Abstract

In computer aided geometric design a polynomial is usually represented in Bernstein form. This paper presents a family of compensated algorithms to accurately evaluate a polynomial in Bernstein form with floating point coefficients. The principle is to apply error-free transformations to improve the traditional de Casteljau algorithm. At each stage of computation, round-off error is passed on to first order errors, then to second order errors, and so on. After the computation has been "filtered" $(K-1)$ times via this process, the resulting output is as accurate as the de Casteljau algorithm performed in K times the working precision. Forward error analysis and numerical experiments illustrate the accuracy of this family of algorithms.

Contents

1 Introduction

In computer aided geometric design, polynomials are usually expressed in Bernstein form. Polynomials in this form are usually evaluated by the de Casteljau algorithm. This algorithm has a round-off error bound which grows only linearly with degree, even though the number of arithmetic operations grows quadratically. The Bernstein basis is optimally suited ([FR87], [DP15], [MP05]) for polynomial evaluation; it is typically more accurate than the monomial basis, for example in Figure 1 evaluation via Horner's method produces a jagged curve for points near a triple root, but the de Casteljau algorithm produces a smooth curve. Nevertheless the de Casteljau algorithm returns results arbitrarily less accurate than the working precision $\mathbf{u}$ when evaluating $p(s)$ is ill-conditioned. The relative accuracy of the computed evaluation with the de Casteljau algorithm (`DeCasteljau`) satisfies ([MP99]) the following a priori bound:

$$\frac{|p(s) - \texttt{DeCasteljau}(p, s)|}{|p(s)|} \leq \operatorname{cond}(p, s) \times \mathcal{O}(\mathbf{u}). \tag{1}$$

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

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

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

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

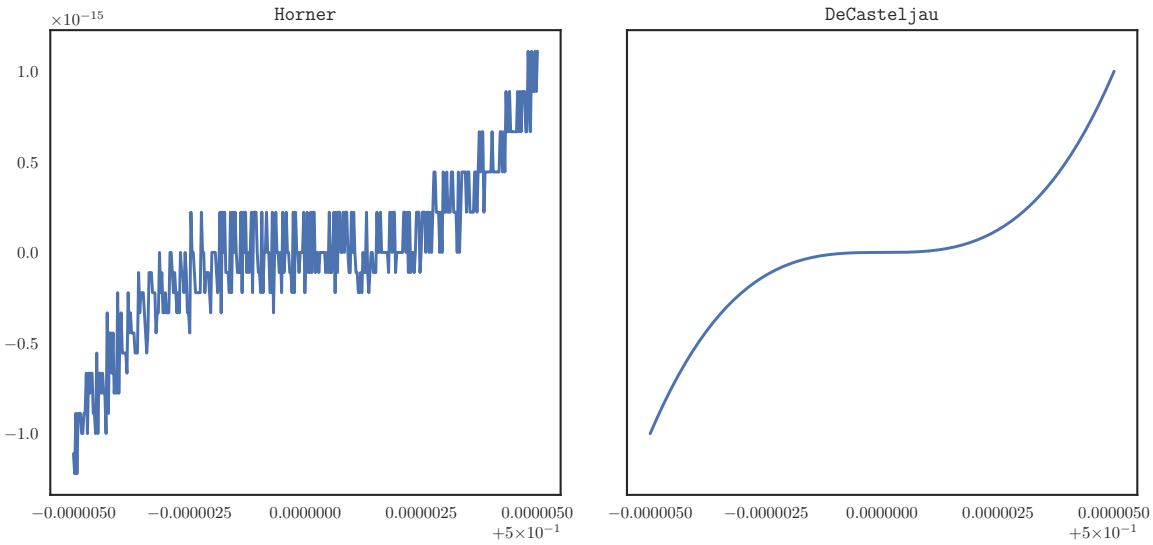

Figure 1: Comparing Horner's method to the de Casteljau method for evaluating $p(s) = (2s - 1)^3$ in the neighborhood of its multiple root $1/2$.

