# Numerical stability of barycentric interpolation

Doctoral Dissertation submitted to the
Faculty of Informatics of the *Università della Svizzera italiana*
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
**Chiara Fuda**

under the supervision of
Prof. Kai Hormann

December 2024

## Dissertation Committee

| | |
|---|---|
| **Prof. Kai Hormann** | Università della Svizzera italiana, Switzerland |
| **Prof. Michael Multerer** | Università della Svizzera italiana, Switzerland |
| **Prof. Stefan Wolf** | Università della Svizzera italiana, Switzerland |
| | |
| **Prof. Annie Cuyt** | University of Antwerp, Belgium |
| **Prof. Stefano De Marchi** | Università degli Studi di Padova, Italy |

Dissertation accepted on 20 December 2024

**Prof. Kai Hormann**
Research Advisor
Università della Svizzera italiana, Switzerland

**The PhD program Director**
Prof. Walter Binder / Prof. Stefan Wolf

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Chiara Fuda
Lugano, 20 December 2024

"In mathematics, precision is key; even the slightest error can lead to significant consequences."

Leonhard Euler (1707–1783)

# Acknowledgements

My PhD journey is now coming to an end. It feels just like yesterday I arrived in Lugano as a fresh master's student, and now here I am, four years later as an almost PhD.

It has been an amazing experience, and for this, I cannot thank my supervisor, Prof. Kai Hormann, enough. His guidance and support have been crucial throughout my PhD. He has always been available for discussions and eager to share new insights to help me grow professionally. Thank you for the meticulous care and attention to every detail, I hope I have gathered as much as possible from your mentorship.

I really would like to thank all the members of the committee, Prof. Michael Multerer, Prof. Stefan Wolf, Prof. Annie Cuyt, and Prof. Stefano De Marchi, for agreeing to be part of it and for having evaluated this thesis.

Once again, I would like to express my gratitude to my Master's thesis supervisor, Prof. Francesco Dell'Accio. He has always demonstrated his support and trust in me, even from afar.

This experience would not have been the same without my incredible office mates and colleagues from the $4^{\text{th}}$ floor (and beyond). Whether you have been with me from the beginning or have only shared a small part of these four years, each of you has left me with special memories. Thank you for all the lunches, coffees, BBQs, and hikes.

L'USI mi ha dato tanto, non solo a livello professionale, ma anche personale. Mi ha riavvicinato alla mia piú grande passione, il ballo, e grazie ad essa ho conosciuto tante persone nuove. Oggi ho la fortuna di poter considerare alcune di loro delle vere amiche, e non posso che ringraziarle per tutte le avventure, le serate e le chiacchierate. Grazie Sam e Giuli, avete reso tutto molto piú memorabile!

Grazie alla mia amica Hele per le nostre interminabili chiacchierate al telefono. Nonostante la distanza, sei sempre vicina con il cuore.

Grazie a te, Lore, per essermi stato vicino in ogni momento da quando ci siamo conosciuti. Non sono veramente sicura che tu abbia sempre attivamente ascoltato i miei discorsi matematici, ma in qualche modo hai sempre avuto la parola giusta al momento giusto, in ogni circostanza. Sei stato, e sei tutt'ora, il mio valore aggiunto.

Infine, non per importanza ovviamente, un ringraziamento speciale va alla mia famiglia. Credo non sia stato facile vedermi andare sempre piú lontano negli anni, ma, nonostante ció, avete sempre supportato le mie scelte, mettendo me e la mia felicitá al primo posto. Grazie di cuore, niente di tutto questo sarebbe stato possibile senza di voi!

# Abstract

In the field of Numerical Analysis, it is common practice to solve an arbitrary mathematical *problem* through the implementation of a numerical *algorithm* on a computer. Since not all data of a problem can be represented exactly on a computer as floating-point numbers, it can happen that the algorithm starts with rounding errors already from the initial set of input that then propagate throughout the process. Consequently, it is important to study the *numerical stability* of an algorithm, that is, its behaviour with respect to the propagation of the errors that occur during the arithmetic operations executed by the computer. In this dissertation we focus on the *barycentric interpolation* problem, both in the univariate and the bivariate setting. In the first case, we theoretically discuss the numerical stability of all algorithms that implement a *barycentric rational interpolant*, providing conditions under which it is possible to know a priori whether they are stable. In the second case, we focus on the barycentric interpolant defined on a planar polygon, and this leads to the study of the numerical stability of generalized barycentric coordinates, particularly the *mean value coordinates*.

In the first part, we analyse the numerical stability of the algorithms that compute univariate barycentric rational interpolants. We begin by showing more generally that the evaluation of any function that can be expressed as $r(x) = \sum_{i=0}^{n} a_i(x) f_i \big/ \sum_{j=0}^{m} b_j(x)$ in terms of data values $f_i$ and some functions $a_i$ and $b_j$ for $i = 0, \ldots, n$ and $j = 0, \ldots, m$ is forward and backward stable under certain assumptions. The proof considers the simplest and classical algorithm that involves summing the terms in the numerator and denominator first, followed by a final division. This result includes the two barycentric forms of rational interpolation as special cases. Our analysis further reveals that the stability of the second barycentric form depends on the Lebesgue constant associated with the interpolation nodes, which typically grows with $n$, whereas the stability of the first barycentric form depends on a similar, but different quantity, that can be bounded in terms of the mesh ratio, regardless of $n$. We support our theoretical results with numerical experiments.

These findings contribute to the development of a new `C++` class, named BRI (Barycentric Rational Interpolation), which contains all variables and functions related to linear barycentric rational interpolation. This class is designed to autonomously select the best method to use on a case-by-case basis, as it takes into account our theoretical results regarding the efficiency and numerical stability of barycentric rational interpolation. Moreover, we describe a new technique that makes the code robust and less prone to overflow and underflow errors. In addition to the standard `C++` data types, the BRI template variables can also be defined with arbitrary precision, because the BRI class is compatible with the Multiple Precision Floating-Point Reliable (MPFR) library.

The second part of the thesis focuses on the numerical stability of algorithms that compute generalized barycentric coordinates on polygons with more than three vertices. Among the

different constructions proposed, mean value coordinates have emerged as a popular choice, particularly due to their suitability for the non-convex setting. Since their introduction, they have found applications in numerous fields, and several equivalent formulas for their evaluation have been presented in the literature. However, so far, there has been no study regarding their numerical stability. We show that all the known methods exhibit instability in some regions of the domain. To address this problem, we introduce a new formula for computing mean value coordinates, explain how to implement it, and formally prove that our new algorithm provides a stable evaluation of mean value coordinates. We finally validate our results through numerical experiments.

Lastly, since the results of the first part can be applied to a broad range of numerical methods, we examine several algorithms used for evaluating Bézier curves. The de Casteljau algorithm is the first method introduced for evaluating polynomial Bézier curves, later also generalized to the rational case and surfaces. Although it presents an elegant definition through convex combinations and generally yields stable results, it has quadratic time complexity. This represents a significant limitation, especially when dealing with high-degree curves and real-time applications. For this reason, numerous studies have been conducted in order to provide alternative approaches and more efficient algorithms. We present a collection of the most commonly used algorithm in the state-of-the-art and provide a comparison of their numerical stability. Notably, although some error analyses exist only for some specific algorithm, the literature lacks an in-depth investigation into the numerical stability of all these methods, along with a corresponding comparison. Therefore, this represents the first comprehensive study of its kind.

# Contents

# Figures

# Tables

# Chapter 1

# Introduction

## 1.1 Motivation

*Numerical stability* is a favorable property of numerical algorithms that concerns how they behave with respect to the propagation of errors that occur during their execution. In general, whenever we want to solve a problem with a numerical algorithm on a computer, we must be aware of the fact that we cannot obtain its exact solution because of the errors introduced by all the arithmetic operations performed. However, the study of the numerical stability provides us with the tools to measure how far the result produced by the algorithm is from the exact answer to the problem. This analysis is of great importance when several algorithms exist to solve the same problem, because we can know a priori which one gives the most reliable answer according to the data set.

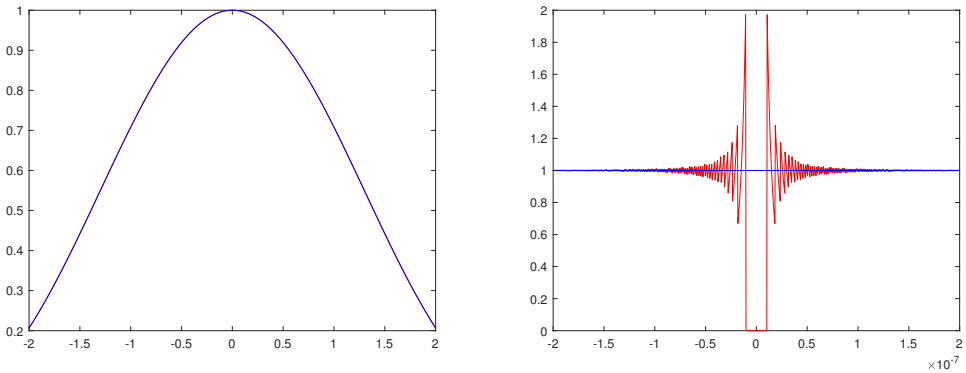


Figure 1.1. Plot of the functions $f(x) = (1-\cos(x)^2)/x^2$ (red) and $g(x) = \sin(x)^2/x^2$ (blue) for $x \in [-2, 2]$ (left) and $x \in [-2 \times 10^{-7}, 2 \times 10^{-7}]$ (right).

To understand the importance of studying the numerical stability of algorithms, we present a concrete example by considering the two mathematically equivalent functions $f(x) = (1-\cos(x)^2)/x^2$ and $g(x) = \sin(x)^2/x^2$. We compute and plot both functions for 1000 equidistant points in the interval $[-2, 2]$ on a machine that uses finite-precision arithmetic, and, as expected, observe from Figure 1.1 that they overlap and approach 1 as $x \to 0$ (left). How-

ever, going very close to $x = 0$ (right) and repeating the same procedure for the interval $[-2 \times 10^{-7}, 2 \times 10^{-7}]$, we can see that they behave completely differently. In fact, $g(x)$ gives the expected result, while $f(x)$ does not, because it is affected by rounding errors. This demonstrates that theoretically equivalent formulas can give rise to different algorithms in terms of numerical stability. Accordingly, if a problem can be solved with multiple algorithms, then it is very important to study their numerical stability to determine a priori which is the most suitable choice for each specific scenario.

The problem covered in this thesis is *barycentric interpolation*. Given a set of $n+1$ points $X_n = (x_0, \dots, x_n)$ in $\mathbb{R}^m$ with some associated weights $w_0, \dots, w_n \in \mathbb{R}$, the barycenter of this system is the unique point $x \in \mathbb{R}^m$ that satisfies

$$\sum_{i=0}^{n} w_i(x - x_i) = 0,$$

or, equivalently,

$$x = \frac{\sum_{i=0}^{n} w_i x_i}{\sum_{i=0}^{n} w_i}. \tag{1.1}$$

Barycentric interpolation approaches this problem from another point of view, that is, given a fixed set of *nodes* $X_n \in \mathbb{R}^{m\times(n+1)}$ and an arbitrary point $x \in \mathbb{R}^m$, find the weights $w_0, \dots, w_n \in \mathbb{R}$ such that (1.1) holds. These weights are called *barycentric coordinates* of $x$ with respect to $x_0, \dots, x_n$ and Möbius [64] showed that they always exist as long as $n \geq m$. Moreover, the barycentric coordinates $w_i$ are *homogeneous*, meaning that they can be multiplied by a non-zero constant and (1.1) is still true. Once we have these weights, we can focus our attention on the *normalized* barycentric coordinates, which are the functions $\phi_i \colon \mathbb{R}^m \to \mathbb{R}$ defined as

$$\phi_i(x) = \frac{w_i(x)}{\sum_{j=0}^{n} w_j(x)}, \qquad i = 0, \dots, n. \tag{1.2}$$

Because of (1.1), the functions $\phi_i$, $i = 0, \dots, n$, satisfy the *barycentric property*,

$$\sum_{i=0}^{n} \phi_i(x) x_i = x, \tag{1.3}$$

and they form a *partition of unity*,

$$\sum_{i=0}^{n} \phi_i(x) = 1. \tag{1.4}$$

An additional requirement for barycentric coordinates is that the functions $\phi_i$ should satisfy the *Lagrange property*

$$\phi_i(x_j) = \delta_{ij}, \qquad i, j = 0, \dots, n, \tag{1.5}$$

meaning that they are 1 at the corresponding node and 0 at all other nodes. Because of these three properties, the normalized barycentric coordinates can also be seen as a set of *barycentric basis functions* and Möbius [64] showed that they are unique when $n + 1 = m$ and linear. If we now associate to the nodes $X_n$ some *data* $Y_n = (y_0, y_1, \dots, y_n)$ in $\mathbb{R}$, then its *barycentric interpolant* is defined as

$$r(x) = \sum_{i=0}^{n} \phi_i(x) y_i. \tag{1.6}$$

It follows from (1.5) that $r$ interpolates $y_i$ at $x_i$, $i = 0, \dots, n$, and from (1.3) and (1.4) that this kind of interpolation is exact for linear functions, that is, if $y_i = f(x_i)$ and $f$ is a linear polynomial. We focus both on the univariate setting, where $x_0, \dots, x_n \in \mathbb{R}$, and on the bivariate setting, in particular for the special case where the nodes are the vertices of a polygon in $\mathbb{R}^2$.

In the univariate case, considering the set of nodes $X_n \in \mathbb{R}^{n+1}$, one choice of barycentric basis functions is given by the *Lagrange polynomials*

$$\phi_i = \prod_{j=0,\, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}, \qquad i = 0, \dots, n.$$

If we are given the nodes in ascending order $x_0 < x_1 < \cdots < x_n$, then it is known that these functions can also be expressed as in (1.2) [8] with

$$w_i(x) = \frac{\gamma_i}{x - x_i}, \qquad i = 0, \dots, n \tag{1.7}$$

and

$$\gamma_i = \prod_{j=0,\, j\neq i}^{n} \frac{1}{x_i - x_j}, \qquad i = 0, \dots, n. \tag{1.8}$$

Furthermore, expressing the barycentric coordinates as in (1.7), we can find other sets of barycentric basis functions, such as the rational ones, by properly defining the non-zero weights $\gamma_0, \dots, \gamma_n$. Consequently, associating to $X_n$ some data $Y_n$, we mainly focus on the *barycentric rational interpolant* $r \colon \mathbb{R} \to \mathbb{R}$ expressed in *second barycentric form* [74] as

$$r(x) = \sum_{i=0}^{n} \frac{\frac{\gamma_i}{x - x_i}}{\sum_{i=0}^{n} \frac{\gamma_i}{x - x_i}} y_i. \tag{1.9}$$

We consider *linear* barycentric rational interpolation, meaning that the denominator does not depend on the data $Y_n$. The simplest expression for the weights $\gamma_i$ is surely the one given by Berrut [5] as

$$\gamma_i = (-1)^i, \qquad i = 0, 1, \dots, n, \tag{1.10}$$

for which he proved that the resulting rational interpolant is free of poles. Note that, if we choose the weights as in (1.10), then the functions $\phi_i$ in (1.2) satisfy the property (1.3) only when $n$ is odd [46]. Afterwards, Floater and Hormann [30] proposed a new linear barycentric rational interpolant by considering a parameter $d \in \{0, 1, \dots, n\}$ and the *barycentric weights*

$$\gamma_i = \sum_{k=\max(i-d,0)}^{\min(i,n-d)} (-1)^k \prod_{j=k,\, j\neq i}^{k+d} \frac{1}{x_i - x_j}, \qquad i = 0, 1, \dots, n, \tag{1.11}$$

which again guarantee the absence of poles in the expression of the interpolant (1.6). Actually, this method is a generalization of Berrut's case since, for $d = 0$, the weights in (1.11) are exactly the ones in (1.10), but now, for $d \geq 1$, all properties (1.3)–(1.5) are satisfied. Furthermore, also polynomial interpolation is covered by this method, since the weights in (1.8) are the ones in (1.11) with $d = n$. The barycentric formula (1.9) is widely used to evaluate the interpolant $r$ as it can be implemented with an efficient $O(n)$ algorithm and the $\gamma_i$ can be rescaled by a common factor to prevent overflow and underflow errors [8]. However, there exists another mathematically equivalent formula to evaluate $r$, namely the *first barycentric form*

$$r(x) = \sum_{i=0}^{n} \frac{\frac{\gamma_i}{x - x_i}}{\sum_{i=0}^{n-d} \lambda_i(x)} y_i, \tag{1.12}$$

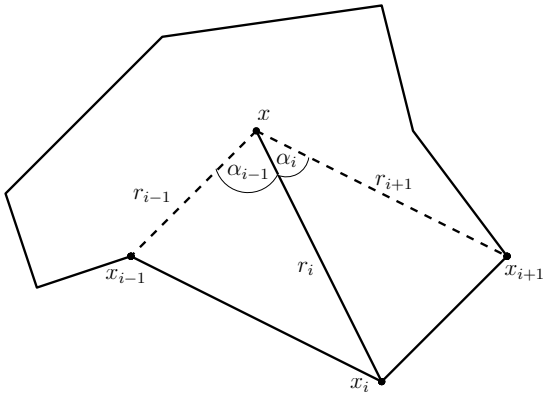


Figure 1.2. Notations for mean value coordinates.

where

$$\lambda_i(x) = \frac{(-1)^i}{(x-x_i)\cdots(x-x_{i+d})}, \qquad i = 0, 1, \ldots, n-d. \tag{1.13}$$

This formula is slightly inferior in terms of efficiency, because it requires $O(nd)$ operations to be computed straightforwardly, but there exists a more efficient way of computing its denominator in $O(n)$ operations [34]. Therefore, the first part of this thesis focuses on the study of the numerical stability of the algorithms that implement the two formulas (1.9) and (1.12), in order to know a priori if there are cases in which one turns out to be better than the other.

In the bivariate case, our concern is to study the normalized barycentric coordinates $\phi_i : P \to \mathbb{R}$ defined as in (1.2) assuming to have some nodes $x_1, \ldots, x_n$ that are the vertices of a planar polygon $P \subset \mathbb{R}^2$. Since they satisfy the barycentric property (1.3), these values allow each interior point of $P$ to be expressed as an affine combination of the vertices. In addition, they are of particular interest in many applications, such as interpolation, curve and surface modelling in computer graphics, the finite element method, and many others. In the case of a triangle, that is, for $n = 3$, these coordinates are uniquely determined by (1.3) and (1.4) and positive [64]. Regarding instead an arbitrary polygon, the functions in (1.2) are no longer uniquely defined and different constructions have been proposed during the years. For example, for convex polygons, an attractive choice is represented by the Wachspress coordinates [79], which can be expressed as rational functions and have many other nice properties, including positivity. However, they are not defined for arbitrary simple polygons and, in that case, the *mean value coordinates* [28] are the most popular option, though positivity is lost in general. Denoting by $\alpha_i(x) \in (-\pi, \pi)$ the angles at $x$ in the triangle $[x, x_{i+1}, x_i]$, $i = 1, \ldots, n-1$, and $r_i = \|x - x_i\|$, $i = 1, \ldots, n$, (see Figure 1.2) they are defined by (1.2) and

$$w_i(x) = \frac{\tan(\alpha_{i-1}/2) + \tan(\alpha_i/2)}{r_i}, \qquad i = 1, \ldots, n.$$

Since several other formulas that are mathematically equivalent to the previous one can be found in the literature, the second part of this thesis regards the study of the numerical stability of the algorithms that implement the mean value coordinates.

## 1.2 Outline of the thesis

This thesis includes all the research I conducted during my period as PhD student at *Università della Svizzera italiana*. Below, I present how it is organized throughout the dissertation.

In Chapter 2, we present the essential preliminary results needed for the study of the numerical stability of the algorithms that compute the barycentric interpolation. We begin by presenting the floating-point number system in Section 2.1, with focus on how floating-point numbers are defined and the limitations of using this discrete set to approximate the infinite set of real numbers. In Section 2.2, we introduce the concept of condition number. Although it is not directly related to the algorithm itself, it is strictly related to its numerical stability as it measures the sensitivity of the solution of a problem to small perturbations of the given data. Finally, in Section 2.3, we mathematically define the numerical stability of an algorithm, that is its behaviour with respect to the propagation of the errors that occur during the arithmetic operations executed by the computer using floating-point numbers.

In Chapter 3, we first summarize the state of the art on the forward and backward stability of barycentric rational interpolation in the univariate case in Section 3.1 and then present our contributions in Section 3.2. Afterwards, we focus on a general rational function that includes both the first and second barycentric forms of rational interpolation as special cases. In particular, we demonstrate that both are forward and backward stable under certain conditions in Section 3.3 and 3.5, respectively. First, we need forward stable algorithms to compute the weights $\gamma_i$ and the functions $\lambda_i$, which is the case of both formulas in (1.11) and (1.13), as proved in Section 3.4. Additionally, we present a new algorithm to compute the functions $\lambda_i$ that is more efficient than the existing methods, but comes at the cost of slightly reduced stability. Second, our analysis reveals that, on the one hand, the stability of the second barycentric form depends on the Lebesgue constant associated with the interpolation nodes, which typically grows with $n$ and has been widely studied over the years. On the other hand, the stability of the first barycentric form depends on a different quantity that has never been presented in the literature before. We prove that the latter can be bounded in terms of the mesh ratio, regardless of $n$, in Section 3.6. We conclude with numerical experiments that support our theoretical results in Section 3.7. This chapter is based on our published paper [34]

**Fuda, C., Campagna, R. and Hormann, K. [2022]. On the numerical stability of linear barycentric rational interpolation, Numerische Mathematik 152(4): 761–786.**

In Chapter 4, we present our `C++` class, named BRI (Barycentric Rational Interpolation), which contains all the necessary functions for handling a barycentric rational interpolant with all its features through a *robust* and *efficient* implementation. Furthermore, the BRI class supports not only the standard `C++` data types, but also allows for arbitrary precision using the Multiple Precision Floating-Point Reliable (MPFR) library [33]. While there already exist several libraries [1, 11, 62, 77] that evaluate a barycentric rational interpolant using the second form in (1.9) without any further considerations, the BRI class is designed to autonomously select the best method to use on a case-by-case basis. It does so by taking into account the results regarding the efficiency and numerical stability of barycentric rational interpolation presented in Chapter 3. Moreover, we describe a new technique that makes the code robust and less prone to overflow and underflow errors. Therefore, after a brief overview of all the functions contained in the BRI class in Section 4.1, we present some preliminary results needed to handle overflow and underflow errors in Section 4.2. Then, in Section 4.3 and 4.4, we discuss the robust implementations of the barycentric weights $\gamma_i$ and of the interpolant $r$. Finally, we discuss

the algorithms used to compute the functions related to numerical stability in Section 4.5. This part of the thesis is based on our published paper [35]

**Fuda, C. and Hormann, K. [2024]. Algorithm 1048: A C++ Class for Robust Linear Barycentric Rational Interpolation, ACM Transaction on Mathematical Software 50(3).**

Chapter 5 investigates the numerical stability of the algorithms that implement the mean value coordinates with a similar approach to that used in Chapter 3. In particular, we begin by presenting all the formulas that has been introduced in the literature to compute mean value coordinates in Section 5.1 and our contribution regarding their numerical stability in Section 5.2. Afterwards, Section 5.3 shows empirically that all these methods can exhibit numerical instability in certain situations. Therefore, to address this issue, we introduce in Section 5.4 a new formula for expressing the mean value coordinates and explain how to properly implement it, so as to prevent potential numerical issues. Then, we adapt the mathematical definition of numerical stability to the case of mean value coordinates in Section 5.5, and we prove in Section 5.6 that our new formula provides a stable way to compute them, while all the others can be unstable under certain circumstances. Finally, we validate our results with numerical experiments and compare the various methods both in terms of numerical stability and efficiency in Section 5.7. This part of the thesis is based on our published paper [36]

**Fuda, C. and Hormann, K. [2024]. A new stable method to compute mean value coordinates, Computer Aided Geometric Design 111: Article 102310, 16 pages. Proceedings of GMP.**

Chapter 6 is an application of the results presented in Chapter 3 to a different context. In fact, these findings have broader implications that go beyond barycentric rational interpolation and can be applied to a variety of computational methods. Specifically, we examine the numerical stability of the most commonly used algorithms for computing rational Bézier curves. Therefore, we first present all these methods in Section 6.1, also accurately describing how we implement them. Afterward, we derive an upper bound on their relative errors in Section 6.2. For most algorithms, these bounds are a direct consequence of a theorem presented in Chapter 3, while, in some cases, new derivations are required. Finally, we present some numerical experiments to support our results in Section 6.3. This chapter is based on our published paper [37]

**Fuda, C., Ramanantoanina, A. and Hormann, K. [2024]. A comprehensive comparison of algorithms for evaluating rational Bézier curves, Dolomites Research Notes on Approximation, 17(3): 56-79.**

Finally, Chapter 7 summarizes the thesis by focusing on the key findings and contributions of the previous chapters and outlining potential directions for future research.

# Chapter 2

# Preliminaries

Numerical stability is a property of numerical algorithms that indicates how numerical errors present in the initial data (*input*) propagate during the execution of all the arithmetic operations leading to the final result (*output*). In general, perturbations in the data can arise from any source, but we only focus on the errors introduced by the floating-point number system. Moreover, this property is totally independent of the problem that the algorithm wants to solve, but is closely related to its *conditioning*, which estimates how a small variation in the data can affect the final solution.

Below, we first introduce the floating-point numbers and their arithmetic, and we then define the conditioning of a general problem. Finally, we discuss how to evaluate the performance of a numerical algorithm by studying its *forward* and *backward stability*.

## 2.1 Floating-point number system

The set $\mathbb{F} \subset \mathbb{R}$ of *floating-point numbers* is the standard discrete set used to approximate real numbers on a computer. Considering a fixed *base* $\beta \in \mathbb{N}$ with $\beta \geq 2$ and a *precision* $t \in \mathbb{N}$ with $t \geq 1$, the set $\mathbb{F}$ is composed by the number 0 together with all numbers $y$ that can be expressed as [43, 65, 69, 78]

$$y = \pm\mu \times \beta^E, \tag{2.1}$$

where the *mantissa* (or *significand*) $\mu \in \mathbb{N}$ lies in the range $[0, 1-\beta^{-t}]$ and the *exponent* $E$ is an arbitrary integer. The latter can obviously vary only in a finite interval of $\mathbb{Z}$, so we denote the two *extreme exponents* by $E_{\min}$ and $E_{\max}$ and generally $E_{\min} < 0 < E_{\max}$. For the purpose of obtaining a unique representation of any $y \in \mathbb{F}$ such that $y \neq 0$, we can further suppose $\mu \geq \beta^{-1}$ and, in this case, we say that the related $y$ is a *normal* number.

In our analysis, we consider a *binary* floating-point number system [78], which is characterized by the choice of $\beta = 2$. In such a system, every element of $\mathbb{F}$ has a mantissa $\mu \in [\mu_{\min}, \mu_{\max}] = [2^{-1}, 1-2^{-t}]$ and an exponent $E \in \{E_{\min}, E_{\min}+1, \dots, E_{\max}\}$, and the smallest and largest positive *normal* floating-point number are

$$F_{\min} = \mu_{\min} 2^{E_{\min}} \qquad \text{and} \qquad F_{\max} = \mu_{\max} 2^{E_{\max}}. \tag{2.2}$$

Therefore, the floating-point number system is uniquely characterized by the triple ($t$, $E_{\min}$, $E_{\max}$) and the free choice of this values may bring to the creation of many different sets. For

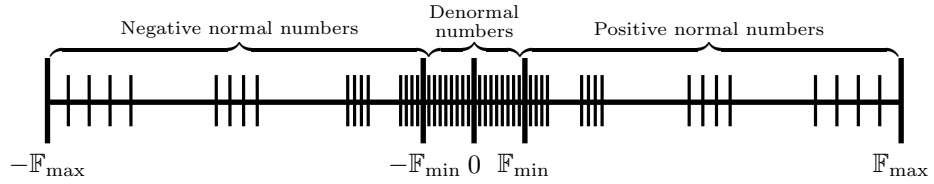


Figure 2.1. Density of floating-point numbers.

example, the standard `float` type represents a *single precision* floating-point number with $t = 23$, $E_{\min} = -125$, $E_{\max} = 128$, while the standard `double` type represents a *double precision* floating-point number with $t = 52$, $E_{\min} = -1021$, $E_{\max} = 1024$. Note that the IEEE standard 754 [52] specifies these numbers slightly differently in terms of the numbers of digits in the significand $p = t + 1$, the maximum exponent $emax = E_{\max} - 1$, and the minimum exponent $emin = 1 - emax = E_{\min} - 1$. It is still possible to extend $\mathbb{F}$ by adding the *subnormal* or *denormal* numbers in $(-F_{\min}, F_{\min})$, which are defined as $\pm\mu \times 2^{E_{\min}}$ with $0 < \mu < 2^{t-1}$ and have reduced precision.

The set $\mathbb{F}$ is not equally spaced, but it becomes denser toward zero, as shown in Figure 2.1. In particular, the floating-point numbers are uniformly distributed in each interval $\left[2^E, 2^{E+1}\right]$ and at a distance of $h_E = 2^{E-t}$. This means that, if we allow the presence of subnormal numbers in $\mathbb{F}$, then they are equidistant as they all have the same exponent. Consequently, we introduce a different error for any $x \in \mathbb{R}$ that we approximate by $\mathrm{fl}(x) \in \mathbb{F}$, where $\mathrm{fl}(x)$ is the closest floating-point number to $x$. In this regard, it was shown (see [43, Theorem 2.2], for example) that, for any $x \in \mathbb{R}$, $x \neq 0$, the relative error $(\mathrm{fl}(x) - x)/x$ is bounded from above by the number $\epsilon = 2^{-t}$, known as the *machine epsilon* or *unit roundoff*. This is equivalent to saying that we can always find some $\delta \in \mathbb{R}$ with $|\delta| < \epsilon$ such that

$$\mathrm{fl}(x) = x(1 + \delta). \tag{2.3}$$

Thus, whenever we use real data as input for an algorithm, they are inevitably perturbed and introduce rounding errors into the process that propagate through to the final result.

### 2.1.1 Overflow and underflow

Before delving into the details of overflow and underflow error, let us begin with a toy example to show this situation. We consider $x = 10, 20, 40, 80, 160, 320, 640, 1280$ and compute $f(x) = e^x$ on a machine that employs the double precision floating-point number system. Both the `C++` code and the related output are shown in Program 1. We note that, up to $x = 640$, the code behaves as expected, while for $x = 1280$ the exponential function gives `inf`. This occurs because the true result should be $\approx 7.8875 \times 10^{555}$, which exceeds the maximum representable floating-point number in double precision, that is $F_{\max} = 1.79769313486232 \times 10^{308}$. Consequently, $e^{1280}$ cannot be represented and the code gives an overflow error.

In general, all $x \in \mathbb{R}$ that are not elements of the set $\mathbb{F}$ cannot be represented as floating-point numbers and, in this case, we talk about *overflow* if $|x| > F_{\max}$ and *underflow* if $|x| < F_{\min}$ (or *gradual underflow* if $\mathbb{F}$ includes the denormal numbers), as shown in Figure 2.2. On the one hand, overflow typically produces $\pm\infty$, which makes the error easily detectable. On the other hand, underflow can result in a number that lacks full precision if denormal numbers are

```cpp
#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

int main(){
    cout.precision(20);

    int n = 8;
    double x = 10;
    vector<double> f(n,1);

    for (int i=0; i<n; i++){
        f[i] = exp(x);
        cout << "f(" << x << ") = " << f[i] << endl;
        x *= 2;
    }
}
```

```
f(10) = 22026.465794806717895
f(20) = 485165195.40979027748
f(40) = 235385266837020000
f(80) = 5.5406223843935098345e+034
f(160) = 3.069849640644242423e+069
f(320) = 9.4239768161635848851e+138
f(640) = 8.8811339031588737941e+277
f(1280) = inf
```

Program 1. C++ code to compute $f(x) = e^x$ for $x = 10, 20, 40, 80, 160, 320, 640, 1280$ in double precision (left) and its output (right).

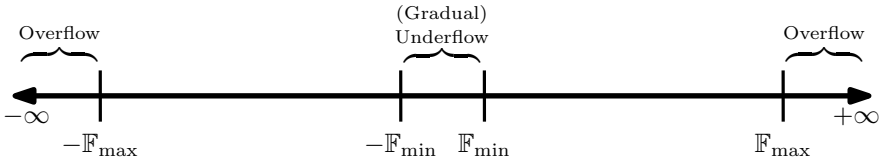


Figure 2.2. Overflow and (gradual) underflow regions with respect to the real line.

stored, or it can jump directly to zero otherwise. This makes underflow errors difficult to detect since the results appear as ordinary numbers.

However, there exist certain techniques that try to prevent overflow and underflow errors during computations with floating-point values. In particular, the approach that we use for these circumstances involves performing some rescale operations to maintain the results within safe bounds. Since it relies on knowing the interval in which each operation's outcome lies, we present the following proposition to estimate the range for the result of all the basic arithmetic operations between two arbitrary positive floating-point numbers.

**Proposition 2.1.** *Let $a = +\alpha \times 2^A \in \mathbb{F}$ and $b = +\beta \times 2^B \in \mathbb{F}$. Suppose we compute $c = a * b = \gamma \times 2^C \in \mathbb{F}$, where $* \in \{+, -, \cdot, \div\}$ is one of the standard arithmetic operations. Then, except in the case where $c = 0$, the exponent $C$ is guaranteed to be in the set $\{C_{\min}, C_{\min}+1, \ldots, C_{\max}\}$, where*

$$(C_{\min}, C_{\max}) = \begin{cases} (A+B-1, A+B), & \text{if } c = a \cdot b, \\ (A-B, A-B+1), & \text{if } c = a \div b, \\ (\max\{A,B\}, \max\{A,B\}+1), & \text{if } c = a + b, \\ (A-t+1, A-1), & \text{if } c = a - b \text{ and } A = B, \\ (\max\{A,B\}-t, \max\{A,B\}), & \text{if } c = a - b \text{ and } |A-B| = 1, \\ (\max\{A,B\}-1, \max\{A,B\}), & \text{if } c = a - b \text{ and } |A-B| > 1. \end{cases} \tag{2.4}$$

*Proof.* In the case of multiplication, since $\alpha, \beta \in [\mu_{\min}, \mu_{\max}]$ with $\mu_{\min} = 1/2$ and $\mu_{\max} = 1 - 2^{-t} < 1$, we have

$$\alpha \cdot \beta \geq \mu_{\min}^2 = 2^{-1}\mu_{\min} \qquad \text{and} \qquad \alpha \cdot \beta \leq \mu_{\max}^2 < 2^0 \mu_{\max},$$

so that $\alpha \cdot \beta = \gamma \times 2^D$ for some $\gamma \in [\mu_{\min}, \mu_{\max}]$ and $D \in \{-1, 0\}$, and the statement follows from the fact that $c = \alpha \cdot \beta \times 2^{A+B}$.

In the case of division, we have

$$\alpha/\beta \geq \mu_{\min}/\mu_{\max} > 2^0 \mu_{\min} \qquad \text{and} \qquad \alpha/\beta \leq \mu_{\max}/\mu_{\min} = 2\mu_{\max},$$

so that $\alpha/\beta = \gamma \times 2^D$ for some $\gamma \in [\mu_{\min}, \mu_{\max}]$ and $D \in \{0, 1\}$, and the statement follows from the fact that $c = \alpha/\beta \times 2^{A-B}$.

Without loss of generality, we assume $A \geq B$ and we prove the statement for $\max\{A, B\} = A$. Furthermore, we recall that the smallest positive floating-point number is $2^{-t}$, meaning that $2^{-t} \leq 2^{B-A} \leq 1$. In the case an addition, we have

$$\alpha + \beta 2^{B-A} \geq \mu_{\min} + \mu_{\min} 2^{-t} > 2^0 \mu_{\min} \qquad \text{and} \qquad \alpha + \beta 2^{B-A} \leq 2\mu_{\max},$$

so that $\alpha + \beta 2^{B-A} = \gamma \times 2^D$ for some $\gamma \in [\mu_{\min}, \mu_{\max}]$ and $D \in \{0, 1\}$, and the statement follows from the fact that $c = (\alpha + \beta 2^{B-A}) \times 2^A$.

In the case a subtraction, if $A = B$, then $\alpha \neq \beta$ otherwise $c = 0$. Moreover, since the minimum distance between two consecutive floating-point numbers is $2^{-t} = 2^{-t+1}\mu_{\min}$, we have

$$|\alpha - \beta| \geq 2^{-t+1}\mu_{\min} \qquad \text{and} \qquad |\alpha - \beta| \leq \mu_{\max} - \mu_{\min} = \mu_{\min} - 2^{-t} < 2^0 \mu_{\min},$$

so that $\alpha - \beta = \gamma \times 2^D$ for some $\gamma \in [\mu_{\min}, \mu_{\max}]$ and $D \in \{-t+1, -t+2, \ldots, -1\}$, where $D = 0$ is not included because $|\alpha - \beta|$ is always strictly less than $\mu_{\min}$. Therefore, the statement follows from the fact that $c = (\alpha - \beta) \times 2^A$. If $A = B + 1$ we have

$$\begin{aligned} &\alpha - \beta 2^{-1} \geq \mu_{\min} - \mu_{\max} 2^{-1} = 2^{-t}\mu_{\min} \qquad \text{and} \\ &\alpha - \beta 2^{-1} \leq \mu_{\max} - \mu_{\min} 2^{-1} < 2^0 \mu_{\max}, \end{aligned}$$

so that $\alpha - \beta 2^{-1} = \gamma \times 2^D$ for some $\gamma \in [\mu_{\min}, \mu_{\max}]$ and $D \in \{-t, -t+1, \ldots, 0\}$, and the statement follows from the fact that $c = (\alpha - \beta 2^{-1}) \times 2^A$. Finally, if $A > B + 1$ we have

$$\begin{aligned} &\alpha - \beta 2^{B-A} \geq \mu_{\min} - \mu_{\max} 2^{-2} > 2^{-1}\mu_{\min} \qquad \text{and} \\ &\alpha - \beta 2^{B-A} \leq \mu_{\max} - \mu_{\min} 2^{-t} < 2^0 \mu_{\max}, \end{aligned}$$

so that $\alpha - \beta 2^{B-A} = \gamma \times 2^D$ for some $\gamma \in [\mu_{\min}, \mu_{\max}]$ and $D \in \{-1, 0\}$, and the statement follows from the fact that $c = (\alpha - \beta 2^{B-A}) \times 2^A$. □

### 2.1.2 Floating-point arithmetic operations

In addition to the rounding errors introduced by mapping each $x \in \mathbb{R}$ to the closest floating-point approximation $\mathrm{fl}(x) \in \mathbb{F}$, we must also consider the errors introduced by any arithmetic operation performed between floating-point numbers. In this context, we denote by $\circledast$ the floating-point analogue of $* \in \{+, -, \times, \div\}$, that is,

$$x \circledast y = \mathrm{fl}(x * y),$$

for any $x, y \in \mathbb{F}$, and recall [78, Lecture 13] that, as for the operator fl, also $\circledast$ introduces a relative error of size at most $\epsilon$. This is equivalent to saying that, for any $x, y \in \mathbb{F}$ there exists some $\delta \in \mathbb{R}$ with $|\delta| < \epsilon$, such that

$$x \circledast y = (x * y)(1 + \delta). \tag{2.5}$$

We note that $\mathrm{fl}(-x) = -x$ for all $x \in \mathbb{F}$, so that multiplying a floating-point number by $(-1)^i$ or taking its absolute value does not entail any rounding error.

These are precisely the errors that we consider in our numerical stability analysis and, in particular, we examine how they propagate through the algorithm to the final result. To address this, we now present some fundamental facts that we frequently use in our studies, especially when dealing with cases involving multiple operations, such as sums or products.

- By Taylor expansion,
$$\frac{1}{1+y} = \sum_{k=0}^{\infty} (-1)^k y^k$$
for any $y \in \mathbb{R}$ with $|y| < 1$. Consequently, if $y = O(\epsilon)$, then
$$\frac{1}{1+y} = 1 - y + O(\epsilon^2). \tag{2.6}$$
Moreover, for any $\delta \in \mathbb{R}$ with $|\delta| \leq \epsilon$, there exists some $\delta' \in \mathbb{R}$ with $|\delta'| \leq \epsilon + O(\epsilon^2)$, such that
$$\frac{1}{1+\delta} = 1 + \delta'. \tag{2.7}$$
This observation is useful for "moving" the perturbation (2.5) caused by a floating-point operation from the denominator to the numerator.

- For any $\delta_1, \ldots, \delta_m \in \mathbb{R}$ with $|\delta_i| \leq C_i \epsilon$ for some $C_i > 0$, $i = 0, \ldots, n$, there exists some $\delta \in \mathbb{R}$ with $|\delta| \leq C\epsilon + O(\epsilon^2)$, where $C = \sum_{i=1}^{m} C_i$, such that
$$\prod_{j=1}^{m} (1 + \delta_j) = 1 + \delta. \tag{2.8}$$
We use this observation to gather the perturbations caused by computing the product of $m$ terms into a single perturbation. Moreover, by its definition, we note that $C$ depends on the integer $m$.

- For any $t_0, \ldots, t_m \in \mathbb{F}$, there exist some $\varphi_0, \ldots, \varphi_m \in \mathbb{R}$ with $|\varphi_0|, \ldots, |\varphi_m| \leq m\epsilon + O(\epsilon^2)$, such that
$$\mathrm{fl}\left(\sum_{i=0}^{m} t_i\right) = (\cdots((t_0 \oplus t_1) \oplus t_2) \cdots \oplus t_m) = \sum_{i=0}^{m} t_i(1 + \varphi_i). \tag{2.9}$$
This follows from the previous observation, and we use it to estimate the rounding error introduced by simple iterative summation of $m + 1$ floating-point numbers. It is worth pointing out that, in the literature, many algorithms have been developed to compute summations in a more stable way. Some of these methods yield smaller upper bounds on the relative errors, even independent of $m$, as in the case of the Kahan's summation algorithm [42].

### 2.1.3 Floating-point elementary functions

Since some of the algorithms we consider involve not only basic operations, but also square root and trigonometric functions, we must address how to manage and analyze the errors introduced in these cases. In general, we do not have results on the numerical stability of the elementary function implementations in standard libraries. However, some of them give information about the maximum relative errors in their specific implementations, such as the CUDA programming model [66] and the GNU library [60]. In our analysis, we always assume that we have stable algorithms to evaluate the elementary functions that we need. In other words, we assume that, for any $x \in \mathbb{F}$, there exist some $\delta_{\text{sqrt}}, \delta_{\sin}, \delta_{\arctan}, \delta_{\tan} \in \mathbb{R}$, such that

$$\text{fl}(\sqrt{x}) = \sqrt{x}(1+\delta_{\text{sqrt}}), \qquad |\delta_{\text{sqrt}}| \le D_{\text{sqrt}}\epsilon + O(\epsilon^2), \tag{2.10}$$

$$\text{fl}(\sin x) = \sin x(1+\delta_{\sin}), \qquad |\delta_{\sin}| \le D_{\sin}\epsilon + O(\epsilon^2), \tag{2.11}$$

$$\text{fl}(\cos x) = \cos x(1+\delta_{\cos}), \qquad |\delta_{\cos}| \le D_{\cos}\epsilon + O(\epsilon^2), \tag{2.12}$$

$$\text{fl}(\arctan x) = \arctan x(1+\delta_{\arctan}), \qquad |\delta_{\arctan}| \le D_{\arctan}\epsilon + O(\epsilon^2) \tag{2.13}$$

$$\text{fl}(\tan x) = \tan x(1+\delta_{\tan}), \qquad |\delta_{\tan}| \le D_{\tan}\epsilon + O(\epsilon^2) \tag{2.14}$$

for some constants $D_{\text{sqrt}}$,$D_{\sin}$,$D_{\cos}$,$D_{\arctan}$, and $D_{\tan}$. For example, for the IEEE standard 754 floating-point arithmetic [52], it is known [71] that $|\delta_{\text{sqrt}}| \le 1 - 1/\sqrt{1+2\epsilon}$, hence, by Taylor expansion, $D_{\text{sqrt}} = 1$.

While the bounds in (2.10)–(2.14) assume that the argument $x$ is a floating-point number, let us now derive these bounds for a perturbed data. Therefore, we now consider an arbitrary argument $y \in \mathbb{R}$, which is first rounded to a floating-point value $z = \text{fl}(y)$ and then passed to the elementary functions.

**Lemma 2.2.** *Let $z = y(1+\gamma) \in \mathbb{F}$, where $y \in \mathbb{R}$ and $\gamma \in \mathbb{R}$ satisfies $|\gamma| \le C\epsilon$, for some $C > 0$, and $f$ be a differentiable function at $y$. If $f(y) \neq 0$, then there exists some $\gamma' \in \mathbb{R}$ such that*

$$f(z) = f(y)(1+\gamma'), \qquad |\gamma'| \le \frac{|f'(y)y|}{|f(y)|}C\epsilon + O(\epsilon^2).$$

*Proof.* The statement follows immediately from the Taylor expansion of $f$ around $y$, that is,

$$f(z) = f(y) + f'(y)y\gamma + O(\epsilon^2) = f(y)\left(1 + \frac{f'(y)y}{f(y)}\gamma + O(\epsilon^2)\right).$$

□

**Corollary 2.3.** *Let $z = y(1+\gamma) \in \mathbb{F}$, where $y \in \mathbb{R}$ and $\gamma \in \mathbb{R}$ satisfies $|\gamma| \le C\epsilon$, for some $C > 0$. If $\sin y \neq 0$, $\cos y \neq 0$, and $\arctan y \neq 0$, then there exist some $\delta'_{\sin}, \delta'_{\cos}, \delta'_{\arctan} \in \mathbb{R}$, such that*

$$\text{fl}(\sin z) = \sin y(1+\delta'_{\sin}), \qquad |\delta'_{\sin}| \le (|\cot y||y|C + D_{\sin})\epsilon + O(\epsilon^2), \tag{2.15}$$

$$\text{fl}(\cos z) = \cos y(1+\delta'_{\cos}), \qquad |\delta'_{\cos}| \le (|\tan y||y|C + D_{\cos})\epsilon + O(\epsilon^2), \tag{2.16}$$

$$\text{fl}(\arctan z) = \arctan y(1+\delta'_{\arctan}), \qquad |\delta'_{\arctan}| \le (C + D_{\arctan})\epsilon + O(\epsilon^2). \tag{2.17}$$

*If $y \notin \{(2k+1)\pi/2, k \in \mathbb{Z}\}$ and $\tan y \neq 0$, then there exists some $\delta'_{\tan} \in \mathbb{R}$, such that*

$$\text{fl}(\tan z) = \tan y(1+\delta'_{\tan}), \qquad |\delta'_{\tan}| \le (2|y|/|\sin 2y|C + D_{\tan})\epsilon + O(\epsilon^2). \tag{2.18}$$

*If $y > 0$, then there exists some $\delta'_{\text{sqrt}} \in \mathbb{R}$, such that*

$$\text{fl}(\sqrt{z}) = \sqrt{y}(1+\delta'_{\text{sqrt}}), \qquad |\delta'_{\text{sqrt}}| \le (C/2 + D_{\text{sqrt}})\epsilon + O(\epsilon^2). \tag{2.19}$$

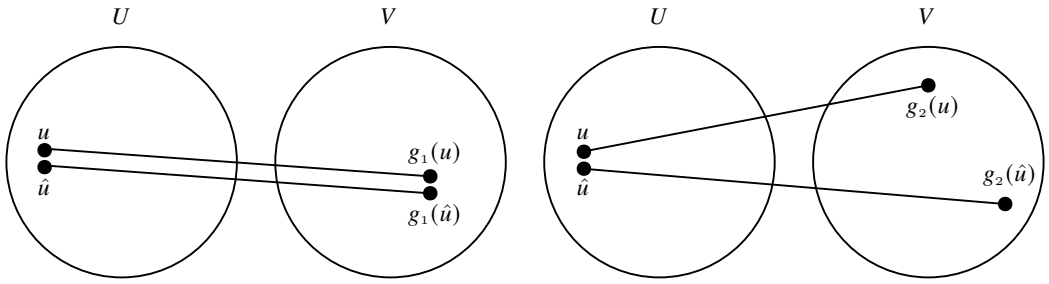


Figure 2.3. Comparison between the behaviors of a well-conditioned (left) and an ill-conditioned (right) problem, $g_1$ and $g_2$ respectively.

