Algorithm 1048:
A C++ class for robust linear barycentric rational interpolation

Chiara Fuda · Kai Hormann

Abstract

Barycentric rational interpolation is a recent interpolation method with several favourable properties. In this paper, we present the BRI class, which features a new C++ class template that contains all variables and functions related to linear barycentric rational interpolation. While several methods exist to evaluate a barycentric rational interpolant, the class is designed to autonomously select the best method to use on a case-by-case basis, as it takes into account the latest results regarding the efficiency and numerical stability of barycentric rational interpolation [15]. 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 [14].

Citation Info

Journal
ACM Transactions on Mathematical Software

Volume
50(3), September 2024

Article
#23, 17 pages

DOI
10.1145/3681781

1 Introduction

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 [6, 22] and for solving PDEs [26, 27], but also in more applied contexts. For example, the method proposed by Floater and Hormann [13] is used in statistics [2], engineering [32], as well as in the medical [12], aerospace [21, 25], and chemical sciences [23, 24].

Given a set of $n+1$ interpolation *nodes* $X_n=(x_0,x_1,\dots,x_n)$ with $x_0<x_1<\dots<x_n$ and some associated *data* $Y_n=(y_0,y_1,\dots,y_n)$, any *rational interpolant* $r\colon\mathbb{R}\to\mathbb{R}$ that interpolates y_i at x_i can be expressed in its *second barycentric form* as

$$r(x)=\frac{\sum_{i=0}^{n}\frac{w_i}{x-x_i}y_i}{\sum_{i=0}^{n}\frac{w_i}{x-x_i}},\tag{1}$$

for some non-zero *weights* $W_n=(w_0,w_1,\dots,w_n)$ [7]. We focus on *linear* barycentric rational interpolation, meaning that the denominator does not depend on the data Y_n, and we use weights w_i that guarantee the absence of poles in the domain $[x_0,x_n]$. The first to introduce a method that is linear and prevents poles was Berrut [3], who defined the weights as

$$w_i=(-1)^i,\qquad i=0,1,\dots,n.\tag{2}$$

Afterwards, Floater and Hormann [13] proposed a new linear barycentric interpolant by considering a parameter $d\in\{0,1,\dots,n\}$ and the *barycentric weights*

$$w_i=\sum_{k=\max(i-d,0)}^{\min(i,n-d)}(-1)^k\prod_{j=k,\,j\neq i}^{k+d}\frac{1}{x_i-x_j},\qquad i=0,1,\dots,n.\tag{3}$$

Actually, this method is a generalization of Berrut's case since, for $d=0$, the weights in (3) are exactly the ones in (2). Furthermore, the *polynomial* interpolant [8] can be expressed as in (1) with weights

$$w_i=\prod_{j=0,\,j\neq i}^{n}\frac{1}{x_i-x_j},\qquad i=0,1,\dots,n,\tag{4}$$

which can be also obtained from (3) for $d=n$.

The barycentric formula (1) is widely used to evaluate the interpolant r as it can be implemented with an efficient $O(n)$ algorithm (see Algorithm 1) and the w_i can be rescaled by a common factor to prevent

overflow and underflow errors [8]. Moreover, it has been proven [13] that this family of rational interpolants guarantees a fast convergence rate on the order of $O(h^{d+1})$, where h denotes the maximum distance between consecutive nodes. However, there exists another mathematically equivalent formula to evaluate r, namely the *first barycentric form*

$$r(x) = \frac{\sum_{i=0}^{n} \frac{w_i}{x-x_i} f_i}{\sum_{i=0}^{n-d} \lambda_i(x)}, \tag{5}$$

where

$$\lambda_i(x) = \frac{(-1)^i}{(x-x_i)\cdots(x-x_{i+d})}, \qquad i = 0, 1, \ldots, n-d. \tag{6}$$

While this formula is slightly inferior in terms of efficiency, because it requires $O(nd)$ operations to be computed straightforwardly (see Algorithm 2), there exists a more efficient way of computing its denominator in $O(n)$ operations. The resulting algorithm (see Algorithm 3) has the same computational cost as (1), but it is slightly less numerically precise than the standard implementation [15]. Moreover, it may happen that (5) is more numerically stable than (1) for some configurations of nodes. In fact, it is known that the forward stability of both (1) and (5) depends on the condition number $\kappa(x; X_n, Y_n)$ related to the problem and on a second function that is different for both cases [15]. On the one hand, for the second barycentric form, the bound on the relative forward error is affected by the Lebesgue function $\Lambda_n(x; X_n)$, which is known to grow logarithmically in n and exponentially in d for equidistant nodes [10, 11]. On the other hand, the numerical stability of the first form depends on a function $\Gamma_d(x; X_n)$, which is independent of n (see Section 6), but the only proven upper bound so far depends on μ^d, where μ is the *mesh ratio* of the nodes X_n [15]. However, numerical experiments indicate that Γ_d is usually considerably smaller than Λ_n, suggesting that the first barycentric form may provide a more stable solution when the Lebesgue function related to X_n is large. Finally, the choice of the integer d is very important, because, contrary to the approximation order, the numerical stability of the barycentric rational interpolation may be compromised when d grows. For equidistant, quasi-equidistant [18], or conformally shifted nodes [5], this class of rational interpolants exhibits excellent approximation properties due to the favourable behaviour of the Lebesgue function. In particular, for equidistant nodes, Güttel and Klein [16] provide insights on how to choose d with respect to n in order to preserve both the convergence rate and the stability. However, for cases like Chebyshev nodes, where polynomial interpolation is effective, these interpolants may become unstable under certain circumstances [4], so it is then better to use the polynomial interpolant in barycentric rational form by choosing $d = n$.

