FAST ROBUST ARITHMETICS FOR GEOMETRIC ALGORITHMS AND APPLICATIONS TO GIS

Tinko Bartels[a,*], Vissarion Fisikopoulos [b]

[a] Technical University of Berlin, Germany - t.bartels@tu-berlin.de [b] Oracle, Greece - vissarion.fysikopoulos@oracle.com

KEY WORDS: Spatial Predicates, Delaunay Triangulation, Floating-Point, Robustness, Triangulated Irregular Network.

ABSTRACT:

Geometric predicates are used in many GIS algorithms, such as the construction of Delaunay Triangulations for Triangulated Irregular Networks (TIN) or geospatial predicates. With floating-point arithmetic, these computations can incur roundoff errors that may lead to incorrect results and inconsistencies, causing computations to fail. This issue has been addressed using a combination of exact arithmetics 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 show examples of how this approach produces correct results for GIS data sets that could lead to incorrect predicate results for naive implementations. We also show benchmark results that demonstrate that our implementation can compete with state-of-the-art solutions.

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 and Delaunay triangulation. 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 GIS applications. On the other hand, computing with floating-point arithmetic, these computations can incur roundoff errors that may lead to incorrect results and inconsistencies, causing computations to fail (Kettner et al., 2004).

Among other applications, Delaunay triangulations are important for the construction of Triangulated Irregular Networks (TIN). TINs are used in GIS applications to represent terrains in Digital Elevation Models and to produce Digital Surface Models or Digital Terrain Models, as discussed in (Li et al., 2005). Predicate failures in the underlying Delaunay triangulation may lead to issues with the mesh quality and cause crashes due to invalid triangulations or failure to terminate, as discussed in (Shewchuk, 1997). The issue of predicate robustness is therefore not limited to use cases with high precision requirements.

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 violates basic assumptions on partitioned spaces.

Switching to exact computations can guarantee correct results but is very slow for practical purposes. To improve performance, these computations are made adaptive in the sense that exact arithmetic is only performed if a priori error estimates 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 (almost) static, semi-static and dynamic. In the first case, the error is pre-computed very efficiently using a priori bounds on the input but attains very low accuracy. In semi-static filters, the error estimation depends on the input. They are still a bit slower than static filters and improve a bit on the accuracy and require no a priori bounds on the input. The slowest and more accurate are the dynamic filters that use 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 (Shewchuk, 1997), Shewchuk introduced robust, adaptive implementations for orientation-, incircle- and insphere-predicates that can be used, for example, in the construction of Delaunay triangulations. He uses 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 (Brönnimann et al., 1998). For Delaunay triangulations (Devillers and Pion, 2003) propose a set of efficient static and semi-static filters and experimentally compare them with several alternatives including (Shewchuk, 1997). In (Meyer and Pion, 2008), the authors present FPG, a general purpose code analyzer and generator for almost static filtered predicates. In (Nanevski et al., 2003), they extend Shewchuk's method to arbitrary polynomial expressions and implement an expression compiler that takes a function and produces a program that computes the sign of the source function at any given floating points arguments. More recently, in (Ozaki et al., 2016), they develop a new semi-static floating-point filter for the 2D orientation predicate, which handles

* Corresponding author

floating-point exceptions such as overflow and underflow, as well as an improved fully-static filter. Regarding non-linear geometries, there is work on filters for circular arcs (Devillers et al., 2000). Moreover, robust predicates could be extended to provide robust constructions such as points of intersection of linestrings (Attene, 2020).

Our contribution. The contribution of this paper is two-fold. First, we present a new implementation based on C++ meta-programming techniques that produces fast, robust predicates at compile-time for arbitrary, polynomial computations. It is extensible and based on the C++ library Boost.Geometry (Gehrels et al., 2021). The main advantage of our proposed implementation is the ability to automatically generate filters based on some expression. On the other hand, it can be used with manual hand-crafted filters, as illustrated by the use of our axis-aligned filter for the incircle predicate (Section 4.2).

Second, we perform an experimental analysis of our generated filters as well as a comparison with the state-of-the-art. We perform our benchmarks with synthetic as well as real datasets arising in GIS, e.g. a Triangulated irregular network representing Maribor city in Slovenia (Špelič et al., 2008). Following our, analysis we propose staged orientation and incircle predicates, i.e. predicates that use a pipeline of filters. Our predicates outperform the current implementations of state-of-the-art libraries (Shewchuk, 1996, Brönnimann et al., 2021). Finally, we highlight the importance of robust predicates by showing a number of issues and inconsistencies in the course of standard GIS algorithms (e.g. see Table 5) that can be fixed by using robust predicate while adding a small computational cost.

2. ROBUST GEOMETRIC PREDICATES

In this section we define and explain all the preliminaries needed to present our implementation method in the next section.

