

Robustness issues in CGAL : arithmetics and the kernel

Sylvain Pion

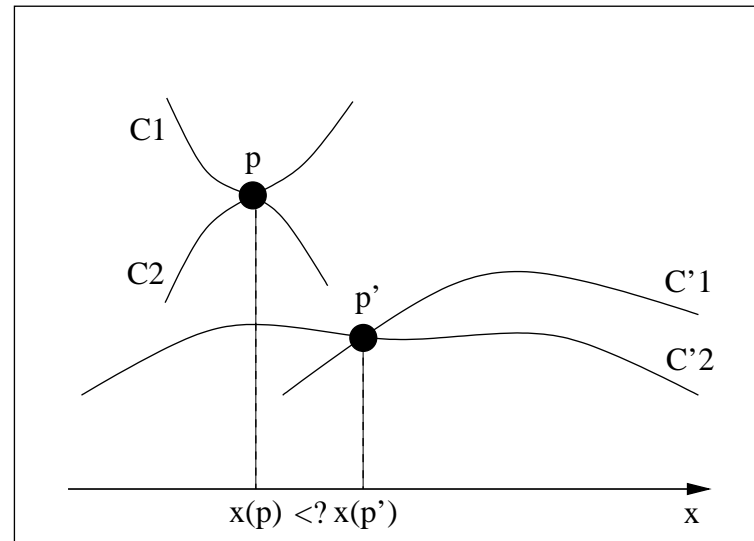
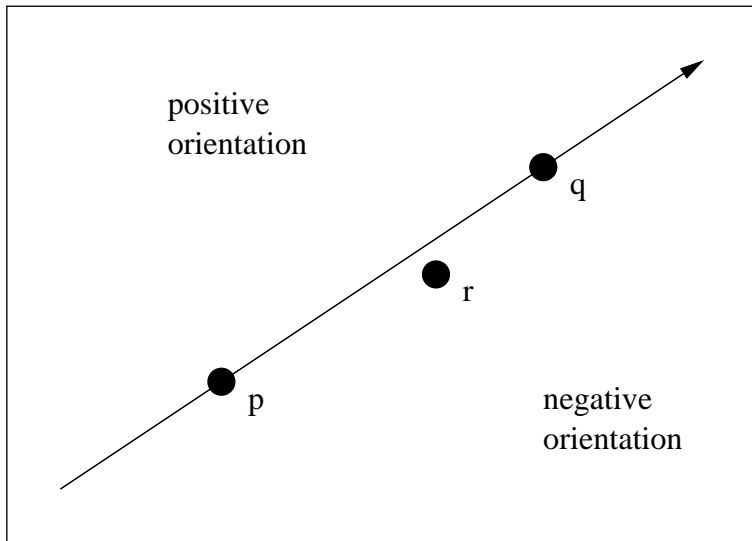
INRIA Sophia Antipolis

Plan

- Links between geometry and arithmetics
- Floating point arithmetic
- Exact arithmetic
- Arithmetic filters
- CGAL implementation

