DOI: 10.1111/cgf.70403

EUROGRAPHICS 2026 / B. Masia and J. Thies
(Guest Editors)

COMPUTER GRAPHICS *forum*
Volume 45 (2026), Number 2

Register-Efficient Linear-Time Evaluation in the Bernstein Basis

Gábor Valasek[1] and Anna Lili Horváth[1]

Eötvös Loránd University, Hungary
{valasek, hal}@inf.elte.hu

Abstract

We investigate the evaluation of points and derivatives of Bézier curves and surfaces on modern architectures, focusing on performance and guided by numerical error bounds. While the de Casteljau algorithm remains the reference for numerical robustness, its linear working-set size imposes substantial register pressure on GPUs. We introduce a linear-time, constant-storage evaluation framework derived from the ladder algorithm that attains de Casteljau-level robustness and demonstrate that it outperforms other methods both on the GPU and CPU. Our analysis provides backward-error bounds for points and derivatives and it is also supported by empirical tests across degrees commonly used in rendering of curves and surfaces. Moreover, we show that fused multiply-add (FMA) instructions, now ubiquitous in hardware, can improve robustness even for linear interpolation. We advocate a nested FMA formulation that reconstructs endpoints exactly, in contrast to the subtraction-and-FMA pattern prevalent in shader compilers. Together, these results yield reduced memory bandwidth and register pressure, and improved performance.

CCS Concepts

• ***Mathematics of computing*** → ***Computations on polynomials;*** • ***Computing methodologies*** → ***Parametric curve and surface models;***

1. Introduction

The Bernstein basis has been subject to extensive research since its conception, both as a theoretical device and as a practitioner's tool. It has a particular prevalence in computer aided geometric design and computer graphics, used for representing freeform shapes in geometric modeling, vector graphics elements, such as in SVG, and font outlines, including in FreeType and TrueType fonts among other applications. It has been established as a geometrically intuitive and numerically stable reference to represent curves, surfaces, and volumes [Far01]. This body of research developed a rich theoretical and practical foundation, with various approaches to the evaluation of constructs represented in the Bernstein basis. However, comparatively less emphasis has been placed on efficiency as a multifaceted concept, that is, beyond traditional measures of arithmetic complexity.

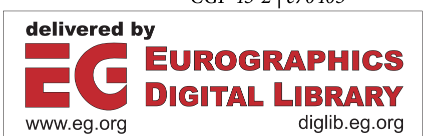
Efficiency is inherently context-dependent: what is optimal in one application domain may be suboptimal in another. On mod-ern parallel architectures, such as GPUs, efficiency is not deter-mined by the number of arithmetic operations solely, but it is also constrained by register usage, memory bandwidth, locality, and en-ergy consumption. Our goal is to investigate how efficiency inter-acts with robustness across different computational settings, and to clarify the trade-offs that arise when seeking practically sustainable evaluation strategies. This, however, cannot be done without a rig-orous look at the numerical characteristics of competing methods.

We show that linear-time, constant-storage (LTCS) Bézier evaluation algorithms offer similar error bounds for evaluation as the standard approaches, in particular, the de Casteljau method.

While the advantages of linear time procedures over quadratic ones are straightforward to interpret as runtime improvements, working-set sizes, or storage requirements, are frequently overlooked aspects when one considers optimization opportunities. Still, these are equally significant, especially in computational environments where memory bandwidth, battery usage, or register space is at a premium. Minimal working-set sizes help to minimize unnecessary traffic between the memory hierarchy and the arithmetic units. The resulting improvements are just as serious in terms of power drain as in runtime statistics. By Gustafson's estimates [Gus15], moving 64 bits of data from DRAM to CPU consumes $\times 65$ more energy than a fused multiply-add (FMA) operation on double precision arguments. Thus, this aspect of algorithms is particularly important from a sustainable computing perspective.

The de Casteljau algorithm is the de facto standard to evaluate and modify geometric entities represented in the Bernstein basis. It is an inherently quadratic algorithm in computational complexity and linear in storage, both with respect to the degree of the polynomial. Although adapted throughout the industry due to its favorable numerical properties [FG96], these do not translate to optimal performance on modern computational architectures. However, no replacement is viable unless it addresses the incredible range of what

© 2026 The Author(s). Computer Graphics Forum published by Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd.
This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.



a single round of de Casteljau algorithm can cover. It not only evaluates points on Bézier curves and surfaces, its intermediary data can be used to compute derivatives or even the control point nets for subdomains.

We show that linear-time, constant-storage algorithms are capable of delivering significant performance and bandwidth gains in value and derivative evaluations with similar theoretical error bounds as the de Casteljau algorithm. There has been a renewed interest in linear time constructs recently [WC20; CW24; FRH24] and we discuss these and the relevant literature in Section 2. We augment these pieces of work by showing what additional optimization opportunities are there to address any remaining numerical concerns in new formulations and how to truly harness their performance potential on CPUs and GPUs.

This requires us to revisit one of the most fundamental building blocks of computer graphics and computer aided geometric design: linear interpolation. The necessary notational and theoretical background for this discussion is covered in Section 3. In Section 4.1, we discuss why the current standard implementation of linear interpolation – subtraction followed by a fused multiply-add – is a suboptimal choice and argue for a better alternative based on a nested fused multiply-add pair. Based on these, in Section 4.2 we show that this nested FMA formulation improves both the theoretical error bound and the empirical behavior of the de Casteljau algorithm.

Section 5 presents our results on the efficient evaluation of Bézier curves. We show that a generalization of Warren's ladder [War95] algorithm can be used to evaluate values and derivatives of Bézier curves. Moreover, we highlight what additional performance and numerical precision optimizations are possible both for this suite of algorithms as well as the de Casteljau method. These results are extended to surfaces in Section 6. Our empirical tests, discussed in Section 7, demonstrate that these techniques behave similarly to the de Casteljau algorithm in terms of precision, while significantly overperforming it both on the CPU and GPU. In particular, we tested our algorithms on an Intel CPU and an NVIDIA GPU. We summarize our findings in Section 8.

Our main results are as follows:

- we provide a theoretical error analysis of linear interpolation and the de Casteljau algorithms,
- we show that fused multiply-add instructions can be used to improve both theoretical and empirical error within these methods, and advocate for the use of nested FMA implementations,
- we extend the ladder algorithmic scheme to the evaluation of points and derivatives of Bézier curves, tensor-product and triangular surfaces,
- we present a comprehensive comparison between various evaluation methods both on the CPU and GPU.

2. Related work

The de Casteljau algorithm is perhaps the best known method to evaluate a point on a Bézier curve. Let $\boldsymbol{b}_0, \ldots, \boldsymbol{b}_n \in \mathbb{R}^d$ denote the control points of a Bézier curve. Then the $\boldsymbol{b}(t)$ point corresponding to parameter $t \in [0,1]$ is defined by de Casteljau's recurrence as

$$\boldsymbol{b}_i^{r+1}(t) = (1-t)\boldsymbol{b}_i^r(t) + t\boldsymbol{b}_{i+1}^r(t) \ , \tag{1}$$

with $\boldsymbol{b}(t) = \boldsymbol{b}_0^n(t)$, where $\forall i \in \{0,\ldots,n\} : \boldsymbol{b}_i^0(t) \equiv \boldsymbol{b}_i$ and $\boldsymbol{b}_i^r(t)$ are such that $r = 0,\ldots,n$, $i = 0,\ldots,n-r$, [Far01]. The de Casteljau algorithm is quadratic in the number of control points and requires linear storage – both multiplied by the dimension d. It possesses remarkable numerical stability with respect to both rounding errors and perturbation of input [DMP23].

A recent work by Yuksel [Yuk24] aims to improve the efficiency of this evaluation approach. It carries out recursive linear interpolations, however, it uses auxiliary, so called Seiler points as arguments to the linear interpolations – derived from the original control points – to achieve a lower total number of operations. Although our tests confirm that it has overall good performance and precision, it is only applicable to curves up to degree 5. Beyond quintics and its generalizations to surfaces is not known currently.

The standard closed-form expression of Bézier curves is

$$\boldsymbol{b}(t) = \sum_{i=0}^{n} \boldsymbol{b}_i B_i^n(t) \ , \tag{2}$$

where $B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}$ are the degree n Bernstein basis polynomials, $i = 0,\ldots,n$. This formulation highlights a way to address the dimension-dependent multiplier in storage and arithmetic complexity: by decoupling the basis function evaluations from that of the curve. This was done in the AllBernstein formulation in Piegl and Tiller [PT97]. Once the basis function values are obtained, the weighted sum of the control points can be computed in linear time and constant storage, multiplied by the dimension. Unfortunately, this improvement is not sufficient to enable register-pressure sensitive use cases.

Schumaker and Volk [SV86] proposed another approach to optimize the evaluation of Bézier triangles. Their method relied on two ingredients: a change of variables and the premultiplication of the control points by the binomial coefficients. Their idea, adapted to curves, is as follows. They noted that one can divide Equation (2) by $(1-t)^n$ $(t \neq 1)$ and the right hand side of $\frac{\boldsymbol{b}(t)}{(1-t)^n} = \sum_{i=0}^{n} \binom{n}{i} \boldsymbol{b}_i \left(\frac{t}{1-t}\right)^i$ can be evaluated by the Horner scheme with $x = \frac{t}{1-t}$. This, and $(1-t)^n$, can be computed in linear time and constant storage, unless a copy has to be made of the original control points for the binomial premultiplication. This formulation is a particularly good fit for rational polynomials [FRH24] as the normalization term on the left hand side disappears. However, $x = \frac{t}{1-t}$ becomes unstable as t approaches 1 and it is preferable to use the $x = \frac{1-t}{t}$ substitution whenever $t \geq \frac{1}{2}$ and choose $x = \frac{t}{1-t}$ otherwise. In contrast, Warren's ladder method and its extensions that we propose are branchless and do not require a change of variables nor premultiplication.

While the de Casteljau and Seiler methods offer an intuitive geometric interpretation, this does not hold for Volk and Schumaker's VS algorithm. This was addressed by Woźny and Chudy [WC20]. They presented a linear-time, constant-storage algorithm using linear interpolations. Their recurrence is written as

$$h_0 = 1 \ , \ h_i = \frac{w_i h_{i-1} t(n-i+1)}{w_{i-1} i(1-t) + w_i h_{i-1} t(n-1+1)} \tag{3}$$

$$\boldsymbol{t}_0 = \boldsymbol{b}_0 \ , \ \boldsymbol{t}_i = (1-h_i)\boldsymbol{t}_{i-1} + h_i \boldsymbol{b}_i \ , \tag{4}$$

© 2026 The Author(s).

where $w_i \in \mathbb{R}$, $i = 0, \ldots, n$ are the weights of the control points, h_i, $i = 0 \ldots, n$ are intermediate coefficients, and t_n is the result. Similarly to the VS algorithm, this also requires a change of variables to mitigate numerical issues but it incorporates the computation of the binomial coefficients. For this reason, we chose this algorithm as a benchmark for ours. Our method offers significant performance improvements over this formulation, however, it also delivers slightly worse numerical precision.

A simpler, linear-time, constant-storage implementation for Bézier curves can be found in the additional content for Farin's book [Far01]. The routine `hornbez` coincides with our proposed specialization of Warren's ladder scheme to the evaluation of Bézier curves. However, the routine is not referenced in the text and no numerical or performance evaluation was found.

The most influential work for our paper is Warren's ladder algorithm [War95] that proposes a linear-time evaluation scheme for constructs represented in the Pólya basis – of which the Bernstein basis is one instance. This work did not consider the computation of the binomial coefficients in constant storage and was not focusing on precision and performance. Most of our algorithms can be considered as the application of the ladder scheme more directly to various use cases. However, we focus on performance and precision and extend the algorithmic toolbox to surfaces.

A recent work from Fuda et al. [FRH24] investigated the theoretical upper bounds on round-off error in various evaluation methods. They noted that basis-conversion based methods, such as adapting the control points to the Wang-Ball basis, are numerically less stable. As such, we ruled these out from our comparisons. They also stressed the necessity of adapting the change of variables to either $\frac{1-t}{t}$ or $\frac{t}{1-t}$ in the VS and Woźny-Chudy algorithms, making them ill suited for branchless constraints.