2.1 Geometric Predicates and Robustness Issues

In the context of this paper we define geometric predicates to be functions that return yes or no answers to geometric questions based on evaluating the sign of a polynomial. One example is the planar orientation predicate. Given three points a, b and 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

$$p\left(a_x, a_y, b_x, b_y, c_x, c_y\right) \coloneqq \begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix} \quad (1)$$

For this definition of the orientation predicate, postive, 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. Consider a floating-point number (FPN) system F and let $\mathrm{rd} : \mathbb{R} \to F$ be the rounding-function, which maps any number to its nearest floating-point representation in F, breaking ties towards the number with an even mantiss. We denote the floating-point operators on F by $\oplus$, $\ominus$ and $\otimes$, with $a \circledcirc b \coloneqq \mathrm{rd}\left(a \circ b\right)$ for $\circ \in \{+, -, \cdot\}$. We call

$$\tilde{p}\left(a_x, a_y, b_x, b_y, c_x, c_y\right) \coloneqq (a_x \ominus c_x) \otimes (b_y \ominus c_y) \ominus (a_y \ominus c_y) \otimes (b_x \ominus c_x) \quad (2)$$

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

Architecture	$\tilde{c}$ and t_1	c and t_2	c and $t_1 \cup t_2$
-march=haswell	outside	outside	inside
-march=ivybridge	touch	touch	inside
exact	inside	outside	inside

Table 1. Relationships of point $c = (0, -0.01)$ to polygon $\tilde{t}_1 \coloneqq \{(-1, 0), \tilde{a}, \tilde{b}\}$ and $t_2 \coloneqq \{(1, 0), \tilde{b}, \tilde{a}\}$, where $a = (-0.01, -0.59)$, $b = (0.01, 0.57)$.

As an illustration, 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 double precision, $\tilde{a}, \tilde{b}, \tilde{c}$, however, are not collinear but close to collinear which makes the case sensitive to rounding errors.

As a second example, let us evaluate the spatial predicate within, defined as 'inside' in (Egenhofer and Herring, 1990), for c and the polygons $\tilde{t}_1 \coloneqq \{(-1, 0), \tilde{a}, \tilde{b}\}$ and $t_2 \coloneqq \{(1, 0), \tilde{b}, \tilde{a}\}$ using the winding-number algorithm (Sunday, 2021). 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 internally inconsistent. The final row can be obtained using any implementation of the orientation predicate that guarantees correct results, such as the implementation of Shewchuk (Shewchuk, 1996) or CGAL's kernels epick or epeck (Brönnimann et al., 2021).

Remark 1. The difference between the architecture is due to GCC producing an assembly involving the FMA-instruction from polynomial (1). FMA can be defined as $\mathrm{FMA}\left(a, b, c\right) \coloneqq \mathrm{rd}\left(a \cdot b + c\right)$. This instruction causes loss of anti-commutativity for difference, i.e. $a \otimes b \ominus c \otimes d = -c \otimes d \ominus a \otimes b$ holds if no range errors occur, but $\mathrm{FMA}\left(a, b, -c \otimes d\right) = -\mathrm{FMA}\left(c, d, -a \otimes b\right)$ is not necessarily true. When inserted into the orientation predicate, this can lead to situations in which swapping two input points does not necessarily reverse the sign of the result.

Inconsistencies can occur without FMA as well. Consider $\tilde{a}, \tilde{b}, \tilde{d} \coloneqq (\mathrm{rd}\left(0.15\right), \mathrm{rd}\left(8.69\right))$ and $\tilde{e} \coloneqq (\mathrm{rd}\left(0.07\right), \mathrm{rd}\left(4.05\right))$. The floating-point realisation (2), 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.

2.2 Exact Arithmetic

A natural idea to solve the precision issues of floating-point arithmetics would be to perform the evaluations at a higher precision. For example, it can be sensible to perform calculations with single-precision inputs using double-precision. This approach has immediate limitations. Consider, for example, 2^{40} and 2^{-40}, which are both representable in single precision (32 bit). Their exact sum would require more than 80 bits just for the significant to be represented as a floating-point number. As soon as the precision is extended beyond the floating-point number systems for which operations are hardware-accelerated

the time cost of elemantary operations can increase by one or two orders of magnitude.

Floating-point expansions are an approach to overcome the precision limitations of FPN. The idea is to represent values that are not representable in a FPN system themselves as sums of components that are individually representable in the original system. E.g., the result of $2^{40} + 2^{-40}$ can be represented as the two-component expansion $\left(2^{-40}, 2^{40}\right)$. This technique is sometimes called double-double arithmetic, for expansions with two components, or double-quad arithmetic for expansions with four components. The computation of such expansions can be expressed in terms of elementary operations in the original FPN system and, hence, make use of hardware-acceleration. The application of floating-point expansions for the exact evaluation of geometric predicates was described in (Shewchuk, 1997).