Introduction

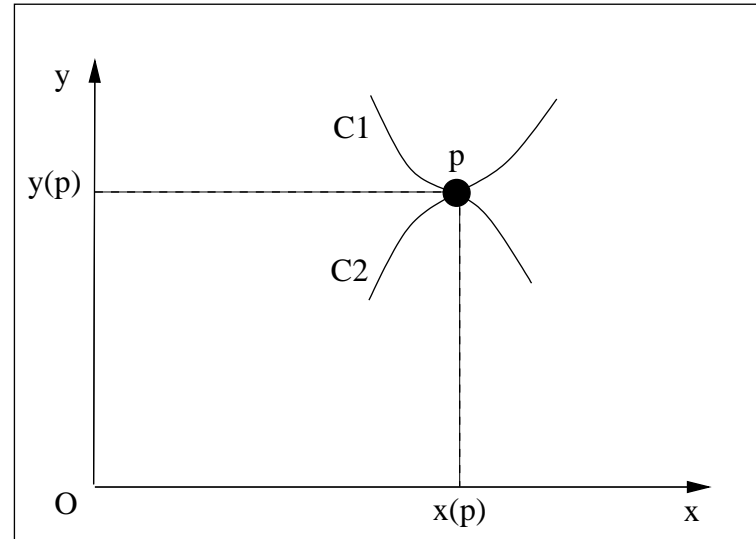
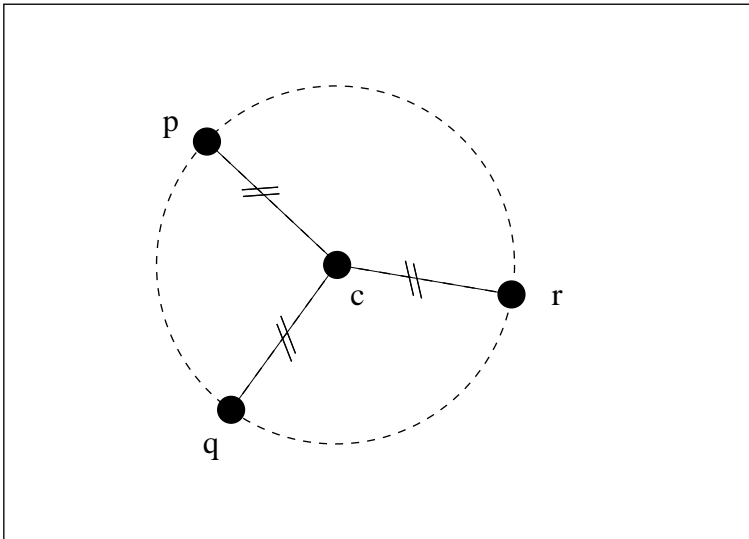
Examples of geometric predicates



$$\text{orientation}(p, q, r) = \text{sign}((x(p) - x(r)) \times (y(q) - y(r)) - (x(q) - x(r)) \times (y(p) - y(r)))$$

Predicate of **degree 2**.

Examples of geometric constructions



From geometry to arithmetic

Geometric algorithm

⇒ Geometric operations (predicates and constructions)

⇒ Algebraic operations over coordinates/coefficients

⇒ Arithmetic operations ($+$, $-$, \times , \div , $\sqrt{\dots}$)

Arithmetic \Rightarrow Geometry

Cost of arithmetic \Rightarrow Time complexity of geometric algorithms

Approximate arithmetic \Rightarrow robustness problems of geometric algorithms

The Real-RAM model

Real computer model with random access (RAM = Random access machine).

Theoretical model specifying the behavior of real arithmetic on computers.

- All arithmetic **operations** over reals cost **$O(1)$ time** (and are exact).
- All real variables take **$O(1)$ memory space**.

Complexity analyses of geometric algorithms are traditionally performed within this model.

Relationship with the reality of computers ?

Two approaches :

- Floating point arithmetic, **approximate**.
- Exact arithmetic, **slower**.

For geometry : which approach is the best in practice ?

What is the precise cost of the exact approach ?

Floating point arithmetic

IEEE 754 Standard

Standardization of basic FP operations on computers (1985).

Machine representation of $(-1)^s \times 1.m \times 2^e$ (for double precision, 64 bits):

s	exponent	mantissa
1	11	52

- **5 operations** : $+$, $-$, \times , \div , $\sqrt{\quad}$
- **4 rounding modes** : to nearest (representable number), towards 0, towards $+\infty$, towards $-\infty$.
- **Special values** : $+\infty$, $-\infty$, denormals, NaNs.
- Relatively well supported by the industry (languages, compilers, processors).

Ref : <http://stevehollasch.com/cgindex/coding/ieeefloat.html>

Rounding errors

Definition : x being a positive FP value, and y the smallest FP value greater than x , we define $\text{ulp}(x) = y - x$ (Unit in the Last Place).

Remark 1 : $\text{ulp}(x)$ is a power of 2 (or ∞).

Remark 2 : In normal cases : $\text{ulp}(x) \simeq x2^{-53}$

Property : For all operations $+$, $-$, \times , \div , $\sqrt{\quad}$, the difference between the computed value r and the exact value, the **rounding error**, is smaller than :

$\text{ulp}(r)/2$ for the rounding to nearest mode, and $\text{ulp}(r)$ otherwise.

