FPG: A code generator for fast and certified geometric predicates

Andreas Meyer, Sylvain Pion

FPG: A code generator for fast and certified geometric predicates

Andreas Meyer
INRIA Sophia-Antipolis
ameyer3000@googlemail.com

Sylvain Pion
INRIA Sophia-Antipolis
Sylvain.Pion@sophia.inria.fr

Abstract

We present a general purpose code analyzer and generator for filtered predicates, which are critical for geometric algorithms. While there already exist such code generators, our contribution is to generate "almost static filters", a type of filter which could not be generated previously. The generated and safe filtered predicates are almost as fast as their inexact floating point counterparts, in most cases.

1 Motivation

Floating point arithmetic is not exact and suffers from rounding and under/overflow effects. While some applications are immune, some are very sensitive [6] to these problems, especially algorithms that manipulate combinatorial data structures, such as many geometric algorithms. For geometric algorithms, the basic building blocks are geometric predicates like the 2D orientation test which tells if three points p, q, r in the plane form a left turn, right turn or if they are collinear (Figure 1). The vast majority of these predicates evaluate a homogeneous polynomial (such as the determinant) and compute the sign of this polynomial, discarding its actual value.

```
float det2x2( float a00, float a01,
              float a10, float a11  )
{ return a00*a11 - a10*a01; }

int orientationC2( float px, float py,
                   float qx, float qy,
                   float rx, float ry )
{ return sign( det2x2( qx-px, qy-py,
                       rx-px, ry-py ); }
```

Figure 1: A prominent geometric predicate: the Cartesian 2D orientation test.

Ultimately, we are interested in robust and efficient algorithms. One solution is to implement predicates such that they always return the correct result for their given input, called the Exact Geometric Computation Paradigm [10]. Of course, using only exact arithmetic is a brute force approach, so floating point filters have been proposed to "filter out" the easy cases, where the floating point hardware is sufficiently precise, and only the hard cases (called filter failures) are evaluated using exact arithmetic.

There are many different filtering schemes, ranging from fully dynamic to schemes where a strong hypothesis on the input values is used to create specialized and fast predicates. Most of them have in common, that writing such a filter manually is an error prone and tedious task, making it an ideal candidate for automating.

1.1 Previous work

A very general solution is to use interval arithmetic [2], which solves the problem directly at the number type level, minimizing per-predicate programming effort. Additionally, filtered predicates which have as input constructed geometric objects (like the midpoint of two points) are inherently easy to implement, if constructed objects are represented as intervals. Although interval arithmetic is faster than exact arithmetic, it is still several times slower than plain floating point arithmetic.

On the other hand, one can achieve better performance by preprocessing a given predicate and performing error analysis, or even generate code for the exact evaluation, as in Fortune and Van Wyk's LN package [5]. However, LN only allows integer values as input, and the user has to provide a bound on the bit-length, at predicate generation time. Later, Funke et al. presented a tool EXPCOMP [3], which allows floating point values as input and generates a two-phase filter: first exploit a global input bound (given at runtime) and if that fails, use a more precise semi-static filter that adapts to the actual input. Furthermore, EXPCOMP allows constructed values as predicate input. A very interesting tool from Nanevski et al. [8] generates adaptive precision predicates in the spirit of Jonathan Shewchuk [9], where predicates are cleverly divided into different stages of increasing precision. Intermediate results from less precise stages are accumulated and reused until the correct sign is known.

While all these tools are of interest and each one has unique features, they generate either static filters or semi-static filters. Static filters depend on global bounds of the input, which is restrictive. Semi-static filters basically evaluate the formula twice: once for the actual value and once for the error bound. Only a small amount of work can be precomputed.

An "almost static filter" combining the advantages of static and semi-static filters has been proposed by Pion and Devillers [4]. No bounds on the input are required, while the overall running time in an algorithm is still close to the one achieved with plain floating point arithmetic, at least for not-too-degenerate input configurations. Later, Pion and Melquiond [7] also automatically certified the generated error bounds, for some handpicked predicates.

1.2 Our Contribution

Our contribution is a tool that automatically generates almost static filters, which take into account the actual input to adapt/scale a precomputed error bound. The tool is applicable to a wide range of functions, i.e. those that compute signs of homogeneous polynomials. We demonstrate the utility in the context of CGAL [1], the Computational Geometry Algorithms Library.

Underflow and overflow are often regarded as esoteric, because on most platforms, it is easy to test a posteriori if an under/overflow has occurred. However, some platforms lack this ability (for example the Java Runtime Environment), hence another method [4] is needed to prevent under/overflow. Our tool implements such a method.

2 Almost Static Filters

Although the approach itself has been described already [4], we try to give more insight for how and why it works.

Roughly, the idea is to precompute a bound $\delta(1)$ on the absolute error. Consider an expression e in $\text{sign}(e)$, and assume that for each variable v occurring in e it holds that

$|v| \leq 1$. This precomputation simply amounts to forward error analysis of the expression e, starting with an absolute bound of 1 and the absolute error 0, propagating an ever-increasing error term along the abstract syntax tree of e (see Appendix-B).

