Bivariate Polynomials in Barycentric Bernstein Form. *Computing*, 77(1):97–111, Dec 2005.

[MS03] L.G. Margolin and Mikhail Shashkov. Second-order sign-preserving conservative interpolation (remapping) on general grids. *Journal of Computational Physics*, 184(1):266–298, Jan 2003.

[MXS09] S. E. Mousavi, H. Xiao, and N. Sukumar. Generalized Gaussian quadrature rules on arbitrary polygons. *International Journal for Numerical Methods in Engineering*, 2009.

[ORO05] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate Sum and Dot Product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, Jan 2005.

[PBP09] P.-O. Persson, J. Bonet, and J. Peraire. Discontinuous Galerkin solution of the Navier–Stokes equations on deformable domains. *Computer Methods in Applied Mechanics and Engineering*, 198(17-20):1585–1595, Apr 2009.

[Per98] Alain Perronnet. Triangle, tetrahedron, pentahedron transfinite interpolations. Application to the generation of C0 or G1-continuous algebraic meshes. In *Proc. Int. Conf. Numerical Grid Generation in Computational Field Simulations, Greenwich, England*, pages 467–476, 1998.

[PUdOG01] C.C. Pain, A.P. Umpleby, C.R.E. de Oliveira, and A.J.H. Goddard. Tetrahedral mesh optimisation and adaptivity for steady-state and transient finite element calculations. *Computer Methods in Applied Mechanics and Engineering*, 190(29-30):3771–3796, Apr 2001.

[PVMZ87] J Peraire, M Vahdati, K Morgan, and O.C Zienkiewicz. Adaptive remeshing for compressible flow computations. *Journal of Computational Physics*, 72(2):449–466, Oct 1987.

[SN90] T.W. Sederberg and T. Nishita. Curve intersection using Bézier clipping. *Computer-Aided Design*, 22(9):538–549, Nov 1990.

[SP86] Thomas W Sederberg and Scott R Parry. Comparison of three curve intersection algorithms. *Computer-Aided Design*, 18(1):58–63, Jan 1986.

[Tis01] Françoise Tisseur. Newton's Method in Floating Point Arithmetic and Iterative Refinement of Generalized Eigenvalue Problems. *SIAM Journal on Matrix Analysis and Applications*, 22(4):1038–1057, Jan 2001.

[Wac75] Eugene Wachspress. *A Rational Finite Element Basis*. Academic Press, Amsterdam, Boston, 1975.

[WFA+13] Z.J. Wang, Krzysztof Fidkowski, Rémi Abgrall, Francesco Bassi, Doru Caraeni, Andrew Cary, Herman Deconinck, Ralf Hartmann, Koen Hillewaert, H.T. Huynh, Norbert Kroll, Georg May, Per-Olof Persson, Bram van Leer, and Miguel Visbal. High-order CFD methods: current status and perspective. *International Journal for Numerical Methods in Fluids*, 72(8):811–845, Jan 2013.

[Zlá73] Miloš Zlámal. Curved Elements in the Finite Element Method. I. *SIAM Journal on Numerical Analysis*, 10(1):229–240, Mar 1973.

[Zlá74] Miloš Zlámal. Curved Elements in the Finite Element Method. II. *SIAM Journal on Numerical Analysis*, 11(2):347–362, Apr 1974.

A

Algorithms

Find here concrete implementation details on the EFTs described in Theorem 2.1. They do not use branches, nor access to the mantissa that can be time-consuming.

Algorithm A.1 *EFT of the sum of two floating point numbers.*

function $[S, \sigma] = \texttt{TwoSum}(a, b)$
$S = a \oplus b$
$z = S \ominus a$
$\sigma = (a \ominus (S \ominus z)) \oplus (b \ominus z)$
end function

In order to avoid branching to check which among $|a|, |b|$ is largest, `TwoSum` uses 6 flops rather than 3.

Algorithm A.2 *Splitting of a floating point number into two parts.*

function $[h, \ell] = \texttt{Split}(a)$
$z = a \otimes (2^r + 1)$
$h = z \ominus (z \ominus a)$
$\ell = a \ominus h$
end function

For IEEE-754 double precision floating point number, $r = 27$ so $2^r + 1$ will be known before `Split` is called. In all, `Split` uses 4 flops.

