FAST ROBUST ARITHMETICS FOR GEOMETRIC ALGORITHMS

T. Bartels

Technical University of Berlin, Germany

FOSDEM, 2022

Geometric Predicates.

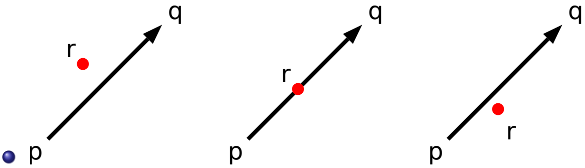
- Geometric predicates are functions that accept geometries and return discrete results.
- Here: Functions that take a fixed number of points and answer an elementary geometric question.
- Geometric predicates are used as subroutines of various geometric constructions and spatial predicates.

Examples of Geometric Predicates (1).

- 2D orientation: For $p, q, r \in \mathbb{R}^2$, the position of r w.r.t. the oriented line $\vec{pq}$ is

$$\begin{vmatrix} p_x - r_x & p_y - r_y \\ q_x - r_x & q_y - r_y \end{vmatrix} \begin{cases} > 0 & \text{left.} \\ = 0 & \text{on.} \\ < 0 & \text{right.} \end{cases}$$

(determinant of a 2×2-matrix, degree 2 polynomial)



Examples of Geometric Predicates (2).

- 2D incircle: For $p, q, r, s \in \mathbb{R}^2$, the position of s w.r.t. the ccw-oriented circle through p, q, r is

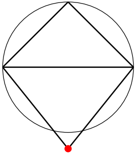
  $$\begin{vmatrix} p_x - s_x & p_y - s_y & (p_x - s_x)^2 + (p_y - s_y)^2 \\ q_x - s_x & q_y - s_y & (q_x - s_x)^2 + (q_y - s_y)^2 \\ r_x - s_x & r_y - s_y & (r_x - s_x)^2 + (r_y - s_y)^2 \end{vmatrix} \begin{cases} > 0 & \text{inside.} \\ = 0 & \text{on the boundary.} \\ < 0 & \text{outside.} \end{cases}$$

  (determinant of a 3×3-matrix, degree 4 polynomial)
- 3D orientation and insphere: determinants of 3×3-matrix, degree 3 polynomial and 4×4-matrix, degree 5 polynomial respectively.

Applications of Geometric Predicates in Algorithms.

- 2D orientation: spatial predicates, such as point-within-polygon, construction of convex hulls or triangulations.

- 2D incircle: verifying the Delaunay property in Triangulated irregular networks (TIN).



Limitations of Computer Arithmetic.

- Floating-point numbers can not represent all real values, e.g. the value of `double a` $= 0.1$ is closer to 0.100000000000000006.
- Floating-point operations generally incurs round-off errors, i.e.

  $$x + y - \varepsilon(x \oplus y) \leq x \oplus y \leq x + y + \varepsilon(x \oplus y)$$

  with machine-epsilon ε and floating-point addition $\oplus$.
- Floating-point and integer operations can overflow.
- Signs of determinants or polynomials can be computed incorrectly.

Limitations of Computer Arithmetic: Example.

- Example: Consider $p := (-0.01, -0.59)$, $q := (0.01, 0.57)$, $r := (0.15, 8.69)$ and $s := (0.07, 4.05)$.
- They all lie on the line $f(x) = 58x - 0.01$ but their nearest approximations in double precision are not collinear.
- A naive implementation of the 2D orientation predicate in double precision yields:

$$p_{\mathrm{O2D}}(p, q, s) = 0$$

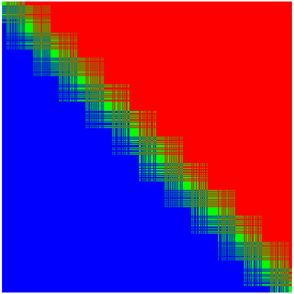
$$p_{\mathrm{O2D}}(p, r, s) = 0$$

$$p_{\mathrm{O2D}}(p, q, r) \neq 0.$$

- These results are incorrect and self-contradictory.

