BIT Numerical Mathematics (2023) 63:31
https://doi.org/10.1007/s10543-023-00975-x

Fast floating-point filters for robust predicates

Tinko Bartels[1] · Vissarion Fisikopoulos[2] · Martin Weiser[3]

Received: 25 August 2022 / Accepted: 25 April 2023 / Published online: 17 May 2023

Abstract
Geometric predicates are at the core of many algorithms, such as the construction of Delaunay triangulations, mesh processing and spatial relation tests. These algorithms have applications in scientific computing, geographic information systems and computer-aided design. With floating-point arithmetic, these geometric predicates can incur round-off errors that may lead to incorrect results and inconsistencies, causing computations to fail. This issue has been addressed using a combination of exact arithmetic for robustness and floating-point filters to mitigate the computational cost of exact computations. The implementation of exact computations and floating-point filters can be a difficult task, and code generation tools have been proposed to address this. We present a new C++ meta-programming framework for the generation of fast, robust predicates for arbitrary geometric predicates based on polynomial expressions. We combine and extend different approaches to filtering, branch reduction, and overflow avoidance that have previously been proposed. We show examples of how this approach produces correct results for data sets that could lead to incorrect predicate results with naive implementations. Our benchmark results demonstrate that our implementation surpasses state-of-the-art implementations.

Keywords Floating-point arithmetic · Floating-point filter · Roundoff error · Computational geometry

Communicated by Elisabeth Larsson.

✉ Tinko Bartels
t.bartels@tu-berlin.de

Vissarion Fisikopoulos
vfisikop@di.uoa.gr

Martin Weiser
weiser@zib.de

[1] Technische Universität Berlin, Straße des 17. Juni 135, 10623 Berlin, Germany

[2] Department of Informatics & Telecommunications, National & Kapodistrian University of Athens, Athens, Greece

[3] Zuse Institute Berlin, Takustr. 9, 14195 Berlin, Germany

Mathematics Subject Classification 65G50 · 68U05

1 Introduction

Basic geometric predicates, such as computing the orientation of a triangle or testing if a point is inside a circle, are at the core of many computational geometry algorithms such as convex hull, Delaunay triangulation and mesh generation [5]. Interestingly, those predicates also appear in geospatial computations such as topological spatial relations that determine the relationship among geometries. Those operations are fundamental in many Geographic Information System (GIS) applications. If evaluated with floating-point arithmetic, these computations can incur round-off errors that can, due to the ill-conditioning of discrete decisions, lead to incorrect results and inconsistencies, causing computations to fail [20].

Among other applications, Delaunay triangulations are important for the construction of triangular meshes [19, 31] and Triangulated Irregular Networks (TIN) [21]. Predicate failures in the underlying Delaunay triangulation may lead to suboptimal mesh quality and cause invalid triangulations or termination failure [30].

Robust geometric predicates can also be used in spatial predicates to guarantee correct results for floating-point geometries. Spatial predicates are used to determine the relationship between geometries and have applications in spatial databases and GIS applications. Examples of such predicates include intersects, crosses, touches, or within. Using non-robust spatial predicates, for example, a point that lies close to the shared edge of two triangles can be found to be within both or neither of them, which is not only incorrect but also inconsistent and violates basic assumptions on partitioned spaces.

Exact computations can guarantee correct results for floating-point input but are very slow for practical purposes. Since predicates are usually ill-conditioned only on a set of measure zero and extremely well-conditioned everywhere else, an adaptive evaluation can improve average performance by using exact arithmetic only if an a priori error estimate can not guarantee correctness for the faster, approximate computations. In other words, the expensive computations are filtered out by using those error estimates.

Now, the main question is how difficult it is to compute those error estimates. There are several approaches that provide a trade-off in efficiency and accuracy of error estimation. The three main types of filters are static, semi-static and dynamic.

In the first case, the error is pre-computed very efficiently using a priori bounds on the input and typically attains very low accuracy. In semi-static filters, the error estimation depends on the input. They are somewhat slower than static filters but improve on the accuracy and require no a priori bounds on the input. The slowest and most accurate are dynamic filters using floating-point interval arithmetic to better control the error and achieve fewer filter failures.

Previous work. Many techniques have been proposed in the past for efficient and robust arithmetic. In his seminal paper [30], Shewchuk introduced robust, adaptive implementations for orientation-, incircle- and insphere-predicates that can be used, for example, in the construction of Delaunay triangulations. They use a sequence of semi-static filters of ever-increasing accuracy. The phases are attempted in order, each phase building on the result from the previous one until the correct sign is obtained. On the other hand, efficient dynamic filters are proposed in [6]. For Delaunay triangulations, in [11] they propose a set of efficient static and semi-static filters and experimentally compare them with several alternatives including [30]. Meyer and Pion develop FPG [24], a general-purpose code analyzer and generator for static filtered predicates. The generated filters, however, include multiple branch instructions, which was found in [26] to cause suboptimal performance.

Nanevski et al. extend Shewchuk's method to arbitrary polynomial expressions and implement an expression compiler that takes a function and produces a predicate, consisting of semi-static filters and an exact stage that computes the sign of the source function at any given floating point arguments [25]. Their filters, however, are not robust with respect to overflow or underflow.

In [8], Burnikel et al. present EXPCOMP, a C++ wrapper class and an expression compiler that generates fast semi-static filters for predicates involving the operations $+, -, \cdot, /, \sqrt{\cdot}$, which include arbitrary polynomials and handle all kinds of floating-point exceptions. In their benchmarks, they found a 25-30% runtime overhead for their C++ wrapper class when compared with their expression compiler and their error bound constants are comparatively pessimistic (see Subsection 4.3 for an example).

More recently, Ozaki et al. developed an improved static filter as well as a new semi-static filter for the 2D orientation predicate, where the latter also handles floating-point exceptions such as overflow and underflow [26]. This approach yields a close-to-optimal error bound constant, however, it is not designed for arbitrary polynomial predicates.

Regarding non-linear geometries, there is work on filters for circular arcs [10]. Moreover, robust predicates could be extended to provide robust constructions such as points of intersection of linestrings [2]. Recently, GPU implementations of robust predicates have been presented, providing a constant (3 to 4 times) speedup over standard CPU implementations [9, 27].

In [13], they employ dynamic determinant computations to speed up the computation of sequences of determinants that appear in high-dimensional (typically more than 6) geometric algorithms such as convex hull and volume computation.

In [3], the authors present a C++ metaprogramming framework that produces fast, robust predicates and illustrate how GIS applications can benefit from it.

Our contribution. The contribution of this paper is three-fold. First, we present an algorithm that generates semi-static or static filters for robust predicates based on arbitrary polynomials. These filters are shown to be valid for all input numbers, regardless of range issues such as overflow or underflow. They also require only a single comparison and can therefore be evaluated encountering only a single, easy-to-predict branch. To the best of our knowledge, this is the first filter design combining generality, range robustness, and branch efficiency.

Second, we present a new implementation based on C++ meta-programming techniques that produces fast, robust code at compile-time for predicates. It is extensible, based on the C++ library Boost.Geometry [14] and publicly available at [4].

The main advantage of our implementation is the ability to automatically generate filters for arbitrary polynomial predicates without relying on external code generation tools. In addition, it can be complemented seamlessly with manual handcrafted filters, as illustrated by the use of our axis-aligned filter for the incircle predicate (see example 8).

Last, we perform an experimental analysis of the generated filters as well as a comparison with the state of the art. We perform benchmarks for 2D Delaunay triangulation, 3D Polygon Mesh processing and 3D Mesh refinement. The algorithms tested in the benchmarks make use of four different geometric predicates of different complexity. We show that our predicates outperform the state of the art libraries [7, 29] in all benchmark cases, which includes both synthetic and real data. Unlike Burnikel et al. in [8] we find no performance penalty for our C++ implementation over generated code.

2 Robust geometric predicates

In this section we review the basic concepts and notation necessary for presenting our filter design in Sect. 3 and implementation approach in Sect. 4.1.

2.1 Geometric predicates and robustness issues

In the context of this paper, we define geometric predicates to be functions that return discrete answers to geometric questions based on evaluating the sign of a polynomial. One example is the planar orientation predicate. Given three points $a, b, c \in \mathbb{R}^2$, it determines the location of c with respect to the straight line going through a and b by evaluating the sign of

$$p_{\text{orientation\_2}}(a, b) := \begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix} \tag{1}$$

For this definition of the orientation predicate, positive, zero, and negative determinants correspond to the locations left of the line, on the line and right of the line, respectively. This geometric predicate has applications in the construction of Delaunay triangulations, convex hulls, and in spatial predicates such as within for 2D points, lines or polygons.

While expression (1) always gives the correct answer in real arithmetic, this is not necessarily the case for floating-point arithmetic.

Definition 1 (*Floating-Point Number System*) For a given precision $p \in \mathbb{N}_{\geq 2}$ and minimum and maximum exponents $e_{\min}, e_{\max} \in \mathbb{Z}$ we define by

$$N_{p,e_{\min},e_{\max}} := \{(-1)^{\sigma}\left(1+\sum_{i=1}^{p-1} b_i 2^{-i}\right) 2^e \mid \sigma, b_1, \ldots, b_p \in \{0, 1\}, e_{\min} \leq e \leq e_{\max}\}$$

the set of *normalised binary floating-point numbers* and by

$$S_{p,e_{\min}} := \{(-1)^{\sigma} \left(\sum_{i=1}^{p-1} b_i 2^{-i} \right) 2^{e_{\min}} \mid \sigma, b_1, \ldots, b_p \in \{0, 1\}\}$$

the set of *subnormal binary floating-point numbers*.

For the remainder we will drop the parameters in the subscript. A *binary Floating-Point Number system* (FPN) is defined by $F := N \cup S \cup \{-\infty, \infty, \mathrm{NaN}\}$. For a number $a \in F$ given in the representation

$$(-1)^{\sigma} \left(1 + \sum_{i=1}^{p-1} b_i 2^{-i} \right) 2^{e} \text{ or } (-1)^{\sigma} \left(\sum_{i=1}^{p-1} b_i 2^{-i} \right) 2^{e_{\min}},$$

we call the tuple $\left(b_1, \ldots, b_{p-1}\right)$ *significand*. It is sometimes called *mantissa* in literature. The significand is called *even* if $b_{p-1} = 0$.

Definition 2 (*Rounding function*) For a given FPN F we define the *rounding-function* $\mathrm{rd} : \mathbb{R} \to F$ as follows

$$\mathrm{rd}\,(a) := \begin{cases} -\infty & a \leq -2^{e_{\max}} \left(2 - 2^{-p}\right) \\ \text{closest number to } a \text{ in } F & -2^{e_{\max}} \left(2 - 2^{-p}\right) < a < 2^{e_{\max}} \left(2 - 2^{-p}\right) \\ \infty & a \geq 2^{e_{\max}} \left(2 - 2^{-p}\right) \end{cases}$$

If there are two nearest numbers in F, the one with an even significand is chosen.

Remark 1 The above definition of subnormal numbers includes zero while zero is neither a normal nor subnormal number in the IEEE standard 754-2008 [18]. This deviation from the standard does not affect the rounding error analysis in this paper.