*Proof.* Equations (2.15), (2.16), (2.18) and (2.19) follow directly from (2.11), (2.12), (2.14) and (2.10), respectively, and Lemma 2.2. Regarding (2.17), Lemma 2.2 and (2.13) give

$$\mathrm{fl}(\arctan z) = \arctan y(1+\delta'_{\arctan}), \qquad |\delta'_{\arctan}| \leq \left(\left|\frac{y}{(1+y^2)\arctan y}\right| C + D_{\arctan}\right)\epsilon + O(\epsilon^2).$$

We note that $g(y) = y/[(1+y^2)\arctan y]$ is always positive, because $y$ and $\arctan y$ have the same sign. So, to complete the proof, it remains to show that $g(y) \leq 1$ for all $y > 0$. The first derivative of $g$ is given by

$$g'(y) = \frac{\arctan y - y^2 \arctan y - y}{[(1+y^2)\arctan y]^2}.$$

Since $h(y) = \arctan y - y$ is a decreasing function, we have $h(y) < h(0) = 0$ and therefore $g'(y) < 0$. This means that $g$ is a strictly decreasing function. Additionally, we know that $\lim_{x\to 0} \arctan x/x = 1$ and conclude

$$g(y) < \lim_{x\to 0} g(x) = \lim_{x\to 0} \frac{x}{(1+x^2)\arctan x} = 1.$$

□

## 2.2 Conditioning of a problem

An important concept related to the numerical stability of an algorithm is the *conditioning* of a problem, that is, the sensitivity of the solution to small perturbations of the data. Following Trefethen and Bau [78], a problem can be viewed as a function $g : U \to V$ from a normed vector space $(U, \|\cdot\|_U)$ of data to a normed vector space $(V, \|\cdot\|_V)$ of solutions. Furthermore, we say that $g$ is *well-conditioned* if *any* small perturbation of the data $u \in U$ leads to a small change of the solution $g(u) \in V$, while it is *ill-conditioned* if *some* small perturbation of the data $u$ leads to a large change of the solution $g(u)$. For instance, considering a well-conditioned and an ill-conditioned problem, $g_1$ and $g_2$ respectively, we observe in Figure 2.3 their different behaviours for an arbitrary data $u \in U$ and one of its problematic perturbation $\hat{u} \in U$ for the problem $g_2$.

The quantity that describes how the solution of $g$ behaves with respect to a perturbation of the data is called the *condition number* and it expresses the worst error that can affect the solution of the problem due to a possible perturbation of the data. There are two different

ways to define the conditioning of a problem, depending on the type of changes in both data and solution we are interested in. On the one hand, considering some data $u \in U$, $u \neq 0$, such that $g(u) \neq 0$ and denoting by $h \in U$ a possible small perturbation of $u$, then they can be measured by their relative errors $\|h\|_U/\|u\|_U$ and $\|g(u+h)-g(u)\|_V/\|g(u)\|_V$, respectively, and these are related by the *relative normwise condition number* [17, 38, 69, 78] as

$$c(u) = \lim_{\delta\to 0} \sup_{\|h\|_U \leq \delta} \left( \frac{\|g(u+h)-g(u)\|_V}{\|g(u)\|_V} \Big/ \frac{\|h\|_U}{\|u\|_U} \right).$$

On the other hand, supposing that $u = 0$ or $g(u) = 0$, then the right quantities to consider are now the absolute errors $\|h\|_U$ and $\|g(u+h)-g(u)\|_V$, so we define the *absolute normwise condition number* [17, 69, 78] as

$$\tilde{c}(u) = \lim_{\delta\to 0} \sup_{\|h\|_U \leq \delta} \frac{\|g(u+h)-g(u)\|_V}{\|h\|_U}.$$

The greater these values, the more ill-conditioned the problem. Regardless of the distinction between relative and absolute normwise conditioning number, this is the most general definition we can find in the literature for an arbitrary problem $g$. However, it may happen that we must deal with some restriction on the perturbations in $u$ and, in this case, it makes sense to define the condition number in such a way to take this additional information into account.

Let us now consider a small positive value $\delta \in \mathbb{R}$ and we define $H_\delta$ as the set of all possible perturbations $h \in U \setminus \{0\}$ of the data $u$ satisfying the componentwise inequality

$$|h_i| < \delta |u_i|, \qquad i = 1, \ldots, n, \tag{2.20}$$

where $n$ is the dimension of the space $U$. If we consider again some data $u \in U$, $u \neq 0$, such that $g(u) \neq 0$, then we can relate the relative error in the norm of the output with the componentwise relative error of the input through the *mixed relative condition number* [38]

$$m(u) = \lim_{\delta\to 0} \sup_{h\in H_\delta} \left( \frac{\|g(u+h)-g(u)\|_V}{\|g(u)\|_V} \Big/ \max_{\substack{i=1,\ldots,n \\ u_i\neq 0}} \frac{|h_i|}{|u_i|} \right).$$

Otherwise, we can use the componentwise relative error also for the output and this leads to the definition of the *componentwise relative condition number* [38]

$$\kappa(u) = \lim_{\delta\to 0} \sup_{h\in H_\delta} \left( \max_{\substack{i=1,\ldots,m \\ g_i(u)\neq 0}} \frac{|g_i(u+h)-g_i(u)|}{|g_i(u)|} \Big/ \max_{\substack{i=1,\ldots,n \\ u_i\neq 0}} \frac{|h_i|}{|u_i|} \right), \tag{2.21}$$

where $m$ is the dimension of the space $V$.

Condition (2.20) is common when the perturbed data $u + h \in U$ comes from the finite-precision approximation of the real data $u$ through an element of the set $\mathbb{F}$, and this makes the choice between $\kappa(u)$, $m(u)$, and $c(u)$ crucial for some problem $g$. In fact, there are cases where it is possible to find an arbitrary perturbation $h$ that makes $g$ ill-conditioned in $u$, while $g$ is well-conditioned with respect to all $h \in H_\delta$.

Below, we will see how the conditioning of a problem is related to the numerical stability of an algorithm designed to solve it.

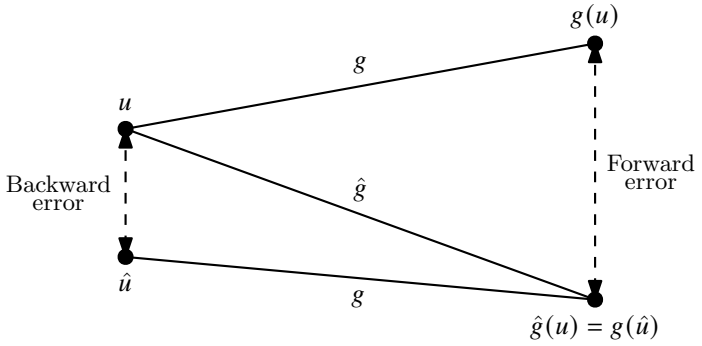


Figure 2.4. Relation between forward and backward errors.

## 2.3 Numerical stability of an algorithm

Suppose we want to solve a general problem $g: U \to V$ with a numerical algorithm on a computer, that, following again Trefethen and Bau [78], can be viewed as another function $\hat{g}: U \to V$ between the same normed vector spaces of $g$. As seen previously, the floating-point arithmetic leads to rounding errors during the computation, so we cannot expect to get the exact solution of a problem by solving it with an algorithm. Then, beyond the concept of conditioning, which is related to the behaviour of the solution of $g$ independently of the algorithm used, we also need to introduce the *stability* of an algorithm, which instead expresses how *accurate* the answer of $\hat{g}$ to the problem is. More precisely, there are two types of stability: the *forward stability*, which states how far the computed solution is from the exact one, and the *backward stability*, which instead quantifies how much we need to perturb the data in order to produce the approximate solution.

Given some data $u \in U$ at which we want to evaluate the problem $g$, the usual quantity to consider is the difference between the computed solution $\hat{g}(u)$ and the exact solution $g(u)$, which is captured by the *absolute forward error* $\|\hat{g}(u) - g(u)\|_V$ or the *relative forward error* $\|\hat{g}(u) - g(u)\|_V / \|g(u)\|_V$. Assuming $g(u) \neq 0$, then the algorithm $\hat{g}$ is called *accurate* or *forward stable*, if

$$\frac{\|\hat{g}(u) - g(u)\|_V}{\|g(u)\|_V} = O(\epsilon) \tag{2.22}$$

for all $u \in U$, where the notation $x = O(\epsilon)$ means that there exists some positive constant $C$, such that $|x| \leq C\epsilon$ as $\epsilon \to 0$. On the other hand, if $g(u) = 0$, we can use the same definition, but considering the absolute forward error on the left-hand side of (2.22). Furthermore, given some nonzero data $u \in U$, the algorithm $\hat{g}$ is called *backward stable*, if

$$\hat{g}(u) = g(\hat{u}) \qquad \text{for some} \quad \hat{u} \in U \quad \text{with} \quad \frac{\|\hat{u} - u\|_U}{\|u\|_U} = O(\epsilon), \tag{2.23}$$

where the quantity $\|\hat{u} - u\|_U / \|u\|_U$ is known as the *relative backward error* and, in the case of $u = 0$, it can be replaced by the *absolute backward error* $\|\hat{u} - u\|_U$. We can also see graphically how forward and backward errors relate to each other in Figure 2.4.

The forward stability of an algorithm is influenced by the condition number of the problem that it is designed to solve. In some definitions, this dependence is made explicit. For instance,

Harbrecht and Multerer [41] define an algorithm as relative forward stable if

$$\frac{\|\hat{g}(u)-g(u)\|_V}{\|g(u)\|_V} \leq Kc(u)\epsilon,$$

for some moderately large positive constant $K$. This approach allows an algorithm to be considered forward stable even if the problem is ill-conditioned. Instead, the definition in (2.22) that we adopt for our study is equivalent to the previous one, except that the condition number is incorporated implicitly within the constant $K$. Consequently, under our definition, if a problem is ill-conditioned, no algorithm developed to address it can be considered forward stable.

To conclude, the forward and backward stability are not independent of each other, but they are joined by the conditioning of the problem. In fact, Trefethen and Bau [78, Theorem 15.1] proved that, supposing we have a backward stable algorithm $\hat{g}$ to solve a well-conditioned problem $g$ on a computer satisfying (2.3) and (2.5), then $\hat{g}$ is also forward stable, while it might not be if $g$ is ill-conditioned. Moreover, knowing the relative (absolute) backward error and the relative (absolute) normwise condition number, their product gives an upper bound on the relative (absolute) forward error [17], that is,

$$\frac{\|\hat{g}(u)-g(u)\|_V}{\|g(u)\|_V} \leq c(u)\frac{\|\hat{u}-u\|_U}{\|u\|_U}.$$

# Chapter 3

# The numerical stability of linear barycentric rational interpolation

## 3.1 State of the art

In this chapter, we focus on the barycentric interpolation problem in the univariate case. Therefore, let us first recall the *classical interpolation problem*: given $n+1$ distinct interpolation nodes $X_n = (x_0, \ldots, x_n) \in \mathbb{R}^{n+1}$ with associated data $Y_n = (y_0, \ldots, y_n) \in \mathbb{R}^{n+1}$, we search for a function $r : \mathbb{R} \to \mathbb{R}$ that interpolates $y_i$ at $x_i$, that is,

$$r(x_i) = y_i, \qquad i = 0, \ldots, n. \tag{3.1}$$

As mentioned in the introduction, in the literature there are many options for choosing this function, including both polynomial and rational forms. Below, we examine all these methods and report what is already known about their numerical stability.

In the case of *polynomial* interpolation of degree at most $n$, this problem has a unique solution, which can be expressed in Lagrange form as

$$r(x) = \sum_{i=0}^{n} \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} y_i.$$

While this form is advantageous for theoretical analysis, its evaluation requires $O(n^2)$ operations and can be numerically unstable. It is advisable to consider instead the *first polynomial barycentric form* of $r$,

$$r(x) = \prod_{j=0}^{n} (x - x_j) \sum_{i=0}^{n} \frac{\gamma_i}{x - x_i} y_i, \qquad \gamma_i = \prod_{j=0, j\neq i}^{n} \frac{1}{x_i - x_j}. \tag{3.2}$$

The latter is more efficient than the Lagrange form, as it can be evaluated in $O(n)$ operations, after computing the $\gamma_i$, which are independent of $x$, in $O(n^2)$ operations in a preprocessing step. Higham [44] shows that this evaluation is backward stable with respect to perturbations of the data $y_i$. Another means of evaluating $r$ in polynomial form is given by the *second barycentric form*

$$r(x) = \frac{\sum_{i=0}^{n} \frac{\gamma_i}{x - x_i} y_i}{\sum_{i=0}^{n} \frac{\gamma_i}{x - x_i}} \tag{3.3}$$

| Type of interpolant | Forward stability | Backward stability |
|---|---|---|
| First barycentric polynomial | ✗ | Higham [44] |
| Second barycentric polynomial | Higham [44] | ✗ |
| First barycentric rational | ✗ | de Camargo [18] |
| Second barycentric rational | Salazar Celis [72] | Mascarenhas and Camargo [63] |

Table 3.1. State of the art on the numerical stability of polynomial and rational barycentric interpolation; if the forward or backward stability has not been covered for a specific type of interpolant, then it is denoted by ✗, otherwise there is the corresponding reference.

with Lagrange weights $\gamma_i$ in (3.2). This formula can be derived from (3.2) by noticing that the function that interpolates the constant one function with data $y_i = 1$, $i = 0, \dots, n$, is given by $1 = \ell(x)\sum_{i=0}^{n} \frac{\gamma_i}{x-x_i}$. Evaluating this formula also requires $O(n)$ operations, but it comes with the advantage that the $\gamma_i$ can be rescaled by a common factor to avoid underflow and overflow [8]. Moreover, the second barycentric form is forward stable, as long as the Lebesgue constant associated with the interpolation nodes $x_i$ is small [44], which is the case, for example, for Chebyshev nodes of the first and the second kind, but not for equidistant nodes [16].

In the *rational* case, the interpolation problem (3.1) no longer has a unique solution, but Berrut and Mittelmann [7] show that every rational interpolant of degree at most $n$ can be expressed in the second barycentric form (3.3) for a specific choice of weights $\gamma_i$. Vice versa, Schneider and Werner [74] note that for any set of non-zero weights $\gamma_i$, the function $r$ in (3.3) is a rational interpolant of degree at most $n$. An important subset of these barycentric rational interpolants are those that do not have any poles in $\mathbb{R}$. This is obviously true for the Lagrange weights in (3.2), but also for the Berrut weights

$$\gamma_i = (-1)^i, \qquad i = 0, \dots, n, \tag{3.4}$$

and for the family of weights

$$\gamma_i = \sum_{k=\max(i-d,0)}^{\min(i,n-d)} (-1)^k \prod_{j=k,\, j\neq i}^{k+d} \frac{1}{x_i - x_j}, \qquad d \in \{0, \dots, n\}, \tag{3.5}$$

where the Berrut and the Lagrange weights are obtained for $d = 0$ and $d = n$, respectively. Moreover, Floater and Hormann observe that this rational interpolant in (3.3) with weights in (3.5) is mathematically equivalent to the first barycentric form

$$r(x) = \sum_{i=0}^{n} \frac{\frac{\gamma_i}{x-x_i}}{\sum_{i=0}^{n-d} \lambda_i(x)} y_i, \tag{3.6}$$

where

$$\lambda_i(x) = \frac{(-1)^i}{(x-x_i)\cdots(x-x_{i+d})}, \qquad i = 0, 1, \dots, n-d. \tag{3.7}$$

The result of Higham [44] can be extended to show that the evaluation of the second barycentric form (3.3) is forward stable, not only in the case of polynomial interpolation, but for general barycentric rational interpolants, provided that the weights $\gamma_i$ can be computed with a forward stable algorithm and that the corresponding Lebesgue constant is small [72]. For Berrut's

rational interpolant with weights in (3.4), this is the case for all well-spaced interpolation nodes [14], including equidistant and Chebyshev nodes. For the family of barycentric rational interpolants with weights in (3.5), the Lebesgue constant is known to grow logarithmically in $n$, for any constant $d > 0$ and equidistant [13] as well as quasi-equidistant [48] nodes, and the formula in (3.5) turns out to be a forward stable means of computing the weights $\gamma_i$ [18]. Regarding backward stability, Mascarenhas and Camargo [63] show that the second barycentric form is backward stable under the same assumptions, namely forward stable weights and small Lebesgue constant. Moreover, Camargo [18] proves that the first barycentric form is backward stable, as long as the denominator in (3.6) is computed with a special algorithm in $O(nd)$ operations.

We summarize the existing studies on the numerical stability of both polynomial and rational barycentric interpolation in Table 3.1, also highlighting what was previously unknown. The next section outlines instead our contributions, which address and fill all these gaps in the literature.

## 3.2 Our contribution

In this chapter, we conduct a comprehensive analysis of the error propagation that occurs in the various barycentric interpolation methods introduced earlier. While our approach mostly aligns with the existing research, it also introduces some key innovations. A primary difference is the adoption of a new notation to express the upper bounds on the relative errors. In fact, all the references cited in Table 3.1 use the Stewart's error counter $\langle k \rangle$ to measure the amplification of rounding errors in the various algorithms, that is,

$$\langle k \rangle = \prod_{i=1}^{k} (1+\delta_i)^{\rho_i}, \qquad \rho_i = \pm 1, \qquad |\delta_i| \leq \epsilon.$$

Then, assuming $k\epsilon < 1$, they use the fact that $|\langle k \rangle - 1| \leq k\epsilon/(1-k\epsilon)$ [43] to determine the corresponding upper bound on the relative error. In other words, if a quantity $x \in \mathbb{R}$ satisfies $\mathrm{fl}(x) = x\langle k \rangle$, then it holds

$$\frac{|\mathrm{fl}(x) - x|}{|x|} = |\langle k \rangle - 1| \leq \frac{k\epsilon}{1-k\epsilon}.$$

In our study, we instead use the notation

$$\mathrm{fl}(x) = x(1+\delta), \qquad |\delta| \leq k\epsilon + O(\epsilon^2),$$

therefore the upper bound on $|\delta|$ also provides the upper bound on the relative error of $x$.

Regarding forward stability, we further generalize the proof of Salazar Celis [72], such that it can also be used for proving the forward stability of the first barycentric form (3.6), with two important changes. On the one hand, the result relies on the fact that not only the $\gamma_i$, but also the $\lambda_i$ can be evaluated with a forward stable algorithm. This is indeed the case for the formula in (3.7), which requires $O(d(n-d))$ operations for $0 < d < n$, as well as a more efficient, but slightly less stable formula that gets by with $O(n)$ operations. On the other hand, the Lebesgue constant must be replaced by a similar quantity that depends on the functions $\lambda_i$, for which we prove that it is at most on the order of $O(\mu^d)$, where $\mu$ is the *mesh ratio* of the interpolation nodes. Moreover, we show that a more efficient formula [49] for computing the weights in (3.5) is forward stable, too.

For backward stability, we present a novel approach that can be used for both barycentric forms. For the second barycentric form, our result provides an upper bound on the perturbation of the data $y_i$ that is smaller than the upper bound by Mascarenhas and Camargo [63]. For the first barycentric form, our upper bound is larger than the one found by Camargo [18], but it comes with the advantage of holding for a more efficient way of computing the denominator in (3.6) in $O(n)$ operations, which is based on our new $O(n)$ algorithm for evaluating the $\lambda_i$.

Our numerical experiments confirm that our new method leads to a considerably faster evaluation of the first barycentric form (3.6) of the rational interpolant with weights in (3.5) than the algorithm proposed by Camargo, especially for larger $d$, at the price of only marginally larger forward errors. Evaluating the interpolant using the second barycentric form (3.3) is even faster and can be as stable, but may also result in significantly larger errors for certain choices of interpolation nodes. However, we also report a case in which the second form is stable and the first is not. Furthermore, we analyzed both methods in scenarios where the evaluation points are near a root of the interpolant or an interpolation node.

Finally, it is important to note that our results hold under the assumption that the input values to the algorithm, $x_i$, $y_i$, and $x$, are given as floating-point numbers, so they do not introduce any additional error when we compute both forms (3.3) and (3.6). Consequently, our stability analysis does not cover errors that result from initially rounding the given values to floating-point numbers.

## 3.3 Forward stability

For analysing the relative forward error of barycentric rational interpolation, we first observe that (3.3) and (3.6) can both be written in the common form

$$r(x) = \frac{\sum_{i=0}^{n} a_i(x) y_i}{\sum_{j=0}^{m} b_j(x)}, \tag{3.8}$$

where $a_i(x) = \gamma_i/(x - x_i)$ for both forms, while $m = n$ and $b_j(x) = a_j(x)$ for the second form and $m = n - d$ and $b_j(x) = \lambda_j(x)$ for the first form. Next, we define the functions

$$\alpha(x; Y_n) = \frac{\sum_{i=0}^{n} |a_i(x) y_i|}{|\sum_{i=0}^{n} a_i(x) y_i|} \tag{3.9}$$

and

$$\beta(x) = \frac{\sum_{j=0}^{m} |b_j(x)|}{|\sum_{j=0}^{m} b_j(x)|}. \tag{3.10}$$

Assuming now that we have forward stable algorithms for computing with the floating-point arithmetic $a_i(x)$ as $\mathrm{fl}(a_i(x))$ and $b_j(x)$ as $\mathrm{fl}(b_j(x))$, we can derive a general bound on the relative forward error for the function $r$ in (3.8).

**Theorem 3.1.** *Suppose that there exist $\alpha_0, \ldots, \alpha_n \in \mathbb{R}$ with*

$$\mathrm{fl}(a_i(x)) = a_i(x)(1 + \alpha_i), \qquad |\alpha_i| \leq A\epsilon + O(\epsilon^2), \qquad i = 0, \ldots, n$$

*and $\beta_0, \ldots, \beta_m \in \mathbb{R}$ with*

$$\mathrm{fl}(b_j(x)) = b_j(x)(1 + \beta_j), \qquad |\beta_j| \leq B\epsilon + O(\epsilon^2), \qquad j = 0, \ldots, m$$

*for some constants A and B. Then, assuming $Y_n \in \mathbb{F}^{n+1}$, the relative forward error of r in* (3.8) *satisfies*

$$\frac{|\mathrm{fl}(r(x)) - r(x)|}{|r(x)|} \leq (n+2+A)\alpha(x;Y_n)\epsilon + (m+B)\beta(x)\epsilon + O(\epsilon^2), \tag{3.11}$$

*for $\epsilon$ small enough.*

*Proof.* We first notice that $\mathrm{fl}(r)$ is given by

$$\begin{aligned}\mathrm{fl}(r(x)) &= \frac{\sum_{i=0}^n \mathrm{fl}(a_i(x))y_i(1+\delta_i^\times)(1+\varphi_i^N)}{\sum_{j=0}^m \mathrm{fl}(b_j(x))(1+\varphi_j^D)}(1+\delta^\div) \\ &= \frac{\sum_{i=0}^n a_i(x)(1+\alpha_i)y_i(1+\delta_i^\times)(1+\varphi_i^N)}{\sum_{j=0}^m b_j(x)(1+\beta_j)(1+\varphi_j^D)}(1+\delta^\div),\end{aligned}$$

where $\delta_i^\times$, $\varphi_i^N$, $\varphi_j^D$ and $\delta^\div$ are the relative errors introduced by the product $\mathrm{fl}(a_i(x))y_i$, the sums in the numerator and the denominator, and the final division, respectively. It then follows from (2.5) that $|\delta_i^\times|, |\delta^\div| \leq \epsilon$, while from (2.9) we have $|\varphi_i^N| \leq n\epsilon + O(\epsilon^2)$ and $|\varphi_j^D| \leq m\epsilon + O(\epsilon^2)$. By (2.8), there exist some $\eta_i, \mu_j \in \mathbb{R}$ with

$$\begin{aligned}|\eta_i| &\leq (n+2+A)\epsilon + O(\epsilon^2), \qquad i = 0,\dots,n, \\ |\mu_j| &\leq (m+B)\epsilon + O(\epsilon^2), \qquad j = 0,\dots,m,\end{aligned} \tag{3.12}$$

such that

$$\mathrm{fl}(r(x)) = \frac{\sum_{i=0}^n a_i(x)y_i(1+\eta_i)}{\sum_{j=0}^m b_j(x)(1+\mu_j)}. \tag{3.13}$$

Therefore,

$$\begin{aligned}\frac{\mathrm{fl}(r(x))}{r(x)} &= \frac{\sum_{i=0}^n a_i(x)y_i(1+\eta_i)}{\sum_{j=0}^m b_j(x)(1+\mu_j)} \Big/ \frac{\sum_{i=0}^n a_i(x)y_i}{\sum_{j=0}^m b_j(x)} \\ &= \frac{\sum_{i=0}^n a_i(x)y_i(1+\eta_i)}{\sum_{j=0}^m a_i(x)y_i} \frac{\sum_{j=0}^m b_j(x)}{\sum_{j=0}^m b_j(x)(1+\mu_j)} \\ &= \left(1 + \frac{\sum_{i=0}^n a_i(x)y_i\eta_i}{\sum_{j=0}^m a_i(x)y_i}\right) \frac{1}{1 + \dfrac{\sum_{j=0}^m b_j(x)\mu_j}{\sum_{j=0}^m b_j(x)}}.\end{aligned}$$

Since, by the triangle inequality,

$$\left|\frac{\sum_{i=0}^n a_i(x)\eta_i}{\sum_{i=0}^n a_i(x)}\right| \leq \alpha(x) \max_{i=0,\dots,n} |\eta_i| \leq \alpha(x)(n+2+A)\epsilon + O(\epsilon^2) \tag{3.14}$$

and

$$\left|\frac{\sum_{j=0}^m b_j(x)\mu_j}{\sum_{j=0}^m b_j(x)}\right| \leq \beta(x) \max_{j=0,\dots,m} |\mu_j| \leq \beta(x)(m+B)\epsilon + O(\epsilon^2), \tag{3.15}$$

we can use (2.6) to express this ratio as

$$\frac{\mathrm{fl}(r(x))}{r(x)} = \left(1 + \frac{\sum_{i=0}^{n} a_i(x) y_i \eta_i}{\sum_{j=0}^{m} a_i(x) y_i}\right)\left(1 - \frac{\sum_{j=0}^{m} b_j(x)\mu_j}{\sum_{j=0}^{m} b_j(x)} + O(\epsilon^2)\right)$$
$$= 1 + \frac{\sum_{i=0}^{n} a_i(x) y_i \eta_i}{\sum_{j=0}^{m} a_i(x) y_i} - \frac{\sum_{j=0}^{m} b_j(x)\mu_j}{\sum_{j=0}^{m} b_j(x)} + O(\epsilon^2)$$

and obtain the relative forward error of $r$ as

$$\frac{|\mathrm{fl}(r(x)) - r(x)|}{|r(x)|} = \left|\frac{\mathrm{fl}(r(x))}{r(x)} - 1\right| = \left|\frac{\sum_{i=0}^{n} a_i(x) y_i \eta_i}{\sum_{j=0}^{m} a_i(x) y_i} - \frac{\sum_{j=0}^{m} b_j(x)\mu_j}{\sum_{j=0}^{m} b_j(x)} + O(\epsilon^2)\right|.$$

The upper bound in (3.11) then follows immediately by using again (3.14) and (3.15). □

While Theorem 3.1 holds for any function $r$ that can be expressed as in (3.8), we shall now focus on the special cases of the two different forms of the barycentric rational interpolant. For the second barycentric form, the only assumption we need is that the weights $\gamma_i$ can be computed with floating-point arithmetic as $\mathrm{fl}(\gamma_i)$ with a forward stable algorithm. Moreover, we recall the definition of the *Lebesgue function* of the barycentric rational interpolant [13] as

$$\Lambda_n(x; X_n) = \frac{\sum_{i=0}^{n} \left|\frac{\gamma_i}{x - x_i}\right|}{\left|\sum_{i=0}^{n} \frac{\gamma_i}{x - x_i}\right|}.$$

**Corollary 3.2.** *Assume that there exist $\psi_0, \ldots, \psi_n \in \mathbb{R}$ with*

$$\mathrm{fl}(\gamma_i) = \gamma_i(1 + \psi_i), \qquad |\psi_i| \le W\epsilon + O(\epsilon^2), \qquad i = 0, \ldots, n \tag{3.16}$$

*for some constant $W$. Then, assuming $Y_n \in \mathbb{F}^{n+1}$, the relative forward error of the second barycentric form in* (3.3) *satisfies*

$$\frac{|\mathrm{fl}(r(x)) - r(x)|}{|r(x)|} \le (n + 4 + W)\,\kappa(x; X_n, Y_n)\epsilon + (n + 2 + W)\Lambda_n(x; X_n)\epsilon + O(\epsilon^2), \tag{3.17}$$

*for $\epsilon$ small enough.*

*Proof.* We first notice that $a_i(x) = \gamma_i/(x - x_i)$ can be computed with one subtraction and one division, so that, by (2.5), (2.7), (2.8), and (3.16),

$$\mathrm{fl}(a_i(x)) = a_i(x)(1 + \alpha_i), \qquad i = 0, \ldots, n$$

for some $\alpha_i \in \mathbb{R}$ with $|\alpha_i| \le (2+W)\epsilon + O(\epsilon^2)$. Hence, the constants in (3.11) are $A = 2 + W$ and further $B = 2 + W$, because $b_j(x) = a_j(x)$ in case of the second barycentric form. From the latter, it also follows immediately that $\beta(x)$ in (3.10) is equal to $\Lambda_n(x; X_n)$ in this case, and it only remains to show that $\alpha(x; Y_n)$ in (3.9) is equal to $\kappa(x; X_n, Y_n)$ in (2.21). To this end, we first use the triangle inequality to see that for any $\delta > 0$ and any $H_n = (h_0, \ldots, h_n) \in \mathbb{R}^{n+1} \setminus \{(\underbrace{0, \ldots, 0}_{n+1 \text{ times}})\}$ with $|h_i| \le \delta |y_i|$,

$$\left|\sum_{i=0}^{n} a_i(x) h_i\right| = \left|\sum_{i=0}^{n} a_i(x) y_i \frac{h_i}{y_i}\right| \le M \sum_{i=0}^{n} |a_i(x) y_i|, \qquad M = \max_{\substack{i=0,\ldots,n \\ y_i \neq 0}} \frac{|h_i|}{|y_i|},$$

where equality is attained for $H_n$ with $h_i = \delta \operatorname{sign}(a_i(x))|y_i|$, $i = 0, \dots, n$. Dividing both sides of this inequality by $M$ and $\left|\sum_{i=0}^{n} a_i(x) y_i\right|$, taking the supremum over all admissible $H_n$ and the limit $\delta \to 0$, gives $\kappa(x; X_n, Y_n) = \alpha(x; Y_n)$. □

For the first barycentric form, we additionally need to assume that the $\lambda_i(x)$ can be computed with a forward stable algorithm as $\mathrm{fl}(\lambda_i(x))$, and we note that $\beta(x)$ in (3.10) is equal to

$$\Gamma_d(x; X_n) = \frac{\sum_{i=0}^{n-d} |\lambda_i(x)|}{\left|\sum_{i=0}^{n-d} \lambda_i(x)\right|} \tag{3.18}$$

in this case.

**Corollary 3.3.** *Assume that the weights $\gamma_0, \dots, \gamma_n$ can be computed as in* (3.16) *and that there exist $\varphi_0, \dots, \varphi_{n-d} \in \mathbb{R}$ with*

$$\mathrm{fl}(\lambda_i(x)) = \lambda_i(x)(1 + \varphi_i), \qquad |\varphi_i| \leq C\epsilon + O(\epsilon^2), \qquad i = 0, \dots, n-d \tag{3.19}$$

*for some constant $C$. Then, assuming $Y_n \in \mathbb{F}^{n+1}$, the relative forward error of the first barycentric form in* (3.6) *satisfies*

$$\frac{|\mathrm{fl}(r(x)) - r(x)|}{|r(x)|} \leq (n + 4 + W)\,\kappa(x; X_n, Y_n)\epsilon + (n - d + C)\Gamma_d(x; X_n)\epsilon + O(\epsilon^2), \tag{3.20}$$

*for $\epsilon$ small enough.*

*Proof.* As the numerator of the first and second barycentric form are identical, the only difference to the proof of Corollary 3.2 is that $B = C$ in (3.11), because $b_j(x) = \lambda_j(x)$ and $m = n - d$. □

While upper bounds for the Lebesgue function $\Lambda_n(x; X_n)$ can be found in the literature [13, 14], we are unaware of any previous work bounding the function $\Gamma_d(x; X_n)$, and we derive such an upper bound in Section 3.6.

## 3.4 Computing the weights $\gamma_i$ and evaluating the functions $\lambda_i$

It remains to work out the constants $W$ and $C$, related to the computation of the weights $\gamma_i$ in (3.3) and the evaluation of the functions $\lambda_i$ in (3.6). In particular, we study the error propagation that occurs in the implementation of different algorithms and further analyse them in terms of computational cost.

Regarding the $\gamma_i$, it was shown by Higham [44] that the Lagrange weights in (3.2) can be computed stably with $W = 2n$ in (3.16), and the Berrut weights in (3.4) can be represented exactly in $\mathbb{F}$, so that $W = 0$. The same holds for the weights in (3.5) if the interpolation nodes are equidistant, because they simplify to the integers

$$\gamma_i = (-1)^{i-d} \sum_{j=\max(i-d,0)}^{\min(i,n-d)} \binom{d}{i-j}, \qquad i = 0, \dots, n$$

in this special case [30]. For the general case, Camargo [18, Lemma 1] shows that $W = 3d$, if the $\gamma_i$ are computed with a straightforward implementation of the formula in (3.5). While

this construction requires $O(nd^2)$ operations, Hormann and Schaefer [49] suggest an improved $O(nd)$ pyramid algorithm, which turns out to have the same precision. Their algorithm starts from the values

$$v_i^d = 1, \qquad i = 0, \dots, n-d \tag{3.21}$$

and iteratively computes

$$v_i^l = \frac{v_{i-1}^{l+1}}{x_{i+l} - x_{i-1}} + \frac{v_i^{l+1}}{x_{i+l+1} - x_i}, \qquad i = 0, \dots, n-l \tag{3.22}$$

for $l = d-1, d-2, \dots, 0$, tacitly assuming $v_i^l = 0$ for $i < 0$ and $i > n-l$. They show that the resulting values $v_i^0$ are essentially the weights $\gamma_i$ in (3.5), up to a factor of $(-1)^{i-d}$.

**Lemma 3.4.** *For any $x_0, \dots, x_n \in \mathbb{F}$, there exist $\phi_0^0, \dots, \phi_n^0 \in \mathbb{R}$ with $|\phi_0^0|, \dots, |\phi_n^0| \leq W\epsilon + O(\epsilon^2)$ for $W = 3d$, such that the $v_i^0$ in (3.22) satisfy*

$$\mathrm{fl}(v_i^0) = v_i^0(1 + \phi_i^0), \qquad i = 0, \dots, n.$$

*Proof.* The statement is a special case of the more general observation that there exists for any $l = d, d-1, \dots, 0$ and $i = 0, \dots, n-l$ some $\phi_i^l \in \mathbb{R}$ with $|\phi_i^l| \leq 3(d-l)\epsilon + O(\epsilon^2)$, such that $\mathrm{fl}(v_i^l) = v_i^l(1+\phi_i^l)$, which can be shown by induction over $l$. The base case $l = d$ follows trivially from (3.21). For the inductive step from $l+1$ to $l$, we conclude from (2.5), (2.7), and (2.8), that $\mathrm{fl}(v_i^l)$, computed with the formula in (3.22), satisfies

$$\mathrm{fl}(v_i^l) = \frac{\mathrm{fl}(v_{i-1}^{l+1})}{x_{i+l} - x_{i-1}}(1+\rho_1) + \frac{\mathrm{fl}(v_i^{l+1})}{x_{i+l+1} - x_i}(1+\rho_2)$$

for some $\rho_1, \rho_2 \in \mathbb{R}$ with $|\rho_1|, |\rho_2| \leq 3\epsilon + O(\epsilon^2)$, since both terms are affected by one subtraction, one division, and one sum. By induction hypothesis and (2.8), we can then assume the existence of some $\sigma_1, \sigma_2 \in \mathbb{R}$ with $|\sigma_1|, |\sigma_2| \leq 3(d-l)\epsilon + O(\epsilon^2)$, such that

$$\mathrm{fl}(v_i^l) = \frac{v_{i-1}^{l+1}}{x_{i+l} - x_{i-1}}(1+\sigma_1) + \frac{v_i^{l+1}}{x_{i+l+1} - x_i}(1+\sigma_2),$$

and the intermediate value theorem further guarantees that

$$\mathrm{fl}(v_i^l) = \left(\frac{v_{i-1}^{l+1}}{x_{i+l} - x_{i-1}} + \frac{v_i^{l+1}}{x_{i+l+1} - x_i}\right)(1+\phi_i^l)$$

for some $\phi_i^l \in [\min(\sigma_1, \sigma_2), \max(\sigma_1, \sigma_2)]$ with $|\phi_i^l| \leq 3(d-l)\epsilon + O(\epsilon^2)$. □

Let us now focus on the functions $\lambda_i$ that appear in the barycentric formula (3.6) and first study the error propagation when computing them straightforwardly, following the formula in (3.7).

**Lemma 3.5.** *For any $x \in \mathbb{F}$ and $x_0, \dots, x_n \in \mathbb{F}$, there exist $\theta_0, \dots, \theta_{n-d} \in \mathbb{R}$ with $|\theta_0|, \dots, |\theta_{n-d}| \leq C\epsilon + O(\epsilon^2)$ for $C = 2d + 2$, such that the $\lambda_i(x)$ in (3.7) satisfy*

$$\mathrm{fl}(\lambda_i(x)) = \lambda_i(x)(1+\theta_i), \qquad i = 0, \dots, n-d.$$

*Proof.* Since computing $\mathrm{fl}(\lambda_i(x))$ requires $d+1$ subtractions, $d$ products, and one division, the result follows directly from (2.5), (2.8), and (2.7). □

Evaluating $\lambda_i$ in this way clearly has a computational cost of $O(d)$ for $d > 0$ and $O(1)$ for $d = 0$, so that computing all $\lambda_i(x)$ requires $O(d(n-d))$ operations for $0 < d < n$ and $O(n)$ operations for $d = 0$ and $d = n$. However, for $0 < d < n$ this can be improved by exploiting the fact that $\lambda_i(x)$ and $\lambda_{i+1}(x)$ have $d$ common factors in the denominator, which in turn suggests to first compute the "central" $\lambda_m(x)$ for $m = \left\lfloor \frac{n-d}{2} \right\rfloor$ as above and then the remaining $\lambda_i(x)$ iteratively as

$$\begin{aligned} \lambda_{i-1}(x) &= -\lambda_i(x)\frac{(x-x_{i+d})}{(x-x_{i-1})}, \qquad i = m, m-1, \ldots, 1, \\ \lambda_{i+1}(x) &= -\lambda_i(x)\frac{(x-x_i)}{(x-x_{i+1+d})}, \qquad i = m, m+1, \ldots, n-d-1. \end{aligned} \tag{3.23}$$

Computing all $\lambda_i(x)$ this way requires only $O(n)$ operations, but it comes at the price of a likely reduced precision.

**Lemma 3.6.** *For any $x \in \mathbb{F}$ and $x_0, \ldots, x_n \in \mathbb{F}$, there exist $\zeta_0, \ldots, \zeta_{n-d} \in \mathbb{R}$ with $|\zeta_0|, \ldots, |\zeta_{n-d}| \leq C\epsilon + O(\epsilon^2)$ for $C = 2n+4$, such that the $\lambda_i(x)$ in* (3.23) *satisfy*

$$\mathrm{fl}(\lambda_i(x)) = \lambda_i(x)(1+\zeta_i), \qquad i = 0, \ldots, n-d.$$

*Proof.* By Lemma 3.5, we know that $\mathrm{fl}(\lambda_m(x)) = \lambda_m(x)(1+\zeta_m)$ with $|\zeta_m| \leq (2d+2)\epsilon + O(\epsilon^2)$. Each step in (3.23) involves two subtractions, one division, and one product, and therefore introduces a perturbation of $(1+\delta)$ with $|\delta| \leq 4\epsilon + O(\epsilon^2)$, and these perturbations accumulate during the iteration. Since the number of steps is at most $(n-d+1)/2$, the overall perturbation for each $\lambda_i(x)$ is at most $(1+\zeta_i)$ with $|\zeta_i| \leq [(2d+2) + 4(n-d+1)/2]\epsilon + O(\epsilon^2)$. □

## 3.5 Backward stability

Similar to how we established the forward stability of both barycentric forms in a unified way in Section 3.3, we can prove the backward stability in general for the function $r$ in (3.8) and then derive upper bounds on the perturbation of the data for both forms as special cases.

**Theorem 3.7.** *Suppose that there exist $\alpha_0, \ldots, \alpha_n \in \mathbb{R}$ with*

$$\mathrm{fl}(a_i(x)) = a_i(x)(1+\alpha_i), \qquad |\alpha_i| \leq A\epsilon + O(\epsilon^2), \qquad i = 0, \ldots, n$$

*and $\beta_0, \ldots, \beta_m \in \mathbb{R}$ with*

$$\mathrm{fl}(b_j(x)) = b_j(x)(1+\beta_j), \qquad |\beta_j| \leq B\epsilon + O(\epsilon^2), \qquad j = 0, \ldots, m$$

*for some constants A and B. Then there exists for any $Y_n \in \mathbb{F}^{n+1}$ some perturbed $\hat{Y}_n \in \mathbb{F}^{n+1}$ with*

$$\frac{\|\hat{Y}_n - Y_n\|_\infty}{\|Y_n\|_\infty} \leq (n+2+A)\epsilon + (m+B)\max_x \beta(x)\epsilon + O(\epsilon^2),$$

*for $\epsilon$ small enough, such that the numerical evaluation of $r$ in* (3.8) *satisfies* $\mathrm{fl}(r(x; Y_n)) = r(x; \hat{Y}_n)$.

*Proof.* Starting from (3.13), with the $\eta_i$ and $\mu_j$ satisfying (3.12), we get, again with the help of (2.6),

$$\begin{aligned}
\mathrm{fl}(r(x;X_n,Y_n)) &= \frac{\sum_{i=0}^n a_i(x)y_i(1+\eta_i)}{\sum_{j=0}^m b_j(x)+\sum_{j=0}^m b_j(x)\mu_j} \\
&= \frac{\sum_{i=0}^n a_i(x)y_i(1+\eta_i)}{\sum_{j=0}^m b_j(x)\Big(1+\frac{\sum_{j=0}^m b_j(x)\mu_j}{\sum_{j=0}^m b_j(x)}\Big)} \\
&= \frac{\sum_{i=0}^n a_i(x)y_i(1+\eta_i)}{\sum_{j=0}^m b_j(x)}\Big(1-\frac{\sum_{j=0}^m b_j(x)\mu_j}{\sum_{j=0}^m b_j(x)}+O(\epsilon^2)\Big) \\
&= \frac{\sum_{i=0}^n a_i(x)\hat{y}_i}{\sum_{j=0}^m b_j(x)} = r(x;X_n,\hat{Y}_n),
\end{aligned}$$

where

$$\hat{y}_i = y_i(1+\eta_i)\Big(1-\frac{\sum_{j=0}^m b_j(x)\mu_j}{\sum_{j=0}^m b_j(x)}+O(\epsilon^2)\Big), \qquad i=0,\dots,n.$$

By (3.15), this means that there exist some $\xi_0,\dots,\xi_n\in\mathbb{R}$ with $|\xi_0|,\dots,|\xi_n|\leq (m+B)\max_x\beta(x)\epsilon+O(\epsilon^2)$, such that

$$\hat{y}_i = y_i(1+\eta_i)(1+\xi_i), \qquad i=0,\dots,n,$$

and by (2.8) and (3.12) we can further assume the existence of some $\varphi_0,\dots,\varphi_n$ with $|\varphi_0|,\dots,|\varphi_n|\leq (n+2+A)\epsilon+(m+B)\max_x\beta(x)\epsilon+O(\epsilon^2)$, such that

$$\hat{y}_i = y_i(1+\varphi_i), \qquad i=0,\dots,n.$$

The statement then follows directly from

$$\begin{aligned}
\|\hat{Y}_n-Y_n\|_\infty &= \max_{i=0,\dots,n}|\hat{y}_i-y_i| = \max_{i=0,\dots,n}|y_i\varphi_i| \\
&\leq \max_{i=0,\dots,n}|y_i|\max_{i=0,\dots,n}|\varphi_i| = \|Y_n\|_\infty\max_{i=0,\dots,n}|\varphi_i|.
\end{aligned}$$

□

The special cases of Theorem 3.7 for the two different forms of the barycentric rational interpolant then follow with the same reasoning as in Section 3.3.

**Corollary 3.8.** *Assume that the weights $\gamma_0,\dots,\gamma_n$ can be computed as in* (3.16)*. Then there exists for any $Y_n\in\mathbb{F}^{n+1}$ some perturbed $\hat{Y}_n\in\mathbb{F}^{n+1}$ with*

$$\frac{\|\hat{Y}_n-Y_n\|_\infty}{\|Y_n\|_\infty}\leq (n+4+W)\epsilon+(n+2+W)\max_x\Lambda_n(x;X_n)\epsilon+O(\epsilon^2),$$

*for $\epsilon$ small enough, such that the second barycentric form in* (3.3) *satisfies* $\mathrm{fl}(r(x;X_n,Y_n))=r(x;X_n,\hat{Y}_n)$.

**Corollary 3.9.** *Assume that the weights $\gamma_0,\dots,\gamma_n$ can be computed as in* (3.16) *and the values $\lambda_0(x),\dots,\lambda_{n-d}(x)$ as in* (3.19)*. Then there exists for any $Y_n\in\mathbb{F}^{n+1}$ some perturbed $\hat{Y}_n\in\mathbb{F}^{n+1}$ with*

$$\frac{\|\hat{Y}_n-Y_n\|_\infty}{\|Y_n\|_\infty}\leq (n+4+W)\epsilon+(n-d+C)\max_x\Gamma_d(x;X_n)\epsilon+O(\epsilon^2),$$

*for $\epsilon$ small enough, such that the first barycentric form in* (3.6) *satisfies* $\mathrm{fl}(r(x;X_n,Y_n)) = r(x;X_n,\hat{Y}_n)$.

