Geometric Predicates

Kurt Mehlhorn

Christoph Burnikel Rudolf Fleischer Stefan Funke Susan Hert
Stefan Näher (Trier)
Michael Seel Stefan Schirra Susanne Schmitt

Max-Planck-Institute for Computer Science

Overview

- geometric computing and predicate evaluation
- dangers of imprecise arithmetic
- predicate evaluation
  - rational arithmetic and floating point filters
  - sign tests for algebraic numbers
- an application: sweeping circular arcs
- kernel issues: coordinate-centered versus geometry-centered
- LOOK: a proposal for a geometry-centered kernel
- high-level filtering

feel free to interrupt at any time

A High-Level View of Geometric Computing

- takes numerical data, e.g., a set of n points given by their Cartesian coordinates.
- computes a combinatorial object, e.g., the Voronoi diagram or the convex hull, defined by the objects.
- branches based on geometric predicates, e.g., $orientation(p,q,r)$ or $in\_circle(p,q,r,s)$.
- tests are usually not independent
- may reach "impossible" states, if predicates are evaluated incorrectly, e.g.,
  - a point outside a polygon sees no edge of the polygon
  - a point outside a polygon sees all edges of the polygon
- algorithm will crash or compute nonsense.

Floating Point Arithmetic Destroys Geometry

The intersection point of two lines lies on both lines.

- create four points a, b, c, and d with random integer coordinates in $[0..10^4]$
- create the lines $l_1(a,b)$ and $l_2(c,d)$
- increment *count* if l_1 and l_2 are parallel or $p = l_1 \cap l_2$ lies on l_1
- repeat experiment 10^5 times

```
count = 0; max_coord = 10000;
for (i = 0; i < 100000; i++)
{ point a, b, c, d, p;
  random_point(a,max_coord);                    // same for b, c, and d

  line l1(a,b), l2(c,d);                        // create lines

  if ( !l1.intersection(l2,p) || l1.contains(p) )  count++;
     // if l1 and l2 are parallel or intersection point lies on l1
}
cout << count; // outputs 33178
```

and with LEDA's Exact Geometry Kernel

```
count = 0;  max_coord = 10000;
for (i = 0; i < 100000; i++)
{ rat_point a, b, c, d, p;                // !!!!
  random_point(a,max_coord);
  rat_line l1(a,b), l2(c,d);             // !!!!
  if ( !l1.intersection(l2,p) || l1.contains(p) ) count++;
}
cout << count; // outputs 100000
```

- CGAL's and LEDA's exact geometry kernel **guarantee** that software objects behave like their mathematical counterparts
- overhead in running time is a small constant factor

Academic Problem??? Boolean Operations on Polygons

- construct a regular n-gon P, (or cylinder)
- obtain Q from P by a rotation by α degrees about its center,
- compute the union of P and Q (= a $4n$ gon).

System	n	α	time	output
ACIS	1000	1.0e-4	5 min	correct
ACIS	1000	1.0e-5	4.5 min	correct
ACIS	1000	1.0e-6	30 sec	problem too difficult
Microstation95	100	1.0e-2	2 sec	correct
Microstation95	100	0.5e-2	3 sec	incorrect answer
Rhino3D	200	1.0e-2	15sec	correct
Rhino3D	400	1.0e-2	–	CRASH
CGAL/LEDA	5000	6.175e-06	30 sec	correct
CGAL/LEDA	5000	1.581e-09	34 sec	correct
CGAL/LEDA	20000	9.88e-07	141 sec	correct

Exact Evaluation of Predicates

- amounts to computing the sign of expressions, e.g.,

$$orientation(p,q,r) = sign \begin{vmatrix} 1 & 1 & 1 \\ x_p & x_q & x_r \\ y_p & y_q & y_r \end{vmatrix}$$

- sign evaluation
  - for rational expressions
    * floating point filter
    * exact evaluation with rational arithmetic as fall back
  - for algebraic expressions:
    * separation bounds
    * sign test = numerical evaluation + separation bounds
- high level predicates:
  - which face contains the point p?

