

A Generic Lazy Evaluation Scheme for Exact Geometric Computations

Sylvain Pion Andreas Fabri

INRIA Sophia-Antipolis GeometryFactory SARL

LCSD – October 22, 2006

Plan

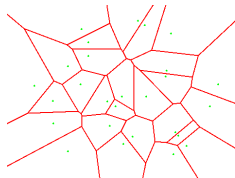
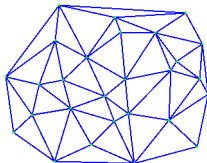
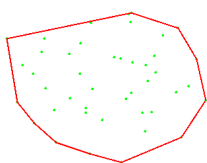
- 1 Context
- 2 Numerical robustness
- 3 Optimizing at the geometric level
- 4 Kernels
- 5 Conclusion

Plan

- 1 Context
- 2 Numerical robustness
- 3 Optimizing at the geometric level
- 4 Kernels
- 5 Conclusion

Computational Geometry

- Convex hulls, triangulations, Voronoi diagrams

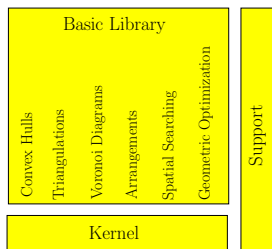


- Surface reconstruction, meshing
- Boolean operations on polygons and polyhedra
- ...

Application domains: CAD/CAM, GIS, molecular biology, medical imaging...
Handling large data sets require efficient and robust computations.

CGAL: *Computational Geometry Algorithms Library*

- Goal: implement the most important geometric algorithms
- Criteria: adaptability, efficiency, robustness
- C++ (generic programming)
- Overall architecture:



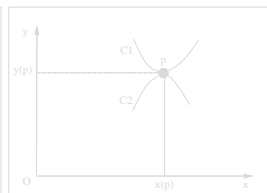
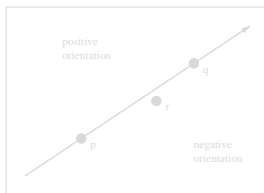
Kernel of geometric primitives

Algorithms are logically decoupled in:

- a **combinatorial** part (building a graph)
- a **numerical** part (refers to coordinates)

The latter calls kernel primitives:

- **Basic objects:** points, segments, lines, circles...
- **Predicates:** orientation, abscissa comparisons, intersection tests...
- **Constructions:** distance computations, intersection computations...



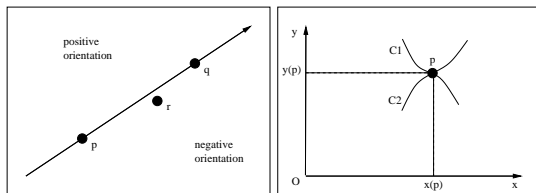
Kernel of geometric primitives

Algorithms are logically decoupled in:

- a **combinatorial** part (building a graph)
- a **numerical** part (refers to coordinates)

The latter calls kernel primitives:

- **Basic objects**: points, segments, lines, circles...
- **Predicates**: orientation, abscissa comparisons, intersection tests...
- **Constructions**: distance computations, intersection computations...



Plan

- 1 Context
- 2 Numerical robustness**
- 3 Optimizing at the geometric level
- 4 Kernels
- 5 Conclusion

Orientation predicate

`orientation(p, q, r)` is the sign of:

$$\begin{vmatrix} 1 & px & py \\ 1 & qx & qy \\ 1 & rx & ry \end{vmatrix} = \begin{vmatrix} qx - px & qy - py \\ rx - px & ry - py \end{vmatrix}$$

```
Sign orientation(Point_2 p, Point_2 q, Point_2 r)
{
  T det = (q.x() - p.x()) * (r.y() - p.y())
        - (r.x() - p.x()) * (q.y() - p.y());
  return (det > 0) ? 1 : (det < 0) ? -1 : 0;
}
```

Wrong result due to approximate computation can cause crashes or loops (invariant violations).

Orientation predicate

`orientation(p, q, r)` is the sign of:

$$\begin{vmatrix} 1 & px & py \\ 1 & qx & qy \\ 1 & rx & ry \end{vmatrix} = \begin{vmatrix} qx - px & qy - py \\ rx - px & ry - py \end{vmatrix}$$

```
Sign orientation(Point_2 p, Point_2 q, Point_2 r)
{
  T det = (q.x() - p.x()) * (r.y() - p.y())
          - (r.x() - p.x()) * (q.y() - p.y());
  return (det > 0) ? 1 : (det < 0) ? -1 : 0;
}
```

Wrong result due to approximate computation can cause crashes or loops (invariant violations).

Arithmetics

Integer/Rational arithmetic makes it robust... but slow.

Interval arithmetic is faster, and can be used to filter out easy cases.

Filtering scheme:

- evaluate values with intervals, and
- if later computations show insufficient precision, recompute with exact arithmetic.

→ functions are parameterized by the type of arithmetic (number type).

Arithmetics

Integer/Rational arithmetic makes it robust... but slow.

Interval arithmetic is faster, and can be used to filter out easy cases.

Filtering scheme:

- evaluate values with intervals, and
- if later computations show insufficient precision, recompute with exact arithmetic.

→ functions are parameterized by the type of arithmetic (number type).

Arithmetics

Integer/Rational arithmetic makes it robust... but slow.

Interval arithmetic is faster, and can be used to filter out easy cases.