## 3.6 Upper bound for $\Gamma_d$

In case of the first barycentric form, the bounds for the forward and backward stability depend on the function $\Gamma_d$ in (3.18), and we still need to show that this function is bounded from above. Note that $\Gamma_0 = \Lambda_n$, because $\gamma_i = (-1)^i$ and $\lambda_i = (-1)^i/(x-x_i)$ if $d = 0$, so that the bound for the Lebesgue constant of Berrut's interpolant [14] also holds for $\Gamma_d$ in this case. In the following, we therefore assume $d \geq 1$ and define

$$N(x) = \sum_{i=0}^{n-d} |\lambda_i(x)| \qquad \text{and} \qquad D(x) = \left| \sum_{i=0}^{n-d} \lambda_i(x) \right|,$$

so that $\Gamma_d(x;X_n) = N(x)/D(x)$. We further assume that $x \in (x_k, x_{k+1})$ for some $k \in \{0,\dots,n-1\}$. It then follows from the definition in (3.7) that all $\lambda_i$ with index in

$$I_3 = I \cap \{k-d+1,\dots,k\},$$

where $I = \{0,\dots,n-d\}$, have the same sign as $(-1)^{k+d}$ and that the sign alternates for decreasing indices "to the left" and increasing indices "to the right" of $I_3$. More precisely, the $\lambda_i$ with index in

$$I_2 = I \cap \{k-d, k-d-2, \dots\} \qquad \text{or} \qquad I_4 = I \cap \{k+1, k+3, \dots\}$$

have the same sign as the ones with index in $I_3$, while the sign is opposite, if the index is in

$$I_1 = I \cap \{k-d-1, k-d-3, \dots\} \qquad \text{or} \qquad I_5 = I \cap \{k+2, k+4, \dots\}.$$

Without loss of generality, we assume that the $\lambda_i$ are positive for $i \in I_2, I_3, I_4$ and negative for $i \in I_1, I_5$, since multiplying all $\lambda_i$ with a common constant does not change $\Gamma_d$. Letting

$$S_j = \sum_{i \in I_j} \lambda_i(x), \qquad j = 1,\dots,5,$$

we then find that

$$N(x) = -S_1 + S_2 + S_3 + S_4 - S_5 \qquad \text{and} \qquad D(x) = S_1 + S_2 + S_3 + S_4 + S_5. \tag{3.24}$$

To bound $\Gamma_d$, we need to bound the sum of the negative $\lambda_i$ with $i \in I_1, I_5$ as well as the $\lambda_i$ with $i \in I_3$, which in turn requires to first bound the terms $x - x_i$. To this end, let $h_i = x_{i+1} - x_i$ for $i \in \{0,\dots,n-1\}$ and define

$$h_{\min} = \min\{h_0,\dots,h_{n-1}\} \qquad \text{and} \qquad h_{\max} = \max\{h_0,\dots,h_{n-1}\}.$$

**Proposition 3.10.** *For any $i \in \{0,\dots,n\}$, the distance between $x \in (x_k, x_{k+1})$ and $x_i$ is bounded as*

$$\frac{h_{\min}}{2}(1+t+2(k-i)) \leq x - x_i \leq \frac{h_{\max}}{2}(1+t+2(k-i)), \qquad i \leq k,$$

$$\frac{h_{\min}}{2}(1-t+2(i-k-1)) \leq x_i - x \leq \frac{h_{\max}}{2}(1-t+2(i-k-1)), \qquad i \geq k+1,$$

*where $t = 2(x-x_k)/h_k - 1 \in (-1,1)$.*

*Proof.* The statement follows directly by noting that

$$x = x_k + \frac{h_k}{2}(1+t) = x_{k+1} + \frac{h_k}{2}(1-t)$$

for the given $t$ and because $h_{\min} \le h_i \le h_{\max}$ for any $i \in \{0,\dots,n-1\}$. □

For bounding the negative $\lambda_i$ from above, it turns out to be useful to consider them in pairs, with indices from $I_1$ and $I_5$ at the same "distance" from $I_3$.

**Lemma 3.11.** *For any $j \in \mathbb{N}$ and $x \in (x_k, x_{k+1})$,*

$$-\lambda_{k-d-2j+1}(x) - \lambda_{k+2j}(x) \le \left(\frac{1}{h_{\min}}\right)^{d+1}\left(\frac{1}{\prod_{m=0}^{d}(2j+m)} + \frac{1}{\prod_{m=0}^{d}(2j-1+m)}\right),$$

*where we set $\lambda_i(x) = 0$ for any $i \notin I$.*

*Proof.* Since the denominator of $\lambda_{k-d-2j+1}(x)$, for $k-d-2j+1 \ge 0$, contains the terms $x - x_{k-2j+1-m}$ for $m = 0,\dots,d$, it follows from Proposition 3.10 that

$$\frac{-1}{\lambda_{k-d-2j+1}(x)} \ge \left(\frac{h_{\min}}{2}\right)^{d+1} \prod_{m=0}^{d}(4j+2m-1+t),$$

with $t \in (-1,1)$ and likewise

$$\frac{-1}{\lambda_{k+2j}(x)} \ge \left(\frac{h_{\min}}{2}\right)^{d+1} \prod_{m=0}^{d}(4j+2m-1-t),$$

for $k+2j \le n-d$. Combining both bounds, we get

$$-\lambda_{k-d-2j+1}(x) - \lambda_{k+2j}(x) \le \left(\frac{2}{h_{\min}}\right)^{d+1} g(t),$$

where

$$g(t) = \frac{1}{\prod_{m=0}^{d}(4j+2m-1+t)} + \frac{1}{\prod_{m=0}^{d}(4j+2m-1-t)}.$$

As $g$ is clearly even and

$$g(1) = \frac{1}{2^{d+1}}\left(\frac{1}{\prod_{m=0}^{d}(2j+m)} + \frac{1}{\prod_{m=0}^{d}(2j-1+m)}\right),$$

it remains to show that $g(t) \le g(1)$ for $t \in [0,1]$. To this end, note that $g(t) = p(t)/q(t)$ for

$$p(t) = \prod_{m=0}^{d}(4j+2m-1+t) + \prod_{m=0}^{d}(4j+2m-1-t)$$

and

$$q(t) = \prod_{m=0}^{d}\left((4j+2m-1)^2 - t^2\right).$$

By the product rule,

$$p'(t) = \sum_{l=0}^{d} \Bigg( \prod_{m=0,\, m\neq l}^{d} (4j+2m-1+t) - \prod_{m=0,\, m\neq l}^{d} (4j+2m-1-t) \Bigg)$$

and

$$q'(t) = -2t \sum_{l=0}^{d} \prod_{m=0,\, m\neq l}^{d} \big( (4j+2m-1)^2 - t^2 \big).$$

For $t \in [0,1]$, we observe that $p(t) > 0$, $q(t) > 0$, $p'(t) \geq 0$, and $q'(t) \leq 0$, hence

$$p'(t)q(t) \geq 0 \geq p(t)q'(t),$$

and it follows from the quotient rule that $g$ is monotonically increasing over $[0,1]$. □

Next, let us bound the $\lambda_i$ with indices in $I_3$ from below.

**Lemma 3.12.** *For any $i \in I_3$ and $x \in (x_k, x_{k+1})$,*

$$\lambda_i(x) \geq \left( \frac{1}{h_{\max}} \right)^{d+1} \frac{4}{d!}. \tag{3.25}$$

*Proof.* Since the denominator of $\lambda_i(x)$, for $i \in I_3$, contains the factors $x - x_{k-m}$ for $m = 0, \ldots, k-i$ and the factors $x_{k+1+l} - x$ for $l = 0, \ldots, i+d-k-1$, we conclude from Proposition 3.10 that

$$\begin{aligned}
\frac{1}{\lambda_i(x)} &\leq \left( \frac{h_{\max}}{2} \right)^{d+1} \prod_{m=0}^{k-i} (1+t+2m) \prod_{l=0}^{i+d-k-1} (1-t+2l) \\
&= \left( \frac{h_{\max}}{2} \right)^{d+1} (1-t^2) \prod_{m=1}^{k-i} (1+t+2m) \prod_{l=1}^{i+d-k-1} (1-t+2l) \\
&\leq \left( \frac{h_{\max}}{2} \right)^{d+1} \prod_{m=1}^{k-i} (2+2m) \prod_{l=1}^{i+d-k-1} (2+2l) \\
&= \left( \frac{h_{\max}}{2} \right)^{d+1} 2^{d-1} (k-i+1)!(i+d-k)!,
\end{aligned}$$

and the statement then follows, because $a!b! \leq (a+b-1)!$ for any positive integers $a$ and $b$. □

We are now ready to derive a general upper bound on the function $\Gamma_d$ in (3.18), which turns out to depend on $d$ and the *mesh ratio*

$$\mu = \frac{h_{\max}}{h_{\min}}.$$

**Theorem 3.13.** *If $d \geq 1$, then*

$$\Gamma_d(x; X_n) \leq 1 + \frac{\mu^{d+1}}{2d} \tag{3.26}$$

*for any set of ascending interpolation nodes $X_n = (x_0, \ldots, x_n) \in \mathbb{R}^{n+1}$ and any $x \in [x_0, x_n]$.*

*Proof.* If $x = x_k$ for some $k \in \{0,\dots,n\}$, then, after multiplying both $N(x)$ and $D(x)$ by $\prod_{i=0}^{n}|x-x_i|$, we find that $\Gamma_d(x;X_n)=1$, which is clearly smaller than the upper bound in (3.26). Otherwise, there exists some $k \in \{0,\dots,n-1\}$, such that $x \in (x_k, x_{k+1})$, and it follows from Lemma 3.11 that

$$\begin{aligned}
-S_1 - S_5 &\le \sum_{j=1}^{\infty}\left(\frac{1}{h_{\min}}\right)^{d+1}\left(\frac{1}{\prod_{m=0}^{d}(2j+m)} + \frac{1}{\prod_{m=0}^{d}(2j-1+m)}\right)\\
&= \left(\frac{1}{h_{\min}}\right)^{d+1}\sum_{j=1}^{\infty}\prod_{m=0}^{d}\frac{1}{j+m}\\
&= \left(\frac{1}{h_{\min}}\right)^{d+1}\frac{1}{d\cdot d!},
\end{aligned}$$

where the sum of the series can be found in [15, p. 464]. Together with Lemma 3.12, this implies

$$\frac{-2(S_1+S_5)}{\lambda_i(x)} \le \frac{\mu^{d+1}}{2d}$$

for any $i \in I_3$. As $\lambda_i(x) \le S_3 \le D(x)$ and $N(x)-D(x) = -2(S_1+S_5)$, we conclude that

$$N(x)-D(x) \le \frac{\mu^{d+1}}{2d}D(x),$$

and the statement then follows immediately. □

In the case of equidistant nodes, when the mesh ratio is $\mu = 1$, the upper bound in (3.26) is simply $1 + 1/(2d)$, so it becomes smaller as $d$ grows. For other nodes, $\mu$ may depend on $n$, which may result in very large upper bounds. For example, in the case of Chebyshev nodes, one can show that $\mu$ grows asymptotically linear in $n$. However, our numerical experiments suggest that the function $\Gamma_d$ is always small, and we believe that the upper bound in Theorem 3.13 can be improved significantly in future work.

## 3.7 Numerical experiments

We performed numerous experiments to verify the results proven in the previous sections numerically and report some representative results below. In particular, we analyze the various algorithms that implement the first barycentric form (3.6) both in terms of stability and computational cost (Section 3.7.1) and focus on the comparison between the first and the second form for some example where $\Lambda_n \gg \Gamma_d$ (Section 3.7.2), as well as in scenarios where evaluation points are close to a root of the interpolant or a node (Section 3.7.3).

Our experimental platform is a Windows 10 laptop with 1.8 GHz Intel Core i7-10510U processor and 16 GB RAM, and we implemented all algorithms in *C++*. In what follows, the 'exact' values were computed in multiple-precision (1024 bit) floating-point arithmetic using the *MPFR* library [33], while all other values were computed in standard double precision. Moreover, we took care of providing all input data (interpolation nodes, data values, and evaluation points) in double precision, so that they do not cause any additional error.

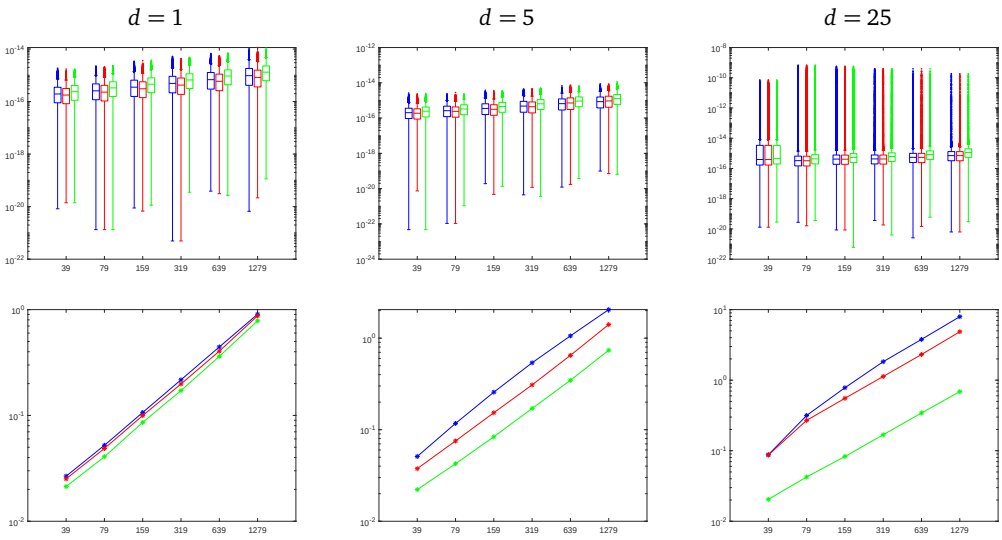


Figure 3.1. Distribution of the relative forward errors of the first barycentric form for equidistant nodes at 50,000 random evaluation points (top) and overall running time in seconds (bottom), both on a logarithmic scale, for different $n$ and three choices of $d$ (left, middle, right), using the standard algorithm (blue), Camargo's algorithm (red), and our efficient variant of the standard algorithm (green).

### 3.7.1 Comparison of algorithms for the first barycentric form

For the first example, we consider the case of $n+1$ interpolation nodes $x_i = \mathrm{fl}(t_i) \in [-1,1]$ for $i = 0,\dots,n$, derived from the equidistant nodes $t_i = 2i/n - 1$ by rounding them to double precision, and associated (rounded) data $y_i = \mathrm{fl}(f(t_i))$, sampled from the test function

$$f(x) = \tfrac{3}{4}e^{-\frac{(9x-2)^2}{4}} + \tfrac{3}{4}e^{-\frac{(9x+1)^2}{49}} + \tfrac{1}{2}e^{-\frac{(9x-7)^2}{4}} + \tfrac{1}{5}e^{-(9x-4)^2},$$

and we compare three ways of evaluating the first barycentric form in (3.6), which differ in the way the denominator is computed.

The first algorithm simply evaluates the functions $\lambda_i$ as in (3.7), leading to the error mentioned in Lemma 3.5 and then sums up these values to get the denominator. The second algorithm by Camargo [18, Section 4.1] instead increases the stability of the summation by first computing sums of pairs of $\lambda_i$'s such that all these sums have the same sign. The third algorithm implements our iterative strategy in (3.23) before taking the sum, which is more efficient than the first algorithm, but less precise (cf. Lemma 3.6). All three algorithms compute the numerator of the first barycentric form in the same way, first dividing the weights $\gamma_i$ by $x - x_i$, then multiplying the results by $y_i$, and finally summing up these values. The $\gamma_i$ themselves are precomputed with the pyramid algorithm in (3.21) and (3.22). Note that, although the weights for equidistant nodes are integer multiples of each other in theory [30], they do not have this property in this example, because the nodes $x_i$ are not exactly equidistant, because of the rounding.

To compare these three algorithms, we used them to evaluate the barycentric rational interpolant with weights in (3.5) for $d \in \{1,5,25\}$ and an increasing number of interpolation

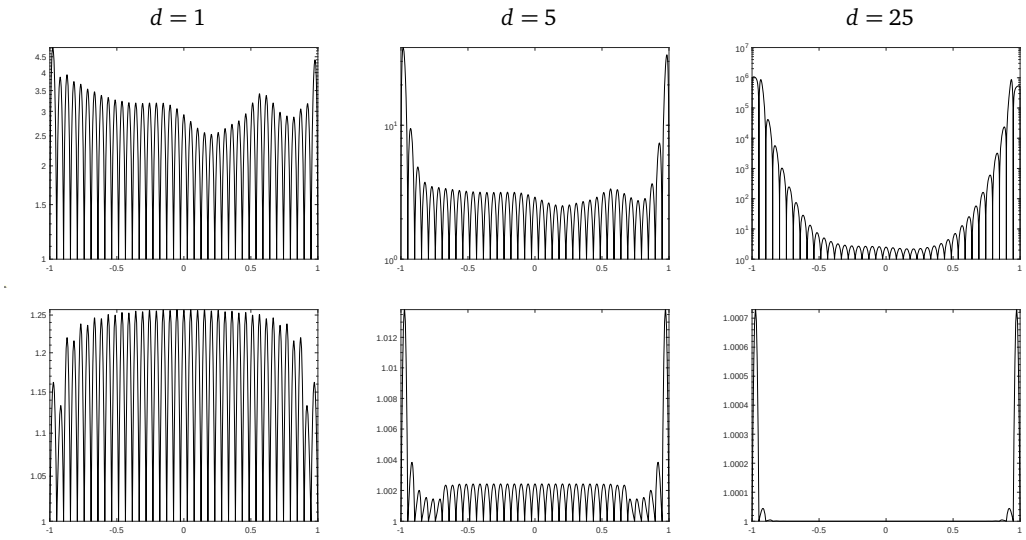


Figure 3.2. Plots of $\kappa(x; X_n, Y_n)$ (top) and $\Gamma_d(x; X_n)$ (bottom) for equidistant nodes and $x \in [-1, 1]$, both on a logarithmic scale, for $n = 39$ and three choices of $d$ (left, middle, right).

nodes, $n \in \{39, 79, 159, 319, 639, 1279\}$, at 50,000 random points from $[-1 + 10^3\epsilon, 1 - 10^3\epsilon] \setminus \{x_0, \ldots, x_n\}$. Figure 3.1 shows the corresponding running times and the distribution of the relative forward error of the computed values. For the latter, we chose a box plot, where the bottom and top of each box represent the interquartile range, from the first to the third quartile, and the line in the middle shows the median of the relative forward errors. The whiskers range from the minimum to the maximum, excluding those values that are more than 1.5 times the interquartile range greater than the third quartile, which are instead considered outliers and shown as isolated points.

On the one hand, we observe that Camargo's algorithm beats the standard algorithm in terms of running time, but that our efficient algorithm is the fastest, especially as $d$ grows, because its time complexity does not depend on $d$. On the other hand, our efficient algorithm is less precise than the standard algorithm, as predicted by Lemma 3.6 and Camargo's algorithm gives the smallest errors, except for $d = 5$ and $n \in \{639, 1279\}$. Nevertheless, the computations confirm the forward stability for all three algorithms. For Camargo's algorithm, this follows from the backward stability, which is proven in [18], and for the other two algorithms it is implied by Corollary 3.3 and Lemmas 3.4–3.6.

The rather large errors of the outliers in the case $d = 25$ can be explained by the behaviour of the componentwise relative condition number $\kappa$, shown in Figure 3.2. While $\kappa$ is small for all $x \in [-1, 1]$ in the case of $d = 1$, it starts to grow considerably close to the end points of the interval as $d$ grows, up to $10^6$ for $n = 39$ and $10^8$ for $n = 1279$ in the case of $d = 25$, and so does the upper bound on the relative forward error in (3.20). While this upper bound also depends on $\Gamma_d$, Figure 3.2 shows that this function is always small and its maximum even decreases as $d$ grows, independently of $n$. This is in agreement with Theorem 3.13, because $\mu \approx 1$ in this example. The fact that the maximum error still seems to decrease for $d = 25$ as $n$ increases is simply due to the fact that the 50,000 sample points are not sufficiently many to "catch" the worst case, because the region near the endpoints where $\kappa$ grows rapidly actually

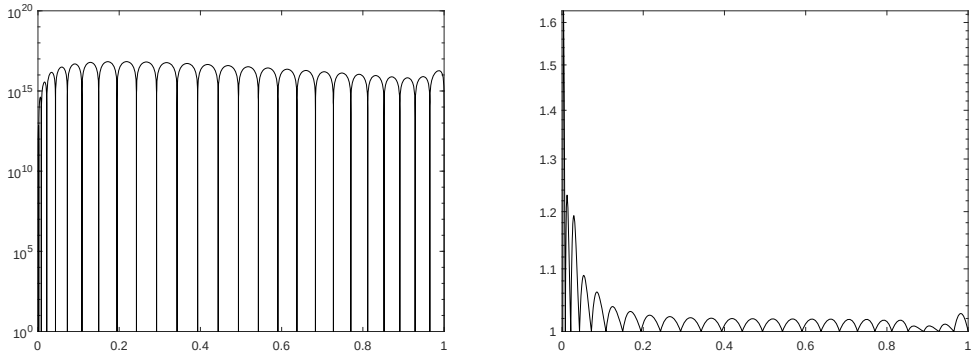


Figure 3.3. Plots of $\Lambda_n(x;X_n)$ (left) and $\Gamma_d(x;X_n)$ (right) for a non-regular distribution of nodes and $x \in [0,1]$, both on a logarithmic scale.

shrinks as $n$ grows.

Of course, it is also possible to evaluate the rational interpolant using the second barycentric form (3.3), which is actually the best choice for this example, giving relative forward errors that are similar to the ones of the standard algorithm for the first barycentric form, but being roughly twice as fast as the efficient algorithm, both of which is not surprising. Regarding the efficiency, the second form is superior, because the denominator can be computed "on-the-fly" at almost no extra cost during the evaluation of the numerator. As for the error, we note that $\Lambda_n$ is much smaller than $\kappa$ for the nodes used in this example [48] and so the upper bound in (3.17) is dominated by $\kappa$, exactly as the upper bound in (3.20). However, for non-uniform nodes, the situation can be quite different, as the next example shows.

### 3.7.2 Worst-case comparison of first and second barycentric form

The aim of the second example is to compare the standard algorithm for the first barycentric form in (3.6), as described in Section 3.7.1, with a straightforward implementation of the second barycentric form, following the formula in (3.3). The weights $\gamma_i$ are again precomputed with the pyramid algorithm.

We consider $n = 29$, $d = 3$, and interpolation nodes $x_i = \mathrm{fl}(t_i) \in [0,1]$ for $i = 0,\dots,n$, obtained by rounding to double precision the values $t_i = F(\tau_i)$, where

$$F(t) = \begin{cases} 0, & t = 0, \\ e^{1-\frac{1}{t}}, & t \in (0,1] \end{cases} \tag{3.27}$$

and $\tau_i = i/n$. We choose these nodes, because the functions $\Lambda_n$ and $\Gamma_d$ behave completely differently in this case, as shown in Figure 3.3. While $\Lambda_n$ reaches huge values, up to $10^{17}$, $\Gamma_d$ is small (even though the latter is not guaranteed by Theorem 3.13). Hence, the upper bounds in Corollaries 3.2 and 3.3 suggest that we may see a big difference in the forward stability of the first and the second barycentric form, if we choose the data such that the condition number $\kappa$ is small.

One such choice, which is also presented by Higham for the case $d = n$ with equidistant

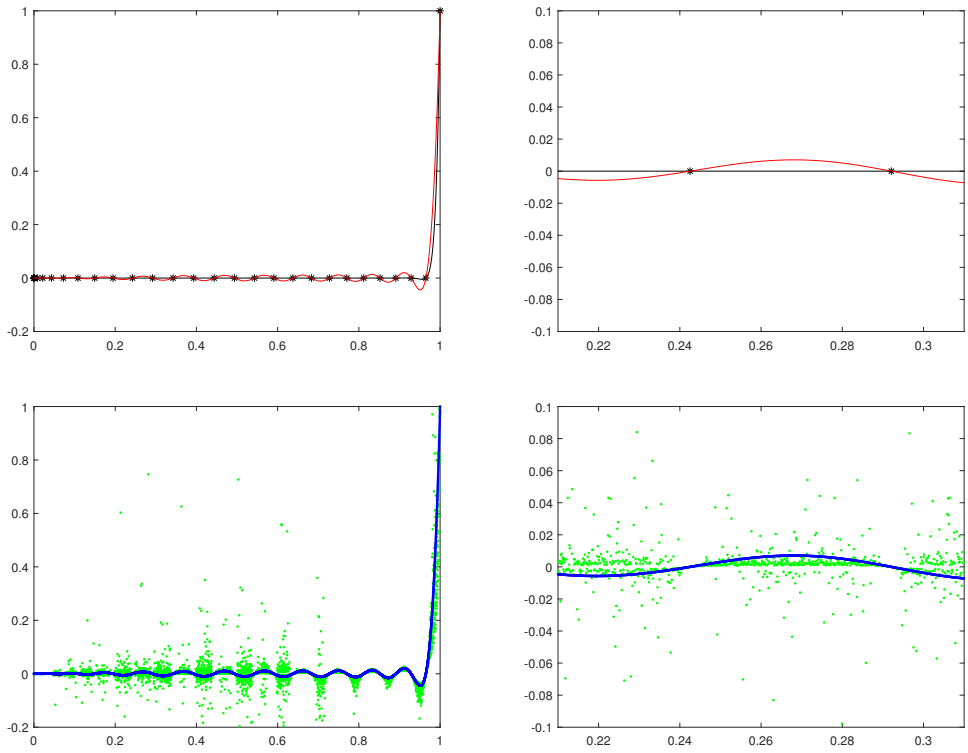

Figure 3.4. Plots of $l_n(x)$ (black) and the barycentric rational interpolant $r(x)$ for $d = 3$ (red) for non-regularly distributed interpolation nodes over the whole interval $[0,1]$ (top left) and a close-up view over $[0.21, 0.31]$ (top right). Evaluating $r(x)$ at $10,000$ equidistant evaluation points in $[10^3\epsilon, 1-10^3\epsilon]$ with the standard implementations of the first (blue dots) and the second (green dots) barycentric form shows that the first form is stable, while the second form is not (bottom).

nodes [44], is to take the data

$$y_i = \begin{cases} 0, & i = 0, \ldots, n-1, \\ 1, & i = n, \end{cases}$$

which can be interpreted as having been sampled from the $n$-th Lagrange basis polynomial $l_n(x) = \prod_{i=0}^{n-1} \frac{x-x_i}{x_n-x_i}$, that is, $y_i = \mathrm{fl}(l_n(t_i))$. For this data, we know that $\kappa(x; X_n, Y_n) = 1$ for all $x \in [0,1]$, so the upper bounds on the relative forward errors in (3.17) and (3.20) are dominated by $\Lambda_n$ and $\Gamma_d$, respectively. Consequently, as shown in Figure 3.4, the barycentric rational interpolant is reproduced faithfully by the first form, but not by the second, because the relative forward error for the first form is on the order of $\epsilon$, while it can be on the order of 1 for the second form; see Figure 3.5 (left).

However, the opposite may happen as well. If we consider the data

$$y_i = 1, \qquad i = 0, \ldots, n,$$

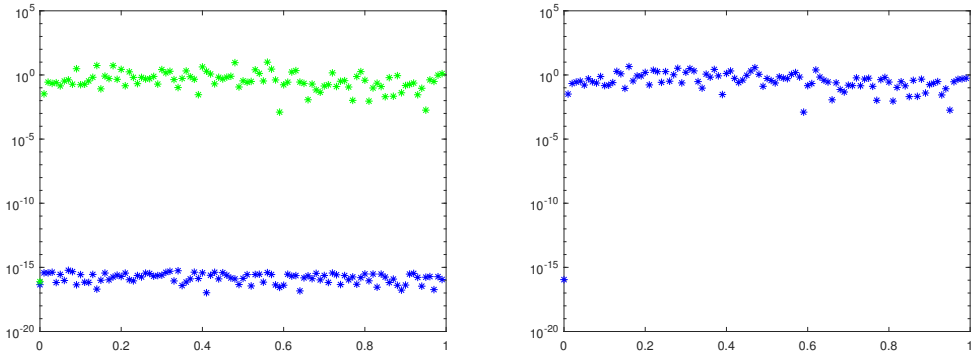


Figure 3.5. Plot of relative forward errors of the first (blue) and second (green) barycentric form for a non-regular distribution of nodes at 100 equidistant evaluation points in $[10^3\epsilon, 1-10^3\epsilon]$ with data sampled from the $n$-th Lagrange basis polynomial (left) and the constant one function (right). Since both plots are on a logarithmic scale and the second form is exact in the latter case, the corresponding errors are missing in the plot on the right.

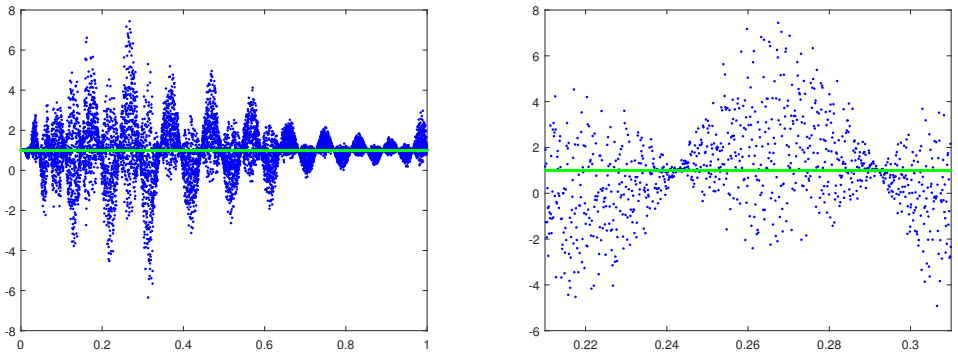


Figure 3.6. Even though the barycentric rational interpolant of the constant one function for non-regularly distributed interpolation nodes is simply $r(x) = 1$, evaluating it at $10,000$ equidistant evaluation points in $[10^3\epsilon, 1-10^3\epsilon]$ shows that the second form (green dots) is stable, while the first form (blue dots) is not.

sampled from the constant one function, then $\kappa = \Lambda_n$, and the upper bounds in (3.17) and (3.20) suggest that both forms can be unstable, even though the barycentric rational interpolant is simply $r(x) = 1$. Figure 3.5 (right) and Figure 3.6 confirm that this is indeed the case for the first barycentric form. However, the second barycentric form is perfectly stable, because the numerator and the denominator in (3.3) are identical and cancel out to always give the correct function value 1.

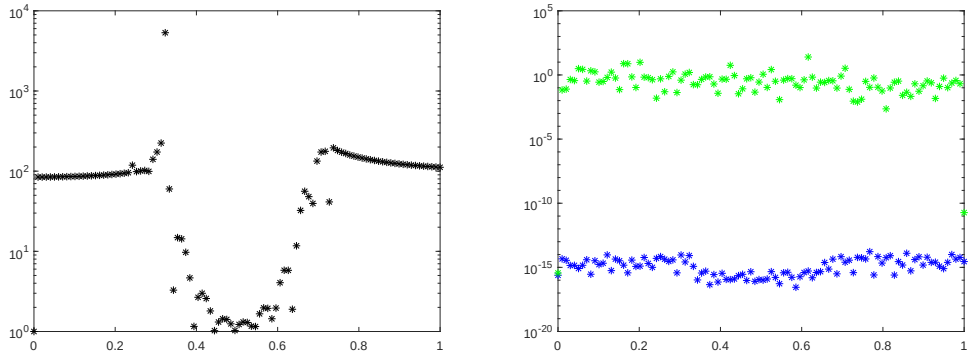

Figure 3.7. Plots of $\kappa(x; X_n, Y_n)$ (left) and relative forward errors (right) of the first (blue) and second (green) barycentric form for 100 equidistant evaluation points in $[10^3\epsilon, 1-10^3\epsilon]$ and data sampled from $f(x)$, both on a logarithmic scale.

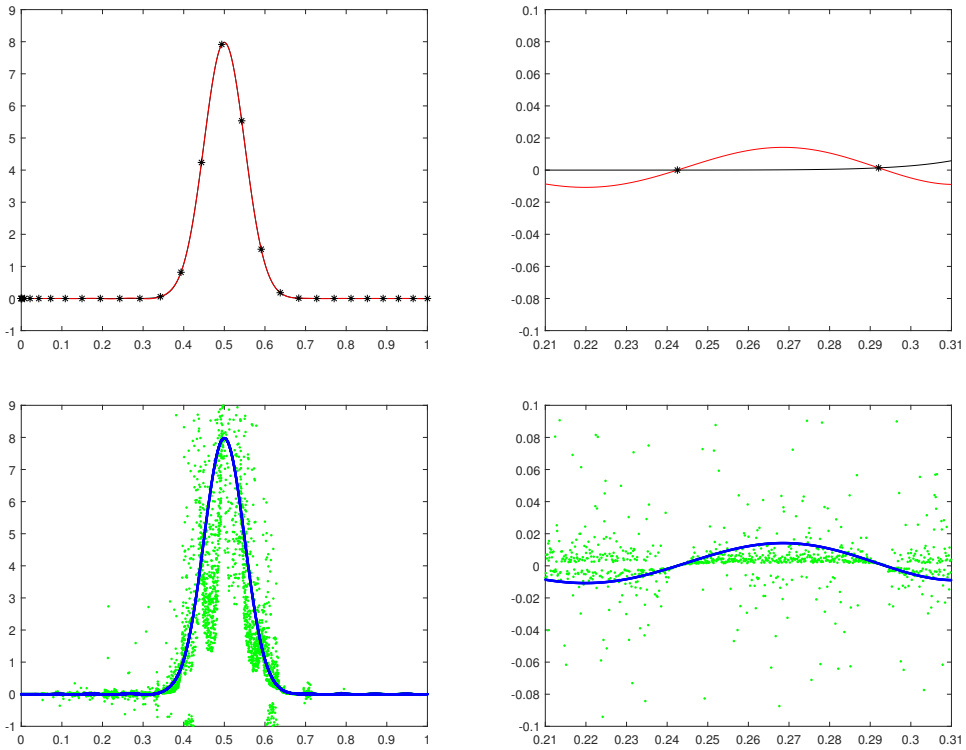

Figure 3.8. Same as Figure 3.1, but for data sampled from $f(x)$ (black).

### 3.7.3 Evaluation close to roots and nodes

Let us consider another example similar to the first one discussed in Section 3.7.2. In this case, we again have $n = 29$ and $d = 3$, using the same interpolation nodes generated by the

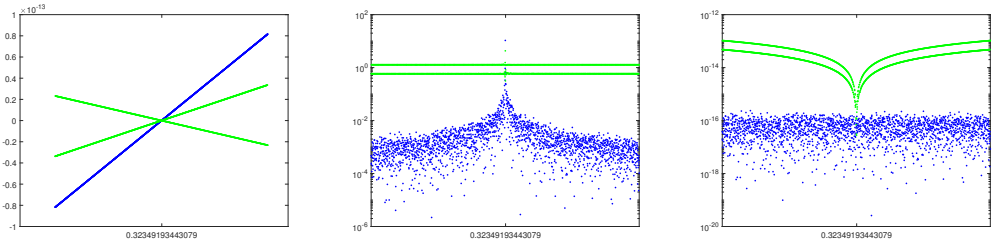


Figure 3.9. Evaluation of $r(x)$ in Figure 3.8 at the 2000 closest double floating-point numbers to the root $x = 0.32349193443079$ with the standard implementations of the first (blue dots) and the second (green dots) barycentric form (left) and plots of the relative (middle) and absolute (right) forward errors on a logarithmic scale.

function $F(t)$ in (3.27), but we now associate the data $y_i = \mathrm{fl}(f(t_i))$ sampled from the normal distribution

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2} \tag{3.28}$$

with $\mu = 0.5$ and $\sigma = 0.05$. The componentwise relative condition number $\kappa$ is rather well-behaved for this example. In fact, Figure 3.7 shows that $\kappa$ does not get bigger than $10^2$ for almost all the 100 equidistant evaluation points that we used, except for those that are too close to the zeroes of the interpolant, where $\kappa$ has poles, by definition. Thus, the condition number cannot dominate the upper bounds on the relative forward error in (3.17) and (3.20) and we expect that both algorithms will behave similarly to the case shown in Figure 3.4. Indeed, it turns out that the first barycentric form is forward stable, while the second is not. The latter is evident from the relative forward errors in Figure 3.7 and can also be seen in Figure 3.8, which shows the actual function values of the interpolant computed with both algorithms. Note that we clipped the bottom-left plot to the range $[-1, 9]$, since the minimal and maximal values computed with the second form are approximately $-3572.5$ and $1457.2$, respectively.

We now focus on another aspect of this example, that is the fact that the exact interpolant $r(x)$ has roots within the domain $[0, 1]$. Consequently, the condition number and the relative forward error are undefined at those points and extremely large nearby. But does this imply that the methods are unstable in these regions? Not necessarily. In fact, the appropriate measure to consider in such cases is the absolute error rather than the relative error for more reliable results. Figure 3.9 illustrates the difference between these two types of error by examining the 2000 double-precision floating-point numbers closest to the root $x = 0.32349193443079$. The relative error suggests that both the first and second barycentric forms are unstable. However, the absolute error indicates that the first form is perfectly stable, while the second form is unstable, except when extremely close to the root. Additionally, Figure 3.9 (left) shows the oscillatory behavior of the second barycentric form, which explains its unusual error patterns. This analysis indicates that a large relative error near a root does not necessarily reflect an algorithm's instability and emphasizes the importance of choosing the appropriate error measure for a reliable assessment.

Finally, we also investigate the behavior of both algorithms close to a node, as the division by $x - x_i$ may lead to overflow or underflow errors. In general, this problem is addressed by setting $r(x) = y_i$ whenever $|x - x_i|$ is below a certain threshold. Nevertheless, double-precision floating-point arithmetic usually manages calculations near a node without signifi-

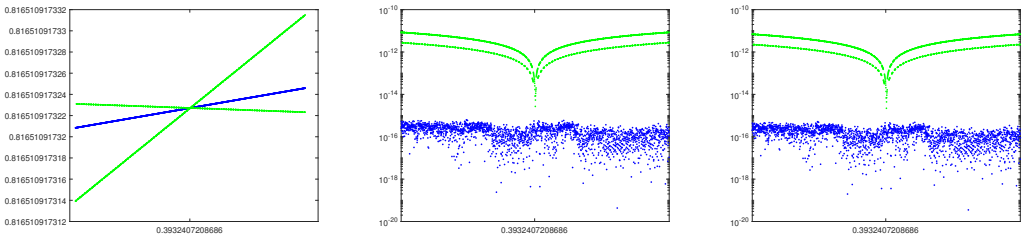


Figure 3.10. Same as Figure 3.9, but for the node $x_{15} = 0.393240720868598$.

cant problems. In fact, Figure 3.10 shows the barycentric interpolants and the errors for the 2000 double-precision floating-point numbers closest to the node $x_{15} = 0.393240720868598$, excluding the node itself where we know that $\mathrm{fl}(r(x_{15})) = y_{15}$. The results indicate that the evaluations of $r(x)$ are well-defined for both algorithms, as well as the relative and absolute errors, which in this case give the same results as the exact interpolant is far from 0.

However, it is important to mention that, even if the divisions by $x - x_i$ are well-defined, overflow and underflow can still occur due to other operations, such as the product by the weights $\gamma_i$ or by the interpolation data $y_i$ in the numerator. This issue can still be resolved by rescaling the weights or shifting the values $y_i$. We will explore these solutions in detail in the next chapter (Section 4.4).

# Chapter 4

# A C++ class for robust linear barycentric rational interpolation

Barycentric rational interpolation is an efficient and numerically robust interpolation method that performs well even in the case of equidistant nodes where polynomial interpolation can fail badly. Because of these favourable properties, barycentric rational interpolation has recently become popular not only in mathematics, where it is used for the definition of quadrature rules [55, 6] and for solving PDEs [61, 59], but also in more applied contexts. For example, the method proposed by Floater and Hormann [30] is used in statistics [4], engineering [83], as well as in the medical [25], aerospace [58, 53], and chemical sciences [57, 56].

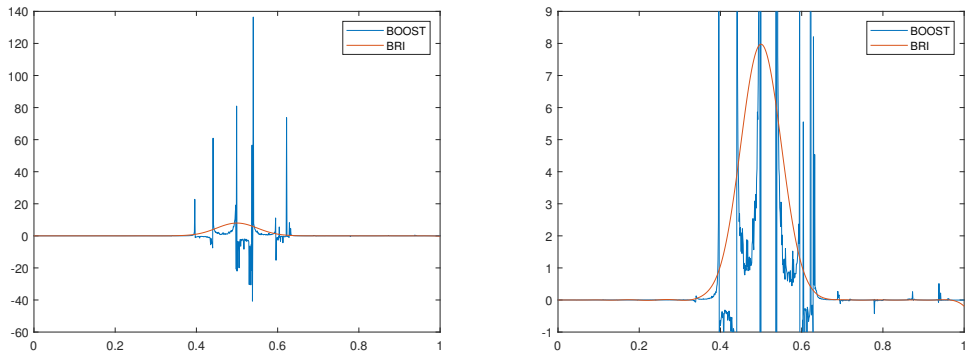


Figure 4.1. Plots of the barycentric rational interpolant obtained by using the same setting as in Figure 3.8 with both the BOOST and BRI libraries (left) and close-up view of the $y$-axis over the interval $[-1, 9]$ (right).

For this reason, there are already many available libraries [11, 1, 62, 77] that implement barycentric interpolation, but all of them simply use the second barycentric form in (3.3) without further considerations. This led us to develop a new library, the BRI library, that incorporates the theoretical findings presented in Chapter 3. In fact, if for example we implement the barycentric interpolant of Figure 3.8, that is, $n = 29$, $d = 3$, interpolation nodes generated by

**Algorithm 1** Second barycentric form

```
1: function SECOND(x)
2:     N := 0
3:     D := 0
4:     for i = 0, 1, ..., n do
5:         if x = x_i then
6:             return y_i
7:         z := γ_i/(x − x_i)
8:         N := N + z y_i
9:         D := D + z
10:    return N/D
```

**Algorithm 2** First barycentric form (standard version)

```
1: function FIRST_DEF(x)
2:     N := 0
3:     D := 0
4:     for i = 0, 1, ..., n do
5:         if x = x_i then
6:             return y_i
7:         N := N + γ_i/(x − x_i) y_i
8:     for i = 0, 1, ..., n − d do
9:         λ_i := (−1)^i
10:        for j = i, i + 1, ..., i + d do
11:            λ_i := λ_i/(x − x_j)
12:        D := D + λ_i
13:    return N/D
```

**Algorithm 3** First barycentric form (efficient version)

```
1: function FIRST_EFF(x)
2:     N := 0
3:     for i = 0, 1, ..., n do
4:         if x = x_i then
5:             return y_i
6:         N := N + γ_i/(x − x_i) y_i
7:     m := ⌊(n − d)/2⌋
8:     λ_m := (−1)^m
9:     for j = m, m + 1, ..., m + d do
10:        λ_m := λ_m/(x − x_j)
11:    D := λ_m
12:    for i = m, m − 1, ..., 1 do
13:        λ_{i−1} := −λ_i (x − x_{i+d})/(x − x_{i−1})
14:        D := D + λ_{i−1}
15:    for i = m, m − 1, ..., n − d − 1 do
16:        λ_{i+1} := −λ_i (x − x_i)/(x − x_{i+1+d})
17:        D := D + λ_{i+1}
18:    return N/D
```

the function $F(t)$ in (3.27), and associated data sampled from the normal distribution in (3.28), with both the BOOST [1] and the BRI libraries, then Figure 4.1 clearly demonstrates that, while the former produces an unstable result, the latter yields accurate outcomes. In the following chapter, we present the BRI library, describing its features and explaining how it integrates the theoretical insights related to numerical stability and efficiency.

## 4.1 Class overview

The BRI class arises from the idea of providing the user with a class that holds both variables and functions related to barycentric rational interpolation, with a simple and intuitive interface and in an efficient programming environment. This idea results in a `C++` class template, which comes with the advantage of having all input variables and function outputs defined as generic types. In addition to the basic data types available in `C++` , such as `float` or `double`, the BRI class supports arbitrary precision thanks to the compatibility with the Multiple Precision Floating-Point Reliable (MPFR) library [33][1] and the MPFR `C++` interface [45] [2]. The BRI class does not have any dependencies other than the `C++` standard libraries and, if arbitrary precision is used, the MPFR library.

To initialize a new instance of this class, the user has to specify the integer $d$, the nodes $X_n$, and the data $Y_n$, and optionally their type and the type of the output results, which are both considered to be `double` by default. The vectors $X_n$ and $Y_n$ are stored internally as private vectors, but the class provides the user with some functions to access or modify them. Moreover, since they can be passed in different ways, the class includes different constructors. In particular,

[1] All information about installing the MPFR library can be found at https://www.mpfr.org.

[2] More information can be found at https://github.com/advanpix/mpreal.

the user can pass both nodes and data by reference or they can be read from external files by specifying the filenames. Another constructor variant allows the user to pass one of the three keywords `UNIFORM`, `CHEBYSHEV`, or `EXTENDED_CHEBYSHEV`, together with the values $a$, $b$, and $n$. In this case, the class automatically generates $X_n$ as a vector of *uniform nodes*

$$x_i = a + (b-a)\frac{i}{n}, \qquad i = 0, 1, \ldots, n \tag{4.1}$$

or *Chebyshev nodes*

$$x_i = \frac{a+b}{2} - \frac{b-a}{2}\cos\frac{(2i+1)\pi}{2n+2}, \qquad i = 0, 1, \ldots, n \tag{4.2}$$

or *extended Chebyshev nodes*

$$x_i = \frac{a+b}{2} - \frac{b-a}{2}\cos\frac{(2i+1)\pi}{2n+2}\Big/\cos\frac{\pi}{2n+2}, \qquad i = 0, 1, \ldots, n$$

over the interval $[a, b]$. Of course, it would be possible to extend the class to allow for other predefined sets of interpolation nodes. As for the data $Y_n$, the user can also call the constructor with a pointer to a procedure `f`, which implements some function $f : \mathbb{R} \to \mathbb{R}$, so that the data is automatically generated as $f(X_n)$ by the class. After having defined these initial parameters, the class allows the user to modify them at any moment. Furthermore, the constructors create the vector $W_n = (\gamma_0, \ldots, \gamma_n)$ of barycentric weights (3.5), which get updated automatically whenever the nodes $X_n$ are changed. Like nodes and data, the weights are private and can be accessed using suitable functions.

The core of the BRI class is the evaluation of the barycentric rational interpolant $r$, defined by the input $X_n$, $Y_n$, and $d$, through the EVAL function. The latter takes $x$ as input, which can be either a single evaluation point or a vector of points, and outputs $r(x)$, which in turn is a point or a vector, respectively. If the user wants to specify explicitly which algorithm to use for the evaluation, the EVAL function has an optional second parameter, which is one of the following keywords: `SECOND` to evaluate $r$ with the second barycentric form (Algorithm 1), `FIRST_DEF` to use the standard implementation of the first barycentric form (Algorithm 2), or `FIRST_EFF` for its more efficient variant (Algorithm 3). Without this second parameter, the code uses by default the algorithm that best balances efficiency and numerical stability. Likewise, the user can call the functions NUMERATOR and DENOMINATOR to evaluate the numerator and denominator of $r$, respectively. In case of the denominator, one of the three aforementioned keywords can be added to explicitly ask for the denominator to be computed by the corresponding algorithms.

