An introduction to computational geometry

ENSG Option Géologie Numérique

Marc Pouget

présentation trés largement inspirée du travail de Xavier Goaoc

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE
INRIA

Loria
Laboratoire lorrain de recherche
en informatique et ses applications

Question 1: What is it all about?

It's about designing algorithmic solutions to geometric problems.

Ex: *path planning, geometric model manipulation, visibility computation...*

It's about designing algorithmic solutions to geometric problems.

Ex: *path planning, geometric model manipulation, visibility computation...*

Ideally we want solutions that are:

- ⋆ Proven correct.
- ⋆ Proven efficient.
- ⋆ Work in practice.

It's about designing algorithmic solutions to geometric problems.

Ex: *path planning, geometric model manipulation, visibility computation...*

Ideally we want solutions that are:

⋆ Proven correct.

In a mathematical sense.

Often based on geometric arguments.

Often requires new geometric insight.

⋆ Proven efficient.

In the sense of complexity theory.

Complexity of algorithms are analyzed in an adequate model of computation.

Understanding the complexity of the problems themselves is important.

⋆ Work in practice.

Algorithms that are simple enough to be implemented.

Implementations that handle degeneracies and finite precision arithmetic.

It's about designing algorithmic solutions to geometric problems.

Ex: *path planning, geometric model manipulation, visibility computation...*

Ideally we want solutions that are:

- ⋆ Proven correct.
  - In a mathematical sense.
  - Often based on geometric arguments.
  - Often requires new geometric insight.
- ⋆ Proven efficient.
  - In the sense of complexity theory.
  - Complexity of algorithms are analyzed in an adequate model of computation.
  - Understanding the complexity of the problems themselves is important.
- ⋆ Work in practice.
  - Algorithms that are simple enough to be implemented.
  - Implementations that handle degeneracies and finite precision arithmetic.

Model of computation

Definition of the operations allowed in an algorithm and their cost.

Goal: estimate the ressources required by an algorithm

as a function of the input size.

Ex: execution time, memory space, number of I/O transfers, number of processors...

Principle: we do not want a precise cost estimation

We want to speak of algorithms independently of the technology.

We want to compare algorithms, not implementations.

Model of computation

Definition of the operations allowed in an algorithm and their cost.

Goal: estimate the ressources required by an algorithm

as a function of the input size.

Ex: execution time, memory space, number of I/O transfers, number of processors...

Principle: we do not want a precise cost estimation

We want to speak of algorithms independently of the technology.

We want to compare algorithms, not implementations.

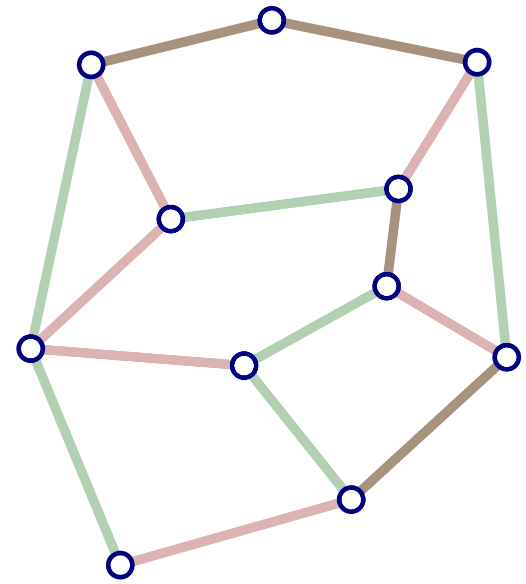
Classical model in CG: Real RAM model

Allows manipulation of real (as in $\mathbb{R}$) numbers.

Input size n → complexity $f(n) = \max_{\text{input } |X|=n} f(X)$

Care about asymptotic order of magnitude of f $(O(), \Omega(), \Theta())$.

Brute force does not scale well (or: why should we think?)

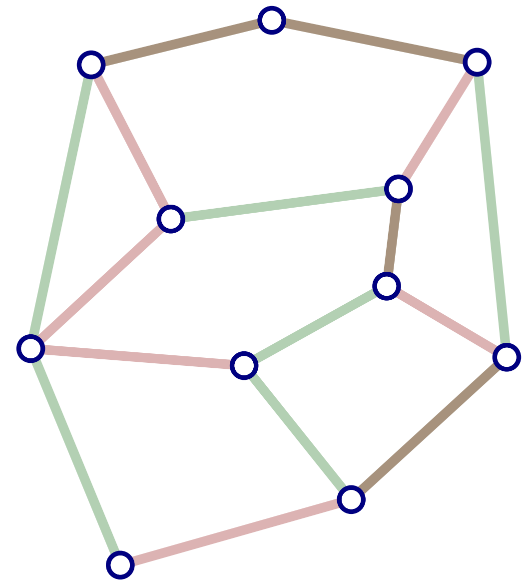


The ”Travelling salesman problem”.

Input: n cities and all inter-city distances.

Output: order on the cities that minimizes the distance travelled.

Brute force does not scale well (or: why should we think?)



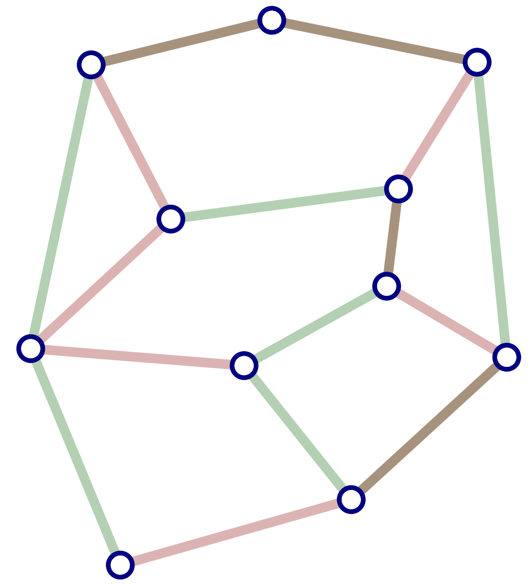
The "Travelling salesman problem".

Input: n cities and all inter-city distances.

Output: order on the cities that minimizes the distance travelled.

Brute-force approach: test all $n!$ orders and pick the best.

Brute force does not scale well (or: why should we think?)



The "Travelling salesman problem".

Input: n cities and all inter-city distances.

Output: order on the cities that minimizes the distance travelled.

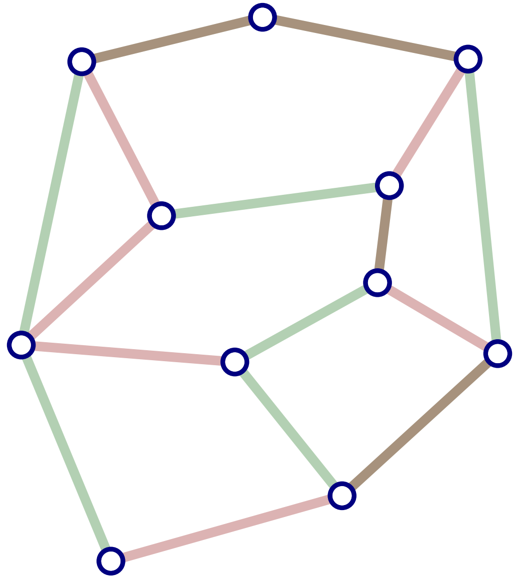
Brute-force approach: test all $n!$ orders and pick the best.

Assume your computer can process 10^{15} orders per second.

Generate the order, add-up the distances, compare to the current best...

A very generous over-estimation.

Brute force does not scale well (or: why should we think?)



The "Travelling salesman problem".

Input: n cities and all inter-city distances.

Output: order on the cities that minimizes the distance travelled.

Brute-force approach: test all $n!$ orders and pick the best.

Assume your computer can process 10^{15} orders per second.

Generate the order, add-up the distances, compare to the current best...

A very generous over-estimation.

Start the computation now. It will end...

in 30-60 min. for $n = 20$.

in two weeks for $n = 22$.

in twenty years for $n = 24$.

in four centuries for $n = 25$.

in the dark for $n = 30$.

Orders of magnitude

Sort by increasing asymptotic orders of magnitude:

$$n,\ 2^n,\ n^2,\ n!,\ \sqrt{n},\ \log n,\ \log^* n,\ 2^{n^2}$$

Orders of magnitude

Sort by increasing asymptotic orders of magnitude:

$$n,\ 2^n,\ n^2,\ n!,\ \sqrt{n},\ \log n,\ \log^* n,\ 2^{n^2}$$

$$\log^* n \ll \log n \ll \sqrt{n} \ll n \ll n^2 \ll 2^n \ll n! \ll 2^{n^2}$$

Orders of magnitude

Sort by increasing asymptotic orders of magnitude:

$$n,\ 2^n,\ n^2,\ n!,\ \sqrt{n},\ \log n,\ \log^* n,\ 2^{n^2}$$

$$\log^* n \ll \log n \ll \sqrt{n} \ll n \ll n^2 \ll 2^n \ll n! \ll 2^{n^2}$$

$$\log^* n \ll \log^a n \ll n^b \ll 2^{cn} \ll (n!)^d \ll 2^{en^2}$$

$$\forall a, b, c, d, e \in \mathbb{R}_+$$

Orders of magnitude

Sort by increasing asymptotic orders of magnitude:

$$n,\ 2^n,\ n^2,\ n!,\ \sqrt{n},\ \log n,\ \log^* n,\ 2^{n^2}$$

$$\log^* n \ll \log n \ll \sqrt{n} \ll n \ll n^2 \ll 2^n \ll n! \ll 2^{n^2}$$

$$\log^* n \ll \log^a n \ll n^b \ll 2^{cn} \ll (n!)^d \ll 2^{en^2}$$

$\forall a, b, c, d, e \in \mathbb{R}_+$

Three classes of problems

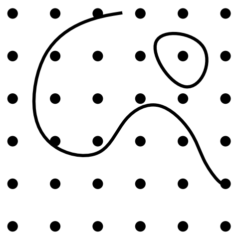
Undecidable: no algorithm will solve the problem. Ever.

NP-hard: conjectured unlikely that a polynomial-time algorithm exists.

Polynomial-time: solvable by an algorithm with complexity $O(n^c)$

for some constant c.

Hilbert's tenth problem



Input: a polynomial P in n variables with integer coefficients.

Output: yes if P has a integer solution, no otherwise.

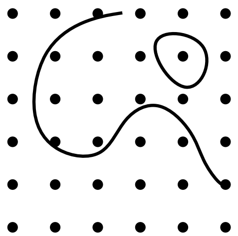
Ex: $P(x_1, x_2, x_3) = x_1^2 + 3x_1x_2 - 2x_2^2 + 4x_3 + 3$

Tenth question in Hilbert's list of *Problèmes futurs des mathématiques*.

Raised in 1900. Algorithmic question before the age of computers.

"Solved" in 1970 by Y. Matiyasevitch.

Hilbert's tenth problem



Input: a polynomial P in n variables with integer coefficients.

Output: yes if P has a integer solution, no otherwise.

Ex: $P(x_1, x_2, x_3) = x_1^2 + 3x_1x_2 - 2x_2^2 + 4x_3 + 3$

Tenth question in Hilbert's list of *Problèmes futurs des mathématiques*.

Raised in 1900. Algorithmic question before the age of computers.

"Solved" in 1970 by Y. Matiyasevitch.

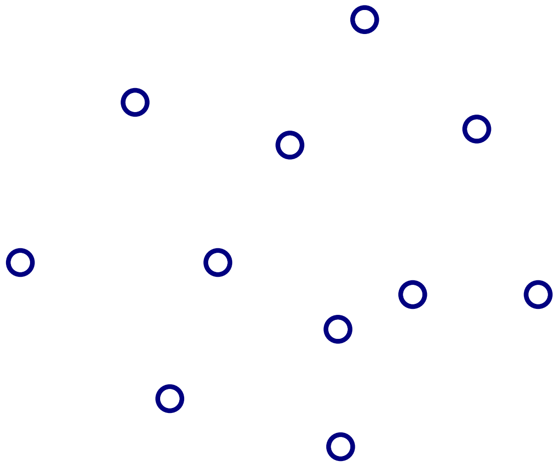
UNDECIDABLE

Minimum-weigth triangulation

Given a set of points in the plane

find a triangulation of the convex hull

that minimizes the sum of edge lengths.

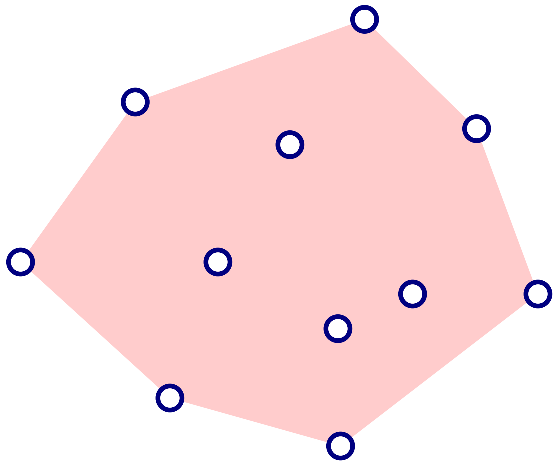


Minimum-weigth triangulation

Given a set of points in the plane

find a triangulation of the convex hull

that minimizes the sum of edge lengths.