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

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

As in (1), the accuracy of the compensated result (2) may be arbitrarily bad for ill-conditioned polynomial evaluations. For example, as the condition number grows in Figure 2, some points have relative error exactly equal to 1; this indicates that $\texttt{CompDeCasteljau}(p, s) = 0$, which is a complete failure to evaluate the order of magnitude of $p(s)$. For root-finding problems $\texttt{CompDeCasteljau}(p, s) = 0$ when $p(s) \neq 0$ can cause premature convergence and incorrect results. We describe how to defer rounding into progressively smaller error groups and improve the accuracy of the computed result by a factor of $\mathbf{u}$ for every error group added. So we derive `CompDeCasteljauK`, a K-fold compensated de Casteljau algorithm that satisfies the following a prior bound for any arbitrary integer K:

$$\frac{|p(s) - \texttt{CompDeCasteljauK}(p, s)|}{|p(s)|} \leq \mathbf{u} + \operatorname{cond}(p, s) \times \mathcal{O}\left(\mathbf{u}^K\right). \tag{3}$$

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

The paper is organized as follows. Section 2 introduces some basic notation and results about floating point arithmetic, error-free transformations and the de Casteljau algorithm. In Section 3, the compensated algorithm for polynomial evaluation from [JLCS10] is reviewed and notation is established for the expansion. In Section 4, the K compensated algorithm is provided and a forward error analysis is performed. Finally, in Section 5 we give numerical tests to illustrate the practical efficiency of our algorithms.

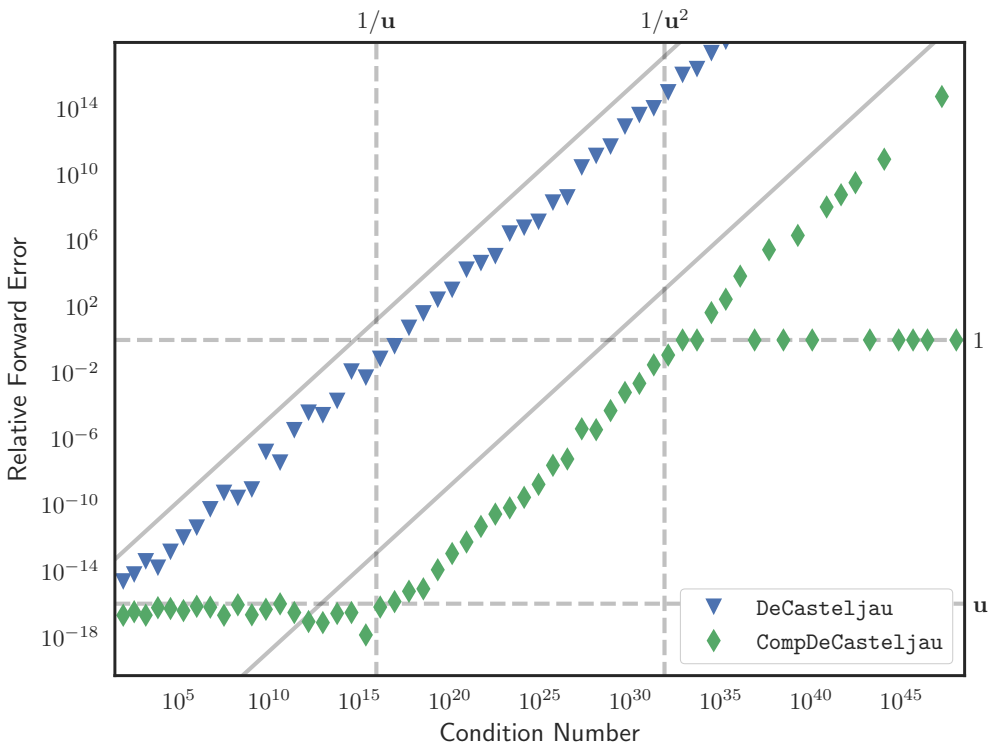

Figure 2: Evaluation of $p(s) = (s-1)(s-3/4)^7$ represented in Bernstein form.

2 Basic notation and results

Throughout this paper we assume that the computation in floating point arithmetic obeys the model

$$a \star b = \mathrm{fl}(a \circ b) = (a \circ b)(1+\varepsilon_1) = (a \circ b)/(1+\varepsilon_2) \tag{4}$$

where $\star \in \{\oplus, \ominus, \otimes, \oslash\}$, $\circ \in \{+, -, \times, \div\}$ and $|\varepsilon_1|, |\varepsilon_2| \leq \mathbf{u}$. The symbol $\mathbf{u}$ is the unit round-off and $\star$ represents the floating point computation, e.g. $a \oplus b = \mathrm{fl}(a+b)$. (For IEEE-754 floating point double precision, $\mathbf{u} = 2^{-53}$.) We also assume that the computed result of $\alpha \in \mathbf{R}$ in floating point arithmetic is denoted by $\widehat{\alpha}$ or $\mathrm{fl}(\alpha)$ and $\mathbf{F}$ denotes the set of all floating point numbers (see [Hig02] for more details). Following [Hig02], we will use the following classic properties in error analysis.