The goal of this work is to present our *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. One main feature of the BRI class is that it provides the user with an "intelligent" function for evaluating r, which computes the best result in terms of numerical stability, efficiency, or both, by autonomously choosing one of the three evaluation options (Algorithms 1–3). The other main feature is the possibility to choose a "guarded" version of the evaluation algorithms, which avoids potential overflow and underflow errors by adaptively scaling the numerator and the denominator of r. The functionality of our class exceeds those of currently available libraries [1, 9, 28, 30], which simply implement the second barycentric form in (1) without any further considerations.

After a brief overview of all the functions contained in the BRI class (Section 2), we present some preliminary results needed to handle overflow and underflow errors in the guarded version (Section 3). Then we discuss the implementations of the barycentric weights w_i in (3) (Section 4) and of the interpolant r (Section 5). Finally, we describe the algorithms related to the computation of the functions κ, Λ_n, and Γ_d (Section 6).

2 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 [14] and the MPFR `C++` interface [17].

Algorithm 1 Second barycentric form

```
function SECOND(x)
  N := 0
  D := 0
  for i = 0,1,...,n do
    if x = x_i then
      return y_i
    z := w_i/(x - x_i)
    N := N + z y_i
    D := D + z
  return N/D
```

Algorithm 2 First barycentric form (standard version)

```
function FIRST_DEF(x)
  N := 0
  D := 0
  for i = 0,1,...,n do
    if x = x_i then
      return y_i
    N := N + w_i/(x - x_i) y_i
  for i = 0,1,...,n-d do
    λ_i := (-1)^i
    for j = i,i+1,...,i+d do
      λ_i := λ_i/(x - x_j)
    D := D + λ_i
  return N/D
```

Algorithm 3 First barycentric form (efficient version)

```
function FIRST_EFF(x)
  N := 0
  for i = 0,1,...,n do
    if x = x_i then
      return y_i
    N := N + w_i/(x - x_i) y_i
  m := ⌊(n-d)/2⌋
  λ_m := (-1)^m
  for j = m,m+1,...,m+d do
    λ_m := λ_m/(x - x_j)
  D := λ_m
  for i = m,m-1,...,1 do
    λ_{i-1} := -λ_i(x - x_{i+d})/(x - x_{i-1})
    D := D + λ_{i-1}
  for i = m,m-1,...,n-d-1 do
    λ_{i+1} := -λ_i(x - x_i)/(x - x_{i+1+d})
    D := D + λ_{i+1}
  return N/D
```

The BRI class does not have any other 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, 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*

$$x_i = a + (b-a)\frac{i}{n}, \qquad i = 0,1,\dots,n \tag{7}$$

or *Chebyshev nodes*

$$x_i = \frac{a+b}{2} - \frac{b-a}{2}\cos\frac{(2i+1)\pi}{2n+2}, \qquad i = 0,1,\dots,n \tag{8}$$

or *extended Chebyshev nodes*

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

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 of barycentric weights (3), 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 κ, the Lebesgue function Λ_n, and the function Γ_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.

3 Preliminary results on the floating-point number system

We consider the *binary* floating-point number system [31], which is characterized by a *precision* $t \in \mathbb{N}$ and two *exponents* $E_{\min}, E_{\max} \in \mathbb{Z}$. For example, the standard `float` type in `C++` 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 [20] 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 $emin = 1 - emax = E_{\min} - 1$. In such a system, every element of the set $\mathbb{F}$ of *floating point numbers* is 0 or has the form $\pm\mu \times 2^E$, with a *mantissa* (or *significand*) $\mu \in [\mu_{\min}, \mu_{\max}] = [2^{-1}, 1 - 2^{-t}]$ and an *exponent* $E \in \{E_{\min}, E_{\min} + 1, \dots, E_{\max}\}$, and the smallest and largest positive *normal* floating-point numbers are

$$F_{\min} = \mu_{\min} 2^{E_{\min}} \qquad \text{and} \qquad F_{\max} = \mu_{\max} 2^{E_{\max}}. \tag{9}$$

There also exist so-called *subnormal* floating-point numbers, but they have reduced precision and are not relevant in our context as all the rescale operations are performed with respect to the interval $[F_{\min}, F_{\max}]$.

In the following proposition, we present some basic facts about two arbitrary positive floating-point numbers that will be used frequently later.