Floating Point Filter

To determine $sign(E)$

- evaluate with floating point arithmetic: $\tilde{E}$
- compute an error bound B
- if $(|\tilde{E}| > B)$ return the sign of $\tilde{E}$
- otherwise, evaluate E with exact arithmetic and return the sign obtained

Insight: sign determination is possible with approximate value computation

- all geometric predicates in LEDA and CGAL are filtered in this way.
- generation of filters is automated (expression compiler by Stefan Funke and Sylvain Pion)
- experience: exact evaluation is rarely needed

Error Bound Computation

- we use a clever variant of interval arithmetic, see LEDAbook for proofs
- $B = 2^{-53} \cdot ind_E \cdot mes_E$
- ind_E is precomputed from the structure of E.
- mes_E is computed on-line

E	$\tilde{E}$	mes_E	ind_E
a, integer	$fl(a)$	$\lvert fl(a)\rvert$	1
a, float int	$fl(a)$	$\lvert fl(a)\rvert$	0
$A+B$	$\tilde{A} \oplus \tilde{B}$	$mes_A \oplus mes_B$	$1+\max(ind_A, ind_B) \cdot \delta$
$A-B$	$\tilde{A} \ominus \tilde{B}$	$mes_A \oplus mes_B$	$1+\max(ind_A, ind_B) \cdot \delta$
$A \cdot B$	$\tilde{A} \odot \tilde{B}$	$mes_A \odot mes_B$	(*)

$$(*) = 1 + (ind_A + ind_B + 2^{-53} \cdot ind_A \cdot ind_B) \cdot \delta \qquad \delta = 1 + 2^{-53}$$

- standard interval arithmetic is an alternative, slightly slower

Efficacy of Floating Point Filter

	Orientation			Side of circle		
d	number	exact	%	number	exact	%
8	130431	0	0.00	64176	0	0.00
10	147814	0	0.00	77409	136	0.18
12	149233	0	0.00	78693	105	0.13
22	149057	0	0.00	78695	113	0.14
32	149059	0	0.00	78695	115	0.15
42	149059	0	0.00	78695	115	0.15
∞	149059	0	0.00	78695	115	0.15

Delaunay diagram computation for 10000 random points in the unit square.

$d =$ precision (number of binary places) used for the Cartesian coordinates

number of tests, number (percentage) of tests requiring exact arithmetic

Efficiency of Floating Point Filter

d	Float kernel	Rat kernel	RK without filter
8	2.58	3.59	16.33
10	2.8	3.98	18.36
12	2.83	4.04	18.63
22	2.82	4.02	20.51
32	2.86	3.96	20.77
42	2.83	4.01	26.02
∞	2.83	3.99	33.2

10000 random points in the unit square

d = the precision (number of binary places) used for the Cartesian coordinates

running time in seconds for floating point kernel, for rational kernel with floating point filter, for rational kernel without floating point kernel

Sign Test of Algebraic Expressions

Examples:

- determine sign of $\sqrt{17}+\sqrt{21}-\sqrt{38+2\sqrt{361}}$
- compare $\frac{17+\sqrt{21}}{19}$ and $\frac{18+\sqrt{22}}{20}$

Motivation

- evaluation of geometric predicates (incircle, side-of) amounts to sign determination
- non-linear objects and distances lead to expressions involving roots
- want a general method for computing signs of algebraic numbers

The Separation Bound Approach

- a *separation bound* for a class of expressions is an **easily computable** function *sep* mapping expressions into **positive** real numbers such that

$$\text{for every expression } E: \qquad val(E) \neq 0 \quad \Longrightarrow \quad |val(E)| \geq sep(E)$$

The Separation Bound Approach

- a *separation bound* for a class of expressions is an **easily computable** function *sep* mapping expressions into **positive** real numbers such that

$$\text{for every expression } E: \qquad \mathit{val}(E) \neq 0 \quad \Longrightarrow \quad |\mathit{val}(E)| \geq \mathit{sep}(E)$$

- separation bounds yield sign test based on numerical computation

$\varepsilon \leftarrow 0.5$;
while (true)
{ compute an approximation $\tilde{E}$ with $|\mathit{val}(E) - \tilde{E}| < \varepsilon$;
  if ($|\tilde{E}| \geq \varepsilon$) return $\mathit{sign}(\tilde{E})$;
  if ($\varepsilon < \mathit{sep}(E)/2$) return "sign is zero"; // since $|\mathit{val}(E)| \leq \varepsilon + \varepsilon < \mathit{sep}(E)$
  $\varepsilon \leftarrow \varepsilon^2$;
}

- worst case complexity is determined by separation bound:

  logarithm of separation bound = # of bits required for the computation

- easy cases are decided quickly (a **big** plus of the separation bound approach)

The Separation Bound Approach

- a *separation bound* for a class of expressions is an **easily computable** function *sep* mapping expressions into **positive** real numbers such that

$$\text{for every expression } E: \qquad val(E) \neq 0 \quad \Longrightarrow \quad |val(E)| \geq sep(E)$$

- Mignotte (82), Canny (87), Yap/Dube (95), BFMS (97), Scheinermann (00), Li/Yap (01), this paper (01)
- separation bound approach to sign determination is used in
  - number type *Expr* in the CORE-package (Dube/Li/Yap)
  - number type *real* in LEDA

The BFMS-bound

$E =$ an expression, integral operands, operations $+, -, *, /, \sqrt[k]{\ }$ for integral $k \geq 2$.