Then, during predicate execution and given an actual input bound b, $\delta(1)$ is scaled to match the actual input, using a scaling function $\alpha : \star \rightarrow \mathrm{R}$: $\delta(b) = \alpha(b)\delta(1)$. Now, it becomes feasible to compare the computed value $\tilde{e}$ with $\delta(b)$: if $|\tilde{e}| > \delta(b)$, then $\mathrm{sign}(\tilde{e}) = \mathrm{sign}(e)$.

Algebraically, the expression e is a homogeneous polynomial of degree d. Given known input bounds at runtime: how can we reuse the precomputed $\delta(1)$?

For multilinear functions like determinants (which are in fact homogeneous polynomials) we know that $\det(\lambda_1 a_1, \lambda_2 a_2 \ldots \lambda_d a_d) = \lambda_1 \lambda_2 \ldots \lambda_d \det(a_1, a_2 \ldots a_d)$ where a_i is a vector of dimension d. If a_1 to a_d are assumed to be in $[-1, 1]^d$, the individual λ_i just correspond to the maximum absolute values of the vector's coordinate components, which are easily computed at runtime. Of course, it looks tempting to somehow reuse these λ_i. But, before going into detail how the scaling function α is defined in this case, let us discuss polynomials where we do not know that they are determinants and/or have the property "multilinear".

Consider a polynomial represented as $p = \Sigma_i c_i \Pi_j^d x_{i,j}$ where d is the degree, c_i are coefficients and $x_{i,j}$ denotes the (possibly equal) variables used throughout the polynomial. We can choose a k such that $p = \lambda_k \Sigma_i c_i \frac{x_{ik}}{\lambda_k} \left(\Pi_{j \neq k}^d x_{i,j} \right)$ where $\lambda_k = \max_i \{x_{ik}\}$. The choice is not completely arbitrary, at least in some cases as we will see later. Repeatedly pulling out the λ_k, we obtain a polynomial p_1 where the input variables are artificially bounded by 1. Now, $\delta(1)$ is applicable:

$$|p_1| = \left| \Sigma_i c_i \Pi_j^d \frac{x_{ij}}{\lambda_j} \right| \quad > \quad \delta(1) \Rightarrow \text{Sign of } p_1 \text{ is correct}$$

This is only a statement about the scaled polynomial. Now, we can easily obtain a statement about the original, unscaled polynomial, by multiplying with $\Pi_i^d \lambda_i$:

$$\begin{aligned} \left| \left(\Pi_i^d \lambda_i \right) \Sigma_i c_i \Pi_j^d \frac{x_{ij}}{\lambda_j} \right| &> \delta(1) \Pi_i^d \lambda_i \\ \left| \Sigma_i c_i \Pi_j^d x_{i,j} \right| &> \delta(1) \Pi_i^d \lambda_i \Rightarrow \text{Sign of } \left(\Sigma_i c_i \Pi_j^d x_{i,j} \right) \text{ is correct} \end{aligned}$$

To account for second order rounding errors done in the multiplication $\delta(1)\Pi_i^d \lambda_i$, $\delta(1)$ has to be multiplied with a constant $(1+\epsilon)^d$, ϵ being the machine epsilon and rounding the constant towards $+\infty$, of course. To sum up: we can slightly overestimate the error for the actual problem and replace its computation with a precomputed error for a bounded problem and a series of multiplications/`fabs`-operations.

Translation Filter As mentioned by Devillers and Pion [4], a dynamic but still absolute bound on the input might be too pessimistic. Consider a point-set with large coordinates, where in some algorithm, mostly predicate invocations between neighboring points would be required. In this case, it makes sense to translate points to the origin, before computing the bound. Of course, this does not change the order of magnitude in the worst case, but since it can be easily implemented and improves performance, we also support this in our tool.

The only difference is: instead of just setting λ to the maximum value of all input variables, we check if there are subtractions directly on input variables and mark those expressions as to-be-used for computing the bound. This effectively halves the number of `fabs` operations, in the best case. Similarly, the forward error analysis needs to be slightly adapted: the error

$$\begin{vmatrix} a & b & a^2+b^2 \\ c & d & c^2+d^2 \\ e & f & e^2+f^2 \end{vmatrix}$$

Columns=products												Rows=sums of columns	λ_i
a	a	c	c	e	e	a	a	c	c	e	e		$\lambda_1 = \max\{a, c, e\}$
d	d	f	f	b	b	f	f	b	b	d	d		$\lambda_2 = \max\{b, d, f\}$
e	f	a	b	c	d	c	d	e	f	a	b		$\lambda_3 = \max\{a, b, c, d, e, f\}$
e	f	a	b	c	d	c	d	e	f	a	b		$\lambda_3 = \max\{a, b, c, d, e, f\}$

Table 1: (a) 3×3 determinant with one column of squared lengths. (b) Polynomial representation. Columns denote products.

is propagated as before, but for those expressions that have been marked, we reset the bound to 1.