Proposition 3.1. *Let $a = +\alpha \times 2^A \in \mathbb{F}$ and $b = +\beta \times 2^B \in \mathbb{F}$. Suppose we compute $c = a \circledast b = \gamma \times 2^C \in \mathbb{F}$, where $\circledast \in \{+, -, \cdot, \div\}$ is one of the standard arithmetic operations. Then, except in the case where $c = 0$, the exponent C is guaranteed to be in the range $\{C_{\min}, C_{\min} + 1, \dots, C_{\max}\}$, where*

$$(C_{\min}, C_{\max}) = \begin{cases} (A + B - 1, A + B), & \text{if } c = a \cdot b, \\ (A - B, A - B + 1), & \text{if } c = a \div b, \\ (\max\{A, B\}, \max\{A, B\} + 1), & \text{if } c = a + b, \\ (A - t + 1, A - 1), & \text{if } c = a - b \text{ and } A = B, \\ (\max\{A, B\} - t, \max\{A, B\}), & \text{if } c = a - b \text{ and } |A - B| = 1, \\ (\max\{A, B\} - 1, \max\{A, B\}), & \text{if } c = a - b \text{ and } |A - B| > 1. \end{cases} \tag{10}$$

Proof. The proof is straightforward and follows directly by some consideration on the interval that contains c for each case, recalling the fact that $\gamma \in [\mu_{\min}, \mu_{\max}] = [1/2, 1 - 2^{-t}]$. □

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

$$r_i = \prod_{j=1}^{J} a_{i,j} \qquad \text{and} \qquad s_i = \prod_{k=1}^{K} b_{i,k}, \tag{11}$$

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

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

which can be expressed in floating point encoding as

$$r_i = \rho_i \times 2^{R_i}, \qquad \text{where} \quad \rho_i = \prod_{j=1}^{J} \alpha_{i,j} \quad \text{and} \quad R_i = \sum_{j=1}^{J} A_{i,j}, \tag{12}$$

and

$$s_i = \sigma_i \times 2^{S_i}, \qquad \text{where} \quad \sigma_i = \prod_{k=1}^{K} \beta_{i,k} \quad \text{and} \quad S_i = \sum_{k=1}^{K} B_{i,k}, \tag{13}$$

suppose we want to compute the quantity

$$f = \sum_{i=0}^{M} \frac{r_i}{s_i}. \tag{14}$$

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 3.2. *Let $M, J, K \in \mathbb{N}$ and consider r_i and s_i as in (11), ρ_i and R_i as in (12), σ_i and S_i as in (13) for $i = 0, 1, \ldots, M$, and f as in (14). Suppose that all $p_i = r_i/s_i$ have the same sign and define*

$$L = \min_{0 \le i \le M} (R_i - S_i) - J + 1 \qquad \text{and} \qquad U = \max_{0 \le i \le M} (R_i - S_i) + K + M \tag{15}$$

and

$$E = \left\lceil \frac{E_{\max} + E_{\min} - L - U}{2} \right\rceil. \tag{16}$$

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 (16) that

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

which, together with (15) and the hypothesis $U - L < E_{\max} - E_{\min}$, gives

$$E_{\min} \le E_{\min} + J - 1 \le R_i - S_i + E \le E_{\max} - K - M < E_{\max}, \tag{17}$$

meaning that the computation of p_i in line 4 is always safe. By Proposition 3.1, 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 (17) guarantees that $p_i \in \mathbb{F}$. Finally, Proposition 3.1 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 (17), we can thus be sure that also $\hat{f} \in \mathbb{F}$. □

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

procedure WEIGHTS
  $W_n = (w_0, w_1, \ldots, w_{n-d}, w_{n-d+1}, \ldots, w_n) := (1, 1, \ldots, 1, 0, \ldots, 0)$
  for $l = d-1, d-2, \ldots, 0$ **do**
    $w_{n-l} := w_{n-l-1}/(x_n - x_{n-l-1})$
    for $i = n-l-1, n-l-2, \ldots, 1$ **do**
      $w_i := w_{i-1}/(x_{i+l} - x_{i-1}) + w_i/(x_{i+l+1} - x_i)$
    $w_0 := w_0/(x_{l+1} - x_0)$
  $W_n := (-1)^d (w_0, -w_1, \ldots, (-1)^n w_n)$

Corollary 3.1. *Let $M, J, K \in \mathbb{N}$ and consider r_i and s_i as in (11), ρ_i and R_i as in (12), σ_i and S_i as in (13) for $i = 0, 1, \ldots, M$, and f as in (14). 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 $R_{\min} \le R_i \le R_{\max}$ and $S_{\min} \le S_i \le S_{\max}$ for $i = 0, 1, \ldots, M$, and define*

$$L = R_{\min} - S_{\max} - J + 1 \qquad \textit{and} \qquad U = R_{\max} - S_{\min} + K + M \tag{18}$$

and

$$E = \left\lceil \frac{E_{\max} + E_{\min} - L - U}{2} \right\rceil. \tag{19}$$

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 $L \le \min_{0 \le i \le M}(R_i - S_i) - J + 1$ and $U \ge \max_{0 \le i \le M}(R_i - S_i) + K + M$, the statement follows with the same arguments as in the proof of Proposition 3.2. □

4 Barycentric weights

As for the barycentric weights in (3), the simplest method to implement them is to follow their definition, thus obtaining an algorithm that computes all w_i in $O(nd^2)$ operations. However, Hormann and Schaefer [19] proposed a *pyramid algorithm* that requires only $O(nd)$ operations and has the same precision [15]. This algorithm starts with the values

$$v_i^d = 1, \qquad i = 0, 1, \ldots, n-d \tag{20}$$

and iteratively defines

$$v_i^l = \frac{v_{i-1}^{l+1}}{x_{i+l} - x_{i-1}} + \frac{v_i^{l+1}}{x_{i+l+1} - x_i}, \qquad i = 0, 1, \ldots, n-l, \tag{21}$$

for $l = d-1, d-2, \ldots, 0$, tacitly assuming that $v_i^l = 0$ for $i < 0$ and $i > n-l$. The barycentric weights are then given as

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

To save memory, the BRI class uses the vector of weights also to store the intermediate values v_i^l in w_i, which requires the assignment in (21) to be performed in reverse order (see Algorithm 5).