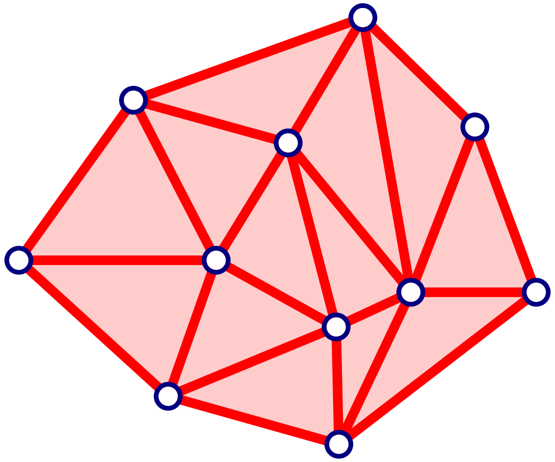


Minimum-weigth triangulation

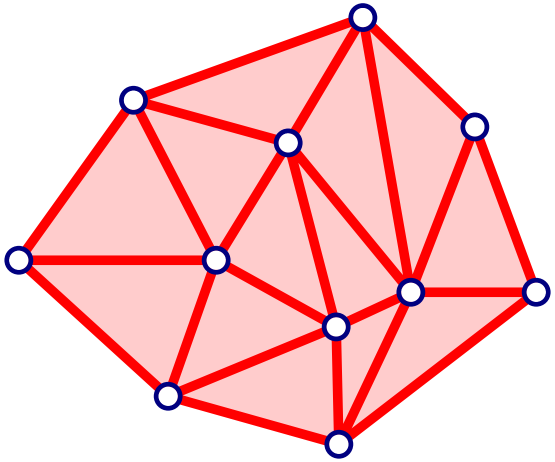
Given a set of points in the plane

find a triangulation of the convex hull

that minimizes the sum of edge lengths.



Minimum-weigth triangulation



Given a set of points in the plane

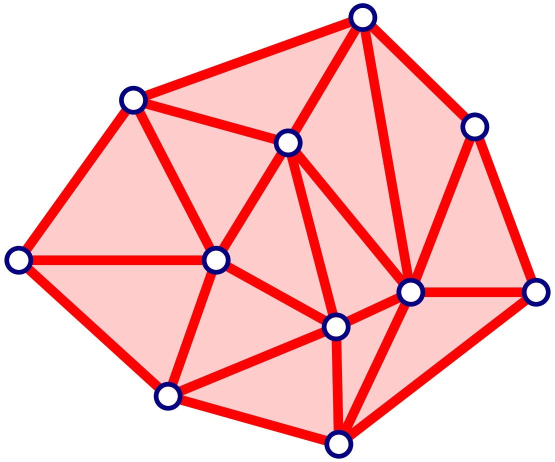
find a triangulation of the convex hull

that minimizes the sum of edge lengths.

Raised in the early 1970's.

"Solved" in 2008 by Mulzer and Rote.

Minimum-weigth triangulation



Given a set of points in the plane

find a triangulation of the convex hull

that minimizes the sum of edge lengths.

Raised in the early 1970's.

"Solved" in 2008 by Mulzer and Rote.

NP-hard

Problems solvable in polynomial time

Algorithms for the same problem may have different complexities.

Ex: Merge sort has $\Theta(n \log n)$ complexity.

Bubble sort has $\Theta(n^2)$ complexity.

Quick sort has $\Theta(n^2)$ complexity but $O(n \log n)$ average-case complexity.

Problems solvable in polynomial time

Algorithms for the same problem may have different complexities.

Ex: Merge sort has $\Theta(n \log n)$ complexity.

Bubble sort has $\Theta(n^2)$ complexity.

Quick sort has $\Theta(n^2)$ complexity but $O(n \log n)$ average-case complexity.

This can have a drastic impact.

http://www.sorting-algorithms.com/

Wrap-up: what is it about?

Algorithmic solutions to geometric problems.

Proofs of correctness and complexity bounds.

Beware of undecidable or NP-hard problems.
Asymptotic complexity matters in practice.

(Attention to degeneracy and numerical issues.)

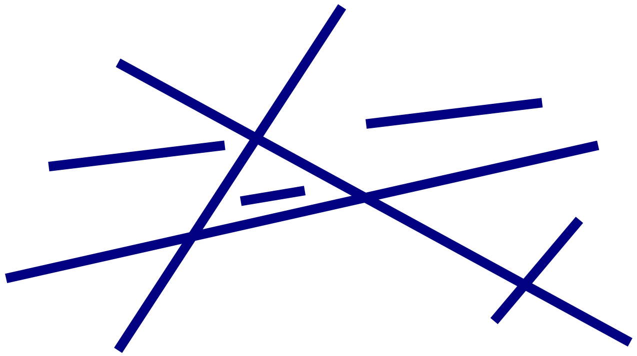
Question 2 (getting started)

How to compute the intersections among n segments in 2D?

Question 2 (getting started)

How to compute the intersections among n segments in 2D?

Input:

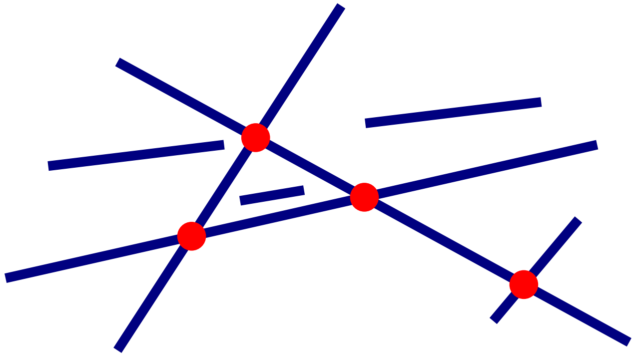


Question 2 (getting started)

How to compute the intersections among n segments in 2D?

Input:

Output:

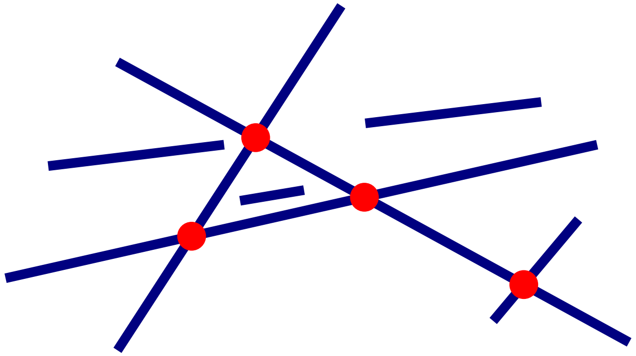


Question 2 (getting started)

How to compute the intersections among n segments in 2D?

Input:

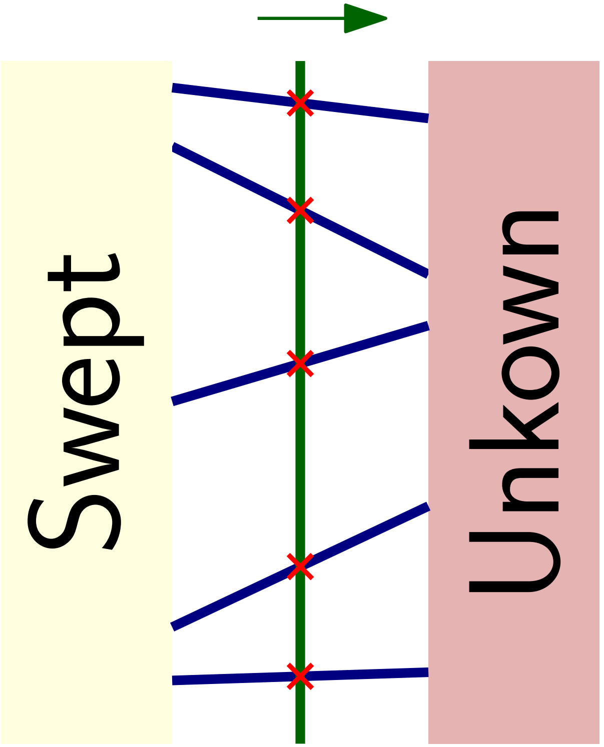
Output:



Any idea?

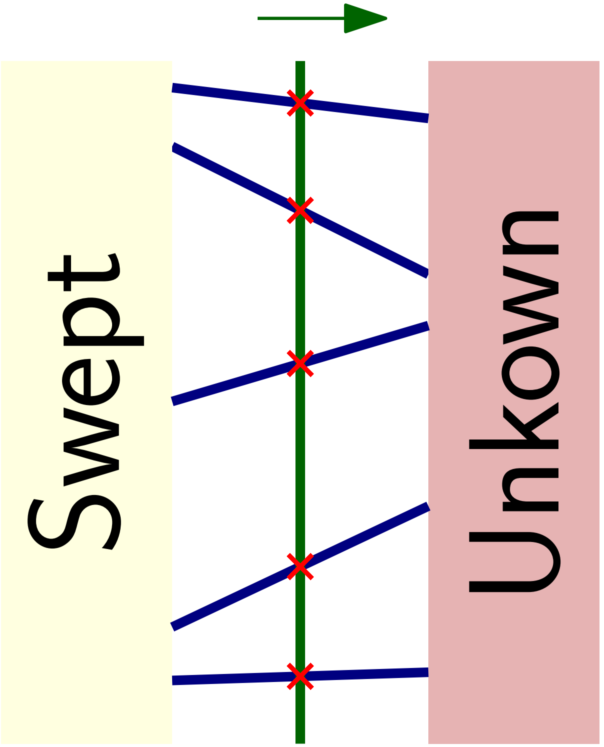
Use geometry to avoid testing unnecessary pairs.

Idea: sweep the plane using a line.



Use geometry to avoid testing unnecessary pairs.

Idea: sweep the plane using a line.

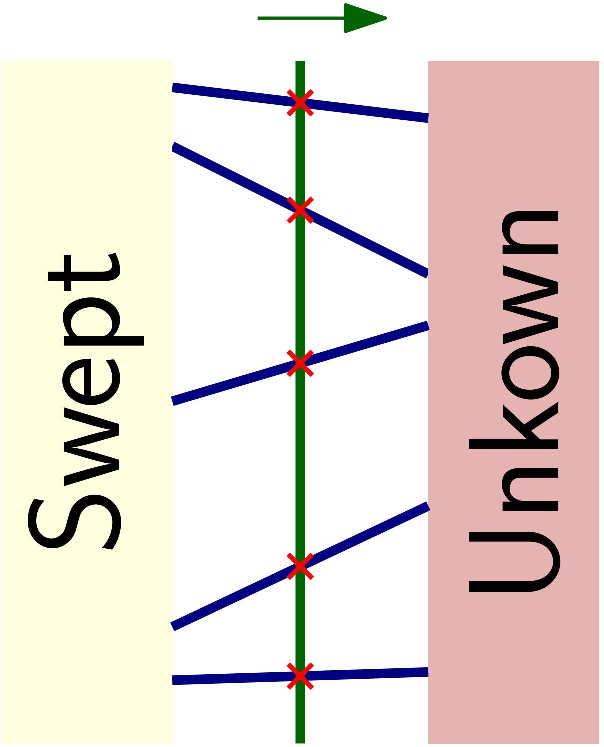


Principle:

Two segments that intersect
must meet the sweep line consecutively
before it reaches the intersection point.

Use geometry to avoid testing unnecessary pairs.

Idea: sweep the plane using a line.



Principle:

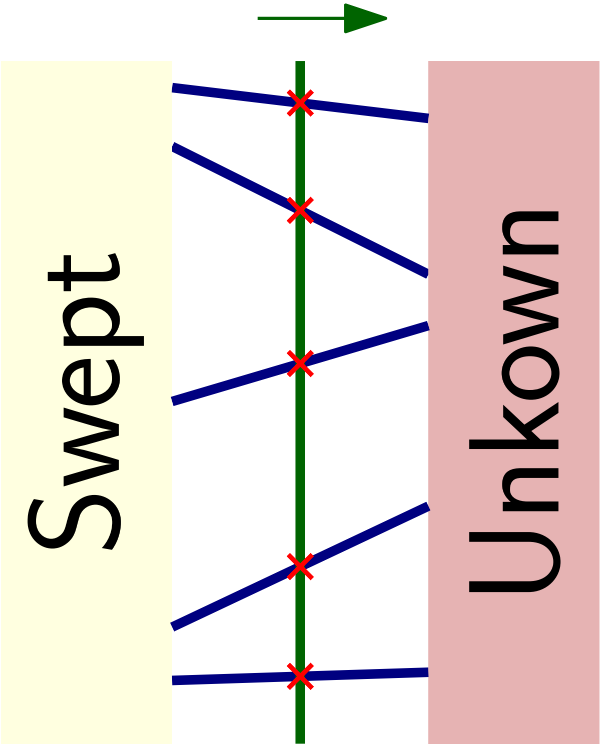
Two segments that intersect
must meet the sweep line consecutively
before it reaches the intersection point.

Idea of algorithm:

maintain the ordered list of
segments intersecting the sweep line.

Use geometry to avoid testing unnecessary pairs.

Idea: sweep the plane using a line.



Principle:

Two segments that intersect
must meet the sweep line consecutively
before it reaches the intersection point.

Idea of algorithm:

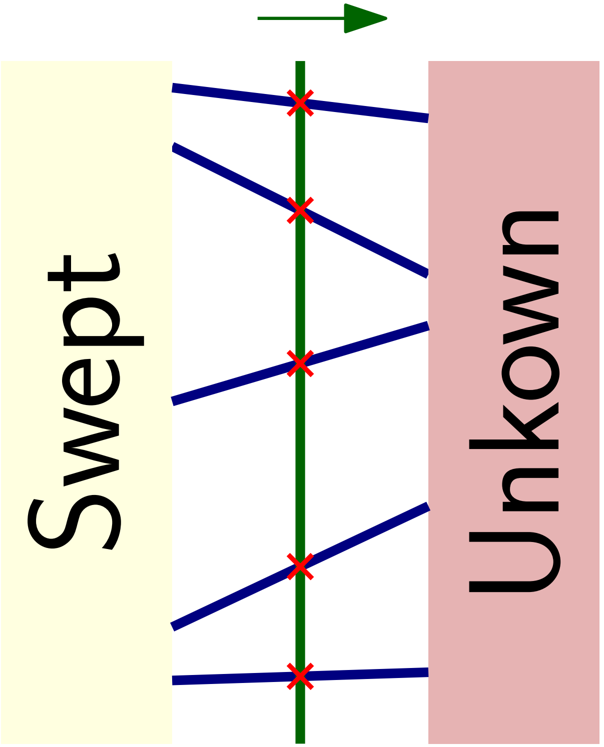
maintain the ordered list of
segments intersecting the sweep line.