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

Now let us introduce some results concerning error-free transformation (EFT). For a pair floating point numbers $a, b \in \mathbf{F}$, there exists a floating point number y satisfying $a \circ b = x + y$ where $x = \mathrm{fl}(a \circ b)$ and $\circ \in \{+, -, \times\}$. The transformation $(a, b) \longrightarrow (x, y)$ is regarded as an error-free transformation. The error-free transformation algorithms of the sum and product of two floating point numbers used later in this paper are the `TwoSum` algorithm by Knuth [Knu97] and `TwoProd` algorithm by Dekker [Dek71], respectively. The following theorem exhibits the important properties of `TwoSum` and `TwoProd` algorithms (see [ORO05]).

Theorem 1 ([ORO05])**.** For $a, b \in \mathbf{F}$ and $P, \pi, S, \sigma \in \mathbf{F}$, `TwoSum` and `TwoProd` satisfy

- $[S, \sigma] = \texttt{TwoSum}(a, b)$, $x = \text{fl}(a + b)$, $S + \sigma = a + b$, $\sigma \leq \mathbf{u}|S|$, $\sigma \leq \mathbf{u}|a + b|$
- $[P, \pi] = \texttt{TwoProd}(a, b)$, $P = \text{fl}(a \times b)$, $P + \pi = a \times b$, $\pi \leq \mathbf{u}|P|$, $\pi \leq \mathbf{u}|a \times b|$.

The letters σ and π are used to indicate that the errors came from sum and product, respectively. See the appendix in Section 6 for implementation details.

Next, we recall the de Casteljau algorithm:

Algorithm 1 *de Casteljau algorithm for polynomial evaluation.*

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

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

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

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

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

$$|p(s) - \texttt{DeCasteljau}(p, s)| \leq \gamma_{2n} \sum_{j=0}^{n} |b_j| B_{j,n}(s). \tag{5}$$

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

$$\text{cond}(p, s) = \frac{\widetilde{p}(s)}{|p(s)|} = \frac{\sum_j |b_j| B_{j,n}(s)}{|p(s)|}, \tag{6}$$

where $B_{j,n} \geq 0$.

3 Compensated de Casteljau

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

$$[\widehat{r}, \rho] = \texttt{TwoSum}(1, -s) \tag{7}$$

$$[P_1, \pi_1] = \texttt{TwoProd}\left(\widehat{r}, \widehat{b}_j^{(k+1)}\right) \tag{8}$$

$$[P_2, \pi_2] = \texttt{TwoProd}\left(s, \widehat{b}_{j+1}^{(k+1)}\right) \tag{9}$$

$$\left[\widehat{b}_j^{(k)}, \sigma_3\right] = \texttt{TwoSum}(P_1, P_2) \tag{10}$$

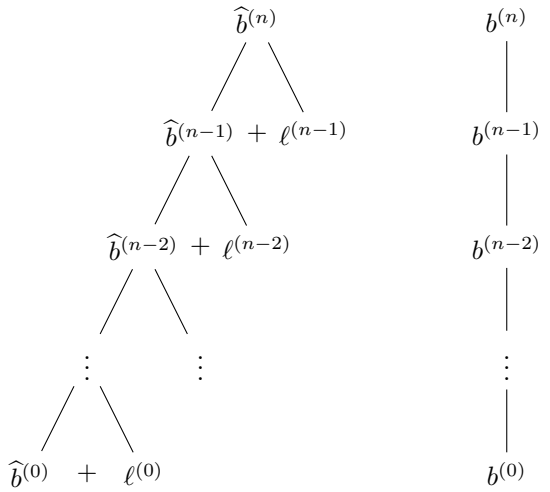

Figure 3: Local round-off errors

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

$$\ell_{1,j}^{(k)} = \pi_1 + \pi_2 + \sigma_3 + \rho \cdot \widehat{b}_j^{(k+1)} \tag{11}$$

$$(1-s) \cdot \widehat{b}_j^{(k+1)} + s \cdot \widehat{b}_{j+1}^{(k+1)} = \widehat{b}_j^{(k)} + \ell_{1,j}^{(k)}. \tag{12}$$

By defining the global errors at each step

$$\partial b_j^{(k)} = b_j^{(k)} - \widehat{b}_j^{(k)} \tag{13}$$

we can see (Figure 3) that the local errors accumulate in $\partial b^{(k)}$:

$$\partial b_j^{(k)} = (1-s) \cdot \partial b_j^{(k+1)} + s \cdot \partial b_{j+1}^{(k+1)} + \ell_{1,j}^{(k)}. \tag{14}$$