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 (8) (that is, $n = 9$) over $[0, 10^{-12}]$ and $d = 3$, then

$$w_0 = \frac{-1}{(x_3 - x_0)(x_2 - x_0)(x_1 - x_0)} \approx -5.5257 \times 10^{38}, \tag{22}$$

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 $w_0 = -$`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 $w_i = (-1)^{i+1}$`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 $f(x) = \log(1.2-x)/(x^2+2)$ [16]. 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 $w_i = (-1)^{i+1}$`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 [29] 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 (21) for $i = 0,1,\dots,n-l$ 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 (1).

Proposition 4.1. *Let $L_d = U_d = 0$ and*

$$L_l = L_{l+1} - H_{\max} - E_l - 1 \qquad \text{and} \qquad U_l = U_{l+1} - H_{\min} - E_l + 2, \qquad l = d-1, d-2, \dots, 0, \tag{23}$$

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

$$B_{\min}^l = \frac{B_{\min}^{l+1}}{(l+1)h_{\max}} \qquad \text{and} \qquad B_{\max}^l = \frac{2B_{\max}^{l+1}}{(l+1)h_{\min}} \tag{24}$$

and noticing that

$$0 < (k-j)h_{\min} \le x_k - x_j \le (k-j)h_{\max}, \tag{25}$$

for any $j < k$, it follows from (21) that

$$B_{\min}^l \le \frac{v_{i-1}^{l+1}}{(l+1)h_{\max}} + \frac{v_i^{l+1}}{(l+1)h_{\max}} \le v_i^l \le \frac{v_{i-1}^{l+1}}{(l+1)h_{\min}} + \frac{v_i^{l+1}}{(l+1)h_{\min}} \le B_{\max}^l \tag{26}$$

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 (24) 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

$$2^{E_l} \le l+1 \le 2^{E_l+1}, \tag{27}$$

it then follows from Proposition 3.1 and (24) 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 (23). □

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 4.1 that $L_{l+1} \le V_i \le U_{l+1}$ and, from (25) and (27), that $H_{\min} + E_l \le X_i \le H_{\max} + E_l + 1$. Moreover, we can observe that the expression for

Algorithm 6 Guarded version of the WEIGHTS procedure

procedure WEIGHTS
  initialize W_n as in line 2 of Algo. 5 and set $L := 0$ and $U := 0$
  for $l = d-1, d-2, \dots, 0$ **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
      $L := \operatorname{expo}(\min_{0 \le i \le n-l} w_i) - 1$ and $U := \operatorname{expo}(\max_{0 \le i \le n-l} w_i)$ {expo(x) returns e for $x = \pm m \times 2^e$}
      $L := L - H_{\max} - E_l - 1$ 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
          $L := \operatorname{expo}(\min_{0 \le i \le n-l} w_i) - 1$ and $U := \operatorname{expo}(\max_{0 \le i \le n-l} w_i)$
  add the alternating signs to W_n as in line 8 of Algo. 5

the values v_i^l in (21) fits into the more general formula (14) for $M = J = K = 1$. This means that L_l and U_l in (23) are exactly L and U in (18), 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 3.1, 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.

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 $l = d-1, d-2, \dots, 0$. Therefore, if we first compute L_l and U_l as in (23) 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 $\hat{l} \in \{0, 1, \dots, d-1\}$, 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

$$L_{\hat{l}} = \min_{0 \le i \le n-l} V_i - 1 \qquad \text{and} \qquad U_{\hat{l}} = \max_{0 \le i \le n-l} V_i. \tag{28}$$

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 $i = 0, 1, \dots, n-\hat{l}$ without overflow nor underflow errors, then we define $L_{\hat{l}}$ and $U_{\hat{l}}$ as in (28), 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

$$C_w = \sum_{l=0}^{d-1} C_l. \tag{29}$$

For example, considering the setting of Example 4.1, we can solve the overflow problem with Algorithm 6, which returns the weights rescaled by the constant $C_w = 2^{-127}$.