Note that we still have a homogeneous polynomial: in this case, the variables are represented by the result of a subtraction and a precomputed relative error.

Groups Consider some polynomial expression *p=abc-def+ghi*, where $a \ldots i$ are input variables. How can we automatically choose where to pull out the λ_i values, in each summand? Does the choice make a difference, in the first place? Ultimately, the goal is to obtain a maximal number of different minimal variable sets for λ_i. These sets need not be disjoint, but disjoint sets are preferred because it improves the precision of our filter. In the context of geometric predicates, a separation along dimensions of input coordinates seems natural: one λ_x for the x-coordinate, one for y and one for z. Then, for a 3×3 determinant computation, the scaling function α that is used to scale $\delta(1)$ is $\lambda_x\lambda_y\lambda_z$ (ignoring second order rounding errors).

But, the tool does not know about x and y coordinates. It only knows variables, which are not necessarily named p_x or q_z. A straightforward solution is to explicitly group variables, which is what we do in our tool. One might ask, if it is really important, that the λ_i partition variables according to the coordinate axis. And indeed, for multilinear functions like determinants, there are many ways along which the variables could be partitioned into groups.

Problems arise for more complicated cases: consider a 3×3 determinant, where each row consists of a 2-dimensional vector and its squared length as third component (Table 1 (a)). Table 1 (b) should be read like this: each column denotes one product of the polynomial. Each row contains one set of variables, that are used to compute the bound λ_i. Observe that the way to write down the individual sums and products occurring in each polynomial has an influence on the variable set for each λ_i: in the worst case, we only get one big $\lambda_1 = \max\{a, b, c, d, e, f\}$, which is just the uniform bound of all input variables we wanted to avoid.

It is not completely obvious, how to automatically arrange individual columns in the above table, to obtain good variable sets for λ_i. Therefore, we resort to group annotations for input variables. In the above example, a, c, e should be declared as one group, and b, d, f as another one. Based upon these declarations, our tool computes the sets, as above.

Higher Degree Numbers Geometric predicates are often dealing with distances. Sometimes, an approximate distance is computed and supplied as an argument to a predicate. When using an almost static filter, there are interactions with two concepts.

(1). Consider $ab+c$: usually, this would be recognized as a non-homogeneous polynomial. If, however, c is the result of multiplying two numbers which are homogeneous to a and b, then the result can still be regarded as a homogeneous polynomial. Whether the multi-

plication is performed inside or outside the predicate, the result should be (conceptually) the same. This means our tool needs to have a special notion of homogeneity: the user has to declare, if an input number is supposed to be the result of a multiplication, and he or she has to tell the degree of that number (see Appendix-A). Note that user mistakes are not dramatic and will only lead to more filter failures.

(2). Assume a naive computation of $\lambda = \max\{a, b, c\}$. This λ might be used in $\lambda^2\delta(1)$, which is wrong, because the already squared value c is squared again. Instead, the computation should be $\max\{\max\{a, b\}^2, c\}\delta(1)$. Therefore, our tool considers this special case when performing symbolic group computation.

Group computation Algebra on groups is defined using inductive rules on a recursive data type. There are three types of nodes: leaf, product, and alternatives. A leaf represents one group (as declared by the user) and the degree of its variables. Internally, groups are identified as integers, starting with 1. Additionally, leafs maintain a list of variables. At the beginning of symbolic group computation, all input variables are "abstracted" with a symbolic leaf node, containing only this variable. Products and alternatives maintain a list of nodes. As expected, a product represents a product of groups. An alternative contains a list of nodes, whose "concrete values" have to be compared at runtime to compute one λ_i (required for higher degree numbers).

Enumerating all individual rules would go beyond the scope of this paper. Instead, we only describe them informally. The two base cases are:

```
add (Leaf index  degree vars1), (Leaf index  degree vars2) = Leaf index degree merge(vars1, vars2)
add (Leaf index1 degree vars1), (Leaf index2 degree vars2) = Leaf 0     degree merge(vars1, vars2)
```

which means that adding two leaves merges the variable set, and if indices differ, we set the result group index to 0, representing "don't know". Beyond that, addition and multiplication are defined recursively in a rather straightforward way: adding two products results in a product with individual nodes being added, where nodes have similar structure. For remaining nodes, an alternative is appended to the product's result list. Adding two group nodes of different degree is not defined.

For example, $ab + cd$ would be represented as `Prod(Leaf 1 1 {a,c}, Leaf 2 1 {b,d}`, given that all variables have degree 1, a and c are in group 1 and b and d are in group 2.

Another example: c has degree 2, is in group 3, a, b have degree 1 and group indices 1 and 2, respectively. Then $ab + c$ would be represented as `Alt(Prod(Leaf 1 1 {a}, Leaf 2 1 {b}), Leaf 3 2 {c})`,

Finally, the group computation result is used to generate the scaling function α in our tool, exploiting the recursive definition of the data type.

Under/Overflow Protection There are two options to prevent underflow and overflow: (1) After computation, check floating point flags to see if such an event did really occur. (2) Compare λ_i with some global bounds, to prevent underflow and overflow.