Algorithm A.3 *EFT of the product of two floating point numbers.*

function $[P, \pi] = \texttt{TwoProd}(a, b)$
$P = a \otimes b$
$[a_h, a_\ell] = \texttt{Split}(a)$
$[b_h, b_\ell] = \texttt{Split}(b)$
$\pi = a_\ell \otimes b_\ell \ominus (((P \ominus a_h \otimes b_h) \ominus a_\ell \otimes b_h) \ominus a_h \otimes b_\ell)$
end function

This implementation of `TwoProd` requires 17 flops. For processors that provide a fused-multipy-add operator (`FMA`), `TwoProd` can be rewritten to use only 2 flops:

Algorithm A.4 *EFT of the sum of two floating point numbers with a FMA.*

function $[P, \pi] = \texttt{TwoProdFMA}(a, b)$
  $P = a \otimes b$
  $\pi = \texttt{FMA}(a, b, -P)$
end function

The following algorithms from [ORO05] can be used as a compensated method for computing a sum of numbers. The first is a vector transformation that is used as a helper:

Algorithm A.5 *Error-free vector transformation for summation.*

function $\texttt{VecSum}(p)$
  $n = \texttt{length}(p)$
  for $j = 2, \ldots, n$ **do**
    $[p_j, p_{j-1}] = \texttt{TwoSum}\,(p_j, p_{j-1})$
  end for
end function

The second (`SumK`) computes a sum with results that are as accurate as if computed in K times the working precision. It requires $(6K - 5)(n - 1)$ floating point operations.

Algorithm A.6 *Summation as in K-fold precision by $(K-1)$-fold error-free vector transformation.*

function $\texttt{result} = \texttt{SumK}(p, K)$
  for $j = 1, \ldots, K - 1$ **do**
    $p = \texttt{VecSum}(p)$
  end for
  $\texttt{result} = p_1 \oplus p_2 \oplus \cdots \oplus p_n$
end function

Since the final error $\widehat{\partial^{K-1} b}$ will not track the errors during computation, we have a non-EFT version of Algorithm 5.2:

Algorithm A.7 *Compute the local error (non-EFT).*

function $\widehat{\ell} = \texttt{LocalError}(e, \rho, \delta b)$
  $L = \texttt{length}(e)$

  $\widehat{\ell} = e_1 \oplus e_2$

for $j = 3, \ldots, L$ **do**
$\widehat{\ell} = \widehat{\ell} \oplus e_j$
end for

$\widehat{\ell} = \widehat{\ell} \oplus (\rho \otimes \delta b)$
end function

In order to discuss varied implementations of Newton's method in Chapter 6, we define a generic algorithm that takes an "update function" (`update_fn`), i.e. a callable that will produce the next Newton update $p(s)/p'(s)$ computed in a problem-specific way.

Algorithm A.8 *Generic Newton's method for scalar functions.*

function $x_* =$ `NewtonGeneric`$($`update_fn`$, x_0,$ `tol`$,$ `max_iter`$)$
$x = x_0$

for $j = 1, \ldots,$ `max_iter` **do**
`update` $=$ `update_fn`(x)
$x = x \ominus$ `update`
if $|$`update`$| <$ `tol` **then**
break
end if
end for

$x_* = x$
end function

In [JLCS10, Algorithm 3], the `CompDeCasteljau` (Algorithm 5.1) is modified for the computation of the derivative $p'(s)$. Since $p'(s)$ has coefficients $n\Delta b_j$ where $c_j = \Delta b_j = b_{j+1} - b_j$, we can begin the computation with nonzero $\widehat{\partial c}_j$ (as opposed to $\widehat{\partial b}_j^{(n)} = 0$ in `CompDeCasteljau`).

Algorithm A.9 *Compensated de Casteljau algorithm for polynomial first derivative evaluation.*

function `result` $=$ `CompDeCasteljauDer`(b, s)
$n =$ `length`$(b) - 1$
$[\widehat{r}, \rho] =$ `TwoSum`$(1, -s)$

for $j = 0, \ldots, n-1$ **do**
$\left[\widehat{c}_j^{(n-1)}, \widehat{\partial c}_j^{(n-1)}\right] =$ `TwoSum`$(b_{j+1}, -b_j)$
end for

for $k = n-2, \ldots, 0$ **do**

for $j = 0, \ldots, k$ **do**
$[P_1, \pi_1] = \texttt{TwoProd}\left(\widehat{r}, \widehat{c}_j^{(k+1)}\right)$
$[P_2, \pi_2] = \texttt{TwoProd}\left(s, \widehat{c}_{j+1}^{(k+1)}\right)$
$\left[\widehat{c}_j^{(k)}, \sigma_3\right] = \texttt{TwoSum}(P_1, P_2)$
$\widehat{\ell}_{1,j}^{(k)} = \pi_1 \oplus \pi_2 \oplus \sigma_3 \oplus \left(\rho \otimes \widehat{c}_j^{(k+1)}\right)$
$\widehat{\partial c}_j^{(k)} = \widehat{\ell}_{1,j}^{(k)} \oplus \left(s \otimes \widehat{\partial c}_{j+1}^{(k+1)}\right) \oplus \left(\widehat{r} \otimes \widehat{\partial c}_j^{(k+1)}\right)$
end for
end for

$\texttt{result} = n \otimes \left[\widehat{c}_0^{(0)} \oplus \widehat{\partial c}_0^{(0)}\right]$
end function

For an applied usage of `CompDeCasteljau` (Algorithm 5.1) in Chapter 6, both the non-compensated value $\widehat{b}$ and the compensation term $\widehat{\partial b}$ are needed. So we define a partial EFT. We say **partial** because the compensation term is rounded.