5 Evaluation of the barycentric rational interpolant

As already mentioned in Section 2, 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

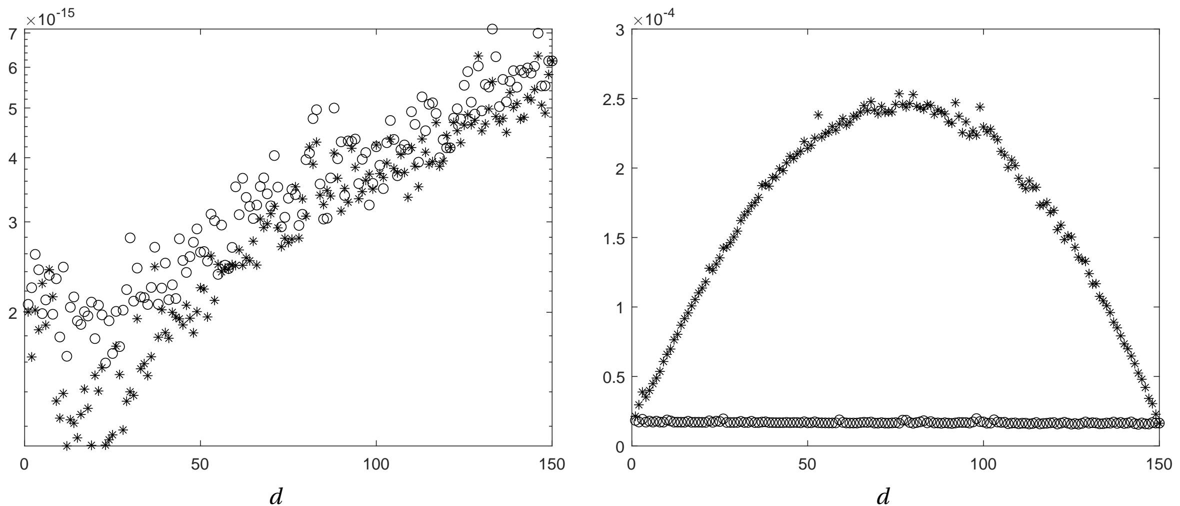

Figure 1: Maximal relative forward error of the first barycentric form for $n+1$ equidistant nodes, $n = 150$, and $d \in \{1,2,\dots,n\}$ at 1000 random evaluation points with data sampled from the n-th Lagrange basis polynomial on a logarithmic scale (left) and overall running time in seconds (right)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

`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 [15], 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 (1) depends on the function Λ_n, which grows with n, whereas the stability of (5) depends on the function Γ_d, whose upper bound is independent of n. However, numerical experiments show that the estimated upper bound on Γ_d seems to be always much larger than Γ_d itself. For this reason, if $\Lambda_n(x) > 10^2$, then the result is computed with Algorithm 2. If it happens that also $\Gamma_d(x) > 10^2$, 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 λ_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 1 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 $i = 0,\dots,n-1$ 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 $d \in \{1,2,\dots,n\}$. 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 neighbourhood 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 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 (1) and (5) involve both subtractions and additions that may lead to cancellation errors. Moreover, from Proposition 3.1, 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 [15], 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

Algorithm 7 EVAL function

```
function EVAL(x)
  if the stability flag is turned on then
    if κ(x) > 10^3 then
      the user gets a warning
    if Λ_n(x) > 10^2 then
      if Γ_d(x) > 10^2 then
        the user gets a warning
      if the efficiency flag is turned on then
        return r, computed with Algo. 3
      else
        return r, computed with Algo. 2
    else
      return r, computed with Algo. 1
  else
    return r, computed with Algo. 1
```

and one for the negative terms, which, in case the condition of Proposition 3.2 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 (1) and (5) using Proposition 3.2 (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 $w_i = \omega_i \times 2^{W_i}$, 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

$$
C_1 = \left\lceil \frac{E_{\max} + E_{\min} - L_1 - U_1}{2} \right\rceil, \qquad C_2 = \left\lceil \frac{E_{\max} + E_{\min} - L_2 - U_2}{2} \right\rceil, \qquad C_3 = \left\lceil \frac{E_{\max} + E_{\min} - L_3 - U_3}{2} \right\rceil. \tag{30}
$$

After that, denoting the common numerator of (1) and (5), the denominator of (1), and the denominator of (5) 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 (30) 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

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

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 (29), 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.

Algorithm 8 Guarded evaluation of the second barycentric form

function SECOND(x)
  $L_1 := \min_{0\le i\le n} \operatorname{expo}(w_i y_i/(x-x_i))$
  $U_1 := \max_{0\le i\le n} \operatorname{expo}(w_i y_i/(x-x_i))+1$
  if $U_1 - L_1 < E_{\max} - E_{\min}$ **then**
    $C_1 := \lfloor (E_{\max}+E_{\min}-L_1-U_1)/2 \rceil$
    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
      $C_1 := 0$
  $L_2 := \min_{0\le i\le n} \operatorname{expo}(w_i/(x-x_i))+1$
  $U_2 := \max_{0\le i\le n} \operatorname{expo}(w_i/(x-x_i))+1$
  if $U_2 - L_2 < E_{\max} - E_{\min}$ **then**
    $C_2 := \lfloor (E_{\max}+E_{\min}-L_2-U_2)/2 \rceil$
    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)
  $L_1 := \min_{0\le i\le n} \operatorname{expo}(w_i y_i/(x-x_i))$
  $U_1 := \max_{0\le i\le n} \operatorname{expo}(w_i y_i/(x-x_i))+1$
  if $U_1 - L_1 < E_{\max} - E_{\min}$ **then**
    $C_1 := \lfloor (E_{\max}+E_{\min}-L_1-U_1)/2 \rceil$
    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
      $C_1 := 0$
  $L_3 := \min_{0\le i\le n-d} \operatorname{expo}([(x-x_i)\dots(x-x_{i+d})]^{-1})+1$
  $U_3 := \max_{0\le i\le n-d} \operatorname{expo}([(x-x_i)\dots(x-x_{i+d})]^{-1})+d+1$
  if $U_3 - L_3 < E_{\max} - E_{\min}$ **then**
    $C_3 := \lfloor (E_{\max}+E_{\min}-L_3-U_3)/2 \rceil$
    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 $r = 2^{C_3-C_1-C_w}\frac{\hat{N}}{\hat{D}_f}$