Definition 2 (Expansion)**.** For a binary FPN system F, an expansion is a tuple of n floating-point numbers $x_1, \ldots, x_n \in F$, which are called components. An expansion is a, not necessarily unique, representation of a real number $x = x_1 + \ldots + x_n$ that can be represented in binary with finitely many digits but is not necessarily in F. We call an expansion $x_1, \ldots, x_n$ ordered (by increasing magnitude with possibly interspersed zeroes) if $|x_i| \leq |x_j|$ or $x_i = 0$ or $x_j = 0$ for all $1 \leq i \leq j \leq n$. We call two components $|x| \leq |y|$ nonoverlapping if the least significant nonzero bit of y is more significant than the most significant digit of x. We call an ordered expansion nonoverlapping if that property holds for each pair of components.

We previously mentioned that exact computations on expansions can be performed based on elementary floating-point operations. For illustration purposes we will show an example of this using the algorithm that is presented as Fast-Two-Sum in (Shewchuk, 1997) and due to (Dekker, 1971) to compute the exact sum of two floating-point numbers.

Example 3 (Expansion)**.** This example will assume radix 2, 5 bits of precision and for readability we will give all numbers in fixed-point notation. Consider the numbers $a = 10101$ and $b = 1.01$. The FP approximation $x \coloneqq a \oplus b$ of the result is 10110 after rounding and the exact result is $a + b = 110110.01$ with 0.01 being the round-off error. All less significant digits of b, the number with the smaller exponent, are necessarily rounded off in the result. We can make use of that notion by first computing the part of b that is not rounded off in the result via $b_{\text{virtual}} \coloneqq x \ominus a = 1$. Note, that with exact operations or sufficiently high precision that number would always be 0 because there would be no round-off error. Using b_{virtual} we then obtain the exact round-off error as $y \coloneqq b \ominus b_{\text{virtual}} = 0.01$. The expansion exactly representing $a + b$ is then $(10101, 0.01)$.

Further algorithms for the exact summation and multiplication of floating-point numbers and expansions, including invariants and proofs of correctness, are discussed in (Shewchuk, 1997). The exact evaluation of $+$, $-$ and $\cdot$ for two floating-point numbers can be performed using constant time, constant auxiliary storage and produces expansions of, at most, length 2. The algorithm Grow-Expansion computes the sum of an expansion of length n and a floating-point number using linear time, constant auxiliary storage and producing an expansion of, at most, length $n + 1$. The algorithm Fast-Expansion-Sum computes the sum of two expansions of lengths n and m producing an expansion of, at most, length $n + m$ and has similar performance characteristics to the merge-step in MergeSort. The algorithm Scale-Expansion computes the product of an expansion of length n and a floating-point number using linear time, constant auxiliary storage and producing an expansion of, at most, length $2n$. By repeatedly applying Scale-Expansion and Fast-Expansion-Sum, the product of two expansions of lengths n and m can be computed in $\mathcal{O}(mn \cdot \log mn)$, resulting in an expansion of length, at most, $2mn$.

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 $\otimes$ 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 it produces a robust but computationally expensive implementation, even for simple predicates. To mitigate this issue we make use of filters.

Definition 4 (Filter)**.** For a predicate $\text{sign}\left(p\left(x_1, \ldots, x_n\right)\right)$ and an FP 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 $f\left(x_1, \ldots x_n\right) = \text{sign}\left(p\left(x_1, \ldots, x_n\right)\right)$ or $f\left(x_1, \ldots, x_n\right) = \text{uncertain}$ holds. The latter case is referred to as filter failure.

Adopting the terminology used in (Devillers and Pion, 2003), 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 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. We will now present some filters as examples.

Example 5 (Shewchuk's Stage A orientation predicate)**.** Consider the predicate (1) and its floating-point realisation (2). Then,

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

with the error bound

$$e\left(a_x, \ldots, c_y\right) \coloneqq \left(3\epsilon + 16\epsilon^2\right) \otimes \left(\left|\left(a_x \ominus c_x\right) \otimes \left(b_y \ominus c_y\right)\right| \oplus \left|\left(a_y \ominus c_y\right) \otimes \left(b_x \ominus c_x\right)\right|\right),$$

where $\tilde{p} \coloneqq \tilde{p}\left(a_x, \ldots, c_y\right)$ and ϵ is the machine-epsilon of the FPN, is a valid filter for all inputs that do not cause overflow or denormalisation (Shewchuk, 1997).

This filter can be considered semi-static filter with its static component being $3\epsilon + 16\epsilon^2$. 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 6 (FPG orientation filter)**.** Consider predicate (1) and its floating-point realisation (2). Let

$m_x := \max\{|a_x \ominus c_x|, |b_x \ominus c_x|\}$ and $m_y := \max\{|a_y \ominus c_y|, |b_y \ominus c_y|\}$. If $\max\{m_x, m_y\} > 1.67597599124282407923e + 153$, $0 \neq \min\{m_x, m_y\} \geq 5.00368081960964690982e - 147$ or $|\tilde{p}| \leq 8.88720573725927976811e - 16 \otimes m_x \otimes m_y \neq 0$, then "uncertain" is returned, otherwise the sign of $\tilde{p}$ is returned.

This filter is generated by the program FPG, which is presented in (Meyer and Pion, 2008) for the 2D orientation predicate. It is valid with double precision arithmetic for all double precision inputs.