Filtering scheme:

- evaluate values with intervals, and
- if later computations show insufficient precision, recompute with exact arithmetic.

→ functions are parameterized by the type of arithmetic (number type).

Arithmetics

Integer/Rational arithmetic makes it robust... but slow.

Interval arithmetic is faster, and can be used to filter out easy cases.

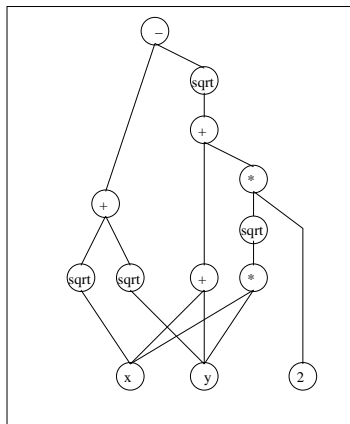
Filtering scheme:

- evaluate values with intervals, and
- if later computations show insufficient precision, recompute with exact arithmetic.

→ functions are parameterized by the type of arithmetic (number type).

Putting it all together in a "lazy exact" number type

Storing the DAG of operations in memory, ex: $\sqrt{x} + \sqrt{y} - \sqrt{x + y + 2\sqrt{xy}}$



Each node stores: its type, pointers to operands, interval, pointer to exact.

Plan

- 1 Context
- 2 Numerical robustness
- 3 Optimizing at the geometric level**
- 4 Kernels
- 5 Conclusion

Saving memory

This scheme requires lots of memory.

First thing to do is to deal with predicates ("leaf" functions):
exploit the regrouping of operations to remove the need for intermediate nodes inside.

Run the predicate with intervals, with uncertain decisions reported e.g. by exceptions.

If necessary, re-run it with multiprecision arithmetic.

Regrouping interval operations also helps saving rounding-mode changes.

Making it generic

Predicates as generic functors:

```
template <class Kernel>
struct Orientation_2
{
    typedef Kernel::Point_2    Point_2;
    typedef Kernel::FT         Number_type;

    Sign
    operator()(Point_2 p, Point_2 q, Point_2 r) const
    {
        return ...;
    }
};
```

Making it generic

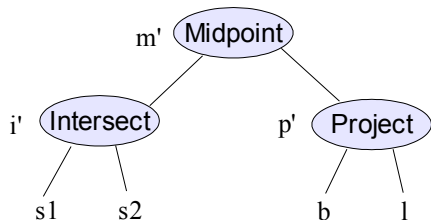
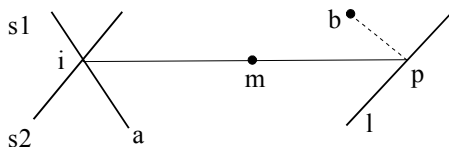
```
template <class EP, class AP, class C2E, class C2A>
struct Filtered_predicate
{
    AP    approx_predicate;    C2A    c2a;
    EP    exact_predicate;    C2E    c2e;

    typedef EP::result_type    result_type;

    template <class A1, class A2>
    result_type
    operator()(const A1 &a1, const A2 &a2) const
    {
        try {
            return approx_predicate(c2a(a1), c2a(a2));
        } catch (Interval::unsafe_comparison) {
            return exact_predicate(c2e(a1), c2e(a2));
        }
    }
};
```

Dealing with constructions

Idea: one DAG node per geometric constructions instead of arithmetic operation.



Dealing with constructions

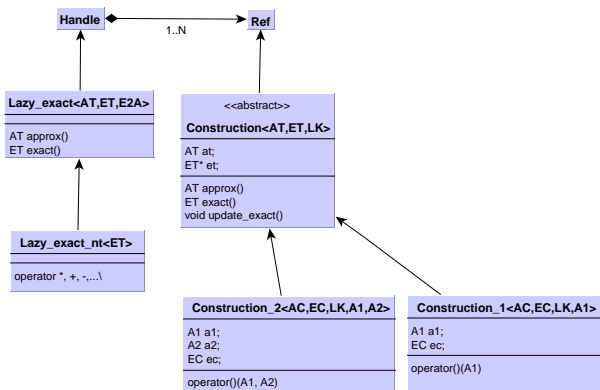
A node has 2 "types":

- a static type: `Point_2`, `Segment_3...`
- a dynamic type: the construction which constructed it.

It stores:

- an interval approximation of the static type.
- a pointer to an exact object of the static type.
- pointers to operands

Dealing with constructions



Plan

- 1 Context
- 2 Numerical robustness
- 3 Optimizing at the geometric level
- 4 Kernels**
- 5 Conclusion

Kernels

A generic functor adaptor works for constructions like `Filtered_predicate`.

Kernels regroups dozens of predicates and constructions.

Macros apply the adaptors to all functors.

Benchmarks

Kernel	time	time	mem
	g++ 3.4	g++ 4.1	
SC<Gmpq>	71	70	70
SC<Lazy_exact_nt<Gmpq>>	9.4	7.4	501
Lazy_kernel<SC<Gmpq>> (2)	4.9	3.6	64
Lazy_kernel<SC<Gmpq>>	4.1	2.8	64
SC<double>	0.98	0.72	8.3

Plan

- 1 Context
- 2 Numerical robustness
- 3 Optimizing at the geometric level
- 4 Kernels
- 5 Conclusion**

Open questions

- Is such a lazy evaluation scheme applicable to other fields?
- Specifying the level of regrouping is done manually. Can we do better?
- Expression templates do this on a statement/type level.