Each option has its advantages: the first is easy to implement, precise and fast (given that there is no function call overhead). However, it may not be supported on all platforms. On the other hand, the second option requires additional runtime computations: $\min_i\{\lambda_i\}$ and $\max_i\{\lambda_i\}$, whose results are then compared with two precomputed constant numbers $\lambda_{\min}$ and $\lambda_{\max}$. Because it is not clear which option is better, we support both.

How to determine the global bounds for under/overflow? First, we need to prevent the actual computation of e from overflowing. Consider a function $\text{over}(e, \lambda) \to \text{bool}$ that symbolically evaluates e using floating point arithmetic and all input variables set to λ. During evaluation, we maintain bounds for each value, conservatively approximating to $+\infty$. If during such an evaluation, an overflow event occurred (i.e., $\text{over}(e, \lambda) = \text{true}$), it might also happen during the evaluation of the actual function, using round-to-nearest mode. On the other hand, if no overflow occurred during the symbolic evaluation, an overflow event is not possible for a concrete evaluation, where values are bounded by λ. We claim that the function $\text{over}(e, \lambda)$ is monotonous, that is, there is a certain upper bound $\lambda_{\max}$, where the function result changes from false to true. This upper bound can be safely approximated by a simple dichotomy over the range of floating point values. An overflow in the computation of $\delta(1)\Pi_i^d \lambda_i$ is safe and directly leads to a filter failure.

An underflow in $\delta(1)\Pi_i^d \lambda_i$ would decrease the error-bound and in turn, would lead to false positives (i.e., filter failures would be reported as filter "success"). To prevent this, consider the following inequalities:

$$\begin{aligned} \text{mindouble} &\leq \delta(1)\lambda_{\min}^d \\ \sqrt[d]{\frac{\text{mindouble}}{\delta(1)}} &\leq \lambda_{\min} \end{aligned}$$

If all $\lambda_i > \lambda_{\min}$ no underflow can occur here.

An underflow in e could either increase or decrease the value of e, but the effect is very small and negligible. Predicates with input of different degree need special treatment by the tool.

3 Implementation

Input Language Our input language is a minimal subset of the C language, with support for basic arithmetic (`+`,`-`,`*`), assignments, function calls, conditional statements/expressions and Boolean operators (see Figure 2). Arrays, pointers or structs would only complicate the interface and are not supported. Possible data types are `int` and `float`, where float arithmetic automatically triggers error analysis. Two special functions `sign` and `abs` are recognized. Occurrences of `sign` expressions are automatically rewritten, if they involve derived floating point values. Groups of variables (like x-coordinates, y-coordinates) have to be declared explicitly, using the `group` declaration. They should be chosen such that the polynomials occurring in the predicate can be split according to these groups.

External functions can be defined (for example, `CGAL_assertion` or `CGAL_max`) which may accept integer or non-derived floating point values and are otherwise ignored during analysis.

Generated functions are declared inline, to avoid function calling overhead.

Implementation Details Internally, the input is represented as an abstract syntax tree. This tree can be traversed by visitors. There are visitors to transform the tree or add statements, visitors to derive and propagate properties like "contains floating point comparison on derived values", or abstract interpretation visitors that evaluate functions, using abstract values. Important abstract values are the bound estimation value, or the group computation value. In total, there are 9000 lines of code.

```
float det2x2(
    float a00, float a01 ,
    float a10, float a11 )
{
    return a00*a11 - a10*a01;
}

int orientationC2(
    float px, float py ,
    float qx, float qy ,
    float rx, float ry )
group px qx rx;
group py qy ry;
{
    return sign(
        det2x2( qx-px, qy-py ,
                rx-px, ry-py )
              );
}
```

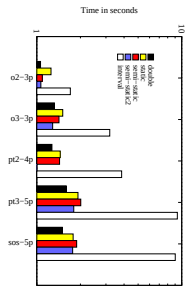
```
inline int orientationC2( double px, double py, double qx,
                          double qy, double rx, double ry )
{   double a00 = qx - px, a01 = qy - py;
    double a10 = rx - px, a11 = ry - py;
    int int_tmp_result;
    double d = a00 * a11 - a10 * a01;
    double max1 = std::fabs(a01);
    double max2 = std::fabs(a10);
    if( max1 < std::fabs(a11) ) max1 = std::fabs(a11);
    if( max2 < std::fabs(a00) ) max2 = std::fabs(a00);
    double lower_bound = max1, upper_bound = max1;
    if( max2 < lower_bound )
        lower_bound = max2;
    else if( max2 > upper_bound )
        upper_bound = max2;
    if( lower_bound < 5.0036808196096469082e-147 ) {
        if( lower_bound == 0.0 )
            int_tmp_result = 0;
        else
            return FPG_UNCERTAIN_VALUE;
    } else {
        if( upper_bound > 1.6759759912428240792e+153 )
            return -999;
        double eps = 8.8872057372592797681e-16 * max2 * max1;
        if( d > eps )
            int_tmp_result =  1;
        else if( d < -eps )
            int_tmp_result = -1;
        else
            return FPG_UNCERTAIN_VALUE;
    }
    return int_tmp_result;
}
```

Figure 2: Example input and FPG result for `orientationC2`. p, q, r are exact input points. Note that `det2x2` has been directly inlined.

