Formal Certification of Arithmetic Filters for Geometric Predicates

Guillaume Melquiond  Sylvain Pion

Arénaire, LIP ENS Lyon
Geometrica, INRIA Sophia-Antipolis

July 11th 2005
Geometric algorithms and predicates

- Numerical functions are not the main basis of computational geometry algorithms. **Predicates** are: they provide the bridge between numerical inputs and combinatorial output.

- Example: orientation of three points $p$, $q$, and $r$ in the plane.

\[
\text{orient}_2(p, q, r) = \text{sign} \begin{vmatrix} q_x - p_x & r_x - p_x \\ q_y - p_y & r_y - p_y \end{vmatrix}
\]

Only three answers: clockwise, counter-clockwise, aligned.
Geometric algorithms and predicates

- Numerical functions are not the main basis of computational geometry algorithms. Predicates are: they provide the bridge between numerical inputs and combinatorial output.

- Example: orientation of three points \( p, q, \) and \( r \) in the plane.

\[
\text{orient}_2(p, q, r) = \text{sign} \left| \begin{array}{cc}
q_x - p_x & r_x - p_x \\
q_y - p_y & r_y - p_y
\end{array} \right|
\]

Only three answers: clockwise, counter-clockwise, aligned.

- Geometric algorithms are highly sensitive to the result of the predicates. Their results have to be guaranteed.
Implementing robust yet efficient predicates

- Floating-point numbers suffer from limited precision and range. In this implementation, the computed value may be different enough from the real value for their signs to differ.

```c
double pqx = qx - px, ppy = qy - py;
double prx = rx - px, pry = ry - py;
double det = pqx * pry - ppy * prx;
if (det > 0) return POSITIVE;
if (det < 0) return NEGATIVE;
return ZERO;
```
Implementing robust yet efficient predicates

- Floating-point numbers suffer from limited precision and range. In this implementation, the computed value may be different enough from the real value for their signs to differ.

```c
double pqx = qx - px, pqy = qy - py;
double prx = rx - px, pry = ry - py;
double det = pqx * pry - pqy * prx;
if (det > 0) return POSITIVE;
if (det < 0) return NEGATIVE;
return ZERO;
```

- On the other hand, computing the determinant with exact arithmetic is possible, but it is too slow to be usable.
Implementing robust yet efficient predicates

- Floating-point numbers suffer from limited precision and range. In this implementation, the computed value may be different enough from the real value for their signs to differ.

  ```c
  double pqx = qx - px, pgy = qy - py;
  double prx = rx - px, pry = ry - py;
  double det = pqx * pry - pgy * prx;
  if (det > 0) return POSITIVE;
  if (det < 0) return NEGATIVE;
  return ZERO;
  ```

- On the other hand, computing the determinant with exact arithmetic is possible, but it is too slow to be usable.

- Best of both worlds: floating-point computations, and if the computed sign may be wrong, fall back to exact arithmetic.
Outline

Introduction

Formalizing homogeneous floating-point arithmetic
   A correct floating-point filter
   Bounding expressions by homogeneous intervals
   Homogeneous floating-point arithmetic

Error bound and implementation
   Using Gappa to bound the error
   CGAL implementation of the predicates

Conclusion
A correct floating-point filter

- If the distance between the computed value $\det$ and the real value $\det$ is bounded by $\epsilon < \text{eps}$, this is a correct first stage for the predicate.

```c
double det = pqx * pry - pqy * prx;
if (det > +eps) return POSITIVE;
if (det < -eps) return NEGATIVE;
// fall back to an exact computation
```

- How to compute $\epsilon$ and guarantee it is a correct bound?
Interval arithmetic:

\[ k \cdot A = \{ k \cdot a \mid a \in A \} \]
\[ A + B = \{ a + b \mid a \in A, b \in B \} \]
\[ A \times B = \{ a \cdot b \mid a \in A, b \in B \} \]

If \( A \) and \( B \) are intervals (closed and bounded subsets of the real numbers \( \mathbb{R} \)), then \( k \cdot A \), \( A + B \), and \( A \times B \) are intervals too. Intervals are represented by their lower and upper bounds:

\[ X = [x, \overline{x}] = \{ x \in \mathbb{R} \mid x \leq x \leq \overline{x} \}. \]
Bounding homogeneous expressions

- If \( a \in k \cdot A \) and \( b \in m \cdot B \), then
  \[
  a \cdot b \in (k \cdot A) \times (m \cdot B) = (k \cdot m) \cdot (A \times B).
  \]