Algorithm A.10 *EFT for de Casteljau algorithm for polynomial evaluation.*

function $\left[\widehat{b}, \widehat{\partial b}\right] = \texttt{DeCasteljauEFT}(b, s)$
$n = \texttt{length}(b) - 1$
$[\widehat{r}, \rho] = \texttt{TwoSum}(1, -s)$

for $j = 0, \ldots, n$ **do**
$\widehat{b}_j^{(n)} = b_j$
$\widehat{\partial b}_j^{(n)} = 0$
end for

for $k = n-1, \ldots, 0$ **do**
for $j = 0, \ldots, k$ **do**
$[P_1, \pi_1] = \texttt{TwoProd}\left(\widehat{r}, \widehat{b}_j^{(k+1)}\right)$
$[P_2, \pi_2] = \texttt{TwoProd}\left(s, \widehat{b}_{j+1}^{(k+1)}\right)$
$\left[\widehat{b}_j^{(k)}, \sigma_3\right] = \texttt{TwoSum}(P_1, P_2)$
$\widehat{\ell}_{1,j}^{(k)} = \pi_1 \oplus \pi_2 \oplus \sigma_3 \oplus \left(\rho \otimes \widehat{b}_j^{(k+1)}\right)$
$\widehat{\partial b}_j^{(k)} = \widehat{\ell}_{1,j}^{(k)} \oplus \left(s \otimes \widehat{\partial b}_{j+1}^{(k+1)}\right) \oplus \left(\widehat{r} \otimes \widehat{\partial b}_j^{(k+1)}\right)$
end for
end for

$\widehat{b} = \widehat{b}_0^{(0)}$
$\widehat{\partial b} = \widehat{\partial b}_0^{(0)}$
end function

B

Proof Details

Proof of Lemma 5.2. We'll start with the $F = 1$ case. Recall where the terms originate:

$$[P_1, e_1] = \texttt{TwoProd}\left(\widehat{r}, \widehat{b}_j^{(k+1)}\right) \tag{B.1}$$

$$[P_2, e_2] = \texttt{TwoProd}\left(s, \widehat{b}_{j+1}^{(k+1)}\right) \tag{B.2}$$

$$\left[\widehat{b}_j^{(k)}, e_3\right] = \texttt{TwoSum}\left(P_1, P_2\right). \tag{B.3}$$

Hence Theorem 2.1 tells us that

$$|P_1| \leq (1+\mathbf{u})\left|\widehat{r}\cdot\widehat{b}_j^{(k+1)}\right| \leq (1+\mathbf{u})^2(1-s)\left|\widehat{b}_j^{(k+1)}\right| \tag{B.4}$$

$$|e_1| \leq \mathbf{u}\left|\widehat{r}\cdot\widehat{b}_j^{(k+1)}\right| \leq \mathbf{u}(1+\mathbf{u})(1-s)\left|\widehat{b}_j^{(k+1)}\right| \tag{B.5}$$

$$|P_2| \leq (1+\mathbf{u})s\left|\widehat{b}_{j+1}^{(k+1)}\right| \tag{B.6}$$

$$|e_2| \leq \mathbf{u}s\left|\widehat{b}_{j+1}^{(k+1)}\right| \tag{B.7}$$

$$|e_3| \leq \mathbf{u}\,|P_1| + \mathbf{u}\,|P_2| \tag{B.8}$$

$$\left|\rho\cdot\widehat{b}_j^{(k+1)}\right| \leq (1+\mathbf{u})(1-s)\left|\widehat{b}_j^{(k+1)}\right|. \tag{B.9}$$

In general, we can swap $\mathbf{u}\,|P_j|$ for $(1+\mathbf{u})\,|e_j|$ based on how closely related the bound on the result and the bound on the error are. Thus

$$\widetilde{\ell}_{1,j}^{(k)} = |e_1| + |e_2| + |e_3| + \left|\rho\cdot\widehat{b}_j^{(k+1)}\right| \tag{B.10}$$

$$\leq (2+\mathbf{u})\left(|e_1| + |e_2|\right) + (1+\mathbf{u})(1-s)\left|\widehat{b}_j^{(k+1)}\right| \tag{B.11}$$

$$\leq \left[(1+\mathbf{u})^3 - 1\right](1-s)\left|\widehat{b}_j^{(k+1)}\right| + \left[(1+\mathbf{u})^2 - 1\right]s\left|\widehat{b}_{j+1}^{(k+1)}\right| \tag{B.12}$$

$$\leq \gamma_3\left((1-s)\left|\widehat{b}_j^{(k+1)}\right| + s\left|\widehat{b}_{j+1}^{(k+1)}\right|\right). \tag{B.13}$$