The BRI class also provides three flags: the `stability` flag for controlling the numerical stability of the result, the `efficiency` flag for indicating a preference for the fastest evaluation routine whenever the first barycentric form is used, and the `guard` flag for activating additional checks that prevent overflow and underflow. The user can turn these flags on or off at any moment. In the following sections, we present in detail how the code decides which algorithm guarantees the numerically most stable or most efficient result if the respective flag is activated and how the guarding mechanism avoids possible overflow or underflow errors, without affecting the numerical accuracy of the result.

Finally, the class also allows to evaluate all the functions that are related to the numerical stability of the barycentric rational interpolant, namely the condition number $\kappa$, the Lebesgue function $\Lambda_n$, and the function $\Gamma_d$. They can be called either with an input $x$, which again can be an evaluation point or a vector of evaluation points, or without any arguments, upon which the algorithm searches for the maximum of the function using Newton's method.

**Algorithm 4** Robust procedure to compute $\hat{f} = 2^E f$

```
1: procedure RESCALE(f, E)
2:     f̂ := 0
3:     for i = 0, 1, ..., M do
4:         p_i := 2^{R_i − S_i + E}
5:         for j = 1, 2, ..., J do
6:             p_i := p_i · α_{i,j}
7:         for k = 1, 2, ..., K do
8:             p_i := p_i / β_{i,k}
9:         f̂ := f̂ + p_i
```

## 4.2 Robust procedure for rescale operation

Let $M, J, K \in \mathbb{N}$ and consider, for any $i = 0, 1, \ldots, M$, the numbers $a_{i,j} = \alpha_{i,j} \times 2^{A_{i,j}} \in \mathbb{F}$ for $j = 1, 2, \ldots, J$ and $b_{i,k} = \beta_{i,k} \times 2^{B_{i,k}} \in \mathbb{F}$ for $k = 1, 2, \ldots, K$. Denoting the products of these numbers by

$$r_i = \prod_{j=1}^{J} a_{i,j} \qquad \text{and} \qquad s_i = \prod_{k=1}^{K} b_{i,k}, \tag{4.3}$$

which can be expressed in floating-point encoding as

$$r_i = \rho_i \times 2^{R_i}, \qquad \text{where} \quad \rho_i = \prod_{j=1}^{J} \alpha_{i,j} \quad \text{and} \quad R_i = \sum_{j=1}^{J} A_{i,j}, \tag{4.4}$$

and

$$s_i = \sigma_i \times 2^{S_i}, \qquad \text{where} \quad \sigma_i = \prod_{k=1}^{K} \beta_{i,k} \quad \text{and} \quad S_i = \sum_{k=1}^{K} B_{i,k}, \tag{4.5}$$

suppose we want to compute the quantity

$$f = \sum_{i=0}^{M} \frac{r_i}{s_i}. \tag{4.6}$$

Even if we know that $f \in [F_{\min}, F_{\max}]$, it may happen that the algorithm that implements $f$ runs into overflow or underflow errors in some of its intermediate steps. However, we can try to avoid this problem by appropriately rescaling the intermediate floating-point values by some constant $C$, so that they are kept far from the overflow and underflow regions. Moreover, if we use a rescale factor of the type $2^E$, for some $E \in \mathbb{Z}$, then this operation modifies only the exponent of each floating-point number without changing the mantissa, meaning that we are not introducing any additional rounding error. Therefore, the goal now is to properly define the exponent $E$ so that we are sure that $\hat{f} = 2^E f$ can be safely computed, without any overflow or underflow error.

**Proposition 4.1.** *Let $M, J, K \in \mathbb{N}$ and consider $r_i$ and $s_i$ as in* (4.3)*, $\rho_i$ and $R_i$ as in* (4.4)*, $\sigma_i$ and $S_i$ as in* (4.5) *for $i = 0, 1, \ldots, M$, and $f$ as in* (4.6)*. Suppose that all $p_i = r_i/s_i$ have the same sign and define*

$$L = \min_{0 \le i \le M} (R_i - S_i) - J + 1 \qquad \textit{and} \qquad U = \max_{0 \le i \le M} (R_i - S_i) + K + M \tag{4.7}$$

*and*

$$E = \left\lceil \frac{E_{\max} + E_{\min} - L - U}{2} \right\rceil. \tag{4.8}$$

*If $U - L < E_{\max} - E_{\min}$, then we can compute $\hat{f} = 2^E f$ without any overflow or underflow error using Algorithm 4.*

*Proof.* The goal is to show that all the operations in Algorithm 4 are performed in $\mathbb{F}$. It follows from (4.8) that

$$\frac{E_{\max} + E_{\min} - L - U}{2} \leq E \leq \frac{E_{\max} + E_{\min} - L - U}{2} + \frac{1}{2},$$

which, together with (4.7) and the hypothesis $U - L < E_{\max} - E_{\min}$, gives

$$E_{\min} \leq E_{\min} + J - 1 \leq R_i - S_i + E \leq E_{\max} - K - M < E_{\max}, \tag{4.9}$$

meaning that the computation of $p_i$ in line 4 is always safe. By Proposition 2.1, we know that the multiplications in the loop of lines 5–6 can decrease the exponent $R_i - S_i + E$ by at most $J - 1$ and the divisions in the loop of lines 7–8 can increase it by at most $K$, but in both cases (4.9) guarantees that $p_i \in \mathbb{F}$. Finally, Proposition 2.1 guarantees that every summation in line 9 increases the exponent $R_i - S_i + E$ by at most 1, for a maximum of $M$ times. Again, by (4.9), we can thus be sure that also $\hat{f} \in \mathbb{F}$. □

**Corollary 4.2.** *Let $M, J, K \in \mathbb{N}$ and consider $r_i$ and $s_i$ as in* (4.3)*, $\rho_i$ and $R_i$ as in* (4.4)*, $\sigma_i$ and $S_i$ as in* (4.5) *for $i = 0, 1, \ldots, M$, and $f$ as in* (4.6)*. Suppose that all $p_i = r_i / s_i$ have the same sign, that there exist some $R_{\min}, R_{\max}, S_{\min}, S_{\max} \in \mathbb{Z}$, such that $R_{\min} \leq R_i \leq R_{\max}$ and $S_{\min} \leq S_i \leq S_{\max}$ for $i = 0, 1, \ldots, M$, and define*

$$L = R_{\min} - S_{\max} - J + 1 \qquad \text{and} \qquad U = R_{\max} - S_{\min} + K + M \tag{4.10}$$

*and*

$$E = \left\lceil \frac{E_{\max} + E_{\min} - L - U}{2} \right\rceil. \tag{4.11}$$

*If $U - L < E_{\max} - E_{\min}$, then we can compute $\hat{f} = 2^E f$ without any overflow or underflow error using Algorithm 4.*

*Proof.* Since $L \leq \min_{0 \leq i \leq M}(R_i - S_i) - J + 1$ and $U \geq \max_{0 \leq i \leq M}(R_i - S_i) + K + M$, the statement follows with the same arguments as in the proof of Proposition 4.1. □

## 4.3 Barycentric weights

As for the barycentric weights in (3.5), the simplest method to implement them is to follow their definition, thus obtaining an algorithm that computes all $\gamma_i$ in $O(nd^2)$ operations. However, Hormann and Schaefer [49] proposed the pyramid algorithm in (3.21) and (3.22) to compute the weights $v_i^l$, $l = d, d-1, \ldots, 0$ and $i = 0, \ldots, n-l$, with only $O(nd)$ operations and achieving the same precision [34]. Finally, the barycentric weights are then given as

$$\gamma_i = (-1)^{i-d} v_i^0, \qquad i = 0, 1, \ldots, n.$$

To save memory, the BRI class uses the vector of weights also to store the intermediate values $v_i^l$ in $\gamma_i$, which requires the assignment in (3.22) to be performed in reverse order (see Algorithm 5).

**Algorithm 5** Computing the barycentric weights $W_n$ for a given set of nodes $X_n$

```
1: procedure WEIGHTS
2:     W_n = (γ_0, γ_1, ..., γ_{n-d}, γ_{n-d+1}, ..., γ_n) := (1, 1, ..., 1, 0, ..., 0)
3:     for l = d − 1, d − 2, ..., 0 do
4:         γ_{n-l} := γ_{n-l-1} / (x_n − x_{n-l-1})
5:         for i = n − l − 1, n − l − 2, ..., 1 do
6:             γ_i := γ_{i-1} / (x_{i+l} − x_{i-1}) + γ_i / (x_{i+l+1} − x_i)
7:         γ_0 := γ_0 / (x_{l+1} − x_0)
8:     W_n := (−1)^d (γ_0, −γ_1, ..., (−1)^n γ_n)
```

After having initialized a new instance of the BRI class with the variables $X_n$, $Y_n$, and $d$, the vector $W_n$ is automatically computed as described above, which is efficient and robust. However, there are cases, albeit extreme, in which the WEIGHTS function runs into overflow or underflow errors.

**Example 4.1.** If we consider 10 Chebyshev nodes (4.2) (that is, $n = 9$) over $[0, 10^{-12}]$ and $d = 3$, then

$$\gamma_0 = \frac{-1}{(x_3 - x_0)(x_2 - x_0)(x_1 - x_0)} \approx -5.5257 \times 10^{38}, \tag{4.12}$$

and if the variable types are set to `float` for both input and output, then Algorithm 5 overflows in line 7 in the last iteration, when $l = 0$, so that $\gamma_0 = -\texttt{inf}$, because the smallest negative floating-point number in single precision is $-F_{\max} = -2^{127}(2 - 2^{-23}) \approx -3.4028 \times 10^{38}$. In fact, we get $\gamma_i = (-1)^{i+1}\texttt{inf}$ for *all* $i = 0, \ldots, 9$ in this example.

**Example 4.2.** Let us consider 3333 equidistant nodes over $[-1, 1]$ and $d = 333$. This choice of $n$ and $d$ guarantees the best balance between the theoretical and the actual numerical error of the barycentric rational interpolant, for example, if the data are sampled from the function $f(x) = \log(1.2 - x)/(x^2 + 2)$ [40]. However, if we compute the weights with Algorithm 5, using `double` for both input and output, we run into overflow problems and all weights turn out to be $\gamma_i = (-1)^{i+1}\texttt{inf}$.

**Example 4.3.** The BRI class also covers polynomial interpolation, that is, the case $d = n$, and in this setting underflow and overflow errors are even more common. For example, Pachón [67] shows that the weights can be computed safely for 500 Chebyshev nodes in $[-2, 2]$ in double precision, while problems arise, if the interval is changed to $[-0.2, 0.2]$ or $[-20, 20]$, leading to overflow in the former and underflow in the latter case. Furthermore, increasing the value of $n$ may result in similar issues. For example, for 1500 Chebyshev nodes in $[-2, 2]$, Algorithm 5 causes the first and the last 51 weights to underflow.

It is precisely for situations like these that we provide the `guard` flag, which, if activated, calls code that tries to prevent overflow and underflow errors. The basic idea is to rescale intermediate floating-point values, so that they are kept far from the overflow and underflow regions. To do this, we multiply the values $v_i^l$ in (3.22) for $i = 0, 1, \ldots, n - l$ in each step of the pyramid algorithm with a suitable common constant $2^{C_l}$, with $C_l \in \mathbb{Z}$. Of course, we need to define these constants appropriately to make sure that these rescaling operations are safe and do not in turn cause any overflow or underflow errors. To this end, we shall first work out intervals $I_l = \left[\mu_{\min} 2^{L_l}, \mu_{\max} 2^{U_l}\right]$ that are guaranteed to contain all $v_i^l$. Before delving into these details, it is worth recalling that such rescaling operations do not change the second barycentric formula in (3.3).

**Proposition 4.3.** *Let $L_d = U_d = 0$ and*

$$L_l = L_{l+1} - H_{\max} - E_l - 1 \qquad \text{and} \qquad U_l = U_{l+1} - H_{\min} - E_l + 2, \qquad l = d-1, d-2, \dots, 0, \tag{4.13}$$

*where $E_l = \lfloor \log_2(l+1) \rfloor$ and $H_{\min}, H_{\max} \in \mathbb{Z}$ are the exponents of the floating-point representation of the minimal and the maximal distance between neighbouring nodes, that is,*

$$h_{\min} = \min_{1 \le i \le n} (x_j - x_{j-1}) = \eta_1 \times 2^{H_{\min}} \qquad \text{and} \qquad h_{\max} = \max_{1 \le i \le n} (x_j - x_{j-1}) = \eta_2 \times 2^{H_{\max}}.$$

*Then, $v_i^l \in I_l = \left[\mu_{\min} 2^{L_l}, \mu_{\max} 2^{U_l}\right]$ for $i = 0, 1, \dots, n-l$ and any $l \in \{0, 1, \dots, d-1\}$.*

*Proof.* Assume that $B_{\min}^{l+1} \le v_i^{l+1} \le B_{\max}^{l+1}$ for $i = 0, 1, \dots, n-l-1$ and some $l \in \{0, 1, \dots, d-1\}$. Then, letting

$$B_{\min}^l = \frac{B_{\min}^{l+1}}{(l+1)h_{\max}} \qquad \text{and} \qquad B_{\max}^l = \frac{2B_{\max}^{l+1}}{(l+1)h_{\min}} \tag{4.14}$$

and noticing that

$$0 < (k-j)h_{\min} \le x_k - x_j \le (k-j)h_{\max}, \tag{4.15}$$

for any $j < k$, it follows from (3.22) that

$$B_{\min}^l \le \frac{v_{i-1}^{l+1}}{(l+1)h_{\max}} + \frac{v_i^{l+1}}{(l+1)h_{\max}} \le v_i^l \le \frac{v_{i-1}^{l+1}}{(l+1)h_{\min}} + \frac{v_i^{l+1}}{(l+1)h_{\min}} \le B_{\max}^l \tag{4.16}$$

for $i = 0, 1, \dots, n-l$. Consequently, defining $B_{\min}^d = B_{\max}^d = 1$ and $B_{\min}^l$ and $B_{\max}^l$ recursively as in (4.14) for $l = d-1, d-2, \dots, 0$, it follows by induction that $v_i^l \in \left[B_{\min}^l, B_{\max}^l\right]$ for $i = 0, 1, \dots, n-l$ and any $l \in \{0, 1, \dots, d-1\}$.

Denoting the exponents of the floating-point representation of $B_{\min}^l$ and $B_{\max}^l$ by $P_{\min}^l, P_{\max}^l \in \mathbb{Z}$ and noticing that

$$2^{E_l} \le l + 1 \le 2^{E_l + 1}, \tag{4.17}$$

it then follows from Proposition 2.1 and (4.14) that

$$B_{\min}^l \ge \mu_{\min} 2^{P_{\min}^{l+1} - H_{\max} - E_l - 1} \qquad \text{and} \qquad B_{\max}^l \le \mu_{\max} 2^{P_{\max}^{l+1} - H_{\min} - E_l + 2},$$

and therefore $v_i^l \in \left[B_{\min}^l, B_{\max}^l\right] \subset \left[\mu_{\min} 2^{L_l}, \mu_{\max} 2^{U_l}\right]$, where $L_l$ and $U_l$ are defined as in (4.13). □

Considering the floating-point representation of each value $v_i^{l+1} = \nu_i \times 2^{V_i}$ and the differences $x_{i+l+1} - x_i = \xi_i \times 2^{X_i}$, for some $l \in \{0, 1, \dots, d-1\}$ and $i = 0, 1, \dots, n-l$, we know from Proposition 4.3 that $L_{l+1} \le V_i \le U_{l+1}$ and, from (4.15) and (4.17), that $H_{\min} + E_l \le X_i \le H_{\max} + E_l + 1$. Moreover, we can observe that the expression for the values $v_i^l$ in (3.22) fits into the more general formula (4.6) for $M = J = K = 1$. This means that $L_l$ and $U_l$ in (4.13) are exactly $L$ and $U$ in (4.10), where $R_{\min} = L_{l+1}$, $R_{\max} = U_{l+1}$, $S_{\min} = H_{\min} + E_l$, and $S_{\max} = H_{\max} + E_l + 1$. Consequently, by Corollary 4.2, if $U_l - L_l < E_{\max} - E_{\min}$, then we can define

$$C_l = \left\lceil \frac{E_{\max} + E_{\min} - L_l - U_l}{2} \right\rceil$$

and compute $2^{C_l} v_i^l$ for $i = 0, 1, \dots, n-l$ without any overflow or underflow error using Algorithm 4.

**Algorithm 6** Guarded version of the WEIGHTS procedure

**procedure** WEIGHTS
  initialize $W_n$ as in line 2 of Algorithm 5 and set $L := 0$ and $U := 0$
  **for** $l = d-1, d-2, \ldots, 0$ **do**
    $L := L - H_{\max} - E_l - 1$ and $U := U - H_{\min} - E_l + 2$
    **if** $U - L < E_{\max} - E_{\min}$ **then**
      $C_l := \lceil (E_{\max} + E_{\min} - L_l - U_l)/2 \rceil$
      update $W_n$ with $\hat{W}_n = 2^{C_l} W_n$ as in lines 4–7 of Algo. 5, but using Algo. 4
      $L := L + C_l$ and $U := U + C_l$
    **else** ▷ expo$(x)$ returns $e$ for $x = \pm m \times 2^e$
      $L := \text{expo}(\min_{0 \le i \le n-l} \gamma_i) - 1$ and $U := \text{expo}(\max_{0 \le i \le n-l} \gamma_i)$
      $L := L - H_{\max} - E_l - 1$ and $U := U - H_{\min} - E_l + 2$
      **if** $U - L < E_{\max} - E_{\min}$ **then**
        $C_l := \lceil (E_{\max} + E_{\min} - L_l - U_l)/2 \rceil$
        update $W_n$ with $\hat{W}_n = 2^{C_l} W_n$ as in lines 4–7 of Algo. 5, but using Algo. 4
        $L := L + C_l$ and $U := U + C_l$
      **else**
        update $W_n$ as in lines 4–7 of Algo. 5
        **if** some overflow/underflow error happened **then**
          report an error and exit
        **else**
          $L := \text{expo}(\min_{0 \le i \le n-l} \gamma_i) - 1$ and $U := \text{expo}(\max_{0 \le i \le n-l} \gamma_i)$
  add the alternating signs to $W_n$ as in line 8 of Algo. 5

Therefore, the "guarded" version of the pyramid algorithm (see Algorithm 6) rescales the values $v_i^l$ by the constant $2^{C_l}$ in each step $l = d-1, d-2, \ldots, 0$. Therefore, if we first compute $L_l$ and $U_l$ as in (4.13) and find that $U_l - L_l < E_{\max} - E_{\min}$, then we can be sure that computing the values $2^{C_l} v_i^l$ with Algorithm 4 will not cause any overflow or underflow. If this latter condition is not satisfied for some $\hat{l} \in \{0, 1, \ldots, d-1\}$, it means that we cannot be sure anymore that all $v_i^l$ are normal floating-point numbers. Therefore, we try to repeat the same procedure by first re-defining

$$L_{\hat{l}} = \min_{0 \le i \le n-l} V_i - 1 \qquad \text{and} \qquad U_{\hat{l}} = \max_{0 \le i \le n-l} V_i. \tag{4.18}$$

If it fails again, then the algorithm proceeds normally by updating the weights as in lines 3–7 of Algorithm 5. If the calculations are successful for every $i = 0, 1, \ldots, n - \hat{l}$ without overflow nor underflow errors, then we define $L_{\hat{l}}$ and $U_{\hat{l}}$ as in (4.18), and we continue with the remaining iterations. Otherwise, the code stops and reports an error message to the user. At the end, the weights are rescaled with respect to the constant $2^{C_w}$, where

$$C_w = \sum_{l=0}^{d-1} C_l. \tag{4.19}$$

The next example illustrates how the guarded version can solve the overflow problem observed in Example 4.1.

**Example 4.4.** We consider $n = 9$, $d = 3$, and $n + 1$ Chebyshev interpolation nodes $x_i \in [0, 10^{-12}]$ with associated data $y_i = 1$, for $i = 0, \ldots, n$. We define all variables in input and output as `float` and we see how to initialize this instance of the BRI class and its resulting

```
#include "BRI.h"

using namespace std;

int main(){
    cout.precision(20);

    int n = 9;
    int d = 3;
    vector<float> Yn(n+1,1);

    BRI<float,float> r(CHEBYSHEV,0,1e-12,n,Yn,d);
    r.guard_on();
    int C;
    vector<float> W = r.get_weights(C);
    cout << "Cw = " << C << endl;

    for (int i=0; i<=n; i++)
        cout << "W[" << i << "] = " << W[i] << endl;
}
```

```
Cw = -127
W[0] = -3.2477385997772216797
W[1] = 6.8478851318359375
W[2] = -5.8251581192016601562
W[3] = 3.5328421592712402344
W[4] = -2.4203362464904785156
W[5] = 2.4203360080718994141
W[6] = -3.53284454345703125
W[7] = 5.8251657485961914062
W[8] = -6.8478937149047851562
W[9] = 3.2477431297302246094
```

Program 2. Code used to compute and display the guarded weights $w_i$ and the rescaling exponent $C_w$ for 10 Chebyshev nodes in $[0, 10^{-12}]$ and $d = 3$ in single precision (left) and its output (right).

weights in Program 2. We note that the weights do not overflow anymore as in (4.12) in guarded mode, but they are now rescaled by the constant $2^{-127}$.

## 4.4 Evaluation of the barycentric rational interpolant

As already mentioned in Section 4.1, the evaluation of the barycentric rational interpolant $r$ can be realized by using the second rational barycentric form with Algorithm 1 or the first form with either Algorithm 2 or 3. The BRI class allows the user to choose one of the three algorithms by calling the EVAL function with two input parameters, the first is the evaluation point or vector $x$ and the second is one of the three keywords `FIRST_DEF`, `FIRST_EFF`, or `SECOND`. However, the second parameter can be omitted, obtaining Algorithm 7 that autonomously chooses the best algorithm to use. Let us now focus on the latter case.

By default, the class always calls Algorithm 1, because it is generally the one that best combines numerical stability and efficiency of the method. On the other hand, if the user asks for the most stable solution by turning on the `stability` flag, then the code first evaluates $\kappa(x)$ and $\Lambda_n(x)$ and decides, based on these values, which is the best algorithm to use. In particular, it is known that the condition number affects the upper bound on the relative forward error of both the first and the second barycentric form [34], so if $\kappa(x) > 10^3$, then the class warns the user about this and continues without interruption. It is further known that the stability of (3.3) depends on the function $\Lambda_n$, which grows with $n$, whereas the stability of (3.6) depends on the function $\Gamma_d$, whose upper bound is independent of $n$. However, numerical experiments show that the estimated upper bound on $\Gamma_d$ seems to be always much larger than $\Gamma_d$ itself. For this reason, if $\Lambda_n(x) > 10^2$, then the result is computed with Algorithm 2. If it happens that also

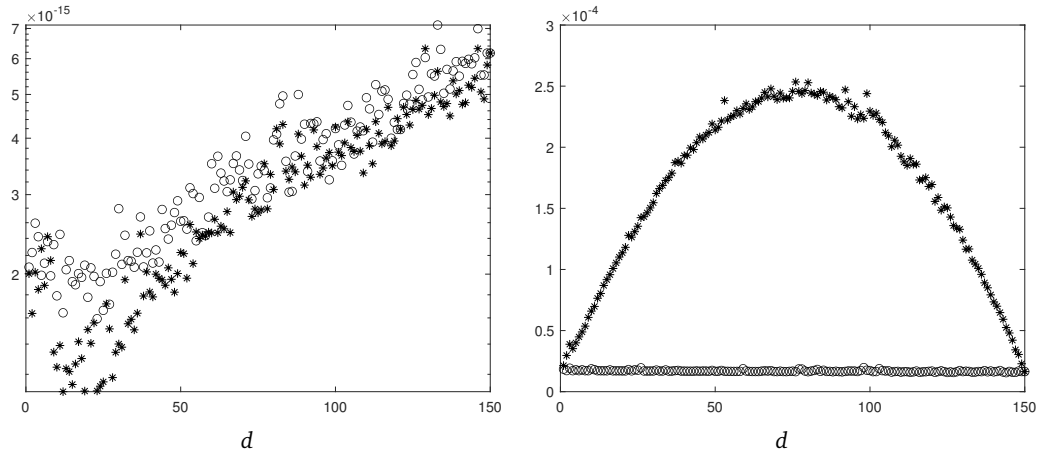


Figure 4.2. Maximal relative forward error of the first barycentric form for $n+1$ equidistant nodes, $n = 150$, and $d \in \{1, 2, \ldots, n\}$ at 1000 random evaluation points with data sampled from the $n$-th Lagrange basis polynomial on a logarithmic scale (left) and overall running time in seconds (right) using Algorithm 2 (asterisks) and Algorithm 3 (circles).

$\Gamma_d(x) > 10^2$, then the user simply receives a warning without the code stopping. Finally, if both the `stability` and the `efficiency` flags are active and it turns out that the most stable solution is given by the first barycentric form, then it is computed with Algorithm 3, which computes all the $\lambda_i$ with a more efficient iterative strategy. From a computational point of view, we are improving by switching from an $O(nd)$ algorithm to an $O(n)$ algorithm, therefore we know that for small $d$ there is no big gain, but, as $d$ increases, the efficient implementation can result in a lower execution time, without losing much in terms of stability. For example, Figure 4.2 compares the stability and the running time of the two algorithms considering $n+1$ equidistant interpolation nodes $x_i \in [0, 1]$ for $n = 150$ with associated data $y_i = 0$, for $i = 0, \ldots, n-1$ and $y_n = 1$. In particular, on the left we see the maximum of both relative forward errors evaluated at 1000 random points in $[0, 1]$ and on the right their execution times in seconds as a function of $d \in \{1, 2, \ldots, n\}$. We observe that Algorithm 2 is slightly more precise than Algorithm 3, as expected, but the latter always wins in terms of efficiency, especially in the neighborhood of $d = n/2$. It is worth noting that, if the `stability` flag is turned off, then the `efficiency` flag does not play any role, because Algorithm 7 takes the second barycentric form by default.

As for the weights, also in this case the user can decide to turn on the `guard` flag to prevent overflow and underflow errors. The basic idea is again the same as in Section 4.3 for the weights, that is, rescaling the numerator and the denominator properly in order to perform all floating-point operations far from the overflow and underflow regions and to avoid introducing any further rounding errors. In this case, we have to further take into account that the summations in the numerator and denominator of (3.3) and (3.6) involve both subtractions and additions that may lead to cancellation errors. Moreover, from Proposition 2.1, preventing non-underflow subtractions could be tricky, especially if we are subtracting numbers of similar magnitudes. However, we recall that cancellation errors are already taken into consideration in the study of the numerical stability of the interpolant [34], so, if the method is stable, then it is proven that they cannot happen. Otherwise, one option would be to split each summation into one for the positive terms and one for the negative terms, which, in case the condition of

**Algorithm 7** EVAL function

```
1: function EVAL(x)
2:     if the stability flag is turned on then
3:         if κ(x) > 10^3 then the user gets a warning
4:         if Λ_n(x) > 10^2 then
5:             if Γ_d(x) > 10^2 then the user gets a warning
6:             if the efficiency flag is turned on then
7:                 return r, computed with Algo. 3
8:             else
9:                 return r, computed with Algo. 2
10:        else
11:            return r, computed with Algo. 1
12:    else
13:        return r, computed with Algo. 1
```

Proposition 4.1 is satisfied, can be safely computed using Algorithm 4, and making only one final subtraction. The problem with this procedure is that, from our experiments, it seems to lose precision compared to the normal iterative algorithm when there are cancellation errors. Furthermore, since the two summations consist only of additions, if the addends have high orders of magnitude and $n$ is large, then it is much more likely to run into overflow errors. For these reasons, we decided to safely compute only the addends of the summations in (3.3) and (3.6) using Proposition 4.1 (for the special case $M = 0$), and finally to calculate the sums with the classic iterative algorithm. Therefore, no adjustment is considered that could prevent overflow or underflow errors during the summations, but we check only afterwards if something like this has happened. In particular, considering the floating-point representation of each weight $\gamma_i = \omega_i \times 2^{W_i}$, data $y_i = \upsilon_i \times 2^{Y_i}$, and difference $x - x_i = \xi_i \times 2^{X_i}$, we let

$$
\begin{aligned}
L_1 &= \min_{0\le i\le n}(W_i + Y_i - X_i) - 1, & U_1 &= \max_{0\le i\le n}(W_i + Y_i - X_i) + 1,\\
L_2 &= \min_{0\le i\le n}(W_i - X_i), & U_2 &= \max_{0\le i\le n}(W_i - X_i) + 1,\\
L_3 &= \min_{0\le i\le n-d}(-X_i - \cdots - X_{i+d}), & U_3 &= \max_{0\le i\le n-d}(-X_i - \cdots - X_{i+d}) + d + 1,
\end{aligned}
$$

and we define

$$
C_j = \left\lceil \frac{E_{\max} + E_{\min} - L_j - U_j}{2} \right\rceil, \qquad j = 1, 2, 3. \tag{4.20}
$$

After that, denoting the common numerator of (3.3) and (3.6), the denominator of (3.3), and the denominator of (3.6) by $N$, $D_s$, and $D_f$, respectively, if $U_j - L_j < E_{\max} - E_{\min}$ for all $j \in \{1, 2, 3\}$, then we can compute

$$
\hat{N} = 2^{C_1} N, \qquad \hat{D}_s = 2^{C_2} D_s, \qquad \text{and} \qquad \hat{D}_f = 2^{C_3} D_f
$$

with Algorithm 4, but, since the exponents in (4.20) do not consider additions and subtractions, at the end it is necessary to check whether overflow or underflow errors have happened. Finally, if every operation was computed safely, then we get the result with the second barycentric form as

$$
r(x) = 2^{C_2 - C_1} \frac{\hat{N}(x)}{\hat{D}_s(x)}
$$

**Algorithm 8** Guarded evaluation of the second barycentric form

**function** SECOND(x)
  $L_1 := \min_{0\le i\le n} \operatorname{expo}(\gamma_i y_i/(x-x_i))$
  $U_1 := \max_{0\le i\le n} \operatorname{expo}(\gamma_i y_i/(x-x_i)) + 1$
  **if** $U_1 - L_1 < E_{\max} - E_{\min}$ **then**
    $C_1 := \lceil (E_{\max} + E_{\min} - L_1 - U_1)/2 \rceil$
    compute $\hat{N}$ as $N$ in Algo. 1, but using Algo. 4
    **if** some overflow/underflow happened **then**
      report an error and exit
  **else**
    compute $\hat{N}$ as $N$ in Algo. 1
    **if** some overflow/underflow happened **then**
      report an error and exit
    **else**
      $C_1 := 0$
  $L_2 := \min_{0\le i\le n} \operatorname{expo}(\gamma_i/(x-x_i)) + 1$
  $U_2 := \max_{0\le i\le n} \operatorname{expo}(\gamma_i/(x-x_i)) + 1$
  **if** $U_2 - L_2 < E_{\max} - E_{\min}$ **then**
    $C_2 := \lceil (E_{\max} + E_{\min} - L_2 - U_2)/2 \rceil$
    compute $\hat{D}_s$ as $D$ in Algo. 1, but using Algo. 4
    **if** some overflow/underflow happened **then**
      report an error and exit
  **else**
    compute $\hat{D}_s$ as $D$ in Algo. 1
    **if** some overflow/underflow happened **then**
      report an error and exit
    **else**
      $C_2 := 0$
  **return** $r = 2^{C_2 - C_1} \frac{\hat{N}}{\hat{D}_s}$

**Algorithm 9** Guarded evaluation of the first barycentric form

**function** FIRST(x)
  $L_1 := \min_{0\le i\le n} \operatorname{expo}(\gamma_i y_i/(x-x_i))$
  $U_1 := \max_{0\le i\le n} \operatorname{expo}(\gamma_i y_i/(x-x_i)) + 1$
  **if** $U_1 - L_1 < E_{\max} - E_{\min}$ **then**
    $C_1 := \lceil (E_{\max} + E_{\min} - L_1 - U_1)/2 \rceil$
    compute $\hat{N}$ as $N$ in Algo. 1, but using Algo. 4
    **if** overflow/underflow happened **then**
      report an error and exit
  **else**
    compute $\hat{N}$ as $N$ in Algo. 1, but using Algo. 4
    **if** some overflow/underflow happened **then**
      report an error and exit
    **else**
      $C_1 := 0$
  $L_3 := \min_{0\le i\le n-d} \operatorname{expo}([(x-x_i)\dots(x-x_{i+d})]^{-1}) + 1$
  $U_3 := \max_{0\le i\le n-d} \operatorname{expo}([(x-x_i)\dots(x-x_{i+d})]^{-1}) + d + 1$
  **if** $U_3 - L_3 < E_{\max} - E_{\min}$ **then**
    $C_3 := \lceil (E_{\max} + E_{\min} - L_3 - U_3)/2 \rceil$
    compute $\hat{D}_f$ as $D$ in Algo. 2 or Algo. 3, but using Algo. 4
    **if** some overflow/underflow happened **then**
      report an error and exit
  **else**
    compute $\hat{D}_f$ as $D$ in Algo. 2 or Algo. 3
    **if** some overflow/underflow happened **then**
      report an error and exit
    **else**
      $C_3 := 0$
  **return** $r = 2^{C_3 - C_1 - C_w} \frac{\hat{N}}{\hat{D}_f}$

or with the first barycentric form as

$$r(x) = 2^{C_3 - C_1 - C_w} \frac{\hat{N}(x)}{\hat{D}_f(x)}.$$

Otherwise, the code stops and gives an error message to the user. Note that the guarded result of the first form involves also the rescale factor $C_w$ in (4.19), because its denominator does not depend on the weights. If $U_j - L_j \ge E_{\max} - E_{\min}$ for some $j \in \{1,2,3\}$, then the algorithm tries to compute the corresponding summation as in Algorithm 1, 2, or 3. Again, if some overflow or underflow error happens, then the code stops, otherwise the corresponding rescale factor $C_j$ is set to 0 for $j \in \{1,2,3\}$ and the final result is computed as before.

**Example 4.5.** As in Example 4.1, we consider $n = 9$ and $n+1$ Chebyshev interpolation nodes in $[0, 10^{-12}]$, but we take $d = 2$ so that the weights do not overflow even in non-guarded mode. We then define $y_i = f(x_i)$ for $i = 0, 1, \dots, n$, where

$$f(x) = \tfrac{3}{4} e^{-\frac{(9x-2)^2}{4}} + \tfrac{3}{4} e^{-\frac{(9x+1)^2}{49}} + \tfrac{1}{2} e^{-\frac{(9x-7)^2}{4}} + \tfrac{1}{5} e^{-(9x-4)^2}, \tag{4.21}$$

and we output $r(x)$ using first Algorithms 1 and 2 and then Algorithms 8 and 9, with $x = 10^{-12}/2$. We also use the functions NUMERATOR and DENOMINATOR to see the values of $N$, $D_s$, and $D_f$ with their rescaling factors. All variables in input and output are set as `float` and

```
#include "BRI.h"

using namespace std;

float Franke(float x) {
    return exp(-(9*x-2)*(9*x-2)/4)*3/4
             +exp(-(9*x+1)*(9*x+1)/49)*3/4
             +exp(-(9*x-7)*(9*x-7)/4)/2
             +exp(-(9*x-4)*(9*x-4))/5;
}

int main(){
    cout.precision(10);

    int n = 9;
    int d = 2;

    BRI<float,float> r(CHEBYSHEV,0,1e-12,n,Franke,d);
    float x = 1e-12/2;
    //r.guard_on();

    int C1;
    int C2;
    int C3;

    float f_def = r.eval(x,FIRST_DEF);
    float s = r.eval(x,SECOND);
    float N = r.numerator(x,C1);
    float Ds = r.denominator(x,C2,SECOND);
    float Df = r.denominator(x,C3,FIRST_DEF);
    cout << "FIRST FORM - DEF: r(x) = " << f_def <<
     endl;
    cout << "SECOND FORM: r(x) = " << s << endl;
    cout << "N = " << N << " and C1 = " << C1 << endl
     ;
    cout << "Ds = " << Ds << " and C2 = " << C2 <<
     endl;
    cout << "Df = " << Df << " and C3 = " << C3 <<
     endl;
}
```

```
GUARD FLAG TURNED OFF

FIRST FORM - DEF: r(x) = -nan
SECOND FORM: r(x) = -nan
N = -nan and C1 = 0
Ds = -nan and C2 = 0
Df = inf and C3 = 0

GUARD FLAG TURNED ON

FIRST FORM - DEF: r(x) = 1.010761023
SECOND FORM: r(x) = 1.010760903
N = 5.134361744 and C1 = -44
Ds = 10.15939903 and C2 = -43
Df = 8.936301055e+13 and C3 = -84
```

Program 3. Code used to compute and display $r(x)$, $N(x)$, $D_s(x)$, and $D_f$ with their rescaling factors $C_1$, $C_2$, and $C_3$ for 10 Chebyshev nodes $x_i$ in $[0, 10^{-12}]$ with associated data $y_i$ sampled from the function $f$ in (4.21), $x = 10^{-11}/2$, and $d = 2$ in single precision (left). The output (right) is shown both in non-guarded and guarded mode, where the latter is obtained by removing the comment in the line with the command `r.guard_on();`.

we present the input code and its output in Program 3. Note that, up to single precision, the evaluation of the interpolant at $x$ is the same for both algorithms, that is, $r(x) = 1.01076$, and the values of numerator and denominators are

$$N(x) = 2^{44}5.13436, \qquad D_s(x) = 2^{43}10.1594, \qquad \text{and} \qquad D_f(x) = 2^{84+C_w}8.93630e+13,$$

where $C_w = -84$ in this setting. Furthermore, using the efficient implementation of the first form, we would get the same result.

Finally, we analysed how much the `guard` and the `stability` flags can affect the computational cost of the method. In particular, we conducted several tests to compare the running times with these flags both turned on and off and also considering variations in the input, such as $d$, the number of nodes and data $n$, and the number of evaluation points. Notably, if we activate only the `guard` flag, then we observed a maximum increase in the execution time of one order of magnitude. Instead, in the worst-case scenario where we enable both the `guard` and the `stability` flag, the increase can reach at most two orders of magnitude.

### 4.4.1 Evaluation close to a node

One of the properties of the set $\mathbb{F}$ is that it is not equally spaced, but it becomes denser in the proximity of $F_{\min}$. In particular, we recall that the floating-point numbers are equally spaced in each interval $\left[2^E, 2^{E+1}\right]$ and at a distance of $h_E = 2^{E-t}$. Hence, if we want to evaluate $r$ very close to a node $x_j \in \left[2^E, 2^{E+1}\right]$ for some $E \in \mathbb{N}$, then we cannot get any closer than $h_E$. Consequentially, if the evaluation point is $x = x_j + h$ and $x_j$ and $h$ have different orders of magnitude, then we can lose accuracy because too small increments are ignored. Similarly, if we get too close to $x_j$, in view of the continuity of the interpolant $r$, we have $r(x_j + h) = y_j + u$ and if $u \in \mathbb{R}$ is too small, then the code could round the final result to $y_j$. However, we note that

$$r(x_j + h; X_n, Y_n) = \frac{\sum_{i=0}^{n} \frac{\gamma_i}{x_j + h - x_i} y_i}{\sum_{i=0}^{n} \frac{\gamma_i}{x_j + h - x_i}} = \frac{\sum_{i=0}^{n} \frac{\gamma_i}{h - (x_i - x_j)} (y_i - y_j)}{\sum_{i=0}^{n} \frac{\gamma_i}{h - (x_i - x_j)}} + y_j = r(h; X_n - x_j, Y_n - y_j) + y_j,$$

so that shifting the domain and the range by $x_j$ and $y_j$, respectively, we can compute $u = r(h)$ and get the final result as $y_j + u$. The advantage of this procedure is that both $h$ and $u$ are more accurate and reach up to a magnitude of $2^{E_{\min}-1-t}$, that is, the distance between floating-point numbers in $\left[2^{E_{\min}-1}, 2^{E_{\min}}\right]$. For this reason, the BRI class automatically uses this strategy if the EVAL function receives as input the index $j$ and the value $h$ and it returns $u$.

**Example 4.6.** We consider $n = 9$, $d = 2$, and $n + 1$ equidistant interpolation nodes $x_i \in [1, 2]$ with associated data $y_i = f(x_i)$ for $i = 0, 1, \ldots, n$ and $f(x) = x$. We want to evaluate the interpolant $r$ very close to the first node $x_0 = 1$ at $x = x_0 + h$ for $h = 10^{-20}$, and we set all variables in input and output as `double`. We know that the closest double floating-point number to $x_0 = 1$ is $1 + \epsilon$, where $\epsilon = 2.2204 \times 10^{-16}$. So, using any of the three algorithms discussed in Section 4.4, we expect to get $r(x) = y_0$, since $x$ is rounded to $x_0$ in double precision. We compare the result obtained by evaluating $r(x)$ in the classical way with the strategy presented above in Program 4. As expected, the classical method outputs $r(x) = 1$, but with the new strategy we obtain $u = r(h) = 9.9999999999999949376e-21$, meaning that $r(x) = y_0 + u = 1.00000000000000000001$, which is also what we would get by computing $r(x)$ in multiple-precision (1024 bits).

## 4.5 Stability-related functions

As mentioned in Section 4.4, the BRI class uses the functions $\Lambda_n$, $\Gamma_d$, and $\kappa$ internally to decide, if needed, which method should be used for the computation of $r(x)$. Moreover, it also allows the user to evaluate them explicitly by using the functions LEB, GAMMA, and COND with the evaluation point or vector $x$ as input. However, for purposes of numerical stability, what really

```
#include "BRI.h"
#include "NODES.h"

using namespace std;

int main(){
    cout.precision(20);

    int n = 9;
    int d = 2;

    NODES<double> obj;
    vector<double> Xn = obj.uniform(1,2,n);
    double h = 1e-20;
    double x = xn[0] + h;

    BRI<> r(Xn,Xn,d);

    double r1 = r.eval(x);
    double r2 = r.eval(0,h);

    cout << "eval(x) outputs:" << endl;
    cout << "r(x) = " << r1 << endl;
    cout << "eval(0,h) outputs:" << endl;
    cout << "r(h) = " << r2 << endl;
}
```

```
eval(x) outputs:
r(x) = 1
eval(0,h) outputs:
r(h) =  9.9999999999999949376e-21
```

Program 4. Input used to compute and display $r(x)$ for 10 equidistant nodes $x_i$ in $[1,2]$ with associated data $y_i = x_i$ and $d = 2$ at $x = x_0 + h$ for $h = 10^{-20}$ in double precision (left). The output (right) shows the result of the eval function, both for $x$ and the pair $(0,h)$ as input.

matters is the maximum of these functions in the range $[x_0, x_n]$ and this is what these functions return when they have no argument as input. Let us see how they compute it.

First, we recall that $\Lambda_n$, $\Gamma_d$, and $\kappa$ can all be written in the common form

$$g(x) = \sum_{i=0}^{n} \left| \frac{b_i(x)}{\sum_{i=0}^{n} b_i(x)} \right|, \tag{4.22}$$

where $b_i(x) = \gamma_i/(x-x_i)$ for $\Lambda_n$, $b_i(x) = (-1)^i/\prod_{j=i}^{i+d}(x-x_j)$ for $\Gamma_d$, and $b_i(x) = \gamma_i y_i/(x-x_i)$ for $\kappa$. One possible way to compute their maxima is to sample these functions very densely on the domain and search for the largest values, but, to be really accurate, we have to consider a big number of samples, thus losing efficiency. However, it is clear from the triangle inequality that a general function of the type (4.22) is greater than or equal to 1. Furthermore, in the specific case of $\Lambda_n$, $\Gamma_d$, and $\kappa$, the minimum with value 1 is assumed exactly at the nodes $x_i$ and, from numerical experiments (see Figure 4.3), it appears that they are all concave in every sub-interval $[x_i, x_{i+1}]$. This means that, considering these functions locally in every open sub-interval $(x_i, x_{i+1})$, the point where the first derivative vanishes is a local maximum. Therefore, we apply *Newton's method* to find the root of $\Lambda_n'$, $\Gamma_d'$, and $\kappa'$ in each sub-interval $(x_i, x_{i+1})$ and we finally search for the maximum point among them. In general, in order to find a root $t \in \mathbb{R}$ of a function $g : \mathbb{R} \to \mathbb{R}$, Newton's method starts with some $t_0 \in \mathbb{R}$ and then generates a sequence

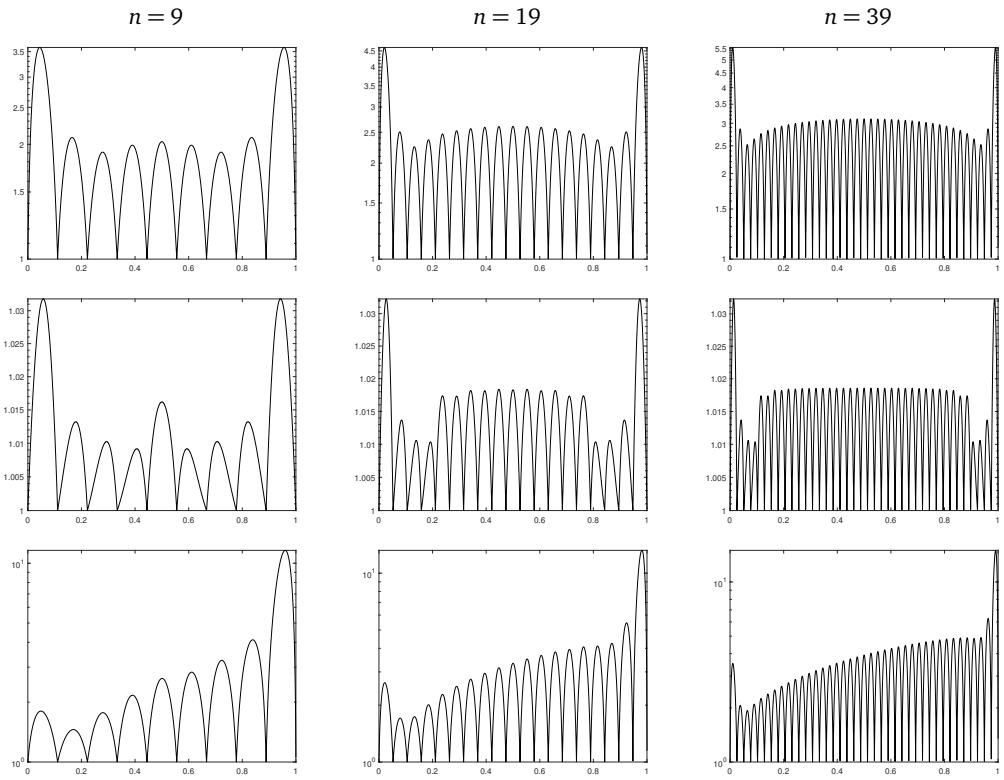


Figure 4.3. Plots of $\Lambda_n$ (top), $\Gamma_d$ (middle), and $\kappa$ (bottom) for $n+1$ equidistant nodes and $x \in [0,1]$, all on a logarithmic scale, for $d = 3$ and three choices of $n$ (left, middle, right). For the computation of the condition number $\kappa$, the data are sampled from Runge's function $f(x) = 1/(1+25x^2)$.

of $t_1, t_2, \cdots \in \mathbb{R}$ by applying the iterative formula

$$t_{j+1} = t_j - \frac{g(t_j)}{g'(t_j)}. \tag{4.23}$$

Although this method usually converges quadratically, it may also fail, for example, if the initial guess $t_0$ is too far from the correct solution. Consequently, the choice of the initial value $t_0$ is an important step of Newton's method. Figure 4.3 shows that $\Lambda_n$, $\Gamma_d$, and $\kappa$ take their local maxima approximately in the midpoint of every sub-interval $[x_i, x_{i+1}]$, so that we consider $t_0 = (x_i + x_{i+1})/2$ in each sub-interval $(x_i, x_{i+1})$. After that, by (4.22), it is easy to see that the method can be implemented straightforwardly by computing at each iteration $j$ first

$$g'(t_j) = \sum_{i=0}^{n} \operatorname{sign}\Big(\frac{b_i(t_j)}{\sum_{i=0}^n b_i(t_j)}\Big) \frac{b_i'(t_j)\sum_{i=0}^n b_i(t_j) - b_i(t_j)\sum_{i=0}^n b_i'(t_j)}{(\sum_{i=0}^n b_i(t_j))^2}$$

and

$$g''(t_j) = \sum_{i=0}^{n} \operatorname{sign}\Big(\frac{b_i(t_j)}{\sum_{i=0}^n b_i(t_j)}\Big) \frac{b_i''(t_j)\sum_{i=0}^n b_i(t_j) - b_i(t_j)\sum_{i=0}^n b_i''(t_j)}{(\sum_{i=0}^n b_i(t_j))^2} - 2\frac{g'(t_j)}{\sum_{i=0}^n b_i(t_j)} \sum_{i=0}^{n} b_i'(t_j),$$

```
#include "mpreal.h"
#include "BRI.h"

using namespace std;
using mpfr::mpreal;

double Runge(double x){
    return 1/(1+25*x*x);
}

int main(){
    int my_mpreal_precision = 1024;
    mpreal::set_default_prec(my_mpreal_precision);

    int n = 9;
    int d = 3;
    BRI<double,mpreal> r(UNIFORM,0,1,n,Runge,d);

    cout.precision(20);
    cout << "Maximum of the Lebesgue function:" << endl;
    cout << r.leb() << endl;
    cout << "Maximum of the function Gamma:" << endl;
    cout << r.gamma() << endl;
    cout << "Maximum of the condition number:" << endl;
    cout << r.cond() << endl;
}
```

```
n = 9

Maximum of the Lebesgue function:
3.5886287189761606401
Maximum of the function Gamma:
1.0318045847764588507
Maximum of the condition number:
11.609466977862612706

n = 19

Maximum of the Lebesgue function:
4.6127100859322925745
Maximum of the function Gamma:
1.0322814978345268598
Maximum of the condition number:
13.210805972626199269

n = 39

Maximum of the Lebesgue function:
5.5370777898252804523
Maximum of the function Gamma:
1.0323229058478393483
Maximum of the condition number:
14.979125760718810308
```

and then using (4.23) to set $t_{j+1} = t_j - g'(t_j)/g''(t_j)$, until the error $|t_{j+1} - t_j| < 10\epsilon$, where $\epsilon$ is the machine epsilon.