Attention : This is only true for operations taken one at a time.

Some properties of FP arithmetic

There is **no underflow** for $+$, $-$:

$$a - b = 0 \iff a = b$$

Detection of NaNs:

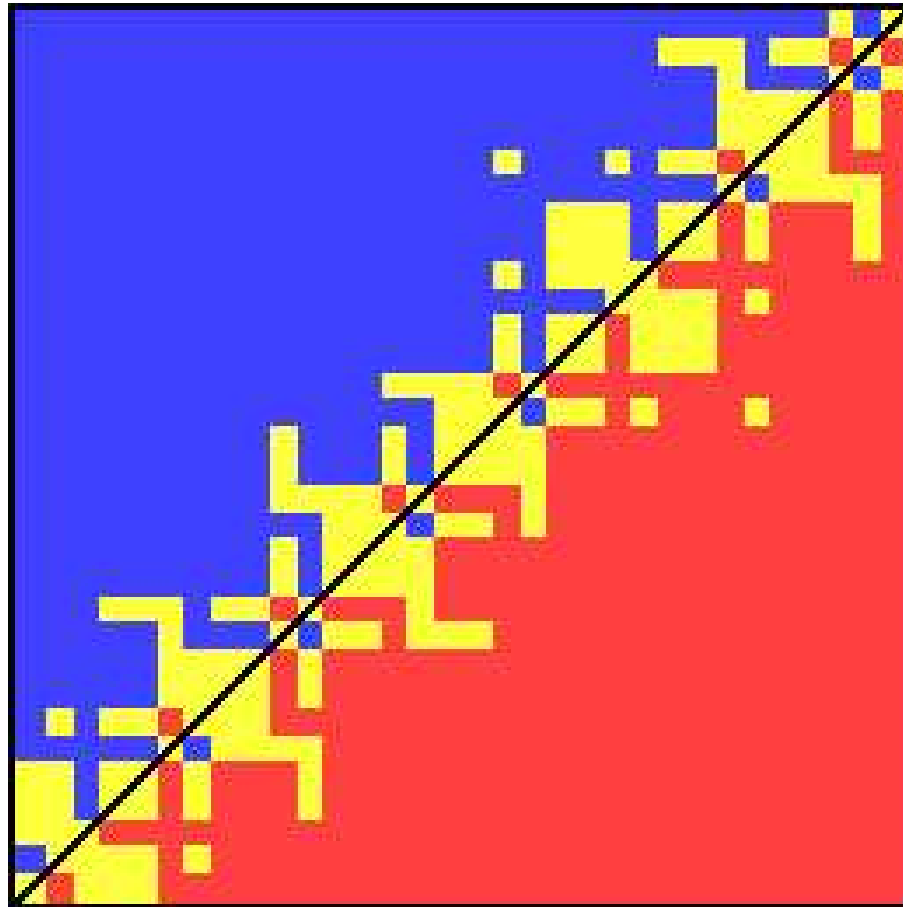
$$a = a \iff a \text{ is not a NaN}$$

Monotonicity for a given rounding mode:

$$a + b \leq c + d \text{ computed} \iff a + b \leq c + d \text{ exact}$$

(idem for the other operations)

Geometry of the approximate orientation predicate



[Kettner-Mehlhorn-Schirra-P-Yap 04]

Multiple precision computation

Multiple precision

Exact computing over integers (\mathbb{Z}) :

- $O(n = \log N)$ memory
- $+, -$: $O(n)$ time.
- \times, \div :

$O(n^2)$	if n small
$O(n^{\approx 1.5})$	if n average (Karatsuba)
$O(n \log n \log \log n)$	if n large (Schönhage Strassen)

Exact evaluation of polynomials over integral inputs of size $O(n)$: $\geq O(nd)$

Libraries : GMP, LEDA, CGAL, BigNum...