Main loop First, all function calls are substituted, until each function definition only contains function calls, whose function definitions do not contain floating point comparisons on derived values. Then, for each function definition that has floating point comparisons on derived values, the following is done: (1) determine lower and upper bound (2) find values used to compute λ_i (either plain input values, or the result of subtracting two input values) (3) for all λ_i expressions: eliminate duplicate binary expressions (4) compute floating point errors (5) compute groups (6) replace sign expressions, using gathered knowledge.

Replacing sign(e) expressions triggers most of the code generation: the runtime λ_i have to be computed, they have to be compared against the precomputed $\lambda_{\min}$ and $\lambda_{\max}$, and finally the scaled $\delta\Pi_i^d\lambda_i$ is compared with the actual value $\tilde{e}$.

Example In Figure 2, one can see the code FPG generates, for the `orientationC2` function. Most "features" are present: input bound computation, comparison with precomputed lower and upper bound and comparison with the scaled error. The variables `a00` are an artifact of substituting a function call. Temporary `int` variables are required to support `sign` expression multiplications. This function is almost the same as the one hand-coded in [7], modulo assignments to useless temporary variables, which can easily be optimized away by a compiler. Another example is listed in Appendix-A.



	Full Name
`o2-3p`	2D Orientation
`o3-4p`	3D Orientation
`pt2-4p`	2D Power test
`pt3-5p`	3D Power test
`sbs-5p`	Side of bounded sphere
`-5p`	.. on 5 points

Figure 3: Time to evaluate different important predicates 10e6 times in a loop, ignoring the exact evaluation stage. Average of 10 runs. Logarithmic scale.

4 Results

Integration into CGAL CGAL has different layers for geometry and arithmetic. This layered design made it particularly easy, to almost automatically generate statically filtered predicates from existing CGAL source code. Additionally, the output of FPG needs to be integrated in CGAL - a task which we also automated. The order of predicate evaluation has 3 stages: first, the generated almost static filters are used. If they fail, interval arithmetic is used, as previously done in CGAL. Only if this should fail, exact arithmetic is used.

Going a step further, we also automatically generated a test-suite that cross-checks the runtime behavior of each generated predicate with the already existing interval arithmetic predicates (which is assumed to be correct), using degenerate random input.

Benchmarks We compared the runtime of several filtering methods (see Figure 3): floating point arithmetic (`double`), almost static filters (`static`) and interval arithmetic (`interval`) using CGAL's interval numbertype (setting rounding mode once, before predicate evaluation).

Semi-static filters (`semistatic`) are also supported by FPG, as a by-product, although the translation trick is not yet supported. Thus, FPG's semi-static filters are less precise and need more operations per predicate. To work around this limitation, four important semi-static predicates have been hand-optimized (`semistatic2`).

We only measured the speed of the filter stage itself: if it fails, we simply ignore it, and no exact evaluation is performed. As input, we used an array of random numbers. Before each of the 10 million predicate invocations, input data was shuffled to prevent the compiler from optimizing away the function call or doing something too clever. A-posteriori under/overflow checks were implemented as inline functions, to avoid function call overhead. All benchmarks were performed on an Intel Core 2 Duo with 2.33 GHz, using `g++-4.3.0 -O3 -DNDEBUG -march=pentium-m -msse2 -mfpmath=sse`.

Interval arithmetic performs quite poorly in this synthetic benchmark, as it needs many conditional branches and up to 4 times the number of floating point operations a plain floating point evaluation would require.

	2D orientation				3D orientation				Side of oriented sphere			
	$+$	$*$	$\|x\|$	$<$	$+$	$*$	$\|x\|$	$<$	$+$	$*$	$\|x\|$	$<$
semistatic	10(5)	5(3)	7	2	28(14)	18(9)	13	2	74(37)	80(40)	16	2
semistatic2	6(1)	3(1)	3	2	19(5)	13(4)	10	2	60(23)	57(17)	16	2
static	5(0)	4(2)	4	8	14(0)	12(3)	15	14	37(0)	45(5)	12	16

Table 2: Total number of operations (bound computation overhead in braces).

	Levels of "Degenerateness"					Different Scales				
size	$\leftarrow 1 \rightarrow$					1e-70	1e-50	1	1e+50	1e+70
ϵ	1e-04	1e-08	1e-12	1e-16	1e-20	$\leftarrow$ size$\times$1e-10 $\rightarrow$				
interval	3.17	3.18	3.22	3.83	4.70	68.7	3.12	3.13	3.14	24.9
semistatic	0.69	0.71	0.82	1.60	2.40	68.9	0.72	0.71	0.74	23.0
semistatic2	0.61	0.61	0.61	1.36	2.27	68.8	0.62	0.61	0.62	23.3
static	0.64	0.64	0.63	1.39	2.24	70.7	0.66	0.63	0.66	23.3
static2	0.60	0.60	0.60	1.37	2.27	70.1	0.60	0.60	0.59	23.1
double	0.57	0.57	0.57	—	—	—	0.57	0.57	0.57	—

Table 3: Average time to construct a 3D regular triangulation, in seconds. Input: 200000 cube grid points in $[-\text{size}, \text{size}]$ perturbed by ϵ. The `double` implementation often crashed.