3. Preliminaries

3.1. Notation and Error Analysis

We use Higham's approach and notation for error analysis [Hig02].

Conceptually, the set of floating point numbers $\mathbb{F}$ is a finite subset of the rational numbers. We base our discussion on the IEEE floating point standard where $\mathbb{F}$ is extended by signed infinities and a special entity called not-a-number (NaN). We do not differentiate between the various flavors of NaNs. Let $\text{fl}(x) : \mathbb{R} \to \mathbb{F}$ denote the truncation operator using a user prescribed rounding mode, that is, the conversion of a real number $x \in \mathbb{R}$ to a float. We denote floating point entities by a hat, i.e., $\hat{x} \in \mathbb{F}$.

Let $\epsilon_M > 0$ denote the machine epsilon for the M bit IEEE floating point representation. This is the distance between 1.0 and the next larger floating point number. For binary32 and binary64 IEEE 754 floats, the machine epsilons are $\epsilon_{32} = 2^{-23} \approx 1.19 \cdot 10^{-7}$ and $\epsilon_{64} = 2^{-52} \approx 2.22 \cdot 10^{-16}$, respectively.

Given an $\hat{f} : \mathbb{F}^d \to \mathbb{F}$ floating point implementation of a computation $f : \mathbb{R}^d \to \mathbb{R}$, we use $R(f(x)) = \frac{f(x) - \hat{f}(x)}{f(x)}$ to denote the relative error. By unit roundoff, $u > 0$, we refer to the rounding mode dependent maximum relative error between a real number and its floating point version. We have $u_M = \frac{\epsilon_M}{2}$ in rounding to nearest even mode for radix 2. We assume that $|R(\text{fl}(x))| \leq u$.

The unit in the last place (ULP) of a $\hat{x} \in \mathbb{F}$ floating point number is the distance between the two closest $\hat{a}, \hat{b} \in \mathbb{F}$, $\hat{a} \neq \hat{b}$, such that $\hat{a} \leq \hat{x} \leq \hat{b}$, assuming no upper bound on the exponent range.

We assume that the implementation of elementary operations is such that their relative error is smaller or equal to the u unit roundoff. By convention, this is most often written as

$$\text{fl}(x \,\text{op}\, y) = (x \,\text{op}\, y)(1+\delta), \quad |\delta| \leq u, \quad \text{op} \in \{+,-,*,/\} .$$

The floating point counterparts of the arithmetic operations on floating point arguments are circled around: $\oplus, \ominus, \odot, \oslash$. These imply a rounding, that is,

$$\hat{a} \oplus \hat{b} \oplus \hat{x} = \left((\hat{a}+\hat{b})(1+\delta_1)+\hat{x}\right)(1+\delta_2) .$$

Note that no reassociation is possible on the left hand side, as all arguments are floating point numbers that are not associative. On the other hand, the expression on the right may be freely reassociated and distributed, as it is an expression in real numbers.

IEEE 754-2008 has standardized the fused-multiply-add (FMA) operations. This carries out a pair of multiplication and addition/subtraction with a single rounding, that is,

$$\text{fl}(\text{fma}(x,y,\pm z)) = \text{fl}(xy \pm z) = (xy \pm z)(1+\delta) .$$

The rounding error analysis of a sequence of operations usually creates multiplicative chains of $(1+\delta_i)$ terms. These can be compressed into a single expression using the following result [Hig02]:

Lemma 1. *If* $\forall i \in \{1, \ldots, n\} : |\delta_i| \leq u, \rho_i = \pm 1$, *and* $n \cdot u < 1$, *then*

$$\prod_{i=1}^{n} (1+\delta_i)^{\rho_i} = 1 + \theta_n,$$

where

$$|\theta_n| \leq \frac{nu}{1-nu} =: \gamma_n.$$

Corollary 1. *Let* $n \in \mathbb{N}, n \geq 1$. *If* $(n+1)u < 1$, *then* $\gamma_{n+1} > \gamma_n$.

We use γ_n to characterize the levels of error in a computation. For notational simplicity, $|\boldsymbol{a}| = \|\boldsymbol{a}\|_\infty$ denotes the maximum norm of $\boldsymbol{a} \in \mathbb{R}^d$. Vector operations, such as $\boldsymbol{ab}$, are per component. In derivatations, we will often index δ and we assume that $|\delta_i| \leq u$, $|\theta_i| \leq \gamma_i$ for the remainder of the paper.

In what follows, we discard the error in Bézier control data and evaluation parameters. However, this is only to simplify notation; in practice, any combination of control point and evaluation parameter exactness and inexactness is a feasible case.

Exact control points and evaluation parameters arise naturally in design applications, where users specify control point positions explicitly and rendering relies on evaluations at exact floating-point parameters. In general, however, evaluation parameters need not be exactly representable as floating point numbers, for example, when obtained from a closest point query. Similarly, control point coordinates may be only floating-point approximations of real numbers, as in cases where they result from continuity constraint resolution or optimization procedures such as fairing functionals.

© 2026 The Author(s).
Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.

3.2. Warren's Ladder Scheme

Warren introduced the ladder scheme for the evaluation of univariate polynomials expressed in a class of bases known as the Pólya basis. The members of the Pólya class satisfy the Marsden equality $(x-t)^n = \sum_{i=0}^{n} d_i^n(t) N_i(x)$, where $N_i(x)$ is the i-th degree n B-spline basis function over some nondecreasing $t_1,\ldots,t_{2n}$ knot sequence, with $t \in (t_n, t_{n+1})$. In closed form, Pólya basis functions can be written as $d_i^n(t) = \prod_{j=1}^{n}(t_{i+j}-t)$. Polynomial bases used in practice, such as the monomial, Lagrange, and Bernstein, can be cast in such form by fixing $t_1,\ldots,t_{2n}$. For example, the Bernstein basis is obtained with the choice $t_1 = \cdots = t_n = 0$, $t_{n+1},\ldots,t_{2n} = 1$, yielding $d_i^n(t) = (-1)^{n-i}\binom{n}{i}B_{n-i}^n(t)$.

Warren devised an efficient linear time algorithm for the evaluation of Pólya form. Let $p_0,\ldots,p_n \in \mathbb{R}$ denote the coefficients of a polynomial $d(t) = \sum_{i=0}^{n} p_i d_i^n(t)$. The algorithm initializes as

$$l_0 = 1\ ,\ u_0 = p_0$$

and evaluates the recurrence

$$\begin{aligned} l_{i+1} &= l_i \cdot (t_{i+n+1} - t) \\ u_{i+1} &= u_i \cdot (t_{i+1} - t) + p_{i+1} l_{i+1} \end{aligned}$$

for $i = 0,\ldots,n-1$. Then, Warren showed that $d(t) = u_n$. In particular for the Bernstein basis, this requires a transformation of the b_i control data as $p_i = (-1)^{n-i}\frac{n!}{i!(n-i)!}b_i$. Our implementation does not require this preprocessing.

4. Implementation Considering Error Analysis

In this section, we discuss various implementations of two standard algorithms: linear interpolation and the de Casteljau method. We show that a nested fused multiply-add realization provides better error bounds than the traditional non-FMA formulations for both algorithms. Additionally, we highlight that the widespread subtract-then-FMA implementation suffers from various numerical issues.

4.1. The Four Lerps

Linear interpolation is a fundamental building block of computational algorithms. It also lies at the heart of the Bernstein basis evaluation algorithms we discuss later in this paper. In its most standard formulation, it is written as

$$\text{lerp}(t) = (1-t)\boldsymbol{a} + t\boldsymbol{b}\ , \tag{5}$$

where $\boldsymbol{a},\boldsymbol{b} \in \mathbb{R}^d$ and $t \in \mathbb{R}$. The parameter is often restricted to $t \in [0,1]$, resulting in a convex barycentric combination. Despite the apparent simplicity of this formulation, there are actually multiple practical realizations of the computation of Equation (5). We restrict our discussion to the ones that take advantage of fused multiply-add operations. The resulting three main options are listed in Algorithm 1.

Algorithm 1c must be the most familiar to GPU practitioners, as it is the default implementation of the `lerp/mix` instructions in all the GPU shader compilers we are aware of. Interestingly, it also happens to be the only option out of the three that does not guarantee reconstruction of the endpoints. Indeed, for example in binary32, if $a = -1$ and $b = 2^{-24}$, then $\text{fl}(b-a) = 1$ because $|b|$ is smaller than the ULP of $|a|$. Thus, $\text{LerpSubFMA}(-1, 2^{-24}, 1) = 0$, which is incorrect, as $\text{lerp}(-1, 2^{-24}, 1) = 2^{-24}$. Note that 2^{-24} is exactly representable in binary16, binary32, and binary64.

Similarly, if a and b are large and of opposing signs, $b-a$ may no longer be a finite number within the chosen floating point representation. In binary16, this means that this formulation cannot be used to interpolate between $a = -32760, b = 32759$ as $\text{LerpSubFMA}(-32760, 32759, t) \equiv \infty$. Algorithm 1a, Algorithm 1b, and Algorithm 1d are not affected by either of these two problems. Consequently, we do not consider Algorithm 1c a viable implementation for our proposed algorithms from a numerical perspective.

Note the emphasis on numerical aspects: a nested FMA implementation does not necessarily translate to better performance compared to the subtract-then-FMA strategy. For example, on AMD RDNA architectures subtraction (`v_subrev_f32`) is encoded as a 32 bit VOP2 command while the machine code for an FMA (`v_fmac_f32`) is a 64 bit VOP3 instruction [25]. This lowers the number of instructions within an instruction cache line.

The first document we could find that discussed Algorithm 1d is Fabian Giesen's blog entry [Gie12]. Its broader class of FMA-based linear interpolations were dubbed as the 'lerp of the future' – unfortunately, that is still an apt name for the nested FMA variant, despite its better numerical properties.

In terms of forward error estimates, the following holds:

Lemma 2. *The relative error bounds of the four linear interpolation implementations shown in Algorithm 1 are*

$$|R(\textit{LerpDirect}(\boldsymbol{a},\boldsymbol{b},t))| \le \kappa_{lerp}(\boldsymbol{a},\boldsymbol{b},t)\cdot|\gamma_3| \tag{6}$$

$$|R(\textit{LerpDirectFMA}(\boldsymbol{a},\boldsymbol{b},t))| \le \kappa_{lerp}(\boldsymbol{a},\boldsymbol{b},t)\cdot|\gamma_2| \tag{7}$$

$$|R(\textit{LerpSubFMA}(\boldsymbol{a},\boldsymbol{b},t)| \le \kappa_{lerp}(\boldsymbol{a},\boldsymbol{b},t)\cdot|\gamma_2| \tag{8}$$

$$|R(\textit{LerpTwoFMA}(\boldsymbol{a},\boldsymbol{b},t)| \le \kappa_{lerp}(\boldsymbol{a},\boldsymbol{b},t)\cdot|\gamma_2| \tag{9}$$

where $\gamma_n \le \frac{nu}{1-nu}$, u is the unit roundoff, and

$$\kappa_{lerp}(\boldsymbol{a},\boldsymbol{b},t) = \frac{|(1-t)|\cdot|\boldsymbol{a}| + |t|\cdot|\boldsymbol{b}|}{|(1-t)\cdot\boldsymbol{a} + t\cdot\boldsymbol{b}|}\ . \tag{10}$$

Proof Let $\boldsymbol{r} = (1-t)\boldsymbol{a} + t\boldsymbol{b} \in \mathbb{R}^d$ denote the real result.