This filter is also semi-static with the static component of the error bound being $8.88720573725927976811e - 16$. A fully 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 homogenous polynomials if group annotations for the input variables are provided. 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.

Another example of a semi-static error bound filter for the 2D orientation predicate, that can handle overflow, underflow and rounding-errors with less branches than the filter generated by FPG, can be found in (Ozaki et al., 2016).

The next example is not strictly an error-bound filter.

Example 7 (Shewshuk's stage B orientation predicate)**.** Consider the predicate 1 and its floating-point realisation 2. 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 (Shewchuk, 1997) and is valid for all inputs that do not produce overflow or underflow. The full version in (Shewchuk, 1997) also includes an error-bound 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 fully dynamic filter.

Example 8 (Interval arithmetic filter)**.** Consider a predicate and one of its floating-point realisations. Given the inputs, compute for each floating-point operation $\oplus, \ominus, \otimes$ 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 (Brönnimann et al., 1998).

In (Devillers and Pion, 2003), (Ozaki et al., 2016) and (Shewchuk, 1997) failure probabilities and performance experiments for various sequences of filters, types of inputs and algorithms are presented. We will present our own experiments in Section 4.

```
template <std::size_t Index>
struct argument { /*...*/ };
template <typename Float>
struct constant
{
  static constexpr Float
    value = /*...*/
};
template <typename Left, typename Right>
struct sum :
  public internal_binary_node<Left, Right>
{ /*...*/ };
template <typename Left, typename Right>
struct difference :
  public internal_binary_node<Left, Right>
{ /*...*/ };
template <typename Left, typename Right>
struct product :
  public internal_binary_node<Left, Right>
{ /*...*/ };
```

Figure 1. Templates for the representation of expression trees

3. NEW META-PROGRAMMING IMPLEMENTATION

3.1 Code design and expression trees

After having described a number of filters as well as algorithms for the exact evaluation, we will now describe the framework we implemented for the generation of filtered predicates in C++. At the core of our framework are expression trees. Our expression trees consist of internal binary or unary nodes and leaves. Binary nodes represent operations such as $+, -, \cdot, \max, \min$. Our only current unary node template represents the operation abs $(\cdot)$. Leaves represent indexed arguments, which are placeholders for input values, and constants. Trees are represented in the C++ type system as class templates. There is a class template for each node type that requires one type argument for unary nodes or two type arguments for binary nodes, which represent the subtrees below the node. Arguments are represented by class templates that require an index as template argument. The implementation is open source and publicly available: `https://github.com/BoostGSoC20/geometry`

Figure 1 shows some of the templates in our binary expression trees. Figure 2 illustrates how they can be used later to represent the expression of the 2D orientation predicate of expression (1) in the C++ type system.

3.2 Evaluation of expression trees

To evaluate the expression trees in given arguments, our framework provides two function templates. The function `evaluate_expression` takes an expression and an array of input values as parameters and evaluates the expression in the given inputs using the given type. This is realized by first obtaining a post-order traversal of the expression tree as a type list and then eliminating all duplicates and leaves. The remaining type list then contains expression trees representing all interim results in the order in which they need to be computed.

The expression tree processing is performed at compile-time and does not affect performance. Because the number of interim computations is also known at compile-time, the interim results can be stored on the stack and no dynamic heap-allocations are required, as long as simple types like float or double without internal heap-allocations are used.

The design of the framework is generic. Therefore any input type that supports all operators used in the expression tree can be used with the function template. For floating-point types like double or float, this function template can be used to obtain floating-point approximations of polynomials or to compute error bounds for known error expressions. If the function template is called with an interval arithmetic type, it can be used to provide a dynamic filter, like in Example 8. Finally, the function can yield an exact filter useful for the final stage of a staged predicate by using an arbitrary precision number type like CGAL::Gmpzf, which is based on GMP library (Granlund and the GMP development team, 2012).

The second function template that we provide to evaluate expression trees is `eval_expansions`. It also processes expression trees in post-order and computes the total number of components for all interim expansions at compile-time. The operations $+$, $-$, $\cdot$ are replaced by the appropriate expansion arithmetic algorithms. The function template accepts policies that allow finer control of implementation details of expansion arithmetics, in particular regarding zero-elimination and the choice of summation algorithms, see the end of Section 2.4 of (Shewchuk, 1997) for a brief discussion of these aspects.

Because we again know the exact number of components at compile-time, we can perform all computations without heap allocations. For very complicated expressions, the space on the stack may be insufficient. A possible future version could include thresholds for when to chose heap storage over stack storage for expansion components.

As straight-forward applications of the `eval_expansions` function template and the expression trees, we provide the templates `stage_d` and `stage_b`. Both get an expression tree and a calculation type as template arguments and provide a variadic static apply method. The apply method of the `stage_d` class template accepts a number of values and evaluate the expression exactly using expansion arithmetic. Then returns the sign of the expression evaluated in the inputs. The `stage_b` template first performs all interim computations that involve differences of two arguments (leaves). If all can be performed without round-off error, the remaining expression is evaluated using expansion arithmetic but simplified, because it is known that the difference nodes of height 1 only have one component each, rather than two. If a round-off error occurs within one of the differences of height 1, a value indicating uncertainty is returned. This template therefore implements the filter of Example 7 for arbitrary polynomials.