For $\widetilde{\ell}_{F+1}$, we want to relate the "current" errors $e_1, \ldots, e_{5F+3}$ to the "previous" errors $e'_1, \ldots, e'_{5F-2}$ that show up in $\widetilde{\ell}_F$. In the same fashion as above, we track where the current errors come from:

$$[S_1, e_1] = \texttt{TwoSum}\,(e'_1, e'_2) \tag{B.14}$$
$$[S_2, e_2] = \texttt{TwoSum}\,(S_1, e'_3) \tag{B.15}$$
$$\vdots$$
$$[S_{5F-3}, e_{5F-3}] = \texttt{TwoSum}\left(S_{5F-4}, e'_{5F-2}\right) \tag{B.16}$$
$$[P_{5F-2}, e_{5F-2}] = \texttt{TwoProd}\left(\rho, \widehat{\partial^{F-1} b_j}^{(k+1)}\right) \tag{B.17}$$
$$\left[\widehat{\ell}^{(k)}_{F,j}, e_{5F-1}\right] = \texttt{TwoSum}\,(S_{5F-3}, P_{5F-2}) \tag{B.18}$$
$$[P_{5F}, e_{5F}] = \texttt{TwoProd}\left(s, \widehat{\partial^{F} b_{j+1}}^{(k+1)}\right) \tag{B.19}$$
$$[S_{5F+1}, e_{5F+1}] = \texttt{TwoSum}\left(\widehat{\ell}^{(k)}_{F,j}, P_{5F}\right) \tag{B.20}$$
$$[P_{5F+2}, e_{5F+2}] = \texttt{TwoProd}\left(\rho, \widehat{\partial^{F} b_j}^{(k+1)}\right) \tag{B.21}$$
$$\left[\widehat{\partial^{F} b_j}^{(k)}, e_{5F+3}\right] = \texttt{TwoSum}\,(S_{5F+1}, P_{5F+2})\,. \tag{B.22}$$

Arguing as we did above, we start with $|e_1| \leq \mathbf{u}\,|e'_1| + \mathbf{u}\,|e'_2|$ and build each bound recursively based on the previous, e.g. $|e_2| \leq \mathbf{u}\,|S_1| + \mathbf{u}\,|e'_3| \leq (1+\mathbf{u})\mathbf{u}\,|e'_1| + (1+\mathbf{u})\mathbf{u}\,|e'_2| + \mathbf{u}\,|e'_3|$. Proceeding in this fashion, we find

$$\widetilde{\ell}^{(k)}_{F+1,j} = |e_1| + \cdots + |e_{5F+3}| + \left|\rho \cdot \widehat{\partial^F b_j}^{(k+1)}\right| \tag{B.23}$$
$$\leq \gamma_{5F}\,|e'_1| + \gamma_{5F}\,|e'_2| + \gamma_{5F-1}\,|e'_3| + \cdots + \gamma_4\left|e'_{5F-2}\right| + \gamma_4\left|\rho \cdot \widehat{\partial^{F-1} b_j}^{(k+1)}\right| \tag{B.24}$$
$$+ \gamma_3(1-s)\left|\widehat{\partial^F b_j}^{(k+1)}\right| + \gamma_3 s\left|\widehat{\partial^F b_{j+1}}^{(k+1)}\right| \tag{B.25}$$
$$\leq \gamma_3\left((1-s)\left|\widehat{\partial^F b_j}^{(k+1)}\right| + s\left|\widehat{\partial^F b_{j+1}}^{(k+1)}\right|\right) + \gamma_{5F} \cdot \widetilde{\ell}^{(k)}_{F,j} \tag{B.26}$$

as desired. ■

Proof of Lemma 5.3. First, note that for **any** sequence $v_0, \ldots, v_{k+1}$ we must have

$$\sum_{j=0}^{k} \left[(1-s)v_j + sv_{j+1}\right] B_{j,k}(s) = \sum_{j=0}^{k+1} v_j B_{j,k+1}(s). \tag{B.27}$$

For example of this in use, via (5.26), we have

$$L_{1,k} \leq \gamma_3 \sum_{j=0}^{k+1} \left|\widehat{b}_j^{(k+1)}\right| B_{j,k+1}(s). \tag{B.28}$$