Limitations of Computer Arithmetic: Visualisation.

- Visualisation for 2D orientation results with a naive double precision implementation for $p := (19, 19)$, $q := (16, 16)$ and r in a very small neighbourhood of $(3.8, 3.8)$.



Robustness Issues.

- Typically, geometric algorithms are formulated and analyzed for real numbers with exact computations (real RAM).
- Incorrect predicate results can cause inconsistencies in the execution of algorithms, which can lead to incorrect results, invalid constructions, crashes or infinite loops.
- Examples:
  - Triangulations can be incorrectly connected.
  - Sequences of Delaunay edge flips may never terminate.
  - A point could be found outside of two closed polygons but within their union.
- This may be unacceptable even if correctness for edge cases is not critical.

Possible Solutions.

- Predicates could be evaluated with exact numbers types.
  - Operations on exact number types can be orders of magnitude slower than operations on built-in types.
  - This performance penalty may be prohibitive when predicates are called millions of times.
- Redundant predicate calls could be avoided to rule out inconsistencies.
  - Deciding whether a predicate call is redundant may be computationally hard.
- Inputs could be perturbed to eliminate degeneracies near-collinear points.
- The solution in this implementation uses floating-point filters:
  - Non-degenerate inputs are processed quickly and correctly.
  - Degenerate inputs are processed using exact arithmetic.

Floating-Point Filters.

- A floating-point filter is a function that returns either the correct predicate result if it can decide the problem returns that it is uncertain.
- In practice, very few predicate calls are so degenerate that they can not be decided by a filter.
- One or more filters can be used in sequence. If all filters fail, an exact stage is required.
- If the filters are fast and most predicate calls are easily decidable, we obtain robust predicates without a severe performance penalty on average.
- Existing implementations include [Shewchuk, 1997] (filters and exact stages for 2D / 3D orientation, incircle and insphere predicates by J. R. Shewchuk) and FPG (a code generator for floating-point filters presented in [Meyer and Pion, 2008]).

Floating-Point Filters: Example.

- Filter for 2D orientation: If, using native floating-point operations, the absolute value of

  $$(p_x - r_x)(q_y - r_y) - (p_y - r_y)(q_x - r_x)$$

  is greater than or equal to

  $$\left(3\varepsilon + 16\varepsilon^2\right)\left(|p_x - r_x||q_y - r_y| + |p_y - r_y||q_x - r_x|\right),$$

  then its sign is guaranteed to be correct. otherwise, we can go to higher precision.
- Otherwise, the filter fails and we can try again with a more precise filter or exact computation.
- Based on forward error analysis (proof in [Shewchuk, 1997]) that can be tedious to implement by hand.

Our implementation: Overview.

- Implemented as a project for Google Summer of Code 2020 (published at github.com/BoostGSoC20/geometry) with Boost.Geometry.
- Project was mentored by Vissarion Fisikopoulos.
- Generates filters and exact stages at compile-time.
- Header-only implementation, no special build-dependencies or steps required.
- Generates multi-stage predicates that can be extended with custom filters.

Our Implementation: Expressions.

- Arbitrary polynomial expressions can be specified in C++-syntax at compile-time.
- Polynomials are represented in the type system using expression templates.
- Notation for variables in expression is inspired by std::placeholders.

-
  ```
  //example
  constexpr auto orientation2d =
        (_3 - _1) * (_6 - _2)
      - (_5 - _1) * (_4 - _2);
  ```

Our Implementation: Floating-point filters.

- Filters are based on compile-time forward error analysis, similar to stage A in [Shewchuk, 1997].
- The expressions and constants for error bounds are computed at compile-time using template metaprogramming.
- Any floating-point type with correct rounding, such as float or double, is supported.
-

```
using filter = forward_error_semi_static
                   <orientation2d, double>;
```

Our Implementation: Exact stage and extensibility.