**Example 4.7.** We consider the same setting as in Figure 4.3 for $n = 9$, that is, $n+1 = 10$ equidistant interpolation nodes $x_i \in [0,1]$ with associated data $y_i = f(x_i)$ for $i = 0,\ldots,n$ and $f(x) = 1/(1+25x^2)$ and $d = 3$. We create a new instance of the BRI class that takes `double` input and returns multiple precision (1024 bits) output using the MPFR library, and we ask for the maximum of the functions $\Lambda_n$, $\Gamma_d$, and $\kappa$.

# Chapter 5

# A new stable method to compute mean value coordinates

## 5.1 State of the art

Mean value coordinates were initially introduced as a generalization of barycentric coordinates to polygons and polyhedra [28, 32, 47, 54]. Since then, they have emerged as a valuable tool in a wide range of domains, such as interpolation, curve and surface modeling in computer graphics, mesh parameterization, the finite element method, and various other fields. Moreover, they stand out for their capability to extend barycentric coordinates to the non-convex setting, unlike other commonly used coordinates. For more details about generalized barycentric coordinates, we refer to Hormann and Sukumar [50].

Let $P \subset \mathbb{R}^2$ be a simple planar polygon with $n \geq 3$ vertices $v_1, \ldots, v_n$ arranged in anticlockwise ordering and $v \in \mathbb{R}^2$ be an arbitrary point in the interior of $P$. The *mean value coordinates* of $v$ with respect to $P$ are initially defined by Floater [28] as

$$\phi_i(v) = \frac{w_i(v)}{\sum_{j=1}^{n} w_j(v)}, \qquad w_i(v) = \frac{1}{r_i}\left(\tan\frac{\alpha_{i-1}}{2} + \tan\frac{\alpha_i}{2}\right), \qquad i = 1, \ldots, n, \tag{5.1}$$

with $\alpha_i \in (-\pi, \pi)$ denoting the signed angle at $v$ in the triangle $[v, v_i, v_{i+1}]$ and $r_i = \|v - v_i\|$. Note that indices are considered cyclically with respect to the range $[1, 2, \ldots, n]$; for example, $v_{n+1} = v_1$ and $\alpha_0 = \alpha_n$. As already seen in Section 1.1, these coordinates satisfy the properties (1.3), (1.4), and (1.5). Moreover, they are actually well-defined for all $v \in \mathbb{R}^2 \setminus \partial P$, positive inside the kernel of $P$, invariant to similarity transformations of $P$, and their extension is linear along the edges of $P$ and smooth except at the vertices $v_i$, where it is only $C^0$ [47].

Afterwards, Floater et al. [31] note that mean value coordinates are a particular member of a family of *three-point coordinates* for convex polygons, which can be derived by normalizing a set of weight functions $w_i$ that each depend on three consecutive vertices of $P$. In this context, they show that mean value coordinates can be expressed as

$$\phi_i(v) = \frac{w_i(v)}{\sum_{j=1}^{n} w_j(v)}, \qquad w_i(v) = \frac{r_{i-1}A_{i,i+1} - r_i A_{i-1,i+1} + r_{i+1}A_{i-1,i}}{2A_{i-1,i}A_{i,i+1}}, \qquad i = 1, \ldots, n, \tag{5.2}$$

where $A_{i,j} = \det(v_i - v, v_j - v)/2$ denotes the signed area of the triangle $[v, v_i, v_j]$. The advantage

of this formula over the original definition in (5.1) is that it avoids the computation of the angles $\alpha_i$ and that it gets by without the use of trigonometric functions.

While mean value coordinates were initially considered only for points inside the kernel of star-shaped polygons [28], Hormann and Floater [47] prove that they are well-defined for any $v \in \mathbb{R}$ and (sets of) arbitrary planar polygons without self-intersection. They also propose another way of evaluating mean value coordinates that avoids trigonometric functions. In particular, they first use the half-angle formula for the tangent to get

$$\phi_i(v) = \frac{w_i(v)}{\sum_{j=1}^n w_j(v)}, \qquad w_i(v) = \frac{1}{r_i}\left(\frac{1-\cos\alpha_{i-1}}{\sin\alpha_{i-1}} + \frac{1-\cos\alpha_i}{\sin\alpha_i}\right), \qquad i = 1,\dots,n, \tag{5.3}$$

and then, denoting the dot product of $v_i - v$ and $v_j - v$ by $D_{i,j} = (v_i - v)\cdot(v_j - v)$ and recalling that $D_{i,i+1} = r_i r_{i+1}\cos\alpha_i$ and $2A_{i,i+1} = r_i r_{i+1}\sin\alpha_i$, they conclude that the mean value coordinates in (5.1) can be written as

$$\phi_i(v) = \frac{w_i(v)}{\sum_{j=1}^n w_j(v)}, \qquad w_i(v) = \frac{1}{r_i}\left(\frac{r_{i-1}r_i - D_{i-1,i}}{2A_{i-1,i}} + \frac{r_i r_{i+1} - D_{i,i+1}}{2A_{i,i+1}}\right), \qquad i = 1,\dots,n. \tag{5.4}$$

The advantage of implementing this formula over (5.2) is that it allows to easily "catch" the case when $v$ is on the boundary of $P$, say $v = (1-\mu)v_k + \mu v_{k+1}$ for some $\mu \in [0,1]$ and some $k \in \{1,\dots,n\}$, as this happens if and only if $A_{k,k+1} = 0$ and $D_{k,k+1} \le 0$. In this case, the mean value coordinates of $v$ are just $\phi_k(v) = 1-\mu$, $\phi_{k+1}(v) = \mu$, and $\phi_i(v) = 0$ for $i \neq k, k+1$.

One potential problem with the formulas in (5.2)–(5.4) is that the coordinates $\phi_i(v)$ are not well-defined if $v$ is on the line supporting the edge $[v_k, v_{k+1}]$ of $P$. We can overcome this problem by using the alternative half-angle formula for the tangent, that is,

$$\phi_i(v) = \frac{w_i(v)}{\sum_{j=1}^n w_j(v)}, \qquad w_i(v) = \frac{1}{r_i}\left(\frac{\sin\alpha_{i-1}}{1+\cos\alpha_{i-1}} + \frac{\sin\alpha_i}{1+\cos\alpha_i}\right), \qquad i = 1,\dots,n, \tag{5.5}$$

to obtain

$$\phi_i(v) = \frac{w_i(v)}{\sum_{j=1}^n w_j(v)}, \qquad w_i(v) = \frac{1}{r_i}\left(\frac{2A_{i-1,i}}{r_{i-1}r_i + D_{i-1,i}} + \frac{2A_{i,i+1}}{r_i r_{i+1} + D_{i,i+1}}\right), \qquad i = 1,\dots,n. \tag{5.6}$$

This formula gives rise to an implementation that has the same advantages as the one derived from (5.4), but is well-defined even if $A_{k,k+1} = 0$ for some $k$.

All the formulas above have the limitation that they are not well-defined on the boundary of the polygon $P$ and, moreover, (5.2)–(5.4) can be used only if all $A_{i,i+1} \neq 0$. This motivated Floater [29] to introduce yet another formula for mean value coordinates, which is also valid on the boundary, namely

$$\phi_i(v) = \frac{\hat{w}_i(v)}{\sum_{j=1}^n \hat{w}_j(v)}, \qquad \hat{w}_i(v) = \sigma_i\sqrt{r_{i-1}r_{i+1} - D_{i-1,i+1}}\prod_{j\neq i-1,i}\sqrt{r_j r_{j+1} + D_{j,j+1}}, \qquad i = 1,\dots,n, \tag{5.7}$$

where $\sigma_i \in \{+1,-1\}$ is a sign related to the weight function $\hat{w}_i$. Initially, this formula was presented without $\sigma_i$, which limits its applicability to points $v$ inside convex polygons, but Anisimov [2, Section 3.2.4] demonstrates how to define $\sigma_i$, such that it can be used for any $v \in \mathbb{R}^2$ and arbitrary simple polygons. The disadvantage of this formula is that its implementation requires $O(n^2)$ operations, while the formulas (5.1)–(5.6) give rise to $O(n)$ algorithms.

To the best of our knowledge, there is no study regarding the numerical stability of any algorithm that implements normalized barycentric coordinates. However, Anisimov et al. [3] noticed that both Wachspress and mean values coordinates become unstable close to the boundary of the polygon $P$. In the case of the former, this problem can be overcome by multiplying the barycentric coordinates by an appropriate constant, although this affects the computational cost of the algorithm. For the latter, only the formula in (5.7) appears to be stable. In this thesis, we want to investigate more deeply the numerical stability of the mean value coordinates, using a similar procedure to that used in case of barycentric rational interpolation.

## 5.2 Our contribution

The purpose of this chapter is to investigate the numerical stability of algorithms that implement the mean value coordinates using the formulas in (5.1)–(5.7). First, we empirically compare the behavior of these methods on a specific polygon by evaluating their absolute and relative forward errors. This analysis reveals that each formulation suffers from numerical instability in certain parts of the domain. Therefore, based on this observation, we develop a new method that outperforms the others and explain its implementation to avoid potential numerical issues. Next, we conduct a theoretical analysis on the numerical stability of these formulas with the same approach used for the univariate barycentric interpolant in Chapter 3. Hence, we derive upper bounds for the relative forward errors of all methods and mathematically demonstrate that our new method is the only one that is stable across the entire domain. Finally, we validate our theoretical findings with numerical experiments, comparing the different methods in terms of both numerical stability and efficiency.

## 5.3 Comparative empirical study on the numerical stability

Let us now focus on understanding the circumstances under which the implementations of the formulas above may exhibit stability problems. One potential problem is the fact that they are rational, which can lead to the issue of vanishing denominators. This is actually not a problem for the $\phi_i$, since the sum of the weights $w_i$ in (5.1)–(5.6) never vanishes for any $v \in \mathbb{R}^2 \setminus \partial P$ [47] and likewise for the sum of the weights $\hat{w}_i$ in (5.7). But what about the weights themselves? Considering some fixed $k \in \{1, \dots, n\}$, the weight $w_k$ in (5.1), (5.5) or (5.6) is not well-defined, if either $\alpha_{k-1}$ or $\alpha_k$ is equal to $\pm\pi$, or if $v = v_k$, which happens only if $v$ lies on the edges $[v_{k-1}, v_k]$ or $[v_k, v_{k+1}]$. On the other hand, when computing $w_k$ with (5.2), (5.3) or (5.4), we could potentially have problems even inside the polygon. In fact, the areas $A_{k-1,k}$ and $A_{k,k+1}$, as well as the values $\sin\alpha_{i-1}$ and $\sin\alpha_i$, vanish not only on the edges $[v_{k-1}, v_k]$ or $[v_k, v_{k+1}]$, where $\alpha_{k-1}$ or $\alpha_k$ is $\pm\pi$ or $v = v_k$, but also on the entire lines supporting them, where $\alpha_{k-1}$ or $\alpha_k$ equals 0. Based on this initial analysis, it is reasonable to expect that the computation of mean value coordinates is sensitive to rounding errors near the regions where they are not well-defined mathematically. Regarding instead the weight $\hat{w}_k$ in (5.7), even though it is well-defined for any $v \in \mathbb{R}^2$, problems can still arise, for example when subtracting two nearby numbers. This may happen if $D_{k-1,k+1}$ is approximately equal to $r_{k-1}r_{k+1}$ or if $D_{j,j+1}$ is close to $-r_j r_{j+1}$, for some $j \neq k-1, k$, that is, whenever $\alpha_{k-1} + \alpha_k$ is close to zero or some $\alpha_j$ approaches $\pm\pi$. In other words, we expect the weights $\hat{w}_k$ to be unstable when $v$ approaches the set

$$Z_k = \{v \in \mathbb{R}^2 : \hat{w}_k(v) = \phi_k(v) = 0\}, \tag{5.8}$$

| Formula | Instability regions of $w_k$ and $\hat{w}_k$ | Instability regions of $\phi_k$ |
|---|---|---|
| (5.1) | Close to $[v_{k-1}, v_k]$ and $[v_k, v_{k+1}]$ | Close to the boundary |
| (5.2) | Close to lines supporting $[v_{k-1}, v_k]$ and $[v_k, v_{k+1}]$ | Close to all lines supporting the edges |
| (5.3) | Close to lines supporting $[v_{k-1}, v_k]$ and $[v_k, v_{k+1}]$ | Close to all lines supporting the edges |
| (5.4) | Close to lines supporting $[v_{k-1}, v_k]$ and $[v_k, v_{k+1}]$ | Close to all lines supporting the edges |
| (5.5) | Close to $[v_{k-1}, v_k]$ and $[v_k, v_{k+1}]$ | Close to the boundary |
| (5.6) | Close to $[v_{k-1}, v_k]$ and $[v_k, v_{k+1}]$ | Close to the boundary |
| (5.7) | Close to $Z_k = \{v \in \mathbb{R}^2 : \hat{w}_k(v) = \phi_k(v) = 0\}$ | Close to $Z_i$, $i = 1, \dots, n$ |

Table 5.1. Expected instability regions for both weights and mean value coordinates for all formulas and $k \in \{1, \dots, n\}$.

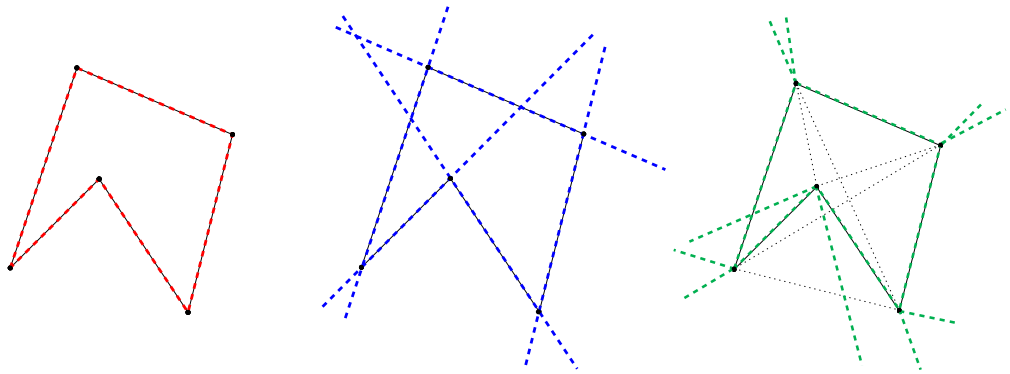

Figure 5.1. Expected instability regions when computing the mean value coordinates for a pentagon using the formulas in (5.1), (5.5),(5.6) (red), (5.2)–(5.4) (blue), and (5.7) (green).

which consists of the edges that are not adjacent to $v_k$ and the line through $v_{k-1}$ and $v_{k+1}$, except for the (open) segment $(v_{k-1}, v_{k+1})$ itself. Therefore, Table 5.1 summarizes this discussion and provides a clear overview of the regions where we expect the weights to be unstable for each formula. Additionally, it does the same for the mean value coordinates, which, as we recall, involve all the weights in their definition. We can also have a visual understanding from Figure 5.1, which highlights the instability regions for each formula when computing the mean value coordinates for a pentagon.

To determine if such scenarios can indeed occur in practice, we examine the behaviour of the mean value coordinates for a specific polygon and visualize the numerical errors introduced by each of the previously mentioned formulas. For a given index $k \in \{1, \dots, n\}$, we compute the absolute error

$$E_a(v) = |\mathrm{fl}(\phi_k(v)) - \phi_k(v)|, \tag{5.9}$$

where $\phi_k(v)$ is the "exact" value computed in multiple-precision (1024 bit) floating-point arithmetic using the MPFR library [33] and $\mathrm{fl}(\phi_k(v))$ is the result of the standard double precision implementation. Let us consider the pentagon in Figure 5.2, which is the same of Figure 5.1, and the index $k \in \{1, 2, 3, 4, 5\}$ of the vertex $v_k$ marked by the magenta dot. We examine the values $E_a(v)$ in (5.9) across a uniform grid of dimension $500 \times 500$ containing the polygon and we compare the results with our expectations, shown in Figure 5.1. The results are obtained for $\phi_k(v)$ computed with all the formulas (5.1)–(5.7) and with our new formula (5.11), which will be introduced in Section 5.4. If $E_a(v)$ is on the order of the machine epsilon, which is ap-

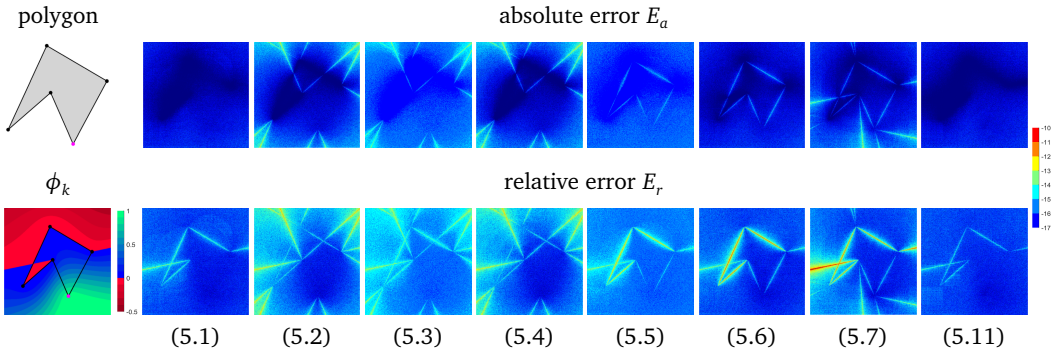


Figure 5.2. Plots of the absolute and relative errors on a $\log_{10}$ scale made by the algorithms that implement formulas (5.1)–(5.7) and (5.11) to evaluate the mean value coordinate $\phi_k$ related to the vertex $v_k$ (magenta dot) for an arbitrary pentagon.

proximately $10^{-16}$ in double precision, then it means that the method is stable for $v$, otherwise it suggests a potential instability. The plots in Figure 5.2 show that the original formula in (5.1) seems to be the only one among the already known formulas that is stable everywhere. This outcome is particularly surprising, because it suggests that this method can effectively handle the division by small numbers in the weights $w_i$, $i = 1,\dots,5$, contrary to our initial expectations. Instead, computing $\phi_k(v)$ with (5.2), (5.3) or (5.4), we observe numerical issues around the lines supporting the edges, but not close to the edges themselves, especially in the first and last cases. While this partially aligns with our prediction of encountering issues along the entire lines, the theoretical analysis on the stability of these formulas explains why we do not have any problems near the edges. In particular, it turns out that the numerical errors introduced by (5.2) and (5.4) are bounded when $v$ approaches the edges, that is, when some $\alpha_i$ is close to $\pm\pi$ (see Corollaries 5.5 and 5.7). This is not the case for (5.3), but, in this specific example, it appears that this method also handles division by small numbers in the weights $w_i$, $i = 1,\dots,5$, relatively well. Then, as expected, the methods resulting from (5.5) and (5.6) have numerical problems near the boundary of the polygon. Finally, formula (5.7) appears to be the least stable as it exhibits a substantial absolute error close to all sets $Z_i$, $i = 1,\dots,5$, which also aligns with our initial considerations.

We further extend the stability analysis and also consider the relative error

$$E_r(v) = \frac{|\mathrm{fl}(\phi_k(v)) - \phi_k(v)|}{|\phi_k(v)|}, \tag{5.10}$$

which, although not mathematically defined on the set $Z_k$, provides valuable insights into the stability compared to the actual magnitude of $|\phi_k|$. This metric offers a zoomed-in view of the domain region where $|\phi_k|$ becomes notably small and indicates how close we can approach $Z_k$ before encountering significant relative errors. In fact, examining the values of $E_r$ in Figure 5.2, we observe that all methods exhibit relatively high errors in the vicinity of the set $Z_k$, but with different divergence rate. In particular, it appears that (5.1) and (5.3) may have potential instability over a wider region near $Z_k$ compared to (5.2) and (5.4), while (5.5), (5.6) and (5.7) confirm to be the worst also in relative terms. Additionally, we observe that the relative error aligns with the information provided by the absolute error in the remaining part of the domain.

To summarize, the original formula (5.1) is the most robust in terms of numerical stability. In fact, despite some suggestions of instability close to $Z_k$ given by the relative error plot, the

formula yields absolutely stable results across the entire domain. However, there exist situations where even the original formula (5.1) can be unstable, and we give an example in Section 5.7.

## 5.4 A new stable formula for mean value coordinates

After observing that all known methods for computing mean value coordinates have some flaw in terms of numerical stability, the goal of our work is to derive a new formula that is potentially stable everywhere. Like Floater [29], we aim to ensure that this new method is defined not only in the interior, but also along the boundary of the polygon. To achieve this, we first employ a similar trick and multiply both the numerator and denominator of the $\phi_i$ in (5.1) by a common constant, which is then included into the redefined weights. In addition, we focus on minimizing operations that are more likely to introduce instability in the results, such as square roots and summations. We now present the new formula and explain how to implement it in a stable way.

**Theorem 5.1.** *The mean value coordinates can be expressed as*

$$\phi_i(v) = \frac{\tilde{w}_i(v)}{\sum_{j=1}^{n} \tilde{w}_j(v)}, \qquad \tilde{w}_i = \sin\frac{\alpha_{i-1}+\alpha_i}{2} \prod_{j\neq i} r_j \prod_{j\neq i-1,i} \cos\frac{\alpha_j}{2}, \qquad i = 1,\dots,n, \tag{5.11}$$

*and this formula is well-defined for all $v \in \mathbb{R}^2$, as long as the signed angle $\alpha_i$ is defined as $\pi$ or $-\pi$ for $v \in (v_i, v_{i+1})$ and in some arbitrary way for $v \in \{v_i, v_{i+1}\}$.*

*Proof.* Starting from (5.1), using the fact that $\tan(\alpha_i/2) = \sin(\alpha_i/2)/\cos(\alpha_i/2)$, and applying the angle sum identity for the sine function, we have

$$w_i = \sin\frac{\alpha_{i-1}+\alpha_i}{2} \Big/ \left( r_i \cos\frac{\alpha_{i-1}}{2} \cos\frac{\alpha_i}{2} \right).$$

We now eliminate the zeros in the denominator by multiplying all $w_i$ by $F = \prod_{i=1}^{n} r_i \cos(\alpha_i/2)$ and, denoting the result by $\tilde{w}_i$, we obtain the new formula (5.11).

This formula is well-defined for any $v \in \mathbb{R}^2 \setminus \partial P$, because both the sum of the $w_i$ and $F$ do not vanish, and the denominator of $\phi_i(v)$ in (5.11) is just $\sum_{j=1}^{n} \tilde{w}_j(v) = F \sum_{j=1}^{n} w_j(v) \neq 0$. Moreover, the formula also works if $v \in \partial P$. On the one hand, if $v$ is a vertex of $P$, that is, $v = v_k$ for some $k \in \{1,\dots,n\}$, then $r_k = 0$ and $r_j \neq 0$ for $j \neq k$, so the only non-vanishing weight is $\tilde{w}_k$ and consequently $\phi_k(v) = 1$ and $\phi_i(v) = 0$ for $i \neq k$. On the other hand, if $v$ lies on an (open) edge of $P$, say $v = (1-\mu)v_k + \mu v_{k+1}$ for some $\mu \in (0,1)$ and some $k \in \{1,\dots,n\}$, then $\alpha_k = \pm\pi$, so that $\sin(\alpha_k/2) = \pm 1$ as well as $\cos(\alpha_k/2) = 0$ and $\cos(\alpha_j/2) \neq 0$ for $j \neq k$. Therefore, all $\tilde{w}_i$ vanish, except for $\tilde{w}_k$ and $\tilde{w}_{k+1}$, which turn out to be

$$\tilde{w}_k = r_{k+1} S, \qquad \tilde{w}_{k+1} = r_k S, \qquad S = \sin\frac{\alpha_k}{2} \prod_{j\neq k,k+1} r_j \prod_{j\neq k} \cos\frac{\alpha_j}{2}.$$

Since $r_k = \mu e_k$ and $r_{k+1} = (1-\mu)e_k$, where $e_k = \|v_{k+1} - v_k\|$, it follows that $\phi_k(v) = 1-\mu$, $\phi_{k+1}(v) = \mu$ and $\phi_i(v) = 0$ for $i \neq k, k+1$. □

Comparing our new formula in (5.11) to the one in (5.7), we observe that it also leads to an $O(n^2)$ algorithm for computing mean value coordinates, but we successfully eliminated all square roots, which can compromise the precision and the efficiency of the method, and we minimized the use of sum operations, as they can introduce numerical cancellation errors. In

**Algorithm 10** Stable implementation of formula (5.11) for computing the mean value coordinates $\phi_1, \ldots, \phi_n$

```
1:  function MVC(v, v_1, ..., v_n)
2:      W := 0
3:      for i = 1, ..., n do                ▷ indices are defined cyclically over [1, ..., n], e.g., v_{n+1} = v_1
4:          β_i := ANGLE(v_{i+1} − v_i, v − v_i)   ▷ ANGLE((a_1, a_2), (b_1, b_2)) returns ATAN2(a_1 b_2 − a_2 b_1, a_1 b_1 + a_2 b_2)
5:          γ_i := ANGLE(v_i − v_{i+1}, v − v_{i+1})
6:          s_i := β_i + γ_i
7:          r_i := ‖v_i − v‖
8:      for i = 1, ..., n do
9:          α_{i−1,i+1} := ANGLE(v_{i−1} − v, v_{i+1} − v)
10:         s_{i−1,i+1} := π · [sign(s_{i−1}) + sign(s_i)] − s_{i−1} − s_i      ▷ s_{i−1,i+1} = α_{i−1} + α_i
11:         if sign(α_{i−1,i+1}) ≠ sign(s_{i−1,i+1}) then      ▷ in this case, α_{i−1,i+1} = s_{i−1,i+1} − 2π · sign(s_{i−1,i+1})
12:             α_{i−1,i+1} := −α_{i−1,i+1}      ▷ sin((α_{i−1} + α_i)/2) = sin(−α_{i−1,i+1}/2)
13:         w_i := r_{i−1} · sin(α_{i−1,i+1}/2)
14:         for j = 1, ..., n do
15:             if j ≠ i − 1, i then
16:                 w_i := w_i · r_j · sin(|s_j|/2)
17:         W := W + w_i
18:     for i = 1, ..., n do
19:         φ_i := w_i/W
20:     return φ_1, ..., φ_n
```

fact, the only sum in (5.11) is $\alpha_{i-1} + \alpha_i \in [-2\pi, 2\pi]$, but we can actually avoid computing this sum by noting that it is equal to the angle at $v$ in the triangle $[v, v_{i-1}, v_{i+1}]$, denoted by $\alpha_{i-1,i+1}$.

In our implementation (see Algorithm 10), we compute the signed angle $\theta$ between two vectors $a = (a_x, a_y)$ and $b = (b_x, b_y)$ using the ATAN2 function as $\theta = \text{ATAN2}(a_x b_y - a_y b_x, a_x b_x + a_y b_y) \in [-\pi, \pi]$. This is fine for all angles $\alpha_i$, but a bit more care is needed in the case of $\alpha_{i-1,i+1}$. Indeed, if $|\alpha_{i-1} + \alpha_i| > \pi$, then the ATAN2 function returns $\alpha_{i-1,i+1} = \alpha_{i-1} + \alpha_i - 2\pi \cdot \text{sign}(\alpha_{i-1} + \alpha_i)$. However, this "mismatch" by $\pm 2\pi$ is detected easily, because the signs of $\alpha_{i-1} + \alpha_i$ and $\alpha_{i-1,i+1}$ differ whenever it happens. And since $\sin((\alpha_{i-1} + \alpha_i)/2) = \sin(-\alpha_{i-1,i+1}/2)$ in this case, we can resolve this problem by changing the sign of $\alpha_{i-1,i+1}$ (cf. lines 11 and 12 in Algorithm 10). Note that the same problem can occur if $|\alpha_{i-1} + \alpha_i| = \pi$, because ATAN2 may return $\pi$ or $-\pi$ in this case, but it can be fixed in the same manner.

Yet, there might still be concerns related to the instability of the cosine function for arguments near zero. To prevent this, denoting by $\beta_i$ and $\gamma_i$ the signed angles at $v_i$ and $v_{i+1}$, respectively, in the triangle $[v, v_i, v_{i+1}]$, we use the fact that

$$\cos\frac{\alpha_i}{2} = \sin\frac{\pi - |\alpha_i|}{2} = \sin\frac{|\beta_i + \gamma_i|}{2}$$

(cf. line 16 in Algorithm 10) and recall that the sine function is stable for arguments near $\pi/2$. Note that computing the sum $s_i = \beta_i + \gamma_i$ is not a problem, because both angles are guaranteed to have the same sign, so that there is no risk of cancellation errors. The price for the improved stability is that we have to compute the $2n$ angles $\beta_i$ and $\gamma_i$ and their sums $s_i$. Note that we still need the angles $\alpha_i$ for determining whether it is necessary to change the sign of $\alpha_{i-1,i+1}$ or not, but once we know $\beta_i$ and $\gamma_i$ we can compute them as $\alpha_i = \pi \cdot \text{sign}(s_i) - s_i$ (cf. line 10 in Algorithm 10).

The numerical stability of this algorithm can be observed in Figure 5.2, which confirms that our new formula performs best, even if compared to the result using (5.1), especially close to the region $Z_k$.

So far, we have discussed the numerical stability of the different formulas and supported our claims only with empirical evidence. In the next section, we conduct a mathematical analysis on the numerical stability of mean value coordinates and provide a more formal explanation for our observations.

## 5.5 Theoretical analysis of the numerical stability

As already seen in Chapter 3, a common procedure to theoretically analyse the numerical stability of an algorithm is to establish an upper bound on the relative forward error and to examine its magnitude. In the specific context of mean value coordinates, we need to study the error $E_r$ in (5.10). It is worth noting that bounding $E_r$ from above also gives an upper bound on the absolute error in (5.9), because

$$E_a(v) = E_r(v)|\phi_k(v)|. \tag{5.12}$$

To this end, we can use Theorem 3.1 that establishes an upper bound on the relative error of any function that can be expressed in the form (3.8), that is,

$$r(x) = \frac{\sum_{i=0}^{n} a_i(x) f_i}{\sum_{j=0}^{m} b_j(x)}$$

for some data values $f_i$ and functions $a_i$ and $b_j$, $i = 0, \dots, n$ and $j = 0, \dots, m$. Therefore, we can use this result also for the mean value coordinates

$$\phi_i(v) = \frac{w_i(v)}{\sum_{j=1}^{n} w_j(v)}, \qquad i = 1, \dots, n, \tag{5.13}$$

as their formula fits the expression in (3.8) for $n = 0$, $a_0 = w_i$, $f_0 = 1$, $m = n-1$ and $b_j = w_{i+1}$. Before proceeding, we note that this stability analysis does not account for any errors arising from the initial rounding of the given values to floating-point numbers.

**Corollary 5.2.** *Assume that there exist $\delta_1, \dots, \delta_n \in \mathbb{R}$ with*

$$\mathrm{fl}(w_i(v)) = w_i(v)(1+\delta_i), \qquad |\delta_i| \le D\epsilon + O(\epsilon^2), \qquad i = 1, \dots, n \tag{5.14}$$

*for some constant $D$. Then, assuming that the input values $v_i$ and $v$ are given as floating-point numbers, the relative forward error of the mean value coordinates in* (5.13) *satisfies*

$$\frac{|\mathrm{fl}(\phi_i(v)) - \phi_i(v)|}{|\phi_i(v)|} \le (1+D)\epsilon + (n-1+D)W(v)\epsilon + O(\epsilon^2), \tag{5.15}$$

*where*

$$W(v) = \frac{\sum_{i=1}^{n} |w_i(v)|}{|\sum_{i=1}^{n} w_i(v)|}, \tag{5.16}$$

*for $\epsilon$ small enough.*

Hence, the numerical stability of the mean value coordinates depends on the constant $D$ and the function $W$. As the latter is the same for all the different formulas, what distinguishes their performance in terms of numerical stability is the upper bound $D$ on the relative error associated with the weights $w_i$. Considering the new formula, it can be proven that the constant $D$ related to the weights $\tilde{w}_i$ is always small, while, for all the other formulas, it can be large (see Section 5.6), which agrees with what we observed in Section 5.3.

Finally, we consider two arbitrary vectors $a = (a_x, a_y)$ and $b = (b_x, b_y)$ and present the upper bounds on the relative forward errors of some quantities that we frequently use.

1. Considering the *radius* $r_i = \|v - v_i\|$ for some $i \in \{1, \dots, n\}$, it follows from Theorem 3.1 and (2.19) that there exists some $\rho_i \in \mathbb{R}$, such that $\mathrm{fl}(r_i) = r_i(1 + \rho_i)$ with
$$|\rho_i| \leq (2 + D_{\mathrm{sqrt}})\epsilon + O(\epsilon^2), \qquad i = 1, \dots, n. \tag{5.17}$$

2. Considering the *dot product* $D_{a,b} = a_x b_x + a_y b_y$ between $a$ and $b$, it follows from Theorem 3.1 that there exists some $\delta_{a,b} \in \mathbb{R}$, such that $\mathrm{fl}(D_{a,b}) = D_{a,b}(1 + \delta_{a,b})$ with
$$|\delta_{a,b}| \leq u(D_{a,b})\epsilon + O(\epsilon^2), \qquad u(D_{a,b}) = 4\frac{|a_x b_x| + |a_y b_y|}{|D_{a,b}|}. \tag{5.18}$$
It is important to note that the relative forward error becomes unreliable when the computed quantity approaches zero, as dividing by a small value can result in a significantly large error. In such cases, the right quantity to consider is the absolute forward error, which is given by $|D_{a,b}\delta_{a,b}|$ and, since $|a_x b_x| + |a_y b_y| \leq 2\|a\|\|b\|$, it is bounded from above by $8\|a\|\|b\|\epsilon + O(\epsilon^2)$. Hence, it is reasonable to expect that the computation of $D_{a,b}$ is generally stable, although its upper bound on the forward error may increase when the values $\|a\|$ and $\|b\|$ become large.

3. Considering the 2D *cross product* $C_{a,b} = (a_x b_y - a_x b_y)$ between $a$ and $b$, which is twice the signed area of the triangle $[0, a, b]$, it follows from Theorem 3.1 that there exists some $\gamma_{a,b} \in \mathbb{R}$, such that $\mathrm{fl}(C_{a,b}) = C_{a,b}(1 + \gamma_{a,b})$ with
$$|\gamma_{a,b}| \leq u(C_{a,b})\epsilon + O(\epsilon^2), \qquad u(C_{a,b}) = 4\frac{|a_x b_y| + |a_y b_x|}{|C_{a,b}|}. \tag{5.19}$$
As in the case of the dot product, it may happen that this upper bound is big when the values $\|a\|$ and $\|b\|$ are large, but in general we assume that the computation of $C_{a,b}$ is stable.

4. Considering the *signed angle* $\theta_{a,b} = \arctan(D_{a,b}/C_{a,b})$ between $a$ and $b$, it follows from Theorem 3.1, the previous observations, and (2.17) that there exists some $\sigma_{a,b} \in \mathbb{R}$, such that $\mathrm{fl}(\theta_{a,b}) = \theta_{a,b}(1 + \sigma_{a,b})$ with
$$|\sigma_{a,b}| \leq u(\theta_{a,b})\epsilon + O(\epsilon^2), \qquad u(\theta_{a,b}) = u(D_{a,b}) + u(C_{a,b}) + 1 + D_{\arctan}. \tag{5.20}$$

## 5.6 Error analysis of all formulas

We begin by observing that all the weights $w_i(v)$ in (5.1)–(5.6), $\hat{w}_i(v)$ in (5.7) and $\tilde{w}_i(v)$ in (5.11) can be written in the general form
$$w(v) = \prod_{j=1}^{J} \sum_{k=1}^{K} x_{j,k}(v), \tag{5.21}$$

for some $J, K \in \mathbb{N}$. Thus, we first derive a general bound on the relative forward error for the function $w$ in (5.21) and then apply this result in the specific case of the all the weights mentioned before.

**Theorem 5.3.** *Suppose that there exist $\chi_{j,k} \in \mathbb{R}$, $j = 1, \dots, J$ and $k = 1, \dots, K$, with*

$$\mathrm{fl}(x_{j,k}(v)) = x_{j,k}(v)(1+\chi_{j,k}), \qquad |\chi_{j,k}| \le X_{j,k}\epsilon + O(\epsilon^2),$$

*for some positive constants $X_{j,k}$, $j = 1, \dots, J$ and $k = 1, \dots, K$. Then there exists some $\delta \in \mathbb{R}$, such that $w$ in* (5.21) *satisfies $\mathrm{fl}(w(v)) = w(v)(1+\delta)$ and $|\delta| \le D\epsilon + O(\epsilon^2)$, where*

$$D = \sum_{j=1}^{J} \frac{\sum_{k=1}^{K} |x_{j,k}(v)|(K-1+X_{j,k})}{\left|\sum_{k=1}^{K} x_{j,k}(v)\right|} + J - 1.$$

*Proof.* We first notice that $\mathrm{fl}(w(v))$ is given by

$$\mathrm{fl}(w(v)) = \prod_{j=1}^{J} \sum_{k=1}^{K} [x_{j,k}(v)(1+\chi_{j,k})(1+\delta_{j,k}^{+})](1+\delta^{\times}),$$

where $\delta_{j,k}^{+}$ and $\delta^{\times}$ are the relative errors introduced by the $K-1$ sums and the $J-1$ products, respectively, so they satisfy

$$|\delta_{j,k}^{+}| \le (K-1)\epsilon + O(\epsilon^2) \qquad \text{and} \qquad |\delta^{\times}| \le (J-1)\epsilon + O(\epsilon^2). \tag{5.22}$$

Consequently, there exist some $\eta_{j,k} \in \mathbb{R}$ with

$$|\eta_{j,k}| \le (K-1+X_{j,k})\epsilon + O(\epsilon^2), \qquad j = 1, \dots, J, \quad k = 1, \dots, K,$$

such that

$$\begin{aligned}\mathrm{fl}(w(v)) &= \prod_{j=1}^{J} \sum_{k=1}^{K} [x_{j,k}(v)(1+\eta_{j,k})](1+\delta^{\times}) = \prod_{j=1}^{J} \left[ \sum_{k=1}^{K} x_{j,k}(v) \left( 1 + \frac{\sum_{k=1}^{K} x_{j,k}(v)\eta_{j,k}}{\sum_{k=1}^{K} x_{j,k}(v)} \right) \right] (1+\delta^{\times}) \\ &= w(v) \left( 1 + \sum_{j=1}^{J} \frac{\sum_{k=1}^{K} x_{j,k}(v)\eta_{j,k}}{\sum_{k=1}^{K} x_{j,k}(v)} + \delta^{\times} + O(\epsilon^2) \right).\end{aligned}$$

Therefore, $\delta = \sum_{j=1}^{J} \sum_{k=1}^{K} x_{j,k}(v)\eta_{j,k} / \sum_{k=1}^{K} x_{j,k}(v) + \delta^{\times} + O(\epsilon^2)$, and the statement follows immediately by using the triangle inequality, (5.22), and (3.12). □

**Corollary 5.4.** *For any $v \in \mathbb{F}^2$ and $v_1, \dots, v_n \in \mathbb{F}^2$, there exist $\delta_1, \dots, \delta_n \in \mathbb{R}$, such that the $w_i$ in* (5.1) *satisfy $\mathrm{fl}(w_i(v)) = w_i(v)(1+\delta_i)$ and $|\delta_i| \le D\epsilon + O(\epsilon^2)$ for $i = 1, \dots, n$, where*

$$D = \max_{i=1,\dots,n} F_i(1 + \pi u(\alpha_i) + D_{\tan}) + 5 + D_{\mathrm{sqrt}} \tag{5.23}$$

*and*

$$F_i = \max_{v \in \mathbb{F}^2} \left| \sin \frac{\alpha_{i-1} + \alpha_i}{2} \cos \frac{\alpha_{i-1}}{2} \cos \frac{\alpha_i}{2} \right|^{-1}.$$

*Proof.* We note that $w_i$ in (5.1) can be written as in (5.21) for $J = K = 2$ and

$$x_{1,1} = \tan\frac{\alpha_{i-1}}{2}, \qquad x_{1,2} = \tan\frac{\alpha_i}{2},$$
$$x_{2,1} = \frac{1}{r_i}, \qquad x_{2,2} = 0.$$

It then follows from (5.17), (5.20), and (2.18) that $\mathrm{fl}(x_{j,k}) = x_{j,k}(1+\chi_{j,k})$ with $|\chi_{j,k}| \le X_{j,k}\epsilon + O(\epsilon^2)$ and

$$X_{1,1} = \frac{|\alpha_{i-1}|}{|\sin\alpha_{i-1}|}u(\alpha_{i-1}) + D_{\tan}, \qquad X_{1,2} = \frac{|\alpha_i|}{|\sin\alpha_i|}u(\alpha_i) + D_{\tan},$$
$$X_{2,1} = 3 + D_{\mathrm{sqrt}}, \qquad X_{2,2} = 0.$$

Therefore, we can use Theorem 5.3 to obtain $\mathrm{fl}(w_i(v)) = w_i(v)(1+\delta_i)$ with $|\delta_i| \le D_i\epsilon + O(\epsilon^2)$ and

$$\begin{aligned}
D_i &= \frac{|x_{1,1}|(1+X_{1,1}) + |x_{1,2}|(1+X_{1,2})}{|x_{1,1}+x_{1,2}|} + X_{2,1} + 2 \\
&= \frac{\displaystyle\sum_{j=i-1,i} |\tan(\alpha_j/2)|\left(1 + \frac{|\alpha_j|}{|\sin\alpha_j|}u(\alpha_j) + D_{\tan}\right)}{|\tan(\alpha_{i-1}/2) + \tan(\alpha_i/2)|} + 5 + D_{\mathrm{sqrt}} \\
&\le \frac{\displaystyle\sum_{j=i-1,i} \frac{|\tan(\alpha_j/2)|}{|\sin\alpha_j|}(1 + |\alpha_j|u(\alpha_j) + D_{\tan})}{|\tan(\alpha_{i-1}/2) + \tan(\alpha_i/2)|} + 5 + D_{\mathrm{sqrt}} \\
&\le \frac{\displaystyle\sum_{j=i-1,i} \frac{|\tan(\alpha_j/2)|}{|\sin\alpha_j|}}{|\tan(\alpha_{i-1}/2) + \tan(\alpha_i/2)|}(1 + \pi\max\{u(\alpha_{i-1}), u(\alpha_i)\} + D_{\tan}) + 5 + D_{\mathrm{sqrt}}.
\end{aligned}$$

Finally, we use the double-angle formula for the sine function and get

$$\frac{\displaystyle\sum_{j=i-1,i} \frac{|\tan(\alpha_j/2)|}{|\sin\alpha_j|}}{|\tan(\alpha_{i-1}/2) + \tan(\alpha_i/2)|} = \frac{1}{2}\frac{\cos^2(\alpha_{i-1}/2) + \cos^2(\alpha_i/2)}{|\sin(\alpha_{i-1}+\alpha_i)/2\cos(\alpha_{i-1}/2)\cos(\alpha_i/2)|} \le F_i,$$

which gives $D_i \le D$ for $D$ in (5.23). □

For the following statements, let $d_k = v_k - v$, $k = 1,\dots,n$ and let $C_{i,j}$ denote the cross product of $d_i$ and $d_j$, which we denoted by $C_{d_i,d_j}$ before.