For simple predicates like 2D orientation, semi-static predicates are clearly faster. Almost-static predicates are competetive only for larger predicates, involving more arithmetic where the initial overhead of computing the λ_i (which needs comparisons) is compensated by a greatly reduced number of additions and multiplications (Table 2). On the other hand, small hand-optimized `semistatic2`-predicates have very little overhead (see Appendix-C). In general, the overhead of almost-static predicates depends on the number of input variables and the degree of the polynomial, whereas semi-static predicates depend on the overall amount of arithmetic.

Although the total number of operations for larger static predicates is much less than for their semi-static counterparts, the synthetic benchmarks suggest that static predicates are still slower. Semi-static predicates seem to better profit from SSE2 instructions and/or multiple functional units, processing more than one operation at each processor cycle.

Ultimately, we are interested in real-world applications. Therefore, we used CGAL's 3D regular triangulation and compared different filtering approaches (see Table 3). Most time is spent in the 3D orientation test, and also the 3D power test is used. For filters, speed and precision are equally important. To demonstrate their resilience to degenerate input, we used a range of increasingly degenerate input configurations. Different scales show that our predicates properly detect under/overflow situations. As expected, preventing under/overflow (which needs more comparisons) is slightly slower than just testing the respective floating point flag (as done in `static2`), contrary to the synthetic benchmarks, where `static` and `static2` perform almost similarly (not shown here). Moreover, contrary to the synthetic benchmarks, `static2` is marginally faster than `semistatic2`, when used in CGAL's 3D regular triangulation. This is also true for the 3D delaunay triangulation (Table 4), where 3D orientation tests and side-of-sphere tests are called

`interval`	1.13
`semistatic2`	0.26
`static2`	0.24
`double`	0.22

Table 4: Time to compute a 3D delaunay triangulation of 20000 cube grid points

equally often.

5 Future Work

The most important improvement would be support for constructions. It is not clear how to do it, but maybe one can inject another $\delta(1)$ into the called function, to account for an increased error. The first translation/subtraction step does not work in such a scenario.

Automatically deducing the variable groups would probably be a challenging optimization problem, reducing manual annotation to a minimum.

It would be interesting to see, if our almost static filter can be combined with an adaptive precision exact evaluation phase, for example the one generated by Nanevski's tool [8].

Pion and Melquiond [7] used a tool to formally prove the value of $\delta(1)$. The input for this tool was written manually, but due to the structure of this input (that mostly resembles the input predicate) it is natural to assume that this can be automated, too. Even more, it would be very interesting to also verify if the result of our group computation is valid, maybe using FPG's intermediate steps during group computation as proof hints for a theorem prover.

Supporting the square root operation would be relatively easy to implement.

References

[1] CGAL, *Computational Geometry Algorithms Library*. URL `http://www.cgal.org/`.

[2] H. Brönnimann, C. Burnikel and S. Pion. *Interval arithmetic yields efficient dynamic filters for computational geometry*. Discrete Applied Mathematics, volume 109:25–47 (2001).

[3] Christoph Burnikel, Stefan Funke and Michael Seel. *Exact geometric predicates using cascaded computation*. In Symposium on Computational Geometry, 175–183 (1998).

[4] Olivier Devillers and Sylvain Pion. *Efficient exact geometric predicates for Delaunay triangulations*. In ALENEX, 37–44 (2003).

[5] Steven Fortune and Christopher J. Van Wyk. *Static analysis yields efficient exact integer arithmetic for computational geometry*. ACM Trans. Graph., volume 15(3):223–248 (1996). ISSN 0730-0301.

[6] Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra and Chee-Keng Yap. *Classroom examples of robustness problems in geometric computations*. In ESA, 702–713 (2004).

[7] Guillaume Melquiond and Sylvain Pion. *Formal certification of arithmetic filters for geometric predicates*. In Proc. 17th IMACS World Congress on Scientific , Applied Mathematics and Simulation (2005).

[8] Aleksandar Nanevski, Guy E. Blelloch and Robert Harper. *Automatic generation of staged geometric predicates*. In International Conference on Functional Programming, 217–228 (2001).

[9] Jonathan Richard Shewchuk. *Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates*. Discrete & Computational Geometry, volume 18(3):305–363 (1997).

[10] C. Yap and T. Dubé. *The exact computation paradigm*. In Computing in Euclidian Geometry. World Scientific Press (1994).

A Another Input/Output Example

To also demonstrate support for "higher degree numbers", we append input and output code for the 3D power test predicate. Basically, 3D points are augmented with a weight field. The predicate itself is similar to the more common "side of oriented sphere" predicate.