Example 5.1. 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

$$f(x) = \frac{3}{4}e^{-\frac{(9x-2)^2}{4}} + \frac{3}{4}e^{-\frac{(9x+1)^2}{49}} + \frac{1}{2}e^{-\frac{(9x-7)^2}{4}} + \frac{1}{5}e^{-(9x-4)^2}, \tag{31}$$

$x = 10^{-12}/2$, and all variables in input and output are set as `float`. In this case, both Algorithms 1 and 2 return `-nan`, while Algorithms 8 and 9 give the same result in single precision, with 6 digits equal to the exact value $r(x) = 1.01076\dots$. More precisely, they compute

$$N(x) = 2^{44}5.13436, \qquad D_s(x) = 2^{43}10.1594, \qquad \text{and} \qquad D_f(x) = 2^{125+C_w}40.6375,$$

where $C_w = -84$ in this setting.

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.

5.1 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, the floating-point numbers are equally spaced in each interval $[2^E, 2^{E+1}]$ and at a distance of $h_E = 2^{E-t}$. Hence, if we want to evaluate r very close to a node $x_j \in [2^E, 2^{E+1}]$ 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

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

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 5.2. 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 $i=0,1,\dots,n$ and $f(x)=x$. Suppose 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`. Since the closest double floating-point number to $x_0=1$ is $1+\epsilon$, where $\epsilon=2.2204\times10^{-16}$, using any of the three algorithms discussed in Section 5, we get $r(x)=y_0=1$, because x is rounded to $x_0=1$ in double precision. Instead, with the new strategy we obtain $u=r(h)=9.9999999999999949376\times10^{-21}$, 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).

6 Stability-related functions

As mentioned in Section 5, the BRI class uses the functions Λ_n, Γ_d, and κ 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 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 Λ_n, Γ_d, and κ can all be written in the common form

$$g(x)=\sum_{i=0}^{n}\left|\frac{b_i(x)}{\sum_{i=0}^{n}b_i(x)}\right|, \tag{32}$$

where $b_i(x)=w_i/(x-x_i)$ for Λ_n, $b_i(x)=(-1)^i/\prod_{j=i}^{i+d}(x-x_j)$ for Γ_d, and $b_i(x)=w_iy_i/(x-x_i)$ for κ. 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 (32) is greater than or equal to 1. Furthermore, in the specific case of Λ_n, Γ_d, and κ, the minimum with value 1 is assumed exactly at the nodes x_i and, from numerical experiments (see Figure 2), 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 Λ_n', Γ_d', and κ' 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 of $t_1,t_2,\dots\in\mathbb{R}$ by applying the iterative formula

$$t_{j+1}=t_j-\frac{g(t_j)}{g'(t_j)}. \tag{33}$$

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 2 shows that Λ_n, Γ_d, and κ 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 (32), it is easy to see that the method can be implemented straightforwardly by computing at each iteration j first