If \( a \in k \cdot A \) and \( b \in k \cdot B \), then
  \[
  a + b \in (k \cdot A) + (k \cdot B) = k \cdot (A + B).
  \]
Bounding homogeneous expressions

- If \( a \in k \cdot A \) and \( b \in m \cdot B \), then
  \[
a \cdot b \in (k \cdot A) \times (m \cdot B) = (k \cdot m) \cdot (A \times B).
  \]
- If \( a \in k \cdot A \) and \( b \in k \cdot B \), then
  \[
a + b \in (k \cdot A) + (k \cdot B) = k \cdot (A + B).
  \]

- Example: since
  \[
a, c \in \max(|a|, |c|) \cdot [-1, 1] \quad \text{and} \quad b, d \in \ldots,
  \]
  the range of the determinant is
  \[
  \begin{vmatrix}
  a & b \\
  c & d
  \end{vmatrix} \in \max(|a|, |c|) \cdot \max(|b|, |d|) \cdot [-2, 2].
  \]}
Floating-point rounding

- A floating-point operator behaves as if it was first computing the *infinitely precise* value and then *rounding* it so that it fits in the destination floating-point format.

- Example: if $a$ and $b$ are two floating-point numbers, the result $a \oplus b$ of their floating-point sum is equal to $\circ(a + b)$.
Floating-point rounding

- A floating-point operator behaves as if it was first computing the infinitely precise value and then rounding it so that it fits in the destination floating-point format.

- Example: if $a$ and $b$ are two floating-point numbers, the result $a \oplus b$ of their floating-point sum is equal to $\circ(a + b)$.

- If $x$ is not outside the limited range of floating-point numbers, the rounding error is bounded:

\[
| \circ(x) - x | \leq \max(\eta_0, |x| \cdot \epsilon_0).
\]
Homogeneous floating-point arithmetic

If \( a \in k \cdot A \) and \( b \in m \cdot B \), then

\[
\begin{align*}
    a \cdot b & \in (k \cdot m) \cdot (A \times B) \\
    a + b & \in k \cdot (A + B) \quad \text{if} \quad k = l \\
    \circ(a) - a & \in k \cdot E_a \quad \text{if} \quad [-\eta_0, \eta_0] \subseteq k \cdot E_a
\end{align*}
\]

with \( E_a = A \times [-\epsilon_0, \epsilon_0] \).
Homogeneous floating-point arithmetic

- If $a \in k \cdot A$ and $b \in m \cdot B$, then
  
  $$a \cdot b \in (k \cdot m) \cdot (A \times B)$$
  
  $$a + b \in k \cdot (A + B) \quad \text{if} \quad k = l$$
  
  $$\circ(a) - a \in k \cdot E_a \quad \text{if} \quad [-\eta_0, \eta_0] \subseteq k \cdot E_a$$

  with $E_a = A \times [-\epsilon_0, \epsilon_0]$.

- The rounding error range $k \cdot E$ can be computed by recursively applying these formulas to $\det - det$. The final coefficient is

  $$k = \max(|pqx|, |pqy|) \cdot \max(|prx|, |pry|).$$

- Some sub-expressions have to be rewritten so as to get a tight range. For example, $\circ(a) - b = (\circ(a) - a) + (a - b)$. 
Gappa, a tool to bound expressions

- Gappa verifies range properties on arithmetic expressions, especially expressions containing rounding operations. It also generates a formal proof of these properties.

- Gappa uses a set of theorems relying on interval arithmetic in order to bound the expressions. It rewrites the expressions to get tighter intervals when they involve rounding errors.

- Our model of homogeneous floating-point arithmetic is written so that it is close to Gappa’s own model. As a consequence, Gappa will be able to compute the error bounds in our stead.
Gappa, a tool to bound expressions

Example: the absolute error between the computed determinant and the exact value when all its rounded elements are in the interval \([-1, 1]\).

```plaintext
# some notations:
px = float64(qx - px);
py = float64(qy - py);
rx = float64(rx - px);
ry = float64(ry - py);
det = float64(pqx * pry - pqy * prx);
extact = float64((qx-px)*(ry-py) - (qy-py)*(rx-px));
{ pqx in [-1,1] \/ pqy in [-1,1] \/ prx in [-1,1] \/ pry in [-1,1] -> det - exact in ? }```

Gappa answers: \(|\text{det} - \text{exact}| \leq 6.66... \cdot 10^{-16}\).
Example: the absolute error between the computed determinant and the exact value when all its rounded elements are in the interval $[-1, 1]$.