Weights are squared distances, and therefore of degree 2 which has to be declared using `group[degree=2]`.

```
int
power_testC3( double px, double py, double pz, double pwt,
              double qx, double qy, double qz, double qwt,
              double rx, double ry, double rz, double rwt,
              double sx, double sy, double sz, double swt,
              double tx, double ty, double tz, double twt )
group px qx rx sx tx;
group py qy ry sy ty;
group pz qz rz sz tz;
group[degree=2] pwt qwt rwt swt twt;
{
    double dpx = px - tx;
    double dpy = py - ty;
    double dpz = pz - tz;
    double dpt = square(dpx) + square(dpy) + square(dpz) + (twt- pwt);
    double dqx = qx - tx;
    double dqy = qy - ty;
    double dqz = qz - tz;
    double dqt = square(dqx) + square(dqy) + square(dqz) + (twt - qwt);
    double drx = rx - tx;
    double dry = ry - ty;
    double drz = rz - tz;
    double drt = square(drx) + square(dry) + square(drz) + (twt- rwt);
    double dsx = sx - tx;
    double dsy = sy - ty;
    double dsz = sz - tz;
    double dst = square(dsx) + square(dsy) +
                 square(dsz) + (twt -swt);

    return - sign(det4x4_by_formula(dpx, dpy, dpz, dpt,
                                    dqx, dqy, dqz, dqt,
                                    drx, dry, drz, drt,
                                    dsx, dsy, dsz, dst));
}
```

Our tool successfully propagates the degree 2 attribute to `max5`, and therefore compares it to the square of `max1`, which is related to the sum of squares `square(dpx)+square(dpy)...` in line 10.

```
inline int power_testC3( double px, double py, double pz, double pwt,
                         double qx, double qy, double qz, double qwt,
                         double rx, double ry, double rz, double rwt,
                         double sx, double sy, double sz, double swt,
                         double tx, double ty, double tz, double twt )
{   double dpx = px - tx;
    double dpy = py - ty;
    double dpz = pz - tz;
    double twt_pwt = twt - pwt;
    double dpt = square( dpx ) + square( dpy ) + square( dpz ) + twt_pwt;
    double dqx = qx - tx;
    double dqy = qy - ty;
    double dqz = qz - tz;
    double twt_qwt = twt - qwt;
    double dqt = square( dqx ) + square( dqy ) + square( dqz ) + twt_qwt;
    double drx = rx - tx;
    double dry = ry - ty;
    double drz = rz - tz;
    double twt_rwt = twt - rwt;
    double drt = square( drx ) + square( dry ) + square( drz ) + twt_rwt;
    double dsx = sx - tx;
    double dsy = sy - ty;
    double dsz = sz - tz;
    double twt_swt = twt - swt;
    double dst = square( dsx ) + square( dsy ) + square( dsz ) + twt_swt;
    int int_tmp_result;
    double max2 = std::fabs(dpx);
    if( max2 < std::fabs(dqx) ) max2 = std::fabs(dqx);
    if( max2 < std::fabs(drx) ) max2 = std::fabs(drx);
    if( max2 < std::fabs(dsx) ) max2 = std::fabs(dsx);
    double max3 = std::fabs(dpy);
    if( max3 < std::fabs(dqy) ) max3 = std::fabs(dqy);
    if( max3 < std::fabs(dry) ) max3 = std::fabs(dry);
    if( max3 < std::fabs(dsy) ) max3 = std::fabs(dsy);
    double max4 = std::fabs(dpz);
    if( max4 < std::fabs(dqz) ) max4 = std::fabs(dqz);
    if( max4 < std::fabs(drz) ) max4 = std::fabs(drz);
    if( max4 < std::fabs(dsz) ) max4 = std::fabs(dsz);
    double max1 = max2;
    if( max1 < max3 ) max1 = max3;
    if( max1 < max4 ) max1 = max4;
    double max5 = std::fabs(twt_qwt);
    if( max5 < std::fabs(twt_pwt) ) max5 = std::fabs(twt_pwt);
    if( max5 < std::fabs(twt_rwt) ) max5 = std::fabs(twt_rwt);
    if( max5 < std::fabs(twt_swt) ) max5 = std::fabs(twt_swt);
    double lower_bound_1 = max4,  upper_bound_1 = max4;
    if( max3 < lower_bound_1 )
        lower_bound_1 = max3;
    if( max1 < lower_bound_1 )
        lower_bound_1 = max1;
    else if( max1 > upper_bound_1 )
        upper_bound_1 = max1;
    if( max2 < lower_bound_1 )
        lower_bound_1 = max2;
    if( lower_bound_1 < 1.0589368463633375059 6e-59 || max5 < 1.12134724458593082308e-118
     || upper_bound_1 > 3.213876088517980551 08e+60 || max5 > 1.0328999512347634358 6e+121 )
        return FPG_UNCERTAIN_VALUE;
    double eps = 1.6710680309599047114 7e-13 * max2 * max3 * max4 * std::max( max1 * max1, max5 );
    double det = det4x4_by_formula( dpx, dpy, dpz, dpt,
                                    dqx, dqy, dqz, dqt,
                                    drx, dry, drz, drt,
                                    dsx, dsy, dsz, dst );
    if( det > eps )       int_tmp_result = 1;
    else if( det < -eps ) int_tmp_result = -1;
    else                  return FPG_UNCERTAIN_VALUE;
    return -int_tmp_result;
}
```

B Static Filter Error

For completeness, we present the two main functions that are used to compute error bounds in our implementation. Static filter error is abbreviated with `Sfe`. Our function definition of `ulp` also accounts for double rounding errors that occur in the extended precision implementation of Intel's floating point unit used in x86 processors.