Karatsuba multiplication

We cut the operands x and y in two parts of equal size (most and least significant bits) :

MSBs	LSBs
x_1	x_0

Let b the power of 2 such that $x = x_1b + x_0$ and $y = y_1b + y_0$. We see that :

$$xy = (b^2 + b)x_1y_1 - b(x_1 - x_0)(y_1 - y_0) + (b + 1)x_0y_0$$

So, we use 3 multiplications of numbers of size $n/2$ (instead of 4).

Asymptotic complexity : $O(n^{\log(3)/\log(2)} = 1.585)$

To know more : <http://www.swox.com/gmp/manual/Algorithms.html>

Rational numbers

Just a pair of exact integers : numerator / denominator.

Attention : even the addition doubles the number of bits !

Normalization can be used (not free...) to reduce the size :

- Either we are lucky (small probability).
- Either we missed an algebraic simplification.
- Other cases ?

Otherwise : **exponential** growth with the depth of operations.

Multiple precision floating point numbers

$m2^e$, where m and e are multiple precision integers.

It's possible to add a precision p to x such that :

$$m2^e - 2^p \leq x \leq m2^e + 2^p$$

p can be specified to each operation, or globally.

p can be propagated.

Libraries : MPFR, CGAL::MP_Float.

Error propagation

Let (x, p_x) be a multiprecision FP number and an associated precision corresponding to a real X . Similarly for (y, p_y) .

Then we can get an approximation of $X + Y$ by $(x + y, p_{x+y})$, where:

$$|(X - x) + (Y - y)| \leq |X - x| + |Y - y|$$

$$|(X - x) + (Y - y)| \leq 2^{p_x} + 2^{p_y}$$

$$|(X + Y) - (x + y)| \leq 2^{p_{x+y}}$$

$$\implies p_{x+y} = 1 + \max(p_x, p_y)$$

This is true if $x + y$ is not rounded. Otherwise, it has to be taken into account.

Other arithmetic techniques in brief

- Modular arithmetic
- Separation bounds

The other extreme : filters

Optimize easy cases

Separation bounds : treat the worst cases.

Most expected case : "easy" cases, to be optimized.

Control the FP rounding errors \Rightarrow we use the costly exact computations rarely.

In the "good cases", we get a solution **geometrically exact** for nearly the cost of FP computation.

Dynamic filters : interval arithmetic

Idea : we replace each FP operation by an operation over an interval of FP values $[\underline{x}; \bar{x}]$ which encodes the rounding error.

Inclusion property : at each operation, the interval contains the exact value X .

Operations : we use the IEEE 754 rounding modes :

$$X + Y \longrightarrow [\underline{x} + \underline{y}; \bar{x} + \bar{y}]$$

$$X - Y \longrightarrow [\underline{x} - \bar{y}; \bar{x} - \underline{y}]$$

Optimization :

$$X + Y \longrightarrow [-((-\underline{x}) - \bar{y}); \bar{x} + \bar{y}]$$

Less rounding mode changes.

Multiplication and division of intervals

Multiplication :

$$X \times Y \longrightarrow [\min(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}); \max(\underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}, \underline{x}\underline{y})]$$

In practice, we use comparison tests for the different cases before doing the multiplications.

Division : similar.

Division by zero treatment.

Comparisons

Thanks to the **inclusion property**, if

$$[\underline{x}; \bar{x}] \cap [\underline{y}; \bar{y}] = \emptyset$$

then we can decide if $X < Y$ or $X > Y$.

Otherwise, we can not decide the comparison.

\implies **Filter failure**

Static filters

Static analysis of the rounding error propagation over the evaluation of a polynomial, supposing bounds on the inputs.