Next, we define some special quantities. By $\varepsilon := 2^{-p}$ we denote the *machine epsilon*, which is half of the difference between 1.0 and the next number in F, by $u_N := 2^{-e_{\min}}$ the smallest, positive, normalized number in F and by $u_S := 2^{-e_{\min}-p+1} = 2 \cdot \varepsilon \cdot u_N$ the smallest, positive, subnormal number in F.

Definition 3 (*Floating-point operations*) For a given FPN F and $a, b, c \in F \cap \mathbb{R}$ we define the floating-point operator $\circledcirc : F \times F \to F$ for each $\circ \in \{+, -, \cdot\}$ as

$$a \circledcirc b := \mathrm{rd}\,(a \circ b)\,.$$

and the Fused Multiply-Add (FMA) operator

$$\mathrm{FMA}\,(a, b, c) := \mathrm{rd}\,(ab + c)$$

If a zero is produced in the floating-point multiplication of two non-zero numbers, in an FMA operation with $ab + c \neq 0$ or a subnormal number is produced, this is called an *underflow*. If the result of an operation is ∞ or $-\infty$, this is called an *overflow*. These definitions are extended to arguments with NaN by setting the result to NaN and to infinities in the natural way with the following special cases set to NaN: $\infty \oplus -\infty$, $-\infty \ominus -\infty$, $\infty \ominus \infty$, $\pm\infty \odot 0$.

This definition is consistent with the IEEE standard 754-2008 [18] with the default rounding mode "roundTiesToEven". In [28], a number of error estimates for floating-point operations are given. In the following we use for $a, b, c \in F \cap \mathbb{R}$ and the unit in the first place,

$$\mathrm{ufp}\,(a) := \begin{cases} 0 & a = 0 \\ 2^{\lfloor \log_2 |a| \rfloor}, & \text{otherwise.} \end{cases}$$

It represents the value of the first digit in the significand of a number in floating-point representation (it can be defined the same way for numbers in $\mathbb{R}$). If no overflow occurs, it holds that

$$|a \circledcirc b - a \circ b| \leq \frac{1}{2}\varepsilon \cdot \mathrm{ufp}\,(a \circledcirc b) \leq \varepsilon\, |a \circledcirc b|\,. \tag{2}$$

In the case of underflow, addition and subtraction are exact. For multiplication, assuming no overflow occurs, [28] gives the error bound

$$|a \odot b - ab| = \varepsilon \cdot \mathrm{ufp}\,(a \odot b) + \eta$$

for some $\varepsilon, \eta \in \mathbb{R}$ such that $\epsilon \leq \varepsilon$, $\eta \leq u_S$ and $\varepsilon\eta = 0$. If no underflow occurs, i.e. $a \odot b \geq u_N = \frac{1}{2}\varepsilon^{-1}u_S$, then this implies

$$|a \odot b - ab| \leq \varepsilon\, |a \odot b|\,,$$

otherwise (if underflow occurs)

$$|a \odot b - ab| \leq \frac{1}{2}u_S,$$

and regardless of underflow

$$|a \odot b - ab| \leq \varepsilon\, (|a \odot b| \oplus u_N)\,. \tag{3}$$

We will use similar error bounds for FMA. Assuming no overflow or underflow it holds that

$$|\mathrm{FMA}\,(a, b, c) - ab + c| \leq \varepsilon\, |\mathrm{FMA}\,(a, b, c)|\,, \tag{4}$$

if underflow occurs then

$$|\mathrm{FMA}\,(a, b, c) - ab + c| \leq \frac{1}{2}u_S,$$

Table 1 Relationships of point $c = (0, -0.01)$ to polygon $\tilde{t}_1 := \{(-1, 0), \tilde{a}, \tilde{b}\}$ and $\tilde{t}_2 := \{(1, 0), \tilde{b}, \tilde{a}\}$, where $a = (-0.01, -0.59)$, $b = (0.01, 0.57)$

Architecture	$\tilde{c}$ and $\tilde{t}_1$	c and $\tilde{t}_2$	c and $\tilde{t}_1 \cup \tilde{t}_2$
–march=haswell	Outside	Outside	Inside
–march=ivybridge	Touches	Touches	Inside
Exact	Inside	Outside	Inside

and regardless of whether underflow occurs

$$|\mathrm{FMA}\,(a, b, c) - ab + c| \leq \varepsilon\,(|\mathrm{FMA}\,(a, b, c)| \oplus u_N)\,. \tag{5}$$

Common examples of floating-point number systems include the binary FPN with $p = 24$, $e_{\min} = -126$, $e_{\max} = 127$ called *single-precision* or *FP32* and the binary FPN with $p = 53$, $e_{\min} = -1022$, $e_{\max} = 1023$ *double-precision* or *FP64*.

Remark 2 The requirements of the previous definitions are met by IEEE 754-conforming binary floating-point number systems which include the native single- and double-precision floating-point types of the architectures x86, x86-64, current ARM, common virtual machines running WebAssembly and current CUDA processors. The machine epsilon is sometimes defined as the difference between 1.0 and the next number in F.

We call

$$\begin{aligned}\tilde{p}_{\mathrm{orientation\_2}}\,(a, b, c) := (a_x \ominus c_x) \odot \left(b_y \ominus c_y\right) \ominus \\ \left(a_y \ominus c_y\right) \odot (b_x \ominus c_x)\end{aligned} \tag{6}$$

a floating-point realisation of (1). Due to rounding errors, this realisation can produce incorrect results.

As an example, consider the points $a = (-0.01, -0.59)$, $b = (0.01, 0.57)$, $c = (0, -0.01)$. In real arithmetic, c lies on the straight line through a and b. Their closest approximations in $F_{53,-1022,1023}$ (IEEE 754-2008 binary64 [18], or FP64 for short), $\tilde{a}, \tilde{b}, \tilde{c}$, however, are only very close to collinear, which makes the case sensitive to rounding errors.

As a second example, let us evaluate the spatial relationship between the point c and the closed triangles $\tilde{t}_1 := \{(-1, 0), \tilde{a}, \tilde{b}\}$ and $\tilde{t}_2 := \{(1, 0), \tilde{b}, \tilde{a}\}$ using the winding-number algorithm [32].

Table 1 summarizes the results, all compiled with GCC 11.1 and optimization level O2. The first row is particularly noteworthy because the results are not only incorrect but also mutually contradictory. The final row can be obtained using any implementation of the orientation predicate that guarantees correct results, such as the implementation of Shewchuk [29] or CGAL's kernels epick or epeck [7].

Remark 3 The difference between the architectures is due to GCC producing an assembly code using the FMA instruction for evaluating (1). This instruction causes loss of anticommutativity for difference, i.e. $a \odot b \ominus c \odot d = -(c \odot d \ominus a \odot b)$ holds if no range errors occur, but FMA $(a, b, -c \odot d) = -$FMA $(c, d, -a \odot b)$ is not necessarily true. When inserted into the orientation predicate, this can lead to situations in which swapping two input points does not reverse the sign of the result.

Inconsistencies can occur without FMA as well. Consider $\tilde{a}, \tilde{b}, \tilde{d} :=$ (rd (0.15) , rd (8.69)) and $\tilde{e} :=$ (rd (0.07) , rd (4.05)). The floating-point realisation (6), compiled without FMA-optimizations, will determine $\tilde{a}, \tilde{b}, \tilde{e}$ and $\tilde{b}, \tilde{d}, \tilde{e}$ to be collinear but not $\tilde{a}, \tilde{b}, \tilde{d}$, which is a contradiction.

Besides rounding errors, incorrect predicate results can also be caused by overflow or underflow. It can be easily checked that, in the FP64 number system,

$$\tilde{p}_{\text{orientation\_2}}\left(\left(2^{-801}, 2^{-801}\right), \left(2^{-800}, 2^{-800}\right), \left(2^{-801}, 2^{-800}\right)\right) = 0,$$

due to underflow, and

$$\tilde{p}_{\text{orientation\_2}}\left(\left(2^{800}, 2^{800}\right), \left(2^{800}, 2^{800}\right), (0, 0)\right) = \text{NaN},$$

due to overflow.

Different approaches have been developed to obtain consistent results. We briefly discuss arbitrary precision arithmetic and floating-point filters in the following sections.

2.2 Exact arithmetic

A natural idea to solve the precision issues of floating-point arithmetic would be to perform the computations at higher precision. There are a number of arbitrary-precision libraries that implement number types with increased precision in software, such as GMP [15], the CGAL Number Types package [17] or Boost Multiprecision [23].

In combination with filters, such arbitrary-precision number types are used for exact geometric predicates in the CGAL 2D and 3D kernels, which were documented in [7]. A drawback of software-implemented number types is that basic operations can be orders of magnitude slower than hardware-implemented operations for native number types such as single- or double-precision floating-point operations on most modern processor architectures.

An approach for arbitrary-precision arithmetic that makes use of hardware acceleration is expansion arithmetic. A floating-point expansion is a tuple of multiple floating-point numbers that can represent a single number as an unevaluated sum with greater precision than a single floating-point number. Because the operations on floating-point expansions are implemented in terms of hardware-accelerated operations on the components, they can be faster than more general techniques for arbitrary precision arithmetic. The use of floating-point expansions for exact geometric predicates has been described in [30].

2.3 Floating-point filters

We call an implementation a robust floating-point predicate if it is guaranteed to produce correct results. With expansion arithmetic, we can produce a robust predicate from a floating-point realisation by replacing all rounding floating-point operators $\oplus$, $\ominus$ and $\odot$ with the respective exact algorithms on floating-point expansions. The sign of the resulting expansion is then equal to the sign of its most significant (i.e. largest non-zero) component.

The issue with this naive approach is that even simple predicates become computationally expensive. To mitigate this issue, we resort to expansion arithmetic only in the rare case that the straightforward floating point implementation is not guaranteed to produce the correct result. This decision is made by filters.

Definition 4 (Filter) For a predicate $\operatorname{sign}(p(x_1, \ldots, x_n))$ and an FPN system F, we call $f : M \subseteq F^n \to \{-1, 0, 1, \text{uncertain}\}$ a *floating-point filter*. f is called valid for p on M if for each $(x_1, \ldots, x_n) \in M$ either $f(x_1, \ldots x_n) = \operatorname{sign}(p(x_1, \ldots, x_n))$ or $f(x_1, \ldots, x_n) = \text{uncertain}$ holds. The latter case is referred to as *filter failure*.

Adopting the terminology used in [11], we call filters *dynamic* if they require the computation of an error at every step of the computation, *static* if they use a global error bound that does not depend on the inputs for each call of the predicate, and *semi-static* if their error bound has a static component and a component that depends on the input. A variation of static filters, which require a priori restrictions on the inputs to compute global error bounds, are *almost static* filters, which start with an error bound based on initial bounds on the input and update their error bound whenever the inputs exceed the previous bounds.

Example 1 (Shewchuk's Stage A orientation predicate) Consider the predicate $\operatorname{sign}(p)$ based on (1) and its floating-point realisation $\operatorname{sign}(\tilde{p})$ (6). Then,

$$f\left(a_x, \ldots, c_y\right) := \begin{cases} \operatorname{sign}(\tilde{p}), & \text{if } |\tilde{p}| \geq e\left(a_x, \ldots, c_y\right) \\ \text{uncertain}, & \text{otherwise} \end{cases}$$

with the error bound

$$e\left(a_x, \ldots, c_y\right) := \left(3\varepsilon + 16\varepsilon^2\right) \odot \left(\left|(a_x \ominus c_x) \odot \left(b_y \ominus c_y\right)\right| \oplus \right. \\ \left.\left|\left(a_y \ominus c_y\right) \odot (b_x \ominus c_x)\right|\right),$$

where $\tilde{p} := \tilde{p}\left(a_x, \ldots, c_y\right)$ and ε is the machine-epsilon of the FPN, is a valid filter for all inputs that do not cause underflow [30].

