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...

![Diagram showing positive and negative orientation with points p, q, and r.]
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...

![Diagram](image-url)
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)
{
    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.
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:

```cpp
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_comparation) {
            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: \texttt{Point\_2}, \texttt{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

Lazy_exact<AT,ET,E2A>
AT approx()
ET exact()

Construction<AT,ET,LK>
AT at;
ET* et;
AT approx()
ET exact()
void update_exact()

Lazy_exact_nt<ET>
operator *, +, -, ... \n
Construction_2<AC,EC,LK,A1,A2>
A1 a1;
A2 a2;
EC ec;
operator()(A1, A2)

Construction_1<AC,EC,LK,A1>
A1 a1;
EC ec;
operator()(A1)
# 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

<table>
<thead>
<tr>
<th>Kernel</th>
<th>time</th>
<th>time</th>
<th>mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>g++ 3.4</td>
<td>g++ 4.1</td>
<td></td>
</tr>
<tr>
<td>SC&lt;Gmpq&gt;</td>
<td>71</td>
<td>70</td>
<td>70</td>
</tr>
<tr>
<td>SC&lt;Lazy_exact_nt&lt;Gmpq&gt;&gt;</td>
<td>9.4</td>
<td>7.4</td>
<td>501</td>
</tr>
<tr>
<td>Lazy_kernel&lt;SC&lt;Gmpq&gt;&gt; (2)</td>
<td>4.9</td>
<td>3.6</td>
<td>64</td>
</tr>
<tr>
<td>Lazy_kernel&lt;SC&lt;Gmpq&gt;&gt;</td>
<td>4.1</td>
<td>2.8</td>
<td>64</td>
</tr>
<tr>
<td>SC&lt;double&gt;</td>
<td>0.98</td>
<td>0.72</td>
<td>8.3</td>
</tr>
</tbody>
</table>
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.