Details:

How to detect the changes in the ordered list?
Data structure? Predicates?

Use geometry to avoid testing unnecessary pairs.

Idea: sweep the plane using a line.



Principle:

Two segments that intersect
must meet the sweep line consecutively
before it reaches the intersection point.

Idea of algorithm:

maintain the ordered list of
segments intersecting the sweep line.

Details:

How to detect the changes in the ordered list?
Data structure? Predicates?

Details:

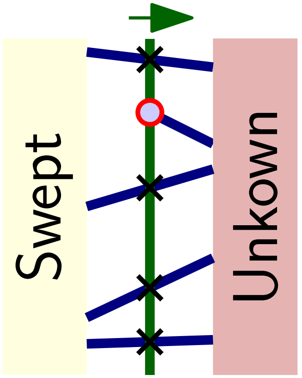
How to detect the changes in the ordered list?

Data structure? Predicates?

Three types of events

each event happens

at a particular x-coordinate



Details:

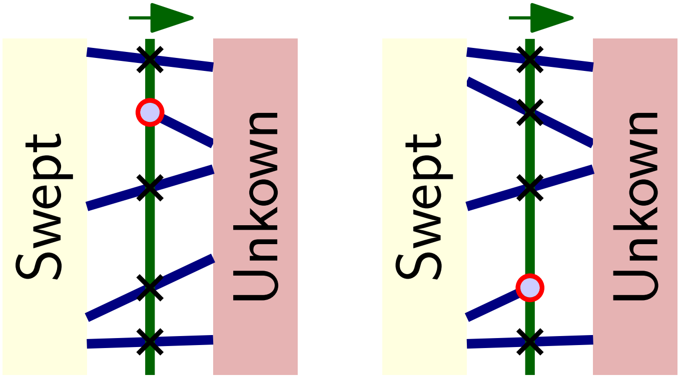
How to detect the changes in the ordered list?

Data structure? Predicates?

Three types of events

each event happens

at a particular x-coordinate



Details:

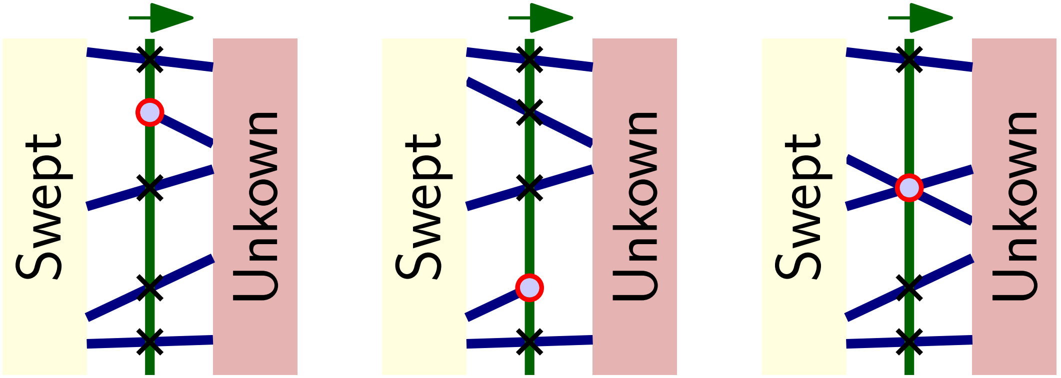
How to detect the changes in the ordered list?

Data structure? Predicates?

Three types of events

each event happens

at a particular x-coordinate



Details:

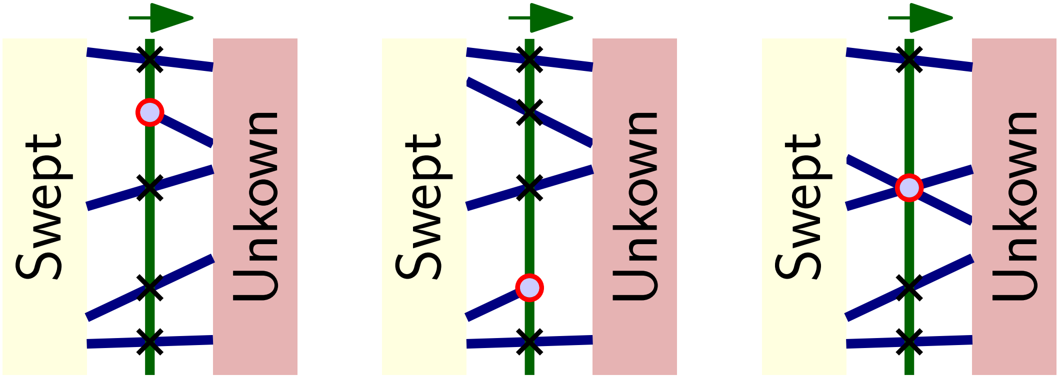
How to detect the changes in the ordered list?

Data structure? Predicates?

Three types of events

each event happens

at a particular x-coordinate

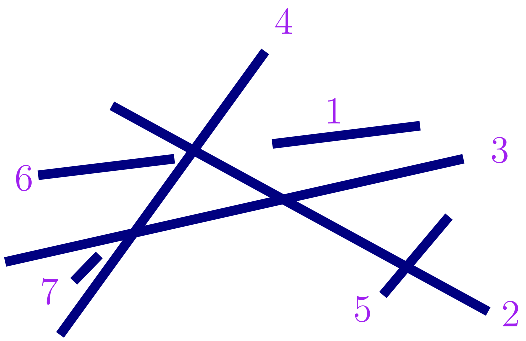


Data structures

Ordered list of segments intersected by the line.
Supports efficient insertion, deletion & exchange.

List of events sorted by x-coordinates.
Supports efficient insertion & deletion.

Algorithm

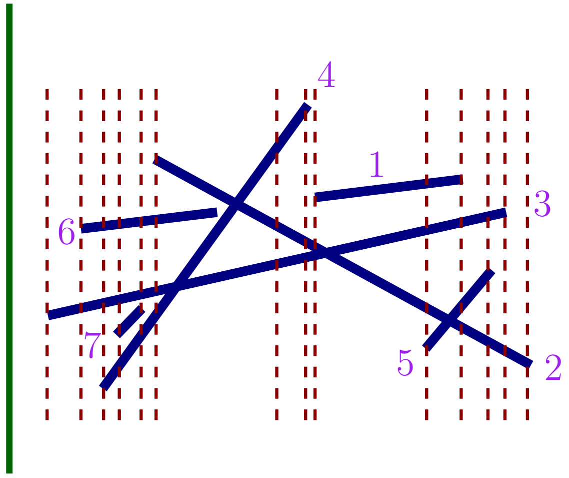


```
Insert the endpoints of all segments in Events.
Sweep $\leftarrow \emptyset$.
While Events $\neq \emptyset$
    Read the next event and remove it from the list.
        Insert, delete or swap segments in Sweep.
        Check intersections between new neighbors in Sweep.
        Add those intersections to the output and to Events.
```

Events: sorted list of events.

Sweep: sorted list of segments intersecting the sweep line.

Algorithm



```
Insert the endpoints of all segments in Events.
Sweep $\leftarrow \emptyset$.
While Events $\neq \emptyset$
    Read the next event and remove it from the list.
        Insert, delete or swap segments in Sweep.
        Check intersections between new neighbors in Sweep.
        Add those intersections to the output and to Events.
```

Events: sorted list of events.

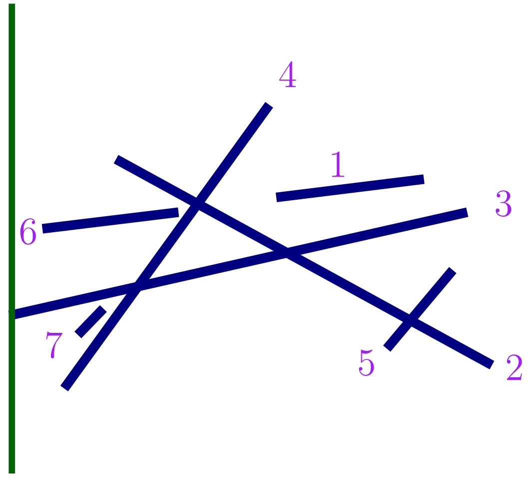
Sweep: sorted list of segments intersecting the sweep line.

Events$= \{L_3, L_6, L_4, L_7, R_7, L_2, R_6, R_4, L_1, L_5, R_1, R_5, R_3, R_2\}$

Sweep$= \{\}$

Output$= \{\}$

Algorithm



```
Insert the endpoints of all segments in Events.
Sweep $\leftarrow \emptyset$.
While Events $\neq \emptyset$
    Read the next event and remove it from the list.
        Insert, delete or swap segments in Sweep.
        Check intersections between new neighbors in Sweep.
        Add those intersections to the output and to Events.
```

Events: sorted list of events.

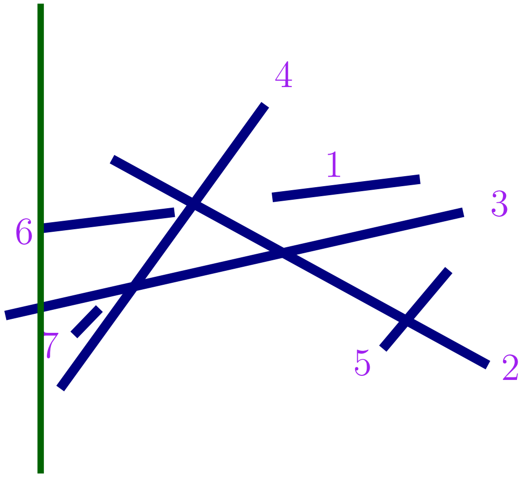
Sweep: sorted list of segments intersecting the sweep line.

Events$= \{L_3, L_6, L_4, L_7, R_7, L_2, R_6, R_4, L_1, L_5, R_1, R_5, R_3, R_2\}$

Sweep$= \{3\}$

Output$= \{\}$

Algorithm



```
Insert the endpoints of all segments in Events.
Sweep $\leftarrow \emptyset$.
While Events $\neq \emptyset$
    Read the next event and remove it from the list.
        Insert, delete or swap segments in Sweep.
        Check intersections between new neighbors in Sweep.
        Add those intersections to the output and to Events.
```

Events: sorted list of events.

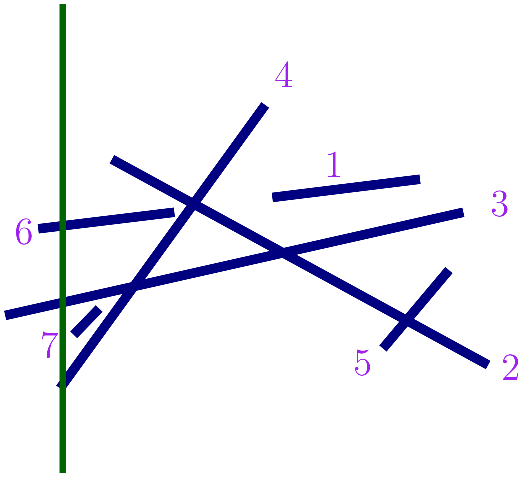
Sweep: sorted list of segments intersecting the sweep line.

Events$= \{L_6, L_4, L_7, R_7, L_2, R_6, R_4, L_1, L_5, R_1, R_5, R_3, R_2\}$

Sweep$= \{6, 3\}$

Output$= \{\}$

Algorithm



```
Insert the endpoints of all segments in Events.
Sweep $\leftarrow \emptyset$.
While Events $\neq \emptyset$
    Read the next event and remove it from the list.
        Insert, delete or swap segments in Sweep.
        Check intersections between new neighbors in Sweep.
        Add those intersections to the output and to Events.
```

Events: sorted list of events.

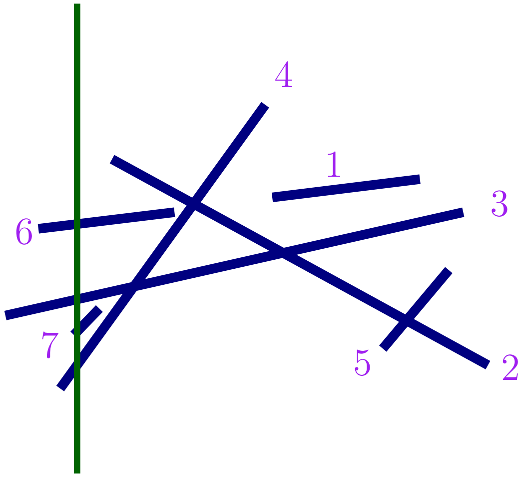
Sweep: sorted list of segments intersecting the sweep line.

Events$= \{L_4, L_7, R_7, L_2, I_{4,3}, R_6, R_4, L_1, L_5, R_1, R_5, R_3, R_2\}$

Sweep$= \{6, 3, 4\}$

Output$= \{(3, 4)\}$

Algorithm



```
Insert the endpoints of all segments in Events.
Sweep $\leftarrow \emptyset$.
While Events $\neq \emptyset$
    Read the next event and remove it from the list.
        Insert, delete or swap segments in Sweep.
        Check intersections between new neighbors in Sweep.
        Add those intersections to the output and to Events.
```

Events: sorted list of events.