In order to work with sums of this form, we define Bernstein-type sums related to $L_{F,k}$:

$$D_{0,k} \coloneqq \sum_{j=0}^{k} \left|\widehat{b}_j^{(k)}\right| B_{j,k}(s) \tag{B.29}$$

$$D_{F,k} \coloneqq \sum_{j=0}^{k} \left|\widehat{\partial^F b}_j^{(k)}\right| B_{j,k}(s). \tag{B.30}$$

Hence Lemma 5.2 gives

$$L_{1,k} \leq \gamma_3 D_{0,k+1} \tag{B.31}$$

$$L_{F+1,k} \leq \gamma_3 D_{F,k+1} + \gamma_{5F} L_{F,k} \tag{B.32}$$

In addition, for $F \geq 1$ since

$$\widehat{\partial^F b}_j^{(k)} = \widehat{\ell}_{F,j}^{(k)} \oplus \left(s \otimes \widehat{\partial^F b}_{j+1}^{(k+1)}\right) \oplus \left((1 \ominus s) \otimes \widehat{\partial^F b}_j^{(k+1)}\right) \tag{B.33}$$

$$= (1-s) \cdot \widehat{\partial^F b}_j^{(k+1)}(1+\theta_3) + s \cdot \widehat{\partial^F b}_{j+1}^{(k+1)}(1+\theta_3) + \widehat{\ell}_{F,j}^{(k)}(1+\theta_2) \tag{B.34}$$

we have

$$D_{F,k} \leq (1+\gamma_3) D_{F,k+1} + (1+\gamma_2) \sum_{j=0}^{k} \left|\widehat{\ell}_{F,j}^{(k)}\right| B_{j,k}(s). \tag{B.35}$$

Since $\ell_{F,j}^{(k)}$ has $5F-1$ terms (only the last of which involves a product), the terms in the computed value will be involved in at most $5F-2$ flops, hence $\left|\widehat{\ell}_{F,j}^{(k)}\right| \leq (1+\gamma_{5F-2})\, \widetilde{\ell}_{F,j}^{(k)}$. Combined with (B.35) and the fact that there is no local error when $F = 0$, this means

$$D_{0,k} \leq (1+\gamma_3) D_{0,k+1} \tag{B.36}$$

$$D_{F,k} \leq (1+\gamma_3) D_{F,k+1} + (1+\gamma_{5F}) L_{F,k}. \tag{B.37}$$

The four inequalities (B.31), (B.32), (B.36) and (B.37) allow us to write all bounds in terms of $D_{0,n} = \widetilde{p}(s)$ and $D_{F,n} = 0$. From (B.36) we can conclude that $D_{0,n-k} \leq (1+\gamma_{3k}) \cdot \widetilde{p}(s)$ and from (B.31) that $L_{1,n-k} \leq \gamma_3 \left(1+\gamma_{3(k-1)}\right) \cdot \widetilde{p}(s)$.

To show the bounds for higher values of F, we'll assume we have bounds of the form $D_{F,n-k} \leq \left(q_F(k)\mathbf{u}^F + \mathcal{O}\left(\mathbf{u}^{F+1}\right)\right) \cdot \widetilde{p}(s)$ and $L_{F,n-k} \leq \left(r_F(k)\mathbf{u}^F + \mathcal{O}\left(\mathbf{u}^{F+1}\right)\right) \cdot \widetilde{p}(s)$ for two families of polynomials $q_F(k), r_F(k)$. We have $q_0(k) = 1$ and $r_1(k) = 3$ as our base cases and

can build from there. To satisfy (B.37), we'd like $q_F(k) = q_F(k-1) + r_F(k)$ and for (B.32) $r_{F+1}(k) = 3q_F(k-1) + 5Fr_F(k)$. Since the forward difference $\Delta q_F(k) = r_F(k+1)$ is known, we can inductively solve for q_F in terms of $q_F(0)$. But $D_{F,n} = 0$ gives $q_F(0) = 0$.

For example, since we have $r_1(k) = 3\binom{k}{0}$ we'll have $q_1(k) = 3\binom{k}{1}$. Once this is known

$$r_2(k) = 3q_1(k-1) + 5r_1(k) = 3 \cdot 3\binom{k-1}{1} + 5 \cdot 3\binom{k}{0} = 9\binom{k}{1} + 6\binom{k}{0}. \tag{B.38}$$

If we write these polynomials in the "falling factorial" basis of forward differences, then we can show that

$$r_F(k) = 3^F\binom{k}{F} + \cdots \tag{B.39}$$

which will complete the proof of the first inequality. To see this, first note that for a polynomial in this basis $f(k) = A\binom{k}{d} + B\binom{k}{d-1} + C\binom{k}{d-2} + D\binom{k}{d-3} + \cdots$ we have

$$f(k+1) = A\binom{k}{d} + (A+B)\binom{k}{d-1} + (B+C)\binom{k}{d-2} + (C+D)\binom{k}{d-3} + \cdots \tag{B.40}$$

$$f(k-1) = A\binom{k}{d} + (B-A)\binom{k}{d-1} + (C-B+A)\binom{k}{d-2} + (D-C+B-A)\binom{k}{d-3} + \cdots \tag{B.41}$$