$D(E) =$ product of the indices of the radical operations in E. (the index of $\sqrt[k]{\ }$ is k.)

define $u(E)$ and $l(E)$ inductively (it suffices to approximate u and l)

	$u(E)$	$l(E)$
integer N	$\lvert N \rvert$	1
$E_1 \pm E_2$	$u(E_1) \cdot l(E_2) + l(E_1) \cdot u(E_2)$	$l(E_1) \cdot l(E_2)$
$E_1 \cdot E_2$	$u(E_1) \cdot u(E_2)$	$l(E_1) \cdot l(E_2)$
E_1 / E_2	$u(E_1) \cdot l(E_2)$	$l(E_1) \cdot u(E_2)$
$\sqrt[k]{E_1}$	$\sqrt[k]{u(E_1)}$	$\sqrt[k]{l(E_1)}$

$$\text{Then} \quad val(E) = 0 \quad \text{or} \quad \left(l(E) u(E)^{D(E)^2 - 1} \right)^{-1} \leq |val(E)| \leq u(E) l(E)^{D(E)^2 - 1}.$$

If E is division-free, $l(E) = 1$, and the above bound holds with $D(E)^2$ replaced by $D(E)$.

Challenges: improved bound for division, larger class of expressions.

The BFMS-Bound for Division-Free Expressions

	$u(E)$
integer N	$\|N\|$
$E_1 \pm E_2$	$u(E_1) + u(E_2)$
$E_1 \cdot E_2$	$u(E_1) \cdot u(E_2)$
$\sqrt[k]{E_1}$	$\sqrt[k]{u(E_1)}$

Then $val(E) = 0$ or

$$\left(u(E)^{D(E)-1}\right)^{-1} \le |val(E)| \le u(E).$$

- $u(E)$ is an upper bound on the value of E and all its conjugates
- the product of $val(E)$ times all its conjugates is an integer

  (namely the constant coefficient of the minimal polynomial)
- thus

$$val(E) \cdot u(E)^{D(E)-1} \ge 1$$

Reducing General Expressions to Division-Free Expressions

- transform any general expression E into E_1/E_2 where E_1 and E_2 are division-free
- apply "division-free" bounds to E_1 and E_2
- BFMS use the obvious transformation rules

$$\frac{A}{B}+\frac{C}{D} \Longrightarrow \frac{AD+BC}{BD} \qquad \frac{A}{B}\cdot\frac{C}{D} \Longrightarrow \frac{AC}{BD} \qquad \sqrt[k]{\frac{A}{B}} \Longrightarrow \frac{\sqrt[k]{A}}{\sqrt[k]{B}}$$

- each root operation is replaced by two and hence the degree is squared

Reducing General Expressions to Division-Free Expressions

- transform any general expression E into E_1/E_2 where E_1 and E_2 are division-free
- apply "division-free" bounds to E_1 and E_2
- BFMS use the obvious transformation rules

$$\frac{A}{B}+\frac{C}{D} \Longrightarrow \frac{AD+BC}{BD} \qquad \frac{A}{B}\cdot\frac{C}{D} \Longrightarrow \frac{AC}{BD} \qquad \sqrt[k]{\frac{A}{B}} \Longrightarrow \frac{\sqrt[k]{A}}{\sqrt[k]{B}}$$

- each root operation is replaced by two and hence the degree is squared
- **Observation:** the following rule may be used instead of the last rule

$$\sqrt[k]{\frac{A}{B}} \Longrightarrow \frac{\sqrt[k]{AB^{k-1}}}{B} \quad \text{or} \quad \frac{A}{\sqrt[k]{A^{k-1}B}}$$

  and the degree stays invariant

- we had only the first alternative; Chee Yap pointed out to us that it is worthwhile to have both (see below)

Improved BFMS-bound (BFMSS)

additional operation $\diamond(j, E_d, \ldots, E_0)$: j-th root of $\sum_i E_i x^i$