```plaintext
# some notations:
pqx = <float64ne>(qx - px);
pqy = <float64ne>(qy - py);
prx = <float64ne>(rx - px);
pry = <float64ne>(ry - py);
det <float64ne>= pqx * pry - pqy * prx;
exact = (qx-px)*(ry-py) - (qy-py)*(rx-px);
{  pqx in [-1,1] \ pqy in [-1,1] \ prx in [-1,1] \ pry in [-1,1] -> det - exact in ? }
```

Gappa answers: $|\text{det} - \text{exact}| \leq 6.66\ldots \cdot 10^{-16}$.

However, this result comes from Gappa’s non-homogeneous model; it would lead to an inefficient implementation of the filter. We have to use the homogeneous model.
Using Gappa with our model

- We have defined two new rounding operators in Gappa that are consistent with our homogeneous model. Gappa can now use "bounded" inputs.

Example: \( \text{pqx} \in \max(|\text{pqx}|, |\text{pqy}|) \cdot [-1, 1] \).

```plaintext
# some notations:
pqx = <homogen80x_init>(qx - px);
pqy = <homogen80x_init>(qy - py);
prx = <homogen80x_init>(rx - px);
pry = <homogen80x_init>(ry - py);
det = <homogen80x> = pqx * pry - pqy * prx;
extact = (qx-px)*(ry-py) - (qy-py)*(rx-px);

# the property Gappa has to find and verify:
{ pqx in [-1,1] \&\& pqy in [-1,1] \&\& prx in [-1,1] \&\& pry in [-1,1] -> det - exact in ? }
```

- Verifying the homogeneity of the expressions is outside the scope of Gappa’s model. But it will still compute the error bound we need:

\[ |\text{det} - \text{exact}| \leq k \cdot 8.88\ldots \cdot 10^{-16}. \]
Our CGAL implementation of the predicate.

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

double maxx = max(abs(pqx), abs(prx));
double maxy = max(abs(pqy), abs(pry));
double eps = 8.8872057375927558e-16 * maxx * maxy;
if (maxx > maxy) swap(maxx, maxy);

if (maxx < 1e-146) { // underflows?
    if (maxx == 0) return ZERO;
} else if (maxy < 1e153) { // no overflow?
    double det = pqx * pry - pqy * prx;
    if (det > eps) return POSITIVE;
    if (det < -eps) return NEGATIVE;
}

// fall back to a more precise, slower method
```

This filter is robust:

- it gives up when an overflow may hinder the computations,
- otherwise it either returns the correct sign or gives up, even when a floating-point operation underflows or suffers from a double rounding.
The correct error bound is only a part of the certification of the filter implementation. Other points need to be checked:

- no overflow occurs,
- no underflow occurs when computing $\epsilon_s$,
- rounding errors also happen when computing $\epsilon_s$, the constant has to be sufficiently overestimated.

All these verifications can be done by Gappa. They use Gappa’s non-homogeneous floating-point model.
CGAL benchmarks of a 3D Delaunay triangulation with various implementations of the orientation and in-sphere predicates.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>uncertified floating-point</td>
<td>3.29</td>
</tr>
<tr>
<td>our filter + interval + exact</td>
<td>4.33</td>
</tr>
<tr>
<td>interval + exact</td>
<td>12.5</td>
</tr>
<tr>
<td>exact</td>
<td>296</td>
</tr>
<tr>
<td>Shewchuk’s predicates</td>
<td>4.39</td>
</tr>
</tbody>
</table>

Note: Shewchuk’s implementation is robust as long as there is no underflow nor overflow nor double rounding.
Designing a geometric predicate that relies on floating-point arithmetic is error-prone, in particular when you try to handle all the special situations. Relying on formal methods and the computer is a big help.
Designing a geometric predicate that relies on floating-point arithmetic is error-prone, in particular when you try to handle all the special situations. Relying on formal methods and the computer is a big help.

Our formalization of floating-point arithmetic applies to any homogeneous formulas. Because geometric predicates handle lengths, instead of unit-less values, most of them are homogeneous.
Designing a geometric predicate that relies on floating-point arithmetic is error-prone, in particular when you try to handle all the special situations. Relying on formal methods and the computer is a big help.

Our formalization of floating-point arithmetic applies to any homogeneous formulas. Because geometric predicates handle lengths, instead of unit-less values, most of them are homogeneous.

Performance-wise, our implementation of the predicates is on par with Shewchuk’s state-of-the-art implementation. But ours is robust, even when degenerate computations happens.
Questions?

Web sites:

- http://www.cgal.org/
- http://lipforge.ens-lyon.fr/www/gappa/

E-mail addresses:

- guillaume.melquiond@ens-lyon.fr
- sylvain.pion@sophia.inria.fr