Using these, we can show that if $r_F(k) = \sum_{j=0}^{F-1} c_j\binom{k}{j}$ then

$$q_F(k) = c_{F-1}\binom{k}{F} + \sum_{j=1}^{F-1}(c_j + c_{j-1})\binom{k}{j} \tag{B.42}$$

$$r_{F+1}(k) = 3\left[-c_0\binom{k}{0} + \sum_{j=1}^{F} c_{j-1}\binom{k}{j}\right] + 5F\left[\sum_{j=0}^{F-1} c_j\binom{k}{j}\right] = 3c_{F-1}\binom{k}{F} + \cdots \tag{B.43}$$

Under the inductive hypothesis $c_{F-1} = 3^F$ so that the lead term in $r_{F+1}(k)$ is $3c_{F-1}\binom{k}{F} = 3^{F+1}\binom{k}{F}$.

For the second inequality, we'll show that

$$\sum_{k=0}^{n-1} \gamma_{3k+5F} L_{F,k} \leq \left[q_{F+1}(n)\mathbf{u}^{F+1} + \mathcal{O}\left(\mathbf{u}^{F+2}\right)\right] \cdot \widetilde{p}(s) \tag{B.44}$$

and then we'll have our result since we showed above that $q_{F+1}(n) = 3^{F+1}\binom{n}{F+1} + \mathcal{O}\left(n^F\right)$. Since $\gamma_{3k+5F} L_{F,k} \leq (3k+5F)L_{F,k}\mathbf{u} + \mathcal{O}\left(\mathbf{u}^{F+2}\right)\widetilde{p}(s)$ it's enough to consider

$$\sum_{k=0}^{n-1}(3k+5F)r_F(n-k) = \sum_{k=1}^{n}(3(n-k)+5F)r_F(k). \tag{B.45}$$

Since $q_F(k) = q_F(k-1) + r_F(k)$ and $q_F(0) = 0$ we have $q_F(n) = \sum_{k=1}^{n} r_F(k)$ thus

$$q_{F+1}(n) = \sum_{k=1}^{n} r_{F+1}(k) = \sum_{k=1}^{n} 3q_F(k-1) + 5Fr_F(k) = \sum_{k=1}^{n} 3\left[\sum_{j=1}^{k-1} r_F(j)\right] + 5Fr_F(k). \quad \text{(B.46)}$$

Swapping the order of summation and grouping like terms, we have our result. ■

Proof of Lemma 5.4. As in (2.7), we can express the compensated de Casteljau algorithm as

$$\partial^F b^{(k)} = U_{k+1}\partial^F b^{(k+1)} + \ell_F^{(k)} \Longrightarrow \partial^F b^{(0)} = \sum_{k=0}^{n-1} U_1 \cdots U_k \ell_F^{(k)} = \sum_{k=0}^{n-1}\left[\sum_{j=0}^{k} \ell_{F,j}^{(k)} B_{j,k}(s)\right]. \quad \text{(B.47)}$$

For the inexact equivalent of these things, first note that $\widehat{r} = (1-s)(1+\delta)$. Due to this, we put the $\widehat{r}$ term at the end of each update step to reduce the amount of round-off:

$$\widehat{\partial^F b}_j^{(k)} = \widehat{\ell}_{F,j}^{(k)} \oplus \left(s \otimes \widehat{\partial^F b}_{j+1}^{(k+1)}\right) \oplus \left(\widehat{r} \otimes \widehat{\partial^F b}_j^{(k+1)}\right) \quad \text{(B.48)}$$

$$= (1-s)\cdot \widehat{\partial^F b}_j^{(k+1)}(1+\theta_3) + s \cdot \widehat{\partial^F b}_{j+1}^{(k+1)}(1+\theta_3) + \widehat{\ell}_{F,j}^{(k)}(1+\theta_2) \quad \text{(B.49)}$$

$$\Longrightarrow \widehat{\partial^F b}^{(k)} = U_{k+1}\widehat{\partial^F b}^{(k+1)}(1+\theta_3) + \widehat{\ell}_F^{(k)}(1+\theta_2) \quad \text{(B.50)}$$

$$\Longrightarrow \widehat{\partial^F b}^{(0)} = \sum_{k=0}^{n-1} U_1 \cdots U_k \widehat{\ell}_F^{(k)}(1+\theta_{3k+2}) = \sum_{k=0}^{n-1}\left[\sum_{j=0}^{k} \widehat{\ell}_{F,j}^{(k)}(1+\theta_{3k+2}) B_{j,k}(s)\right]. \quad \text{(B.51)}$$