If underflow occurs, however, validity is not guaranteed. Consider the example

$$a := \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$
$$b := \begin{pmatrix} 2^{e_{\min}} \\ 0 \end{pmatrix}$$

$$c := \begin{pmatrix} 2^{e_{\min}} \\ 2^{e_{\min}} \end{pmatrix}.$$

Clearly the points are not collinear, however, $e\left(a_x, \ldots, c_y\right)$ and $\tilde{p}$ will evaluate as zero due to underflow, which shows that the filter can certify incorrect signs.

Remark 4 The term "error bound" is used in Example 1 for the quantity $e\left(a_x, \ldots, c_y\right)$ somewhat loosely. It is only proven to be larger than the absolute error of the floating-point result in cases that might produce incorrect signs (and if no underflow occurs), which is sufficient for the validity of the filter. The term "error bound" is similarly used for the bounds in Example 3 and Theorem 1.

This filter can be considered semi-static, with its static component being $3\varepsilon + 16\varepsilon^2$. The error bound is obtained mostly by applying standard forward-error analysis to the floating-point realisation. Shewchuk also described similar filters for the 2D incircle predicate, as well as the 3D orientation and incircle predicates.

Example 2 (FPG orientation filter [24]) Consider predicate (1) and its floating-point realisation (6). Let

$$\begin{aligned} m_x &:= \max\left\{|a_x - c_x|, |b_x - c_x|\right\} \\ m_y &:= \max\left\{\left|a_y - c_y\right|, \left|b_y - c_y\right|\right\}. \end{aligned}$$

If

$$\begin{aligned} &\max\left\{m_x, m_y\right\} > 2^{509}, \\ &0 \neq \min\left\{m_x, m_y\right\} \leq 2^{-485} \end{aligned}$$

or

$$|\tilde{p}| \leq 8.88720573725927 \times 10^{16} \odot m_x \odot m_y \neq 0,$$

then "uncertain" is returned, otherwise the sign of $\tilde{p}$ is returned. The filter is valid with FP64 arithmetic for all FP64 inputs. It is also semi-static with the static component of the error bound being $8.88720573725927 \times 10^{16}$ (roughly 4ε).

A static version of this filter can be obtained if global bounds for m_x and m_y are known a priori. The first two conditions are range-checks that guard against overflow and underflow. Apart from these conditions, the filter is based on an error bound similar to the previous example. The program FPG can generate such filters for arbitrary homogeneous polynomials if group annotations for the input variables are provided. In this context, group annotations are lists of grouped variables that are part of the input for FPG. The group annotations help the code generator with the choice of the scaling factors m_x and m_y. In the example above, the group annotations specified that a_x, b_x and c_x as well as a_y, b_y and c_y form a group.

Example 3 (2D orientation filter by Ozaki et al. [26]) Consider again predicate (1) and its floating-point realisation (6). Let

$$f\left(a_x, \ldots, c_y\right) := \begin{cases} \operatorname{sign}(\tilde{p}), & \text{if } |\tilde{p}| > e\left(a_x, \ldots, c_y\right) \\ \text{uncertain}, & \text{otherwise} \end{cases}$$

with the error bound

$$e\left(a_x, \ldots, c_y\right) := \theta \odot \left(\left|\left(a_x \ominus c_x\right) \odot \left(b_y \ominus c_y\right) \oplus \left(a_y \ominus c_y\right) \odot \left(b_x \ominus c_x\right)\right| + u_N\right),$$

where $\tilde{p} := \tilde{p}\left(a_x, \ldots, c_y\right)$, ε is the machine-epsilon of the FPN, u_N is the smallest, positive normalized number in the floating-point system and

$$\theta := 3\varepsilon - \left(2\left\lfloor\frac{-1+\sqrt{\varepsilon^{-1}+45}}{4}\right\rfloor - 22\right)\varepsilon^2 \in F.$$

Then f is a valid filter for all inputs.

Unlike Example 1, this filter cannot produce incorrect results for inputs that cause underflows, and unlike Example 2 which evaluates multiple inequalities at which it can branch, this filter only contains a single branch. The static constant θ, which is better than in the other two filters, has been obtained in [26] by using a model of floating-point arithmetic that bounds the relative rounding error by the unit in the first place as introduced in [28], which is smaller than the machine epsilon unless the significand of the result is exactly 1, and by considering more carefully the accumulated error of the entire predicate expression, rather than just propagating the maximum possible error in each subexpression and by considering various cases in which the sign can be guaranteed to be correct.

A disadvantage of the filter in this example is that, unlike the previous two filters, it returns “uncertain” for common, simple degeneracies like three points that have the same x-coordinate or the same y-coordinate or contain duplicate points.

The next example is not strictly an error bound filter.

Example 4 (Shewshuk’s stage B orientation predicate) Consider the predicate (1) and its floating-point realisation (6). Let $d_{ax} := a_x \ominus c_x$, $d_{bx} := b_x \ominus c_x$ and analogously for y. If the computations of these values incurred round-off errors, return uncertain. Otherwise compute $d_{ax} \cdot d_{by} - d_{ay} \cdot b_{dx}$ exactly, using expansion arithmetic, and return the sign. This filter is described as stage B in [30] and is valid for all inputs that do not produce overflow or underflow. The full version in [30] also includes an error bound with an error bound on the order of ε^2 check that allows preventing a filter failure if the no-round-off test fails.

Similar filters were presented by Shewchuk for other predicates. This filter is particularly effective for input points that are closer to each other than to (0, 0) because differences of floating-point numbers that are within half/double of each other do not incur round-off errors. In the context of Shewchuk’s multi-staged predicates, this filter also has the advantage that it can reuse computations from stage A and that its interim results can be reused for more precise stages in case of filter failure. As a final example, we present a dynamic filter.

Example 5 (Interval arithmetic filter) Consider a predicate and one of its floating-point realisations. Given the inputs, compute for each floating-point operation $\oplus$, $\ominus$, $\odot$ the lower and the upper bound of the result, including the rounding error, using interval arithmetic. If the final resulting interval contains numbers of different signs, return uncertain. Otherwise, return the shared sign of all numbers in the result interval. This approach is presented in [6].

In [11], [26] and [30] failure probabilities and performance experiments for various sequences of filters, types of inputs and algorithms are presented. We will present our own experiments in Sect. 4.2.

3 Semi-static filters

In this section, we will define a set of rules that allow us to derive error bounds for arbitrary floating-point polynomials. These error bounds will then be used to define semi-static filters. We start with establishing some properties of floating-point operations that will be used in the proof of the validity of our error bounds.

Lemma 1 *Let $a, b \in F$ be floating-point numbers.*

1. *If either a or b is in $\{-\infty, \infty, \mathrm{NaN}\}$, then*

$$a \circledcirc b \in \{-\infty, \infty, \mathrm{NaN}\}$$

and

$$|a \circledcirc b| \in \{\infty, \mathrm{NaN}\}$$

for every $\circledcirc \in \{\oplus, \ominus, \odot\}$. Consequently, the same holds for all floating-point expressions using the operators $\oplus, \ominus, \odot$ and $|\cdot|$ that contain a subexpression that evaluates to $-\infty, \infty$ or NaN.

2. *If an underflow occurs in the computation of $a \oplus b$ or $a \ominus b$, then the result is exact.*

The first statement follows directly from 3 and the second statement is given as Theorem 3.4.1 in [16].

Let $p : \mathbb{R}^m \to \mathbb{R}$ a polynomial in m variables, denoted as $p \in \mathbb{R}[x_1, \ldots, x_m]$. Let $\tilde{p} : F^m \to F$ be a floating-point realisation of p, i.e. a function on F^m involving only the floating-point operations $\oplus$, $\ominus$ and $\odot$ such that it would be equivalent to p if the floating-point operations were replaced by the corresponding exact operations. Note, that $\tilde{p}$ is not unique, e.g. $(x_1 \oplus x_2) \oplus x_3$ is different from $x_1 \oplus (x_2 \oplus x_3)$ but both are floating-point realisations of the real polynomial $x_1 + x_2 + x_3$. We denote by $F[x_1, \ldots, x_m]$ the set of floating-point realisations of polynomials in m variables. The subexpressions of $\tilde{p}$ will be denoted by $\tilde{p}_1, \ldots, \tilde{p}_k$.

We will present a recursive scheme that allows the derivation of error bound expressions for semi-static, almost static and static floating-point filters. We will assume that the final operation of $\tilde{p}$ is a sum or difference, so it holds that $\tilde{p} = \tilde{p}_1 \circledcirc \tilde{p}_2$ with $\circledcirc \in \{\oplus, \ominus\}$. If it were a multiplication, the signs of each factor could be determined independently. These filters will require only one branch like the filters in [26] and will not certify incorrect values for inputs that cause overflow and optionally for inputs that cause underflow.

3.1 Error bounds

As a reminder, semi-static error bounds are partially computed at compile-time and partially computed from the input values at runtime. The static component of our error bounds is a polynomial in the machine epsilon ε, so an element of $\mathbb{R}[\varepsilon]$, with integer coefficients. The runtime component of our semi-static error bounds is an expression in input values $x_1, \ldots, x_m$ and constants with the operators $\oplus, \ominus, \odot$ and $|\cdot|$. We will call the set of such expressions $F'[x_1, \ldots, x_m]$. We will define two *error bound maps* E and E_{UFP} for all subexpressions $\tilde{q}$ of $\tilde{p}$ of the form

$$E, E_{\text{UFP}} : F[x_1, \ldots, x_m] \to \mathbb{R}[\varepsilon] \times F'[x_1, \ldots, x_m], \quad \tilde{q} \mapsto (a, m),$$

such that the following invariants hold:

Either

$$m(x_1, \ldots, x_m) \in \{\infty, \text{NaN}\} \tag{I1}$$

or both

$$|\tilde{q}(x_1, \ldots, x_m)| \leq m(x_1, \ldots, x_m), \tag{I2.1}$$

and

$$|\tilde{q}(x_1, \ldots, x_m) - q(x_1, \ldots, x_m)| \leq a(\varepsilon) \cdot m(x_1, \ldots, x_m). \tag{I2.2}$$

E_{UFP}, where UFP signifies underflow protection, will be constructed such that these invariants hold regardless of whether underflow occurs during any of the computations. For E this will not be guaranteed. The value of the static component $a(\varepsilon)$ of an error bound is in $\mathbb{R}$ but not necessarily representable in F. In an implementation, it can be represented as a list of integer coefficients. Because the polynomial $a(\cdot)$ will only be evaluated in ε, we will omit the argument and will use the polynomial and its value in ε interchangeably. The error bound maps are defined through a list of recursive *error bound rules*,

$$R_{i(,\text{UFP})} : F[x_1, \ldots, x_m] \to \mathbb{R}[\varepsilon] \times F'[x_1, \ldots, x_m]$$

for $1 \le i \le 9$ as follows:

Definition 5 (Error Bound Rules, Error Bound Map) Let $\tilde{q} : F^m \to F$ be a subexpression of a floating-point polynomial $\tilde{p} : F^m \to F$. We define the following error bound rules:

1. For a $\tilde{q}$ of the form $\tilde{q}(x_1, \ldots, x_m) = c$ for some $c \in F$, we set
$$R_1(\tilde{q}) := (0, |c|).$$
2. For a $\tilde{q}$ of the form $\tilde{q}(x_1, \ldots, x_m) = x_i$ for some $1 \le i \le m$, we set
$$R_2(\tilde{q}) := (0, |x_i|).$$
3. For a $\tilde{q}$ of the form $\tilde{q}(x_1, \ldots, x_m) = x_i \circledcirc x_j$ for some $1 \le i, j \le m$ and $\circledcirc \in \{\oplus, \ominus\}$, we set
$$R_3(\tilde{q}) := \left(\varepsilon, \left|x_i \circledcirc x_j\right|\right).$$
4. For a $\tilde{q}$ of the form $\tilde{q}(x_1, \ldots, x_m) = x_i \odot x_j$ for some $1 \le i, j \le m$, we set
$$R_4(\tilde{q}) := \left(\varepsilon, \left|x_i \odot x_j\right|\right).$$
and
$$R_{4,\mathrm{UFP}}(\tilde{q}) := \left(\varepsilon, \left|x_i \odot x_j\right| \oplus u_N\right).$$
5. For a $\tilde{q}$ of the form $\tilde{q}(x_1, \ldots, x_m) = \left(x_i \circledcirc_1 x_j\right) \odot \left(x_h \circledcirc_2 x_g\right)$ for some $1 \le g, h, i, j \le m$ and $\circledcirc_1, \circledcirc_2 \in \{\oplus, \ominus\}$, we set
$$R_5(\tilde{q}) := \left(3\varepsilon - (\phi - 14)\varepsilon^2, \left|\left(x_i \circledcirc_1 x_j\right) \odot \left(x_h \circledcirc_2 x_g\right)\right|\right)$$
and
$$R_{5,\mathrm{UFP}}(\tilde{q}) := \left(3\varepsilon - (\phi - 14)\varepsilon^2, \left|\left(x_i \circledcirc_1 x_j\right) \odot \left(x_h \circledcirc_2 x_g\right)\right| \oplus u_N\right)$$
with
$$\phi := 2\left\lfloor \frac{-1 + \sqrt{4\varepsilon^{-1} + 45}}{4} \right\rfloor.$$
6. For a $\tilde{q}$ of the form $\tilde{q}(x_1, \ldots, x_m) = \tilde{q}_1(x_1, \ldots, x_m) \circledcirc \tilde{q}_2(x_1, \ldots, x_m)$ with $\circledcirc \in \{\oplus, \ominus\}$, we set
$$R_6(\tilde{q}) := ((1 + \varepsilon)\max(a_1, a_2) + \varepsilon, m_1 \oplus m_2)$$

and

$$R_{6,\mathrm{UFP}}(\tilde{q}) := ((1+\varepsilon)\max(a_1, a_2) + \varepsilon, m_1 \oplus m_2)$$

with $(a_i, m_i) := E(\tilde{q}_i)$ and $(a_i, m_i) := E_{\mathrm{UFP}}(\tilde{q}_i)$ respectively for $i = 1, 2$.

7. For a $\tilde{q}$ of the form $\tilde{q}(x_1, \ldots, x_m) = \tilde{q}_1(x_1, \ldots, x_m) \odot \tilde{q}_2(x_1, \ldots, x_m)$, we set

$$R_7(\tilde{q}) := ((1+\varepsilon)(a_1 + a_2 + a_1 a_2) + \varepsilon, m_1 \odot m_2)$$

and

$$R_{7,\mathrm{UFP}}(\tilde{q}) := ((1+\varepsilon)(a_1 + a_2 + a_1 a_2) + \varepsilon, m_1 \odot m_2 \oplus u_N)$$

with $(a_i, m_i) := E(\tilde{q}_i)$ and $(a_i, m_i) := E_{\mathrm{UFP}}(\tilde{q}_i)$ respectively for $i = 1, 2$.

8. For a $\tilde{q}$ of the form $\tilde{q}(x_1, \ldots, x_m) = \mathrm{FMA}\left(x_h, x_i, x_j\right)$ for some $1 \leq h, i, j \leq m$ we set

$$R_8(\tilde{q}) := \left(\varepsilon, \left|\mathrm{FMA}\left(x_h, x_i, x_j\right)\right|\right)$$

and

$$R_{8,\mathrm{UFP}}(\tilde{q}) := \left(\varepsilon, \left|\mathrm{FMA}\left(x_h, x_i, x_j\right)\right| \oplus u_N\right)$$

9. For a $\tilde{q}$ of the form $\tilde{q}(x_1, \ldots, x_m) =$FMA $(\tilde{q}_1(x_1, \ldots, x_m), \ldots, \tilde{q}_3(x_1, \ldots, x_m))$ we set

$$R_9(\tilde{q}) := (a, |\mathrm{FMA}(m_1, m_2, m_3)|)$$

and

$$R_{9,\mathrm{UFP}}(\tilde{q}) := (a, |\mathrm{FMA}(m_1, m_2, m_3)| \oplus u_N)$$

with

$$a := \max((a_1 + a_2 + a_1 a_2)(1+\varepsilon), a_3)(1+\varepsilon) + \varepsilon$$

We define $E(\tilde{q})$ to be the first applicable map out of $R_1, \ldots, R_9$ and analogously $E_{\mathrm{UFP}}(\tilde{q})$ with the respective UFP-variations of the rules.

It is straightforward to see that E and E_{UFP} are well-defined because the rules are exhaustive in the sense that there is no subexpression in a floating-point polynomial for which no rule is applicable and any recursion through R_6 and R_7 or their UFP-variations terminates at the level of individual variables.

Lemma 2 *Let $\tilde{p}$ be an arbitrary floating-point polynomial then the invariants for $E(\tilde{q})$ hold for every subexpression $\tilde{q}$ of $\tilde{p}$ for every choice of floating-point inputs $x_1, \ldots, x_m \in F$ such that no underflow occurs in the evaluation of any subexpression of $\tilde{q}$.*

Following [30], we introduce the following convenient notation that will be used in the proof. We extend the arithmetic operations $\circ$ to sets $A, B \subset \mathbb{R}$ by $A \circ B := \{a \circ b \mid a \in A, b \in B\}$, identify $a \in \mathbb{R}$ with $\{a\}$ for $\circ \in \{+, -, \cdot\}$, and set $A \pm a := A + [-a, a]$.

Proof For any subexpression $\tilde{q}$ to which R_1 or R_2 applies, the statement is obvious. For subexpressions for which R_3, R_4 or R_8 are the first applicable rules and no overflow occurs, (I2.1) holds by the definition of m and (I2.2) follows from the error bounds 2–4 respectively. If overflow occurs, m is infinity and (I1) holds. For subexpressions, to which R_5 applies, either (I1) holds if overflow occurs or, if no overflow occurs, (I2.1) holds by definition and (I2.2) is proven in [26] in Lemma 3.1.

If R_6 is the first applicable rule, we assume that the invariant (I1) or the invariants (I2.1) and (I2.2) hold for $(a_1, m_1) := E(\tilde{q}_1)$ and $(a_2, m_2) := E(\tilde{q}_2)$ and we consider the case that $\tilde{q} = \tilde{q}_1 \oplus \tilde{q}_2$. If $\tilde{q}_1$ or $\tilde{q}_2$ is either ∞ or NaN, by the assumption so are m_1 or m_2 and consequently $m_1 \oplus m_2$ and (I1) holds. If no overflow occurs, we see that

$$\begin{aligned} |\tilde{q}| &= |\tilde{q}_1 \oplus \tilde{q}_2| \\ &\leq |\tilde{q}_1| \oplus |\tilde{q}_2| \\ &\leq m_1 \oplus m_2 \end{aligned}$$

and

$$\begin{aligned} \tilde{q} &= \tilde{q}_1 \oplus \tilde{q}_2 \\ &\in \tilde{q}_1 + \tilde{q}_2 \pm \varepsilon\, |\tilde{q}_1 \oplus \tilde{q}_2| \\ &\subseteq \tilde{q}_1 + \tilde{q}_2 \pm \varepsilon\, (m_1 \oplus m_2) \\ &\subseteq q_1 \pm a_1(\varepsilon)\, m_1 + q_2 \pm a_2(\varepsilon)\, m_2 \pm \varepsilon\, (m_1 \oplus m_2) \\ &\subseteq q \pm \max(a_1(\varepsilon), a_2(\varepsilon))\, (m_1 + m_2) \pm \varepsilon\, (m_1 \oplus m_2) \\ &\subseteq q \pm (\max(a_1(\varepsilon), a_2(\varepsilon))\,(1 + \varepsilon) + \varepsilon)\, (m_1 \oplus m_2), \end{aligned}$$

where we used the assumption that the invariant holds for the two subexpressions and standard floating-point rounding error estimates. The proof for $\tilde{q} = \tilde{q}_1 \ominus \tilde{q}_2$ is analogous.

If R_7 is the first applicable rule, we assume that the invariant holds for $(a_1, m_1) := E(\tilde{q}_1)$ and $(a_2, m_2) := E(\tilde{q}_2)$. Analogous to above, the case of overflow is trivial, so we consider the case that no overflow occurs. Again it holds that

$$\begin{aligned} |\tilde{q}| &= |\tilde{q}_1 \odot \tilde{q}_2| \\ &\leq m_1 \odot m_2. \end{aligned}$$

and

$$\begin{aligned} \tilde{q} &= \tilde{q}_1 \odot \tilde{q}_2 \\ &\in \tilde{q}_1 \cdot \tilde{q}_2 \pm \varepsilon\, |\tilde{q}_1 \odot \tilde{q}_2| \\ &\subseteq \tilde{q}_1 \cdot \tilde{q}_2 \pm \varepsilon\, (m_1 \odot m_2) \end{aligned}$$

$$\begin{aligned}
&\subseteq (q_1 \pm a_1 m_1) \cdot (q_2 \pm a_2 m_2) \pm \varepsilon\,(m_1 \odot m_2)\\
&\subseteq q_1 q_2 \pm a_2 m_2 q_1 \pm a_1 m_1 q_2 \pm a_1 a_2 m_1 m_2 \pm \varepsilon\,(m_1 \odot m_2)\\
&\subseteq q_1 q_2 \pm a_2\,(1+\varepsilon)\,(m_1 \odot m_2) \pm a_1\,(1+\varepsilon)\,(m_1 \odot m_2)\\
&\quad \pm a_1 a_2\,(1+\varepsilon)\,(m_1 \odot m_2) \pm \varepsilon\,(m_1 \odot m_2)\\
&\subseteq q_1 q_2 \pm ((1+\varepsilon)\,(a_1 + a_2 + a_1 a_2) + \varepsilon)\,(m_1 \odot m_2)\,.
\end{aligned}$$

The proof for R_9 combines the steps of the proofs for R_6 and R_7 and uses that the multiplication in the FMA can be assumed to be error-free Because the recursion eventually terminates at a non-recursion case (rules 1–5), the claims for $E\,(\tilde{q}_1)$ and $E\,(\tilde{q}_2)$ hold. □

Lemma 3 *Let $\tilde{p}$ be an arbitrary floating-point polynomial and then the invariants for $E_{\mathrm{UFP}}\,(\tilde{q})$ hold for every subexpression $\tilde{q}$ of $\tilde{p}$ for every choice of inputs $x_1, \ldots, x_m \in F$.*

Proof Because it is useful for the parts of the proof that apply to the recursive rules R_6 and R_7, we will also prove for each rule applied to a subexpression $\tilde{q}$ that

$$q = \tilde{q} \quad \vee \quad m \geq u_N \tag{I3}$$

holds.