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

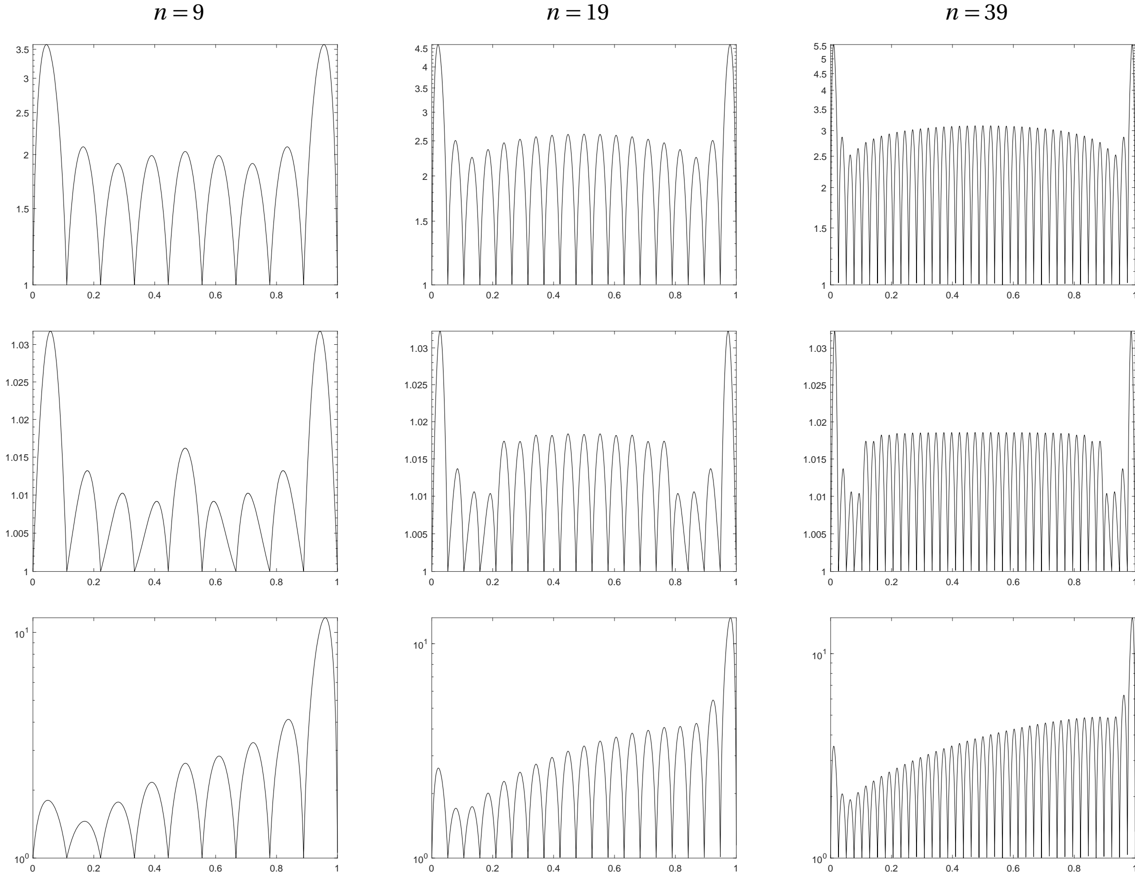

Figure 2: Plots of Λ_n (top), Γ_d (middle), and κ (bottom) for $n+1$ equidistant nodes and $x \in [0,1]$, all on a logarithmic scale, for $d=3$ and three choices of n (left, middle, right). For the computation of the condition number κ, the data are sampled from Runge's function $f(x) = 1/(1+25x^2)$.

and

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

and then using (33) to set $t_{j+1} = t_j - g'(t_j)/g''(t_j)$, until the error $|t_{j+1} - t_j| < 10\epsilon$, where ϵ is the machine epsilon.

Acknowledgements

This work was supported by the Swiss National Science Foundation (SNSF) under project No. 188577.

References

[1] N. Agrawal et al. BOOST 1.81.0 Library Documentation – math toolkit 3.3.0 – Interpolation – Barycentric Rational Interpolation, Dec. 2022. [Online; accessed 23-February-2023].

[2] R. D. Baker and I. G. McHale. Time varying ratings in association football: the all-time greatest team is... *Journal of the Royal Statistical Society: Series A*, 178(2):481–492, Feb. 2015.

[3] J.-P. Berrut. Rational functions for guaranteed and experimentally well-conditioned global interpolation. *Computers & Mathematics with Applications*, 15(1):1–16, 1988.

[4] J.-P. Berrut. Linear barycentric rational interpolation with guaranteed degree of exactness. In G. E. Fasshauer and L. L. Schumaker, editors, *Approximation Theory XV: San Antonio 2016*, pages 1–20. Springer International Publishing, Cham, 2017.

[5] J.-P. Berrut. The conditioning of a linear barycentric rational interpolant. In C. Beattie, P. Benner, M. Embree, S. Gugercin, and S. Lefteriu, editors, *Realization and Model Reduction of Dynamical Systems*, pages 23–37. Springer, Cham, 2022.

[6] J.-P. Berrut, S. A. Hosseini, and G. Klein. The linear barycentric rational quadrature method for Volterra integral equations. *SIAM Journal on Scientific Computing*, 36(1):A105–A123, 2014.

[7] J.-P. Berrut and H. D. Mittelmann. Matrices for the direct determination of the barycentric weights of rational interpolation. *Journal of Computational and Applied Mathematics*, 78(2):355–370, 1997.

[8] J.-P. Berrut and L. N. Trefethen. Barycentric Lagrange interpolation. *SIAM Review*, 46(3):501–517, Sept. 2004.

[9] S. Bochkanov. ALGLIB 3.20.0 User Guide – Interpolation and fitting – Rational interpolation, Dec. 2022. [Online; accessed 23-February-2023].

[10] L. Bos, S. De Marchi, and K. Hormann. On the Lebesgue constant of Berrut's rational interpolant at equidistant nodes. *Journal of Computational and Applied Mathematics*, 236(4):504–510, Sept. 2011. [PDF]

[11] L. Bos, S. De Marchi, K. Hormann, and G. Klein. On the Lebesgue constant of barycentric rational interpolation at equidistant nodes. *Numerische Mathematik*, 121(3):461–471, July 2012. [PDF]

[12] A. J. Ellingsrud, N. Boullé, P. E. Farrell, and M. E. Rognes. Accurate numerical simulation of electrodiffusion and water movement in brain tissue. *Mathematical Medicine and Biology: A Journal of the IMA*, 38(4):516–551, Dec. 2021.

[13] M. S. Floater and K. Hormann. Barycentric rational interpolation with no poles and high rates of approximation. *Numerische Mathematik*, 107(2):315–331, Aug. 2007. [PDF]

[14] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2), June 2007. — The MPFR library can be found at https://www.mpfr.org.