When computed in exact arithmetic, we would have

$$p(s) = \widehat{b}_0^{(0)} + \partial b_0^{(0)} \tag{15}$$

and by using (14), we can continue to compute approximations of $\partial b_j^{(k)}$. The idea behind the compensated de Casteljau algorithm is to compute both the local error and the updates of the global error with floating point operations:

Algorithm 2 *Compensated de Casteljau algorithm for polynomial evaluation.*

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

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

  for $k = n-1, \ldots, 0$ **do**
    for $j = 0, \ldots, k$ **do**
      $[P_1, \pi_1] =$ `TwoProd` $\left(\widehat{r}, \widehat{b}_j^{(k+1)}\right)$

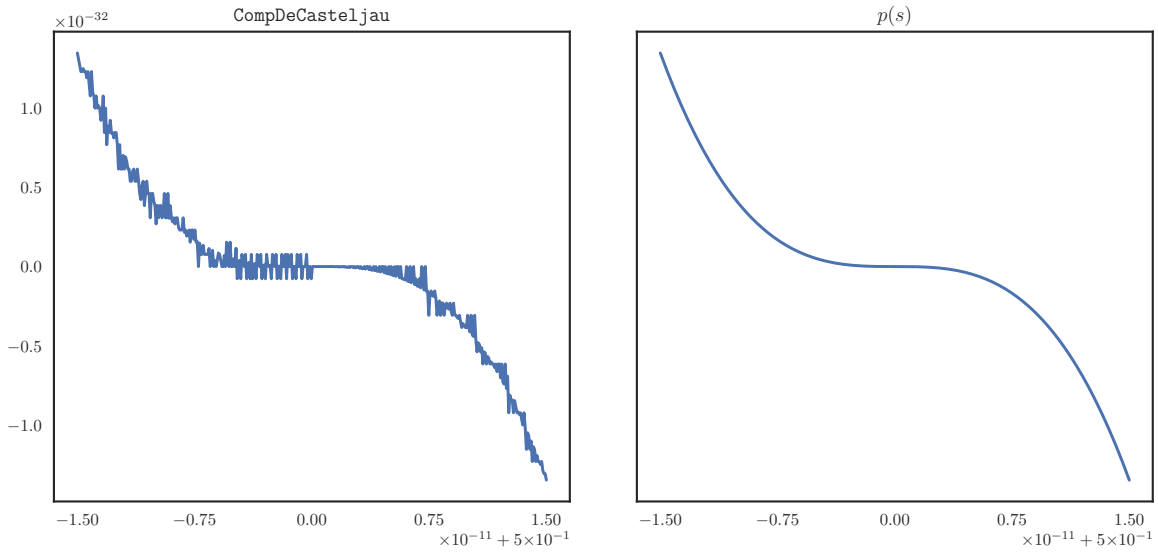

Figure 4: The compensated de Casteljau method starts to lose accuracy for $p(s) = (2s-1)^3(s-1)$ in the neighborhood of its multiple root $1/2$.

$$
\begin{aligned}
&[P_2, \pi_2] = \texttt{TwoProd}\left(s, \widehat{b}_{j+1}^{(k+1)}\right) \\
&\left[\widehat{b}_j^{(k)}, \sigma_3\right] = \texttt{TwoSum}(P_1, P_2) \\
&\widehat{\ell}_{1,j}^{(k)} = \pi_1 \oplus \pi_2 \oplus \sigma_3 \oplus \left(\rho \otimes \widehat{b}_j^{(k+1)}\right) \\
&\widehat{\partial b}_j^{(k)} = \left(\widehat{r} \otimes \widehat{\partial b}_j^{(k+1)}\right) \oplus \left(s \otimes \widehat{\partial b}_{j+1}^{(k+1)}\right) \oplus \widehat{\ell}_{1,j}^{(k)}
\end{aligned}
$$

end for
end for

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

When comparing this computed error to the exact error, the difference depends only on s and the Bernstein coefficients of p:

Theorem 3 ([JLCS10])**.** If no underflow occurs and $n \geq 2$, then

$$\left|\partial b_0^{(0)} - \widehat{\partial b}_0^{(0)}\right| \leq 2\gamma_{3n+1}\gamma_{3(n-1)} \sum_{j=0}^{n} |b_j| \, B_{j,n}(s). \tag{16}$$

Using the bound on the error in ∂b, the algorithm can be shown to be as accurate as if the computations were done in twice the working precision:

Theorem 4 ([JLCS10])**.** If no underflow occurs, $n \geq 2$ and $s \in [0, 1]$

$$\frac{|p(s) - \texttt{CompDeCasteljau}(p, s)|}{|p(s)|} \leq \mathbf{u} + 2\gamma_{3n}^2 \operatorname{cond}(p, s). \tag{17}$$