Notations : x is a real variable, \mathbf{x} its value computed with doubles, $e_{\mathbf{x}}$ and $b_{\mathbf{x}}$ are doubles such that :

$$\begin{cases} e_{\mathbf{x}} \geq |x - \mathbf{x}| \\ b_{\mathbf{x}} \geq |\mathbf{x}| \end{cases}$$

Initially, we can get a rounded value to the nearest (if the values are not representable by a double) :

$$\begin{cases} b_{\mathbf{x}} = |\mathbf{x}| \\ e_{\mathbf{x}} = \frac{1}{2}\text{ulp}(\mathbf{x}) \end{cases}$$

Addition and subtraction

Error propagation over an addition $z = x + y$ is the following :

$$\begin{cases} b_z = b_x + b_y \\ e_z = e_x + e_y + \frac{1}{2}\text{ulp}(z) \end{cases}$$

Indeed :

$$\begin{aligned} |z - \mathbf{z}| &= \left| \underbrace{(z - (x + y))}_{=0} + \underbrace{((x + y) - (x + y))}_{\leq e_x + e_y} + \underbrace{((x + y) - \mathbf{z})}_{\leq \frac{1}{2}\text{ulp}(z)} \right| \\ &\leq e_x + e_y + \frac{1}{2}\text{ulp}(z) \end{aligned}$$

Multiplication

Error propagation for a multiplication $z = x \times y$ is the following :

$$\begin{cases} b_z = b_x \times b_y \\ e_z = e_x \bar{\times} e_y \bar{+} e_y \bar{\times} |x| \bar{+} e_x \bar{\times} |y| \bar{+} \frac{1}{2} \text{ulp}(z) \end{cases}$$

Indeed :

$$\begin{aligned} |z - \mathbf{z}| &= \underbrace{|(z - (x \times y))|}_{=0} + \underbrace{|((x \times y) - (\mathbf{x} \times \mathbf{y}))|}_{=(x-x)(y-y) - (x-x) \times y - (y-y) \times x} + \underbrace{|((\mathbf{x} \times \mathbf{y}) - \mathbf{z})|}_{\leq \frac{1}{2} \text{ulp}(z)} \\ &\leq e_x \bar{\times} e_y \bar{+} e_x \bar{\times} y \bar{+} e_y \bar{\times} x \bar{+} \frac{1}{2} \text{ulp}(z) \end{aligned}$$

Application : orientation predicate

Approximate FP code :

```
int orientation(double px, double py,
               double qx, double qy,
               double rx, double ry)
{
    double pqx = qx - px,  pqy = qy - py;
    double prx = rx - px,  pry = ry - py;

    double det = pqx * pry - pqy * prx;

    if (det > 0)  return 1;
    if (det < 0)  return -1;
    return 0;
}
```

Application : orientation predicate

Code with static filters (for inputs **bounded by 1**) :

```
int filtered_orientation(double px, double py,
                       double qx, double qy,
                       double rx, double ry)
{
    double pqx = qx - px,  pqy = qy - py;
    double prx = rx - px,  pry = ry - py;

    double det = pqx * pry - pqy * prx;

    const double E = 1.33292e-15;

    if (det > E)  return 1;
    if (det < -E) return -1;

    ... // can't decide => call the exact version
}
```

Variants : Ex : computing the bound at run time

```
int filtered_orientation(double px, double py,
                       double qx, double qy,
                       double rx, double ry)
{
    double b = max_abs(px, py, qx, qy, rx, ry);

    double pqx = qx - px,  pqy = qy - py;
    double prx = rx - px,  pry = ry - py;

    double det = pqx * pry - pqy * prx;

    const double E = 1.33292e-15;

    if (det > E*b*b) return 1;
    if (det < -E*b*b) return -1;

    ... // can't decide => call the exact version
}
```


Filter failure rates probabilities

Theoretical study : [Devillers-Preparata-99]

Inputs uniformly distributed in a unit square/cube :

orientation 2D	10^{-15}
orientation 3D	5.10^{-14}
in_circle 2D	10^{-11}
in_sphere 3D	7.10^{-10}

... for data homogeneously distributed.

On more degenerate cases

	Dynamic	Semi-static
Random	0	870
$\varepsilon = 2^{-5}$	0	1942
$\varepsilon = 2^{-10}$	0	662
$\varepsilon = 2^{-15}$	0	8833
$\varepsilon = 2^{-20}$	0	132153
$\varepsilon = 2^{-25}$	10	192011
$\varepsilon = 2^{-30}$	19536	308522
Grid	49756	299505

Number of failures of dynamic and static filters during the computation of Delaunay (10^5 points). Inputs on a integer grid of 30 bits, with relative perturbation.