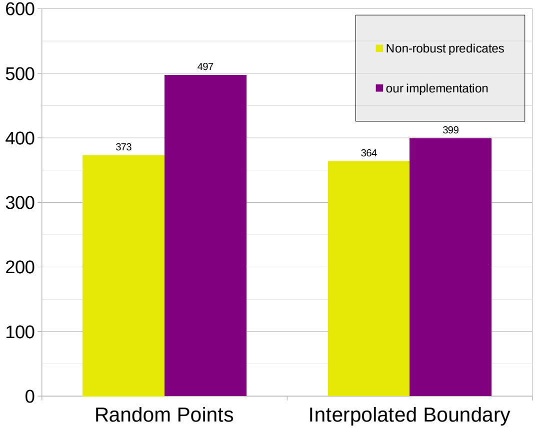
- Exact stages are evaluated using floating-point expansion arithmetic, as described as stage D in [Shewchuk, 1997].
- The basic idea of floating-point expansions is storing numbers in multiple components to extend precision.
- E.g. double-double arithmetic can be viewed as a form of expansion-arithmetic with two components.
- The required memory is known at compile-time (no heap allocation necessary).
- Exact stages and custom filters can be added to our implementation using exact and interval-number types such as those found in CGAL.

Full Example.

```
constexpr auto orientation2d =
      (_3 - _1) * (_6 - _2)
    - (_5 - _1) * (_4 - _2);

using filter = forward_error_semi_static
                 <orientation2d, double>;
using exact_stage =
        stage_d<orientation2d, double>;

staged_predicate<filter, exact_stage> p;

p.apply(px, py, qx, qy, rx, ry);
```

Performance in spatial predicates.

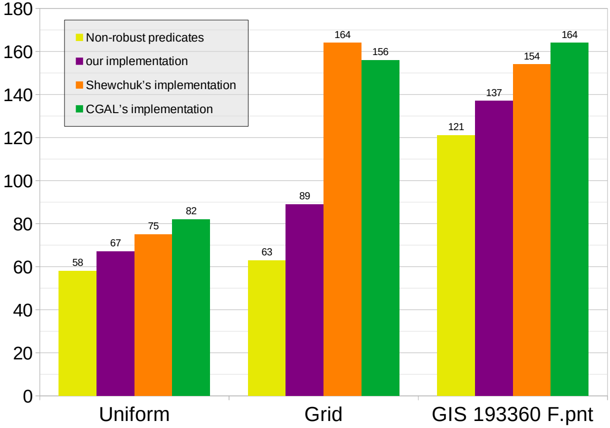
- Comparison of timings in ms to determine whether 20,000 generated points are within a polygon of 22,907 points representing Russia.



- The non-robust version produces multiple incorrect results.

Performance in Delaunay Triangulation.

- Data sets: uniformly random points, grid points and GIS data, described in [Špelič et al., 2008].



- The speed (in ms) of our predicates was compared to the speed of naive predicates and robust predicates of Shewchuk and CGAL.

Conclusion.

- Fast, robust predicates can make algorithms and spatial predicates robust at acceptable runtime cost.
- Our new implementation of robust predicates can be used for arbitrary, polynomial predicate expressions.
- No code-generation tools/steps required.
- The performance of our implementation of robust predicates is competitive when compared to established solutions.

References.

Meyer, A. and Pion, S. (2008).
FPG: A code generator for fast and certified geometric predicates.
In *Real Numbers and Computers*, pages 47–60, Santiago de Compostela, Spain.
`https://hal.inria.fr/inria-00344297`.

Shewchuk, J. R. (1997).
Adaptive precision floating-point arithmetic and fast robust geometric predicates.
Discrete & Computational Geometry, 18(3):305–363.

Špelič, D., Novak, F., and Žalik, B. (2008).
Delaunay triangulation benchmarks.
Journal of Electrical Engineering, 59(1):49–52.