For R_1 and R_2 there is nothing to prove.

For R_3 the reasoning given in the proof for Lemma 2 still applies because the assumption of no underflow occurring was not used. If underflow occurs then $\tilde{q}$ is evaluated exactly, i.e. $\tilde{q} = q$ and if no underflow occurs then $\tilde{q}$ is not subnormal and hence it holds that $m \geq u_N$.

For $R_{4,\mathrm{UFP}}$ and $R_{8,\mathrm{UFP}}$, we first note that m is always non-zero and not smaller than either $\tilde{q}$ or u_N so invariants (I2.1) and (I3) hold. Invariant (I2.2) then follows directly from $2\varepsilon u_N = u_S$ and 3 and 5 respectively.

(I2.2) for $R_{5,\mathrm{UFP}}$ was proven as Lemma 3.1 in [26]. For (I2.1) and (I3) the same reasoning as for $R_{r,\mathrm{UFP}}$ applies.

For the recursive rules $R_{6,\mathrm{UFP}}$ and $R_{7,\mathrm{UFP}}$, we assume at all invariants hold for the respective subexpressions $\tilde{q}_1$ and $\tilde{q}_2$, this is again justified because all recursions will be cases of rules R_1 to $R_{5,\mathrm{UFP}}$ for which the invariants were already proven to hold or to other cases for rules $R_{6,\mathrm{UFP}}$ and $R_{7,\mathrm{UFP}}$.

For $R_{6,\mathrm{UFP}}$, as in Lemma 2, it is obvious that invariant (I2.1) holds. If either m_1 or m_2 is equal to or greater than u_N, then no underflow can occur in the evaluation of m and the invariant holds as proven in Lemma 2 and m is greater than or equal to u_N. If both m_1 and m_2 are smaller than u_N then $\tilde{q}_1$ and $\tilde{q}_2$ are evaluated error-free and $\tilde{q}$ is error-free if underflow occurs. If no underflow occurs in the evaluation of $\tilde{q}$, the invariant also holds as proven in Lemma 2 and in this case m is equal to or greater than u_N.

For $R_{7,\mathrm{UFP}}$, we first note that, as in $R_{4,\mathrm{UFP}}$, m is greater than or equal to both $\tilde{q}$ and u_N, so invariants I2.1 and (I3) hold. To show that (I2.2) holds, we use 3 to obtain

$$\tilde{q} = \tilde{q}_1 \odot \tilde{q}_2$$

$$\begin{aligned}
&\in \tilde{q}_1 \cdot \tilde{q}_2 \pm \varepsilon \left(|\tilde{q}_1 \odot \tilde{q}_2| \oplus u_N\right) \\
&\subseteq (q_1 \pm a_1 m_1) \cdot (q_2 \pm a_2 m_2) \pm \varepsilon \left(m_1 \odot m_2 \oplus u_N\right) \\
&\subseteq q_1 q_2 \pm a_2 m_2 q_1 \pm a_1 m_1 q_2 \pm a_1 a_2 m_1 m_2 \pm \varepsilon \left(m_1 \odot m_2 \oplus u_N\right) \\
&\subseteq q_1 q_2 \pm a_2 (1+\varepsilon)(m_1 \odot m_2) \pm a_1 (1+\varepsilon)(m_1 \odot m_2) \\
&\pm a_1 a_2 (1+\varepsilon)(m_1 \odot m_2) \pm \varepsilon \left(m_1 \odot m_2 \oplus u_N\right) \\
&\subseteq q_1 q_2 \pm \left((1+\varepsilon)(a_1 + a_2 + a_1 a_2) + \varepsilon\right)\left(m_1 \odot m_2 \oplus u_N\right).
\end{aligned}$$

The proof for $R_{9,\mathrm{UFP}}$ combines the steps of the proofs for R_6 and $R_{7,\mathrm{UFP}}$ and uses that the multiplication in the FMA can be assumed to be error-free. □

3.2 Floating-point filters

The following result provides two semi-static filters for floating-point predicates that evaluate the sign of a polynomial. It is only stated for floating-point realisations of polynomials that are sums or differences. For products, the signs of each factor could be obtained individually and then multiplied.

Theorem 1 *Let $p \in \mathbb{R}[x_1, \ldots, x_m]$ be a polynomial and $\tilde{p} \in F[x_1, \ldots, x_m]$ be some floating-point realisation of p.*

1. *Let $\tilde{p}$ be of the form $\tilde{p} = \tilde{p}_1 \oplus \tilde{p}_2$ or $\tilde{p} = \tilde{p}_1 \ominus \tilde{p}_2$, $(a_1, m_1) := E(\tilde{p}_1)$ and $(a_2, m_2) := E(\tilde{p}_2)$. Moreover, let constants $a_3, a_4 \in F$ satisfy*

$$a_3 > \frac{\max(a_1, a_2)}{1-\varepsilon}, \qquad a_4 \geq a_3 (1+\varepsilon)^2,$$

and define

$$e(x_1, \ldots, x_m) := a_4 \odot \left(m_1(x_1, \ldots, x_m) \oplus m_2(x_1, \ldots, x_m)\right).$$

Then, for every choice of $x_1, \ldots, x_m \in F \setminus \{\mathrm{NaN}, \infty, -\infty\}$ such that no underflow occurs in the evaluation of $\tilde{p}$ or e,

$$f(x_1, \ldots, x_m) := \begin{cases} \mathrm{sign}(\tilde{p}(x_1, \ldots, x_m)) & |\tilde{p}| > e \vee e = 0 \\ \text{uncertain} & \textit{otherwise} \end{cases}$$

is a valid filter.

2. *Let $\tilde{p}$ be of the form $\tilde{p} = \tilde{p}_1 \oplus \tilde{p}_2$ or $\tilde{p} = \tilde{p}_1 \ominus \tilde{p}_2$, $(a_1, m_1) := E_{\mathrm{UFP}}(\tilde{p}_1)$ and $(a_2, m_2) := E_{\mathrm{UFP}}(\tilde{p}_2)$. We set a_3 and a_4 as in 1. and*

$$e(x_1, \ldots, x_m) := a_4 \odot \left(m_1(x_1, \ldots, x_m) \oplus m_2(x_1, \ldots, x_m)\right) \oplus u_S.$$

Then for every choice of $x_1, \ldots, x_m \in F \setminus \{\mathrm{NaN}, \infty, -\infty\}$,

$$f(x_1, \ldots, x_m) := \begin{cases} \mathrm{sign}(\tilde{p}(x_1, \ldots, x_m)) & |\tilde{p}| > e \\ \text{uncertain} & \textit{otherwise} \end{cases}$$

is a valid filter.

3. *Let $\tilde{p}$ be of the form $\tilde{p} = \mathrm{FMA}\,(\tilde{p}_1, \tilde{p}_2, \tilde{p}_3)$, $(a_1, m_1) := E\,(\tilde{p}_1)$, $(a_2, m_2) := E\,(\tilde{p}_2)$ and $(a_3, m_3) := E\,(\tilde{p}_3)$. Moreover, let constants $a_4, a_5 \in F$ satisfy*

$$a_4 > \frac{\max\,(a_1 + a_2 + a_1 a_2, a_3)}{1 - \varepsilon}, \qquad a_5 \geq a_4\,(1+\varepsilon)^2\,,$$

and define

$$e\,(x_1, \ldots, x_m) := a_5 \odot \mathrm{FMA}\,(m_1\,(x_1, \ldots, x_m)\,, \ldots, m_3\,(x_1, \ldots, x_m))\,.$$

Then, for every choice of $x_1, \ldots, x_m \in F\setminus\{\mathrm{NaN}, \infty, -\infty\}$ such that no underflow occurs in the evaluation of $\tilde{p}$ or e, f as defined in 1. is a valid filter.

4. *Let $\tilde{p}$ be of the form $\tilde{p} = \mathrm{FMA}\,(\tilde{p}_1, \tilde{p}_2, \tilde{p}_3)$, $(a_1, m_1) := E\,(\tilde{p}_1)$, $(a_2, m_2) := E\,(\tilde{p}_2)$ and $(a_3, m_3) := E\,(\tilde{p}_3)$. We set a_4 and a_5 as in 3. and*

$$e\,(x_1, \ldots, x_m) := a_5 \odot \mathrm{FMA}\,(m_1\,(x_1, \ldots, x_m)\,, \ldots, m_3\,(x_1, \ldots, x_m)) \oplus u_{\mathrm{S}}.$$

Then for every choice of $x_1, \ldots, x_m \in F\setminus\{\mathrm{NaN}, \infty, -\infty\}$, f as defined in 2. is a valid filter.

Note that $|\tilde{p}| > e$ always evaluates as false if e is ∞ or NaN.

Proof We first prove 1. and we assume without loss of generality that $\tilde{p} = \tilde{p}_1 \oplus \tilde{p}_2$. Using Lemma 1 and Lemma 2, it holds that

$$\begin{aligned}\tilde{p} &= \tilde{p}_1 \oplus \tilde{p}_2 \\ &\subseteq (\tilde{p}_1 + \tilde{p}_2) \pm \varepsilon\tilde{p} \\ &\subseteq p \pm \varepsilon\tilde{p} \pm a_1 m_1 \pm a_2 m_2\end{aligned}$$

and equivalently

$$\begin{aligned}p &\in \tilde{p} \pm \varepsilon\tilde{p} \pm a_1 m_1 \pm a_2 m_2 \\ &\subseteq \tilde{p} \pm \varepsilon\tilde{p} \pm \max\,(a_1, a_2)\,(m_1 + m_2)\,.\end{aligned}$$

From this it follows that the signs of p and $\tilde{p}$ are equal if

$$(1-\varepsilon)\,|\tilde{p}| > \max\,(a_1, a_2)\,(m_1 + m_2)\,. \tag{7}$$

The inequality

$$|\tilde{p}| > a_3\,(m_1 + m_2) \tag{8}$$

is equivalent to

$$(1-\varepsilon)\,|\tilde{p}| > a_3\,(1-\varepsilon)\,(m_1 + m_2)\,,$$

which implies (7), so (8) is also a sufficient condition. Lastly, we see that

$$\begin{aligned} a_3 \left(m_1 + m_2\right) \leq & a_3 \left(1+\varepsilon\right) \left(m_1 \oplus m_2\right) \\ \leq & a_4 \odot \left(m_1 \oplus m_2\right), \end{aligned}$$

where the second step uses the no-underflow assumption. Hence,

$$|\tilde{p}| > a_4 \odot \left(m_1 \oplus m_2\right) =: e$$

is a sufficient condition for the signs of p and $\tilde{p}$ being equal. It remains to consider the case $e = 0$. If a_1 and a_2 are both zero, then $\tilde{p}$ is a simple expression and its sign is trivially correct, which makes f always valid. If either a_1 or a_2 is non-zero, then a_4 is easily seen to be non-zero too and e can only be zero if both m_1 and m_2 are zero, since we assumed that no underflow occurs. If $m_1 = m_2 = 0$, then by Lemma 2, $\tilde{p}_1 = \tilde{p}_2 = p_1 = p_2 = 0$, and $\tilde{p} = p = 0$.

