Robustness issues in computational geometry

Marc Pouget

présentation trés largement inspirée du travail de Xavier Goaoc

Loria
Laboratoire lorrain de recherche
en informatique et ses applications

Why are geometric algorithms hard to implement correctly?

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.

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.

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

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

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?

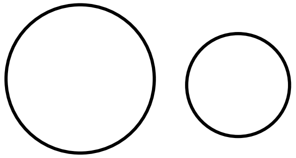
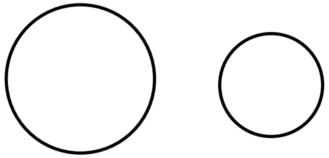
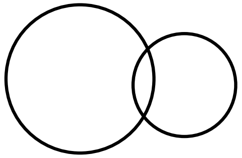
Perturbing? Easier said than done

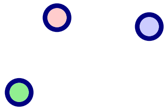
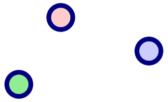
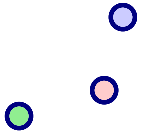
In principle, perturbing the points eliminate degeneracies.

Perturbing? Easier said than done

In principle, perturbing the points eliminate degeneracies.

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

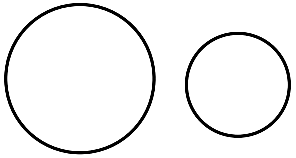
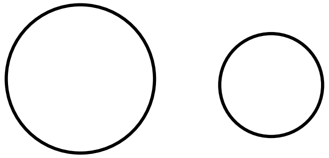
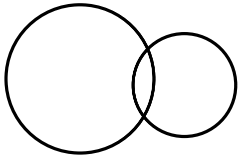

can be perturbed in

but not in


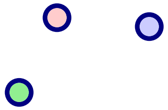
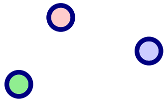
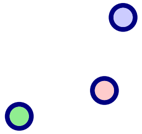

can be perturbed in

but not in


Perturbing? Easier said than done

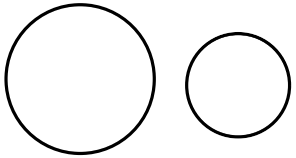
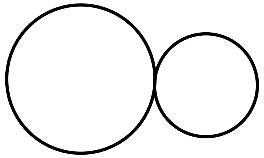
In principle, perturbing the points eliminate degeneracies.

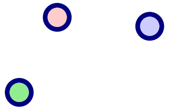
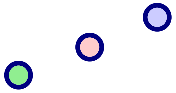
First issue: the perturbation should preserves non-degenerate inputs.


can be perturbed in

but not in



can be perturbed in

but not in


Second issue: the perturbation should not create new degeneracies.

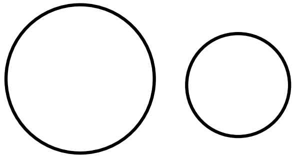

should not be perturbed in



should not be perturbed in


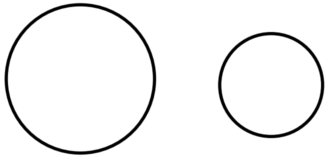
Perturbing? Easier said than done

In principle, perturbing the points eliminate degeneracies.

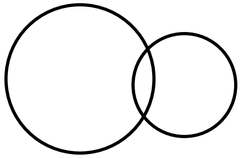
First issue: the perturbation should preserves non-degenerate inputs.

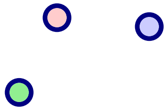


can be perturbed in

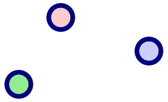


but not in

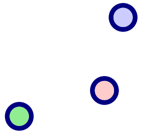




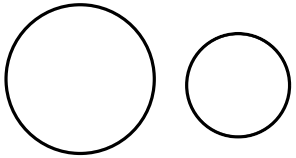
can be perturbed in



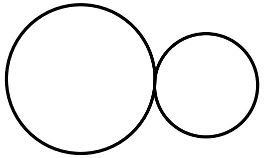
but not in

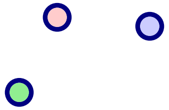


Second issue: the perturbation should not create new degeneracies.

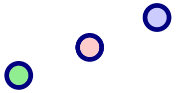


should not be perturbed in





should not be perturbed in



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

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.

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.