Unfortunately, Figure 4 shows how the `CompDeCasteljau` starts to break down in a region of high condition number (caused by a multiple root with multiplicity higher than two). For example, the point $s = \frac{1}{2} + 1001\mathbf{u}$ — which is in the plotted region $\left|s - \frac{1}{2}\right| \leq 1.5 \cdot 10^{-11}$ — evaluates to exactly 0 when it should be $\mathcal{O}\left(\mathbf{u}^3\right)$. As shown in Table 1, the breakdown occurs because $\widehat{b}_0^{(0)} = -\widehat{\partial b}_0^{(0)} = \mathbf{u}/16$.

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

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

4 K Compensated de Casteljau

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

$$b_j^{(k)} = \widehat{b}_j^{(k)} + \partial b_j^{(k)} \tag{18}$$

$$\partial b_j^{(k)} = \widehat{\partial b}_j^{(k)} + \partial^2 b_j^{(k)} \tag{19}$$

$$\partial^2 b_j^{(k)} = \widehat{\partial^2 b}_j^{(k)} + \partial^3 b_j^{(k)} \tag{20}$$

$$\vdots$$

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

$$(1-s) \cdot \widehat{b}_j^{(k+1)} + s \cdot \widehat{b}_{j+1}^{(k+1)} = \widehat{b}_j^{(k)} + \ell_{1,j}^{(k)} \tag{21}$$

$$(1-s) \cdot \widehat{\partial b}_j^{(k+1)} + s \cdot \widehat{\partial b}_{j+1}^{(k+1)} + \ell_{1,j}^{(k)} = \widehat{\partial b}_j^{(k)} + \ell_{2,j}^{(k)} \tag{22}$$

$$(1-s) \cdot \widehat{\partial^2 b}_j^{(k+1)} + s \cdot \widehat{\partial^2 b}_{j+1}^{(k+1)} + \ell_{2,j}^{(k)} = \widehat{\partial^2 b}_j^{(k)} + \ell_{3,j}^{(k)} \tag{23}$$

$$\vdots$$

In `CompDeCasteljau` (Algorithm 2), after a single stage of error filtering we "give up" and use $\widehat{\partial b}$ instead of ∂b (without keeping around any information about the round-off error). In order to obtain results that are as accurate as if computed in K times the working precision, we must continue filtering (see Figure 5) errors down $(K-1)$ times, and only at the final level do we accept the rounded $\widehat{\partial^{K-1} b}$ in place of the exact $\partial^{K-1} b$.