	$u(E)$	$l(E)$
integer N	$\lvert N \rvert$	1
$E_1 \pm E_2$	$u(E_1) \cdot l(E_2) + l(E_1) \cdot u(E_2)$	$l(E_1) \cdot l(E_2)$
$E_1 \cdot E_2$	$u(E_1) \cdot u(E_2)$	$l(E_1) \cdot l(E_2)$
E_1 / E_2	$u(E_1) \cdot l(E_2)$	$l(E_1) \cdot u(E_2)$
$\sqrt[k]{E_1}$ and $u(E_1) \geq l(E_1)$	$\sqrt[k]{u(E_1) l(E_1)^{k-1}}$	$l(E_1)$
$\sqrt[k]{E_1}$ and $u(E_1) < l(E_1)$	$u(E_1)$	$\sqrt[k]{(u(E_1)^{k-1} l(E_1)}$
$\diamond(j, E_d, \ldots, E_0)$	not given here	not given here

$D(E) =$ product of indices of root-ops and degrees of polynomial-ops.

$$val(E) = 0 \quad \text{or} \quad \left(l(E) u(E)^{D(E)-1} \right)^{-1} \leq |val(E)| \leq u(E) l(E)^{D(E)-1}$$

Discussion of Bounds I

- Mignotte (82), Canny (87), Yap/Dube (95), BFMS (97), Scheinermann (00), Li/Yap (01), BFMSS (01)
- BFMS dominates Mignotte, Canny, Yap/Dube and Scheinermann
- Canny's bound applies to more general situations than discussed here
- BFMSS dominates BFMS
- for division-free expressions: Li/Yap = BFMS = BFMSS
- (Li/Yap) BFMS and Li/Yap are incomparable
- BMFSS and Li-Yap are sometimes much better than BFMS

  $F = \sqrt[k]{x/a} \qquad E = F - F \qquad$ x is a k-bit integer, a has $O(1)$ bits

  log of BFMS bound is $\Theta(k^2)$, log of BFMSS bound is $\Theta(k)$
- if there is no sharing of subexpressions: *bit bound* $\leq D(E) \cdot$ *length of expression*

Discussion of Bounds II

- BFMSS and Li/Yap are incomparable

BFMSS wins:

$$E_0 = 17/3 \qquad F_i = \sqrt{E_{i-1}} \qquad E_i = F_i + F_i \qquad E = E_k - E_k$$

log of BFMSS bound is $\Theta(2^k)$, log of Li/Yap bound is $\Theta(2^{k^2})$

Li/Yap wins:

$$E_0 = 17/3 \qquad E = E_0^{2\cdot 2\cdots 2\cdot 1/2\cdot 1/2\cdots 1/2} \qquad k \text{ times}$$

log of BFMSS bound is $\Theta(k^2)$, log of Li/Yap bound is $\Theta(k)$

- implementations should compute both bounds and take the better

Benefit for Users of LEDA Number Type *real*?

- *real*s (as does Li/Yap's EXPR-package) support
  - sign computations and comparisons between expressions having
  - integral or rational or double operands
  - operations $+, -, *, /, \sqrt[k]{\ }$ for integral $k \geq 2$.
  - sign computations and comparisons **are exact**
- reimplementation of *real*'s
  - new separation bounds
  - improved bigfloat implementation for computing approximations
  - improved floating point filters for handling easy cases
- running times

  overall: about three times faster than old implementation with same sep bounds

  new bounds: sometimes significant improvement

  comparison with CORE/Expr: usually much faster due to filter technology

Running Times, Continued

easy instances:

- about 10 to 15 times the running time of *double*
- can be further improved by filtering on the predicate level

hard instances: speedup between three and infinity

$F = \sqrt[k]{x/a} \qquad E = F - F \qquad x$ is a $100k$-bit integer, a has $O(1)$ bits

k	64	128	256	512
old reals	13.8	139.12	1350.7	–
new reals	0.04	0.14	0.47	1.54

log of BFMS bound is $\Theta(k^2)$, required bit length quadruples when k is doubled, running time is multiplied by $4^{\log 3} = 9$

log of BFMSS bound is $\Theta(k)$, bit length doubles when k is doubled, running time is multiplied by $2^{\log 3} = 3$

An Application: Sweeping Circular Arcs

insert slides on circular arc sweep

LOOK: A Proposal for a Geometry-Centered Kernel

insert slides on LOOK

High-Level Filtering

insert slides on high-level filtering

Reduction of Arithemtic Demand: Topological Consistency Approach

Idea:

- Consider structure (topology) before numerics
- Use numerics only to choose between multiple consistent structures.

Results:

- Algorithms that always produce some output no matter how imprecise the numerics.
- Output converges to correct result as precision increases.