LerpDirect: first, let us derive the error bound on non-FMA direct linear interpolation of Algorithm 1a, equivalently written as

$$\text{LerpDirect}(\boldsymbol{a},\boldsymbol{b},t) = (1 \ominus t)\odot\boldsymbol{a} \oplus t\odot\boldsymbol{b}\ . \tag{11}$$

Let $\hat{\boldsymbol{r}} = \text{LerpDirect}(\boldsymbol{a},\boldsymbol{b},t)$, then

$$\begin{aligned} \hat{\boldsymbol{r}} &= ((1-t)(1+\delta_1)\boldsymbol{a}(\boldsymbol{1}+\boldsymbol{\delta}_2) + t\boldsymbol{b}(\boldsymbol{1}+\boldsymbol{\delta}_3))\cdot(\boldsymbol{1}+\boldsymbol{\delta}_4) && (12)\\ &= (1-t)\boldsymbol{a}(\boldsymbol{1}+\boldsymbol{\theta}_3) + t\boldsymbol{b}(\boldsymbol{1}+\boldsymbol{\theta}_2) && (13) \end{aligned}$$

which, after rearranging, gives the relative error bound of

$$\frac{|\hat{\boldsymbol{r}}-\boldsymbol{r}|}{|\boldsymbol{r}|} \le \kappa_{\text{lerp}}(\boldsymbol{a},\boldsymbol{b},t)\cdot\gamma_3\ . \tag{14}$$

LerpDirectFMA: Algorithm 1b can be written as

$$\text{LerpDirectFMA}(\boldsymbol{a},\boldsymbol{b},t) = \text{fma}(\boldsymbol{a}, 1\ominus t, t\odot\boldsymbol{b}) \tag{15}$$

© 2026 The Author(s).

a. *Direct linear interpolation.*

LerpDirect($\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^d, t$)

1: $\hat{s} \leftarrow 1 - t$ ▷ 1 ×SUB
2: $\hat{\boldsymbol{p}} \leftarrow s \cdot \boldsymbol{a}$ ▷ d ×MUL
3: $\hat{\boldsymbol{q}} \leftarrow t \cdot \boldsymbol{b}$ ▷ d ×MUL
4: $\hat{\boldsymbol{r}} \leftarrow \hat{\boldsymbol{p}} + \hat{\boldsymbol{q}}$ ▷ d ×ADD
5: **return** $\hat{\boldsymbol{r}}$

b. *Direct linear interpolation, FMA.*

LerpDirectFMA($\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^d, t$)

1: $\hat{s} \leftarrow 1 - t$ ▷ 1 ×SUB
2: $\hat{\boldsymbol{p}} \leftarrow t \cdot \boldsymbol{b}$ ▷ d ×MUL
3: $\hat{\boldsymbol{r}} \leftarrow \text{fma}(\boldsymbol{a}, \hat{s}, \hat{\boldsymbol{p}})$ ▷ d ×FMA
4: **return** $\hat{\boldsymbol{r}}$

c. *Subtraction followed by FMA.*

LerpSubFMA($\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^d, t$)

1: $\hat{\boldsymbol{d}} \leftarrow \boldsymbol{b} - \boldsymbol{a}$ ▷ d ×SUB
2: $\hat{\boldsymbol{r}} \leftarrow \text{fma}(\hat{\boldsymbol{d}}, t, \boldsymbol{a})$ ▷ d ×FMA
3: **return** $\hat{\boldsymbol{r}}$

d. *Nested FMA implementation.*

LerpTwoFMA($\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^d, t$)

1: $\hat{\boldsymbol{d}} \leftarrow \text{fma}(\boldsymbol{a}, -t, \boldsymbol{a})$ ▷ d ×FMA
2: $\hat{\boldsymbol{r}} \leftarrow \text{fma}(\boldsymbol{b}, t, \hat{\boldsymbol{d}})$ ▷ d ×FMA
3: **return** $\hat{\boldsymbol{r}}$

Algorithm 1: *Linear interpolation variants with fused multiply-add operations: (a) direct definition, (b) direct implemented with an FMA, (c) subtract-then-FMA, (d) two FMAs. Note that variant (c) does not necessarily reproduce endpoints. In contrast, (a), (b), and (d) ensure that $t = 0$ results in $\boldsymbol{a}$ and $t = 1$ in $\boldsymbol{b}$. The relative error bound of (a) is $\kappa_{lerp}(\boldsymbol{a}, \boldsymbol{b}, t) \cdot \gamma_3$, $\gamma_3 = \frac{3u}{1-3u}$, with u denoting the unit roundoff. All FMA variants have a relative error bound of $\kappa_{lerp}(\boldsymbol{a}, \boldsymbol{b}, t) \cdot \gamma_2$, where $\gamma_2 = \frac{2u}{1-2u}$ but (c) and (d) incur one less rounding in total than (b).*

which is bounded, using $\hat{\boldsymbol{r}} = \text{LerpDirectFMA}(\boldsymbol{a}, \boldsymbol{b}, t)$, as

$$\hat{\boldsymbol{r}} = ((1-t)(1+\delta_1)\boldsymbol{a} + t\boldsymbol{b}(1+\boldsymbol{\delta}_2)) \cdot (1+\boldsymbol{\delta}_3) \tag{16}$$
$$= (1-t)\boldsymbol{a}(1+\boldsymbol{\theta}_2) + t\boldsymbol{b}(1+\boldsymbol{\theta}_2') \ . \tag{17}$$

The relative error is

$$\frac{|\text{LerpDirectFMA}(\boldsymbol{a}, \boldsymbol{b}, t) - \boldsymbol{r}|}{|\boldsymbol{r}|} \leq \kappa_{\text{lerp}}(\boldsymbol{a}, \boldsymbol{b}, t) \cdot \gamma_2 \ . \tag{18}$$

LerpSubFMA: Algorithm 1c is written as

$$\text{LerpSubFMA}\,(\boldsymbol{a}, \boldsymbol{b}, t) = \text{fma}(\boldsymbol{b} \ominus \boldsymbol{a}, t, \boldsymbol{a}) \ , \tag{19}$$

thus, $\hat{\boldsymbol{r}} = \text{LerpSubFMA}\,(\boldsymbol{a}, \boldsymbol{b}, t)$ is developed as,

$$\hat{\boldsymbol{r}} = ((\boldsymbol{b} - \boldsymbol{a})(1+\boldsymbol{\delta}_1)t + \boldsymbol{a})\,(1+\boldsymbol{\delta}_2) \tag{20}$$
$$= (\boldsymbol{b} - \boldsymbol{a})t(1+\boldsymbol{\theta}_2) + \boldsymbol{a}(1+\boldsymbol{\delta}) \ , \tag{21}$$

giving us the relative error bound

$$\frac{|\text{LerpSubFMA}\,(\boldsymbol{a}, \boldsymbol{b}, t) - \boldsymbol{r}|}{|\boldsymbol{r}|} \leq \kappa_{\text{lerp}}(\boldsymbol{a}, \boldsymbol{b}, t) \cdot \gamma_2 \ . \tag{22}$$

LerpTwoFMA: Algorithm 1d is

$$\text{LerpTwoFMA}(\boldsymbol{a}, \boldsymbol{b}, t) = \text{fma}\,(\boldsymbol{b}, t, \text{fma}(\boldsymbol{a}, -t, \boldsymbol{a})) \tag{23}$$

where $-t$ is exact in IEEE floating point representation. Let $\hat{\boldsymbol{r}} = \text{LerpTwoFMA}(\boldsymbol{a}, \boldsymbol{b}, t)$, then, recalling that after resolving the floating point operations reassociation is allowed,

$$\hat{\boldsymbol{r}} = (\boldsymbol{b}t + (-t\boldsymbol{a} + \boldsymbol{a})(1+\boldsymbol{\delta}_1))\,(1+\boldsymbol{\delta}_2) \tag{24}$$
$$= \boldsymbol{b}t(1+\boldsymbol{\delta}) + (-t\boldsymbol{a} + \boldsymbol{a})(1+\boldsymbol{\theta}_2) \tag{25}$$
$$= (1-t)\boldsymbol{a}(1+\boldsymbol{\theta}_2) + t\boldsymbol{b}(1+\boldsymbol{\delta}) \ , \tag{26}$$

which completes the relative error bounds as

$$\frac{|\text{LerpTwoFMA}(\boldsymbol{a}, \boldsymbol{b}, t) - \boldsymbol{r}|}{|\boldsymbol{r}|} \leq \kappa_{\text{lerp}}(\boldsymbol{a}, \boldsymbol{b}, t) \cdot \gamma_2 \ . \tag{27}$$

□

These estimates provide only a partial view on the numerical behavior of these algorithms and they rely on the presence of FMA operations. Forcing the compiler to emit the latter may be more involved, always consult the appropriate compiler manuals for details. The FMA based formulation provide better bounds but that is only indicative regarding the worst-case behavior of algorithms. Decisions on upper bounds alone are ill advised, empirical testing has to back these. Based on the tests of Figure 1 and the results of Lemma 2, we propose to use the nested FMA formulation for linear interpolation.

4.2. The de Casteljau algorithm

Algorithm 2 shows the two main variants of interest that implement the computation in Equation (1). It was shown by Hermes [Her18] that the error of Algorithm 2a is characterized as

$$\left|\boldsymbol{b}(t) - \text{deCasteljau}(\{\boldsymbol{b}_i\}_{i=0}^n, t)\right| \leq \gamma_{3n} \cdot \sum_{i=0}^{n} |B_i^n(t)| \cdot |\boldsymbol{b}_i| \ . \tag{28}$$

This formulation emphasizes the backward stability of the implementation, that is, the fact that the result is exact for a slightly perturbed input, with perturbation magnitudes bounded from above by γ_{3n}. Relative error is obtained via division as

$$|R(\text{deCasteljau}(\{\boldsymbol{b}_i\}_{i=0}^n, t))| \leq \kappa_{\text{B}}(\{\boldsymbol{b}_i\}_{i=0}^n, t)) \cdot \gamma_{3n} \ , \tag{29}$$

where the implementation-independent condition number is

$$\kappa_{\text{B}}(\{\boldsymbol{b}_i\}_{i=0}^n, t) = \frac{\sum_{i=0}^n |B_i^n(t)| \cdot |\boldsymbol{b}_i|}{|\sum_{i=0}^n B_i^n(t)\boldsymbol{b}_i|} \ . \tag{30}$$

The relative error bound of Algorithm 2b is given as follows:

Lemma 3. *The relative error of the nested FMA implementation of the de Casteljau algorithm in Algorithm 2b is*

$$|R(\mathit{deCasteljauTwoFMA}(\{\boldsymbol{b}_i\}_{i=0}^n, t))| \leq \kappa_B(\{\boldsymbol{b}_i\}_{i=0}^n, t)) \cdot \gamma_{2n} \ . \tag{31}$$

Proof The recurrence of the algorithm with floating point operations is written as

$$\hat{\boldsymbol{d}}_i^{(k)} = \text{fma}(\boldsymbol{d}_{i+1}^{(k-1)}, t, \text{fma}(\boldsymbol{d}_i^{(k-1)}, -t, \boldsymbol{d}_i^{(k-1)})) \ , \tag{32}$$

© 2026 The Author(s).
Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.

thus, recalling Equation (26), we have

$$\begin{aligned}\hat{\boldsymbol{d}}_0^{(n)} &= \mathrm{fma}(\hat{\boldsymbol{d}}_1^{(n-1)}, t, \mathrm{fma}(\hat{\boldsymbol{d}}_0^{(n-1)}, -t, \hat{\boldsymbol{d}}_0^{(n-1)})) \\ &= (1-t)\hat{\boldsymbol{d}}_0^{(n-1)}(\mathbf{1}+\boldsymbol{\theta}_2^{(n)}) + t\hat{\boldsymbol{d}}_1^{(n-1)}(\mathbf{1}+\boldsymbol{\delta}^{(n)}) \\ &= (1-t)\left(\mathrm{fma}(\hat{\boldsymbol{d}}_1^{(n-2)}, t, \mathrm{fma}(\hat{\boldsymbol{d}}_0^{(n-2)}, -t, \hat{\boldsymbol{d}}_0^{(n-2)}))\right)(\mathbf{1}+\boldsymbol{\theta}_2^{(n)}) \\ &\quad + t\left(\mathrm{fma}(\hat{\boldsymbol{d}}_2^{(n-2)}, t, \mathrm{fma}(\hat{\boldsymbol{d}}_1^{(n-2)}, -t, \hat{\boldsymbol{d}}_1^{(n-2)}))\right)(\mathbf{1}+\boldsymbol{\delta}^{(n)}) \\ &= B_0^2(t)\hat{\boldsymbol{d}}_0^{(n-2)}(\mathbf{1}+\boldsymbol{\theta}_4) + B_1^2(t)\hat{\boldsymbol{d}}_1^{(n-2)}(\mathbf{1}+\boldsymbol{\theta}_3) + B_2^2(t)\hat{\boldsymbol{d}}_2^{(n-2)}(\mathbf{1}+\boldsymbol{\theta}_2) \\ &= \sum_{i=0}^{n} B_i^n(t)\boldsymbol{b}_i(\mathbf{1}+\boldsymbol{\theta}_{2n-i}) \,.\end{aligned}$$

This gives

$$\hat{\boldsymbol{d}}_0^{(n)} - \boldsymbol{b}(t) = \sum_{i=0}^{n} B_i^n(t)\boldsymbol{b}_i\boldsymbol{\theta}_{2n-i} \,, \tag{33}$$

which can be bounded from above as

$$|\hat{\boldsymbol{d}}_0^{(n)} - \boldsymbol{b}(t)| \le \gamma_{2n} \sum_{i=0}^{n} |B_i^n(t)| \cdot |\boldsymbol{b}_i| \,. \tag{34}$$

The statement is obtained after division by $|\boldsymbol{b}(t)|$. □

For the sake of completeness, we include the error and runtime statistics of all four de Casteljau variants in Figure 1. We differentiate between four cases, depending on whether the control points and parameters are exact floating point numbers or not. Note the improved behaviour of the nested FMA variant for $t \in [0, 0.5]$. This is indeed a difficult range, as $t \in \mathbb{F} \cap [0, 0.5]$ does not imply $1 - t \in \mathbb{F}$ in general. The plots suggest that the inexactness of the evaluation parameters is more descriptive of the numerical behavior of the algorithms. Overall, the nested FMA formulation is the most precise.