3.3 Error bounds

For error bound filters, our framework provides two class templates. The template `stage_a_error_bound_expression` takes an expression tree involving $+$, $-$ and $\cdot$ and a calculation type. The tree is then processed at compile-time using forward error analysis in a manner similar to the reasoning in section 4.3 of (Shewchuk, 1997). The result is again given as an expression tree, representing an error bound consisting of a constant, that is a polynomial in ϵ, and a scaling expression involving the inputs. For the expression of the 2D orientation predicate, the error bound expression that is produced is equivalent to the error bound in example 5.

The second error bound template that we provide is named `fpg_error_expression`. It takes as arguments an expression tree representing a homogenous polynomial, a calculation type and, optionally, a type list representing groups of input arguments. If no groups are provided, the template defaults to automatically computed groups. The automatic grouping heuristic processes the expression tree by performing a `constexpr` BFS algorithm on the arguments, considering two arguments adjacent if leaves representing them, appear as children of the same difference node. The error expression is then obtained by decomposing the expression tree into summands and greedily assigning the arguments in the summands to groups. Based on this assignment, an error expression is then computed at compile-time according to the rules described in in (Meyer and Pion, 2008). For the 2D orientation filter, the error expression is equivalent to the error expression in example 6. Unlike the original FPG code generator, our implementation does not support range checks against underflow or overflow. However, our proposed framework can be used to implement such checks.

3.4 Static filters

We provide three templates for use with the above error expressions. The template `semi_static_filter` takes an expression tree, a calculation type and an expression tree representing an error expression, for example obtained using the two previously mentioned templates. The class template provides a static apply method that takes input values and evaluates the expression tree and the error expression tree in these inputs. If the result of the expression is larger or equal in magnitude than the error expression, then its sign is returned. Otherwise a value indicating uncertainty is returned. The templates `stage_a_semi_static` and `fpg_semi_static` are aliases for `semi_static_filter` with error expressions generated from `stage_a_error_bound_expression` and `fpg_error_expression` respectively.

The template `interval` takes an error expression in the input and transforms it at compile-time using interval arithmetic rules. The result is an expression tree that represents the maximum of this error expression, given global upper and lower bounds on the input. This error expression can then be used with the template `static_filter`, which, unlike `semi_static_filter` represents a stateful class that computers its error bound at construction rather than for each call to the apply method. It can be used to generate static filters. A second template, `almost_static_filter` takes an expression, a calculation type and a `static_filter` type, which it stores in its internal state an instance of the given `static_filter` type given the current bounds on the input. If these bounds change, the internally stored `static_filter` is updated to compute a new global error bound.

3.5 Staged predicates

Finally, we provide the template `staged_predicate`. It is a variadic template that accepts a calculation type and an arbitrary number of type arguments. The types are expected to provide the shared interface of the previously described filters and exact staged. The templates optionally take bounds on the input at construction, which is only sensible if its staged predicate contains at least one static filter. It also optionally allows updates to its bounds, which is sensible if it contains an almost static filter. Its apply method will run each of its stages in order until a value other than "uncertain" is returned. Notably, the provided filter types need not be variants of the filters described above. Besides our filters that are automatically generated based on the expression, the user of our framework can also use handcrafted predicates. Figure 2 shows an example of how to generated a two-staged predicate for the 2D orientation problem.

```
using _1 = argument<1>;
/* ... */
using orient2d =
  difference<
    product<
      difference<_1, _5>,
      difference<_4, _6>
    >,
    product<
      difference<_3, _5>,
      difference<_2, _6>
    >
  >;
using predicate_type =
  staged_predicate<
    stage_a_semi_static<orient2d, double>,
    stage_d<orient2d, double>
  >;
predicate_type predicate;
/* ... */
int orientation =
  predicate.apply(ax, ay, bx, by,
                  cx, cy);
```

Figure 2. Example code for the generation of a robust orientation predicate

4. EXPERIMENTS

We perform an experimental analysis of our implementation with synthetic and real data. First we compare the performance and accuracy of various filters. Second, we present performance comparison of filters used in Delaunay triangulation computations.

All benchmarks were performed on a Linux Workstation powered by an Intel Xeon CPU E3-1505M v5 @ 2.80Ghz. For consistency, turbo boost was disabled for all runs. All benchmarks were compiled with GCC 11.1, Boost 1.75, CGAL 4.14.3 and GMP 6.2.1. Timings were recorded for GCC with level 2 optimizations and native optimizations. For each configuration eight timings were recorded. Deviation between runs with the same settings was negligible, so we will show median results for simplicity. Because preliminary tests found little or no performance advantages for static filters over semi-static filters for the data sets that we considered, we omit them from the following experiments.