Comparison : dynamic vs static filters

Can fail more often than interval arithmetic (less precise), but faster.

Static filters harder to write : needs analysis of each predicate.

Fastest scheme : [cascade](#) several methods.

Filters : remarks

Fragile : try to avoid bad cases in algorithms !

- Avoid **cascaded** computations (use original inputs)
- Avoid **testing degenerate cases** if you know them (created by the algorithm).
- Avoid **constructions**, because faster solutions are available for predicates.

Current work

- Automatic code generation, from a generic version, for the best methods.
- Filtering of geometric constructions.
- Rounding of constructions.

Implementation in CGAL

Algorithms and traits classes

Algorithms are parameterized (templates) by **geometric traits classes**, which provide :

- types of the objects manipulated by the algorithm : [Point_2](#), [Tetrahedron_3](#)...
- predicates that the algorithm applies to the objects : [Orientation_2](#), [Side_of_oriented_sphere_3](#)...
- constructions : [Mid_point_2](#), [Construct_circumcenter_3](#), [Compute_squared_length_2](#)...

The last 2 are provided as **function objects**.

Needs of algorithms are described towards its traits parameter as a **concept**.

Kernels

The kernel gathers many objects types, predicates and constructions, and can be used as parameter for the traits classes directly to many algorithms.

Classical kernels, parameterized by number types :

`Cartesian<FT>`

`Homogeneous<RT>`

Ex : `Triangulation_3<Cartesian<double> >`

`Cartesian<double>` is a **model** for the concept `TriangulationTraits_3`.

The kernel functionality is also available via global functions :
`CGAL::orientation(p, q, r)..`

Number types

Valid parameters for the kernels `Cartesian...`

FP :
`double`, `float`

Multi-precision :
`Gmpz`, `Gmpq`, `CGAL::MP_Float`, `leda::integer...`

Number types including some filtering :
`leda::real`, `CORE::Expr`, `CGAL::Lazy_exact_nt<>`

Internal tools

Interval arithmetic : `CGAL::Interval_nt`, `boost::interval`

Generator of filtered predicates (dynamic) using C++ exceptions :
`CGAL::Filtered_predicate<>`

Filtered kernels

`CGAL::Filtered_kernel< K >` provides some predicates with static filters, and all others with dynamic filters.

Recommended kernels :

`CGAL::Exact_predicates_exact_constructions_kernel`

`CGAL::Exact_predicates_inexact_constructions_kernel`

Example

```
template < typename K >
struct My_orientation_2
{
    typedef typename K::RT      RT;
    typedef typename K::Point_2 Point_2;

    CGAL::Orientation
    operator()( const Point_2 &p, const Point_2 &q,
                const Point_2 &r) const
    {
        RT prx = p.x() - r.x();    RT pry = p.y() - r.y();
        RT qrx = q.x() - r.x();    RT qry = q.y() - r.y();
        return static_cast<CGAL::Orientation>(
            CGAL::sign( prx*qry - qrx*pry ) );
    }
};
```

Example

```
// Using it
```

```
typedef CGAL::Cartesian<double> Kernel;
```

```
Kernel::Point_2 p(1, 2), q(2, 3), r(4, 5);
```

```
My_orientation_2<Kernel> orientation;
```

```
CGAL::Orientation ori = orientation(p, q, r);
```

Using Filtered_predicate

```
typedef CGAL:: Simple_cartesian <double > K;
typedef CGAL:: Simple_cartesian <CGAL:: Interval_nt_advanced > FK;
typedef CGAL:: Simple_cartesian <CGAL:: MP_Float > EK;
typedef CGAL:: Cartesian_converter <K, EK> C2E;
typedef CGAL:: Cartesian_converter <K, FK> C2F;

typedef CGAL:: Filtered_predicate <My_orientation_2 <EK>,
                                   My_orientation_2 <FK>,
                                   C2E, C2F> Orientation_2;

...
K:: Point_2 p(1,2), q(2,3), r(3,4);
Orientation_2 orientation;
orientation(p, q, r);
return 0;
```