**Corollary 5.5.** *For any $v \in \mathbb{F}^2$ and $v_1,\dots,v_n \in \mathbb{F}^2$, there exist $\delta_1,\dots,\delta_n \in \mathbb{R}$, such that the $w_i$ in* (5.2) *satisfy* $\mathrm{fl}(w_i(v)) = w_i(v)(1+\delta_i)$ *and* $|\delta_i| \le D\epsilon + O(\epsilon^2)$ *for* $i = 1,\dots,n$, *where*

$$D = \max_{i=1,\dots,n} \frac{3}{4}F_i\big(5 + D_{\mathrm{sqrt}} + \max\{u(C_{i,i+1}), u(C_{i-1,i+1}), u(C_{i-1,i})\}\big) + u(C_{i,i+1}) + u(C_{i-1,i}) + 8 \quad (5.24)$$

*and*

$$F_i = \max_{v\in\mathbb{F}^2}\left|\sin\frac{\alpha_{i-1}+\alpha_i}{2}\sin\frac{\alpha_{i-1}}{2}\sin\frac{\alpha_i}{2}\right|^{-1}.$$

*Proof.* We note that the $w_i$ in (5.2) can be written as in (5.21) for $J = 3$, $K = 3$ and

$$\begin{aligned}
&x_{1,1} = r_{i-1}A_{i,i+1}, && x_{1,2} = -r_i A_{i-1,i+1}, && x_{1,3} = r_{i+1}A_{i-1,i},\\
&x_{2,1} = \frac{1}{A_{i-1,i}}, && x_{2,2} = 0, && x_{2,3} = 0,\\
&x_{3,1} = \frac{1}{A_{i,i+1}}, && x_{3,2} = 0, && x_{3,3} = 0.
\end{aligned}$$

It then follows from (5.17) and (5.19) that $\mathrm{fl}(x_{j,k}) = x_{j,k}(1+\chi_{j,k})$ with $|\chi_{j,k}| \le X_{j,k}\epsilon + O(\epsilon^2)$ and

$$\begin{aligned}
&X_{1,1} = 3 + D_{\mathrm{sqrt}} + u(C_{i,i+1}), && X_{1,2} = 3 + D_{\mathrm{sqrt}} + u(C_{i-1,i+1}), && X_{1,3} = 3 + D_{\mathrm{sqrt}} + u(C_{i-1,i}),\\
&X_{2,1} = 1 + u(C_{i-1,i}), && X_{2,2} = 0, && X_{2,3} = 0,\\
&X_{3,1} = 1 + u(C_{i,i+1}), && X_{3,2} = 0, && X_{3,3} = 0.
\end{aligned}$$

Therefore, we can use Theorem 5.3 to obtain $\mathrm{fl}(w_i(v)) = w_i(v)(1+\delta_i)$ with $|\delta_i| \le D_i\epsilon + O(\epsilon^2)$ and

$$\begin{aligned}
D_i &= \frac{|x_{1,1}|(2+X_{1,1}) + |x_{1,2}|(2+X_{1,2}) + |x_{1,3}|(2+X_{1,3})}{|x_{1,1}+x_{1,2}+x_{1,3}|} + X_{2,1} + X_{3,1} + 6\\
&\le \frac{\sum_{k=1,3}|x_{1,k}|}{|\sum_{k=1,3}x_{1,k}|}(5 + D_{\mathrm{sqrt}} + \max\{u(C_{i,i+1}), u(C_{i-1,i+1}), u(C_{i-1,i})\}) + u(C_{i,i+1}) + u(C_{i-1,i}) + 8.
\end{aligned}$$

Finally, we use some trigonometric identities to obtain

$$\begin{aligned}
\frac{\sum_{k=1,3}|x_{1,k}|}{|\sum_{k=1,3}x_{1,k}|} &= \frac{|\sin\alpha_{i-1}| + |\sin(\alpha_{i-1}+\alpha_i)| + |\sin\alpha_i|}{|\sin\alpha_{i-1} - \sin(\alpha_{i-1}+\alpha_i) + \sin\alpha_i|}\\
&\le \frac{3}{|\sin\alpha_{i-1} - \sin(\alpha_{i-1}+\alpha_i) + \sin\alpha_i|}\\
&= \frac{3}{4|\sin((\alpha_i+\alpha_{i-1})/2)\sin(\alpha_{i-1}/2)\sin(\alpha_i/2)|} = \frac{3}{4}F_i,
\end{aligned}$$

which gives $D_i \le D$ for $D$ in (5.24). □

**Corollary 5.6.** *For any $v \in \mathbb{F}^2$ and $v_1, \dots, v_n \in \mathbb{F}^2$, there exist $\delta_1, \dots, \delta_n \in \mathbb{R}$, such that the $w_i$ in* (5.3) *and* (5.5) *satisfy* $\mathrm{fl}(w_i(v)) = w_i(v)(1+\delta_i)$ *and* $|\delta_i| \le D\epsilon + O(\epsilon^2)$ *for* $i = 1, \dots, n$, *where*

$$D = \max_{i=1,\dots,n} F_i\left(3 + \frac{3}{2}\pi\max\{u(\alpha_{i-1}), u(\alpha_i)\} + \frac{D_{\cos}}{2} + D_{\sin}\right) + 5 + D_{\mathrm{sqrt}} \tag{5.25}$$

*and*

$$F_i = \max_{v\in\mathbb{F}^2}\begin{cases}
\left|\sin\dfrac{\alpha_{i-1}+\alpha_i}{2}\sin\dfrac{\alpha_{i-1}}{2}\sin\dfrac{\alpha_i}{2}\sin\alpha_{i-1}\sin\alpha_i\right|^{-1}, & \text{for } w_i \text{ in (5.3)},\\
4\left|\sin\dfrac{\alpha_{i-1}+\alpha_i}{2}\cos\dfrac{\alpha_{i-1}}{2}\cos\dfrac{\alpha_i}{2}(1+\cos\alpha_{i-1})(1+\cos\alpha_i)\right|^{-1}, & \text{for } w_i \text{ in (5.5)}.
\end{cases}$$

*Proof.* The proof is carried out for the computation of $w_i(v)$ with formula (5.3), but similar arguments can be applied to the case of the weights $w_i(v)$ in (5.5).

We note that $w_i$ in (5.3) can be written as in (5.21) for $J = K = 2$ and

$$x_{1,1} = \frac{1-\cos\alpha_{i-1}}{\sin\alpha_{i-1}}, \qquad x_{1,2} = \frac{1-\cos\alpha_i}{\sin\alpha_i},$$
$$x_{2,1} = \frac{1}{r_i}, \qquad x_{2,2} = 0.$$

It then follows from Theorem 3.1 and (5.17), (5.20), (2.15) and (2.16) that $\mathrm{fl}(x_{j,k}) = x_{j,k}(1+\chi_{j,k})$ with $|\chi_{j,k}| \le X_{j,k}\epsilon + O(\epsilon^2)$ and

$$\begin{aligned}
X_{1,1} &= \frac{2+|\cos\alpha_{i-1}|(|\tan\alpha_{i-1}||\alpha_{i-1}|u(\alpha_{i-1})+D_{\cos}+2)}{|1-\cos\alpha_{i-1}|} + |\cot\alpha_{i-1}||\alpha_{i-1}|u(\alpha_{i-1})+D_{\sin},\\
X_{1,2} &= \frac{2+|\cos\alpha_i|(|\tan\alpha_i||\alpha_i|u(\alpha_i)+D_{\cos}+2)}{|1-\cos\alpha_i|} + |\cot\alpha_i||\alpha_i|u(\alpha_i)+D_{\sin},\\
X_{2,1} &= 3+D_{\mathrm{sqrt}},\\
X_{2,2} &= 0.
\end{aligned}$$

Therefore, we can use Theorem 5.3 to obtain $\mathrm{fl}(w_i(v)) = w_i(v)(1+\delta_i)$ with $|\delta_i| \le D_i\epsilon + O(\epsilon^2)$ and

$$\begin{aligned}
D_i &= \frac{|x_{1,1}|(1+X_{1,1})+|x_{1,2}|(1+X_{1,2})}{|x_{1,1}+x_{1,2}|} + X_{2,1} + 2\\
&= \frac{\displaystyle\sum_{j=i-1,i}\left|\frac{1-\cos\alpha_j}{\sin\alpha_j}\right|\left(1+\frac{2+|\cos\alpha_j|(|\tan\alpha_j||\alpha_j|u(\alpha_j)+D_{\cos}+2)}{|1-\cos\alpha_j|}+|\cot\alpha_j||\alpha_j|u(\alpha_j)+D_{\sin}\right)}{|(1-\cos\alpha_{i-1})/\sin\alpha_{i-1}+(1-\cos\alpha_i)/\sin\alpha_i|}\\
&\quad + 5 + D_{\mathrm{sqrt}}\\
&\le \frac{\displaystyle\sum_{j=i-1,i}\left|\frac{1-\cos\alpha_j}{\sin\alpha_j}\right|\frac{6+\pi u(\alpha_j)+D_{\cos}+2/|\sin\alpha_j|\pi u(\alpha_j)+2D_{\sin}}{|1-\cos\alpha_j|}}{|(1-\cos\alpha_{i-1})/\sin\alpha_{i-1}+(1-\cos\alpha_i)/\sin\alpha_i|} + 5 + D_{\mathrm{sqrt}}\\
&\le \frac{\displaystyle\sum_{j=i-1,i}\left|\frac{1-\cos\alpha_j}{\sin\alpha_j}\right|\frac{6+3\pi u(\alpha_j)+D_{\cos}+2D_{\sin}}{|1-\cos\alpha_j||\sin\alpha_j|}}{|(1-\cos\alpha_{i-1})/\sin\alpha_{i-1}+(1-\cos\alpha_i)/\sin\alpha_i|} + 5 + D_{\mathrm{sqrt}}\\
&\le \frac{\displaystyle\sum_{j=i-1,i}\frac{1}{\sin^2\alpha_j}}{|(1-\cos\alpha_{i-1})/\sin\alpha_{i-1}+(1-\cos\alpha_i)/\sin\alpha_i|}(6+3\pi\max\{u(\alpha_{i-1}),u(\alpha_i)\}+D_{\cos}+2D_{\sin})\\
&\quad + 5 + D_{\mathrm{sqrt}}.
\end{aligned}$$

Finally, we note that

$$\frac{\sum_{j=i-1,i} \frac{1}{\sin^2 \alpha_j}}{|(1-\cos\alpha_{i-1})/\sin\alpha_{i-1} + (1-\cos\alpha_i)/\sin\alpha_i|} \le \frac{2|\sin\alpha_{i-1}\sin\alpha_i|^{-1}}{|\sin\alpha_{i-1}+\sin\alpha_i - \sin(\alpha_{i-1}+\alpha_i)|} = \frac{|\sin\alpha_{i-1}\sin\alpha_i|^{-1}}{2\left|\sin\left(\frac{\alpha_i+\alpha_{i-1}}{2}\right)\sin\left(\frac{\alpha_{i-1}}{2}\right)\sin\left(\frac{\alpha_i}{2}\right)\right|},$$

which gives $D_i \le D$ for $D$ in (5.25). □

**Corollary 5.7.** *For any $v \in \mathbb{F}^2$ and $v_1, \dots, v_n \in \mathbb{F}^2$, there exist $\delta_1, \dots, \delta_n \in \mathbb{R}$, such that the $w_i$ in* (5.4) *and* (5.6) *satisfy* $\mathrm{fl}(w_i(v)) = w_i(v)(1+\delta_i)$ *and* $|\delta_i| \le D\epsilon + O(\epsilon^2)$ *for* $i = 1, \dots, n$, *where*

$$D = \max_{i=1,\dots,n} F_i \max_{j=i-1,i} \Big(1 + \max\{7 + 2D_{\mathrm{sqrt}}, 2 + u(D_{j,j+1})\} + u(C_{j,j+1})\Big) + 5 + D_{\mathrm{sqrt}} \tag{5.26}$$

*with*

$$F_i = \max_{v\in\mathbb{F}^2} \begin{cases} \left|\sin\frac{\alpha_{i-1}+\alpha_i}{2}\sin\frac{\alpha_{i-1}}{2}\sin\frac{\alpha_i}{2}\right|^{-1}, & \text{for } w_i \text{ in (5.4)},\\ 2\left|\sin\frac{\alpha_{i-1}+\alpha_i}{2}\cos\frac{\alpha_{i-1}}{2}\cos\frac{\alpha_i}{2}(1+\cos\alpha_{i-1})(1+\cos\alpha_i)\right|^{-1}, & \text{for } w_i \text{ in (5.6)}. \end{cases}$$

*Proof.* The proof is carried out for the computation of $w_i(v)$ with formula (5.4), but similar arguments can be applied to the case of the weights $w_i(v)$ in (5.6).

We note that $w_i$ in (5.4) can be written as in (5.21) for $J = K = 2$ and

$$x_{1,1} = \frac{r_{i-1}r_i - D_{i-1,i}}{2A_{i-1,i}}, \qquad x_{1,2} = \frac{r_i r_{i+1} - D_{i,i+1}}{2A_{i,i+1}},$$
$$x_{2,1} = \frac{1}{r_i}, \qquad x_{2,2} = 0.$$

It then follows from Theorem 3.1 and (5.17)–(5.19) that $\mathrm{fl}(x_{j,k}) = x_{j,k}(1+\chi_{j,k})$ with $|\chi_{j,k}| \le X_{j,k}\epsilon + O(\epsilon^2)$ and

$$X_{1,1} = \frac{r_{i-1}r_i(7+2D_{\mathrm{sqrt}}) + |D_{i-1,i}|(2+u(D_{i-1,i}))}{|r_{i-1}r_i - D_{i-1,i}|} + u(C_{i-1,i}),$$
$$X_{1,2} = \frac{r_i r_{i+1}(7+2D_{\mathrm{sqrt}}) + |D_{i,i+1}|(2+u(D_{i,i+1}))}{|r_i r_{i+1} - D_{i,i+1}|} + u(C_{i,i+1}),$$
$$X_{2,1} = 3 + D_{\mathrm{sqrt}},$$
$$X_{2,2} = 0.$$

Therefore, we can use Theorem 5.3 to obtain $\mathrm{fl}(w_i(v)) = w_i(v)(1+\delta_i)$ with $|\delta_i| \le D_i\epsilon + O(\epsilon^2)$

and

$$\begin{aligned}
D_i &= \frac{|x_{1,1}|(1+X_{1,1})+|x_{1,2}|(1+X_{1,2})}{|x_{1,1}+x_{1,2}|} + X_{2,1} + 2 \\
&= \frac{\displaystyle\sum_{j=i-1,i} \left|\frac{r_j r_{j+1} - D_{j,j+1}}{2A_{j,j+1}}\right| \left(1 + \frac{|r_j r_{j+1}|(7+2D_{\mathrm{sqrt}}) + |D_{j,j+1}|(2+u(D_{j,j+1}))}{|r_j r_{j+1} - D_{j,j+1}|} + u(C_{j,j+1})\right)}{|(r_{i-1}r_i - D_{i-1,i})/(2A_{i-1,i}) + (r_i r_{i+1} - D_{i,i+1})/(2A_{i,i+1})|} \\
&\quad + 5 + D_{\mathrm{sqrt}} \\
&\le \frac{\displaystyle\sum_{j=i-1,i} \frac{r_j r_{j+1} + |D_{j,j+1}|}{2|A_{j,j+1}|} \Big(1 + \max\{7+2D_{\mathrm{sqrt}}, 2+u(D_{j,j+1})\} + u(C_{j,j+1})\Big)}{|(r_{i-1}r_i - D_{i-1,i})/(2A_{i-1,i}) + (r_i r_{i+1} - D_{i,i+1})/(2A_{i,i+1})|} + 5 + D_{\mathrm{sqrt}} \\
&= \frac{\displaystyle\sum_{j=i-1,i} \frac{1+|\cos\alpha_j|}{|\sin\alpha_j|} \Big(1 + \max\{7+2D_{\mathrm{sqrt}}, 2+u(D_{j,j+1})\} + u(C_{j,j+1})\Big)}{|(1-\cos\alpha_{i-1})/\sin\alpha_{i-1} + (1-\cos\alpha_i)/\sin\alpha_i|} + 5 + D_{\mathrm{sqrt}}.
\end{aligned}$$

Finally, we use some trigonometric identities and observe that

$$\begin{aligned}
\frac{\displaystyle\sum_{j=i-1,i} \frac{1+|\cos\alpha_j|}{|\sin\alpha_j|}}{|(1-\cos\alpha_{i-1})/\sin\alpha_{i-1} + (1-\cos\alpha_i)/\sin\alpha_i|} &\le \frac{4}{|\sin\alpha_{i-1} + \sin\alpha_i - \sin(\alpha_{i-1}+\alpha_i)|} \\
&= \frac{1}{|\sin((\alpha_i+\alpha_{i-1})/2)\sin(\alpha_{i-1}/2)\sin(\alpha_i/2)|} \\
&= F_i,
\end{aligned}$$

which gives $D_i \le D$ for $D$ in (5.26). □

**Corollary 5.8.** *For any $v \in \mathbb{F}^2$ and $v_1, \dots, v_n \in \mathbb{F}^2$, there exist $\delta_1, \dots, \delta_n \in \mathbb{R}$, such that the $\hat{w}_i$ in* (5.7) *satisfy* $\mathrm{fl}(\hat{w}_i(v)) = \hat{w}_i(v)(1+\delta_i)$ *and* $|\delta_i| \le D\epsilon + O(\epsilon^2)$ *for* $i = 1, \dots, n$, *where*

$$D = \max_{i=1,\dots,n} F_i \max\{7+2D_{\mathrm{sqrt}}, 2+u(D_{i-1,i+1}), 2+\max_{j\neq i-1,i} u(D_{j,j+1})\} + (n-1)D_{\mathrm{sqrt}} + n - 2 \quad (5.27)$$

*and*

$$F_i = \max_{v\in\mathbb{F}^2} \left( |1-\cos(\alpha_{i-1}+\alpha_i)|^{-1} + \sum_{j\neq i-1,i} |1+\cos\alpha_j|^{-1} \right).$$

*Proof.* We note that the $\hat{w}_i$ in (5.7), neglecting the signs $\delta_i$, can be written as in (5.21) for $J = n-1$, $K = 1$ and

$$x_{j,1} = \begin{cases} \sqrt{r_{i-1}r_{i+1} - D_{i-1,i+1}}, & j = 1 \\ \sqrt{r_{j-1}r_j + D_{j-1,j}}, & j = 2, \dots, i-1, \\ \sqrt{r_j r_{j+1} + D_{j,j+1}}, & j = i+1, \dots, n. \end{cases}$$

It then follows from Theorem 3.1 and (2.19)–(5.18) that $\mathrm{fl}(x_{j,1}) = x_{j,1}(1+\chi_{j,1})$ with $|\chi_{j,1}| \leq X_{j,1}\epsilon + O(\epsilon^2)$ and

$$X_{j,1} = \begin{cases} \frac{r_{i-1}r_{i+1}(7+2D_{\mathrm{sqrt}})+|D_{i-1,i+1}|(2+u(D_{i-1,i+1}))}{2|r_{i-1}r_{i+1}-D_{i-1,i+1}|} + D_{\mathrm{sqrt}}, & j=1, \\ \frac{r_{j-1}r_j(7+2D_{\mathrm{sqrt}})+|D_{j-1,j}|(2+u(D_{j-1,j}))}{2|r_{j-1}r_j+D_{j-1,j}|} + D_{\mathrm{sqrt}}, & j=2,\dots,i-1, \\ \frac{r_j r_{j+1}(7+2D_{\mathrm{sqrt}})+|D_{j,j+1}|(2+u(D_{j,j+1}))}{2|r_j r_{j+1}+D_{j,j+1}|} + D_{\mathrm{sqrt}}, & j=i+1,\dots,n. \end{cases}$$

Therefore, we can use Theorem 5.3 to get $\mathrm{fl}(\hat{w}_i(v)) = \hat{w}_i(v)(1+\delta_i)$ with $|\delta_i| \leq D_i\epsilon + O(\epsilon^2)$ and

$$\begin{aligned} D_i &= \sum_{j=1}^{n} X_{j,1} + n - 2 \\ &\leq \frac{r_{i-1}r_{i+1}+|D_{i-1,i+1}|}{2|r_{i-1}r_{i+1}-D_{i-1,i+1}|}\max\{7+2D_{\mathrm{sqrt}}, 2+u(D_{i-1,i+1})\} \\ &\quad + \sum_{j\neq i-1,i} \frac{r_j r_{j+1}+|D_{j,j+1}|}{2|r_j r_{j+1}+D_{j,j+1}|}\max\{7+2D_{\mathrm{sqrt}}, 2+u(D_{j,j+1})\} + (n-1)D_{\mathrm{sqrt}} + n - 2 \\ &= \frac{1+|\cos(\alpha_{i-1}+\alpha_i)|}{2|1-\cos(\alpha_{i-1}+\alpha_i)|}\max\{7+2D_{\mathrm{sqrt}}, 2+u(D_{i-1,i+1})\} \\ &\quad + \sum_{j\neq i-1,i} \frac{1+|\cos\alpha_j|}{2|1+\cos\alpha_j|}\max\{7+2D_{\mathrm{sqrt}}, 2+u(D_{j,j+1})\} + (n-1)D_{\mathrm{sqrt}} + n - 2 \\ &\leq \frac{\max\{7+2D_{\mathrm{sqrt}}, 2+u(D_{i-1,i+1})\}}{|1-\cos(\alpha_{i-1}+\alpha_i)|} + \sum_{j\neq i-1,i} \frac{\max\{7+2D_{\mathrm{sqrt}}, 2+u(D_{j,j+1})\}}{|1+\cos\alpha_j|} + (n-1)D_{\mathrm{sqrt}} \\ &\quad + n - 2, \end{aligned}$$

which proves the statement. □

**Corollary 5.9.** *For any $v \in \mathbb{F}^2$ and $v_1,\dots,v_n \in \mathbb{F}^2$, there exist $\delta_1,\dots,\delta_n \in \mathbb{R}$, such that the $\tilde{w}_i$ in* (5.11) *satisfy $\mathrm{fl}(\tilde{w}_i(v)) = \tilde{w}_i(v)(1+\delta_i)$ and $|\delta_i| \leq D\epsilon + O(\epsilon^2)$ for $i = 1,\dots,n$, where*

$$D = \max_{i=1,\dots,n} \frac{\pi}{2}\left(u(\alpha_{i-1,i+1}) + \sum_{j\neq i-1,i}(\max\{u(\beta_j), u(\gamma_j)\}+1)\right) + (n-1)(2+D_{\mathrm{sqrt}}+D_{\sin}) + 2n - 3.$$

*Proof.* We note that the $\tilde{w}_i$ in (5.11) can be written as in (5.21) for $J = 2n-2$, $K = 1$ and

$$x_{j,1} = \begin{cases} \sin\frac{\alpha_{i-1,i+1}}{2}, & j=1 \\ r_{j-1}, & j=2,\dots,i, \\ r_j, & j=i+1,\dots,n, \\ \sin\frac{\beta_{j-n}+\gamma_{j-n}}{2}, & j=n+1,\dots,n+i-2, \\ \sin\frac{\beta_{j-n+2}+\gamma_{j-n+2}}{2}, & j=n+i-1,\dots,2n-2. \end{cases}$$

It then follows from (2.15), (5.17), and (5.20) that $\mathrm{fl}(x_{j,1}) = x_{j,1}(1+\chi_{j,1})$ with $|\chi_{j,1}| \leq X_{j,1}\epsilon +$

$O(\epsilon^2)$ and

$$X_{j,1} = \begin{cases} \left|\cot\frac{\alpha_{i-1,i+1}}{2}\right|\left|\frac{\alpha_{i-1,i+1}}{2}\right| u(\alpha_{i-1,i+1}) + D_{\sin}, & j = 1, \\ 2 + D_{\text{sqrt}}, & j = 2, \dots, n, \\ \left|\cot\frac{\beta_{j-n}+\gamma_{j-n}}{2}\right|\left|\frac{\beta_{j-n}+\gamma_{j-n}}{2}\right| (\max\{u(\beta_{j-n}), u(\gamma_{j-n})\} + 1) + D_{\sin}, & j = n+1, \dots, n+i-2 \\ \left|\cot\frac{\beta_{j-n+2}+\gamma_{j-n+2}}{2}\right|\left|\frac{\beta_{j-n+2}+\gamma_{j-n+2}}{2}\right| (\max\{u(\beta_{j-n+2}), u(\gamma_{j-n+2})\} + 1) + D_{\sin}, & j = n+i-1, \dots, 2n-2. \end{cases}$$

Therefore, we can use Theorem 5.3 to get $\text{fl}(\tilde{w}_i(v)) = \tilde{w}_i(v)(1+\delta_i)$ with $|\delta_i| \leq D_i\epsilon + O(\epsilon^2)$ and

$$D_i = X_{1,1} + (n-1)(2 + D_{\text{sqrt}}) + \sum_{j=n+1}^{2n-2} X_{j,1} + 2n - 3.$$

Since $|x|/|\sin x| \leq \pi/2$ for any $x \in [-\pi/2, \pi/2]$ and $\alpha_{i-1,i+1}, \beta_j + \gamma_j \in [-\pi/2, \pi/2]$, we get

$$D_i \leq \frac{\pi}{2}\left( u(\alpha_{i-1,i+1}) + \sum_{j \neq i-1,i} (\max\{u(\beta_j), u(\gamma_j)\} + 1) \right) + (n-1)D_{\sin} + (n-1)(2 + D_{\text{sqrt}}) + 2n - 3,$$

which proves the statement. □

In summary, we proved that the constants $D$ in the upper bounds of the relative forward errors for the different weights in (5.1)–(5.7) are mainly influenced by the quantity

$$F_i = \max_{v \in \mathbb{F}^2} \begin{cases} \left|\sin\frac{\alpha_{i-1}+\alpha_i}{2}\cos\frac{\alpha_{i-1}}{2}\cos\frac{\alpha_i}{2}\right|^{-1}, & \text{for } w_i \text{ in (5.1)}, \\ \left|\sin\frac{\alpha_{i-1}+\alpha_i}{2}\sin\frac{\alpha_{i-1}}{2}\sin\frac{\alpha_i}{2}\right|^{-1}, & \text{for } w_i \text{ in (5.2) and (5.4)}, \\ \left|\sin\frac{\alpha_{i-1}+\alpha_i}{2}\sin\frac{\alpha_{i-1}}{2}\sin\frac{\alpha_i}{2}\sin\alpha_{i-1}\sin\alpha_i\right|^{-1}, & \text{for } w_i \text{ in (5.3)}, \\ \left|\sin\frac{\alpha_{i-1}+\alpha_i}{2}\cos\frac{\alpha_{i-1}}{2}\cos\frac{\alpha_i}{2}(1+\cos\alpha_{i-1})(1+\cos\alpha_i)\right|^{-1}, & \text{for } w_i \text{ in (5.5) and (5.6)}, \\ |1-\cos(\alpha_{i-1}+\alpha_i)|^{-1} + \sum_{j\neq i-1,i} |1+\cos\alpha_j|^{-1}, & \text{for } \hat{w}_i \text{ in (5.7)}, \end{cases}$$

while the $\tilde{w}_i$ in (5.11) are the only ones to be independent of it. Moreover, in all cases, $F_i$ is the maximum, over the finite set $\mathbb{F}^2$, of some function that diverges to infinity, either at the edges of $P$, along the lines that support them, or at the sets $Z_i$, which explains the big relative errors of the mean value coordinates $\phi_i$ close to those regions. Surprisingly, for the original formula in (5.1), the potentially big relative errors of the weights usually cancel out "magically" during the normalization and do not affect the relative errors of the $\phi_i$, except in some cases that we will discuss in Section 5.7.

Regarding the absolute forward error of $\phi_i$, $i = 1, \dots, n$, it follows from (5.12) and Corollary 5.2 that it is bounded from above by $(1+D)|\phi_i|\epsilon + (n-1+D)W(v)|\phi_i|\epsilon + O(\epsilon^2)$. Therefore, in this case, the quantity that distinguishes the performance of the various methods in terms of absolute error is $D|\phi_i|$.

## 5.7 Numerical experiments

We investigated various examples to compare the different approaches for computing mean value coordinates based on the formulas in (5.1)–(5.7) and (5.11). Overall, we found that

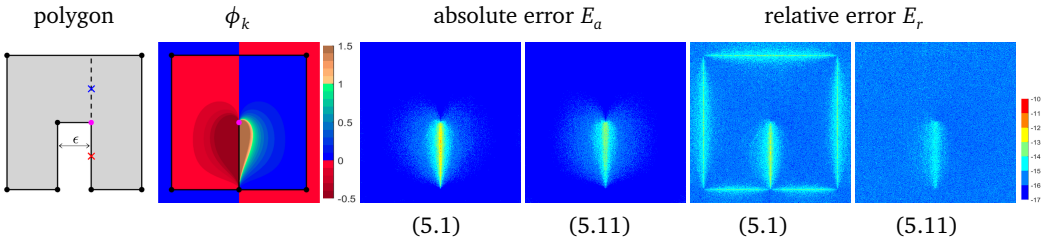


Figure 5.3. Plots of the absolute and relative errors on a $\log_{10}$ scale made by the original (5.1) and the new formula (5.11) to evaluate the mean value coordinate $\phi_k$ related to the vertex $v_k$ (magenta dot) for the polygon on the left with $\epsilon = 0.0001$.

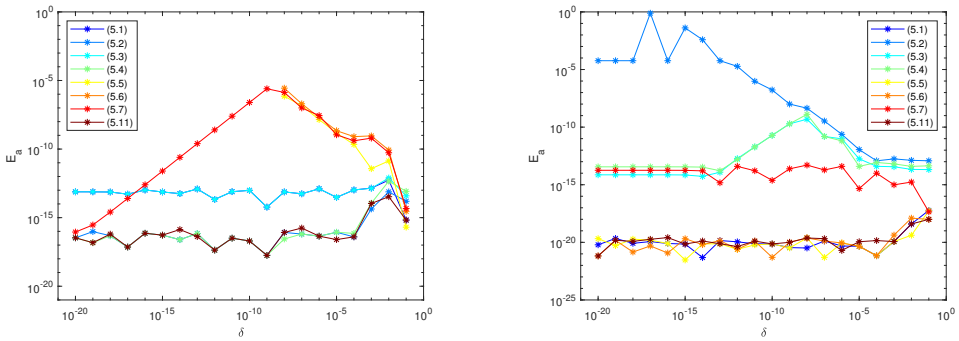


Figure 5.4. Comparison of the absolute errors on a log-log scale for computing $\phi_k$ with the formulas (5.1)–(5.7) and (5.11) close to the points marked by the red cross (left) and the blue cross (right) in Figure 5.3. The plots show $E_a(v)$ for the different algorithms for $v$ at a horizontal distance of $\delta = 10^{-20}, 10^{-19}, \ldots, 10^{-1}$ from the considered points. Some values are not shown for very small $\delta$, because the algorithms return NaN as a result.

our new formula (5.11) consistently provides the most stable results, followed by the original formula (5.1), which usually performs much better than the other formulas and is often almost as stable as (5.11). However, as shown in Section 5.7.1, there are specific cases where our new Algorithm 10 beats the implementation of the original formula by a considerable margin. In Section 5.7.2, we further provide a comprehensive study of the efficiency of all methods. We implemented all algorithms with double precision in C++ and computed the "exact" values of $\phi_k$ in multiple-precision (1024 bit) floating-point arithmetic using the MPFR library [33] for determining the relative and the absolute errors. All tests were run on a Windows 10 laptop with 1.8 GHz Intel Core i7-10510U processor and 16 GB RAM.

### 5.7.1 Stability comparison

Let us begin by comparing the performance of the original and the new formula for the 8-vertex polygon with vertices $(1,1)$, $(-1,1)$, $(-1,-1)$, $(-\epsilon,-1)$, $(-\epsilon,0)$, $(\epsilon,0)$, $(\epsilon,-1)$, and $(1,-1)$, shown in Figure 5.3 (left), where $\epsilon$ indicates the distance between the two vertical edges in the middle. Specifically, we investigate the case $\epsilon = 0.0001$ and turn our focus on the coordinate $\phi_k$ associated with the vertex marked by the magenta dot. In the plots of the

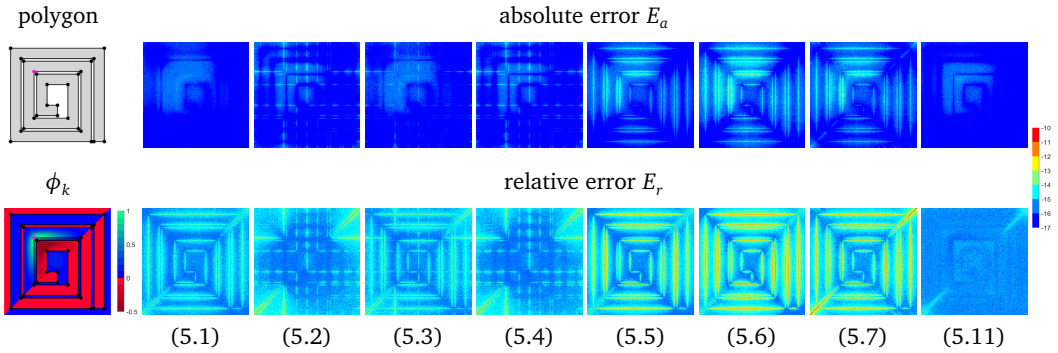


Figure 5.5. Same as Figure 5.2, but for a square spiral polygon.

absolute error $E_a$ and the relative error $E_r$, which were computed on a uniform $500 \times 500$ grid that contains the polygon, we observe that problematic regions with numerical instability exist near the edges $[v_{k-1}, v_k]$ and $[v_k, v_{k+1}]$, but that Algorithm 10 handles them better. One reason for the relatively big errors is the function $W$ in (5.16), which influences the upper bound on the relative error in (5.15) for both formulas and obtains values on the order of $10^3$ in this region. The other reason is the constant $D$, which is about two orders of magnitude bigger for the formula in (5.1) than for our new formula in (5.11).

We also analysed the performance of the other formulas for this example, and Figure 5.4 (left) shows that the implementations of (5.2) and (5.4) are as stable as Algorithm 10 close to the edge $[v_k, v_{k+1}]$. However, both formulas, together with (5.3), are very unstable close to the extension of this line, where instead the implementations of (5.1), (5.5) and (5.6) are stable (see Figure 5.4, right). Interestingly, the worst case for our new formula, in terms of stability, does not occur extremely close to the edge $[v_k, v_{k+1}]$, but at a distance of about $10^{-2}$ to $10^{-3}$, which is again due to the behaviour of the function $W$ in (5.16), and similar for the formulas in (5.2) and (5.4). In contrast, the worst case for the formula in (5.7) happens at a distance of $10^{-8}$ to $10^{-9}$, that is, at roughly $\sqrt{\epsilon}$.

Figure 5.5 compares the errors of the different evaluation procedures for a square spiral polygon. As before, the plots show the absolute errors $E_a$ and the relative errors $E_r$ sampled on a uniform grid of $500 \times 500$ points that contains the polygon. Note that the black pixels in the lower left and the upper right of the relative error plots indicate points $v$ for which $E_r(v)$ is not well-defined, because $\phi_k(v) = 0$. Otherwise, these plots confirm the behaviour that we already observed in Figure 5.2: the new formula (5.11) achieves the best result and the original one (5.1) is second-best, except close to the boundary of the polygon in relative terms. However, since $\phi_k$ is very small in these regions, it makes more sense to focus on the absolute errors. These indicate that (5.1) and (5.11) produce very similar results, but still the new Algorithm 10 is better near the edges $[v_{k-1}, v_k]$ and $[v_k, v_{k+1}]$. As in Figure 5.2, we further note that (5.2), (5.3) and (5.4) exhibit numerical instability along the extensions of the polygon's edges, especially for those related to $[v_{k-1}, v_k]$ and $[v_k, v_{k+1}]$, while (5.5) and (5.6) behaves similarly to (5.1), but with bigger errors close to the boundary. Finally, (5.7) is unstable in the vicinity of all sets $Z_i$. Figure 5.6 shows very similar results for a star-shaped polygon.

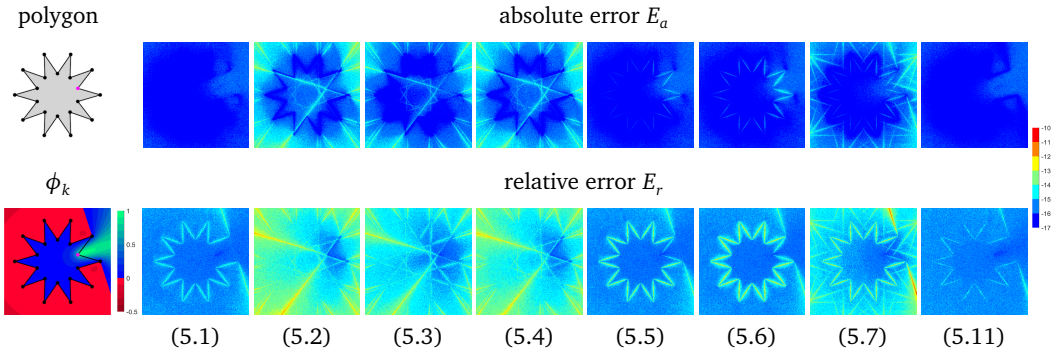


Figure 5.6. Same as Figure 5.2, but for a star-shaped polygon.

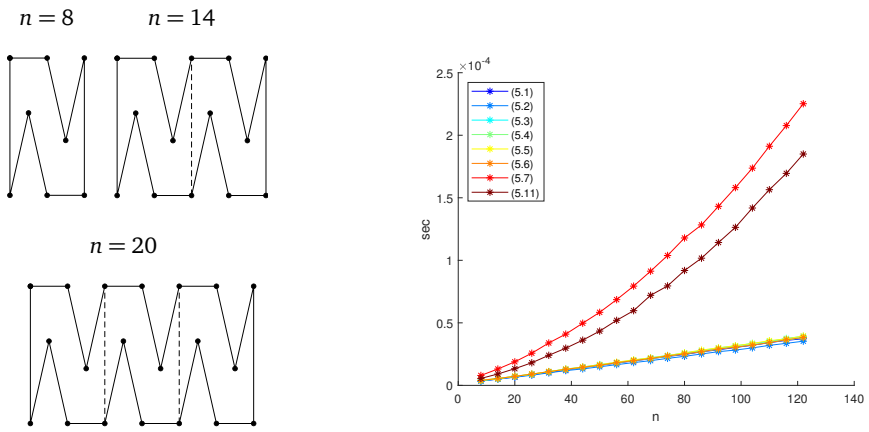


Figure 5.7. Average time in seconds (right) needed by the implementations of the formulas in (5.1)–(5.7) and (5.11) to evaluate all $n$ mean value coordinates for a concave test polygon (left) with $n = 6i + 2$ vertices for $i = 1, \ldots, 20$.

### 5.7.2 Efficiency comparison

To compare the efficiency of the different implementations, we conducted a first experiment using a set of 20 concave polygons, with an increasing number of vertices $n$, specifically with $n = 6i + 2$ for $i = 1, \ldots, 20$. The pattern of the polygons is shown in Figure 5.7 (left) for $i = 1, 2, 3$. The timings are obtained by evaluating the coordinates $\phi_1, \ldots, \phi_n$ at 90000 points and taking the average. The plots in Figure 5.7 (right) clearly indicate the linear time complexity of the algorithms derived from the formulas in (5.1)–(5.6) and the quadratic time complexity of the one that implements formula (5.7) as well as the new Algorithm 10, with the latter being roughly 25% faster. However, despite the unfavourable time complexity, the stable Algorithm 10 is at most twice as expensive as the linear-time algorithms for $n \leq 30$ and only about four times slower for $n = 100$.

In a second experiment, we focus on comparing the efficiency of the different methods for significantly larger values of $n$. Specifically, we construct the test polygons in Figure 5.8 (left) by sampling an epitrochoid curve at $n = 2^i$ points, $i = 3, \ldots, 13$, and then measure and plot (right)

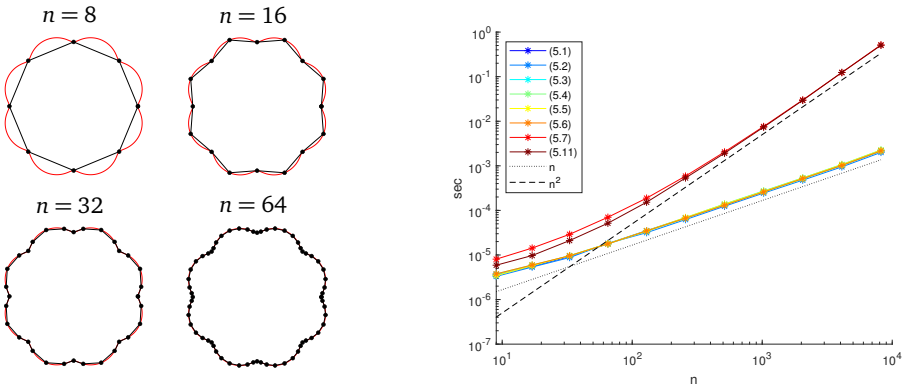


Figure 5.8. Average time in seconds (right) on a log-log scale for the implementations of the formulas in (5.1)–(5.7) and (5.11) to evaluate all $n$ mean value coordinates for a test polygon (left) inscribed to an epitrochoid (red curve) with $n = 2^i$ vertices for $i = 3, \ldots, 13$.

the average running time of all algorithms for computing all $n$ mean value coordinates at 100 evaluation points. In this setting, we gain a more comprehensive understanding of the asymptotic computational cost associated with the implementation of the various formulas, further confirming our previous observations. In fact, formulas (5.1)–(5.6) demonstrate a computational complexity of $O(n)$, while (5.7) and Algorithm 10 exhibit an asymptotic running time on the order of $O(n^2)$. To conclude, although (5.7) and Algorithm 10 have similar behaviour, our new implementation consistently proves to be faster in practice, especially for polygons with less than a thousand vertices.

# Chapter 6

# A comprehensive comparison on the numerical stability of algorithms for evaluating rational Bézier curves

## 6.1 Existing methods for computing rational Bézier curves

Bézier curves were originally introduced in the context of car modeling for major automotive manufacturers [9, 10, 19]. Nowadays, their utility extends across numerous fields, like computer-aided design, simulation, approximation, robotics, artificial intelligence, etc. Many applications in these domains require real-time interactions or live updates, thus necessitating fast evaluation times. For this reason, over the years, there have been numerous studies dedicated to developing efficient evaluation algorithms for Bézier curves. In this chapter, we aim to present a comparison on the numerical stability of the most commonly used algorithms.

Given a set of $n+1$ control points $P_0, \dots, P_n \in \mathbb{R}^2$ with associated positive real weights $w_0, \dots, w_n$, we define a *rational Bézier curve* $P : [0,1] \to \mathbb{R}^2$ as

$$P(t) = \frac{\sum_{i=0}^{n} B_i^n(t) w_i P_i}{\sum_{i=0}^{n} B_i^n(t) w_i}, \tag{6.1}$$

where

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}, \qquad i = 0, \dots, n, \tag{6.2}$$

represents the Bernstein basis composed by polynomials of degree $n$ and $t \in [0,1]$ is the parameter along the curve. There exist numerous methods for computing rational Bézier curves, such as adaptations of the classic de Casteljau algorithm for polynomials in the rational case, or more efficient approaches employing Horner-like schemes or basis conversions. We now present the most commonly used algorithms and, for each of them, we describe how it is implemented.

### 6.1.1 Rational de Casteljau algorithms

The most straightforward approach to compute $P(t)$ is by using the classic quadratic time de Casteljau algorithm for polynomials [12]. In the case of a rational Bézier curve of the type

in (6.1), we recall that it can be considered as the central projection of the spatial polynomial curve

$$\hat{P}(t) = \sum_{i=0}^{n} B_i^n(t)\hat{P}_i, \qquad \hat{P}_i = \begin{pmatrix} w_i P_i \\ w_i \end{pmatrix}, \tag{6.3}$$

under the projection

$$\mathrm{proj}(x, y, z) = \left(\frac{x}{z}, \frac{y}{z}\right). \tag{6.4}$$

This implies that we can apply the classical de Casteljau algorithm to $\hat{P}(t)$ and then project the final result according to (6.4). This process is equivalent to first computing the values of the numerator and the denominator with the recursive formulas

$$\begin{cases} N_i^0 = w_i P_i, \\ N_i^r = N_i^{r-1}(1-t) + N_{i+1}^{r-1} t, \end{cases} \quad \text{and} \quad \begin{cases} D_i^0 = w_i, \\ D_i^r = D_i^{r-1}(1-t) + D_{i+1}^{r-1} t, \end{cases} \tag{6.5}$$

$i = 0, \ldots, n$ and $r = 1, \ldots, n$, respectively, and then the final result as $P(t) = N_0^n / D_0^n$. We also note that this method exhibits quadratic complexity.

Alternatively, Farin [26] adapts this approach into a more robust quadratic time algorithm with additional geometric meaning, given by

$$\begin{cases} w_i^0 = w_i, \\ P_i^0 = P_i, \\ w_i^r = w_i^{r-1}(1-t) + w_{i+1}^{r-1} t, \\ P_i^r = \frac{P_i^{r-1} w_i^{r-1}}{w_i^r}(1-t) + \frac{P_{i+1}^{r-1} w_{i+1}^{r-1}}{w_i^r} t, \end{cases} \tag{6.6}$$

$i = 0, \ldots, n$ and $r = 1, \ldots, n$, and $P(t) = P_0^n$.

### 6.1.2 Horner-like algorithms

Volk and Schumaker [75] are the first to achieve an algorithm for computing polynomial Bézier curves with linear time complexity. Their idea is to use nested multiplications for the computation, which results in a significant gain in terms of efficiency. We present a straightforward extension of the VS algorithm by first applying it on the numerator and the denominator of $P(t)$, and then simplifying some common factors. In particular, we express the rational Bézier curve in (6.1) equivalently as

$$P(t) = \frac{\sum_{i=0}^{n} x^{n-i} \binom{n}{i} w_i P_i}{\sum_{i=0}^{n} x^{n-i} \binom{n}{i} w_i}, \qquad x = \begin{cases} (1-t)/t, & t > 1/2 \\ t/(1-t), & t \le 1/2. \end{cases} \tag{6.7}$$

There are many methods for evaluating a polynomial; Goldman [39] highlights two approaches: one using a Horner scheme, and the other employing a ladder pattern. However, Warren [81] shows that these forms are equivalent for the monomial basis, so we consider the former. Therefore, the VS algorithm evaluates the numerator and the denominator using a Horner scheme as

$$P(t) = \begin{cases} \dfrac{\binom{n}{n} w_n P_n + x\left(\binom{n}{n-1} w_{n-1} P_{n-1} + x\left(\binom{n}{n-2} w_{n-2} P_{n-2} + \cdots + x\left(\binom{n}{1} w_1 P_1 + x\binom{n}{0} w_0 P_0\right)\ldots\right)\right)}{\binom{n}{n} w_n + x\left(\binom{n}{n-1} w_{n-1} + x\left(\binom{n}{n-2} w_{n-2} + \cdots + x\left(\binom{n}{1} w_1 + x\binom{n}{0} w_0\right)\ldots\right)\right)}, & t > 1/2, \\ \dfrac{\binom{n}{0} w_0 P_0 + x\left(\binom{n}{1} w_1 P_1 + x\left(\binom{n}{2} w_2 P_2 + \cdots + x\left(\binom{n}{n-1} w_{n-1} P_{n-1} + x\binom{n}{n} w_n P_n\right)\ldots\right)\right)}{\binom{n}{0} w_n + x\left(\binom{n}{1} w_1 + x\left(\binom{n}{2} w_2 + \cdots + x\left(\binom{n}{n-1} w_{n-1} + x\binom{n}{n} w_n\right)\ldots\right)\right)}, & t \le 1/2. \end{cases}$$

With the same strategy, Farin [27] presents another Horner-like algorithm by setting $s = 1-t$ and computing $P(t)$ in (6.1) as

$$P(t) = \frac{\sum_{i=0}^{n} t^i s^{n-i}\binom{n}{i} w_i P_i}{\sum_{i=0}^{n} t^i s^{n-i}\binom{n}{i} w_i} = \frac{\left(\ldots\left(\left(\binom{n}{0} w_0 P_0 s + \binom{n}{1} w_1 P_1 t\right) s + \binom{n}{2} w_2 P_2 t^2\right) s + \cdots + \binom{n}{n-1} w_{n-1} P_{n-1} t^{n-1}\right) s + \binom{n}{n} w_n P_n t^n}{\left(\ldots\left(\left(\binom{n}{0} w_0 s + \binom{n}{1} w_1 t\right) s + \binom{n}{2} w_2 t^2\right) s + \cdots + \binom{n}{n-1} w_{n-1} t^{n-1}\right) s + \binom{n}{n} w_n t^n}. \quad (6.8)$$

### 6.1.3 Geometric approach

On the one hand, while the rational de Casteljau adaptation by Farin [26] has some nice geometric interpretation, it can only be done in quadratic time. On the other hand, the VS algorithm has linear time complexity, but it lacks geometric interpretation and properties. For this reason, Woźny and Chudy [82] introduce a new linear time algorithm that has a nice geometric interpretation. In particular, $P(t)$ can be computed recursively using a Horner-like scheme and convex combinations as

$$\begin{cases} h_0 = 1, & h_i = \dfrac{w_i h_{i-1} t(n-i+1)}{w_{i-1} i(1-t) + w_i h_{i-1} t(n-i+1)}, \\ T_0 = P_0, & T_i = (1-h_i) T_{i-1} + h_i P_i. \end{cases} \quad (6.9)$$

From these recursive formulas, this algorithm has an elegant geometric interpretation since $T_i \in [T_{i-1}, P_i]$.