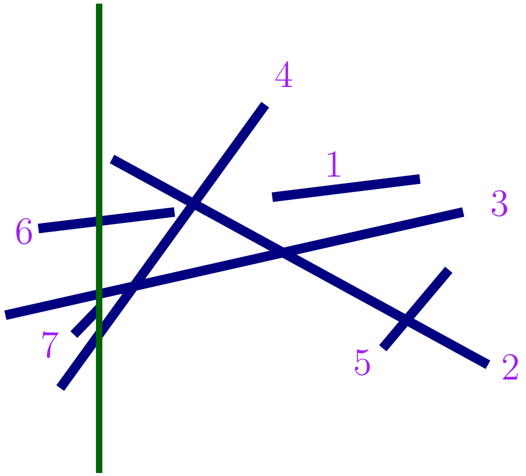
Sweep: sorted list of segments intersecting the sweep line.

Events= $\{L_7, R_7, L_2, I_{4,3}, R_6, R_4, L_1, L_5, R_1, R_5, R_3, R_2\}$

Sweep= $\{6, 3, 7, 4\}$

Output= $\{(3, 4)\}$

Algorithm



```
Insert the endpoints of all segments in Events.
Sweep ← $\emptyset$.
While Events $\neq \emptyset$
    Read the next event and remove it from the list.
        Insert, delete or swap segments in Sweep.
        Check intersections between new neighbors in Sweep.
        Add those intersections to the output and to Events.
```

Events: sorted list of events.

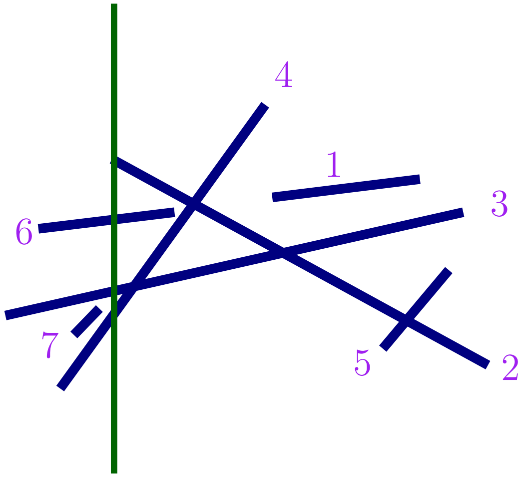
Sweep: sorted list of segments intersecting the sweep line.

Events= $\{R_7, L_2, I_{4,3}, R_6, R_4, L_1, L_5, R_1, R_5, R_3, R_2\}$

Sweep= $\{6, 3, 7, 4\}$

Output= $\{(3, 4)\}$

Algorithm



```
Insert the endpoints of all segments in Events.
Sweep $\leftarrow \emptyset$.
While Events $\neq \emptyset$
    Read the next event and remove it from the list.
        Insert, delete or swap segments in Sweep.
        Check intersections between new neighbors in Sweep.
        Add those intersections to the output and to Events.
```

Events: sorted list of events.

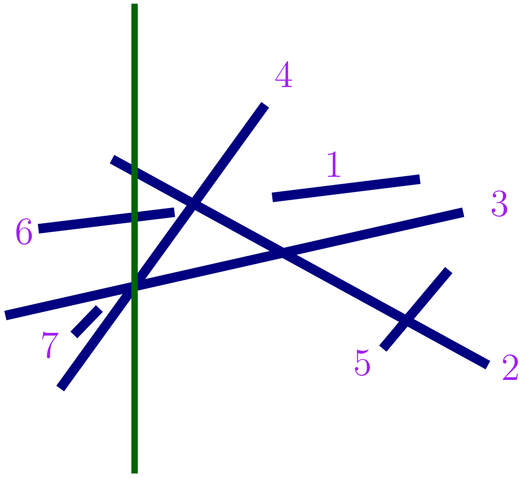
Sweep: sorted list of segments intersecting the sweep line.

Events$= \{L_2, I_{4,3}, R_6, R_4, L_1, L_5, R_1, R_5, R_3, R_2\}$

Sweep$= \{2, 6, 3, 4\}$

Output$= \{(3, 4)\}$

Algorithm



```
Insert the endpoints of all segments in Events.
Sweep $\leftarrow \emptyset$.
While Events $\neq \emptyset$
    Read the next event and remove it from the list.
        Insert, delete or swap segments in Sweep.
        Check intersections between new neighbors in Sweep.
        Add those intersections to the output and to Events.
```

Events: sorted list of events.

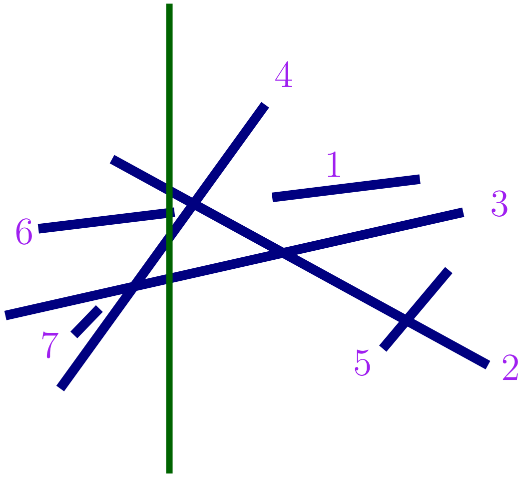
Sweep: sorted list of segments intersecting the sweep line.

Events= $\{I_{4,3}, R_6, R_4, L_1, L_5, R_1, R_5, R_3, R_2\}$

Sweep= $\{2, 6, 4, 3\}$

Output= $\{(3, 4)\}$

Algorithm



```
Insert the endpoints of all segments in Events.
Sweep ← $\emptyset$.
While Events $\neq \emptyset$
    Read the next event and remove it from the list.
        Insert, delete or swap segments in Sweep.
        Check intersections between new neighbors in Sweep.
        Add those intersections to the output and to Events.
```

Events: sorted list of events.

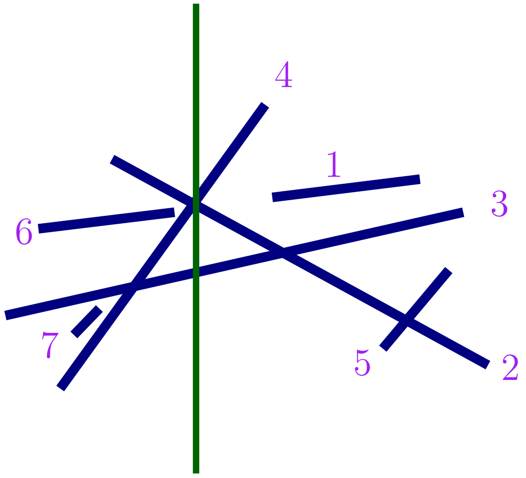
Sweep: sorted list of segments intersecting the sweep line.

Events= $\{R_6, I_{2,4}, R_4, L_1, L_5, R_1, R_5, R_3, R_2\}$

Sweep= $\{2, 6, 4, 3\}$

Output= $\{(3,4), (2,4)\}$

Algorithm



```
Insert the endpoints of all segments in Events.
Sweep $\leftarrow \emptyset$.
While Events $\neq \emptyset$
    Read the next event and remove it from the list.
        Insert, delete or swap segments in Sweep.
        Check intersections between new neighbors in Sweep.
        Add those intersections to the output and to Events.
```

Events: sorted list of events.

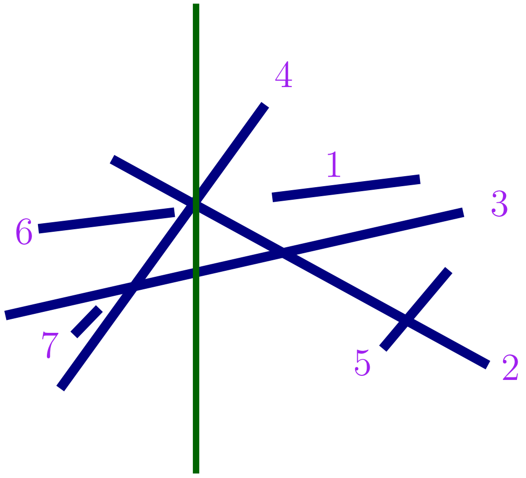
Sweep: sorted list of segments intersecting the sweep line.

Events$= \{I_{2,4}, R_4, L_1, I_{2,3}, L_5, R_1, R_5, R_3, R_2\}$

Sweep$= \{4, 2, 3\}$

Output$= \{(3,4), (2,4), (2,3)\}$

Algorithm



```
Insert the endpoints of all segments in Events.
Sweep $\leftarrow \emptyset$.
While Events $\neq \emptyset$
    Read the next event and remove it from the list.
        Insert, delete or swap segments in Sweep.
        Check intersections between new neighbors in Sweep.
        Add those intersections to the output and to Events.
```

Events: sorted list of events.

Sweep: sorted list of segments intersecting the sweep line.

Events= $\{I_{2,4}, R_4, L_1, I_{2,3}, L_5, R_1, R_5, R_3, R_2\}$

Sweep= $\{4, 2, 3\}$

Output= $\{(3,4), (2,4), (2,3)\}$

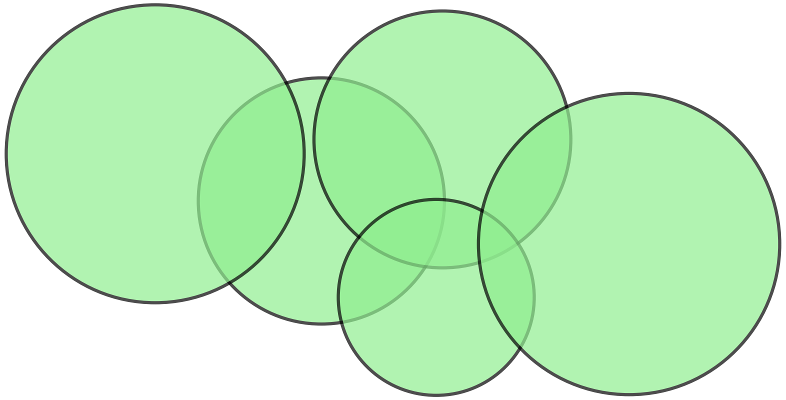
etc...

Correctness? Complexity?

Wrap-up: sweep algorithms

Generic principle, three predicates: x-extreme points, intersection, x-coordinate comparison.

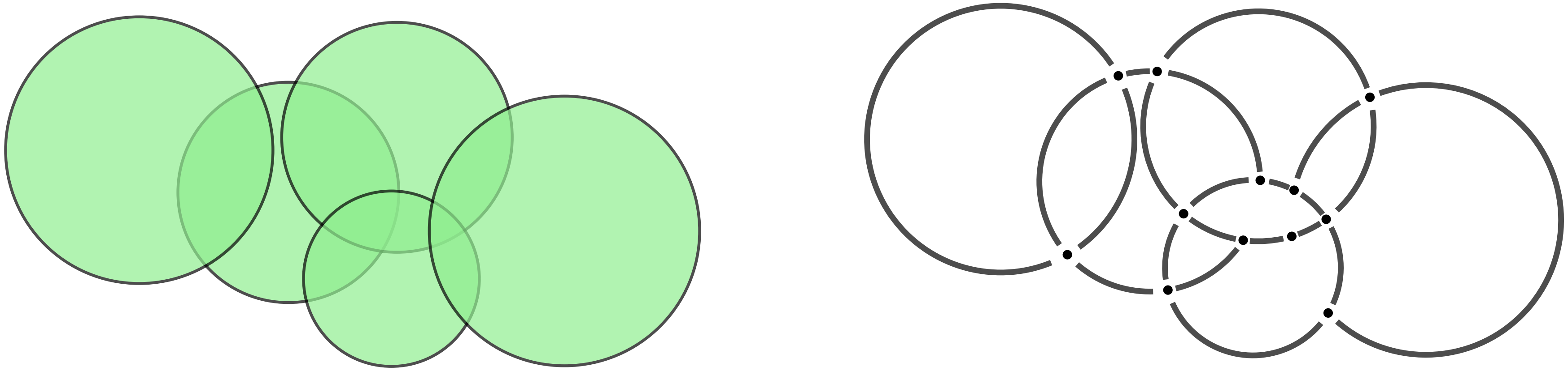
Other objects (polygons, circles, algebraic curves, etc...), other spaces ($\mathbb{S}^2$, $\mathbb{R}^3$, $\mathbb{S}^1 \times \mathbb{S}^1$...).



Wrap-up: sweep algorithms

Generic principle, three predicates: x-extreme points, intersection, x-coordinate comparison.

Other objects (polygons, circles, algebraic curves, etc...), other spaces ($\mathbb{S}^2$, $\mathbb{R}^3$, $\mathbb{S}^1 \times \mathbb{S}^1$...).