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

[16] S. Güttel and G. Klein. Convergence of linear barycentric rational interpolation for analytic functions. *SIAM Journal on Numerical Analysis*, 50(5):2560–2580, 2012.

[17] P. Holoborodko. Multiple precision floating point arithmetic library for C++, Aug. 2008. [Online; accessed 17-June-2024].

[18] K. Hormann, G. Klein, and S. De Marchi. Barycentric rational interpolation at quasi-equidistant nodes. *Dolomites Research Notes on Approximation*, 5:1–6, 2012. [PDF]

[19] K. Hormann and S. Schaefer. Pyramid algorithms for barycentric rational interpolation. *Computer Aided Geometric Design*, 42:1–6, Feb. 2016. [PDF]

[20] IEEE Computer Society, New York. *IEEE Standard for Floating-Point Arithmetic*, July 2019. IEEE Std 754-2019 (Revision of IEEE Std 754-2008).

[21] W. Jianying, L. Haizhao, Q. Zheng, and Y. Dong. Mapped Chebyshev pseudospectral methods for optimal trajectory planning of differentially flat hypersonic vehicle systems. *Aerospace Science and Technology*, 89:420–430, June 2019.

[22] G. Klein and J.-P. Berrut. Linear barycentric rational quadrature. *BIT Numerical Mathematics*, 52(2):407–424, June 2012.

[23] T.-S. Lee, B. K. Radak, M. Huang, K.-Y. Wong, and D. M. York. Roadmaps through free energy landscapes calculated using the multidimensional vFEP approach. *Journal of Chemical Theory and Computation*, 10(1):24–34, Jan. 2014.

[24] T.-S. Lee, B. K. Radak, A. Pabis, and D. M. York. A new maximum likelihood approach for free energy profile construction from molecular simulations. *Journal of Chemical Theory and Computation*, 9(1):153–164, Jan. 2013.

[25] J. Leffell, S. Murman, and T. Pulliam. An extension of the time-spectral method to overset solvers. In *51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, Grapevine, TX, Jan. 2013. American Institute of Aeronautics and Astronautics, Reston, VA.

[26] J. Li and Y. Cheng. Linear barycentric rational collocation method for solving heat conduction equation. *Numerical Methods for Partial Differential Equations*, 37(1):533–545, Jan. 2021.

[27] W.-H. Luo, T.-Z. Huang, X.-M. Gu, and Y. Liu. Barycentric rational collocation methods for a class of nonlinear parabolic partial differential equations. *Applied Mathematics Letters*, 68:13–19, June 2017.

[28] D. MacMillen. Baryrational, Jan. 2021. [Online; accessed 23-February-2023].

[29] R. Pachón. *Algorithms for Polynomial and Rational Approximation*. PhD thesis, University of Oxford, 2010.

[30] S. Teukolsky, B. P. Flannery, W. T. Vetterling, and W. H. Press. *Numerical Recipes in C: The Art of Scientific Computing*, chapter 3.4.1, pages 127–129. Cambridge University Press, New York, third edition, 2007. ISBN 978-0-521-88068-8.

[31] L. N. Trefethen and D. Bau. *Numerical Linear Algebra*. SIAM, Philadelphia, 1997. ISBN 978-0-89871-361-9.

[32] Y. Zhuo, B. Wu, L. Yao, G. Xiao, and Q. Shen. Numerical simulation of the temperature in a train brake disc by barycentric rational interpolation collocation method. SSRN preprint, 22 pages, July 2022.

User Manual for the BRI class in C++

This user guide explains how to use the functions of the BRI C++ class template for barycentric rational interpolation.

1 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.

2 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.

2.1 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

2.2 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

2.3 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

2.4 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 jth 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.

2.5 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

2.6 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 $x_j + h$

2.7 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 $\max_{x\in[x_0,x_n]} \Gamma_d(x)$

3 Examples

In this section, we provide several examples that showcase how to use the BRI class in practice.

3.1 Computation of the weights in guarded mode

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 compute the weights in guarded mode.

```
#include "BRI.h"

using namespace std;

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

3.2 Evaluation of the interpolant

We consider $n = 9$, $n+1$ Chebyshev interpolation nodes in $[0, 10^{-12}]$ and $d = 2$. We then define $y_i = f(x_i)$ for $i = 0, 1, \ldots, n$, where

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

and we output $r(x)$ using both the first and the second barycentric formula, both in non-guarded and guarded mode, with $x = 10^{-12}/2$. We 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`.

```
#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(){
  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 = 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.precision(10);
  cout << "FIRST FORM - DEF: r(x) = " << f << 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 = 40.63759232 and C3 = -125
```

3.3 Evaluation of the interpolant close to a node

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 $i = 0,1,\dots,n$ and $f(x) = x$. We 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`.

```
#include "BRI.h"

using namespace std;

int main(){
  int n = 9;
  int d = 2;

  double a = 1;
  double b = 2;
  vector<double> xn = uniform(a,b,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.precision(20);
  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
```

3.4 Evaluation of the stability-related functions

We consider $n = 9$, $n+1$ equidistant interpolation nodes $x_i \in [0,1]$ with associated data $y_i = f(x_i)$ for $i = 0,\dots,n$ 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 Λ_n, Γ_d, and κ.

```
#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
```