Since

$$\partial^{F+1} b_0^{(0)} = \partial^F b_0^{(0)} - \widehat{\partial^F b}_0^{(0)} = \sum_{k=0}^{n-1}\sum_{j=0}^{k}\left(\ell_{F,j}^{(k)} - \widehat{\ell}_{F,j}^{(k)}(1+\theta_{3k+2})\right) B_{j,k}(s) \quad \text{(B.52)}$$

it's useful to put a bound on $\ell_{F,j}^{(k)} - \widehat{\ell}_{F,j}^{(k)}(1+\theta_{3k+2})$. Via

$$\widehat{\ell}_{F,j}^{(k)} = e_1 \oplus \cdots \oplus e_{5F-2} \oplus \left(\rho \otimes \widehat{\partial^{F-1} b}_j^{(k+1)}\right) \quad \text{(B.53)}$$

$$= e_1(1+\theta_{5F-2}) + \cdots + e_{5F-2}(1+\theta_2) + \rho \cdot \widehat{\partial^{F-1} b}_j^{(k+1)}(1+\theta_2) \quad \text{(B.54)}$$

we see that

$$\left|\ell_{F,j}^{(k)} - \widehat{\ell}_{F,j}^{(k)}(1+\theta_{3k+2})\right| \leq \gamma_{3k+5F} \cdot \widetilde{\ell}_{F,j}^{(k)} \Longrightarrow \left|\partial^{F+1} b_0^{(0)}\right| \leq \sum_{k=0}^{n-1} \gamma_{3k+5F} \sum_{j=0}^{k} \widetilde{\ell}_{F,j}^{(k)} B_{j,k}(s). \quad \text{(B.55)}$$

Applying (5.29) directly gives

$$\left|\partial^{F+1} b_0^{(0)}\right| \leq \left[\left(3^{F+1}\binom{n}{F+1} + \mathcal{O}\left(n^F\right)\right)\mathbf{u}^{F+1} + \mathcal{O}\left(\mathbf{u}^{F+2}\right)\right] \cdot \widetilde{p}(s). \quad \text{(B.56)}$$

Letting $K = F+1$ we have our result. ■

Proof of Theorem 5.2. Since

$$\texttt{CompDeCasteljau}(p,s,K) = \texttt{SumK}\left(\left[\widehat{b}_0^{(0)},\ldots,\widehat{\partial^{K-1}b_0}^{(0)}\right],K\right), \tag{B.57}$$

applying Theorem 5.1 tells us that

$$\left|\texttt{CompDeCasteljau}(p,s,K) - \sum_{F=0}^{K-1}\widehat{\partial^F b_0}^{(0)}\right| \leq \left(\mathbf{u}+3\gamma_{n-1}^2\right)\left|\sum_{F=0}^{K-1}\widehat{\partial^F b_0}^{(0)}\right| + \gamma_{2n-2}^K\sum_{F=0}^{K-1}\left|\widehat{\partial^F b_0}^{(0)}\right|. \tag{B.58}$$

Since

$$p(s) = b_0^{(0)} = \widehat{b}_0^{(0)} + \partial b_0^{(0)} = \cdots = \widehat{b}_0^{(0)} + \widehat{\partial} b_0^{(0)} + \cdots + \widehat{\partial^{K-1}b_0}^{(0)} + \partial^K b_0^{(0)} \tag{B.59}$$

we have

$$\left|\sum_{F=0}^{K-1}\widehat{\partial^F b_0}^{(0)}\right| \leq |p(s)| + \left|\partial^K b_0^{(0)}\right| \quad \text{and} \tag{B.60}$$

$$|\texttt{CompDeCasteljau}(p,s,K) - p(s)| \leq \left|\texttt{CompDeCasteljau}(p,s,K) - \sum_{F=0}^{K-1}\widehat{\partial^F b_0}^{(0)}\right| + \left|\partial^K b_0^{(0)}\right|. \tag{B.61}$$

Due to Lemma 5.4, $\partial^F b_0^{(0)} = \mathcal{O}\left(\mathbf{u}^F\right)\widetilde{p}(s)$, hence

$$\left(\mathbf{u}+3\gamma_{n-1}^2\right)\left|\sum_{F=0}^{K-1}\widehat{\partial^F b_0}^{(0)}\right| \leq \left[\mathbf{u}+\mathcal{O}\left(\mathbf{u}^2\right)\right]|p(s)| + \mathcal{O}\left(\mathbf{u}^{K+1}\right)\widetilde{p}(s) \tag{B.62}$$

$$\gamma_{2n-2}^K\sum_{F=0}^{K-1}\left|\widehat{\partial^F b_0}^{(0)}\right| \leq \gamma_{2n-2}^K\left|\widehat{b}_0^{(0)}\right| + \mathcal{O}\left(\mathbf{u}^{K+1}\right)\widetilde{p}(s) \tag{B.63}$$

$$\leq \gamma_{2n-2}^K\left[|p(s)| + \mathcal{O}\left(\mathbf{u}\right)\widetilde{p}(s)\right] + \mathcal{O}\left(\mathbf{u}^{K+1}\right)\widetilde{p}(s). \tag{B.64}$$