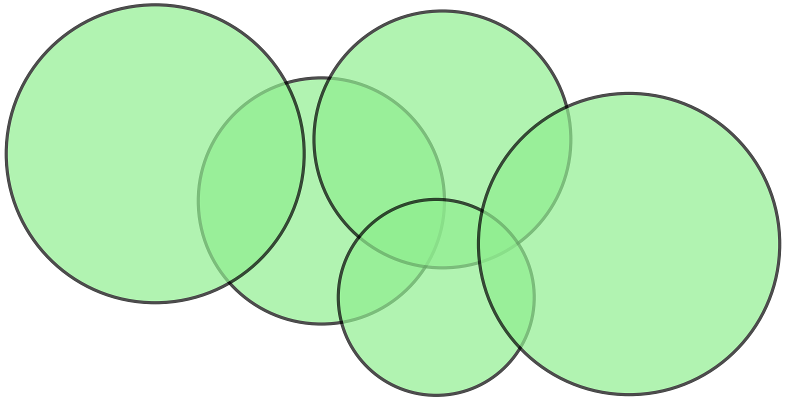


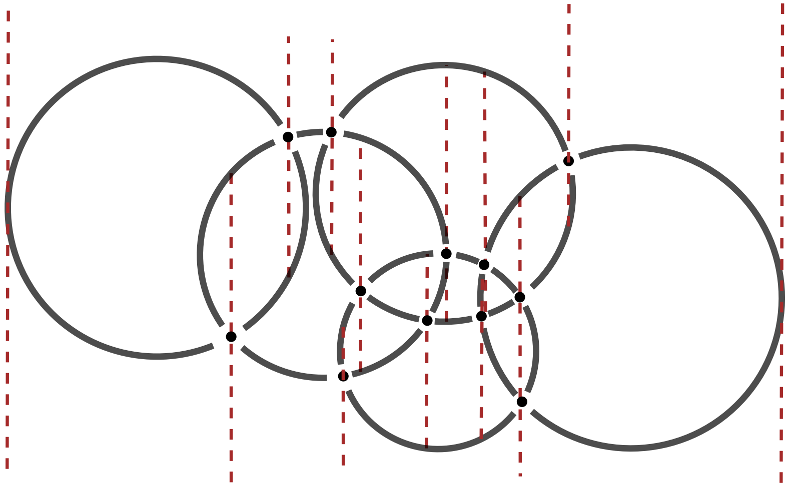
Computing arrangements of geometric objects.

Wrap-up: sweep algorithms

Generic principle, three predicates: x-extreme points, intersection, x-coordinate comparison.

Other objects (polygons, circles, algebraic curves, etc...), other spaces ($\mathbb{S}^2$, $\mathbb{R}^3$, $\mathbb{S}^1 \times \mathbb{S}^1$...).





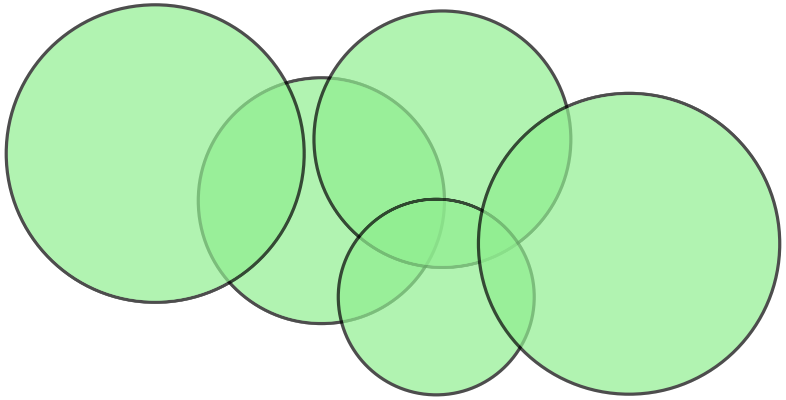
Computing arrangements of geometric objects.

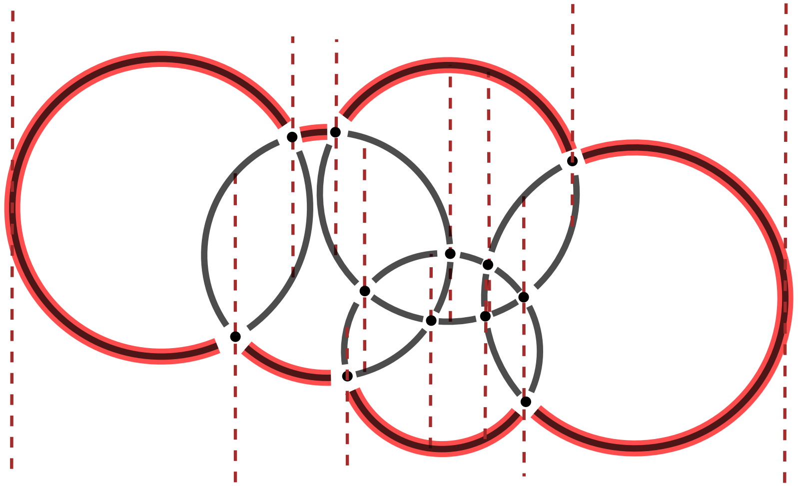
Computing trapezoidal decompositions of arrangements of geometric objects.

Wrap-up: sweep algorithms

Generic principle, three predicates: x-extreme points, intersection, x-coordinate comparison.

Other objects (polygons, circles, algebraic curves, etc...), other spaces ($\mathbb{S}^2$, $\mathbb{R}^3$, $\mathbb{S}^1 \times \mathbb{S}^1$...).





Computing arrangements of geometric objects.

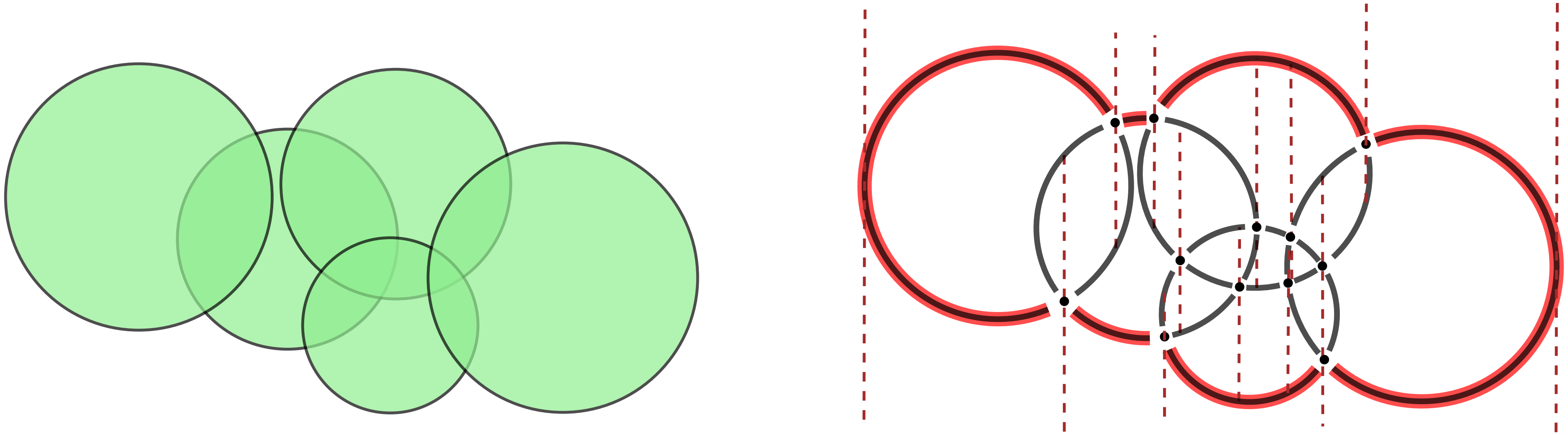
Computing trapezoidal decompositions of arrangements of geometric objects.

Computing substructures of arrangements of geometric objects.

Wrap-up: sweep algorithms

Generic principle, three predicates: x-extreme points, intersection, x-coordinate comparison.

Other objects (polygons, circles, algebraic curves, etc...), other spaces ($\mathbb{S}^2$, $\mathbb{R}^3$, $\mathbb{S}^1 \times \mathbb{S}^1$...).



Computing arrangements of geometric objects.

Computing trapezoidal decompositions of arrangements of geometric objects.

Computing substructures of arrangements of geometric objects.

All that in $O((n+k)\log n)$.

Question 3

Why are geometric algorithms hard to implement correctly?

First issue: degeneracies

Many algorithms are described assuming general position of the input.

$$\left.\begin{array}{l}\text{No two points have the same } x\text{-coordinate.}\\ \text{No three segments intersect in the same point.}\\ \text{No four points lie on the same circle.}\end{array}\right\} \quad \text{properties that hold generically.}$$

First issue: degeneracies

Many algorithms are described assuming general position of the input.

$$\left.\begin{array}{l}\text{No two points have the same } x\text{-coordinate.}\\ \text{No three segments intersect in the same point.}\\ \text{No four points lie on the same circle.}\end{array}\right\}\quad \text{properties that hold generically.}$$

A property that is true only for a subset of measure 0 of the space of inputs is a degeneracy.

First issue: degeneracies

Many algorithms are described assuming general position of the input.

No two points have the same x-coordinate.
No three segments intersect in the same point.
No four points lie on the same circle.

} properties that hold generically.

A property that is true only for a subset of measure 0 of the space of inputs is a degeneracy.

Degeneracy are common. They are often there by design.

Objects in contact are tangent.

Try asking an architect to avoid quadruple of coplanar points when designing a CAD model.

First issue: degeneracies

Many algorithms are described assuming general position of the input.

No two points have the same x-coordinate.
No three segments intersect in the same point.
No four points lie on the same circle.

} properties that hold generically.

A property that is true only for a subset of measure 0 of the space of inputs is a degeneracy.

Degeneracy are common. They are often there by design.

Objects in contact are tangent.
Try asking an architect to avoid quadruple of coplanar points when designing a CAD model.

Some degeneracies come from the problem, others from the algorithm.

First issue: degeneracies

Many algorithms are described assuming general position of the input.

No two points have the same x-coordinate.
No three segments intersect in the same point.
No four points lie on the same circle.

} properties that hold generically.

A property that is true only for a subset of measure 0 of the space of inputs is a degeneracy.

Degeneracy are common. They are often there by design.

Objects in contact are tangent.
Try asking an architect to avoid quadruple of coplanar points when designing a CAD model.

Some degeneracies come from the problem, others from the algorithm.

Can we handle degeneracies without treating each one separately?

Can we at least detect them efficiently?

Hardness of testing degeneracies

The 3**-sum problem**: Given n numbers, decide if three of them sum to 0.

What is the best algorithm you can come-up with?

Hardness of testing degeneracies

The 3-sum problem: Given n numbers, decide if three of them sum to 0.

What is the best algorithm you can come-up with?

Known bounds: $O(n^2)$ and $\Omega(n \log n)$.

Hardness of testing degeneracies

The 3-sum problem: Given n numbers, decide if three of them sum to 0.

What is the best algorithm you can come-up with?

Known bounds: $O(n^2)$ and $\Omega(n \log n)$.

15 years old conjecture: any algorithm solving 3-sum has $\Omega(n^2)$ time complexity.

Hardness of testing degeneracies

The 3-sum problem: Given n numbers, decide if three of them sum to 0.

What is the best algorithm you can come-up with?

Known bounds: $O(n^2)$ and $\Omega(n \log n)$.

15 years old conjecture: any algorithm solving 3-sum has $\Omega(n^2)$ time complexity.

If we can detect triples of aligned 2D points in $o(n^2)$ time then we can solve 3-sum in $o(n^2)$ time.

Hardness of testing degeneracies

The 3-sum problem: Given n numbers, decide if three of them sum to 0.

What is the best algorithm you can come-up with?

Known bounds: $O(n^2)$ and $\Omega(n \log n)$.

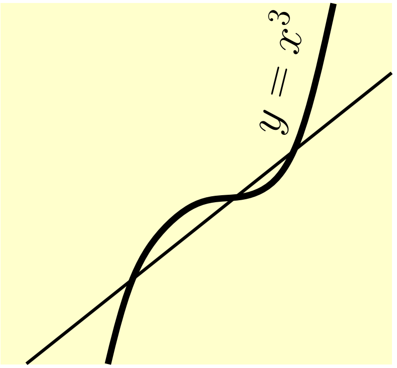
15 years old conjecture: any algorithm solving 3-sum has $\Omega(n^2)$ time complexity.

If we can detect triples of aligned 2D points in $o(n^2)$ time then we can solve 3-sum in $o(n^2)$ time.

numbers $t_1, \ldots, t_n \rightarrow$ points $p_1, \ldots, p_n$ with $p_i = (t_i, t_i^3)$.

$t_i + t_j + t_k = 0 \Leftrightarrow p_i, p_j, p_k$ are aligned.

p, q and r are aligned $\Leftrightarrow \begin{vmatrix} x_p & x_q & x_r \\ y_p & y_q & y_r \\ 1 & 1 & 1 \end{vmatrix} = 0.$



$$\begin{vmatrix} t_i & t_j & t_k \\ t_i^3 & t_j^3 & t_k^3 \\ 1 & 1 & 1 \end{vmatrix} = (t_j - t_i)(t_k - t_i)(t_k - t_j)(t_i + t_j + t_k).$$

Hardness of testing degeneracies

The 3-sum problem: Given n numbers, decide if three of them sum to 0.

What is the best algorithm you can come-up with?

Known bounds: $O(n^2)$ and $\Omega(n \log n)$.

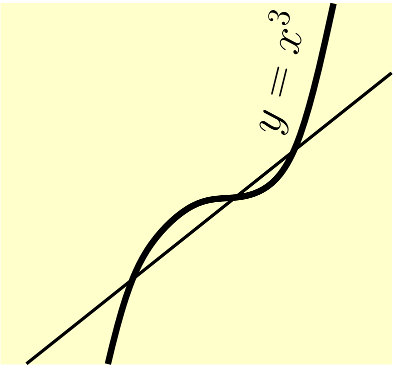
15 years old conjecture: any algorithm solving 3-sum has $\Omega(n^2)$ time complexity.

If we can detect triples of aligned 2D points in $o(n^2)$ time then we can solve 3-sum in $o(n^2)$ time.

numbers $t_1, \ldots, t_n \to$ points $p_1, \ldots, p_n$ with $p_i = (t_i, t_i^3)$.