5. Linear-Time, Constant-Storage Curve Algorithms

5.1. Point Evaluation

Our optimization, which coincides with the ladder scheme, is based on the closed-form formulation of Bézier curves, as listed in Equation (2). In particular, we take advantage of the fact that $B_i^n(t)$ contains $(1-t)$ terms with decreasing, and t terms with increasing exponents, that is,

$$\boldsymbol{b}(t) = \boldsymbol{b}_0\binom{n}{0}t^0(1-t)^n + \boldsymbol{b}_1\binom{n}{1}t^1(1-t)^{n-1} + \boldsymbol{b}_2\binom{n}{2}t^2(1-t)^{n-2} + \ldots \tag{35}$$

This allows us to make a Horner-esque recurrence for the $(1-t)$ terms with an additional multiplicative accumulator for the increasingly exponentiated t terms. The binomial coefficients may be either pregenerated or computed on the fly by using the identity $\binom{n}{i-1}\frac{n-i+1}{i} = \binom{n}{i}$, $i = 1, \ldots, n$. Algorithm 3 summarizes the constant storage approach with three variants. The precomputation of the binomial coefficients does increase the total memory requirements of an implementation, however, these need not be sourced from stack or any memory at all. Indeed, in the presence of aggressive compiler optimizations, the binomial coefficients may be inlined into the machine code for compile-time known degrees. This, however, also holds for the constant time evaluation formulation.

From a pipelined ALU perspective, the recurrence proposed in Algorithm 3 is prone to cause pipeline bubbles. For example, Step 4 requires an updated $\boldsymbol{p}$ and t_k, that is, the retirement of their respective instructions from the previous iteration. Depending on arithmetic pipeline depth, this may cause a stall of a couple of cycles.

This can be alleviated by manually unrolling parts of the loop, at the expensive of register requirements. Similarly to the Estrin scheme [Est60], we can do so in a FMA-friendly manner. The resulting implementation is shown in Algorithm 4.

Theorem 1. *The relative error the linear-time, constant storage algorithms in Algorithm 3 are*

$$|R(LTCS(\{\boldsymbol{b}_i\}_{i=0}^n, t))| \le \kappa_B(\{\boldsymbol{b}_i\}_{i=0}^n, t)) \cdot \gamma_{3n+2} \tag{36}$$

$$|R(LTCSTwoFMA(\{\boldsymbol{b}_i\}_{i=0}^n, t))| \le \kappa_B(\{\boldsymbol{b}_i\}_{i=0}^n, t)) \cdot \gamma_{2n+1} \tag{37}$$

Proof At every iteration, we have the invariant $t_{2k} = t2k = t^{2k}$. Its recurrence can be written as

$$\hat{t}_k = \hat{t}_{k-1} \odot \hat{t} \,, \tag{38}$$

leading to the rounding error form of

$$\begin{aligned}\hat{t}_k &= \hat{t}_{k-1} \odot t = \hat{t}_{k-1} \cdot t \cdot (1+\delta_k) &(39)\\ &= \hat{t}_{k-2} \cdot t \cdot (1+\delta_{k-1}) \cdot t \cdot (1+\delta_k) &(40)\\ &= t^k \cdot (1+\theta_{k-1}) \,. &(41)\end{aligned}$$

The recurrence of Algorithm 3a and its error can be written as

$$\begin{aligned}\hat{\boldsymbol{p}}^{(k)} &= (1 \ominus t) \odot \hat{\boldsymbol{p}}^{(k-1)} \oplus \binom{n}{k} \odot \hat{t}_k \odot \boldsymbol{b}_k &(42)\\ &= (1-t)\hat{\boldsymbol{p}}^{(k-1)}(\mathbf{1}+\boldsymbol{\theta}_3) + \binom{n}{k} t^k \boldsymbol{b}_k(\mathbf{1}+\boldsymbol{\theta}_{k+2}) \,. &(43)\end{aligned}$$

Starting from the final evaluation, the error propagation is

$$\hat{\boldsymbol{p}}^{(n)} = \binom{n}{n} t^n \boldsymbol{b}_n(\mathbf{1}+\boldsymbol{\theta}_{n+2}^{(n)}) + (1-t)\hat{\boldsymbol{p}}^{(n-1)}(\mathbf{1}+\boldsymbol{\theta}_3^{(n)}) \,, \tag{44}$$

where

$$\hat{\boldsymbol{p}}^{(n-1)} = \binom{n}{n-1} t^{n-1} \boldsymbol{b}_{n-1}(\mathbf{1}+\boldsymbol{\theta}_{n+1}^{(n-1)}) + (1-t)\hat{\boldsymbol{p}}^{(n-2)}(\mathbf{1}+\boldsymbol{\theta}_3^{(n-1)}) \tag{45}$$

thus, after recursive substitution, we get

$$\begin{aligned}\hat{\boldsymbol{p}}^{(n)} &= \binom{n}{n} t^n \boldsymbol{b}_n(\mathbf{1}+\boldsymbol{\theta}_{n+2}^{(n)}) + \binom{n}{n-1} t^{n-1}(1-t)\boldsymbol{b}_{n-1}(\mathbf{1}+\boldsymbol{\theta}_{n+4}^{(n-1)}) &(46)\\ &\quad + (1-t)\hat{\boldsymbol{p}}^{(n-2)}(\mathbf{1}+\boldsymbol{\theta}_6^{(n-1)}) &(47)\\ &= \sum_{i=0}^{n} B_i^n(t)\boldsymbol{b}_i(\mathbf{1}+\boldsymbol{\theta}_{3n+2-2i}) \,. &(48)\end{aligned}$$

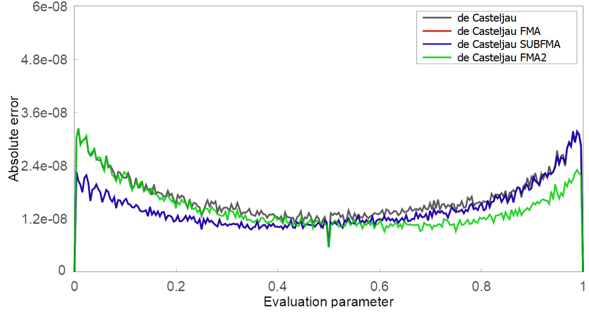
This allows us to bound the rounding error as

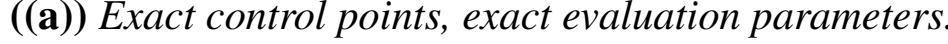
$$|\hat{\boldsymbol{p}}^{(n)} - \boldsymbol{b}(t)| \le \gamma_{3n+2} \cdot \sum_{i=0}^{n} |B_i^n(t)| \cdot |\boldsymbol{b}_i| \,. \tag{49}$$

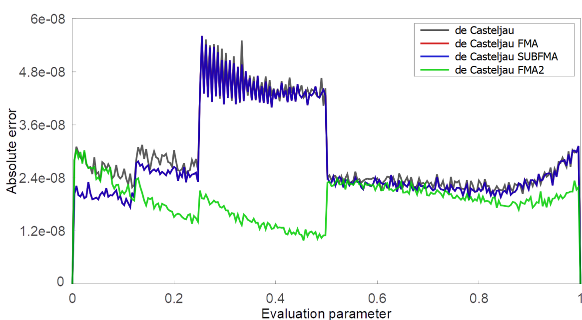
For the FMA formulation the proof is structurally the same, we only have to modify the error statement for the recurrence step as

$$\begin{aligned}\hat{\boldsymbol{p}}^{(k)} &= \mathrm{fma}(\boldsymbol{b}_k, \binom{n}{k}\hat{t}_k, \mathrm{fma}(\hat{\boldsymbol{p}}^{(k-1)}, -t, \hat{\boldsymbol{p}}^{(k-1)})) &(50)\\ &= (1-t)\hat{\boldsymbol{p}}^{(k-1)}(\mathbf{1}+\boldsymbol{\theta}_2) + \binom{n}{k} t^k \boldsymbol{b}_k(\mathbf{1}+\boldsymbol{\theta}_{k+1}) \,. &(51)\end{aligned}$$

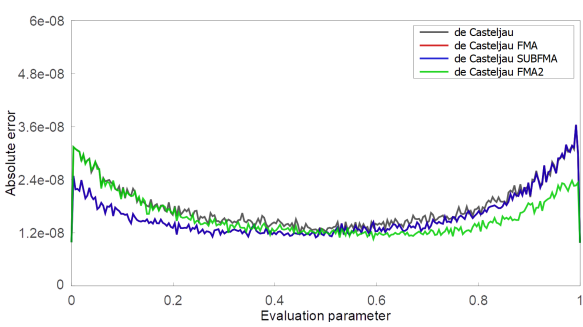
© 2026 The Author(s).
Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.

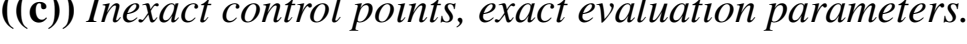


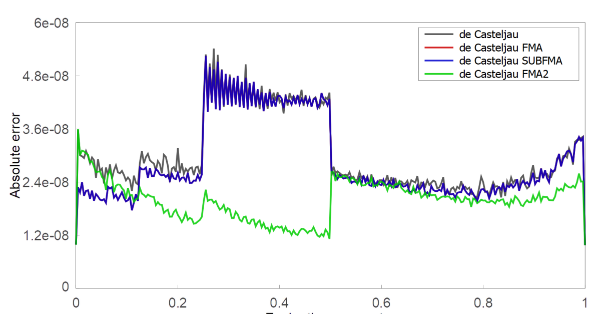

((a)) *Exact control points, exact evaluation parameters.*



((b)) *Exact control points, inexact evaluation parameters.*




((c)) *Inexact control points, exact evaluation parameters.*



((d)) *Inexact control points, inexact evaluation parameters.*

Figure 1: *Absolute error plots of the four the de Casteljau implementations. The black and green lines correspond to the deCasteljau and deCasteljauTwoFMA variants listed in Algorithm 2. These figures aggregate absolute error (Y axis) over the $t \in [0,1]$ range (X axis). These were taken over 32-32 random Bézier curves from degrees 2 to 10, all control data within the range of $[-1,1]$. The curves were evaluated at 255 and 256 parameters using an exact rational ground truth which was compared to the single precision evaluations of the four algorithms. Note that endpoint reconstruction can be observed on the top two figures that list statistics for curves whose control points are exact floating point numbers. The columns correspond to exactly representable evaluation parameters (left) and inexact ones that do not have a finite floating point number representation (right). The bottom row shows error distributions when the control points already have truncation error.*

deCasteljau($\{\mathbf{b}_i \in \mathbb{R}^d\}_{i=0}^n, t \in \mathbb{R}$)