Combining this with (B.58) and (B.61), we see

$$|\texttt{CompDeCasteljau}(p,s,K) - p(s)| \tag{B.65}$$

$$\leq \left[\mathbf{u}+\mathcal{O}\left(\mathbf{u}^2\right)\right]|p(s)| + \left|\partial^K b_0^{(0)}\right| + \mathcal{O}\left(\mathbf{u}^{K+1}\right)\widetilde{p}(s) \tag{B.66}$$

$$\leq \left[\mathbf{u}+\mathcal{O}\left(\mathbf{u}^2\right)\right]|p(s)| + \left[\left(3^K\binom{n}{K}+\mathcal{O}\left(n^{K-1}\right)\right)\mathbf{u}^K + \mathcal{O}\left(\mathbf{u}^{K+1}\right)\right]\widetilde{p}(s). \tag{B.67}$$

Dividing this by $|p(s)|$, we have our result. ■

Lemma B.1. Consider three smooth curves b_0, b_1, b_2 that form a closed loop: $b_0(1) = b_1(0)$, $b_1(1) = b_2(0)$ and $b_2(1) = b_0(0)$. Take **any** smooth map $\varphi(s,t)$ on $\mathcal{U}$ that sends the edges to the three curves:

$$\varphi(r,0) = b_0(r), \quad \varphi(1-r,r) = b_1(r), \quad \varphi(0,1-r) = b_2(r) \quad \text{for } r \in [0,1]\,. \tag{B.68}$$

Then we must have

$$2\int_{\mathcal{U}} \det(D\varphi)\,[F\circ\varphi]\;dt\,ds = \oint_{b_0\cup b_1\cup b_2} H\,dy - V\,dx \tag{B.69}$$

for antiderivatives that satisfy $H_x = V_y = F$.

When $\det(D\varphi) > 0$, this is just the change of variables formula combined with Green's theorem.

Proof. Let $x(s,t)$ and $y(s,t)$ be the components of φ. Define

$$\Delta S = H(x,y)y_s - V(x,y)x_s \quad \text{and} \quad \Delta T = H(x,y)y_t - V(x,y)x_t. \tag{B.70}$$

On the unit triangle $\mathcal{U}$, Green's theorem gives

$$\int_{\mathcal{U}} [\partial_s \Delta T - \partial_t \Delta S]\;dV = \oint_{\partial\mathcal{U}} \Delta S\,ds + \Delta T\,dt. \tag{B.71}$$

The boundary $\partial\mathcal{U}$ splits into the bottom edge E_0, hypotenuse E_1 and left edge E_2.

Since

$$E_0 = \left\{ \begin{bmatrix} r \\ 0 \end{bmatrix} \mid r \in [0,1] \right\} \tag{B.72}$$

we take $\varphi(r,0) = b_0(r)$ hence

$$dx = x_s\,dr, dy = y_s\,dr \Longrightarrow H dx - V dy = \Delta S\,dr. \tag{B.73}$$

We also have $ds = dr$ and $dt = 0$ due to the parameterization, thus

$$\int_{E_0} \Delta S\,ds + \Delta T\,dt = \int_{r=0}^{r=1} \Delta S\,dr = \int_{b_0} H\,dx - V\,dy. \tag{B.74}$$

We can similarly verify that $\int_{E_j} \Delta S\,ds + \Delta T\,dt = \int_{b_j} H\,dx - V\,dy$ for the other two edges. Combining this with (B.71) we have

$$\int_{\mathcal{U}} [\partial_s \Delta T - \partial_t \Delta S]\;dV = \oint_{b_0\cup b_1\cup b_2} H\,dx - V\,dy. \tag{B.75}$$

To complete the proof, we need

$$\int_{\mathcal{U}} [\partial_s \Delta T - \partial_t \Delta S]\;dV = 2\int_{\mathcal{U}} \det(D\varphi)\,[F\circ\varphi]\;dV \tag{B.76}$$

but one can show directly that

$$\partial_s \Delta T - \partial_t \Delta S = 2\,(x_s y_t - x_t y_s)\,F(x,y) = 2\det(D\varphi)\,[F\circ\varphi]\,. \qquad \blacksquare$$