### 6.1.4 Wang–Ball algorithm

Another approach to achieve an algorithm with linear time complexity is by converting the Bernstein basis into a different basis. There exist several methods in this direction, such as transforming the Bernstein into the Wang–Ball basis [22, 68, 80], the DP basis [20, 23, 24], and other similar types of bases [21]; the former is proven to be the most efficient. The rational Wang–Ball curve, defined by the control points $R_0, \ldots, R_n$ with their respective weights $v_0, \ldots, v_n$, is given by

$$P(t) = \frac{\sum_{i=0}^{n} A_i^n(t) v_i R_i}{\sum_{i=0}^{n} A_i^n(t) v_i}, \quad (6.10)$$

where the Wang–Ball basis $\{A_i^n\}_{i=0,\ldots,n}$ is defined as

$$A_i^n(t) = \begin{cases} (2t)^i (1-t)^{i+2}, & 0 \le i \le \lfloor n/2 \rfloor - 1, \\ (2t)^{\lfloor n/2 \rfloor} (1-t)^{\lceil n/2 \rceil}, & i = \lfloor n/2 \rfloor, \\ (2(1-t))^{\lfloor n/2 \rfloor} t^{\lceil n/2 \rceil}, & i = \lceil n/2 \rceil, \\ A_{n-i}^n(1-t), & \lceil n/2 \rceil + 1 \le i \le n. \end{cases} \quad (6.11)$$

Actually, in order to achieve a linear time method, its implementation uses a recursive algorithm similar to (6.6), but for the new set of control points and weights. Specifically, it starts by setting

$$n_0 = n, \qquad v_i^0 = v_i, \qquad \text{and} \qquad R_i^0 = R_i, \qquad i = 0, \ldots, n^0, \quad (6.12)$$

and then, at each step $r = 1, \ldots, n$ of the recursion, it defines $n_r = n - r$ new weights and control points. In particular, if $n_r$ is odd, they are given by

$$\begin{cases} v_i^r = v_i^{r-1}, & i = 0, \ldots, \frac{n_r-3}{2}, \\ v_i^r = v_i^{r-1}(1-t) + v_{i+1}^{r-1} t, & i = \frac{n_r-1}{2}, \\ v_i^r = v_i^{r-1}, & i = \frac{n_r+1}{2}, \ldots, n_r, \end{cases} \quad \text{and} \quad \begin{cases} R_i^r = R_i^{r-1}, & i = 0, \ldots, \frac{n_r-3}{2}, \\ R_i^r = \frac{R_i^{r-1} v_i^{r-1}}{v_i^r}(1-t) + \frac{R_{i+1}^{r-1} v_{i+1}^{r-1}}{v_i^r} t, & i = \frac{n_r-1}{2}, \\ R_i^r = R_i^{r-1}, & i = \frac{n_r+1}{2}, \ldots, n_r, \end{cases} \quad (6.13)$$

while, if $n_r$ is even, they are

$$\begin{cases} v_i^r = v_i^{r-1}, & i = 0,\dots,\frac{n_r}{2}-2, \\ v_i^r = v_i^{r-1}(1-t)+v_{i+1}^{r-1}t, & i = \frac{n_r}{2}-1,\frac{n_r}{2}, \\ v_i^r = v_i^{r-1}, & i = \frac{n_r}{2}+1,\dots,n_r, \end{cases} \quad \text{and} \quad \begin{cases} R_i^r = R_i^{r-1}, & i = 0,\dots,\frac{n_r}{2}-2, \\ R_i^r = \frac{R_i^{r-1}v_i^{r-1}}{v_i^r}(1-t)+\frac{R_{i+1}^{r-1}v_{i+1}^{r-1}}{v_i^r}t, & i = \frac{n_r}{2}-1,\frac{n_r}{2}, \\ R_i^r = R_i^{r-1}, & i = \frac{n_r}{2}+1,\dots,n_r, \end{cases} \tag{6.14}$$

and the result is $P(t) = R_0^n$. Before proceeding with this algorithm, there is a prepossessing step to get the values $v_0,\dots,v_n$ and $R_0,\dots,R_n$. In particular, the weights and control points of the Bézier and Wang–Ball representations can be converted back-and-forth by means of a matrix multiplication [51]. However, for the sake of numerical stability, Dejdumrong et al. [22] present the explicit formulas to obtain the Wang–Ball control points and weights from the corresponding Bézier ones, that are

$$\begin{cases} v_0 = w_0, \\ v_n = w_n, \\ v_i = \frac{1}{2^i}\left[\binom{n}{i}w_i - \sum_{k=0}^{i-1} 2^k\binom{n-2-2k}{i-k}v_k - \sum_{k=n-i+1}^{n} 2^{n-k}\binom{2k-2-n}{k-i}v_k\right], & i < \lfloor n/2\rfloor, \\ v_i = \frac{1}{2^{n-i}}\left[\binom{n}{i}w_i - \sum_{k=0}^{n-i} 2^k\binom{n-2-2k}{i-k}v_k - \sum_{k=i+1}^{n} 2^{n-k}\binom{2k-2-n}{k-i}v_k\right], & i > \lceil n/2\rceil, \\ v_i = \frac{1}{2^i}\left[\binom{n}{i}w_i - \sum_{k=0}^{i-1} 2^k\binom{n-2-2k}{i-k}v_k - \sum_{k=i+2}^{n} 2^{n-k}\binom{2k-2-n}{k-i}v_k\right], & i = \lfloor n/2\rfloor, \\ v_i = \frac{1}{2^{n-i}}\left[\binom{n}{i}w_i - \sum_{k=0}^{i-2} 2^k\binom{n-2-2k}{i-k}v_k - \sum_{k=i+1}^{n} 2^{n-k}\binom{2k-2-n}{k-i}v_k\right], & i = \lceil n/2\rceil \end{cases} \tag{6.15}$$

and

$$\begin{cases} R_0 = P_0, \\ R_n = P_n, \\ R_i = \frac{1}{2^i v_i}\left[\binom{n}{i}w_iP_i - \sum_{k=0}^{i-1} 2^k\binom{n-2-2k}{i-k}v_kR_k - \sum_{k=n-i+1}^{n} 2^{n-k}\binom{2k-2-n}{k-i}v_kR_k\right], & i < \lfloor n/2\rfloor, \\ R_i = \frac{1}{2^{n-i}v_i}\left[\binom{n}{i}w_iP_i - \sum_{k=0}^{n-i} 2^k\binom{n-2-2k}{i-k}v_kR_k - \sum_{k=i+1}^{n} 2^{n-k}\binom{2k-2-n}{k-i}v_kR_k\right], & i > \lceil n/2\rceil, \\ R_i = \frac{1}{2^i v_i}\left[\binom{n}{i}w_iP_i - \sum_{k=0}^{i-1} 2^k\binom{n-2-2k}{i-k}v_kR_k - \sum_{k=i+2}^{n} 2^{n-k}\binom{2k-2-n}{k-i}v_kR_k\right], & i = \lfloor n/2\rfloor, \\ R_i = \frac{1}{2^{n-i}v_i}\left[\binom{n}{i}w_iP_i - \sum_{k=0}^{i-2} 2^k\binom{n-2-2k}{i-k}v_kR_k - \sum_{k=i+1}^{n} 2^{n-k}\binom{2k-2-n}{k-i}v_kR_k\right], & i = \lceil n/2\rceil. \end{cases} \tag{6.16}$$

We note that, before computing $v_k$ and $R_k$, $k = 0,\dots,n$, the weights $v_i$ and $v_{n-i}$ and the control points $R_i$ and $R_{n-i}$, $i = 0,\dots,k-1$, must be computed.

### 6.1.5 Barycentric algorithm

Finally, Ramanantoanina and Hormann [70] propose another alternative to convert the rational Bézier representation to a barycentric rational interpolating form. In particular, given a set of interpolation points $Q_0,\dots,Q_n$ with their respective weights $u_0,\dots,u_n$ and nodes $t_0,\dots,t_n$, a barycentric rational interpolant is defined as

$$P(t) = \frac{\sum_{i=0}^{n}\frac{u_i}{t-t_i}Q_i}{\sum_{i=0}^{n}\frac{u_i}{t-t_i}}. \tag{6.17}$$

The barycentric interpolation points and weights are related with the corresponding Bézier ones as

$$Q_i = P(t_i) \qquad \text{and} \qquad u_i = z(t_i)\prod_{k\neq i}\frac{1}{t_i-t_k}, \qquad i = 0,\dots,n,$$

where $z(t)$ is the denominator of $P(t)$ in (6.1). A common choice for the set of nodes is given by the Chebyshev nodes of the second kind in $[0,1]$, which are defined as $t_{n-i} = 1/2\cos(i\pi/n) + 1/2$, $i = 0,\dots,n$. In this case, the weights turn out to be computed in linear time as [73]

$$u_i = (-1)^i \delta_i z(t_i), \qquad \delta_i = \begin{cases} 1/2, & i = 0 \text{ or } i = n, \\ 1, & i = 1,\dots,n-1. \end{cases}$$

Alternatively, we can also use uniformly distributed nodes $t_i = i/n$, $i = 0,\dots,n$, with weights of the form

$$u_i = (-1)^i \binom{n}{i} z(t_i).$$

For the sake of efficiency, we propose to compute the values $Q_i = P(t_i)$ by evaluating the rational Bézier curve $P$ at $t_i$ through an adapted version of the rational VS algorithm. Doing so, we can also obtain the values $z(t_i)$ within the same algorithm as

$$z(t_i) = \sum_{i=0}^{n} x^{n-i} \binom{n}{i} w_i \times \begin{cases} t^n, & t > 1/2, \\ (1-t)^n, & t \le 1/2, \end{cases} \tag{6.18}$$

for $x$ in (6.7).

## 6.2 Numerical stability

Let us now focus on analysing the numerical stability of the different algorithms that evaluate a rational Bézier curve. In particular, we study the relative error $E \in \mathbb{R}^2$ defined as

$$E(t) = \frac{|\mathrm{fl}(P(t)) - P(t)|}{|P(t)|} = \left( \frac{|\mathrm{fl}(P_x(t)) - P_x(t)|}{|P_x(t)|}, \frac{|\mathrm{fl}(P_y(t)) - P_y(t)|}{|P_y(t)|} \right) \tag{6.19}$$

for each algorithm, where $P(t)$ is the exact result and $\mathrm{fl}(P(t))$ that of its finite-precision implementation. Finally, we always assume that the input data $t$, $w_i$ and $P_i$ are floating-point numbers, so they do not introduce any numerical error during the computation.

### 6.2.1 Convex combinations

We start by examining the numerical stability of algorithms that evaluate a rational Bézier curve $P$ at $t$ through a recursive method defined by convex combinations. Specifically, we focus on the rational de Casteljau algorithm and the Wang–Ball algorithm. Regarding the former defined in (6.6), our analysis begins with a study of the error propagation in the weights $w_i^r$, followed by an investigation into the relative error of the values $P_i^r$. These results lead to an upper bound on the relative error $E$ in (6.19) in the case of $P(t) = P_0^n$.

**Lemma 6.1.** *For any $t, w_0, \dots, w_n \in \mathbb{F}$ and $r \in \{1,\dots,n\}$, there exist $\omega_0^r, \dots, \omega_n^r \in \mathbb{R}$ such that the weights $w_i^r$ in (6.6) satisfy $\mathrm{fl}(w_i^r) = w_i^r(1+\omega_i^r)$, $i = 0,\dots,n$, with $|\omega_i^r| \le U(w_i^r)\epsilon + O(\epsilon^2)$ and*

$$U(w_i^r) = 3r.$$

*Proof.* First, we notice that

$$\begin{aligned} \mathrm{fl}(w_i^r) &= w_i^{r-1}(1+\omega_i^{r-1})(1-t)(1+\delta_1) + w_{i+1}^{r-1}(1+\omega_{i+1}^{r-1})t(1+\delta_2) \\ &= w_i^{r-1}(1-t)(1+\omega_i^{r-1}+\delta_1+O(\epsilon^2)) + w_{i+1}^{r-1}t(1+\omega_{i+1}^{r-1}+\delta_2+O(\epsilon^2)), \end{aligned}$$

where $\delta_1$ and $\delta_2$ are the errors introduced by the operations in the first and second addends, respectively, that are one product and one sum in both cases, plus one subtraction for the first addend only. Therefore, it follows from (2.5) that $|\delta_1|, |\delta_2| \leq 3\epsilon + O(\epsilon^2)$. Moreover, the intermediate value theorem further guarantees that

$$\mathrm{fl}(w_i^r) = (w_i^{r-1}(1-t) + w_{i+1}^{r-1}t)(1+\omega_i^r),$$

for some $\omega_i^r \in [\min(\omega_i^{r-1} + \delta_1 + O(\epsilon^2), \omega_{i+1}^{r-1} + \delta_2 + O(\epsilon^2)), \max(\omega_i^{r-1} + \delta_1 + O(\epsilon^2), \omega_{i+1}^{r-1} + \delta_2 + O(\epsilon^2))]$. Now, we can prove the statement by induction over $r$. The base case follows by the fact that $w_i^0 = w_i$, therefore $\omega_i^0 = 0$ for all $i = 0, \ldots, n$. Finally, the inductive step from $r-1$ to $r$ follows from the fact that $|\omega_i^r| \leq \max_{j=i,i+1} |\omega_j^{r-1}| + 3\epsilon + O(\epsilon^2)$, together with the inductive hypothesis, that is $|\omega_i^{r-1}| \leq 3(r-1)\epsilon + O(\epsilon^2)$, $i = 0, \ldots, n$. □

**Proposition 6.2.** *For any $t, w_0, \ldots, w_n, P_0, \ldots, P_n \in \mathbb{F}$ and $r \in \{1, \ldots, n\}$, the relative errors of the $P_i^r$ in* (6.6) *satisfy*

$$\frac{|\mathrm{fl}(P_i^r(t)) - P_i^r(t)|}{|P_i^r|} \leq \frac{\sum_{k=0}^r B_k^r(t)|P_{i+k}w_{i+k}|}{\left|\sum_{k=0}^r B_k^r(t)P_{i+k}w_{i+k}\right|}(3r^2+5r)\epsilon + O(\epsilon^2),$$

$i = 0, \ldots, n$. *Therefore, the relative error in* (6.19) *for $P(t) = P_0^n$ satisfies*

$$E(t) \leq \frac{\sum_{k=0}^n B_k^n(t)|P_k w_k|}{\left|\sum_{k=0}^n B_k^n(t)P_k w_k\right|}(3n^2+5n)\epsilon + O(\epsilon^2).$$

*Proof.* Denoting by $\varphi_i^r$ the relative errors introduced by the computation of $P_i^r$, $i = 0, \ldots, n$ and $r = 1, \ldots, n$, we first notice that

$$\mathrm{fl}(P_i^r) = \frac{P_i^{r-1}(1+\varphi_i^{r-1})w_i^{r-1}(1+\omega_i^{r-1})(1-t)(1+\delta_1) + P_{i+1}^{r-1}(1+\varphi_{i+1}^{r-1})w_{i+1}^{r-1}(1+\omega_{i+1}^{r-1})t(1+\delta_2)}{w_i^r(1+\omega_i^r)},$$

where $|\omega_j^m| \leq 3m\epsilon + O(\epsilon^2)$, $j = i, i+1$ and $m = r-1, r$, by Lemma 6.1 and $\delta_1$ and $\delta_2$ are the errors introduced by the operations in the first and second addends of the numerator, respectively, that are two products, one sum, and one division each, plus one subtraction for the first addend only. Therefore, it follows from (2.5) that $|\delta_1|, |\delta_2| \leq 5\epsilon + O(\epsilon^2)$. By Taylor expansion, we know that

$$\frac{1}{(1+\omega_i^r)} = 1 - \omega_i^r + O(\epsilon^2),$$

hence

$$\mathrm{fl}(P_i^r) = P_i^r + \tfrac{P_i^{r-1}w_i^{r-1}(1-t)}{w_i^r}(\varphi_i^{r-1} + \omega_i^{r-1} - \omega_i^r + \delta_1 + O(\epsilon^2)) + \tfrac{P_{i+1}^{r-1}w_{i+1}^{r-1}t}{w_i^r}(\varphi_{i+1}^{r-1} + \omega_{i+1}^{r-1} - \omega_i^r + \delta_2 + O(\epsilon^2)).$$

Then, using the fact that $\mathrm{fl}(P_i^r) - P_i^r = P_i^r\varphi_i^r$, the triangle inequality, and the upper bounds on the relative errors introduced by the weights and the operations, we obtain

$$\begin{aligned}
|P_i^r\varphi_i^r w_i^r| &\leq |P_i^{r-1}\varphi_i^{r-1}w_i^{r-1}(1-t) + P_{i+1}^{r-1}\varphi_{i+1}^{r-1}w_{i+1}^{r-1}t| \\
&\quad + |P_i^{r-1}w_i^{r-1}(1-t)(\omega_i^{r-1} - \omega_i^r + \delta_1) + P_{i+1}^{r-1}w_{i+1}^{r-1}t(\omega_{i+1}^{r-1} - \omega_i^r + \delta_2)| + O(\epsilon^2) \\
&\leq |P_i^{r-1}\varphi_i^{r-1}w_i^{r-1}|(1-t) + |P_{i+1}^{r-1}\varphi_{i+1}^{r-1}w_{i+1}^{r-1}|t \\
&\quad + (|P_i^{r-1}w_i^{r-1}|(1-t) + |P_{i+1}^{r-1}w_{i+1}^{r-1}|t)(6r+2)\epsilon + O(\epsilon^2).
\end{aligned} \tag{6.20}$$

In general, we know that[1] $P_j^m w_j^m = \sum_{k=0}^m B_k^m P_{j+k} w_{j+k}$, $j = 0, \ldots, n$ and $m = 1, \ldots, n$, therefore, by also using the relations $B_k^{r-1}(1-t) = (r-k)/r B_k^r$ and $B_k^{r-1} t = (k+1)/r B_{k+1}^r$, $k = 0, \ldots, r-1$, we obtain

$$\begin{aligned}
|P_i^{r-1} w_i^{r-1}|(1-t) + |P_{i+1}^{r-1} w_{i+1}^{r-1}|t &= \sum_{k=0}^{r-1} B_k^{r-1}(1-t)|P_{i+k} w_{i+k}| + \sum_{k=0}^{r-1} B_k^{r-1} t |P_{i+1+k} w_{i+1+k}| \\
&= \sum_{k=0}^{r-1} \frac{r-k}{r} B_k^r |P_{i+k} w_{i+k}| + \sum_{k=0}^{r-1} \frac{k+1}{r} B_{k+1}^r |P_{i+1+k} w_{i+1+k}| \\
&= B_0^r |P_i w_i| + \sum_{k=1}^{r-1} \left( \frac{r-k}{r} + \frac{k}{r} \right) B_k^r |P_{i+k} w_{i+k}| + B_r^r |P_{i+r} w_{i+r}| \\
&= \sum_{k=0}^{r} B_k^r |P_{i+k} w_{i+k}|
\end{aligned} \tag{6.21}$$

and, by (6.20),

$$|P_i^r \varphi_i^r w_i^r| \leq |P_i^{r-1} \varphi_i^{r-1} w_i^{r-1}|(1-t) + |P_{i+1}^{r-1} \varphi_{i+1}^{r-1} w_{i+1}^{r-1}|t + \sum_{k=0}^{r} B_k^r |P_{i+k} w_{i+k}|(6r+2)\epsilon + O(\epsilon^2). \tag{6.22}$$

Now, we can prove the statement by induction over $r$. The base case follows by the fact that $P_i^0 = P_i$, $i = 0, \ldots, n$, hence $\varphi_i^0 = 0$. Finally, the inductive step from $r-1$ to $r$ follows from the inductive hypothesis, that is

$$|P_i^{r-1} \varphi_i^{r-1} w_i^{r-1}| \leq \sum_{k=0}^{r-1} B_k^{r-1} |P_{i+k} w_{i+k}| [3(r-1)^2 + 5(r-1)]\epsilon + O(\epsilon^2), \qquad i = 0, \ldots, n,$$

together with (6.22) and the fact that, by (6.21),

$$\sum_{k=0}^{r-1} B_k^{r-1}(1-t)|P_{i+k} w_{i+k}| + \sum_{k=0}^{r-1} B_k^{r-1} t |P_{i+1+k} w_{i+1+k}| = \sum_{k=0}^{r} B_k^r |P_{i+k} w_{i+k}|.$$

□

We now turn our attention to the definition of the Wang–Ball algorithm in (6.12)–(6.14), which is very similar to the rational de Casteljau method in (6.6), except for two differences. Firstly, only the "central" Wang–Ball weights and control points are updated at each step $r = 1, \ldots, n$. Secondly, we cannot assume that the input data $v_i$ and $R_i$ are exact, as they are themselves the result of the conversion formulas in (6.15)–(6.16). On the one hand, although only a few weights change at each iteration, the final error propagation is the same as for the recursive formulas in (6.6), because some of the $v_i^r$ and $R_i^r$ are modified at each step $r$. Consequently, we can use the same proof technique of Lemma 6.1 to analyse the error propagation in the weights $v_i^r$ and of Proposition 6.2 to get the upper bounds on the relative errors of the values $R_i^r$ and $P(t) = R_0^n$. On the other hand, in this scenario, we also have to consider the initial errors in the weights $v_i$ and control points $R_i$, which are introduced in the preprocessing step that converts the Bézier weights and control points into their corresponding Wang–Ball ones. Therefore, we state below the equivalent of Lemma 6.1 and Proposition 6.2 in the case of Wang–Ball algorithm.

[1] In the proof, we omit the dependence on the variable $t$ of the basis functions, that is, $B_i^n$ means $B_i^n(t)$.

**Lemma 6.3.** *Suppose that there exist* $\upsilon_0^0,\dots,\upsilon_n^0 \in \mathbb{R}$ *with*

$$\mathrm{fl}(v_i) = v_i(1+\upsilon_i^0), \qquad |\upsilon_i^0| \le U(v_i)\epsilon + O(\epsilon^2), \qquad i = 0,\dots,n,$$

*for some constants* $U(v_i)$*. Then, for any* $r \in \{1,\dots,n\}$*, there exist* $\upsilon_0^r,\dots,\upsilon_{n_r}^r \in \mathbb{R}$ *such that the weights* $v_i^r$ *in* (6.13)–(6.14) *satisfy* $\mathrm{fl}(v_i^r) = v_i^r(1+\upsilon_i^r)$*,* $i = 0,\dots,n_r$*, with* $|\upsilon_i^r| \le U(v_i^r)\epsilon + O(\epsilon^2)$ *and*

$$U(v_i^r) = 3r + \max_{j=0,\dots,n} U(v_j).$$

**Proposition 6.4.** *Suppose that there exist* $\upsilon_0^0,\dots,\upsilon_n^0 \in \mathbb{R}$ *with*

$$\mathrm{fl}(v_i) = v_i(1+\upsilon_i^0), \qquad |\upsilon_i^0| \le U(v_i)\epsilon + O(\epsilon^2), \qquad i = 0,\dots,n$$

*and* $\rho_0^0,\dots,\rho_n^0 \in \mathbb{R}$ *with*

$$\mathrm{fl}(R_i) = R_i(1+\rho_i^0), \qquad |\rho_i^0| \le U(R_i)\epsilon + O(\epsilon^2), \qquad i = 0,\dots,n,$$

*for some constants* $U(v_i)$ *and* $U(R_i)$*. Then, for any* $r \in \{1,\dots,n\}$*, the relative errors of the* $R_i^r$ *in* (6.13)–(6.14) *satisfy*

$$\frac{|\mathrm{fl}(R_i^r(t)) - R_i^r(t)|}{|R_i^r|} \le \frac{\sum_{k=0}^r A_k^r(t)|R_{i+k}v_{i+k}|}{\left|\sum_{k=0}^r A_k^r(t)R_{i+k}v_{i+k}\right|}\left(3r^2 + 5r + \max_{j=0,\dots,n} U(v_j) + \max_{k=0,\dots,n} U(R_k)\right)\epsilon + O(\epsilon^2),$$

$i = 0,\dots,n$*. Therefore, the relative error in* (6.19) *for* $P(t) = R_0^n$ *satisfies*

$$E(t) \le \frac{\sum_{k=0}^n A_k^n(t)|R_k v_k|}{\left|\sum_{k=0}^n A_k^n(t)R_k v_k\right|}\left(3n^2 + 5n + \max_{j=0,\dots,n} U(v_j) + \max_{k=0,\dots,n} U(R_k)\right)\epsilon + O(\epsilon^2).$$

Finally, to provide a comprehensive understanding of the error propagation within the Wang–Ball algorithm, we also present an analysis of the numerical stability of the conversion formulas in (6.15)–(6.16), which provides an initial estimate of the constants $U(v_i)$ and $U(R_i)$, $i = 0,\dots,n$, of Lemma 6.3 and Proposition 6.4. Before delving into these details, we introduce some notation to shorten the expressions of the $v_i$ and $R_i$. Considering $i \in \{0,\dots,n\}$, we define $e \in \mathbb{N}$ as

$$e = \begin{cases} i, & i \le \lfloor n/2 \rfloor, \\ n-i, & i \ge \lceil n/2 \rceil \end{cases}$$

and the sets of indexes $I_{1,i}$ and $I_{2,i}$ as

$$I_{1,i} = \begin{cases} \{0,1,\dots,i-1\}, & i \le \lfloor n/2 \rfloor, \\ \{0,1,\dots,i-2\}, & i = \lceil n/2 \rceil, \\ \{0,1,\dots,n-i\}, & i > \lceil n/2 \rceil, \end{cases} \qquad I_{2,i} = \begin{cases} \{n-i+1,n-i+2,\dots,n\}, & i < \lfloor n/2 \rfloor, \\ \{i+2,i+3,\dots,n\}, & i = \lfloor n/2 \rfloor, \\ \{i+1,i+2,\dots,n\}, & i \ge \lceil n/2 \rceil. \end{cases}$$

Then, we set

$$b_i = \binom{n}{i}, \qquad a_k = 2^k\binom{n-2-2k}{i-k}, \qquad \text{and} \qquad c_k = 2^{n-k}\binom{2k-2-n}{k-i},$$

thus we can express the weights $v_i$ in (6.15) as

$$v_i = \frac{1}{2^e}\left(b_i w_i - \sum_{k\in I_{1,i}} a_k v_k - \sum_{k\in I_{2,i}} c_k v_k\right), \qquad i = 0,\dots,n, \tag{6.23}$$

and the control points $R_i$ in (6.16) as

$$R_i = \frac{1}{2^e v_i}\left( b_i w_i P_i - \sum_{k\in I_{1,i}} a_k v_k R_k - \sum_{k\in I_{2,i}} c_k v_k R_k \right), \qquad i = 0,\dots,n. \tag{6.24}$$

Moreover, we denote by $M_i$ the maximum between the constants $A_i = \max\{a_k \mid k \in I_{1,i}\}$ and $C_i = \max\{c_k \mid k \in I_{2,i}\}$, $i = 1,\dots,n-1$.

**Lemma 6.5.** *For any $t, w_0,\dots,w_n \in \mathbb{F}$, there exist $\upsilon_0,\dots,\upsilon_n \in \mathbb{R}$ such that the Wang–Ball weights $v_i$ in* (6.23) *satisfy* $\mathrm{fl}(v_i) = v_i(1+\upsilon_i)$, $i = 0,\dots,n$, *with* $|\upsilon_i| \le U(v_i)\epsilon + O(\epsilon^2)$ *and*

$$\begin{aligned} U(v_i) = {} & \frac{\max_{j=1,n-1,\dots,n-i,i}\left(b_j w_j + \sum_{k\in I_{1,j}} a_k v_k + \sum_{k\in I_{2,j}} c_k v_k\right)}{\left|b_i w_i - \sum_{k\in I_{1,i}} a_k v_k - \sum_{k\in I_{2,i}} c_k v_k\right|} M_1 M_{n-1}\dots M_{n-i} M_i \\ & \times \begin{cases} (2i+1)!, & i < \lceil n/2\rceil, \\ [2(n-i)+2]!, & i \ge \lceil n/2\rceil. \end{cases} \end{aligned} \tag{6.25}$$

*Proof.* First of all, we notice that the weights are computed in the order $v_0, v_n, v_1, v_{n-1}, v_2, v_{n-2}, \dots, v_{m-1}, v_m$, for $m = \lceil n/2\rceil$. Therefore, when computing $v_i$, $i = 1,\dots,n-1$, the number of $v_k$, $k \in I_{1,i} \cup I_{2,i}$, involved in (6.23) are exactly $2i$, if $i < \lceil n/2\rceil$, and $2(n-i)+1$, otherwise. At the end, they are at most $n$, which is the case of the "central" weight $v_m$. The proof is carried out assuming $i < \lceil n/2\rceil$, but similar arguments can be applied to the case $i \ge \lceil n/2\rceil$.

We first notice that[2]

$$\begin{aligned} \mathrm{fl}(v_i) &= \frac{1}{2^e}\left( b_i w_i (1+\delta_i) - \sum_{k\in I_{1,i}} a_k v_k (1+\upsilon_k)(1+\delta_k) - \sum_{k\in I_{2,i}} c_k v_k (1+\upsilon_k)(1+\delta_k) \right) \\ &= v_i + \frac{1}{2^e}\left( b_i w_i \delta_i - \sum_{k\in I_{1,i}} a_k v_k (\upsilon_k + \delta_k + O(\epsilon^2)) - \sum_{k\in I_{2,i}} c_k v_k (\upsilon_k + \delta_k + O(\epsilon^2)) \right), \end{aligned}$$

where $\delta_j$, $j = i$ or $j \in I_{1,i} \cup I_{2,i}$, are the errors introduced by the operations in the addends. In particular, these errors are affected at most[3] by one product and $2i$ sums. Therefore, it follows from (2.5) that $|\delta_j| \le (2i+1)\epsilon + O(\epsilon^2)$. Then, using the fact that $\mathrm{fl}(v_i) - v_i = v_i\upsilon_i$, the triangle inequality, and the upper bounds on the relative errors introduced by the operations in the addends, we obtain

$$|v_i\upsilon_i| \le \frac{1}{2^e}\left[ \left( b_i w_i + \sum_{k\in I_{1,i}} a_k v_k + \sum_{k\in I_{2,i}} c_k v_k \right)(2i+1)\epsilon + \sum_{k\in I_{1,i}} a_k |v_k\upsilon_k| + \sum_{k\in I_{2,i}} c_k |v_k\upsilon_k| + O(\epsilon^2) \right].$$

We know that in $\sum_{k\in I_{1,i}} a_k|v_k\upsilon_k| + \sum_{k\in I_{2,i}} c_k|v_k\upsilon_k|$ are performed $2i-1$ sums, therefore, it follows that

$$|v_i\upsilon_i| \le \frac{1}{2^e}\left[ \left( b_i w_i + \sum_{k\in I_{1,i}} a_k v_k + \sum_{k\in I_{2,i}} c_k v_k \right)(2i+1)\epsilon + (2i-1) M_i \max_{k\in I_{1,i}\cup I_{2,i}} |v_k\upsilon_k| + O(\epsilon^2) \right].$$

[2] Any operation with powers of 2 are exact in floating-point arithmetic, so they do not introduce any relative error.

[3] We assume that the computations of all binomial coefficients involve only integer operations, therefore they do not introduce any floating-point relative error.

Then, we can use this inequality recursively and, recalling that $\upsilon_0 = \upsilon_n = 0$ and each time we go one step back in the recursion the set $I_{1,k} \cup I_{2,k}$ decreases by one, we get

$$|v_i \upsilon_i| \leq \frac{1}{2^e} \max_{j=1,n-1,\dots,n-i,i} \left( b_j w_j + \sum_{k \in I_{1,j}} a_k v_k + \sum_{k \in I_{2,j}} c_k v_k \right) (2i+1) M \epsilon + O(\epsilon^2),$$

where

$$\begin{aligned} M &= 1 + (2i-1)M_i + (2i-1)(2i-2)M_i M_{n-i} + \cdots + (2i-1)! M_i M_{n-i} \dots M_{n-1} M_1 \\ &\leq (2i-1)! M_i M_{n-i} \dots M_{n-1} M_1 \times 2i, \end{aligned}$$

which gives the statement. □

**Lemma 6.6.** *For any $t, w_0, \dots, w_n \in \mathbb{F}$, there exist $\rho_0, \dots, \rho_n \in \mathbb{R}$ such that the Wang–Ball control points $R_i$ in* (6.24) *satisfy* $\mathrm{fl}(R_i) = R_i(1+\rho_i)$, $i = 0, \dots, n$, *with* $|\rho_i| \leq U(R_i)\epsilon + O(\epsilon^2)$ *and*

$$U(R_i) = U(R_i v_i) + U(v_i) + 1,$$

*for $U(v_i)$ in* (6.25) *and*

$$\begin{aligned} U(R_i v_i) \leq & \frac{\max_{j=1,n-1,\dots,n-i,i} \left( b_j w_j |P_j| + \sum_{k \in I_{1,j}} a_k v_k |R_k| + \sum_{k \in I_{2,j}} c_k v_k |R_k| \right)}{\left| b_i w_i P_i - \sum_{k \in I_{1,i}} a_k v_k R_k - \sum_{k \in I_{2,i}} c_k v_k R_k \right|} M_1 M_{n-1} \dots M_{n-i} M_i \\ & \times \begin{cases} (2i+2)!, & i < \lceil n/2 \rceil, \\ [2(n-i)+3]!, & i \geq \lceil n/2 \rceil. \end{cases} \end{aligned} \tag{6.26}$$

*Proof.* The study of the propagation of the error in $R_i v_i = 1/2^e \big( b_i w_i P_i - \sum_{k \in I_{1,i}} a_k v_k R_k - \sum_{k \in I_{2,i}} c_k v_k R_k \big)$ can be done with the same procedure used in Lemma 6.5, with the differences that every addend is now affected by one more product by $P_i$ or $R_k$, and we also have to consider the relative errors $\upsilon_i$ introduced by the weights $v_i$, $i = 0, \dots, n$. Therefore, denoting by $\phi_i$ and $\delta$ the errors introduced by the computation of $R_i v_i$ and the division by $v_i$, respectively, we obtain by (2.5) and Lemma 6.5 that

$$\mathrm{fl}(R_i) = \frac{1}{2^e v_i (1+\upsilon_i)} \left( b_i w_i P_i - \sum_{k \in I_{1,i}} a_k v_k R_k - \sum_{k \in I_{2,i}} c_k v_k R_k \right) (1+\phi_i)(1+\delta), \qquad i = 0, \dots, n, \tag{6.27}$$

for $|\upsilon_i| \leq U(v_i)\epsilon + O(\epsilon^2)$ with $U(v_i)$ in (6.25), $|\phi_i| \leq U(R_i v_i)\epsilon + O(\epsilon^2)$ with $U(R_i v_i)$ in (6.26), and $|\delta| \leq \epsilon$. Therefore, we can use Taylor expansion in (6.27) to get

$$\mathrm{fl}(R_i) = R_i(1+\phi_i)(1+\delta)(1-\upsilon_i + O(\epsilon^2)) = R_i(1+\phi_i - \upsilon_i + \delta + O(\epsilon^2))$$

and the statement follows for $\rho_i = \phi_i - \upsilon_i + \delta$ with

$$|\rho_i| \leq |\phi_i| + |\upsilon_i| + |\delta| \leq (U(R_i v_i) + U(v_i) + 1)\epsilon + O(\epsilon^2).$$

□

It is worth noting that the upper bounds on the relative errors derived for $v_i$ and $R_i$ appear to be large even for moderate values of $n$. However, in our experiments, we did not observe instability in their implementations, even when considering $n = 50$. Therefore, we believe that there is room for improvement in these bounds.

### 6.2.2 Horner schemes

We continue our analysis by studying the error propagation that occurs in the algorithms that evaluate a rational Bézier curve $P$ at $t$ through a Horner scheme, which is the case of the implementations of the two formulas in (6.7) and (6.8). In these specific contexts, we can use Theorem 3.1 as both formulas fit the expression in (3.8) for $N = M = n$, $f_k = P_k$, and $a_k = b_k = x^{n-k}\binom{n}{k}w_k$ or $a_k = b_k = t^k s^{n-k}\binom{n}{k}w_k$, $k = 0, \dots, n$, respectively. Moreover, assuming that the binomial coefficients are implemented without introducing any floating-point relative error via integer arithmetic, the computations of the $a_k$, $k = 0, \dots, n$, involve two products plus at most $n$ subtractions, $n$ divisions, and $n-1$ products for $x^{n-k}$ in (6.7) and two products plus at most $n$ subtractions and $n-1$ products in case of $t^k s^{n-k}$ in (6.8). This implies that $A = B = 3n+1$ in case of formula in (6.7) and $A = B = 2n+1$ in case of (6.8). Therefore, it follows from Theorem 3.1 that the relative error $E$ in (6.19) for $P(t)$ computed with (6.7) satisfies

$$E(t) \leq \frac{\sum_{k=0}^{n} |B_k^n(t) w_k P_k|}{|\sum_{k=0}^{n} B_k^n(t) w_k P_k|}(4n+3)\epsilon + (4n+1)\epsilon + O(\epsilon^2), \tag{6.28}$$

while with (6.8)

$$E(t) \leq \frac{\sum_{k=0}^{n} |B_k^n(t) w_k P_k|}{|\sum_{k=0}^{n} B_k^n(t) w_k P_k|}(3n+3)\epsilon + (3n+1)\epsilon + O(\epsilon^2).$$

Notably, the difference $1-t$ cannot be problematic, because we assume that $t$ is an exact floating-point number. However, if instead $t$ is the floating-point approximation of a real number, then the formula in (6.8) may become unstable when $t$ approaches 1. Conversely, the formula in (6.7) represents a stable way to evaluate $P$ thanks to the distinction of the two cases in the definition of $x$.

### 6.2.3 Geometric approach

We proceed to analyse the error propagation of the recursive algorithm given by the formulas in (6.9). In particular, we first study how the error propagates during the computation of the values $h_i$, $i = 0, \dots, n$, and then we examine the relative errors of the values $T_i$, $i = 0, \dots, n$. This analysis finally leads to an upper bound on the relative error $E$ in (6.19) in the case of $P(t) = T_n$.

**Lemma 6.7.** *For any $t, w_0, \dots, w_n \in \mathbb{F}$, there exist $\eta_0, \dots, \eta_n \in \mathbb{R}$ such that the $h_i$ in* (6.9) *satisfy* $\mathrm{fl}(h_i) = h_i(1+\eta_i)$, $i = 0, \dots, n$, *with* $|\eta_i| \leq U(h_i)\epsilon + O(\epsilon^2)$ *and*

$$U(h_i) = 2^3(2^i - 1).$$

*Proof.* We first notice that

$$\begin{aligned}\mathrm{fl}(h_i) &= \frac{w_i h_{i-1}(1+\eta_{i-1})t(n-i+1)(1+\delta_1)}{w_{i-1}i(1-t)(1+\delta_2) + w_i h_{i-1}(1+\eta_{i-1})t(n-i+1)(1+\delta_3)} \\ &= \frac{w_i h_{i-1} t(n-i+1)(1+\delta_1+\eta_{i-1}+O(\epsilon^2))}{w_{i-1}i(1-t)(1+\delta_2) + w_i h_{i-1} t(n-i+1)(1+\delta_3+\eta_{i-1}+O(\epsilon^2))},\end{aligned}$$

where $\delta_1$ is the error introduced by the floating-point operations in the numerator, that are three products and one division, and $\delta_2$ and $\delta_3$ are those related to the first and second addends in

the denominator, respectively, that are two products, one subtraction, and one sum for the former and three products and one sum for the latter. Therefore, it follows from (2.5) that $|\delta_1|, |\delta_2|, |\delta_3| \leq 4\epsilon + O(\epsilon^2)$. Moreover, the intermediate value theorem further guarantees that

$$\mathrm{fl}(h_i) = \frac{w_i h_{i-1} t(n-i+1)(1+\delta_1+\eta_{i-1}+O(\epsilon^2))}{[w_{i-1} i(1-t) + w_i h_{i-1} t(n-i+1)](1+\delta_{i-1})},$$

for some $\delta_{i-1} \in [\min(\delta_2, \delta_3+\eta_{i-1}+O(\epsilon^2)), \max(\delta_2, \delta_3+\eta_{i-1}+O(\epsilon^2))] = [\delta_2, \delta_3+\eta_{i-1}+O(\epsilon^2)]$, and the Taylor expansion of $1/(1+\delta_{i-1})$ gives

$$\begin{aligned}\mathrm{fl}(h_i) &= \frac{w_i h_{i-1} t(n-i+1)}{w_{i-1} i(1-t) + w_i h_{i-1} t(n-i+1)}(1+\delta_1+\eta_{i-1}-\delta_{i-1}+O(\epsilon^2)) \\ &= h_i(1+\delta_1+\eta_{i-1}-\delta_{i-1}+O(\epsilon^2)).\end{aligned}$$

We define $\eta_i = \delta_1 + \eta_{i-1} - \delta_{i-1} + O(\epsilon^2)$, hence, by using the triangle inequality and the upper bounds on the relative errors introduced by the operations, we have

$$|\eta_i| \leq |\delta_1| + |\eta_{i-1}| + |\delta_{i-1}| + O(\epsilon^2) \leq 8\epsilon + 2|\eta_{i-1}| + O(\epsilon^2), \qquad i = 1, \dots, n.$$

Now, we can prove the statement by induction over $i$. The base case follows by the fact that $h_0 = 1$, therefore $\eta_0 = 0$. Finally, the inductive step from $i-1$ to $i$ follows immediately from the inductive hypothesis, that is $|\eta_{i-1}| \leq 2^3(2^{i-1}-1)\epsilon + O(\epsilon^2)$. □

**Proposition 6.8.** *For any $t, w_0, \dots, w_n, P_0, \dots, P_n \in \mathbb{F}$ and $r \in \{1, \dots, n\}$, the relative errors of the $T_i$ in* (6.9) *satisfy*

$$\frac{|\mathrm{fl}(T_i(t)) - T_i(t)|}{|T_i|} \leq \frac{\sum_{k=0}^{i} B_k^n(t)|P_k w_k|}{\left|\sum_{k=0}^{i} B_k^n(t) P_k w_k\right|}\left(\max_{k=1,\dots,i} \frac{1}{1-h_k} 2^3 i(2^i-1) + 3i\right)\epsilon + O(\epsilon^2),$$

*$i = 1, \dots, n$. Therefore, the relative error in* (6.19) *for $P(t) = T_n$ satisfies*

$$E(t) \leq \frac{\sum_{k=0}^{n} B_k^n(t)|P_k w_k|}{\left|\sum_{k=0}^{n} B_k^n(t) P_k w_k\right|}\left(\max_{k=1,\dots,n} \frac{1}{1-h_k} 2^3 n(2^n-1) + 3n\right)\epsilon + O(\epsilon^2).$$

*Proof.* Denoting by $\tau_i$ the relative errors introduced by the computation of $T_i$, $i = 0, \dots, n$, we first notice that

$$\begin{aligned}\mathrm{fl}(T_i) &= [1-h_i(1+\eta_i)]T_{i-1}(1+\tau_{i-1})(1+\delta_1) + h_i(1+\eta_i)P_i(1+\delta_2) \\ &= (1-h_i)T_{i-1}\left(1 - \frac{h_i\eta_i}{1-h_i} + \tau_{i-1} + \delta_1 + O(\epsilon^2)\right) + h_i P_i(1+\eta_i+\delta_2+O(\epsilon^2)) \\ &= T_i + (1-h_i)T_{i-1}\left(-\frac{h_i\eta_i}{1-h_i} + \tau_{i-1} + \delta_1 + O(\epsilon^2)\right) + h_i P_i(\eta_i+\delta_2+O(\epsilon^2)),\end{aligned}$$

where $|\eta_i| \leq 2^3(2^i-1)\epsilon + O(\epsilon^2)$ by Lemma 6.7 and $\delta_1$ and $\delta_2$ are the errors introduced by the operations in the first and second addends, respectively, that are one product and one sum each, plus one subtraction for the first addend only. Therefore, it follows from (2.5) that $|\delta_1|, |\delta_2| \leq 3\epsilon + O(\epsilon^2)$. Then, using the fact that $\mathrm{fl}(T_i) - T_i = T_i\tau_i$, the triangle inequality, and the upper bounds on the relative errors introduced by the values $h_i$ and the operations, we obtain

$$\begin{aligned}|T_i\tau_i| &\leq (1-h_i)|T_{i-1}|\left(\frac{h_i|\eta_i|}{1-h_i} + |\delta_1|\right) + h_i|P_i|(|\eta_i| + |\delta_2|) + (1-h_i)|T_{i-1}\tau_{i-1}| + O(\epsilon^2) \\ &\leq [(1-h_i)|T_{i-1}| + h_i|P_i|]\left(\frac{1}{1-h_i} 2^3(2^i-1) + 3\right)\epsilon + (1-h_i)|T_{i-1}\tau_{i-1}| + O(\epsilon^2).\end{aligned} \tag{6.29}$$

Recalling that $h_j = B_j^n w_j \big/ \sum_{k=0}^{j}(B_k^n w_k)$ [82], we can use recursively the relation of $T_i$ in (6.9) to express

$$(1-h_i)|T_{i-1}| = \sum_{j=1}^{i}\prod_{k=0}^{i-j}(1-h_{i-k})h_{j-1}|P_{j-1}| = \sum_{j=1}^{i}\prod_{k=0}^{i-j}\left(1-\frac{B_{i-k}^n w_{i-k}}{\sum_{l=0}^{i-k}B_l^n w_l}\right)\frac{B_{j-1}^n w_{j-1}}{\sum_{l=0}^{j-1}B_l^n w_l}|P_{j-1}|$$
$$= \sum_{j=1}^{i}\prod_{k=0}^{i-j}\frac{\sum_{l=0}^{i-k-1}B_l^n w_l}{\sum_{l=0}^{i-k}B_l^n w_l}\frac{B_{j-1}^n w_{j-1}}{\sum_{l=0}^{j-1}B_l^n w_l}|P_{j-1}| = \frac{\sum_{j=1}^{i}B_{j-1}^n|w_{j-1}P_{j-1}|}{\sum_{l=0}^{i}B_l^n w_l},$$

and, by (6.29),

$$|T_i\tau_i| \le \frac{\sum_{k=0}^{i}B_k^n(t)|P_k w_k|}{\sum_{k=0}^{i}B_k^n(t)w_k}\left(\frac{1}{1-h_i}2^3(2^i-1)+3\right)\epsilon + (1-h_i)|T_{i-1}\tau_{i-1}| + O(\epsilon^2). \tag{6.30}$$

Now, we can prove the statement by induction over $i = 1,\dots,n$ and, to this end, we recall $T_i = \sum_{k=0}^{i}B_k^n w_k P_k / \sum_{k=0}^{i}(B_k^n w_k)$ [82]. The base case follows by the fact that $T_0 = P_0$, therefore $\tau_0 = 0$. Finally, the inductive step from $i-1$ to $i$ follows from the inductive hypothesis, that is,

$$|T_{i-1}\tau_{i-1}| \le \frac{\sum_{k=0}^{i-1}B_k^n(t)|P_k w_k|}{\sum_{k=0}^{i-1}B_k^n(t)w_k}\left(\max_{k=1,\dots,i-1}\frac{1}{1-h_k}2^3(i-1)(2^{i-1}-1)+3\right)\epsilon + O(\epsilon^2),$$

together with (6.30) and

$$(1-h_i)\frac{\sum_{k=0}^{i-1}B_k^n(t)|P_k w_k|}{\sum_{k=0}^{i-1}B_k^n(t)w_k} = \frac{\sum_{k=0}^{i-1}B_k^n(t)|P_k w_k|}{\sum_{k=0}^{i}B_k^n(t)w_k} \le \frac{\sum_{k=0}^{i}B_k^n(t)|P_k w_k|}{\sum_{k=0}^{i}B_k^n(t)w_k}.$$

□

### 6.2.4 Barycentric approach

In this case, we observe that the barycentric form of $P$ in (6.17) can be expressed as in (3.8) with $N = M = n$, $f_i = Q_i$, and $a_i = b_i = u_i/(t-t_i)$, $i = 0,\dots,n$, therefore we can use once again Theorem 3.1. However, the latter assumes that the values $f_i$ are floating-point numbers, while the $Q_i$ are the result of a prepossessing step that leads to a set of perturbed initial data. Consequently, we derive an upper bound on the relative error $E$ in (6.19) via Theorem 3.1, while also considering this difference.