1: **for** $i = 0, \ldots, n$ **do** $\boldsymbol{d}_i \leftarrow \boldsymbol{b}_i$ **end for**
2: **for** r = 1, . . . , n **do**
3:     **for** i = 0, . . . , n - r **do**
4:         $\boldsymbol{d}_i \leftarrow (1-t)\boldsymbol{d}_i + t\boldsymbol{d}_{i+1}$
5:     **end for**
6: **end for**
7: **return** $\boldsymbol{d}_0$

a. *Direct implementation of the de Casteljau algorithm. Its relative error is bounded by $\kappa(\{\boldsymbol{b}_i\}_{i=0}^n, t) \cdot \gamma_{3n}$.*

deCasteljauTwoFMA($\{\mathbf{b}_i \in \mathbb{R}^d\}_{i=0}^n, t \in \mathbb{R}$)

1: **for** $i = 0, \ldots, n$ **do** $\boldsymbol{d}_i \leftarrow \boldsymbol{b}_i$ **end for**
2: **for** r = 1, . . . , n **do**
3:     **for** i = 0, . . . , n - r **do**
4:         $\boldsymbol{d}_i \leftarrow \text{fma}(\boldsymbol{d}_{i+1}, t, \text{fma}(\boldsymbol{d}_i, -t, \boldsymbol{d}_i))$
5:     **end for**
6: **end for**
7: **return** $\boldsymbol{d}_0$

b. *Nested FMA implementation of the de Casteljau algorithm. Its relative error is bounded by $\kappa(\{\boldsymbol{b}_i\}_{i=0}^n, t) \cdot \gamma_{2n}$.*

Algorithm 2: *The linear storage formulation of the de Casteljau algorithm. Its time complexity is $\Theta(d \cdot n^2)$, however, the nested FMA implementation offers better theoretical and empirical error bounds.*

Using this, we have

$$\hat{\boldsymbol{p}}^{(n)} = \sum_{i=0}^{n} B_i^n(t)\boldsymbol{b}_i(\mathbf{1} + \boldsymbol{\theta}_{2n+1-i}) \,. \tag{52}$$

The statement for Algorithm 3b follows from rearranging, upper bounding with γ_{2n+1}, and division by $|\boldsymbol{b}(t)|$. □

5.2. Derivative Evaluation

The most straightforward way to evaluate the first derivative of a curve is to run an arbitrary LTCS variant on the on-the-fly computed $n\Delta\boldsymbol{b}_i$ control points. This approach is shown in Algorithm 5.

However, in practice, derivatives are queried in cohorts, i.e., if a request is issued for the r-th derivative, it can be generally assumed

© 2026 The Author(s).
Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.

a. *Abstract linear time evaluation.*

```
1: LTCS({b_i}_{i=0}^n, t)
2: p ← b_0, tk ← t
3: for k = 1...n do
4:     p ← (1−t)·p + C(n,k)·tk·b_k
5:     tk ← tk·t
6: end for
7: return p
```

b. *Nested FMA abstract algorithm.*

```
1: LTCSTwoFMA({b_i}_{i=0}^n, t)
2: p ← b_0, tk ← t
3: for k = 1...n do
4:     q = fma(p, −t, p)
5:     p = fma(b_k, C(n,k)·tk, q)
6:     tk = tk·t
7: end for
8: return p
```

c. *Constant storage via rolling binomial.*

```
1: LTCSRolling({b_i}_{i=0}^n, t)
2: p ← b_0, tk ← t, c ← 1
3: for k = 1...n do
4:     c ← c·(n−k+1)/k
5:     p ← (1−t)·p + c·tk·b_k
6:     tk ← tk·t
7: end for
8: return p
```

Algorithm 3: *Linear time algorithms for the direct evaluation of Bézier curves. Algorithm 3a can be considered as the direct application of the ladder scheme to the Bernstein basis. Algorithm 3b is the numerically most robust realization thereof. The strength of this formulation is that it allows tabulation optimization opportunities, that are not present in the de Casteljau expression. The binomial term may be sourced from a lookup table, for the sake of efficiency, or computed in a rolling manner, as shown in Algorithm 3c.*

a. *Abstract linear time evaluation with unrolled loop.*

```
1: LTCSUnrolled({b_i}_{i=0}^n, t)
2: p ← 0, t2k ← 1
3: for k = 0...(n+1)/2 do
4:     p ← p·(1−t)^2 + (C(n,2k)·(1−t)·b_{2k} + C(n,2k+1)·t·b_{2k+1})·t2k
5:     t2k ← t2k·t^2
6: end for
7: if n mod 2 = 1 then p ← p·(1−t) + b_n·t2k
8: return p
```

b. *Nested FMA linear time unrolled algorithm.*

```
1: LTCSUnrolledTwoFMA({b_i}_{i=0}^n, t)
2: p ← 0, t2 ← t·t, t2k ← 1
3: for k = 0...(n+1)/2 do
4:     l ← fma(p, −t, p)
5:     l ← fma(l, −t, l)                       ▷ p·(1−t)^2
6:     r ← C(n,2k) fma(b_{2k}, −t, b_{2k})
7:     r ← fma(b_{2k+1}, t·C(n,2k+1), r)
8:     p ← fma(r, t2k, l)
9:     t2k = t_{2k}·t^2
10: end for
11: if n mod 2 = 1 then p = fma(b_n, t2k, fma(p, −t, p))
12: return p
```

Algorithm 4: *Unrolled optimization for Algorithm 3. The binomials may be computed in a rolling manner, as shown in Algorithm 3c.*

Algorithm 5 LTCSDerivative($\{\mathbf{b}_i\}_{i=0}^n, t, m$)

```
1: d = 0, t' = t
2: for k = 1...n do
3:     d = (1−t)·d + C(n−1,k−1)·t'·(b_k − b_{k−1})
4:     t' = t'·t
5: end for
6: return n·d
```

that all derivatives up to order r are needed, including the point on the curve. The following formula connects the derivatives in the Bernstein basis and the intermediary de Casteljau points [Far01]:

$$\boldsymbol{b}^{(r)}(t) = \frac{n!}{(n-r)!}\sum_{i=0}^{n-r} B_i^{n-r}(t)\Delta^r \boldsymbol{b}_j = \frac{n!}{(n-r)!}\Delta^r \boldsymbol{b}_0^{n-r}, \tag{53}$$

where $\boldsymbol{b}_i^{n-r}$ are the intermediary de Casteljau points and $\Delta^{k+1}\boldsymbol{b}_i = \Delta^k \boldsymbol{b}_{i+1} - \Delta^k \boldsymbol{b}_i$ are the higher order forward differences, $\Delta^0 \boldsymbol{b}_i = \boldsymbol{b}_i$.

This allows us to formulate a two-phase algorithm for the fast evaluation of all derivatives up to a requested order r, shown in Algorithm 6. First, we use the LTCS method to obtain the r-th level de Casteljau points. This is possible, as $\boldsymbol{b}_i^r = \sum_{j=0}^{n-r} B_j^{n-r}(t)\boldsymbol{b}_{i+j}$, that is, we can run $r+1$ concurrent LTCS evaluations. Once that is done, one has to form the derivatives, starting with $\boldsymbol{b}^{(r)}(t)$, which requires the computation of the forward differences. However, these merely need the binomials with alternating sign, shown in Line 15. The other intermediary de Casteljau control points are overwritten by applying a single step of the de Casteljau algorithm, see Line 16. The storage of this algorithm is linear in the output and its time complexity is $\Theta(d \cdot n \cdot r)$.

This algorithm is the foundation for a hybrid evaluation method that uses LTCS to compute the penultimate de Casteljau point pair and use the de Casteljau algorithm to compute the point on the curve. Algorithm 7 provides the details for this approach.

© 2026 The Author(s).

Algorithm 6 LTCSDerivatives($\{\mathbf{b}_i\}_{i=0}^{n}, r, t$)

1: $ni \leftarrow 1$
2: **for** $i = 0 \dots r$ **do**
3: $\quad nck \leftarrow 1$, $ti \leftarrow t$, $ni \leftarrow ni \cdot (n-i)$
4: $\quad \boldsymbol{d}_i \leftarrow 0$
5: $\quad$ **for** $j = 1 \dots n-r$ **do**
6: $\quad\quad nck \leftarrow nck \cdot \frac{n+1-r-j}{j}$
7: $\quad\quad \boldsymbol{d}_i \leftarrow \boldsymbol{d}_i \cdot (1-t) + nck \cdot ti \cdot \boldsymbol{b}_{j+i}$ ▷ Compute $\boldsymbol{b}_i^{n-r}$
8: $\quad\quad ti \leftarrow ti \cdot t$
9: $\quad$ **end for**
10: **end for**
11: **for** $i = 0 \dots r$ **do**
12: $\quad nck \leftarrow 1$, $sg \leftarrow 1$, $ni \leftarrow \frac{ni}{n-r+i}$
13: $\quad$ **for** $j = 1 \dots r-i$ **do**
14: $\quad\quad nck \leftarrow nck \cdot \frac{r-i+1-j}{j}$, $sg \leftarrow -sg$
15: $\quad\quad \boldsymbol{d}_{r-i} \leftarrow ni \cdot (\boldsymbol{d}_{r-i} + nck \cdot sg \cdot \boldsymbol{d}_{r-i-j})$ ▷ $\boldsymbol{b}^{(r-i)}(t)$
16: $\quad\quad \boldsymbol{d}_{r-i-j} \leftarrow (1-t) \cdot \boldsymbol{d}_{r-i-j} + t \cdot \boldsymbol{d}_{r-i-j+1}$ ▷ $\boldsymbol{b}_{r-i-j+1}^{r-i}$
17: $\quad$ **end for**
18: **end for**
19: **return** $\boldsymbol{d}$

Algorithm 7 LTCS with de Casteljau($\{\mathbf{b}_i\}_{i=0}^{n}, r, t$)

1: **for** $i = 0, \dots, n$ **do** $\boldsymbol{B}_i' \leftarrow \boldsymbol{B}_i$ **end for**
2: **for** $i = 0, \dots, r$ **do** $\boldsymbol{d}_i' \leftarrow \mathbf{0}$ **end for**
3: $n' \leftarrow 1$
4: **for** $i = 0 \dots r$ **do**
5: $\quad nck \leftarrow 1$, $ti \leftarrow t$, $n' \leftarrow n' \cdot (n-i)$
6: $\quad$ **for** $j = 1 \dots n-r$ **do**
7: $\quad\quad nck \leftarrow nck \cdot \frac{n+1-r-j}{j}$
8: $\quad\quad B_i' \leftarrow B_i' \cdot (1-t) + nck \cdot ti \cdot B_{j+i}$
9: $\quad\quad ti \leftarrow ti \cdot t$
10: $\quad$ **end for**
11: **end for**
12: **for** $i = 0 \dots r$ **do**
13: $\quad nck \leftarrow 1$, $sg \leftarrow 1$, $d_{r-i} \leftarrow B_{r-i}'$
14: $\quad n' \leftarrow \frac{n'}{n-r+i}$
15: $\quad$ **for** $j = 1 \dots r-i$ **do**
16: $\quad\quad nck \leftarrow nck \cdot \frac{r-i+1-j}{j}$
17: $\quad\quad sg \leftarrow -sg$
18: $\quad\quad d_{r-i} \leftarrow n' \cdot (d_{r-i} + nck \cdot sg \cdot B_{r-i-j}')$
19: $\quad\quad B_{r-i-j}' \leftarrow (1-t) \cdot B_{r-i-j}' + t \cdot B_{r-i-j+1}'$
20: $\quad$ **end for**
21: **end for**
22: **return** $\boldsymbol{d}$

5.3. Indefinite Integral Evaluation

The definite integral of a Bézier curve is written as

$$\int_0^1 \boldsymbol{b}(t)dt = \frac{1}{n+1} \sum_{i=0}^{n} \boldsymbol{b}_i \,, \tag{54}$$

which is straightforward to implement in constant storage and linear time, as long as a numerically robust summation algorithm is available.