$t_i + t_j + t_k = 0 \Leftrightarrow p_i, p_j, p_k$ are aligned.

p, q and r are aligned $\Leftrightarrow \begin{vmatrix} x_p & x_q & x_r \\ y_p & y_q & y_r \\ 1 & 1 & 1 \end{vmatrix} = 0.$



$$\begin{vmatrix} t_i & t_j & t_k \\ t_i^3 & t_j^3 & t_k^3 \\ 1 & 1 & 1 \end{vmatrix} = (t_j - t_i)(t_k - t_i)(t_k - t_j)(t_i + t_j + t_k).$$

Testing if $d+1$ points lie on a common hyperplane in $\mathbb{R}^d$ is $\lceil \frac{d}{2} \rceil$-sum hard.

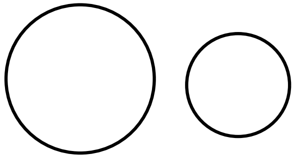
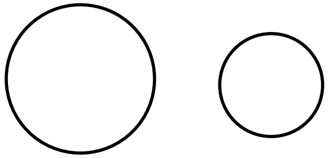
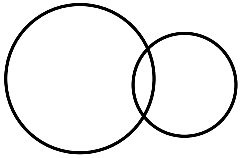
Perturbing? Easier said than done

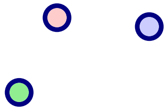
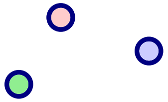
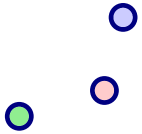
In principle, perturbing the points eliminate degeneracies.

Perturbing? Easier said than done

In principle, perturbing the points eliminate degeneracies.

First issue: the perturbation should preserves non-degenerate inputs.

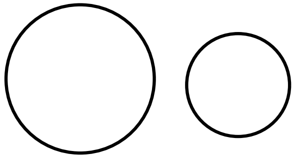

can be perturbed in

but not in



can be perturbed in

but not in


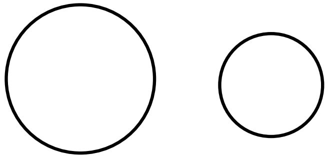
Perturbing? Easier said than done

In principle, perturbing the points eliminate degeneracies.

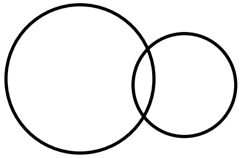
First issue: the perturbation should preserves non-degenerate inputs.

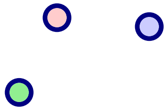


can be perturbed in

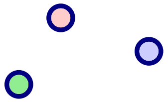


but not in

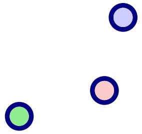




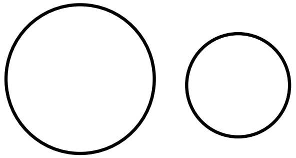
can be perturbed in



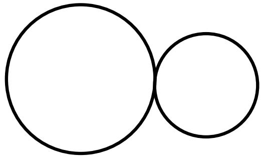
but not in

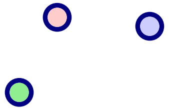


Second issue: the perturbation should not create new degeneracies.

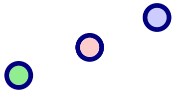


should not be perturbed in





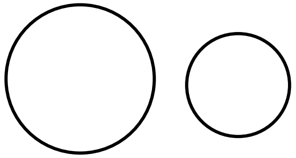
should not be perturbed in



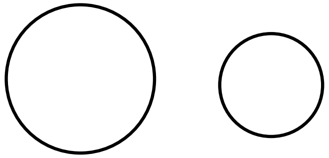
Perturbing? Easier said than done

In principle, perturbing the points eliminate degeneracies.

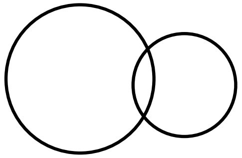
First issue: the perturbation should preserves non-degenerate inputs.

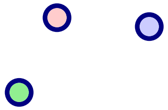


can be perturbed in

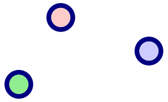


but not in

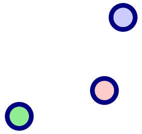




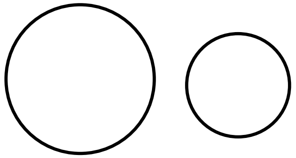
can be perturbed in



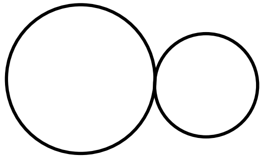
but not in

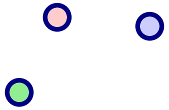


Second issue: the perturbation should not create new degeneracies.

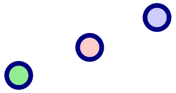


should not be perturbed in





should not be perturbed in



Bottom line: "Epsilon=10^-12" is not an option if we want any kind of guarantee.

Systematic approach: simulation of simplicity

Degeneracy correspond to vanishing of some of the polynomials evaluating the geometric predicates.

Systematic approach: simulation of simplicity

Degeneracy correspond to vanishing of some of the polynomials evaluating the geometric predicates.

Consider a geometric object as a function of one variable t [1990].
The input we are interested in is the value when $t = 0$.
Ex: the point $p = (3, 12)$ becomes $p = (3 + t, 12 + t^2)$.

Make all computations for "$t > 0$ sufficiently small" then take $\lim_{t\to 0}$.

Systematic approach: simulation of simplicity

Degeneracy correspond to vanishing of some of the polynomials evaluating the geometric predicates.

Consider a geometric object as a function of one variable t [1990].
The input we are interested in is the value when $t = 0$.

Ex: the point $p = (3, 12)$ becomes $p = (3 + t, 12 + t^2)$.

Make all computations for ”$t > 0$ sufficiently small” then take $\lim_{t \to 0}$.

Choose the functions so that the relevant polynomials do not identically vanish.

Example: convex hull computation, point-in-polygon.

Predicates are x*-coordinates comparison* and *orientation*.

Replacing p_i by $p_i + (t^{2^{2i}}, t^{2^{2i+1}})$ handles all degeneracies for these predicates.

Systematic approach: simulation of simplicity

Degeneracy correspond to vanishing of some of the polynomials evaluating the geometric predicates.

Consider a geometric object as a function of one variable t [1990].
The input we are interested in is the value when $t = 0$.

Ex: the point $p = (3, 12)$ becomes $p = (3 + t, 12 + t^2)$.

Make all computations for ”$t > 0$ sufficiently small” then take $\lim_{t\to 0}$.

Choose the functions so that the relevant polynomials do not identically vanish.

Example: convex hull computation, point-in-polygon.

Predicates are x*-coordinates comparison* and *orientation*.

Replacing p_i by $p_i + (t^{2^{2i}}, t^{2^{2i+1}})$ handles all degeneracies for these predicates.

Heavy machinery, important slow-down, ignore voluntary degeneracies.

Systematic approach: simulation of simplicity

Degeneracy correspond to vanishing of some of the polynomials evaluating the geometric predicates.

Consider a geometric object as a function of one variable t [1990].
The input we are interested in is the value when $t = 0$.

Ex: the point $p = (3, 12)$ becomes $p = (3 + t, 12 + t^2)$.

Make all computations for "$t > 0$ sufficiently small" then take $\lim_{t \to 0}$.

Choose the functions so that the relevant polynomials do not identically vanish.
Example: convex hull computation, point-in-polygon.

Predicates are x*-coordinates comparison* and *orientation*.

Replacing p_i by $p_i + (t^{2^{2i}}, t^{2^{2i+1}})$ handles all degeneracies for these predicates.

Heavy machinery, important slow-down, ignore voluntary degeneracies.

Partial perturbation: shearing $(x, y) \mapsto (x + ty, y)$

Second issue: numerical rounding

The arithmetic on a computer uses bounded precision (32 bits, 64 bits, IEEE float norms, etc...).

Small errors will be made in computations.

Not to mention that processors make errors from time to time...

Second issue: numerical rounding

The arithmetic on a computer uses bounded precision (32 bits, 64 bits, IEEE float norms, etc...).

Small errors will be made in computations.

Not to mention that processors make errors from time to time...

The question is: **can these error have a significant impact?**

Second issue: numerical rounding

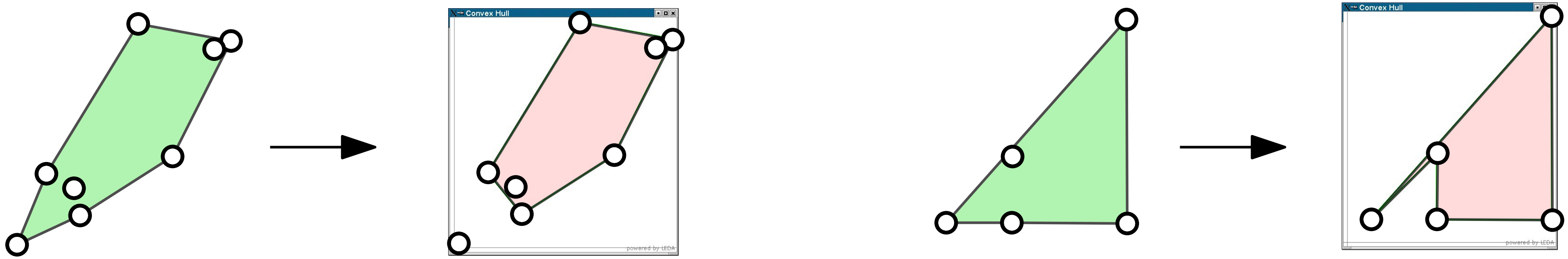
The arithmetic on a computer uses bounded precision (32 bits, 64 bits, IEEE float norms, etc...).

Small errors will be made in computations.

Not to mention that processors make errors from time to time...

The question is: **can these error have a significant impact?**

"Judge for yourself": the example of 2D convex hull computation.



Second issue: numerical rounding

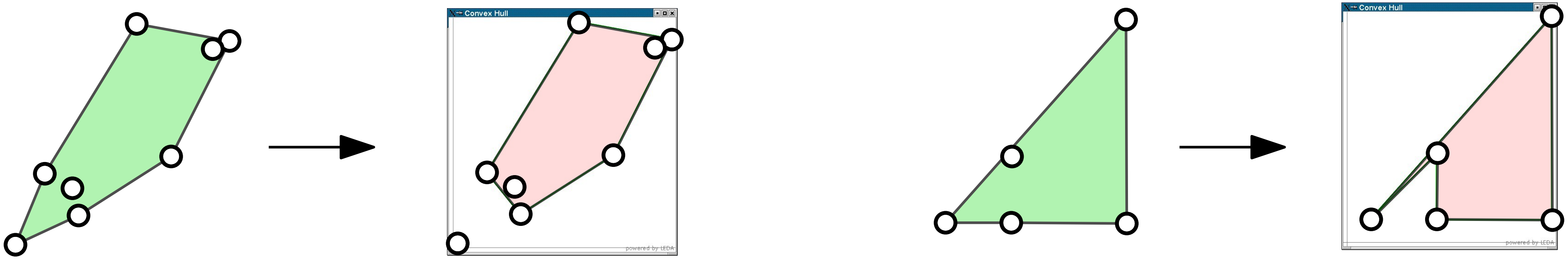
The arithmetic on a computer uses bounded precision (32 bits, 64 bits, IEEE float norms, etc...).

Small errors will be made in computations.

Not to mention that processors make errors from time to time...

The question is: **can these error have a significant impact?**

"Judge for yourself": the example of 2D convex hull computation.

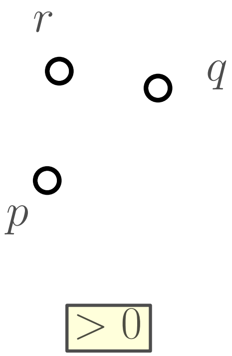


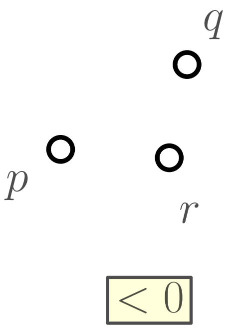
The problem: three points are nearly aligned, and the orientation predicates make inconsistent errors.

"Sometimes left, sometimes right".

A close look at that example

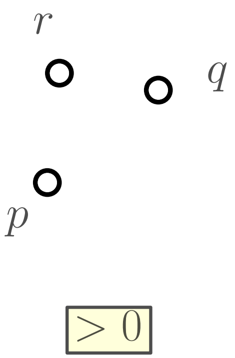
Orientation of (p, q, r) given by the sign of $\begin{vmatrix} x_p & x_q & x_r \\ y_p & y_q & y_r \\ 1 & 1 & 1 \end{vmatrix}$.

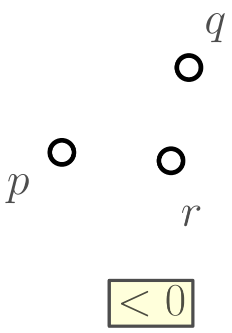




A close look at that example

Orientation of (p, q, r) given by the sign of $\begin{vmatrix} x_p & x_q & x_r \\ y_p & y_q & y_r \\ 1 & 1 & 1 \end{vmatrix}$.