The proof for 2. is analogous except for the constant u_S being added to e in place of assuming no underflow occurring, the omittance of the case $e = 0$, which can not occur in 2 and the usage of Lemma 3 in place of Lemma 2.

The proofs for 3. and 4. are analogous to the proofs for 1. and 2. □

Remark 5 Note that the constants a_3 and a_4 do not depend on the input but only on the expression of $\tilde{p}$ so in practice they can be computed at compile-time in floating point or exact arithmetic.

A detailed example for the construction of a filter based on this approach can be found in a Jupyter notebook using the Cling-Kernel [33] in [4].

This filter differs from the semi-static filters (called stage A) in [30], which do not guarantee valid results in cases of underflow. It also differs from the semi-static filters generated by FPG [24] because only a single condition is evaluated, rather than three conditions, which means that most predicate calls for non-degenerate inputs can be decided on a code path with a single, well-predictable branch.

With these two properties, having only a single branch on the filter success code path and validity for inputs that can cause underflow, this procedure to construct semi-static filters can be seen as a generalisation of the semi-static 2D orientation filter presented in [26]. For the 2D orientation predicate, in particular, our approach produces a more pessimistic error bound than [26], which could be considered as the price to pay for using a more general method.

The semi-static error bound e can be turned into a static error bound by evaluating $m_1 \oplus m_2$ not in specific input values $x_1, \ldots, x_m$ but in bounds on these values, $\left[\underline{x}_1, \overline{x}_1\right], \ldots, \left[\underline{x}_m, \overline{x}_m\right]$ using interval arithmetic, or by obtaining its maximum over some more general domain in F^m. This yields a static or almost static filter.

3.3 Zero-filter

With underflow protection, the right-hand side of our semi-static filter condition will never be zero, hence the filter will always fail, as in returning “uncertain”, if the true sign of p is 0. For inputs in F that approximate a uniform distribution on an interval in $\mathbb{R}$, $p = 0$ is extremely unlikely, but in some practical input data, it might be more common.

Example 6 Consider the 2D orientation predicate with the floating-point realisation

$$\tilde{p} = (a_x \ominus c_x) \odot \left(b_y \ominus c_y\right) \ominus \left(a_y \ominus c_y\right) \odot (b_x \ominus c_x)\,.$$

It is easy to check that if either point a or b coincides with point c or if all points share the same x or y coordinate and no overflow occurs, then $\tilde{p}$ evaluates to zero. Such cases can be common degeneracies in real-world data.

It can also be checked that the error bound e from our semi-static filter without underflow-protection would be zero in either of these cases, so such degeneracies can be decided quickly by the filter. The error bound of the UFP-variation of our filter, though, would not zero because non-zero terms would be introduced in the error bounds of the multiplications and in the definition of e itself.

In this case, a simple filter that can certify common cases for inputs that produce zeroes, can be useful. Such a filter can be produced using the following rules.

Definition 6 (Zero-Filter) Let $\tilde{p}$ be a floating-point realisation of a polynomial and let $x_1, \ldots, x_m \in F$ a given set of input values. We define the following rules.

1. For a subexpression $\tilde{q}$ of the form $\tilde{q} = c$ for some constant $c \in F$ or input value x_i for $i \in \{1, \ldots, m\}$, we define

$$Z_1\left(\tilde{q}; x_1, \ldots, x_m\right) = \begin{cases} \text{true}, & c = 0 \\ \text{false}, & \text{otherwise.} \end{cases}$$

2. For a subexpression $\tilde{q}$ of the form $\tilde{q} = x_i \circledcirc x_j$ for $1 \le i, j \le m$ and $\circledcirc \in \{\oplus, \ominus\}$, we define

$$Z_2\left(\tilde{q}; x_1, \ldots, x_m\right) = \begin{cases} \text{true}, & \tilde{q} = 0 \\ \text{false}, & \text{otherwise.} \end{cases}$$

3. For a subexpression $\tilde{q}$ of the form $\tilde{q} = \tilde{q}_1 \circledcirc \tilde{q}$ for $1 \le i, j \le m$ and $\circledcirc \in \{\oplus, \ominus\}$, we define

$$Z_3\left(\tilde{q}; x_1, \ldots, x_m\right) = Z\left(\tilde{q}_1; x_1, \ldots, x_m\right) \wedge Z\left(\tilde{q}_2; x_1, \ldots, x_m\right).$$

4. For a subexpression $\tilde{q}$ of the form $\tilde{q} = \tilde{q}_1 \odot \tilde{q}_2$, we define

$$Z_4\left(\tilde{q}; x_1, \ldots, x_m\right) = Z\left(\tilde{q}_1; x_1, \ldots, x_m\right) \vee Z\left(\tilde{q}_2; x_1, \ldots, x_m\right).$$

We define $Z\left(\tilde{q}; x_1, \ldots, x_m\right)$ to be the result of the first applicable rule out of Z_1, Z_2, Z_3, and Z_4.

The zero-filter returns the sign 0 if $Z(\tilde{p}; x_1, \ldots, x_m)$ is true and "uncertain" otherwise. It is easy to verify that this filter is valid for all inputs regardless of range issues such as overflow or underflow.

Example 7 Consider again the 2D orientation predicate as in Example 6. Applying Definition 6 to $\tilde{p}$, we first apply Z_3 to $\tilde{p}$, which gives us the condition

$$Z\left((a_x \ominus c_x) \odot \left(b_y \ominus c_y\right)\right) \wedge Z\left(\left(a_y \ominus c_y\right) \odot (b_x \ominus c_x)\right)$$

For each subexpression, we can apply Z_4 to obtain

$$Z(a_x \ominus c_x) \vee Z\left(b_y \ominus c_y\right) \wedge Z\left(a_y \ominus c_y\right) \vee Z(b_x \ominus c_x)$$

and finally, applying Z_2 for each difference

$$(a_x \ominus c_x = 0) \vee \left(b_y \ominus c_y = 0\right) \wedge \left(a_y \ominus c_y = 0\right) \vee (b_x \ominus c_x = 0),$$

which is a sufficient condition of the sign of p being 0 for the given inputs that can be used as a second filter stage after a filter with underflow-protection that is not able to decide simple degenerate cases, like all points sharing the same x- or y-coordinate.

4 Numerical results

The exact predicates derived in the previous section are designed to be fast, applicable to general polynomial expressions, and simple to use. These goals must be reflected in their implementation, which is briefly covered before benchmark results are presented.

4.1 C++ implementation

Our implementation of exact predicates is based on C++ template and constexpr metaprogramming, making use of the Boost.Mp11 library [12]. The main design goal is the avoidance of runtime overhead like the one that was seen in the C++-wrapper implementation in [8] because geometric predicates can be found on performance-critical code paths in geometric algorithms and can make up a large proportion of overall runtime as the benchmarks in Sect. 4.2 show. Further design goals include flexibility and extensibility with regard to the choice and order of filters as well as expressivity and simplicity in the definition of predicate expressions.

Exact predicates are implemented as variadic class templates for staged predicates that hold a tuple of zero or more stages implementing filters or the exact arithmetic evaluation. If all filters are semi-static, the instantiated class is stateless and can be constructed without arguments and with no runtime cost. The static parts of semi-static error bounds are computed at compile-time from the predicate expression and static type information for the calculation type. If almost static or static filters are included, input bounds need to be provided at construction for the computation of error bounds. For almost static filters, an update member function is provided to update error bounds. The exact predicate is called through a variadic function that takes a variable but compile-time static number of inputs in the calculation type and returns an integer out of $-1, 0$ and 1, that represents the result sign.

The individual stages are expected to follow the same basic interface. Each stage provides at least a member function that is called with input values and returns an integer that represents either the result sign or a constant that indicates uncertainty. For stages that require the computation of runtime constants, e.g. static and almost static filters, constructor and update members need to be implemented as well. Otherwise, the stages are default constructed at no runtime cost. This general interface allows users of the library to extend exact predicates with custom filters beyond those provided by our implementation to better suit their algorithms and data sets, such as the filter shown in Example 8.

```
using ssf = semi_static_filter</* ... */>;
using es = predicate_approximation </* ... */, CGAL::Gmpzf>;
// This stage is exact because it uses an exact number type.

staged_predicate<ssf, es> pred;
// default constructed and stateless 2-stage predicate

int sign = pred.apply(ax, ay, bx, ...);
// exact value of the predicate p(a,b,..)
```

Example 8 The 2D incircle predicate on four 2D points $p_1, \ldots, p_4$ decides whether p_4 lies inside, on or outside of the oriented circle passing through p_1, p_2 and p_3, assuming the points do not lie on a line. A pattern of degenerate inputs are four points that form a rectangle. For this input, p_4 clearly lies on the circle (indicated by a sign of 0) but a forward error bound filter could classify the case as undecidable and forward it to a computationally expensive exact stage. The following listing illustrates a custom filter that conforms to the previously described interface and could be used with our implementation of staged predicates.

```
struct incircle_rect_filter
{
  // stateless, no constructor or update method required
  template <typename CalculationType>
  static inline int apply(CalculationType ax, /* ... */)
  {
    if( (ax == bx && by == cy && cx == dx && dy == ay) ||
        /* ... */ )
      return 0;
    else
      return sign_uncertain;
  }
};
```

At the core of the implementation is the compile-time processing of polynomial expressions for the derivation of error bound expressions. Arithmetic expressions are represented in the C++ type system using expression templates, a technique described in [34]. The most basic expressions in our implementation are types representing the leaves of expression trees. Those leaves are either compile-time constants (indexed with zero) or input values (indexed with a positive number). More complex expressions can be built from these placeholders using the elementary operators +, − and *.

Forward error bound expression types are deduced at compile-time based on a list of rule class templates. The interface of each rule class template requires a constexpr function that expects an expression template and returns a bool indicating whether the rule is applicable to the expression, and a class template for the error bound based on the rule. Error bounds are implemented in the form of constexpr integer arrays that represent the coefficients of the polynomial in ε and a magnitude expression template. The rules can be extended through custom rules that conform to the interface.

```
constexpr auto orient2d =
  (_1 - _5) * (_4 - _6) - (_3 - _5) * (_2 - _6);
// expression template representing the 2D orientation
// predicate expression where _1, _2, _3, ... are
// placeholders for ax, ay, bx, ...

using ssf = semi_static_filter<
  orient2d,
  forward_error_bound_expression<
    orient2d,
    double,
    /* ... rules ... */>
  >;
// a shorter alias for this construct is provided
```

Example 9 Consider a 2D orientation problem for points whose coordinates are not binary floating-point numbers, e.g. because the input is given in a decimal or rational format. The rules given in Definition 5 are not designed for this problem but with a custom error bound rule, our implementation can be extended to generate a filter for inputs that are rounded to the nearest floating-point number. Such a filter could be used before going into a more computationally expensive stage operating on decimal or rational numbers.

```
struct rounded_input
{
  template <typename Expression, /* ... */>
  static constexpr bool applicable()
  {
    if constexpr (Expression::is_leaf)
      return Expression::argn > 0;
    else
      return false;
  }

  template <typename Expression, /* ... */>
  struct error_bound
  {
    using magnitude = abs<Expression>;
    static constexpr std::array<long, 3> a
      {1, 0, 0};
    // the entries represent coefficients
    // of the polynomial in eps
  };
};
```

The listing illustrates a custom rule. It is only applicable for expressions that are input values, i.e. expressions of the form $\tilde{q}\,(x_1, \ldots, x_n) = x_i$. In the context of our implementation these expressions are leaves of the expression tree with a positive index, and the error bound is $R\,(\tilde{q}) = (\varepsilon, |x_i|)$. Using a rule set consisting of this custom rule, $R_{6,0}$ and $R_{7,0}$ on the 2D orientation predicate, yields the semi-static error bound

$$\left(5\varepsilon \oplus 32\varepsilon^2\right)\left(\left(|a_x| \oplus |c_x|\right) \odot \left(\left|b_y\right| \oplus \left|c_y\right|\right) \oplus \left(\left|a_y\right| \oplus \left|c_y\right|\right) \odot \left(|b_x| \oplus |c_x|\right)\right).$$

Besides forward error bound based filters discussed in this paper, our implementation also contains templates for filters and exact stages based on the same principles as the stages B and D in [30].