Error propagation starts with an error of 0 and an absolute bound of 1, unless there was a translation on fresh input values, where the error is set to `ulp(1)/2`.

```
double ulp () {
    FPU_round_to_plus_infty();
    return ulp(1);
}

double ulp (double d) {
    // You are supposed to call this function with rounding towards
    // +infinity, and on a positive number.
    d = CGAL_IA_FORCE_TO_DOUBLE(d); // stop constant propagation.
    CGAL_assertion(d>=0);
    double u;
    if (d == 1) // I need to special case to prevent infinite recursion.
        u = (d + CGAL_IA_MIN_DOUBLE) - d;
    else {
        // We need to use the d*ulp formula, in order for the formal proof
        // of homogeneisation to work.
        // u = (d + CGAL_IA_MIN_DOUBLE) - d;
        u = d * ulp();
    }

    // Then add extra bonus, because of Intel's extended precision feature.
    // (ulp can be 2^-53 + 2^-64)
    return u + u / (1<<11);
}

Sfe operator+ (Sfe f1, Sfe f2) {
    FPU_round_to_plus_infty();
    double bound = f1.bound + f2.bound;
    double u = ulp(bound) / 2;
    bound += u;
    double error = u + f1.error + f2.error;
    return Sfe(bound, error);
}

Sfe operator* (Sfe f1, Sfe f2) {
    FPU_round_to_plus_infty();
    double bound = f1.bound * f2.bound;
    double u = ulp(b) / 2;
    bound += u;
    double error = u + f1.error * f2.error + f1.error * f2.bound + f1.bound * f2.error;
    return Sfe(bound, error);
}
```

C Hand-optimized Semi-Static Filters

```
inline int orientationC2( double px, double py, double qx, double qy, double rx, double ry) {
    double qpx = qx-px;
    double rpy = ry-py;
    double rpx = rx-px;
    double qpy = qy-py;
    double detleft = qpx*rpy;
    double detright = rpx*qpy;
    double det = detleft - detright;
    if( std::fabs(det) > (std::fabs(detleft) + std::fabs(detright)) * 8.88178419700125232339e-16
        && fetestexcept( FE_UNDERFLOW | FE_OVERFLOW ) == 0 )
    {
        return det < 0.0 ? -1 : 1;
    }
    else
    {
        ::CGAL::feclearexcept( FE_DIVBYZERO | FE_UNDERFLOW | FE_OVERFLOW | FE_INVALID );
        return FPG_UNCERTAIN_VALUE;
    }
}

inline int orientationC3( double px, double py, double pz,
                          double qx, double qy, double qz,
                          double rx, double ry, double rz,
                          double sx, double sy, double sz )
{
    double qx_px = qx-px;
    double qy_py = qy-py;
    double rx_px = rx-px;
    double ry_py = ry-py;
    double tmp_a = qx_px * ry_py;
    double tmp_b = qy_py * rx_px;
    double m01_bound = std::fabs(tmp_a) + std::fabs(tmp_b);
    double m01 = tmp_a - tmp_b;
    double rz_pz = rz-pz;
    double qz_pz = qz-pz;
    tmp_a = qx_px*rz_pz;
    tmp_b = qz_pz*rx_px;
    double m02_bound = std::fabs(tmp_a) + std::fabs(tmp_b);
    double m02 = tmp_a - tmp_b;
    tmp_a = qy_py*rz_pz;
    tmp_b = qz_pz*ry_py;
    double m12_bound = std::fabs(tmp_a) + std::fabs(tmp_b);
    double m12 = tmp_a - tmp_b;
    double sx_px = sx-px;
    double sy_py = sy-py;
    double sz_pz = sz-pz;
    double m012_bound = m01_bound*std::fabs(sz_pz)
                      + m02_bound*std::fabs(sy_py)
                      + m12_bound*std::fabs(sx_px);
    double m012 = m01*sz_pz - m02*sy_py + m12*sx_px;
    if( std::fabs(m012) > m012_bound * 1.77635683940025046468e-15 &&
        fetestexcept( FE_UNDERFLOW | FE_OVERFLOW ) == 0 )
        return (m012 < 0.0) ? -1 : 1;
    else {
        feclearexcept( FE_UNDERFLOW | FE_OVERFLOW );
        return FPG_UNCERTAIN_VALUE;
    }
}
```