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

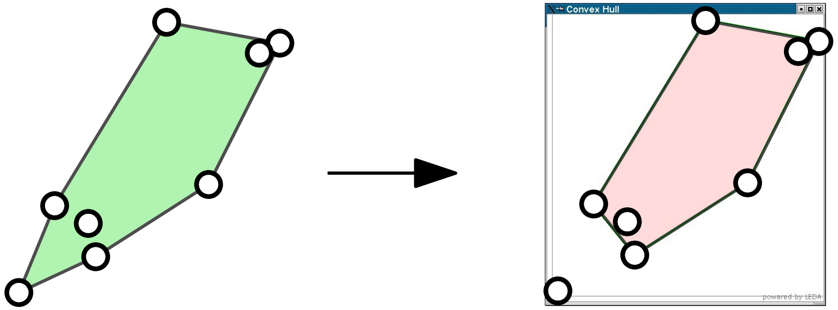
Second issue: numerical rounding

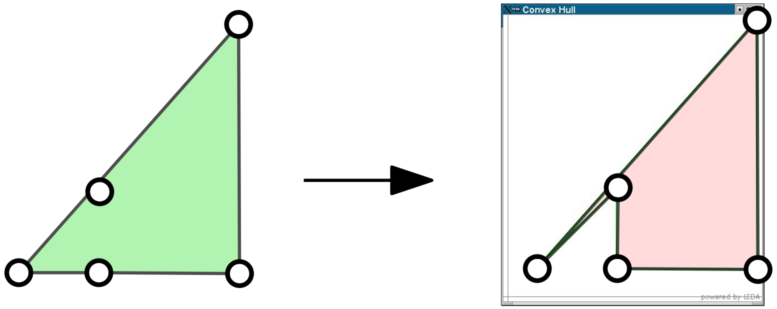
The arithmetic on a computer uses bounded precision (32 bits, 64 bits, IEEE float norms, etc...).

Small errors will be made in computations.

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

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





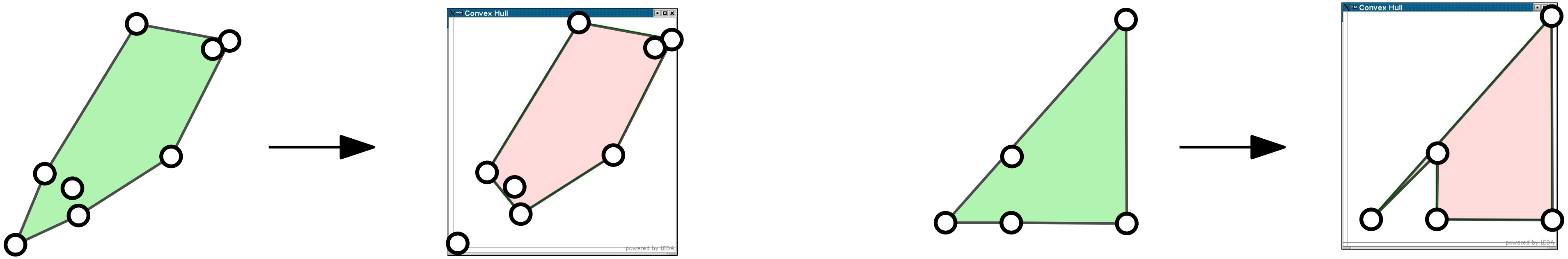
Second issue: numerical rounding

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

Small errors will be made in computations.

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

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

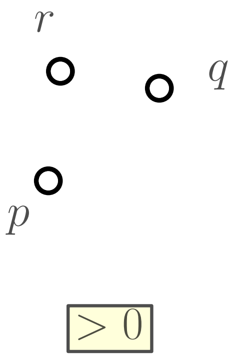


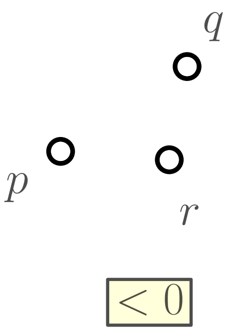
The problem: three points are nearly aligned, and the orientation predicates make inconsistent errors.

"Sometimes left, sometimes right".

A close look at that example

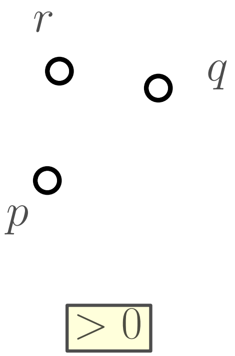
Orientation of (p, q, r) given by the sign of $\begin{vmatrix} x_p & x_q & x_r \\ y_p & y_q & y_r \\ 1 & 1 & 1 \end{vmatrix}$.