We tested 2D orientation and incircle predicates using the 2D Delaunay triangulation provided by the CGAL library. We compared various filters generated using our templated implementation against non robust predicates, Shewchuk's predicates and the exact predicates, inexact construction kernel provided by CGAL. Shewchuk's predicates consist of four stages, named A, B, C and D and their implementation can be found in (Shewchuk, 1996). The exact predicates, inexact construction kernel of the CGAL library provides predicates consisting of three stages. The first stage is a semi-static filter based on FPG with range-checks against overflow and underflow. The second stage uses a dynamic filter based on interval arithmetic. The final stage uses exact evaluation based on a high-precision number type.

4.1 Single filter experiments

For the first experiment, we perform a 2D Delaunay triangulation of uniformly distributed random points with coordinates in $[1, 2]$. This distribution and interval was chosen because it guarantees filter successes for all tested filters. Table 2 shows the averaged time per predicate call over the non-robust predicate, which is always the fastest. Clearly, the semi static filter based on Shewchuk's Stage A is the fastest generated filter. Among the two exact stages, stage_d and the CGAL Gmpzf based computation, stage_d is the faster. It should be noted that the exact evaluation based on CGAL Gmpzf has the advantage that it can also guarantee correct results in cases of overflow.

Timings for 2D orientation predicates	
Filter/Stage	added time (ns)
stage_a_semi_static	6.6
fpg_semi_static	8.5
stage_b	33.9
evaluate_expression (CGAL Interval_nt)	37.2
stage_d	44.8
evaluate_expression (CGAL Gmpzf)	361.9

Table 2. Time added by a filter over a non-robust implementation of the 2D orientation predicate during the computation of a Delaunay triangulation of uniformly distributed points.

Filter accuracy for axis-aligned grid points	
Filter/Stage	Accuracy
stage_a_semi_static	0.997
fpg_semi_static	0.996
stage_b	0.990
evaluate_expression (CGAL Interval_nt)	0.997
stage_d	1
evaluate_expression (CGAL Gmpzf)	1

Table 3. The percentage of successful filter calls i.e. not return uncertain. Results obtained for the computation of the Delaunay triangulation of points on an axis-aligned grid.

Regarding filter accuracy, Table 3 presents filter failures for the triangulation of points aligned to a grid. The distance between two adjacent points was set to 0.1. The test sets consist of 100K points. Interestingly, stage_a_semi_static is also the most accurate among the non exact filters.

For the 2D incircle predicate, the situation is more complicated. We test predicates generated from two alternative formulations of incircle the predicate.

$$\begin{aligned}\tilde{p}_{\text{I2D}} = det(&a_x - d_x,\ a_y - d_y,\ (a_x - d_x)^2 + (a_y - d_y)^2\,,\\ &b_x - d_x,\ b_y - d_y,\ (b_x - d_x)^2 + (b_y - d_y)^2\,,\\ &c_x - d_x,\ c_y - d_y,\ (c_x - d_x)^2 + (c_y - d_y)^2).\end{aligned}$$

and

$$\begin{aligned}\tilde{p}_{\text{I2Ds}} = det(&(a_x - d_x)\,(c_y - d_y) - (a_y - d_y)\,(c_x - d_x)\,,\\ &(c_x - d_x)\,(c_x - a_x) + (c_y - d_y)\,(c_y - a_y)\,,\\ &(a_x - d_x)\,(b_y - d_y) - (a_y - d_y)\,(b_x - d_x)\,,\\ &(b_x - d_x)\,(b_x - a_x) + (b_y - d_y)\,(b_y - a_y)).\end{aligned}$$

The second expression was referred to as a simplified form in the CGAL source code, so we will refer to it as the simplified (S) incircle expression. We test filters for the simple data set of uniformly distributed points (Table 4).

Notably, the simplified expression is preferable across the board especially for the exact filters. Interestingly, the performance gap between the stage A filter and the exact ones is much larger than the orientation case.

Timings for 2D incircle predicates	
Filter/Stage	added time (ns)
stage_a_semi_static	1.5
stage_a_semi_static (S)	1.5
fpg_semi_static	6.6
fpg_semi_static (S)	5.0
evaluate_expression (CGAL::Interval_nt)	116.6
evaluate_expression (CGAL::Interval_nt) (S)	110.8
stage_d	871.9
stage_d (S)	558.7
evaluate_expression (CGAL Gmpzf)	1789.0
evaluate_expression (CGAL Gmpzf) (S)	1568.0

Table 4. Time added by a filter over a non-robust 2D incircle implementation during the computation of a Delaunay triangulation of uniformly distributed points.

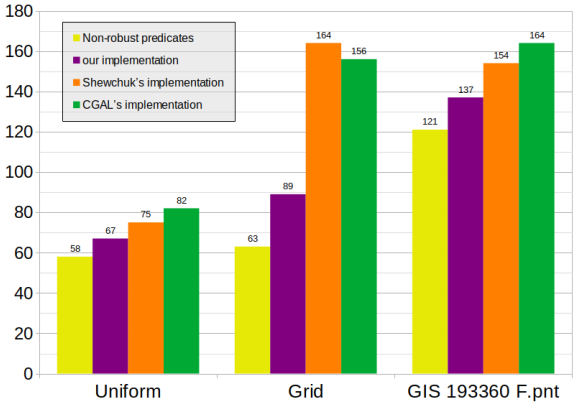

Figure 3. Performance times for the construction of the Delaunay triangulation of three datasets. Timings in ms.