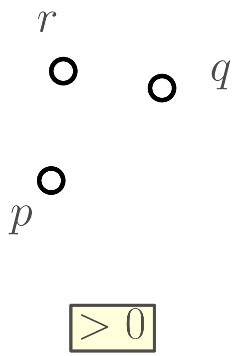


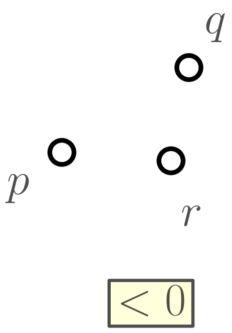


```
Float xp,yp,xq,yq,xr,yr;

Orientation = sign((xq-xp)*(yr-yp)-(xr-xp)*(yq-yp));
```

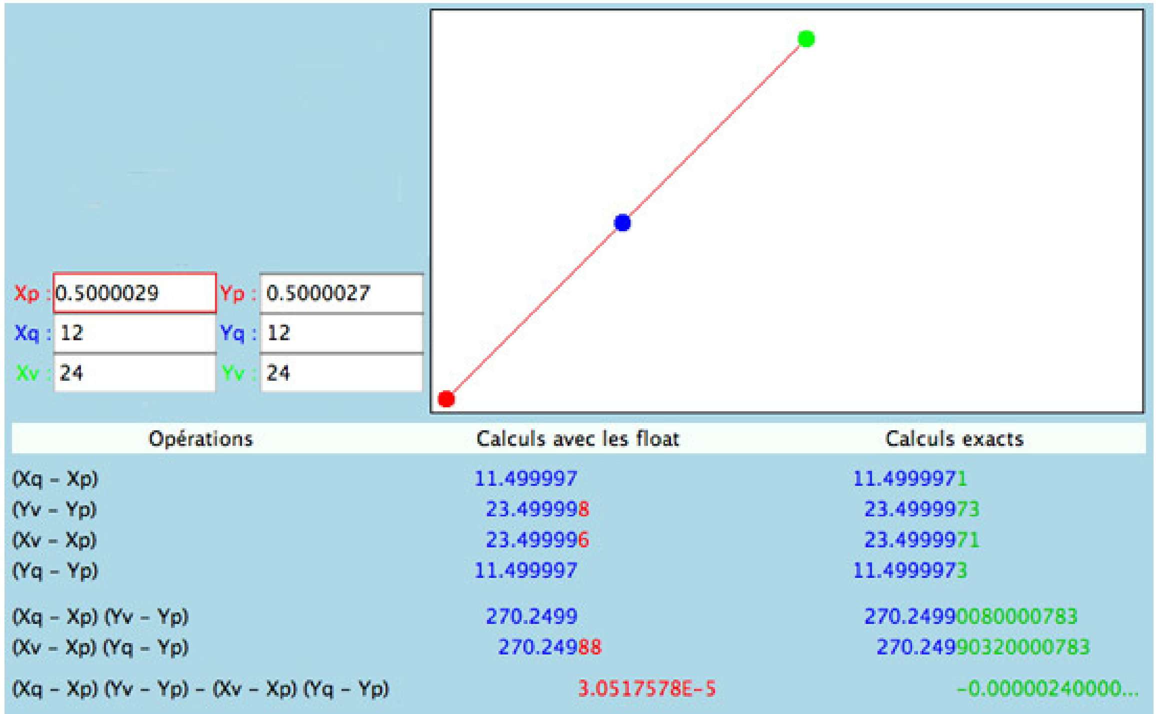
A close look at that example

Orientation of (p, q, r) given by the sign of $\begin{vmatrix} x_p & x_q & x_r \\ y_p & y_q & y_r \\ 1 & 1 & 1 \end{vmatrix}$.





```
Float xp,yp,xq,yq,xr,yr;

Orientation = sign((xq-xp)*(yr-yp)-(xr-xp)*(yq-yp));
```

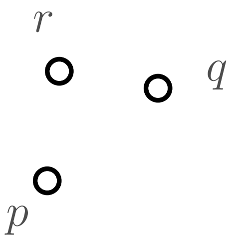


Xp : 0.5000029
Yp : 0.5000027
Xq : 12
Yq : 12
Xv : 24
Yv : 24

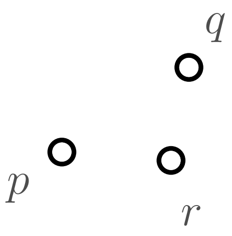
Opérations	Calculs avec les float	Calculs exacts
(Xq - Xp)	11.499997	11.4999971
(Yv - Yp)	23.499998	23.4999973
(Xv - Xp)	23.499996	23.4999971
(Yq - Yp)	11.499997	11.4999973
(Xq - Xp) (Yv - Yp)	270.2499	270.24990080000783
(Xv - Xp) (Yq - Yp)	270.24988	270.24990320000783
(Xq - Xp) (Yv - Yp) - (Xv - Xp) (Yq - Yp)	3.0517578E-5	-0.00000240000...

A close look at that example

Orientation of (p, q, r) given by the sign of $\begin{vmatrix} x_p & x_q & x_r \\ y_p & y_q & y_r \\ 1 & 1 & 1 \end{vmatrix}$.

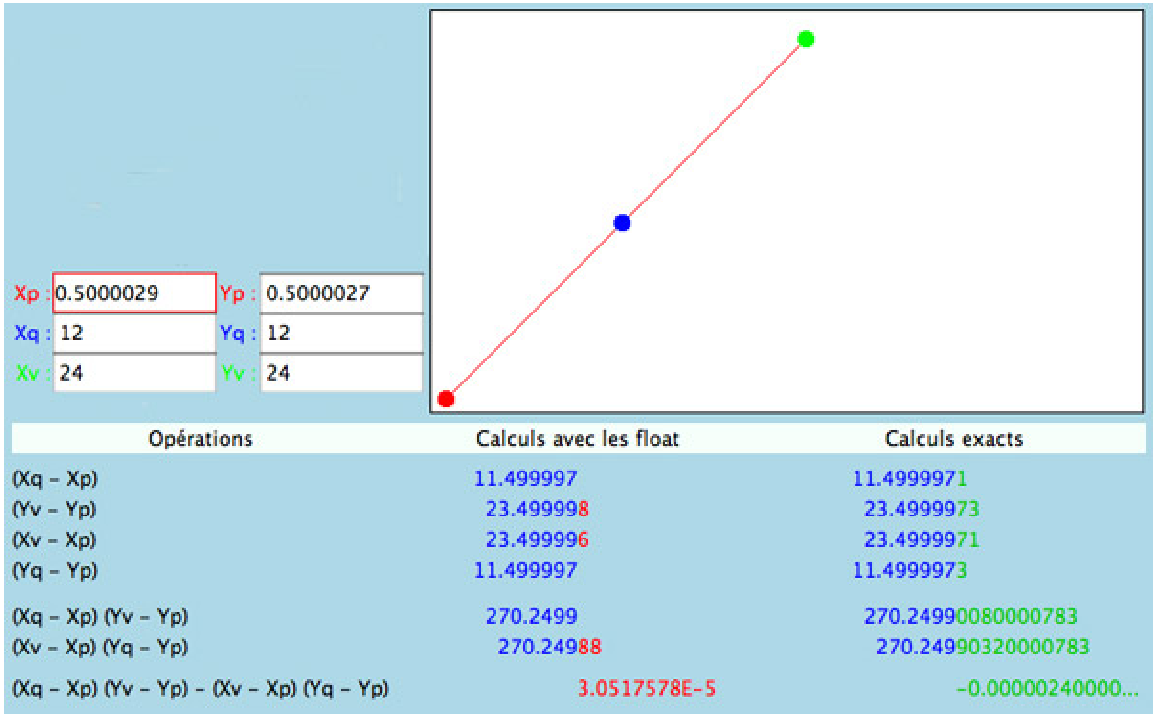


$\boxed{> 0}$



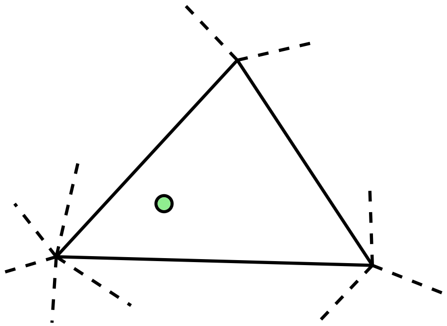
$\boxed{< 0}$

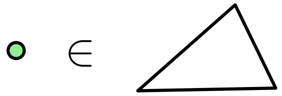
```
Float xp,yp,xq,yq,xr,yr;

Orientation = sign((xq-xp)*(yr-yp)-(xr-xp)*(yq-yp));
```



Xp : 0.5000029
Yp : 0.5000027
Xq : 12
Yq : 12
Xv : 24
Yv : 24

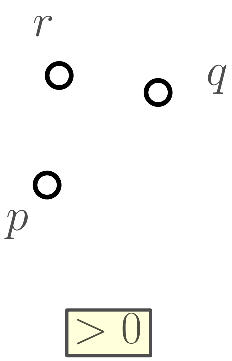
Opérations	Calculs avec les float	Calculs exacts
(Xq - Xp)	11.499997	11.4999971
(Yv - Yp)	23.499998	23.4999973
(Xv - Xp)	23.499996	23.4999971
(Yq - Yp)	11.499997	11.4999973
(Xq - Xp) (Yv - Yp)	270.2499	270.24990080000783
(Xv - Xp) (Yq - Yp)	270.24988	270.24990320000783
(Xq - Xp) (Yv - Yp) - (Xv - Xp) (Yq - Yp)	3.0517578E-5	-0.00000240000...

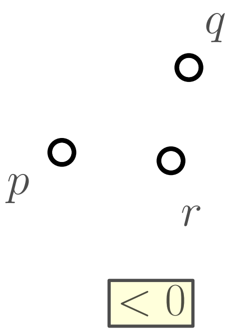




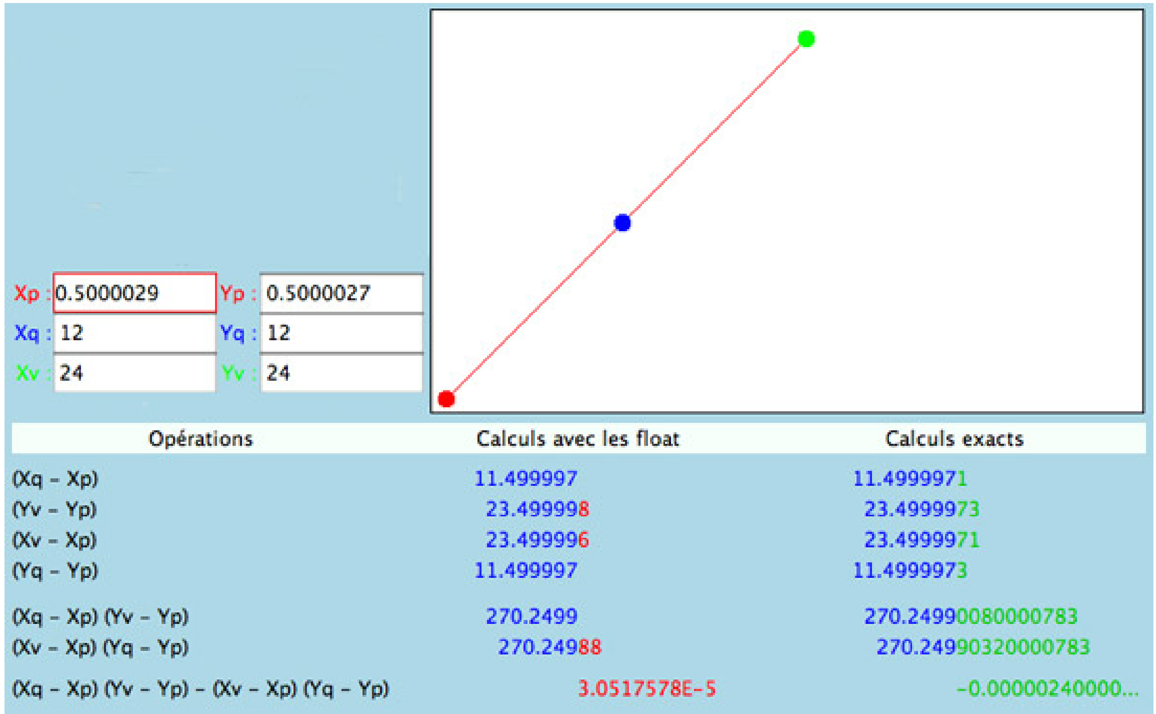
A close look at that example

Orientation of (p, q, r) given by the sign of $\begin{vmatrix} x_p & x_q & x_r \\ y_p & y_q & y_r \\ 1 & 1 & 1 \end{vmatrix}$.



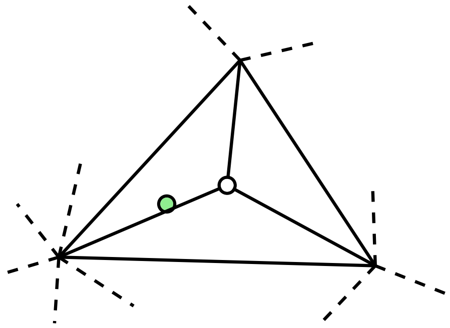


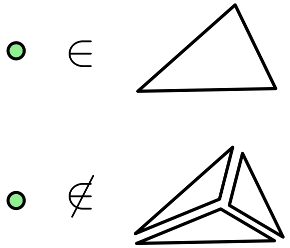
```
Float xp,yp,xq,yq,xr,yr;

Orientation = sign((xq-xp)*(yr-yp)-(xr-xp)*(yq-yp));
```



Xp : 0.5000029 Yp : 0.5000027
Xq : 12 Yq : 12
Xv : 24 Yv : 24

Opérations	Calculs avec les float	Calculs exacts
(Xq - Xp)	11.499997	11.4999971
(Yv - Yp)	23.499998	23.4999973
(Xv - Xp)	23.499996	23.4999971
(Yq - Yp)	11.499997	11.4999973
(Xq - Xp) (Yv - Yp)	270.2499	270.2499008000783
(Xv - Xp) (Yq - Yp)	270.24988	270.2499032000783
(Xq - Xp) (Yv - Yp) - (Xv - Xp) (Yq - Yp)	3.0517578E-5	-0.00000240000...





Consequences of numerical rounding

A ”correct” code can make incorrect decisions. These errors are inconsistent.

Crash, infinite loops, smooth execution but wrong answer... which is the worse?