On the other hand, the indefinite integral is given by

$$\int \boldsymbol{b}(t)dt = \frac{1}{n+1} \sum_{i=0}^{n+1} \left(\sum_{j=0}^{i-1} \boldsymbol{b}_j \right) B_i^{n+1}(t) + C \,,\ C \in \mathbb{R} \,, \tag{55}$$

where the empty sum is defined to be zero. Evaluating the indefinite integral can be done naïvely by generating its control points and evaluating the resulting degree $n+1$ Bézier curve. However, this requires linear storage. Still, linear time evaluation and accumulation based control point computation can be combined to evaluate the indefinite integral of a Bézier curve at a given parameter in constant $\Theta(d)$ storage. Algorithm 8 illustrates this in $\Theta(n \cdot d)$ arithmetic complexity.

Algorithm 8 EvaluateBezierIntegral($\{\mathbf{b}_i\}_{i=0}^{n}, t$)

1: $\boldsymbol{p} \leftarrow \mathbf{0}$, $\boldsymbol{q} \leftarrow \mathbf{0}$, $tk \leftarrow t$
2: **for** $i = 1 \dots n+1$ **do**
3: $\quad \boldsymbol{s} \leftarrow t \cdot \boldsymbol{q} + tk \cdot \mathbf{b}_{i-1}$
4: $\quad \boldsymbol{p} \leftarrow \boldsymbol{p} \cdot (1-t) + \binom{n}{i} \cdot \boldsymbol{s}$
5: $\quad \boldsymbol{q} \leftarrow \boldsymbol{s}$, $tk \leftarrow tk \cdot t$
6: **end for**
7: **return** $\frac{1}{n+1} \cdot \boldsymbol{p}$

6. Linear-Time, Constant-Storage Surface Algorithms

We extend our results to surfaces in this section. Due to space constraints, we only display the abstract algorithms for evaluation and we do not formulate the nested FMA variants anymore. These are most easily obtained by rephrasing the recurrences as linear interpolations.

6.1. Point Evaluation

Algorithm 9 TensorLTCS($\{\mathbf{b}_{ij}\}_{i=0,j=0}^{n,m}, u, v$)

1: $\boldsymbol{p} \leftarrow 0$, $vj \leftarrow 1$
2: **for** $j = 0 \dots m$ **do**
3: $\quad \boldsymbol{q} \leftarrow \mathbf{0}$, $uk \leftarrow 1$
4: $\quad$ **for** $k = 0 \dots n$ **do**
5: $\quad\quad \boldsymbol{q} \leftarrow (1-u) \cdot \boldsymbol{q} + uk \cdot \boldsymbol{b}_{kj}$
6: $\quad\quad uk \leftarrow uk \cdot u \cdot \frac{n-k}{k+1}$
7: $\quad$ **end for**
8: $\quad \boldsymbol{p} \leftarrow (1-v) \cdot \boldsymbol{p} + vj \cdot \boldsymbol{q}$
9: $\quad vj \leftarrow vj \cdot v \cdot \frac{m-j}{j+1}$
10: **end for**
11: **return** $\boldsymbol{p}$

A degree $(n,m) \in \mathbb{N}^+ \times \mathbb{N}^+$ tensor product Bézier surface defined by a grid of control points $\boldsymbol{b}_{ij}$, $i = 0, \dots, n$, $j = 0, \dots, m$ is

$$\boldsymbol{b}(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} \boldsymbol{b}_{ij} B_i^n(u) B_j^m(v) \,. \tag{56}$$

Adapting the LTCS evaluation strategy is listed in Algorithm 9. It requires $\Theta(d)$ storage and $\Theta(n \cdot m \cdot d)$ operations.

Bézier triangles are arguably a more natural generalization of

© 2026 The Author(s).
Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.

Algorithm 10 TriangleLTCS($\{\mathbf{b}_{ijk}\}_{i+j+k=n}, u, v, w$)

```
1:  p = 0, c = 1, vj = 1
2:  for j = 0...n do
3:      q ← 0, ui = 1
4:      for i = 0...n − j do
5:          k = n − i − j
6:          if i = 0 then
7:              if j = 0 then c ← 1 else c ← c · (n−j+1)/j
8:          else
9:              c ← c · (k+1)/i
10:         end if
11:         q ← q · w + b_ijk · ui · c
12:         ui ← ui · u
13:     end for
14:     p ← p + vj · q
15:     vj ← vj · v
16: end for
```

Bézier curves to the surface setting. Instead of imposing a tensor product structure on the domain, it relies on 2D barycentric formulations. Consequently, its parametrization is also phrased in terms of $u, v, w \in \mathbb{R}$ barycentric coordinates, i.e., $u+v+w=1$. Bézier triangles are symmetric in the sense that they poses a singular degree. A point on a degree $n \in \mathbb{N}^+$ Bézier triangle is computed as

$$\boldsymbol{p}(u,v,w) = \sum_{i+j+k=n} \binom{n}{ijk} \cdot u^i \cdot v^j \cdot w^k \cdot \boldsymbol{b}_{ijk} \tag{57}$$

$$= \frac{n!}{i!j!k!} \cdot u^i \cdot v^j \cdot w^k \cdot \boldsymbol{b}_{ijk} \,, \tag{58}$$

where $\binom{n}{ijk}$ are the trinomial coefficients and $\boldsymbol{b}_{ijk} \in \mathbb{R}^d$ form the triangular grid of control points, $i+j+k=n$.

Our algorithm listed in Algorithm 10 evaluates a point on a Bézier triangle in linear time with respect to the number of control points, and constant storage. It uses a row-first linearization of the control data triangular grid.

6.2. Derivative Evaluation

Similarly to curves, one may apply the LTCS scheme directly to the appropriate forward differences of control points and multiply by the direction-dependent factor to obtain derivatives. Extending this to higher order derivatives with a more efficient hybrid LTCS and de Casteljau mixture, as in Algorithm 6, is subject to future work.

7. Test Results

We implemented the numerical precision and runtime performance tests in C++, optimized for the MSVC 17 compiler. The source code with the tests and a demo application are available at `https://github.com/valasekg/LTCSBezier`.

7.1. Numerical Precision

We ran precision tests on random real-valued univariate $b(t): \mathbb{R} \to \mathbb{R}$ Bézier curves. We generated curves of various degrees with random control points in the range of $[-1,1]$, drawn from a uniform random distribution. In numerically sensitive cases, normalization to the $[-1,1]$ control point range is advisable but we caution against doing so by division using the largest magnitude control data. Instead, normalize by the next larger power of two, to avoid an unnecessary rounding.

For each degree, we generated 32 curves and evaluated them at 256 and 257 equidistant parameters from $[0,1]$. This was chosen because $t_i = \frac{i}{255}$, $i = 0,\dots,255$ is not representable in floats, expect for $t_0 = 0, t_{255} = 1$. This is how we simulated the case of non-exact evaluation parameters. On the other hand, every $t_i = \frac{i}{256}$, $i = 0,\dots,256$ is representable, even with half precision floats. This is the exact evaluation parameter error inspection scenario.

The ground truth was computed with exact rational arithmetics, using the Rational library of Boost [Boo15]. The tests on half precision floats was carried out using AMD's `half_float` library [Adv25]. The tests were run on an Intel i9-13900K CPU.

We ran two tests. The first aggregated absolute errors from degree 2 to 5 curves. These degrees are more frequently encountered in computer graphics applications. The degree 5 cap was chosen such that we can benchmark against Seiler' s method [Yuk24]. The results for binary32 are listed in Table 1.

The second suite of tests aggregated results on curves from degrees 2 to 10. While SVG and vector graphics databases are dominated by cubic Bézier curves, higher degrees are often found in geometric modeling and CAD use cases. Achieving general second (G^2), third (G^3), and fourth (G^4) order geometric continuity requires degree 5, 7, and 9 curves, respectively. These are especially relevant for Class-A surfacing, common in premium automotive and consumer products, where G^2 is a baseline with G^3 and sometimes higher are required for aesthetic or functional purposes.

Both tests that we show here used inexact control points and inexact evaluation parameters (ICIP). This is the numerically most challenging scenario. The binary16 case shows that the Woźny-Chudy algorithm possesses remarkable empirical numerical stability. This also suggests that the error bounds derived for it in [FRH24] are overly conservative. We assume that this stability is due to the explicit handling of the two evaluation intervals $[0,0.5]$ and $[0.5,1]$. The LTCS methods are usually close but behind the de Casteljau and Woźny-Chudy methods on binary16 and binary32, however, on double precision numbers LTCS variants are up to par. Note that statistics alone do not allow us to observe local behaviors. To this end, Figure 3 illustrates the average absolute error as a function of evaluation parameter in the ICIP case. Note how the nested FMA formulations improve accuracy over $[0,0.5]$.

The ICIP is the most adversarial case out of the four. The exact control points, exact parameters (ECEP) scenario is one that is just as practical, which also changes the ranking of the methods. Figure 2 shows that the unrolled LTCS variant outperforms the other methods in accuracy on halfs.

7.2. Performance

For CPU performance measurements, we used Google Benchmark [Goo25]. We measured the processing time for a single polynomial

© 2026 The Author(s).
Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.

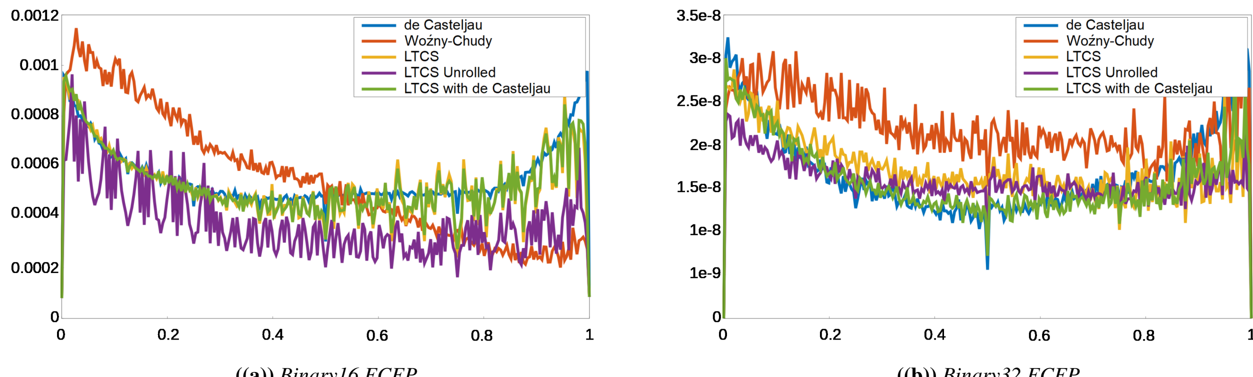
ICIP	de Casteljau	Woźny-Chudy	Seiler	LTCS	LTCS unrolled	LTCS with de Casteljau
mean	$\mathbf{2.51 \cdot 10^{-8}}$	$\mathbf{2.51 \cdot 10^{-8}}$	$2.99 \cdot 10^{-8}$	$2.57 \cdot 10^{-8}$	$2.56 \cdot 10^{-8}$	$2.52 \cdot 10^{-8}$
median	$\mathbf{1.90 \cdot 10^{-8}}$	$1.91 \cdot 10^{-8}$	$2.17 \cdot 10^{-8}$	$1.92 \cdot 10^{-8}$	$1.92 \cdot 10^{-8}$	$1.91 \cdot 10^{-8}$
max	$\mathbf{1.81 \cdot 10^{-7}}$	$2.21 \cdot 10^{-7}$	$3.26 \cdot 10^{-7}$	$2.03 \cdot 10^{-7}$	$2.39 \cdot 10^{-7}$	$1.83 \cdot 10^{-7}$

Table 1: *Absolute errors for binary32 format algorithm runs, taken on random curves from degree 2 to 5. This comparison uses the non-FMA variant of the evaluation algorithms. The ground truth was computed with exact rational arithmetics. This is the inexact control points, inexact parameters (ICIP) use case. Neither the control points, nor the evaluation parameters can be represent in this floating point format.*