4.2 Staged predicate experiments

Now we study the performance of using staged predicates, i.e. pipelines of filters. For the orientation case the staged predicate we propose is simple, consisting of the stage A filter, directly followed by the exact stage D filter.

For the incircle predicate, the performance gap observed in Table 4 indicates the use of further filters between the stage A and D filters. Experiments showed that the most common failure cases for the incircle predicate were found in artificial datasets that include axis aligned rectangles. Thus, we add a custom handwritten filter to catch this case in particular. Another filter that proved particularly helpful with degenerate examples was the dynamic filter based on CGAL interval type, which is still five times as fast as stage D. Now the staged predicate for incircle predicate consist of four filters in this order: the stage A semi-static filter, our axis-aligned-rectangle-filter, a dynamic interval filter and the exact stage D filter. We perform comparison experiments against the state of the art methods.

Delaunay triangulation benchmarks were performed with uniformly distributed points, points aligned on a non-integer grid (step size 0.1) and the dataset GIS_193360_F.pnt from (Špelič et al., 2008). The non robust predicates is consistently the fastest across all data sets while our implementation is always the fastest among the robust ones (Figure 3). The cost of robustness with our predicates amounted to about 13-15 percent over the non-robust predicates for the uniformly distributed points and the GIS data set and around 40 percent for the more challenging grid data set. For both the grid and the GIS data set, incorrect predicate calls were recorded for non robust predicates, regardless of compiler and optimization level, so the performance advantage of the non-robust predicates comes at the cost of possibly incorrect Delaunay triangulations. No crashes were recorded for any test.

Overall, we can see that there is no single filter template, that is optimal. The stage A filter delivered very good performance overall, and we always propose it as a first stage. For the 2D orientation predicate it is sufficient to just add a stage_d filter in the staged predicate. For the 2D incircle predicate, it makes sense to add a dynamic interval filter and an axis-aligned box test, between stage A and D.

This picture might change, if overflow and underflow are concerned. In this case, it makes sense to use an arbitrary-precision number type like Gmpzf for exact evaluation. While none of our filters includes overflow detection, it would probably make sense to add this test to the code that produces FPG-like error bounds, according to the rules given in the description of the original FPG in (Meyer and Pion, 2008).

We also tested the spatial test for the point in polygon problem. The polygon is extracted from the large scale countries vector data set by Natural Earth, introduced in (Kelso and Patterson, 2010), and represents the largest continental component of Russia. Making this test robust, does add about 33% to the runtime cost for randomly generated points (Figure 4). In the case of points that are generated to be close to the boundary the runtime difference shrinks to less than 10%. We conjecture that this is due to the non robust version failing to recognize collinear points which produces an incorrect result and also misses an opportunity to terminate the computation early.

We recorded the number of "touch"-results for the points that are close to the boundary. This number is inconsistent across algorithms and possible compiler settings if non robust predicates are used (Table 5). With robust predicates, consistent results are guaranteed.

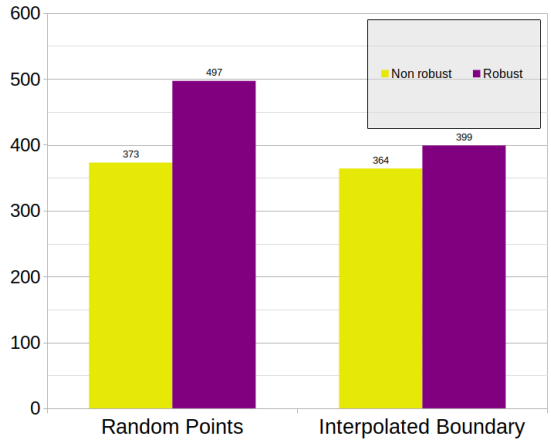

Figure 4. Timings in ms for testing whether a point is in continental Russia polygon from (Kelso and Patterson, 2010).

5. CONCLUSION AND FUTURE WORK

We revisit geometric filters and predicates and study the possible applications to GIS algorithms and software. We propose

Count of "touch" results			
algorithm	predicate	-march=native -O2	-O2
ours	non robust	365	2,716
Boost.Geometry relate	non robust	8,759	8,759
ours	robust	2,714	2,714
CGAL bounded_side_2	robust	2,714	2,714

Table 5. Count of "touch" results between continental Russia polygon (Kelso and Patterson, 2010) and query points resulted from a linear interpolation of the polygon; ours: our winding count implementation.

a generic C++ meta-programming based framework to generate automated as well as manual filters. Then we perform an experimental analysis of various generated filters, we compare them with the current state-of-the-art on synthetic and real world datasets.