4.2 Benchmarks

To test the performance of our approach and implementation, we measured timings for a number of benchmarks that are provided by the CGAL library. The design of the 2D and 3D geometry kernels concept in CGAL as documented in [7] provide a simple way to test our predicates in CGAL algorithms by deriving from the Simple_cartesian<double> kernel and overriding all predicate objects that may suffer from rounding errors with predicates generated from our implementation.

The performance with the resulting custom kernel is then compared to the performance of CGAL's Exact_predicates_inexact_constructions_kernel, which follows a similar paradigm of filtered, exact predicates.

All benchmarks were run on a GNU/Linux workstation with a Intel Core i7-6700HQ CPU using the performance scaling governor, no optional mitigations against CPU vulnerabilities such as Spectre or Meltdown, and disabled turbo for consistency. All code was compiled with GCC 11.1 and the flags "-O3 -march=native". The installed versions of relevant libraries were CGAL 5.4, GMP 6.2.1, MPFR 4.1.0, and Boost 1.79. The code for all benchmarks with instructions on how to replicate them can be found in [4].

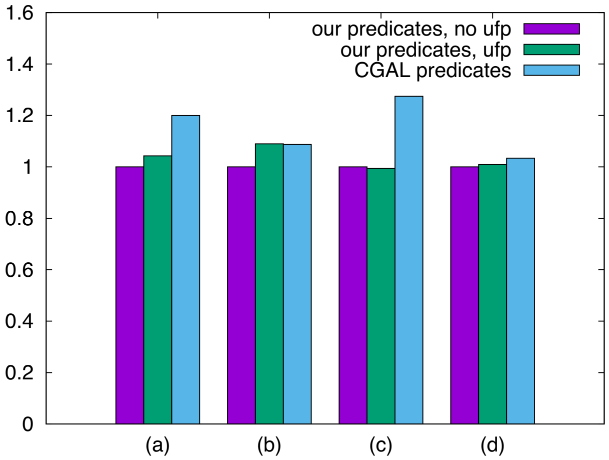


Fig. 1 This chart shows the relative runtime for the construction of a Delaunay triangulation of 1,000,000 points with coordinates sampled from **a** a continuous uniform distribution and **b** an equidistant grid, as well as for CGAL mesh **c** polygon processing and **d** refinement benchmarks. For each benchmark, our filters with and without underflow protection are compared to the predicates implemented in CGAL

4.2.1 2D delaunay triangulation

The 2D Delaunay Triangulation algorithm provided by the CGAL library makes use of the 2D orientation and incircle predicates, which compute the sign of the following expressions:

$$p_{\text{orientation\_2}} = \begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix}$$

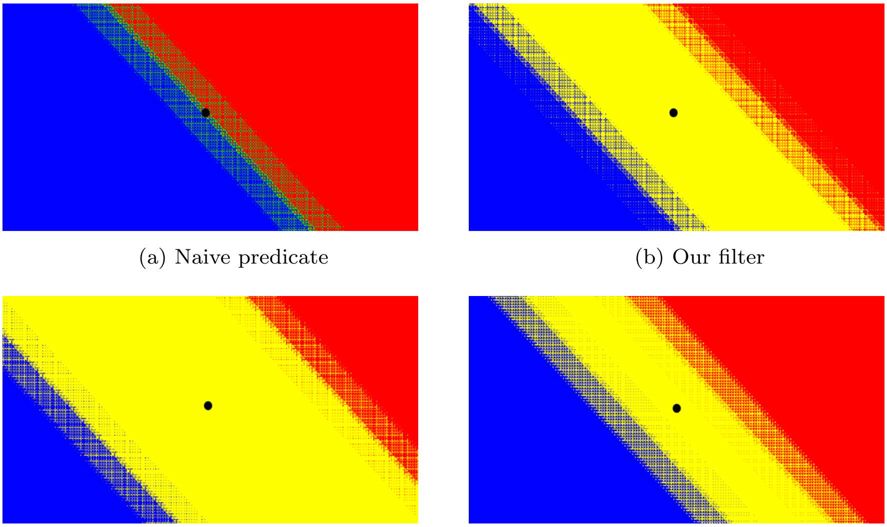
$$p_{\text{incircle\_2}} = \begin{vmatrix} a_x - d_x & a_y - d_y & \left(a_y - d_y\right)^2 + \left(a_y - d_y\right)^2 \\ b_x - d_x & b_y - d_y & \left(b_y - d_y\right)^2 + \left(b_y - d_y\right)^2 \\ c_x - d_x & c_y - d_y & \left(c_y - d_y\right)^2 + \left(c_y - d_y\right)^2 \end{vmatrix}.$$

2D Delaunay Triangulations were computed for two data sets of randomly generated points. The coordinates were sampled either from a continuous uniform distribution (using CGAL's Random_points_in_square_2 generator) or from an equidistant grid (using CGAL's points_on_square_grid_2 generator) and shuffled with 1,000,000 points in each data set. For the continuous distribution, we found a 4.2% performance penalty for the use of underflow guards, which can be explained by the slightly more expensive error expressions. With or without underflow guards, our implementation performed faster than the CGAL predicates, see (a) in Fig. 1. This is expected because all calls can be decided on a code path with a single, well-predictable branch.

For the points sampled from the equidistant grid, the triangulations were, in general, much slower, which is also expected because the input is designed to be degenerate and trigger edge cases. Our predicates with underflow protection and the predicates

Table 2 Number of filter failures for the 2D orientation and 2D incircle predicate with various semi-static filters when constructing the Delaunay triangulation of 1,000,000 points sampled from an equidistant grid

Filter failures in the first stage	No UFP	UFP	CGAL	Total calls
2D orientation	49,375	624,400	49,641	4,121,216
2D incircle	1,112,461	1,490,010	1,641,255	8,455,667



(a) Naive predicate

(b) Our filter

(c) FPG filter

(d) Interval filter

Fig. 2 This figure shows the result of calls to the non-robust 2D orientation predicate and to three 2D orientation filters respectively for the points (20.1, 20.1), (18.9, 18.9) and a small neighbourhood of the point (3.5, 3.5), such that neighbouring pixels represent points with neighbouring floating-point coordinates. The point (3.5, 3.5) is marked with a black circle. The dimensions of the neighbourhood are roughly 6×10^{-13} in width and 3×10^{-13} in height. The colours represent left side (red), collinear (green), right side (blue) and uncertain (yellow). The pattern of green points in (**a**) shows that the naive predicate produces many incorrect results. Our filter (**b**) is more precise than FPG (**c**) but less precise than the significantly slower interval filter. Ozaki's filter produces the exact same image as our filter (colour figure online)

in CGAL show very similar performance (roughly 0.2% difference), while our filter without underflow protection is significantly faster, see (b) in Fig. 1.

By construction, our semi-static filter with underflow protection fails for all cases in which the true sign is zero, most of which can be decided by the zero-filter, though. Table 2 shows the number of filter failures in the first filters for each predicate. For a graphical comparison of the precision of 2D orientation filters, see Fig. 2.

4.2.2 3D polygon mesh processing

The next benchmark was taken from the Polygon Mesh processing benchmark in CGAL. For this benchmark, first, a 3D mesh is taken, and a polyhedral envelope with a distance δ is taken around it. The polyhedral envelope is an approximation of the Minkowski sum of the mesh with a sphere, also known as a buffer. Then, three points are repeatedly chosen in a loop, and if they form a non-degenerate triangle, it is tested whether that triangle is contained in the polyhedral envelope or not. As input, we use the file pig.off, which is provided as a sample in the CGAL tree, and for δ we chose 0.1. This is described in more detail in [22].

The algorithm makes use of the 3D orientation predicate defined as the sign of

$$p_{\text{orientation\_3}} = \begin{vmatrix} a_x - d_x & a_y - d_y & a_z - d_z \\ b_x - d_x & b_y - d_y & b_z - d_z \\ c_x - d_x & c_y - d_y & c_z - d_z \end{vmatrix} \quad .$$

No filter failures were recorded for either implementation, and no performance penalty was measured for the underflow protection. The predicates provided by CGAL caused an additional runtime of around 28% compared to our implementation, see (c) in Fig. 1.

4.2.3 3D mesh refinement

As the last benchmark, we measure the runtime of 3D mesh refinement with CGAL. The algorithm and its parameters are explained in [1]. The predicates used in this benchmark are the 3D orientation predicate and the power side of oriented power sphere predicate, which is defined as the sign of the following expression

$$p = \begin{vmatrix} a_x - e_x & a_y - e_y & a_z - e_z & (a_x - e_x)^2 + \left(a_y - e_y\right)^2 + (a_z - e_z)^2 + (e_w - a_w) \\ b_x - e_x & b_y - e_y & b_z - e_z & (b_x - e_x)^2 + \left(b_y - e_y\right)^2 + (b_z - e_z)^2 + (e_w - b_w) \\ c_x - e_x & c_y - e_y & c_z - e_z & (c_x - e_x)^2 + \left(c_y - e_y\right)^2 + (c_z - e_z)^2 + (e_w - c_w) \\ d_x - e_x & d_y - e_y & d_z - e_z & (d_x - e_z)^2 + (d_x - e_z)^2 + (d_x - e_z)^2 + (e_w - d_w) \end{vmatrix},$$

which has with $d = 5$ the highest degree of all predicates used in our benchmarks and is based on a non-homogeneous polynomial. As input file, we used elephant.off, which is provided as a sample in the CGAL source tree, with a face approximation error of 0.0068, a max facet sign of 0.003 and a maximum tetrahedron size of 0.006.

The underflow guard came with a slight performance penalty of around 1%, and the CGAL predicates were about 3.4% slower, see (d) in Fig. 1. There was a non-zero but negligible number of filter failures of around 0.1% for each of the predicates.

4.3 Error bound comparison

The following table compares error constants and error bounds for various semi-static filtering approaches for the 2D orientation predicate in the double-precision floating-point system. Underflow guards are omitted and error constants are rounded to 9 digits for readability.