Binary16	Degree 2 to 10	de Casteljau	Woźny-Chudy	LTCS	LTCS unrolled	LTCS with de Casteljau
No FMA	mean	$7.18 \cdot 10^{-04}$	$\mathbf{5.55 \cdot 10^{-04}}$	$7.11 \cdot 10^{-04}$	$5.81 \cdot 10^{-04}$	$5.64 \cdot 10^{-04}$
	median	$5.09 \cdot 10^{-04}$	$4.04 \cdot 10^{-04}$	$4.97 \cdot 10^{-04}$	$\mathbf{4.01 \cdot 10^{-04}}$	$4.17 \cdot 10^{-04}$
	max	$8.55 \cdot 10^{-03}$	$4.77 \cdot 10^{-03}$	$7.85 \cdot 10^{-03}$	$7.18 \cdot 10^{-03}$	$\mathbf{4.50 \cdot 10^{-03}}$
SubFMA	mean	$7.18 \cdot 10^{-04}$	$\mathbf{5.55 \cdot 10^{-04}}$	$7.11 \cdot 10^{-04}$	$5.81 \cdot 10^{-04}$	$7.12 \cdot 10^{-04}$
	median	$5.09 \cdot 10^{-04}$	$4.04 \cdot 10^{-04}$	$4.97 \cdot 10^{-04}$	$\mathbf{4.01 \cdot 10^{-04}}$	$5.01 \cdot 10^{-04}$
	max	$8.55 \cdot 10^{-03}$	$\mathbf{4.77 \cdot 10^{-03}}$	$7.85 \cdot 10^{-03}$	$7.18 \cdot 10^{-03}$	$7.85 \cdot 10^{-03}$
TwoFMA	mean	$5.69 \cdot 10^{-04}$	$\mathbf{4.06 \cdot 10^{-04}}$	$8.10 \cdot 10^{-04}$	$6.38 \cdot 10^{-04}$	$5.64 \cdot 10^{-04}$
	median	$4.25 \cdot 10^{-04}$	$\mathbf{3.03 \cdot 10^{-04}}$	$6.13 \cdot 10^{-04}$	$4.82 \cdot 10^{-04}$	$4.17 \cdot 10^{-04}$
	max	$4.46 \cdot 10^{-03}$	$\mathbf{4.02 \cdot 10^{-03}}$	$6.20 \cdot 10^{-03}$	$6.00 \cdot 10^{-03}$	$4.50 \cdot 10^{-03}$
Binary32	**Degree 2 to 10**	**de Casteljau**	**Woźny-Chudy**	**LTCS**	**LTCS unrolled**	**LTCS with de Casteljau**
No FMA	mean	$3.01 \cdot 10^{-08}$	$\mathbf{2.59 \cdot 10^{-08}}$	$3.08 \cdot 10^{-08}$	$3.15 \cdot 10^{-08}$	$3.02 \cdot 10^{-08}$
	median	$2.17 \cdot 10^{-08}$	$\mathbf{1.97 \cdot 10^{-08}}$	$2.19 \cdot 10^{-08}$	$2.25 \cdot 10^{-08}$	$2.16 \cdot 10^{-08}$
	max	$3.23 \cdot 10^{-07}$	$\mathbf{2.81 \cdot 10^{-07}}$	$4.18 \cdot 10^{-07}$	$3.61 \cdot 10^{-07}$	$3.70 \cdot 10^{-07}$
SubFMA	mean	$2.86 \cdot 10^{-08}$	$\mathbf{2.38 \cdot 10^{-08}}$	$2.92 \cdot 10^{-08}$	$3.03 \cdot 10^{-08}$	$2.88 \cdot 10^{-08}$
	median	$2.05 \cdot 10^{-08}$	$\mathbf{1.83 \cdot 10^{-08}}$	$2.09 \cdot 10^{-08}$	$2.16 \cdot 10^{-08}$	$2.06 \cdot 10^{-08}$
	max	$3.19 \cdot 10^{-07}$	$\mathbf{2.07 \cdot 10^{-07}}$	$3.29 \cdot 10^{-07}$	$3.61 \cdot 10^{-07}$	$3.23 \cdot 10^{-07}$
TwoFMA	mean	$\mathbf{1.99 \cdot 10^{-08}}$	$2.34 \cdot 10^{-08}$	$2.23 \cdot 10^{-08}$	$2.25 \cdot 10^{-08}$	$2.09 \cdot 10^{-08}$
	median	$\mathbf{1.48 \cdot 10^{-08}}$	$1.78 \cdot 10^{-08}$	$1.66 \cdot 10^{-08}$	$1.67 \cdot 10^{-08}$	$1.55 \cdot 10^{-08}$
	max	$2.14 \cdot 10^{-07}$	$2.29 \cdot 10^{-07}$	$2.49 \cdot 10^{-07}$	$\mathbf{2.00 \cdot 10^{-07}}$	$2.61 \cdot 10^{-07}$
Binary64	**Degree 2 to 10**	**de Casteljau**	**Woźny-Chudy**	**LTCS**	**LTCS unrolled**	**LTCS with de Casteljau**
No FMA	mean	$5.7624 \cdot 10^{-17}$	$\mathbf{5.2809 \cdot 10^{-17}}$	$6.0041 \cdot 10^{-17}$	$6.1103 \cdot 10^{-17}$	$5.8224 \cdot 10^{-17}$
	median	$\mathbf{3.7517 \cdot 10^{-17}}$	$3.8177 \cdot 10^{-17}$	$4.0222 \cdot 10^{-17}$	$4.1363 \cdot 10^{-17}$	$3.7987 \cdot 10^{-17}$
	max	$8.9380 \cdot 10^{-16}$	$9.3196 \cdot 10^{-16}$	$8.2979 \cdot 10^{-16}$	$\mathbf{8.2093 \cdot 10^{-16}}$	$9.2306 \cdot 10^{-16}$
SubFMA	mean	$5.5064 \cdot 10^{-17}$	$\mathbf{4.8714 \cdot 10^{-17}}$	$5.6910 \cdot 10^{-17}$	$5.8866 \cdot 10^{-17}$	$5.5512 \cdot 10^{-17}$
	median	$3.5386 \cdot 10^{-17}$	$\mathbf{3.5199 \cdot 10^{-17}}$	$3.7981 \cdot 10^{-17}$	$3.9461 \cdot 10^{-17}$	$3.6334 \cdot 10^{-17}$
	max	$8.9380 \cdot 10^{-16}$	$9.3196 \cdot 10^{-16}$	$\mathbf{8.2093 \cdot 10^{-16}}$	$\mathbf{8.2093 \cdot 10^{-16}}$	$8.7301 \cdot 10^{-16}$
TwoFMA	mean	$\mathbf{4.1201 \cdot 10^{-17}}$	$4.8472 \cdot 10^{-17}$	$4.5825 \cdot 10^{-17}$	$4.6372 \cdot 10^{-17}$	$4.2963 \cdot 10^{-17}$
	median	$\mathbf{2.7715 \cdot 10^{-17}}$	$3.4585 \cdot 10^{-17}$	$3.1878 \cdot 10^{-17}$	$3.2213 \cdot 10^{-17}$	$2.8992 \cdot 10^{-17}$
	max	$7.6396 \cdot 10^{-16}$	$9.3196 \cdot 10^{-16}$	$\mathbf{7.5068 \cdot 10^{-16}}$	$8.2093 \cdot 10^{-16}$	$8.2093 \cdot 10^{-16}$

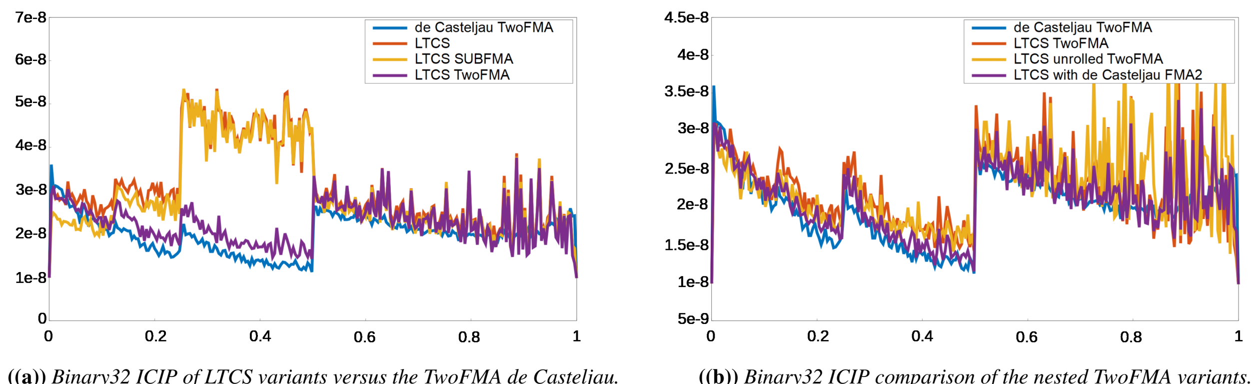
Table 2: *Aggregated accuracy measurements on degree 2 to 10 random Bézier curves against an exact rational ground truth. Both the control points and the evaluation parameters were inexact.*

© 2026 The Author(s).
Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.



((a)) *Binary16 ECEP* **((b))** *Binary32 ECEP*

Figure 2: *Binary16 (left) and binary32 (right) average absolute error (Y axis) plots for the exact control point, exact evaluation parameter (ECEP) use case. The X axis is the polynomial evaluation parameter. Tests were taken on random Bézier curves from degrees 2 to 10.*



((a)) *Binary32 ICIP of LTCS variants versus the TwoFMA de Casteljau.* **((b))** *Binary32 ICIP comparison of the nested TwoFMA variants.*

Figure 3: *Average absolute error (Y axis) trends as a function of polynomial evaluation parameter (X axis). Note that the nested FMA TwoFMA variant mitigates the large errors over* $[0, 0.5]$*. Tests were taken on random Bézier curves from degrees 2 to 10.*

Degree	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
de Casteljau	19.9	26.2	31.4	33.7	39.9	44.5	50	59.4	68.4
de Casteljau SubFMA	18.8	26.4	29.8	33	38.5	42	46.5	51.6	61.4
de Casteljau TwoFMA	20.1	25.5	29.3	35.3	38.5	43.3	48.8	51.6	59.4
Seiler	9.63	9.42	13.5	15.1					
Seiler TwoFMA	9.59	9.52	16.5	15.3					
Woźny-Chudy	6.14	9.21	12.8	16.7	21.8	25.7	31.1	36.9	41.7
Woźny-Chudy SubFMA	5	7.5	10	13.8	17.6	22	26.8	31.4	36.1
Woźny-Chudy TwoFMA	6.28	8.54	11.2	15.3	18.8	23.5	27.3	31.5	36.6
LTCS	5.72	7.11	8.16	9.84	11	12.3	13.8	15	16.4
LTCS SubFMA	**4.71**	5.58	6.7	7.5	8.16	9.21	10.5	10.3	11.5
LTCS TwoFMA	6.42	7.5	9.21	10.5	11.7	14.1	14.4	15.3	28.6
Unrolled LTCS	6.28	6.42	7.67	8.02	9	9.21	11	11.7	12.8
Unrolled LTCS SubFMA	5.31	**5.16**	**6.14**	**6**	**6.98**	**8.16**	**9.21**	**10.3**	**10.7**
Unrolled LTCS TwoFMA	6.98	7.32	9	9.63	11	22.5	24.6	14.1	15
LTCS with de Casteljau	5.78	7.39	8.54	10	12	13.8	16.9	19.9	22.5
LTCS with de Casteljau SubFMA	**4.71**	5.72	6.56	8.16	9.63	11.2	13.2	30.1	34.4
LTCS with de Casteljau TwoFMA	6	7.5	8.58	10.3	11.7	21	16	29.2	33.5

Table 3: *Performance measurements on the CPU, using Google Benchmark to determine the execution time for each method. Timings are in nanoseconds, taken on an Intel i9-13900K.*

© 2026 The Author(s).

evaluation. The results are shown in Table 3, reporting the CPU column from the benchmark results. The degrees of the polynomials were known at compile time, however, note that this does not mean that the compiler always generated unrolled code. Indeed, there are currently no means to control this behavior of the compiler and it, contrary to intuition, retains runtime loops more frequently than one would suspect. Indeed, for example the sudden jump in the LTCS TwoFMA timings from degree 9 to 10 imply that the compiler stopped unrolling the evaluation loop, resulting in significant performance loss. Please consult the source code in the Supplementary Material to inspect what measures we have taken to alleviate these issues. The code was compiled with MSVC 17.13.1. Timings were taken on an `/O2` build with `/arch:AVX2` to enable the generation of FMA machine instructions.