**Corollary 6.9.** *Assuming that the values $Q_i = P(t_i)$, $i = 0,\dots,n$, are computed by evaluating the rational Bézier curve $P$ at $t_i$ through the implementation of the VS formula in* (6.7) *and that the $z(t_i)$ are defined as in* (6.18)*, then the relative error in* (6.19) *for $P(t)$ computed by implementing the barycentric formula in* (6.17) *satisfies*

$$E(t) \le \left(10n+5+\max_{j=0,\dots,n}U(Q_j)\right)\frac{\sum_{i=0}^{n}\left|\frac{u_i Q_i}{t-t_i}\right|}{\left|\sum_{i=0}^{n}\frac{u_i Q_i}{t-t_i}\right|}\epsilon + (10n+3)\frac{\sum_{i=0}^{n}\left|\frac{u_i}{t-t_i}\right|}{\left|\sum_{i=0}^{n}\frac{u_i}{t-t_i}\right|}\epsilon + O(\epsilon^2),$$

*where*

$$U(Q_i) = \frac{\sum_{i=0}^{n}|B_i^n(t_i)w_i P_i|}{|\sum_{i=0}^{n}B_i^n(t_i)w_i P_i|}(4n+3)+4n+1, \qquad i = 0,\dots,n. \tag{6.31}$$

*Proof.* Since the values $Q_i$ are computed with (6.7), we know from (6.28) that there exist $\theta_0,\dots,\theta_n \in \mathbb{R}$ such that they satisfy $\mathrm{fl}(Q_i) = Q_i(1+\theta_i)$, $i = 0,\dots,n$, with $|\theta_i| \le U(Q_i)\epsilon + O(\epsilon^2)$ and $U(Q_i)$ in (6.31). Moreover, in the computation of the $z(t_i)$ we first introduce at most $4n+1$ floating-point relative errors in the VS algorithm to get the denominators $\sum_{i=0}^n x^{n-i}\binom{n}{i}w_i$, and then we perform at most other $2n-1$ products, which happens in the case of $(1-t)^n$. This means that there exist $\zeta_0,\dots,\zeta_n \in \mathbb{R}$ such that the $z(t_i)$ satisfy $\mathrm{fl}(z(t_i)) = z(t_i)(1+\zeta_i)$, $i = 0,\dots,n$, with $|\zeta_i| \le U(z(t_i))\epsilon + O(\epsilon^2)$ and

$$U(z(t_i)) = 6n.$$

Also, we know that the computation of $u_i/z(t_i) = \prod_{k\neq i}\frac{1}{t_i-t_k}$ introduces at most $3n$ floating-point operations [34, Lemma 1], which, together with the previous equation, leads to the existence of $\mu_0,\dots,\mu_n \in \mathbb{R}$ such that the $u_i$ satisfy $\mathrm{fl}(u_i) = u_i(1+\mu_i)$, $i = 0,\dots,n$, with $|\mu_i| \le U(u_i)\epsilon + O(\epsilon^2)$ and

$$U(u_i) = U(z(t_i)) + 3n + 1 = 9n + 1.$$

Finally, the statement follows by Corollary 3.2 that, by also considering the initial errors in the data $Q_i$, gives

$$E(t) \le \left(n + 4 + \max_{i=0,\dots,n} U(u_i) + \max_{j=0,\dots,n} U(Q_j)\right)\frac{\sum_{i=0}^n \left|\frac{u_iQ_i}{t-t_i}\right|}{\left|\sum_{i=0}^n \frac{u_iQ_i}{t-t_i}\right|}\epsilon$$
$$+\left(n + 2 + \max_{i=0,\dots,n} U(u_i)\right)\frac{\sum_{i=0}^n \left|\frac{u_i}{t-t_i}\right|}{\left|\sum_{i=0}^n \frac{u_i}{t-t_i}\right|}\epsilon + O(\epsilon^2).$$

□

### 6.2.5 Summary

By defining the *conditioning functions*

$$\kappa_P(t) = \frac{\sum_{k=0}^n B_k^n(t)|P_kw_k|}{\left|\sum_{k=0}^n B_k^n(t)P_kw_k\right|}, \qquad \kappa_R(t) = \frac{\sum_{k=0}^n A_k^n(t)|R_kv_k|}{\left|\sum_{k=0}^n A_k^n(t)R_kv_k\right|}, \qquad \text{and} \qquad \kappa_Q(t) = \frac{\sum_{i=0}^n \left|\frac{u_iQ_i}{t-t_i}\right|}{\left|\sum_{i=0}^n \frac{u_iQ_i}{t-t_i}\right|}, \tag{6.32}$$

and recalling that

$$\Lambda_n(t) = \frac{\sum_{i=0}^n \left|\frac{u_i}{t-t_i}\right|}{\left|\sum_{i=0}^n \frac{u_i}{t-t_i}\right|} \tag{6.33}$$

represents the Lebesgue function, we proved that the relative error $E$ in (6.19) can be bounded as

$$E(t) \le \begin{cases} \kappa_P(t)(3n^2+5n)\epsilon + O(\epsilon^2), & P(t) = P_0^n \text{ in (6.6)},\\ \kappa_P(t)(4n+3)\epsilon + (4n+1)\epsilon + O(\epsilon^2), & P(t) \text{ in (6.7)},\\ \kappa_P(t)(3n+3)\epsilon + (3n+1)\epsilon + O(\epsilon^2), & P(t) \text{ in (6.8)},\\ \kappa_P(t)\left(\max\limits_{k=1,\dots,n}\dfrac{1}{1-h_k}2^3n(2^n-1)+3n\right)\epsilon + O(\epsilon^2), & P(t) = P_0^n \text{ in (6.9)},\\ \kappa_R(t)\left(3n^2+5n+\max\limits_{j=0,\dots,n}U(v_j)+\max\limits_{k=0,\dots,n}U(R_k)\right)\epsilon + O(\epsilon^2), & P(t) = R_0^n \text{ in (6.12)–(6.14)},\\ \kappa_Q(t)\left(10n+5+\max\limits_{j=0,\dots,n}U(Q_j)\right)\epsilon + \Lambda_n(t)(10n+3)\epsilon + O(\epsilon^2), & P(t) \text{ in (6.17)}. \end{cases}$$

Hence, we expect that all methods that use the Bernstein basis in (6.2), namely those defined by the formulas in (6.6)–(6.9), behave similarly in terms of numerical stability. The only exception might arise with the latter method if any of the $h_k$, $k = 1, \dots, n$, is very close to 1. However, Woźny and Chudy [82] have already addressed this issue by suggesting to use the relation

$$1 - h_k = \frac{h_k}{h_{k-1}} \frac{w_{k-1} k(1-t)}{w_k t(n-k+1)}.$$

Regarding instead the methods that employ a different basis, such as the Wang–Ball and the barycentric algorithms, even under the assumption that all the preprocessing steps are stable, there are scenarios where $\kappa_R$ or $\kappa_Q$ are bigger than $\kappa_P$, or vice versa. As a consequence, these algorithms may exhibit instability even when the formulas in (6.6)–(6.9) are stable. However, for the barycentric form, instability is less likely to occur if Chebyshev nodes are chosen. In fact, multiplying both numerator and denominator of the function $\kappa_Q$ by $\left|\sum_{i=0}^{n} u_i Q_i/(t-t_i)\right|$, we can see that

$$\kappa_Q(t) = \frac{\sum_{i=0}^{n} \left|\frac{u_i Q_i}{t-t_i}\right|}{\left|\sum_{i=0}^{n} \frac{u_i}{t-t_i}\right|} \frac{1}{|P(t)|} \leq \max_{i=0,\dots,n} |Q_i| \Lambda_n(t) \frac{1}{|P(t)|} \leq \max_{i=0,\dots,n} |P_i| \Lambda_n(t) \kappa_P(t) \frac{\sum_{k=0}^{n} B_k^n(t)|w_k|}{\sum_{k=0}^{n} B_k^n(t)|P_k w_k|}.$$

In particular, if $\min_{i=0,\dots,n} |P_i| \neq 0$, then

$$\kappa_Q(t) \leq \kappa_P(t) \Lambda_n(t) \frac{\max_{i=0,\dots,n} |P_i|}{\min_{i=0,\dots,n} |P_i|}. \tag{6.34}$$

Moreover, it is well known [76] that the Lebesgue function grows only logarithmically in $n$ for Chebyshev nodes. Therefore, if $\kappa_P$ has a good behaviour, then we can expect the method to be always stable when the ratio between the biggest and the smallest control points is small. On the contrary, the Lebesgue function related to equidistant nodes exhibits exponential growth in $n$ [76], hence we can have unstable results even for moderate values of $n$ with uniformly distributed nodes. We will show that these scenarios can indeed occur in the next section through numerical experiments.

## 6.3 Numerical experiments

We implemented all the methods in C++ and computed the exact value $P(t)$ of the Bézier curve in multiple-precision (1024 bit) floating-point arithmetic with the *MPFR* library [33]. The results are obtained using a Ubuntu system on a Dell computer with 8 cores i7-10510U CPU 1.80GHz and 16 GiB of RAM. The codes are compiled with *CMake* compiler optimisation flag -O3. Below, we indicate the algorithms presented previously with the following abbreviations:

**RDF** = Rational de Casteljau algorithm implemented with formula in (6.5)

**FDC** = Farin de Casteljau algorithm implemented with formula in (6.6)

**RVS** = Rational VS algorithm implemented with formula in (6.7)

**RHB** = Rational Horner-Bézier algorithm implemented with formula in (6.8)

**LTG** = Linear time geometric algorithm implemented with formula in (6.9)

**RWB** = Rational Wang–Ball algorithm implemented with formula in (6.12)– (6.14)

**CHE** = Barycentric algorithm implemented with formula in (6.17) and Chebyshev nodes

**UNI** = Barycentric algorithm implemented with formula in (6.17) and uniform nodes.

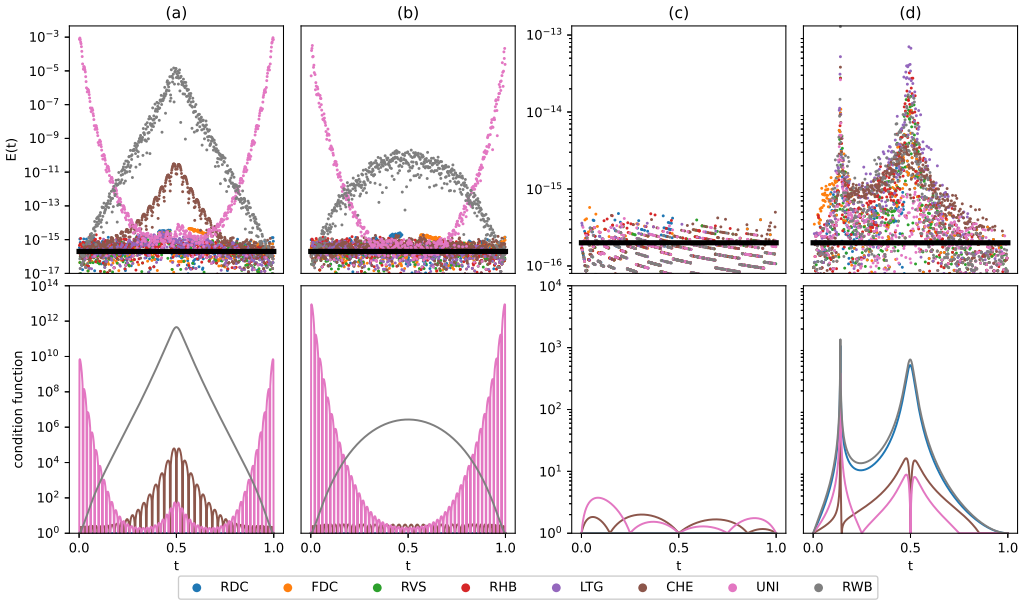


Figure 6.1. Relative errors of all algorithms (top) for computing a rational Bézier curve and their related conditioning function (bottom) on a logarithmic scale. We first consider $n = 50$, $P_i = \binom{x_i}{y_i}$ for $x_i$ and $y_i$ in (6.35), and $w_i = i \mod 2 + 1$, and we observe the results related to the $x$-coordinate (a) and $y$-coordinate (b). Then, we set $n = 4$, $P_0 = \binom{10}{-100}, P_1 = \binom{20}{200}, P_2 = \binom{30}{-200}, P_3 = \binom{40}{101}, P_4 = \binom{50}{101}$ and $w_i = 1$, $i = 0, \ldots, n$, and we see the results for the $x$-coordinate (c) and $y$-coordinate (d). The black line represents the machine epsilon in double precision.

To compare the numerical stability of all the algorithms, we evaluate the relative error $E$ in (6.19) for 1000 equidistant evaluation points in $[0, 1]$ using the various implementations of $\mathrm{fl}(P(t))$ in double precision. If the results are on the order of the machine epsilon, approximately $10^{-16}$, then we can conclude that the method is stable, otherwise it suggests instability.

In the first experiment, we consider a rational Bézier curve of degree $n = 50$ with control points $P_i = \binom{x_i}{y_i}$, $i = 0, \ldots, n$, for

$$x_i = \begin{cases} 1, & i = 0, \ldots, 9 \text{ and } i = 41, \ldots, n \\ 10^6, & i = 10, \ldots, 40 \end{cases} \qquad \text{and} \qquad y_i = \sin \frac{i\pi}{n+1}, \tag{6.35}$$

and weights $w_i = i \mod 2 + 1$. In Figure 6.1 (a,b), we observe that all the methods defined via the Bernstein basis are stable, while the others exhibit numerical problems. In particular, the CHE, UNI, and RWB algorithms are unstable with respect to the $x$-coordinate (a), as well as for the $y$- coordinate (b), except for the CHE algorithm. Our theoretical results in Section 6.2 can explain the causes of these instabilities. Indeed, we proved that the relative error of the RWB algorithm depends on the conditioning functions $\kappa_R$, while those of CHE and UNI on $\kappa_Q$. In this case, even though the initial data give a good conditioning function related to the Bernstein basis, i.e. $\kappa_P(t) = 1$, the conversion to a different basis leads to unfavorable behaviour for both $\kappa_R$ and $\kappa_Q$, as shown in Figure 6.1 (bottom). Moreover, it is worth noting that, as expected, the CHE algorithm exhibits instability with respect to the $x$-coordinate because the ratio between

the biggest and the smallest $|x_i|$ is $10^6$. However, under circumstances where this ratio is not big, the CHE algorithm is typically stable, even for large values of $n$.

Finally, we want to examine a more realistic experiment, thus we take a low degree curve by setting $n = 4$. On the one hand, we observe in Figure 6.1 that the relative error related to the $x$-coordinate (c) is perfectly stable, with all the conditioning functions small. On the other hand, all the relative errors related to the $y$-coordinate (d) exhibit spikes in some parts of the domain, reaching an order of $10^{-13}$. This behaviour is also reflected in the conditioning functions. Yet, where these spikes occur, the values of $P_y(t)$ are very small because the curve is crossing the $t$-axis, therefore this may not be a stability issue, but rather a consequence of dividing by very small values in $E_y(t)$ in (6.19). However, for our 1000 equidistant evaluation points, the minimum absolute value of $P_y(t)$ is 0.13424 in the domain of the first spike and 0.3116 for the second, thus indicating that we are not so close to the values where $P_y(t) = 0$. Furthermore, plotting the absolute errors leads to a similar behaviour without these spikes, but still with magnitudes between $10^{-14}$ and $10^{-13}$ for all methods except the barycentric form with uniform nodes. This latter remains stable, as its conditioning function $\kappa_Q$ is small, apart from the initial spike, and its nodes are far from the instability regions of the RVS algorithm. In contrast, Chebyshev nodes compromise the stability of the method due to the computation of one interpolation point with the RVS algorithm where it is unstable, specifically for the node in $[0.1, 0.2]$. Furthermore, while both uniform and Chebyshev nodes include $t = 0.5$, the RVS is exact at this point with a relative error of 0, thus preserving the stability of the barycentric method with uniform nodes. However, the computation of this node with the Chebyshev formula is not exact, resulting in perturbed data.

# Chapter 7

# Conclusion

The focus of this dissertation is to study the numerical stability of the algorithms that implement different types of barycentric interpolants. We begin with the univariate case, where barycentric interpolation offers a fast and stable means of evaluating both polynomial and rational interpolants using either the first or the second barycentric form. On the one hand, the first polynomial form is always numerically stable, while the second form is stable only for interpolation nodes with a small Lebesgue constant. On the other hand, evaluating a rational interpolant via the second barycentric form comes with the same limitation, but for the special family of barycentric rational interpolants with weights in (3.5) proposed by Floater and Hormann, a computationally more attractive first barycentric form is available. Instead of depending on the Lebesgue constant, both the forward and the backward stability of a straightforward implementation of this first barycentric form depend on the function $\Gamma_d$ in (3.18).

Unlike the Lebesgue constant, Theorem 3.13 shows that the maximum of $\Gamma_d$ is independent of $n$. Moreover, it is guaranteed to be very small for equidistant nodes, regardless of $d$, while the Lebesgue constant is known to grow logarithmically in $n$ and exponentially in $d$ in this case. Based on our numerical experiments, the maximum of $\Gamma_d$ seems to be small for other distributions of interpolation nodes, too, even if the mesh ratio $\mu$ is big, as in the example in Section 3.7.2, and we believe that the upper bound in (3.26) can be improved considerably in future work. For example, if $d = n$, then $\Gamma_d$ is just the constant one function, independent of $\mu$.

Overall, in the case of a linear barycentric rational interpolant with weights in (3.5), we cannot state that one form, whether the first or the second, is generally superior to the other, as the optimal choice depends on the specific scenario. If we know a priori that the interpolation nodes generate a small Legesgue constant, then the most efficient and stable algorithm is undoubtedly the one based on the straightforward implementation of the second barycentric form in (3.3), with weights precomputed with the pyramid algorithm in (3.21) and (3.22). However, if the Lebesgue constant is not small or is unknown, then the first barycentric form is much more likely to provide a stable solution. In such cases, the best approach is to implement first the weights as before in a preprocessing step, and then, for every evaluation point $x$, to compute the values $\lambda_i(x)$ with our iterative strategy in (3.23) and finally the value of the interpolant using a straightforward implementation of the first barycentric form in (3.6). Alternatively computing the sum of the $\lambda_i(x)$ in the denominator with Camargo's algorithm [18] results in slightly smaller forward errors, but is significantly slower, especially for larger $d$.

To address these considerations, we develop the BRI (Barycentric Rational Interpolation)

class, designed to handle all aspects of barycentric rational interpolation in a stable, robust, and efficient manner. The class provides a C++ interface and supports both standard data types and arbitrary precision arithmetic via the Multiple Precision Floating-Point Reliable (MPFR) library. Unlike existing libraries that evaluate barycentric rational interpolants exclusively using the second form, the BRI class autonomously selects the most appropriate method based on the specific scenario using the results discussed before and presented in Chapter 3. Furthermore, we introduce a novel technique to improve robustness and reduce the risk of overflow and underflow errors.

In the bivariate case, our investigations regard the numerical stability and efficiency of the algorithms that evaluate mean value coordinates and they reveal the following, partially surprising insights. First, among the six formulas in (5.1)–(5.6), which give rise to efficient $O(n)$ algorithms, the original expression in (5.1) generally performs best in terms of stability and is as fast as the others. This is contrary to the common belief that using the ATAN2 function (for computing the angles $\alpha_i$) and the TAN function (for evaluating $\tan(\alpha_i/2)$) is slow. At least on our platform, we did not notice any computational disadvantage.

Second, the implementation of the original formula works well, even if $v$ is on one of the edges of the polygon, say $v = (1-\mu)v_k + \mu v_{k+1}$ for some $k$ and $\mu \in (0,1)$, despite the fact that $\alpha_k = \pm\pi$ in this case, hence $\tan(\alpha_k/2)$ is mathematically not well-defined. Since common floating-point implementations cannot represent $\pm\pi/2$ exactly, the TAN function does not return NAN in this case, but rather a number that is extremely big in absolute value, and the mean value coordinates $\lambda_i$ happen to be correct, up to machine precision, in the end. However, we observed major numerical problems for polygons with edges that are very close to each other (see Figure 5.3). In the vicinity of such edges, two of the values $\tan(\alpha_i/2)$ are very big, which eventually leads to a loss of precision.

Third, our new Algorithm 10 handles even such extreme cases and is generally the most stable of all methods. It also works if $v$ is a vertex of the polygon, a case that needs to be detected in the linear-time algorithms by checking if some $r_i$ equals zero. The only other method that does not require any exceptions for handling points on the boundary of the polygon is the one based on (5.7), but it turns out to be slower and less stable than our approach, especially near the sets $Z_k$ in (5.8).

Finally, since the findings of Chapter 3 are valid for a wide range of methods, we applied them to conduct a comprehensive comparison of the most common algorithms used to evaluate a rational Bézier curve in terms of numerical stability. We derived an upper bound on the relative error of the different methods and showed, both theoretically and empirically, that it depends on certain conditioning functions. Specifically, algorithms that use the Bernstein basis depend on the same conditioning function, therefore they have consistent numerical behaviours. Instead, conversion to another basis can lead to different relative errors. In fact, there are scenarios where all algorithms are stable, except from those given by the Wang–Ball and the barycentric basis. However, we proved that, if the Bernstein basis gives a good conditioning function, then also the basis related to the barycentric algorithm with Chebyshev nodes behaves well, as long as the ratio between the largest and smallest control points $P_i$ is small. Lastly, even classical algorithms based on the Bernstein basis may fail if the associated conditioning function is large, particularly when control points have different signs. In such cases, it is possible that the conversion to the barycentric form with nodes located away from instability areas can yield better results.

# Appendix A

# User manual for the BRI class

This appendix explains how to use the functions of the BRI `C++` class template for barycentric rational interpolation presented in Chapter 4.

## Installation

To use this class, include the header file BRI.h with the `#include` directive in all .cpp files that require it and place them in the same folder. Since the BRI class is using templates, the user can decide which types of input and output variables to use, and we denote them respectively by `inT` and `outT` in the following. Other than the standard data types available in `C++` , these can also be defined in arbitrary precision, because the BRI class is compatible with the publicly available MPFR library using the MPFR `C++` interface.

## The BRI class

The BRI class contains all variables and functions related to a barycentric rational interpolant. Even though the variables are private, they still need to be defined by the user via constructors and can also be modified via class member functions. For this reason, we first present the parameters defined within the class and then its public functions.

### Variables

`int n`

number of interpolation nodes minus 1

`int d`

integer parameter related to Floater–Hormann rational interpolation

`vector<inT>& Xn`

vector of dimension $n+1$ containing the interpolation nodes

`vector<inT>& Yn`

vector of dimension $n+1$ containing the data associated with the corresponding node

`vector<outT>& Wn`

vector of dimension $n+1$ containing the barycentric weights associated with $X_n$ and $d$

## Constructors

`BRI(const vector<inT>& Xn, const vector<inT>& Yn, int d)`

takes as input both vectors $X_n$ and $Y_n$ by reference and the integer $d$ by value

`BRI(string nodes, const vector<inT>& Yn, int d)`

fills the vector $X_n$ with the content of the file `nodes`, takes the vector $Y_n$ by reference and the integer $d$ by value

`BRI(Nodes type, inT a, inT b, int n, const vector<inT>& Yn, int d)`

generates the vector $X_n$ by knowing the type of nodes (UNIFORM, CHEBYSHEV, CHEBYSHEV_EXTENDED) and the endpoints of the interval $[a,b]$ in which they are defined, takes the vector $Y_n$ by reference and the integers $n$ and $d$ by value, as well as $a$ and $b$

`BRI(const vector<inT>& Xn, string data, int d)`

takes $X_n$ by reference and the integer $d$ by value and fills the vector $Y_n$ with the content of the file `data`

`BRI(string nodes, string data, int d)`

fills both vectors $X_n$ and $Y_n$ with the content of the files `nodes` and `data`, respectively and takes the integer $d$ by value

`BRI(Nodes type, inT a, inT b, int n, string data, int d)`

generates the vector $X_n$ by knowing the type of nodes (UNIFORM, CHEBYSHEV, CHEBYSHEV_EXTENDED) and the endpoints of the interval $[a,b]$ in which they are defined, fills the vector $Y_n$ with the content of the file `data`, and takes the integers $n$ and $d$ by value, as well as $a$ and $b$

`const BRI(vector<inT>& Xn, inT(*f)(inT x), int d)`

takes $X_n$ by reference and the integer $d$ by value and generates the vector $Y_n = f(X_n)$ using the pointer to an external function $f$

`BRI(string nodes, inT(*f)(inT x), int d)`

fills the vector $X_n$ with the content of the file `nodes`, generates the vector $Y_n = f(X_n)$ using the pointer to an external function $f$, and takes the integer $d$ by value

`BRI(Nodes type, inT a, inT b, int n, inT(*f)(inT x), int d)`

generates the vector $X_n$ by knowing the type of nodes (UNIFORM, CHEBYSHEV, CHEBYSHEV_EXTENDED) and the endpoints of the interval $[a,b]$ in which they are defined, generates the vector $Y_n = f(X_n)$ using the pointer to an external function $f$, and takes the integers $n$ and $d$ by value, as well as $a$ and $b$

## Functions to modify the input

`void set_nodes(const vector<inT>& Xn)`

changes the nodes taking the new vector $X_n$ by reference

`void set_nodes(string nodes)`

changes the nodes filling the new vector $X_n$ with the content of the file `nodes`

`void set_nodes(Nodes type, inT a, inT b, int n)`

changes the nodes generating the new vector $X_n$ by knowing the type of nodes (UNIFORM, CHEBYSHEV, CHEBYSHEV_EXTENDED), the endpoints of the interval $[a, b]$ in which they are defined, and the integer $n$, which are passed by value

`void set_nodes(int j, inT xj)`

changes only the $j$-th entry of the vector $X_n$ with the value $x_j$

`void set_data(const vector<inT>& Yn)`

changes the data taking the new vector $Y_n$ by reference

`void set_data(string data)`

changes the data filling the new vector $Y_n$ with the content of the file `data`

`void set_data(inT(*f)(inT x))`

changes the data generating the new vector $Y_n = f(X_n)$ having the pointer to the external function $f$

`void set_data(int j, inT yj)`

changes only the $j$-th entry of the vector $Y_n$ with the value $y_j$

`void set_degree(int d)`

changes the integer $d$ taking the new one by value

`void add_point(inT xj, inT yj)`

adds $x_j$ and $y_j$ in the vectors $X_n$ and $Y_n$ respectively

`void remove_point(int j)`

removes the $j$-th entry of the vectors $X_n$ and $Y_n$

## Functions to get the input and the weights

`const vector<inT>& get_nodes()`

returns $X_n$

`const inT& get_nodes(int j)`

returns the $j$-th entry of the vector $X_n$

```
const vector<inT>& get_data()
```

returns $Y_n$

```
const inT& get_data(int j)
```

returns the $j$-th entry of the vector $Y_n$

```
const vector<outT>& get_weights()
```

returns $W_n$

```
const outT& get_weight(int j)
```

returns the $j$th entry of the vector $W_n$

```
const vector<outT>& get_weights(int& Cw)
```

returns $W_n$ and the rescaling factor of the weights $C_w$.

## Control flags

```
void guard_on()
void guard_off()
```

the guard flag can be turned on and off; if it is set, then the weights, the evaluation of the barycentric rational interpolant, and the stability-related functions are computed in guarded mode

```
void stability_on()
void stability_off()
```

the stability flag can be turned on and off; if it is set, then the code evaluates the barycentric rational interpolant using the numerically most stable algorithm

```
void efficiency_on()
void efficiency_off()
```

the efficiency flag can be turned on and off; if it is set and the evaluation of the barycentric rational interpolant is done with the first barycentric form, then the most efficient algorithm is used

## Evaluation of the barycentric rational interpolant

Hereafter, all the functions that take an evaluation point $x$ as input can be also called with a vector of evaluation points, returning a vector of values in this case. Moreover, apart from the NUMERATOR function, all the others take also the parameter $A$ of type `Algo` as input. By default, it is set to SMART, so that, if omitted, the function autonomously decides which algorithm to use. Otherwise, it can take one value among {FIRST_DEF, FIRST_EFF, SECOND} to use the standard implementation of the first barycentric form, the efficient variant, or the second barycentric form, respectively.

```
outT numerator(inT x)
vector<outT> numerator(vector<inT>& x)
```

computes the numerator $N$ at the evaluation point $x$

```
outT numerator(inT x, int& CN)
vector<outT> numerator(vector<inT>& x, vector<int>& CN)
```

computes the numerator $N$ at the evaluation point $x$ keeping track of the rescaling factor $C_N$

```
outT denominator(inT x, Algo A = SMART)
vector<outT> denominator(vector<inT>& x, Algo A = SMART)
```

computes the denominator $D$ at the evaluation point $x$

```
outT denominator(inT x, int& CD, Algo A =  SMART)
vector<outT> denominator(vector<inT>& x, vector<int>& CD, Algo A = SMART)
```

computes the denominator $D$ at the evaluation point $x$ keeping track of the rescaling factor $C_D$

```
outT eval(inT x, Algo A = SMART)
vector<outT> eval(vector<inT>& x, Algo A = SMART)
```

evaluates the interpolant $r$ at the evaluation point $x$

```
outT eval(int j, inT h, Algo A = SMART)
```

evaluates the interpolant $r$ at the evaluation point $x_j + h$

## Stability-related functions

```
outT cond(inT x)
vector<outT> cond(vector<inT>& x)
```

computes the condition number $\kappa(x)$ at the evaluation point $x$

```
outT leb(inT x)
vector<outT> leb(vector<inT>& x)
```

computes the Lebesgue function $\Lambda_n(x)$ at the evaluation point $x$

```
outT gamma(inT x)
vector<outT> gamma(vector<inT>& x)
```

computes the function $\Gamma_d(x)$ at the evaluation point $x$

```
outT cond()
```

computes the value $\max_{x\in[x_0,x_n]} \kappa(x)$

```
outT leb()
```

computes the value $\max_{x\in[x_0,x_n]} \Lambda_n(x)$

```
outT gamma()
```

computes the value $\max_{x\in[x_0,x_n]} \Gamma_d(x)$

# Bibliography


[1] Agrawal, N. et al. [2022]. BOOST 1.81.0 Library Documentation – math toolkit 3.3.0 – Interpolation – Barycentric Rational Interpolation, `https://www.boost.org/doc/libs/1_81_0/libs/math/doc/html/math_toolkit/barycentric.html`. [Online; accessed 23-February-2023].

[2] Anisimov, D. [2017]. *Analysis and new constructions of generalized barycentric coordinates in 2D*, PhD thesis, Faculty of Informatics, Universitá della Svizzera italiana.

[3] Anisimov, D., Bommes, D., Hormann, K. and Alliez, P. [2015]. 2D generalized barycentric coordinates, *CGAL User and Reference Manual*, 4.6 edn, CGAL Editorial Board.
**URL:** *https://doc.cgal.org/4.6/Barycentric_coordinates_2/index.html#Chapter_2D_Generalized_Barycentric_Coordinates*

[4] Baker, R. D. and McHale, I. G. [2015]. Time varying ratings in association football: the all-time greatest team is..., *Journal of the Royal Statistical Society: Series A* **178**(2): 481–492.

[5] Berrut, J.-P. [1988]. Rational functions for guaranteed and experimentally well-conditioned global interpolation, *Computers & Mathematics with Applications* **15**(1): 1–16.

[6] Berrut, J.-P., Hosseini, S. A. and Klein, G. [2014]. The linear barycentric rational quadrature method for Volterra integral equations, *SIAM Journal on Scientific Computing* **36**(1): A105–A123.

[7] Berrut, J.-P. and Mittelmann, H. D. [1997]. Lebesgue constant minimizing linear rational interpolation of continuous functions over the interval, *Computers & Mathematics with Applications* **33**(6): 77–86.

[8] Berrut, J.-P. and Trefethen, L. N. [2004]. Barycentric Lagrange interpolation, *SIAM Review* **46**(3): 501–517.

[9] Bézier, P. [1966]. Définition numérique des courbes et surfaces, *Automatisme* **11**: 625–632.

[10] Bézier, P. [1967]. Définition numérique des courbes et surfaces (ii), *Automatisme* **12**: 17–21.

[11] Bochkanov, S. [2022]. ALGLIB 3.20.0 User Guide – Interpolation and fitting – Rational interpolation, `http://www.alglib.net/interpolation/rational.php`. [Online; accessed 23-February-2023].

[12] Boehm, W. and Müller, A. [1999]. On de Casteljau's algorithm, *Computer Aided Geometric Design* **16**(7): 587–605.

[13] Bos, L., De Marchi, S., Hormann, K. and Klein, G. [2012]. On the Lebesgue constant of barycentric rational interpolation at equidistant nodes, *Numerische Mathematik* **121**(3): 461–471.

[14] Bos, L., De Marchi, S., Hormann, K. and Sidon, J. [2013]. Bounding the Lebesgue constant for Berrut's rational interpolant at general nodes, *Journal of Approximation Theory* **169**: 7–22.

[15] Bronshtein, I. N., Semendyayev, K. A., Musiol, G. and Mühlig, H. [2015]. *Handbook of Mathematics*, 6th edn, Springer, Berlin, Heidelberg.

[16] Brutman, L. [1996]. Lebesgue functions for polynomial interpolation - a survey, *Annals of Numerical Mathematics* **4**: 111–128.

[17] Corless, R. M. and Fillion, N. [2013]. *A Graduate Introduction to Numerical Methods*, 1 edn, Springer, New York.

[18] de Camargo, A. P. [2016]. On the numerical stability of Floater–Hormann's rational interpolant, *Numerical Algorithms* **72**(1): 131–152.

[19] de Casteljau, P. [1959]. Outillages methodes calcul, *Technical report*, André Citroën Automobile SA.

[20] Dejdumrong, N. [2006]. Rational DP-Ball curves, *International Conference on Computer Graphics, Imaging and Visualisation*, CGIV'06, pp. 478–483.

[21] Dejdumrong, N. [2008]. Efficient algorithms for non-rational and rational Bézier curves, *International Conference on Computer Graphics, Imaging and Visualisation*, CGIV'08, pp. 109–114.

[22] Dejdumrong, N., Phien, H., Tien, H. and Lay, K. [2001]. Rational Wang–Ball curves, *International Journal of Mathematical Education in Science and Technology* **32**(4): 565–584.

[23] Delgado, J. and Peña, J. M. [2003]. A shape preserving representation with an evaluation algorithm of linear complexity, *Computer Aided Geometric Design* **20**(1): 1–10.

[24] Delgado, J. and Peña, J. M. [2004]. A shape preserving representation for rational curves with efficient evaluation algorithm, *in* M. Sarfraz (ed.), *Advances in Geometric Modeling*, John Wiley, Chichester, chapter 3, pp. 39–54.

[25] Ellingsrud, A. J., Boullé, N., Farrell, P. E. and Rognes, M. E. [2021]. Accurate numerical simulation of electrodiffusion and water movement in brain tissue, *Mathematical Medicine and Biology: A Journal of the IMA* **38**(4): 516–551.

[26] Farin, G. [1983]. Algorithms for rational Bézier curves, *Computer-Aided Design* **15**(2): 73–77.

[27] Farin, G. [2001]. *Curves and Surfaces for CAGD: A Practical Guide*, The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling, 5th edn, Morgan Kaufmann, San Francisco.

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

[29] Floater, M. S. [2014]. Wachspress and mean value coordinates, *in* G. E. Fasshauer and L. L. Schumaker (eds), *Approximation Theory XIV: San Antonio 2013*, Springer International Publishing, Cham, pp. 81–102.

[30] Floater, M. S. and Hormann, K. [2007]. Barycentric rational interpolation with no poles and high rates of approximation, *Numerische Mathematik* **107**(2): 315–331.

[31] Floater, M. S., Hormann, K. and Kós, G. [2006]. A general construction of barycentric coordinates over convex polygons, *Advances in Computational Mathematics* **24**(1–4): 311–331.

[32] Floater, M. S., Kós, G. and Reimers, M. [2005]. Mean value coordinates in 3D, *Computer Aided Geometric Design* **22**(7): 623–631.

[33] Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P. and Zimmermann, P. [2007]. MPFR: A multiple-precision binary floating-point library with correct rounding, *ACM Transactions on Mathematical Software* **33**(2).

[34] Fuda, C., Campagna, R. and Hormann, K. [2022]. On the numerical stability of linear barycentric rational interpolation, *Numerische Mathematik* **152**(4): 761–786.

[35] Fuda, C. and Hormann, K. [2024a]. Algorithm 1048: A C++ Class for Robust Linear Barycentric Rational Interpolation, *ACM Transactions on Mathematical Software* **50**(3).

[36] Fuda, C. and Hormann, K. [2024b]. A new stable method to compute mean value coordinates, *Computer Aided Geometric Design* **111**: Article 102310, 16 pages. Proceedings of GMP.

[37] Fuda, C., Ramanantoanina, A. and Hormann, K. [2024]. A comprehensive comparison of algorithms for evaluating rational bézier curves, *Dolomites Research Notes on Approximation* **17**(3): 56–79.

[38] Gohberg, I. and Koltracht, I. [1993]. Mixed, componentwise, and structured condition numbers, *SIAM Journal on Matrix Analysis and Applications* **14**(3): 688–704.

[39] Goldman, R. [2003]. *Pyramid Algorithms*, The Morgan Kaufmann Series in Computer Graphics, Morgan Kaufmann, San Francisco, chapter 5, pp. 187–306.

[40] Güttel, S. and Klein, G. [2012]. Convergence of linear barycentric rational interpolation for analytic functions, *SIAM Journal on Numerical Analysis* **50**(5): 2560–2580.

[41] Harbrecht, H. and Multerer, M. [2022]. *Algorithmische Mathematik*, 1 edn, Springer Spektrum, Berlin Heidelberg.

[42] Higham, N. J. [1993]. The accuracy of floating point summation, *SIAM Journal on Scientific Computing* **14**(4): 783–799.

[43] Higham, N. J. [2002]. *Accuracy and Stability of Numerical Algorithms*, 2nd edn, SIAM, Philadelphia.

[44] Higham, N. J. [2004]. The numerical stability of barycentric Lagrange interpolation, *IMA Journal of Numerical Analysis* **24**(4): 547–556.

[45] Holoborodko, P. [2008]. Multiple precision floating point arithmetic library for C++, `http://www.holoborodko.com/pavel/mpfr/`. [Online; accessed 17-June-2024].

[46] Hormann, K. [2014]. Barycentric interpolation, *Approximation Theory XIV: San Antonio 2013*, Vol. 83 of *Springer Proceedings in Mathematics & Statistics*, Springer, New York, pp. 197–218.

[47] Hormann, K. and Floater, M. S. [2006]. Mean value coordinates for arbitrary planar polygons, *ACM Transactions on Graphics* **25**(4): 1424–1441.

[48] Hormann, K., Klein, G. and De Marchi, S. [2012]. Barycentric rational interpolation at quasi-equidistant nodes, *Dolomites Research Notes on Approximation* **5**: 1–6.

[49] Hormann, K. and Schaefer, S. [2016]. Pyramid algorithms for barycentric rational interpolation, *Computer Aided Geometric Design* **42**: 1–6.

[50] Hormann, K. and Sukumar, N. (eds) [2017]. *Generalized Barycentric Coordinates in Computer Graphics and Computational Mechanics*, Taylor & Francis, CRC Press, Boca Raton.

[51] Hu, S.-M., Wang, G.-Z. and Jin, T.-G. [1996]. Properties of two types of generalized Ball curves, *Computer-Aided Design* **28**(2): 125–133.

[52] *IEEE Standard for Floating-Point Arithmetic* [2019]. New York. IEEE Std 754-2019 (Revision of IEEE Std 754-2008).

[53] Jianying, W., Haizhao, L., Zheng, Q. and Dong, Y. [2019]. Mapped Chebyshev pseudospectral methods for optimal trajectory planning of differentially flat hypersonic vehicle systems, *Aerospace Science and Technology* **89**: 420–430.

[54] Ju, T., Schaefer, S. and Warren, J. [2005]. Mean value coordinates for closed triangular meshes, *ACM Transactions on Graphics* **24**(3): 561–566.

[55] Klein, G. and Berrut, J.-P. [2012]. Linear barycentric rational quadrature, *BIT Numerical Mathematics* **52**(2): 407–424.

[56] Lee, T.-S., Radak, B. K., Huang, M., Wong, K.-Y. and York, D. M. [2014]. Roadmaps through free energy landscapes calculated using the multidimensional vFEP approach, *Journal of Chemical Theory and Computation* **10**(1): 24–34.

[57] Lee, T.-S., Radak, B. K., Pabis, A. and York, D. M. [2013]. A new maximum likelihood approach for free energy profile construction from molecular simulations, *Journal of Chemical Theory and Computation* **9**(1): 153–164.

[58] Leffell, J., Murman, S. and Pulliam, T. [2013]. An extension of the time-spectral method to overset solvers, *51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, American Institute of Aeronautics and Astronautics, Reston, VA.

[59] Li, J. and Cheng, Y. [2021]. Linear barycentric rational collocation method for solving heat conduction equation, *Numerical Methods for Partial Differential Equations* **37**(1): 533–545.

[60] Loosemore, S., Stallman, R. M., McGrath, R., Oram, A. and Drepper, U. [2023]. Known maximum errors in math functions, *The GNU C Library Reference Manual*, chapter 19.7, pp. 561–601.
**URL:** *https://www.gnu.org/s/libc/manual/pdf/libc.pdf*

[61] Luo, W.-H., Huang, T.-Z., Gu, X.-M. and Liu, Y. [2017]. Barycentric rational collocation methods for a class of nonlinear parabolic partial differential equations, *Applied Mathematics Letters* **68**: 13–19.

[62] MacMillen, D. [2021]. Baryrational, `https://github.com/macd/BaryRational.jl`. [Online; accessed 23-February-2023].

[63] Mascarenhas, W. and Camargo, A. [2014]. The backward stability of the second barycentric formula for interpolation, *Dolomites Research Notes on Approximation* **7**(1): 1–12.

[64] Möbius, A. [1827]. *Der barycentrische Calcül*, Johann Ambrosius Barth Verlag, Leipzig.

[65] Muller, J.-M., Brisebarre, N., de Dinechin, F., Jeannerod, C.-P., Lefèvre, V., Melquiond, G., Revol, N., Stehlé, D. and Torres, S. [2010]. *Handbook of Floating-Point Arithmetic*, 1 edn, Birkhäuser, Boston.

[66] NVIDIA [2024]. Mathematical functions, *CUDA C++ Programming Guide, Release 12.4*, chapter 16, pp. 373–383.
**URL:** *https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf*

[67] Pachón, R. [2010]. *Algorithms for Polynomial and Rational Approximation*, PhD thesis, University of Oxford.

[68] Phien, H. N. and Dejdumrong, N. [2000]. Efficient algorithms for Bézier curves, *Computer Aided Geometric Design* **17**(3): 247–250.

[69] Quarteroni, A., Sacco, R., Saleri, F. and Gervasio, P. [2014]. *Matematica Numerica*, 4 edn, Springer-Verlag, Italia.

[70] Ramanantoanina, A. and Hormann, K. [2021]. New shape control tools for rational Bézier curve design, *Computer Aided Geometric Design* **88**: 102003.

[71] Rump, S. M. [2019]. Error bounds for computer arithmetics, *26th IEEE Symposium on Computer Arithmetic*, ARITH-26, Kyoto, pp. 1–14.

[72] Salazar Celis, O. [2008]. *Practical Rational Interpolation of Exact and Inexact Data: Theory and Algorithms*, PhD thesis, Department of Computer Science, University of Antwerp.

[73] Salzer, H. E. [1972]. Lagrangian interpolation at the Chebyshev points $x_{n,\nu} \equiv \cos(\nu\pi/n)$, $\nu = 0(1)n$; some unnoted advantages, *The Computer Journal* **15**(2): 156–159.

[74] Schneider, C. and Werner, W. [1986]. Some new aspects of rational interpolation, *Mathematics of Computation* **47**(175): 285–299.

[75] Schumaker, L. L. and Volk, W. [1986]. Efficient evaluation of multivariate polynomials, *Computer Aided Geometric Design* **3**(2): 149–154.

[76] Smith, S. J. [2006]. Lebesgue constants in polynomial interpolation., *Annales Mathematicae et Informaticae* **33**: 109–123.

[77] Teukolsky, S., Flannery, B. P., Vetterling, W. T. and Press, W. H. [2007]. *Numerical Recipes in C: The Art of Scientific Computing*, third edn, Cambridge University Press, New York, chapter 3.4.1, pp. 127–129.

[78] Trefethen, L. N. and Bau, D. [1997]. *Numerical Linear Algebra*, SIAM, Philadelphia.

[79] Wachspress, E. L. [1975]. *A rational finite element basis*, Academic Press, New York.

[80] Wang, G. [1987]. Ball curve of high degree and its geometric properties, *Applied Mathematics: A Journal of Chinese Universities* **2**(1): 126–140.

[81] Warren, J. D. [1993]. An efficient algorithm for evaluating polynomials in the Pòlya basis, *in* H. Hagen, G. E. Farin, H. Noltemeier and R. F. Albrecht (eds), *Geometric Modelling, Dagstuhl, Germany, 1993*, Vol. 10 of *Computing Supplementa*, Springer, pp. 357–361.

[82] Woźny, P. and Chudy, F. [2020]. Linear-time geometric algorithm for evaluating Bézier curves, *Computer-Aided Design* **118**: 102760.

[83] Zhuo, Y., Wu, B., Yao, L., Xiao, G. and Shen, Q. [2022]. Numerical simulation of the temperature in a train brake disc by barycentric rational interpolation collocation method, SSRN preprint, 22 pages.