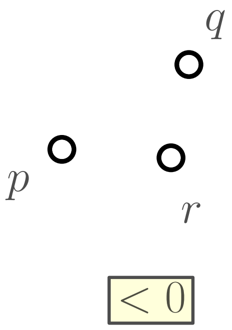




A close look at that example

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

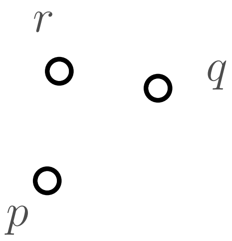




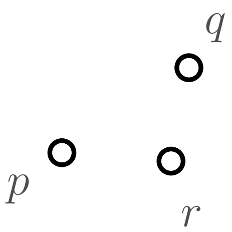
```
Float xp,yp,xq,yq,xr,yr;

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 $\begin{vmatrix} x_p & x_q & x_r \\ y_p & y_q & y_r \\ 1 & 1 & 1 \end{vmatrix}$.

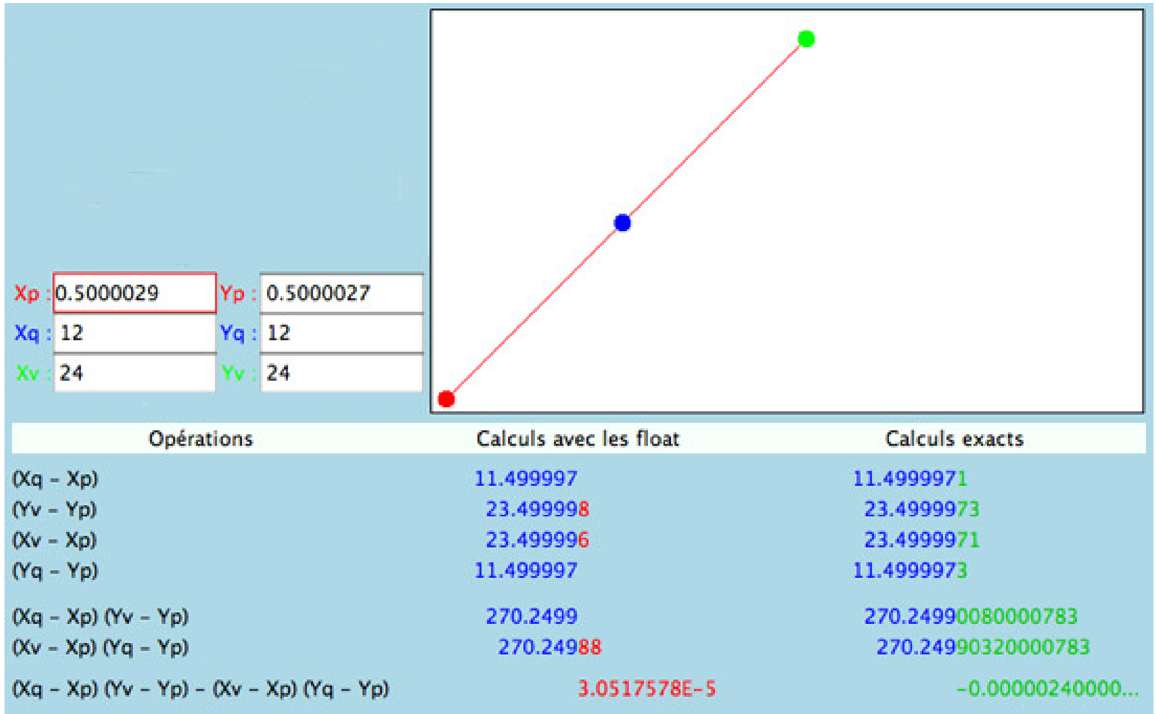


$\boxed{> 0}$



$\boxed{< 0}$

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

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

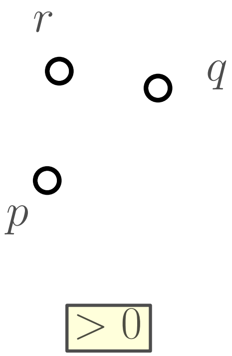


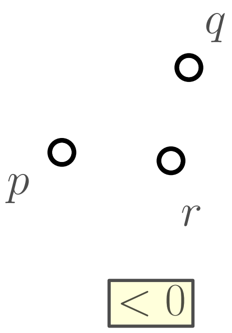
Xp : 0.5000029 Yp : 0.5000027
Xq : 12 Yq : 12
Xv : 24 Yv : 24

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	270.2499008000783
(Xv – Xp) (Yq – Yp)	270.24988	270.2499032000783