We reach the following conclusions based on evidences from the conducted experimental analysis. First, staged predicate with A, D filters is the best option for 2D orientation predicates. However, for incircle predicates the situation is more involved. The FPG error bound always seems to be less accurate than the Stage A error bound and also slower but there is some potential use when overflow is a concern. There is a speed-accuracy trade-off between dynamic interval filters and semi-static stage A filters and it is possible that a staged predicate can benefit from placing a dynamic interval filter between the semi-static stage A filter and the exact stage. For exact stages, stage D is always faster than Gmpzf, but the latter can handle inputs that cause overflow. Stage B and Stage C are in a similar role to the dynamic interval filter. For complicated predicates they may be useful between the semi-static and the exact stage, but we are not aware of any data set for which this can be observed.

As a future work, we would like to generalize our meta-programming framework to compute robust geometric constructions.

ACKNOWLEDGEMENTS

The main part of the work was mostly produced while T.B was supported by Google Summer of Code 2020 grant and V.F. was his mentor within the Boost C++ libraries organization.

REFERENCES

Attene, M., 2020. Indirect Predicates for Geometric Constructions. *Computer-Aided Design*, 126, 102856.

Brönnimann, H., Burnikel, C., Pion, S., 1998. Interval arithmetic yields efficient dynamic filters for computational geometry. *Proc. of the 14th Annual Symposium on Computational Geometry*, ACM, USA, 165–174.

Brönnimann, H., Fabri, A., Giezeman, G.-J., Hert, S., Hoffmann, M., Kettner, L., Pion, S., Schirra, S., 2021. 2D and 3D linear geometry kernel. *CGAL User and Reference Manual*, 5.2.1 edn, CGAL Editorial Board. `https://doc.cgal.org/5.2.1/Manual/packages.html#PkgKernel23`.

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

Devillers, O., Fronville, A., Mourrain, B., Teillaud, M., 2000. Algebraic methods and arithmetic filtering for exact predicates on circle arcs. *Proc. of the 16th Annual Symposium on Computational Geometry*, ACM, USA, 139–147.

Devillers, O., Pion, S., 2003. Efficient exact geometric predicates for delauny triangulations. R. E. Ladner (ed.), *Proceedings of the Fifth Workshop on Algorithm Engineering and Experiments, Baltimore, MD, USA, January 11, 2003*, SIAM, 37–44.

Egenhofer, M., Herring, J., 1990. A mathematical framework for the definition of topological relations. *Proceedings the Fourth International Symposium on Spatial Data Handing*, 803–813.

Gehrels, B., Lalande, B., Loskot, M., Wulkiewicz, A., Karavelas, M., Fisikopoulos, V., 2021. Boost C++ libraries: Geometry, version 1.76. `https://boost.org/libs/geometry`.

Granlund, T., the GMP development team, 2012. GNU MP: The GNU Multiple Precision Arithmetic Library. 5.0.5 edn. `http://gmplib.org/`.

Kelso, N. V., Patterson, T., 2010. Introducing Natural Earth Data - naturalearthdata.com. *Geographia Technica*, 5(82-89), 25.

Kettner, L., Mehlhorn, K., Pion, S., Schirra, S., Yap, C., 2004. Classroom examples of robustness problems in geometric computations. *Algorithms – ESA 2004*, Springer Berlin Heidelberg, 702–713. `https://doi.org/10.1007/978-3-540-30140-0_62`.

Li, Z., Zhu, C., Gold, C., 2005. *Digital Terrain Modeling: Principles and Methodology*. CRC Press. `https://doi.org/10.1201/9780203357132`.

Meyer, A., Pion, S., 2008. FPG: A code generator for fast and certified geometric predicates. *Real Numbers and Computers*, Santiago de Compostela, Spain, 47–60. `https://hal.inria.fr/inria-00344297`.

Nanevski, A., Blelloch, G., Harper, R., 2003. Automatic Generation of Staged Geometric Predicates. *Higher-Order and Symbolic Computation (formerly LISP and Symbolic Computation)*, 16(4), 379–400. `https://doi.org/10.1023/a:1025876920522`.

Ozaki, K., Bünger, F., Ogita, T., Oishi, S., Rump, S. M., 2016. Simple floating-point filters for the two-dimensional orientation problem. *BIT Numerical Mathematics*, 56(2), 729-749.

Shewchuk, J., 1996. Routines for Arbitrary Precision Floating-point Arithmetic and Fast Robust Geometric Predicates. `https://cs.cmu.edu/afs/cs/project/quake/public/code/predicates.c`.

Shewchuk, J. R., 1997. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry*, 18(3), 305–363. `https://doi.org/10.1007/pl00009321`.

Sunday, D., 2021. *Practical Geometry Algorithms: with C++ Code*. Amazon Digital Services LLC. ISBN: 9798749449730.

Špelič, D., Novak, F., Žalik, B., 2008. Delaunay Triangulation Benchmarks. *Journal of Electrical Engineering*, 59(1), 49–52. `http://iris.elf.stuba.sk/JEEEC/data/pdf/1_108-09.pdf`.