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

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

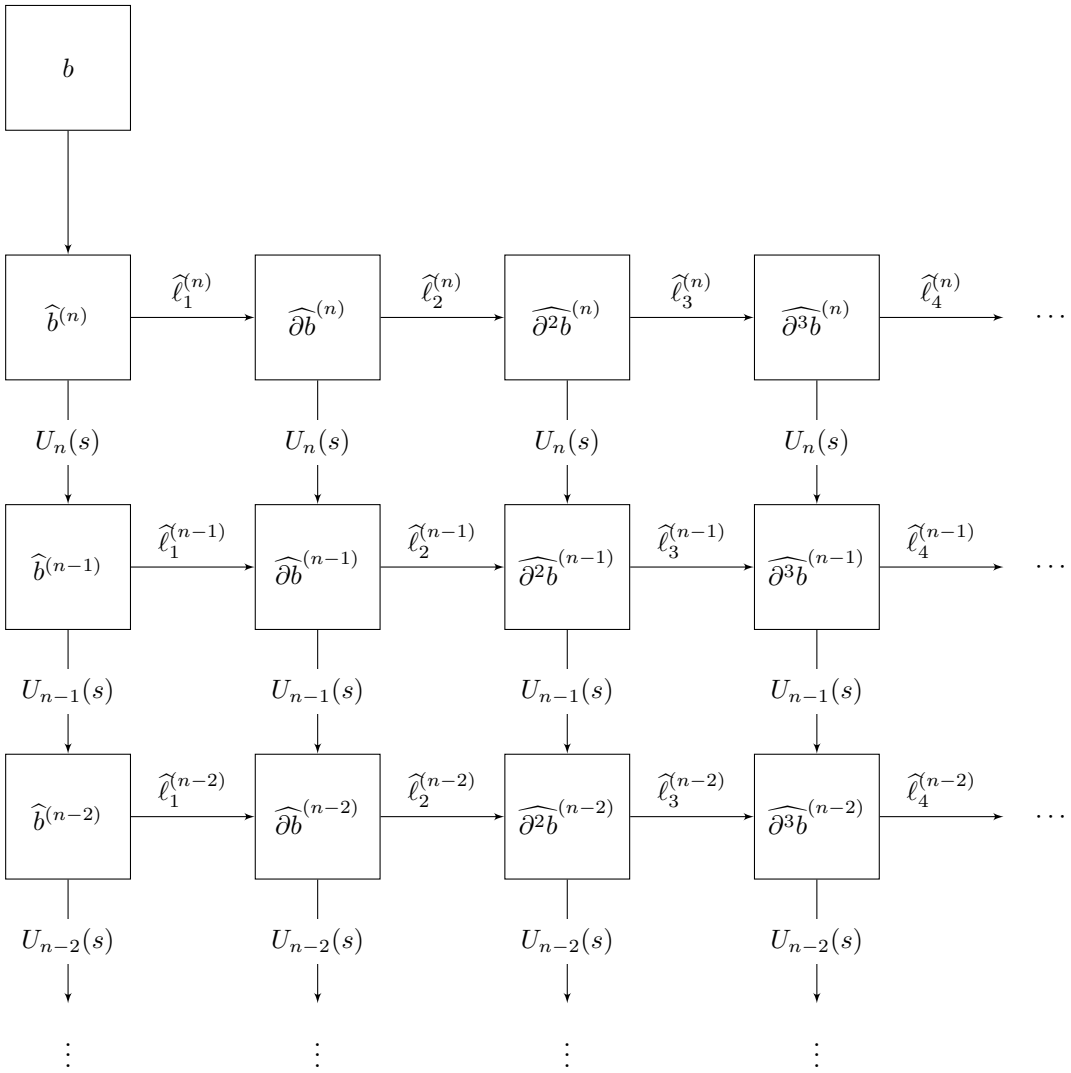

Figure 5: Filtering errors

- an error from the product $s \otimes \widehat{\partial^F b_{j+1}}^{(k+1)}$
- two errors from the two $\oplus$ when combining the three terms in $\left[\widehat{r} \otimes \widehat{\partial^F b_j}^{(k+1)}\right] \oplus \left[s \otimes \widehat{\partial^F b_{j+1}}^{(k+1)}\right] \oplus \widehat{\ell}_{F,j}^{(k)}$

For example, in (11):

$$\ell_{1,j}^{(k)} = \underbrace{\pi_1}_{P_1 = \widehat{r} \otimes \widehat{b}_j^{(k+1)}} + \underbrace{\pi_2}_{P_2 = s \otimes \widehat{b}_{j+1}^{(k+1)}} + \underbrace{\sigma_3}_{P_1 \oplus P_2} + \underbrace{\rho \cdot \widehat{b}_j^{(k+1)}}_{(1-s)\widehat{b}_j^{(k+1)}} \tag{24}$$

After each stage, we'll always have

$$\ell_{F,j}^{(k)} = e_1 + e_2 + \cdots + e_M + \rho \cdot \widehat{\partial^{F-1} b_j}^{(k+1)} \tag{25}$$

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

Algorithm 3 *EFT for computing the local error.*

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

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

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

With this EFT helper in place, we can perform $(K - 1)$ error filtrations:

Algorithm 4 *K-compensated de Casteljau algorithm.*

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

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

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

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

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

$[P_1, \eta_{M+1}] = \texttt{TwoProd}\left(\widehat{r}, \widehat{\partial^F b}_j^{(k+1)}\right)$
$[P_2, \eta_{M+2}] = \texttt{TwoProd}\left(s, \widehat{\partial^F b}_{j+1}^{(k+1)}\right)$
$[S_3, \eta_{M+3}] = \texttt{TwoSum}(P_1, P_2)$
$\left[\widehat{\partial^F b}_j^{(k)}, \eta_{M+4}\right] = \texttt{TwoSum}\left(S_3, \widehat{\ell}\right)$

$e = \eta$
$\delta b = \widehat{\partial^F b}_j^{(k+1)}$
end for

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

$\texttt{res} = \widehat{b}_0^{(0)}$
for $F = 1, \ldots, K-1$ **do**
$\texttt{res} = \texttt{res} \oplus \widehat{\partial^F b}_0^{(0)}$
end for
end function

Noting that $\ell_{F,j}$ contains $5F - 2$ terms, one can show that Algorithm 4 requires

$$(15K^2 + 11K - 34)T_n + K + 5 = \mathcal{O}\left(n^2K^2\right) \tag{26}$$

flops to evaluate a degree n polynomial, where T_n is the nth triangular number. As a comparison, the non-compensated form of de Casteljau requires $3T_n + 1$ flops. In total this will require $(3K - 4)T_n$ uses of `TwoProd`. On hardware that supports FMA, `TwoProdFMA` (Algorithm 8) can be used instead, lowering the flop count by $15(3K - 4)T_n$.