Can be hard to detect...

Interval arithmetic

Keep the precision bounded but keep track of the error.

A number is represented by an interval (reduced to a single element if precision is sufficient).

Define all operations on intervals.

$$24 - 0.5000027 = 23.4999973 \sim 23.499998$$

becomes $[24, 24] - [0.5000027, 0.5000027] = [23.49999, 23.50000]$.

Interval arithmetic

Keep the precision bounded but keep track of the error.

A number is represented by an interval (reduced to a single element if precision is sufficient).

Define all operations on intervals.

$$24 - 0.5000027 = 23.4999973 \sim 23.499998$$

becomes $[24, 24] - [0.5000027, 0.5000027] = [23.49999, 23.50000]$.

If the interval does not contain 0 then we can decide the sign with certainty.

This suffices "most of the time".
Otherwise, we need more precision... Restart the computation with twice as many digits.

Interval arithmetic

Keep the precision bounded but keep track of the error.

A number is represented by an interval (reduced to a single element if precision is sufficient).

Define all operations on intervals.

$$24 - 0.5000027 = 23.4999973 \sim 23.499998$$

becomes $[24, 24] - [0.5000027, 0.5000027] = [23.49999, 23.50000]$.

If the interval does not contain 0 then we can decide the sign with certainty.

This suffices "most of the time".
Otherwise, we need more precision... Restart the computation with twice as many digits.

If the result of the computation is exactly 0 we will never have enough precision...
For those few cases, we need to be able to do the computations exactly.

Exact number types for integers, rational numbers, algebraic numbers.

Algebraic numbers

$\sqrt[5]{23}$ and $\sqrt[7]{25}$ are not integers, but we can still compare them exactly using integer arithmetic.

Algebraic numbers

$\sqrt[5]{23}$ and $\sqrt[7]{25}$ are not integers, but we can still compare them exactly using integer arithmetic.

A real r is algebraic if there exists a polynomial $P(X)$ with integer coefficients such that $P(r) = 0$.

Algebraic numbers

$\sqrt[5]{23}$ and $\sqrt[7]{25}$ are not integers, but we can still compare them exactly using integer arithmetic.

A real r is algebraic if there exists a polynomial $P(X)$ with integer coefficients such that $P(r) = 0$.

What about $n \in \mathbb{N}$, $f \in \mathbb{Q}$, $\sqrt{2}$, $\sqrt[5]{17}$, e, π... ?

Algebraic numbers

$\sqrt[5]{23}$ and $\sqrt[7]{25}$ are not integers, but we can still compare them exactly using integer arithmetic.

A real r is algebraic if there exists a polynomial $P(X)$ with integer coefficients such that $P(r) = 0$.

What about $n \in \mathbb{N}$, $f \in \mathbb{Q}$, $\sqrt{2}$, $\sqrt[5]{17}$, e, π... ?

The set of algebraic numbers is closed under $+, -, \times, /, x \mapsto x^t$ for $t \in \mathbb{Q}$ (in particular $\sqrt{\ }$).

Algebraic numbers

$\sqrt[5]{23}$ and $\sqrt[7]{25}$ are not integers, but we can still compare them exactly using integer arithmetic.

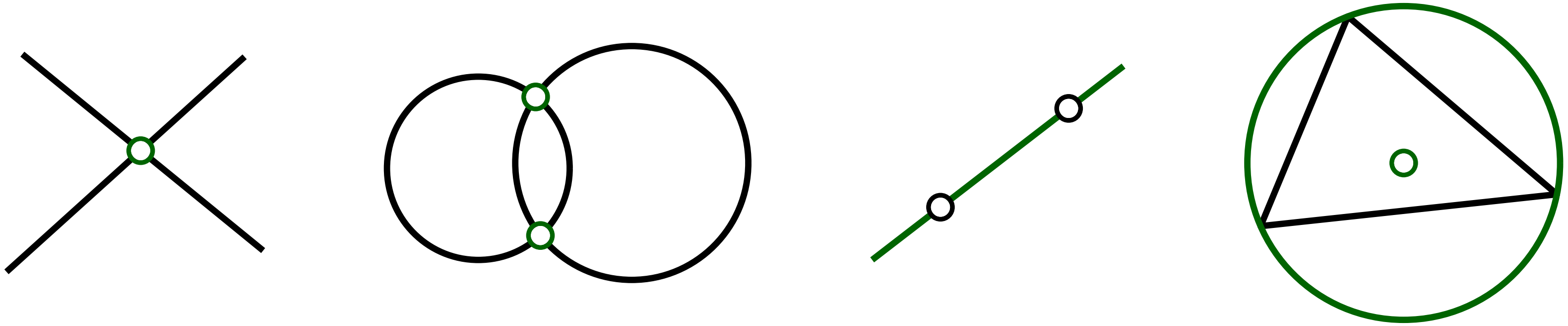
A real r is algebraic if there exists a polynomial $P(X)$ with integer coefficients such that $P(r) = 0$.

What about $n \in \mathbb{N}$, $f \in \mathbb{Q}$, $\sqrt{2}$, $\sqrt[5]{17}$, e, π... ?

The set of algebraic numbers is closed under $+, -, \times, /, x \mapsto x^t$ for $t \in \mathbb{Q}$ (in particular $\sqrt{\ }$).

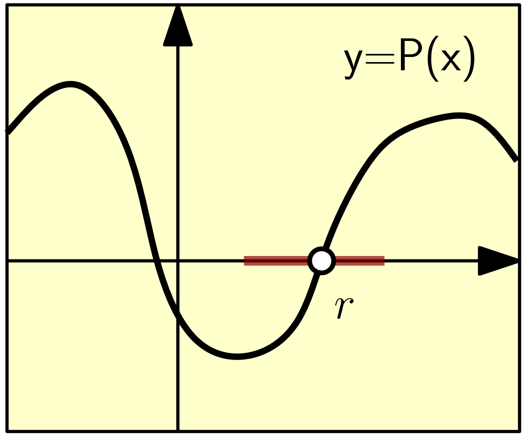
There are few algebraic numbers (ie countably many).

The result of most classical operations on geometric objects defined by integers
can be described using algebraic numbers.



Representing and manipulating algebraic numbers

An algebraic number can be represented by a polynomial (a family of integers) and an isolation interval.

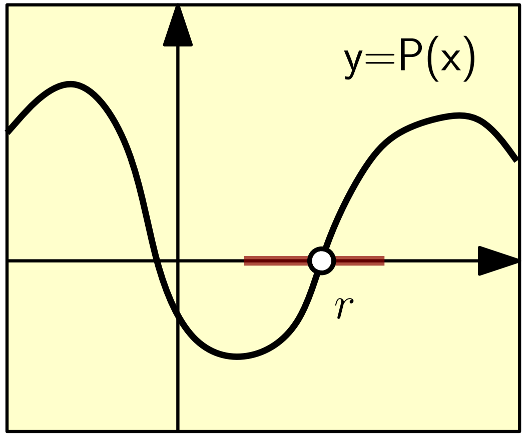


Interval containing a single root of P.

Ex: $\sqrt{2} \simeq (X^2 - 1, [1, 2])$.

Representing and manipulating algebraic numbers

An algebraic number can be represented by a polynomial (a family of integers) and an isolation interval.



Interval containing a single root of P.

Ex: $\sqrt{2} \simeq (X^2 - 1, [1, 2])$.

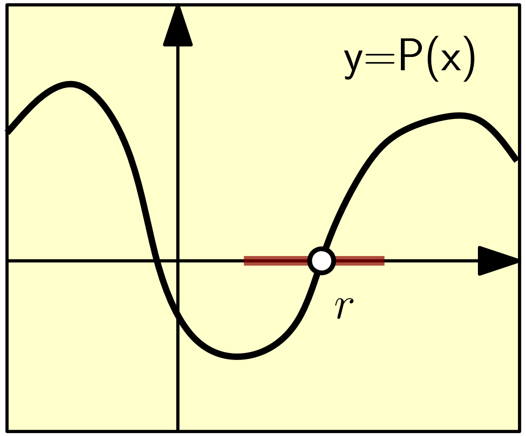
Given two algebraic numbers a and b represented by polynomials and isolation intervals,

we can compute a polynomial / isolation interval that represents:

$a + b, a - b, a \times b, \frac{a}{b}, a^2, \sqrt{a}$, etc...

Representing and manipulating algebraic numbers

An algebraic number can be represented by a polynomial (a family of integers) and an isolation interval.



Interval containing a single root of P.

Ex: $\sqrt{2} \simeq (X^2 - 1, [1, 2])$.

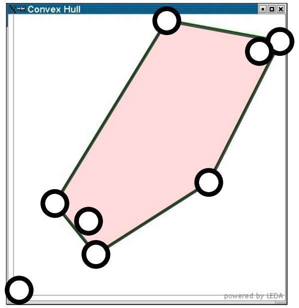
Given two algebraic numbers a and b represented by polynomials and isolation intervals,

we can compute a polynomial / isolation interval that represents:

$$a + b, a - b, a \times b, \tfrac{a}{b}, a^2, \sqrt{a}, \text{ etc...}$$

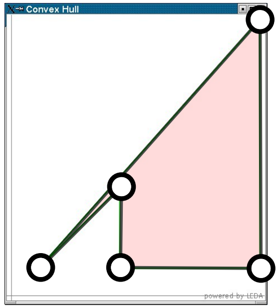
Implemented in the C/C++ CORE library.

Using algebraic numbers



```
Float xp,yp,xq,yq,xr,yr;

Orientation = sign((xq-xp)*(yr-yp)-(xr-xp)*(yq-yp));
```

leads to



These problems can be avoided by using

```
Core::Expr xp,yp,xq,yq,xr,yr;
Orientation = sign((xq-xp)*(yr-yp)-(xr-xp)*(yq-yp));
```

Decision VS constructions

Distinguish between decision (for branching) and constructions.

Decisions are made by evaluating signs of polynomial in the input and can be filtered.

Constructions produce a geometric object from the input; representing exactly that object is costly.

Decision VS constructions

Distinguish between decision (for branching) and constructions.

Decisions are made by evaluating signs of polynomial in the input and can be filtered.

Constructions produce a geometric object from the input; representing exactly that object is costly.

If the decisions are correct, the "type" of the result is correct (constructions do not matter much).

Using repeated substitutions, we can avoid using constructions when branching.

Decision VS constructions

Distinguish between decision (for branching) and constructions.

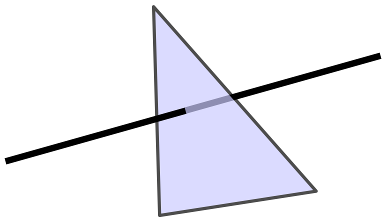
Decisions are made by evaluating signs of polynomial in the input and can be filtered.

Constructions produce a geometric object from the input; representing exactly that object is costly.

If the decisions are correct, the "type" of the result is correct (constructions do not matter much).

Using repeated substitutions, we can avoid using constructions when branching.

Ex: line/triangle intersection test



find intersection with plane, compute barycentric coordinates.

Decision VS constructions

Distinguish between decision (for branching) and constructions.

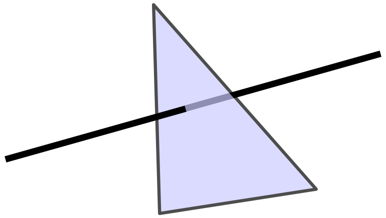
Decisions are made by evaluating signs of polynomial in the input and can be filtered.

Constructions produce a geometric object from the input; representing exactly that object is costly.

If the decisions are correct, the "type" of the result is correct (constructions do not matter much).

Using repeated substitutions, we can avoid using constructions when branching.

Ex: line/triangle intersection test



find intersection with plane, compute barycentric coordinates.
→ evaluate the sign of polynomials of degree 6.

Decision VS constructions

Distinguish between decision (for branching) and constructions.

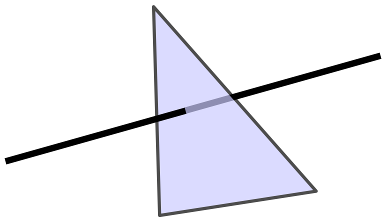
Decisions are made by evaluating signs of polynomial in the input and can be filtered.

Constructions produce a geometric object from the input; representing exactly that object is costly.

If the decisions are correct, the "type" of the result is correct (constructions do not matter much).

Using repeated substitutions, we can avoid using constructions when branching.

Ex: line/triangle intersection test



find intersection with plane, compute barycentric coordinates.
→ evaluate the sign of polynomials of degree 6.

Evaluate 3D orientations of quadruples of points

Decision VS constructions

Distinguish between decision (for branching) and constructions.

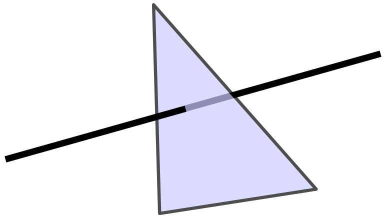
Decisions are made by evaluating signs of polynomial in the input and can be filtered.

Constructions produce a geometric object from the input; representing exactly that object is costly.

If the decisions are correct, the "type" of the result is correct (constructions do not matter much).

Using repeated substitutions, we can avoid using constructions when branching.

Ex: line/triangle intersection test



find intersection with plane, compute barycentric coordinates.
→ evaluate the sign of polynomials of degree 6.

Evaluate 3D orientations of quadruples of points
→ evaluate the sign of polynomials of degree 3.

Wrap-up: robustness

Treating degeneracies requires great care.

Numerical problems will arise.

If not treated properly, they produce crashes, infinite loops or wrong results.

Exact number types exist and are implemented. This is good enough for prototyping.

Reliability and efficiency are achieved by using good predicates
and filtering exact number type with interval arithmetic.

The End