  (this is an informal statement; proof is missing)
- Sugihara is main proponent

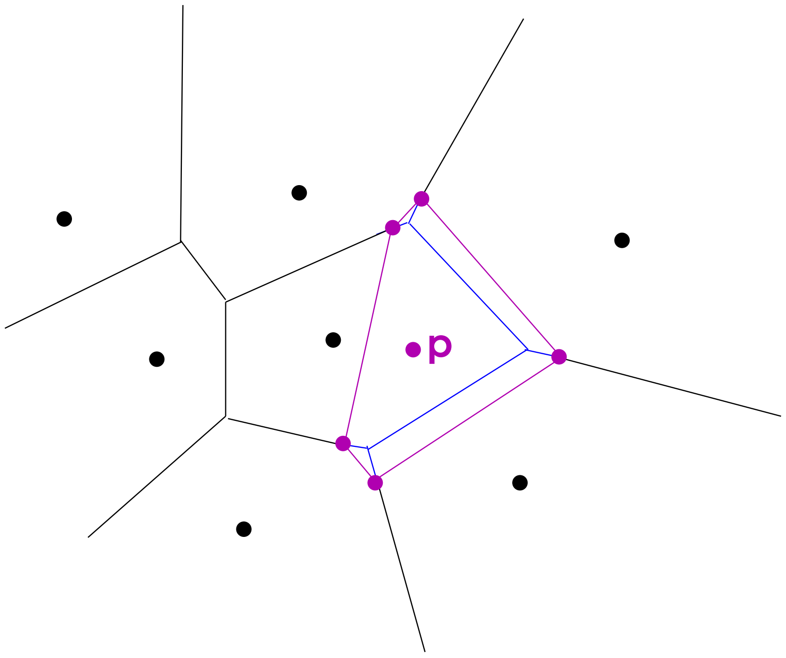
Incremental Construction of Voronoi Diagram

For each point p:

- Construct region R_p for new point p from bisectors with existing points.
- Remove the edges contained in R_p

Facts:

- The edges removed form a tree.
- The edges removed from each region are connected.

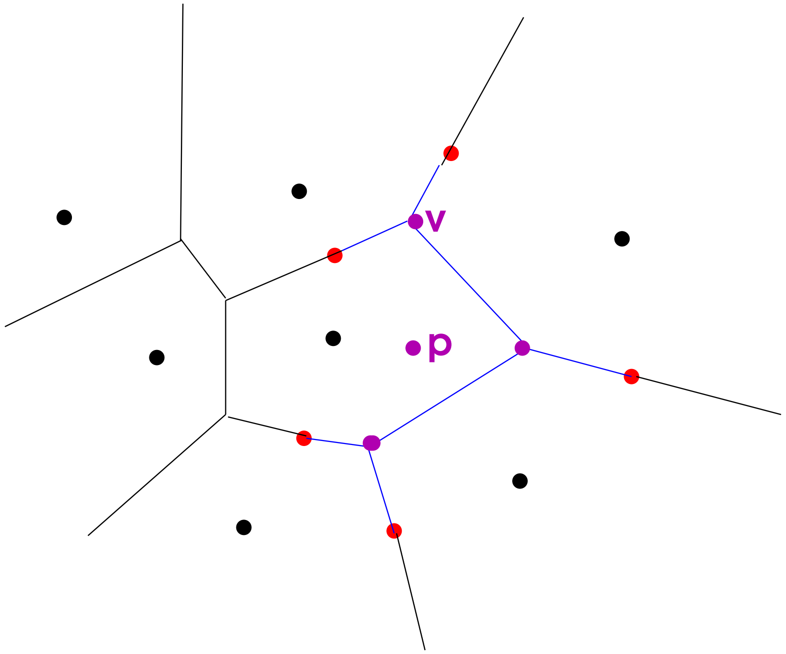


Topologically Consistent Construction

View Voronoi diagram as a graph.

- find a "seed vertex" v
- from v find a set of connected vertices V_0
- generate new nodes in graph on the cut edges for the subtree induced by V_0

Use numerics to decide which vertices to remove.



Topologically Consistent Algorithms

- Enforce structural invariants in their construction
- Shift emphasis from (often faulty) numerics to combinatorics
- Degeneracies are not special since these are defined by numerics
- Equally as correct as non-topologically oriented algorithms
- Results degrade nicely in the presence of imprecision. (informal statement)
- Alg always computes something.
- reasonably wide applicability (Voronoi diagrams, Delaunay diagrams,

Geometric Filtering and Lazy Kernel

insert slides on geometric filtering and lazy kernel