Overall, the unrolled SubFMA variants of the LTCS technique performed the best on the CPU. Oftentimes, the nested FMA methods performed worse than the non-FMA versions, however, as shown in the numerical tests, these do improve numerical behavior. In general, the ladder type algorithms outperform the de Casteljau algorithm significantly. This, combined with their comparable numerical behavior, suggest that it is advisable to rely on these techniques for Bézier construct evaluations in general use cases.

On the GPU, we had two different scenarios. One for rendering algebraic surfaces and another for displaying tensor product surfaces. For the latter, we constructed random Bézier surfaces and rendered them at 500×500 samples. The position of each vertex was obtained by evaluating the Bézier surface. The test application used NVIDIA Falcor [KCK*22]. We have taken GPU traces with NVIDIA Nsight and report the GPU execution time for the draw command that contained the ray march. For the volume rendering scenario, we fit univariate polynomials to algebraic surfaces, showcased in Figure 4. Then we ray march using these univariate Bézier curves. The performance measurements for volumes and surfaces is shown in Table 4. Both of the GPU measurements were taken on an Ampere generation NVIDIA RTX 4500 GPU. For the implicit surface rendering scenario the LTCS methods possess a significant edge over the de Casteljau and Woźny-Chudy methods. This is most likely due to the fact that we have taken up to 2048 steps along each ray. The simpler tessellation scenario only measures the vertex processing time of a 500×500 grid and it shows smaller difference between the LTCS methods and that of Chudy.

Volume	**Diabolo**	**Shape 1**	**Barth**
de Casteljau	160.9	180.5	350.0
Woźny-Chudy	18.9	19.8	29.9
LTCS	8.8	8.1	12.7
LTCS Estrin	9.6	7.3	11.6
Seiler	38.6	28.2	N/A

Surface	**2x2**	**3x3**	**4x4**	**5x5**
de Casteljau	0.38	1.64	2.68	4.10
Woźny-Chudy	0.15	0.15	0.20	0.28
LTCS	0.15	0.15	0.18	0.23

Table 4: *Render times for ray marching $f(x,y,z) = 0$ volumes and vertex-shader based display of randomized $n \times n$ tensor product Bézier surfaces. Measurements were taken on an NVIDIA 4500 RTX (Ampere) GPU at 1280×720 resolution. Timings are in milliseconds, inspected with NVIDIA Nsight. For surfaces, there was no measurable difference between the unrolled and the default LTCS variants.*

8. Conclusions

Building on Warren's ladder scheme, we formulated linear-time, constant-storage (LTCS) evaluators that keep the working set constant while mapping cleanly to FMA-capable hardware and opening up tabularization opportunities. Our contribution is to extend this ladder-style approach in both scope and analysis: beyond point evaluation, we cover derivative and surfaces, and we provide explicit forward-error bounds for the LTCS family, including a nested FMA realization.

As a computational primitive, we compared four lerp realizations and highlighted the fact that the ubiquitous subtract-then-FMA form can both fail endpoint reconstruction and overflow in half precision, while the two-FMA form avoids these pathologies. These, however, may not be evident from the rounding error analysis on its own, underlining the importance of empirical tests and other analysis techniques.

Overall, all investigated algorithms perform comparable in terms of numerical precision. However, in execution time the ladder-type LTCS algorithms offer a significant improvement on the CPU over both the de Casteljau and Woźny-Chudy algorithm. On the GPU, this difference was also observable in the ray marching scenario, however, the surface tests showed a more similar performance between the LTCS and the Chudy-Woźny formulation.

For numerically demanding situations, we recommend the application of the nested FMA technique. If performance is of utmost importance, the subtraction-then-FMA formulation is a significant improvement with generally better error statistics than the non-FMA formulations. In cases when the aforementioned weaknesses of this technique are of no concern, these could be considered good performance-first defaults.

References

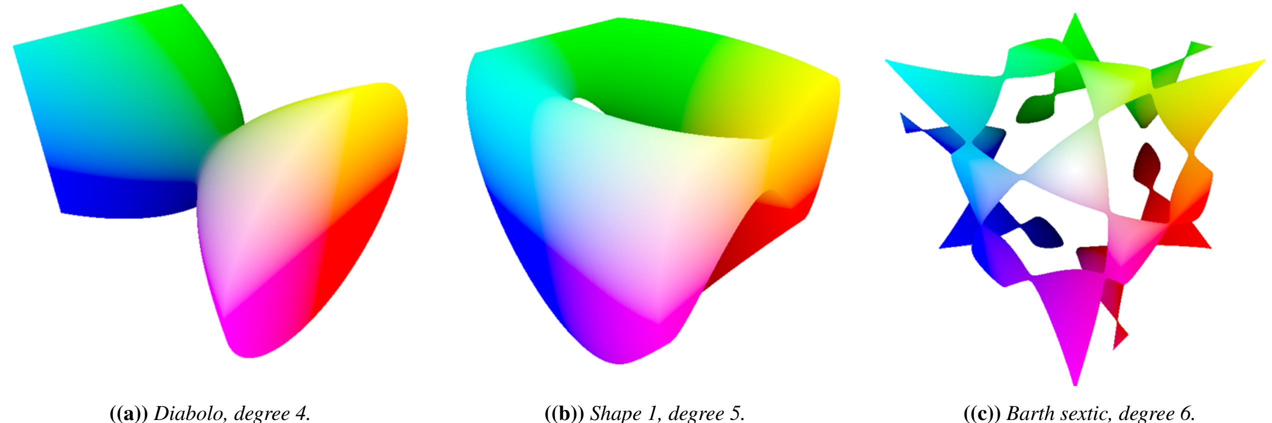
[25] *RDNA4 Instruction Set Architecture: Reference Guide*. Reference Guide. Santa Clara, CA: Advanced Micro Devices, Inc., Apr. 7, 2025. URL: https://www.amd.com/content/dam/amd/en/documents/radeon-tech-docs/instruction-set-architectures/rdna4-instruction-set-architecture.pdf (visited on 09/25/2025) 4.

[Adv25] Advanced Micro Devices, Inc. *half: Half-precision Floating Point Library*. https://github.com/ROCm/half. Version 1.12.0. GitHub repository; MIT License. 2025. (Visited on 09/26/2025) 10.

[Boo15] Boost. *Boost C++ Libraries*. http://www.boost.org/. Last accessed 2015-06-30. 2015 10.

[CW24] Chudy, Filip and Woźny, Paweł. "Fast evaluation of derivatives of Bézier curves". *Computer Aided Geometric Design* 109 (2024), 102277. ISSN: 0167-8396. DOI: https://doi.org/10.1016/j.cagd.2024.102277 2.

© 2026 The Author(s).
Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.



((a)) *Diabolo, degree 4.* **((b))** *Shape 1, degree 5.* **((c))** *Barth sextic, degree 6.*

Figure 4: *Shapes used for testing on GPU. Each shape is given by a polynomial and was rendered using equidistant ray marching algorithm. We fit a 1D Bézier curve along each ray and evaluate this polynomial proxy in lieu of the defining implicit function. The test shapes were adapted from [Gal25] and their equations are as follows:*

$$f_{\text{Diabolo}}(x,y,z) = (y^2+z^2)^2 - x^2 = 0 \,,$$
$$f_{\text{Shape 1}}(x,y,z) = 2y(y^2-3x^2)(1-z^2) + (x^2+y^2)^2 - (9z^2-1)(1-z^2) = 0 \,,$$
$$f_{\text{Barth}}(x,y,z) = -4(2.618x^2-y^2)(2.618y^2-z^2)(2.618z^2-x^2) + 4.236(x^2+y^2+z^2-1)^2 = 0 \,.$$

[DMP23] DELGADO, J., MAINAR, E., and PEÑA, J.M. "On the accuracy of de Casteljau-type algorithms and Bernstein representations". *Computer Aided Geometric Design* 106 (2023), 102243. ISSN: 0167-8396. DOI: https://doi.org/10.1016/j.cagd.2023.102243 2.

[Est60] ESTRIN, GERALD. "Organization of computer systems: the fixed plus variable structure computer". *Papers Presented at the May 3-5, 1960, Western Joint IRE-AIEE-ACM Computer Conference*. IRE-AIEE-ACM '60 (Western). San Francisco, California: Association for Computing Machinery, 1960, 33–40. ISBN: 9781450378697. DOI: 10.1145/1460361.1460365 6.

[Far01] FARIN, GERALD. *Curves and surfaces for CAGD: a practical guide*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. ISBN: 1558607374 1–3, 8.

[FG96] FAROUKI, RIDA T. and GOODMAN, TIM N. T. "On the optimal stability of the Bernstein basis". *Math. Comput.* 65.216 (1996), 1553–1566. DOI: 10.1090/S0025-5718-96-00759-4 1.

[FRH24] FUDA, CHIARA, RAMANANTOANINA, ANDRIAMAHENINA, and HORMANN, KAI. "A comprehensive comparison of algorithms for evaluating rational Bézier curves". *Dolomites Research Notes on Approximation* 17.3 (Sept. 2024), 56–79 2, 3, 10.

[Gal25] GALLERY, IMPLICIT. *Implicit Surfaces*. 2025. URL: http://xrt.wikidot.com/gallery:implicit 14.

[Gie12] GIESEN, FABIAN. *Linear interpolation past, present and future*. *The ryg blog*. Accessed 2025-09-13. 2012. URL: https://fgiesen.wordpress.com/2012/08/15/linear-interpolation-past-present-and-future/ 4.

[Goo25] GOOGLE. *Google Benchmark: A microbenchmark support library*. https://github.com/google/benchmark. Version v1.9.4. 2025. (Visited on 09/26/2025) 10.

[Gus15] GUSTAFSON, JOHN L. *The End of Error: Unum Computing*. 1st. OCLC: (OCoLC)1019902047; (OCoLC)on1019902047. 1 online resource (425 p.). Language: English. Boca Raton; London; New York: CRC Press, Taylor & Francis Group, 2015. ISBN: 9781482239874, 1482239876, 9781482239867, 1482239868, 9781322635231, 1322635234 1.

[Her18] HERMES, DANNY. "A Curious Case of Curbed Condition". (June 2018). DOI: 10.48550/arXiv.1806.05145. arXiv: 1806.05145 [math.NA] 5.

[Hig02] HIGHAM, NICHOLAS J. *Accuracy and Stability of Numerical Algorithms*. 2nd. USA: Society for Industrial and Applied Mathematics, 2002. ISBN: 0898715210 3.

[KCK*22] KALLWEIT, SIMON, CLARBERG, PETRIK, KOLB, CRAIG, et al. *The Falcor Rendering Framework*. Aug. 2022. URL: https://github.com/NVIDIAGameWorks/Falcor 13.

[PT97] PIEGL, LES and TILLER, WAYNE. *The NURBS book (2nd ed.)* Berlin, Heidelberg: Springer-Verlag, 1997. ISBN: 3540615458 2.

[SV86] SCHUMAKER, LARRY L. and VOLK, WOLFGANG. "Efficient evaluation of multivariate polynomials". *Computer Aided Geometric Design* 3.2 (1986), 149–154. ISSN: 0167-8396. DOI: https://doi.org/10.1016/0167-8396(86)90018-X 2.

[War95] WARREN, J. "An Efficient Algorithm for Evaluating Polynomials in the Pólya Basis". *Geometric Modelling*. Ed. by HAGEN, H., FARIN, G., and NOLTEMEIER, H. Vienna: Springer Vienna, 1995, 357–361. ISBN: 978-3-7091-7584-2 2, 3.

[WC20] WOŹNY, PAWEŁ and CHUDY, FILIP. "Linear-time geometric algorithm for evaluating Bézier curves". *Computer-Aided Design* 118 (2020), 102760. ISSN: 0010-4485. DOI: https://doi.org/10.1016/j.cad.2019.102760 2.

[Yuk24] YUKSEL, CEM. "Seiler's Interpolation for Evaluating Polynomial Curves". *ACM SIGGRAPH 2024 Talks*. SIGGRAPH 2024. New York, NY, USA: ACM, July 2024. ISBN: 979-8-4007-0515-1/24/07. DOI: 10.1145/3641233.3664331 2, 10.

© 2026 The Author(s).