Filter	Static constant	Variable component without underflow guards
[30]	$3.3306690739 \times 10^{-16}$	$\lvert (a_x \ominus c_x) \odot (b_y \ominus c_y) \rvert \oplus \lvert (a_y \ominus c_y) \odot (b_x \ominus c_x) \rvert$
[26]	$3.3306690622 \times 10^{-16}$	$\lvert (a_x \ominus c_x) \odot (b_y \ominus c_y) \oplus (a_y \ominus c_y) \odot (b_x \ominus c_x) \rvert$
Our	$3.3306690622 \times 10^{-16}$	$\lvert (a_x \ominus c_x) \odot (b_y \ominus c_y) \rvert \oplus \lvert (a_y \ominus c_y) \odot (b_x \ominus c_x) \rvert$
[24]	$8.8872057373 \times 10^{-16}$	$\max (\lvert a_x \ominus c_x \rvert , \lvert b_x \ominus c_x \rvert) \cdot \max (\lvert a_y \ominus c_y \rvert , \lvert b_y \ominus c_y \rvert)$
[8]	$8.8817841970 \times 10^{-16}$	$(\lvert a_x \rvert \oplus \lvert c_x \rvert) \odot (\lvert b_y \rvert \oplus \lvert c_y \rvert) \oplus (\lvert a_y \rvert \oplus \lvert c_y \rvert) \odot (\lvert b_x \rvert \oplus \lvert c_x \rvert)$

Since the variable component in FPG omits the addition, its error bound should be halfed for comparison to the first three filters in the table. Still, it can be seen that the approaches in FPG [24] and by Burnikel et al. [8] produce more pessimistic error bound constants than the other filters. The approach by Ozaki et al. [26] obtains a slight improvement over the error bound constant by Shewchuk [30], which we directly use in rule R_5 in Definition 5 to obtain a similar constant.

With regard to the input-dependent component, we generally obtain the same expressions as Shewchuk. In comparison, the expression by Ozaki et al. produces smaller error bounds when the products have opposite signs but in these case, there is no cancellation in the determinant computation anyway and the filter would not fail, so this mainly saves one instruction for the computation of the absolute value. The expression generated by the approach of Burnikel et al. does not use that the error of the initial differences, e.g. $a_x \ominus c_x$, can be bounded just in terms of their result, e.g. $|a_x \ominus c_x|$, and will produce much higher values for points that are relatively close to each other.

The expressions generated by FPG are very different because they are based on the idea of computing the rounding error for polynomial under the assumption that all inputs are scaled to 1 and then rescaled, using the maxima for each group of coordinates. A disadvantage of this expression is that it loses much of the polynomials original structure and, for example, produces more pessimistic estimates than the first three expressions when a and c or b and c are equal or very close.

5 Conclusion

We have presented a recursive scheme for the derivation of (semi-)static filters for geometric predicates. The approach is branch-efficient, sufficiently general to handle rounding errors, overflow and underflow and can be applied to arbitrary polynomials.

Our C++-metaprogramming-based implementation is user-friendly in so far as it requires no code generation tools, additional annotations for variables or manual tuning. This is achieved without the additional runtime overhead of previous C++-wrapper-based implementations, and our measurements show that our approach is competitive with and even outperforms the state-of-the-art in some cases.

Future work could include generalisations toward non-polynomial predicates and robust predicates on implicit points that occur as results or interim results of geometric constructions and may not be explicitly representable with floating-point coordinates.

The implementation may also be extended in the future to include further filtering stages to improve the performance for common cases of degenerate inputs.

Funding Open Access funding enabled and organized by Projekt DEAL. The work of M. Weiser was supported by DFG under Grant WE 2937/10-1. No further funding was received to assist with the preparation of this manuscript.

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

References

1. Alliez, P., Jamin, C., Rineau, L., Tayeb, S., Tournois, J., Yvinec, M.: 3D mesh generation. In: CGAL User and Reference Manual, 5.4 edn. CGAL Editorial Board (2022). https://doc.cgal.org/5.4/Manual/packages.html#PkgMesh3
2. Attene, M.: Indirect predicates for geometric constructions. Comput. Aided Des. **126**, 102,856 (2020). https://doi.org/10.1016/j.cad.2020.102856
3. Bartels, T., Fisikopoulos, V.: Fast robust arithmetics for geometric algorithms and applications to GIS. Int. Arch. Photogram. Remote Sens. Spatial Inf. Sci. **XLVI–4/W2–2021**, 1–8 (2021). https://doi.org/10.5194/isprs-archives-XLVI-4-W2-2021-1-2021
4. Bartels, T., Fisikopoulos, V., Weiser, M.: Fast floating-point filters for robust predicates (2023). https://doi.org/10.5281/zenodo.7539355
5. Berg, M.D., Cheong, O., Kreveld, M.V., Overmars, M.: Computational Geometry: Algorithms and Applications, 3rd edn. Springer-Verlag TELOS, Santa Clara (2008)
6. Brönnimann, H., Burnikel, C., Pion, S.: Interval arithmetic yields efficient dynamic filters for computational geometry. In: Proc. of the 14th Annual Symposium on Computational Geometry, pp. 165–174. ACM, USA (1998). https://doi.org/10.1145/276884.276903
7. Brönnimann, H., Fabri, A., Giezeman, G.J., Hert, S., Hoffmann, M., Kettner, L., Pion, S., Schirra, S.: 2D and 3D linear geometry kernel. In: CGAL User and Reference Manual, 5.2.1 edn. CGAL Editorial Board (2021). https://doc.cgal.org/5.2.1/Manual/packages.html#PkgKernel23
8. Burnikel, C., Funke, S., Seel, M.: Exact geometric computation using cascading. Int. J. Comput. Geom. Appl. **11**(03), 245–266 (2001). https://doi.org/10.1142/S0218195901000493
9. de Matos Menezes, M., Magalhães, S.V.G., de Oliveira, M.A., Franklin, W.R., de Oliveira Bauer Chichorro, R.E.: Fast parallel evaluation of exact geometric predicates on GPUs (2021). Submitted
10. Devillers, O., Fronville, A., Mourrain, B., Teillaud, M.: Algebraic methods and arithmetic filtering for exact predicates on circle arcs. In: Proc. of the 16th Annual Symposium on Computational Geometry, pp. 139–147. ACM, USA (2000). https://doi.org/10.1145/336154.336194
11. Devillers, O., Pion, S.: Efficient exact geometric predicates for Delaunay triangulations. In: R.E. Ladner (ed.) Proceedings of the 5th Workshop on Algorithm Engineering and Experiments, Baltimore, MD, USA, January 11, 2003, pp. 37–44. SIAM (2003)
12. Dimov, P.: Boost C++ libraries: Mp11, version 1.76 (2021). https://boost.org/libs/mp11
13. Fisikopoulos, V., Peñaranda, L.: Faster geometric algorithms via dynamic determinant computation. Comput. Geom. **54**, 1–16 (2016). https://doi.org/10.1016/j.comgeo.2015.12.001

14. Gehrels, B., Lalande, B., Loskot, M., Wulkiewicz, A., Karavelas, M., Fisikopoulos, V.: Boost C++ libraries: Geometry, version 1.76 (2021). https://boost.org/libs/geometry
15. Granlund, T.: The GMP development team: GNU MP: The GNU Multiple Precision Arithmetic Library, 5.0.5 edn. (2012). http://gmplib.org/
16. Hauser, J.R.: Handling floating-point exceptions in numeric programs. ACM Trans. Program. Lang. Syst. **18**(2), 139–174 (1996). https://doi.org/10.1145/227699.227701
17. Hemmer, M., Hert, S., Pion, S., Schirra, S.: Number types. In: CGAL User and Reference Manual, 5.4 edn. CGAL Editorial Board (2022). https://doc.cgal.org/5.4/Manual/packages.html#PkgNumberTypes
18. 754-2008 – IEEE standard for floating-point arithmetic. IEEE (2008). https://doi.org/10.1109/IEEESTD.2008.4610935
19. Jamin, C., Alliez, P., Yvinec, M., Boissonnat, J.D.: Cgalmesh: a generic framework for Delaunay mesh generation. ACM Trans. Math. Softw. (2015). https://doi.org/10.1145/2699463
20. Kettner, L., Mehlhorn, K., Pion, S., Schirra, S., Yap, C.: Classroom examples of robustness problems in geometric computations. In: Algorithms – ESA 2004, pp. 702–713. Springer, Berlin (2004). https://doi.org/10.1007/978-3-540-30140-0_62
21. Li, Z., Zhu, C., Gold, C.: Digital Terrain Modeling: Principles and Methodology. CRC Press (2005). https://doi.org/10.1201/9780203357132
22. Loriot, S., Rouxel-Labbé, M., Tournois, J., Yaz, I.O.: Polygon mesh processing. In: CGAL User and Reference Manual, 5.4 edn. CGAL Editorial Board (2022). https://doc.cgal.org/5.4/Manual/packages.html#PkgPolygonMeshProcessing
23. Maddock, J., Kormanyos, C.: Boost C++ libraries: Geometry, version 1.76 (2021). https://boost.org/libs/multiprecision
24. Meyer, A., Pion, S.: FPG: a code generator for fast and certified geometric predicates. In: Real Numbers and Computers, pp. 47–60. Santiago de Compostela, Spain (2008). https://hal.inria.fr/inria-00344297
25. Nanevski, A., Blelloch, G., Harper, R.: Automatic generation of staged geometric predicates. Higher-Order Symb. Comput. LISP Symb. Comput. **16**(4), 379–400 (2003). https://doi.org/10.1023/a:1025876920522
26. Ozaki, K., Bünger, F., Ogita, T., Oishi, S., Rump, S.M.: Simple floating-point filters for the two-dimensional orientation problem. BIT Numer. Math. **56**(2), 729–749 (2016). https://doi.org/10.1007/s10543-015-0574-9
27. Qi, M., Yan, K., Zheng, Y.: Gpredicates: Gpu implementation of robust and adaptive floating-point predicates for computational geometry. IEEE Access **7**, 60868–60876 (2019). https://doi.org/10.1109/ACCESS.2019.2911641
28. Rump, S.M.: Error estimation of floating-point summation and dot product. BIT Numer. Math. **52**(1), 201–220 (2011). https://doi.org/10.1007/s10543-011-0342-4
29. Shewchuk, J.: Routines for arbitrary precision floating-point arithmetic and fast robust geometric predicates (1996). https://cs.cmu.edu/afs/cs/project/quake/public/code/predicates.c
30. Shewchuk, J.R.: Adaptive precision floating-point arithmetic and fast robust geometric predicates. Discrete Comput. Geometry **18**(3), 305–363 (1997). https://doi.org/10.1007/pl00009321
31. Shewchuk, J.R.: Tetrahedral mesh generation by Delaunay refinement. In: Proceedings of the Fourteenth Annual Symposium on Computational Geometry, SCG'98, pp. 86-95. Association for Computing Machinery, New York (1998). https://doi.org/10.1145/276884.276894
32. Sunday, D.: Practical Geometry Algorithms: with C++ Code. Amazon Digital Services LLC (2021). ISBN: 9798749449730
33. Vassilev, V., Canal, P., Naumann, A., Moneta, L., Russo, P.: Cling—The New Interactive Interpreter for ROOT 6. p. 052071. IOP Publishing (2012). https://doi.org/10.1088/1742-6596/396/5/052071
34. Veldhuizen, T.: Expression templates. C++ Rep. **7**(5), 26–31 (1995)