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

$$\frac{|p(s) - \texttt{CompDeCasteljau}(p, s, K)|}{|p(s)|} \leq \left[\mathbf{u} + \mathcal{O}\left(\mathbf{u}^2\right)\right] + \left[(\alpha(n)\mathbf{u})^K + \mathcal{O}\left(\mathbf{u}^{K+1}\right)\right] \operatorname{cond}(p, s). \tag{27}$$

Proof. Note that

$$p(s) = b_0^{(0)} = \widehat{b}_0^{(0)} + \partial b_0^{(0)} = \cdots = \widehat{b}_0^{(0)} + \widehat{\partial b}_0^{(0)} + \cdots + \widehat{\partial^{K-2} b}_0^{(0)} + \partial^{K-1} b_0^{(0)} \tag{28}$$

and

$$\texttt{CompDeCasteljau}(p, s, K) = \widehat{b}_0^{(0)} \oplus \widehat{\partial b}_0^{(0)} \oplus \cdots \oplus \widehat{\partial^{K-2} b}_0^{(0)} \oplus \widehat{\partial^{K-1} b}_0^{(0)}. \tag{29}$$

It will be crucial for us to bound $\partial^K b_0^{(0)} = \partial^{K-1} b_0^{(0)} - \widehat{\partial^{K-1} b}_0^{(0)}$. In addition, note that

$$\widehat{b}_0^{(0)} \oplus \widehat{\partial b}_0^{(0)} \oplus \cdots \oplus \widehat{\partial^{K-2} b}_0^{(0)} \oplus \widehat{\partial^{K-1} b}_0^{(0)} \tag{30}$$

$$= \widehat{b}_0^{(0)}(1 + \theta_{K-1}) + \widehat{\partial b}_0^{(0)}(1 + \theta_{K-1}) + \cdots + \widehat{\partial^{K-2} b}_0^{(0)}(1 + \theta_2) + \widehat{\partial^{K-1} b}_0^{(0)}(1 + \theta_1). \tag{31}$$

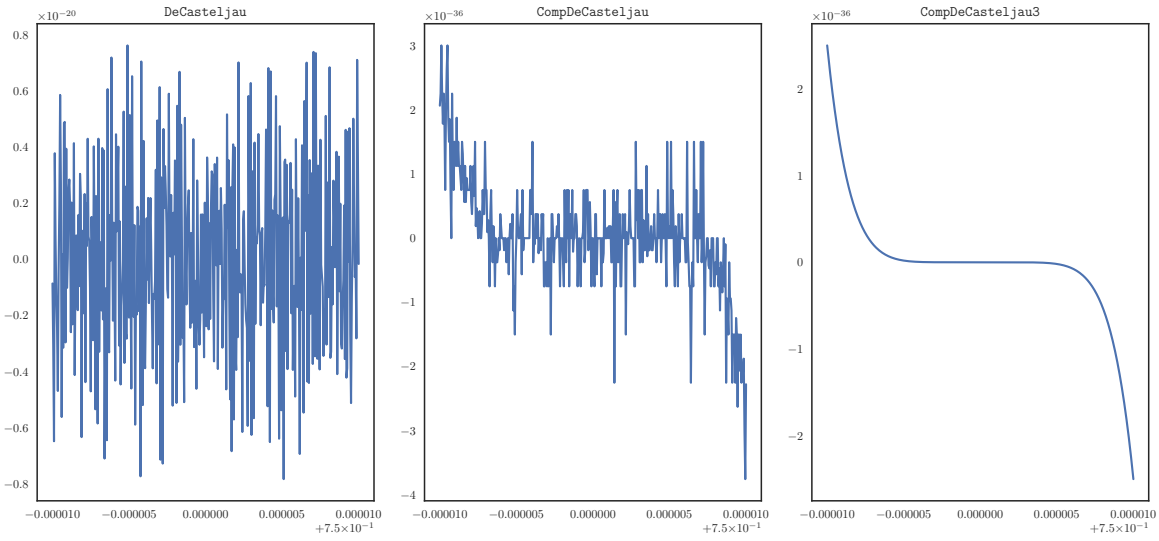

Figure 6: Evaluation of $p(s) = (s-1)(s-3/4)^7$ in the neighborhood of its multiple root $3/4$.

We'll use these two quantities to prove a bound. ■

5 Numerical experiments

All experiments were performed in IEEE-754 double precision. As in [JLCS10], we consider the evaluation in the neighborhood of the multiple root of $p(s) = (s-1)\left(s - \frac{3}{4}\right)^7$, written in Bernstein form.

Figure 6 shows the evaluation of $p(s)$ at the 401 equally spaced[1] points $\left\{\frac{3}{4} + j\frac{10^{-7}}{2}\right\}_{j=-200}^{200}$ with `DeCasteljau` (Algorithm 1), `CompDeCasteljau` (Algorithm 2) and `CompDeCasteljau3` (Algorithm 4 with $K = 3$). We see that `DeCasteljau` fails to get the magnitude correct, `CompDeCasteljau` has the right shape but lots of noise and `CompDeCasteljau3` is able to smoothly evaluate the function. This is in contrast to a similar figure in [JLCS10], where the plot was smooth for the 400 equally spaced points $\left\{\frac{3}{4} + \frac{10^{-4}}{2}\frac{2j-399}{399}\right\}_{j=0}^{399}$. The primary difference is that as the interval shrinks by a factor of $\approx \frac{10^{-4}}{10^{-7}} = 10^3$, the condition number goes up by $\approx 10^{21}$ and `CompDeCasteljau` is no longer accurate.

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

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

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

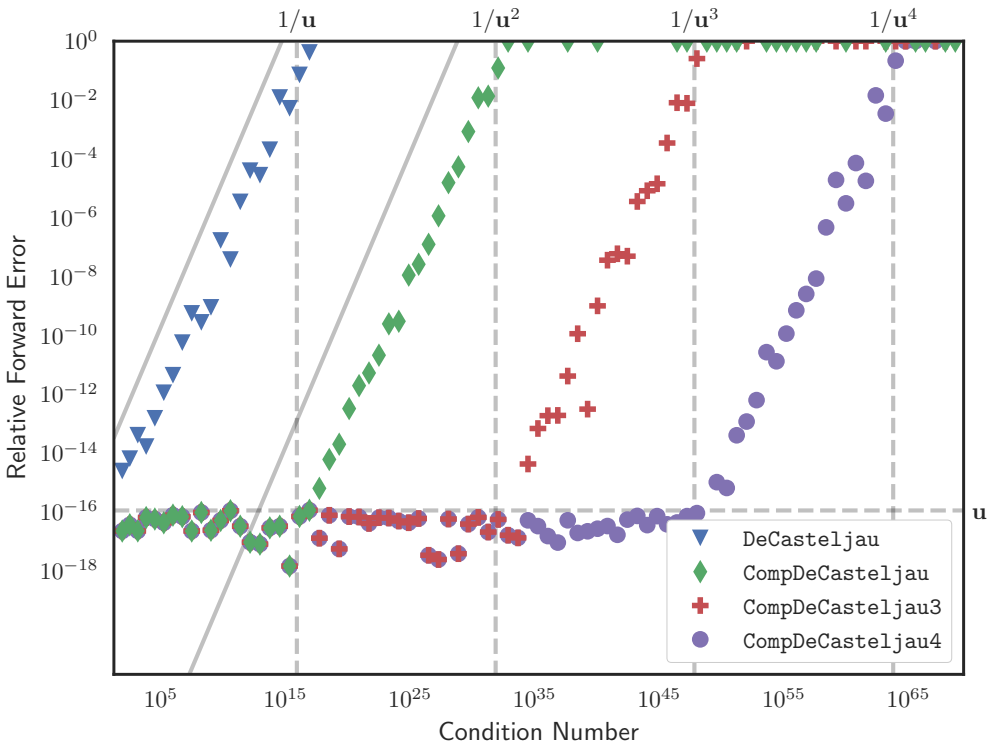

Figure 7: Accuracy of evaluation of $p(s) = (s-1)\,(s-3/4)^7$ represented in Bernstein form.

References

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

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

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

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

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

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

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

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

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

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

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

6 Appendix: EFT Algorithms

Find here concrete implementation details on the EFTs described in Theorem 1. They do not use branches, nor access to the mantissa that can be time-consuming

Algorithm 5 *EFT of the sum of two floating point numbers.*

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

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

Algorithm 6 *Splitting of a floating point number into two parts.*

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

$h = z \ominus (z \ominus a)$
$\ell = a \ominus h$
end function

For IEEE-754 double precision floating point number, $r = 26$ so $2^r + 1$ will be known before `Split` is called. In all, `Split` uses 4 flops.

Algorithm 7 *EFT of the product of two floating point numbers.*

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

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

Algorithm 8 *EFT of the sum of two floating point numbers with a FMA.*

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