(Xq – Xp) (Yv – Yp) – (Xv – Xp) (Yq – Yp)	3.0517578E-5	-0.00000240000...

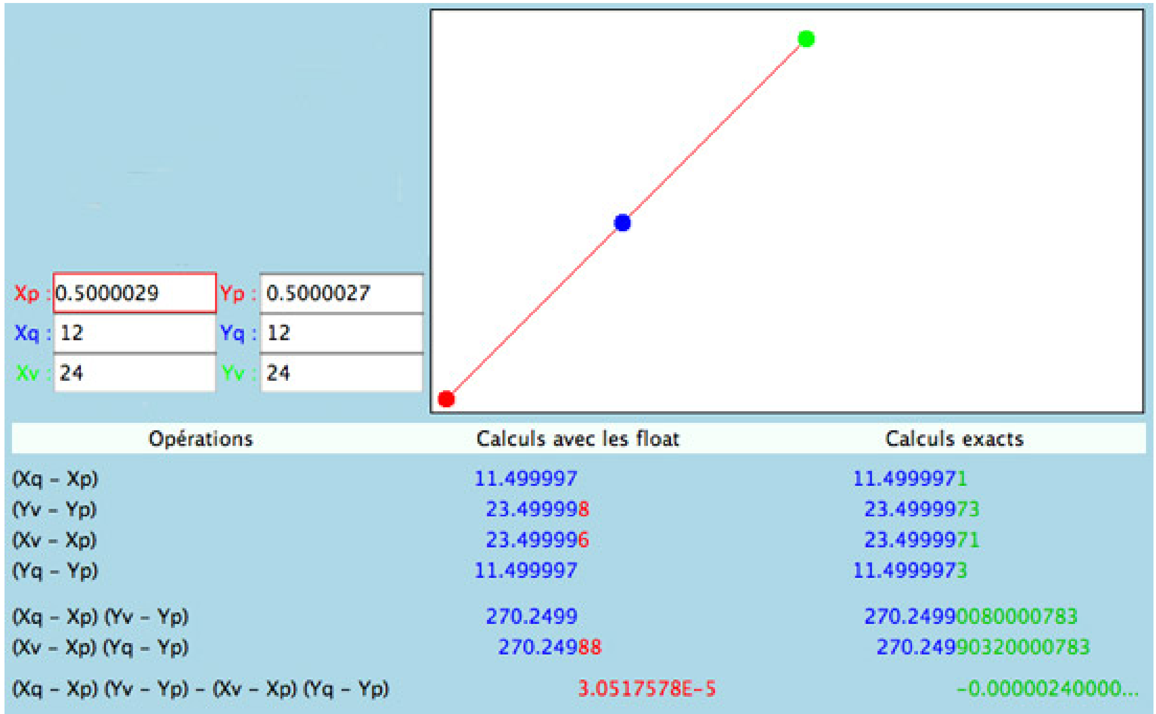
A close look at that example

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



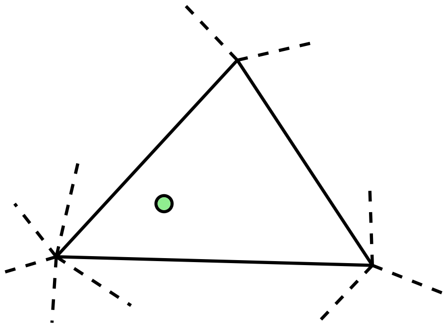


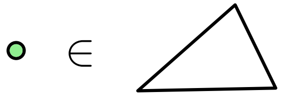
```
Float xp,yp,xq,yq,xr,yr;

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



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

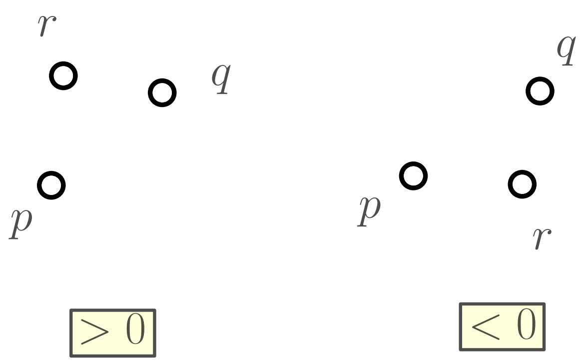
Opérations	Calculs avec les float	Calculs exacts
(Xq - Xp)	11.499997	11.4999971
(Yv - Yp)	23.499998	23.4999973
(Xv - Xp)	23.499996	23.4999971
(Yq - Yp)	11.499997	11.4999973
(Xq - Xp) (Yv - Yp)	270.2499	270.24990080000783
(Xv - Xp) (Yq - Yp)	270.24988	270.24990320000783
(Xq - Xp) (Yv - Yp) - (Xv - Xp)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



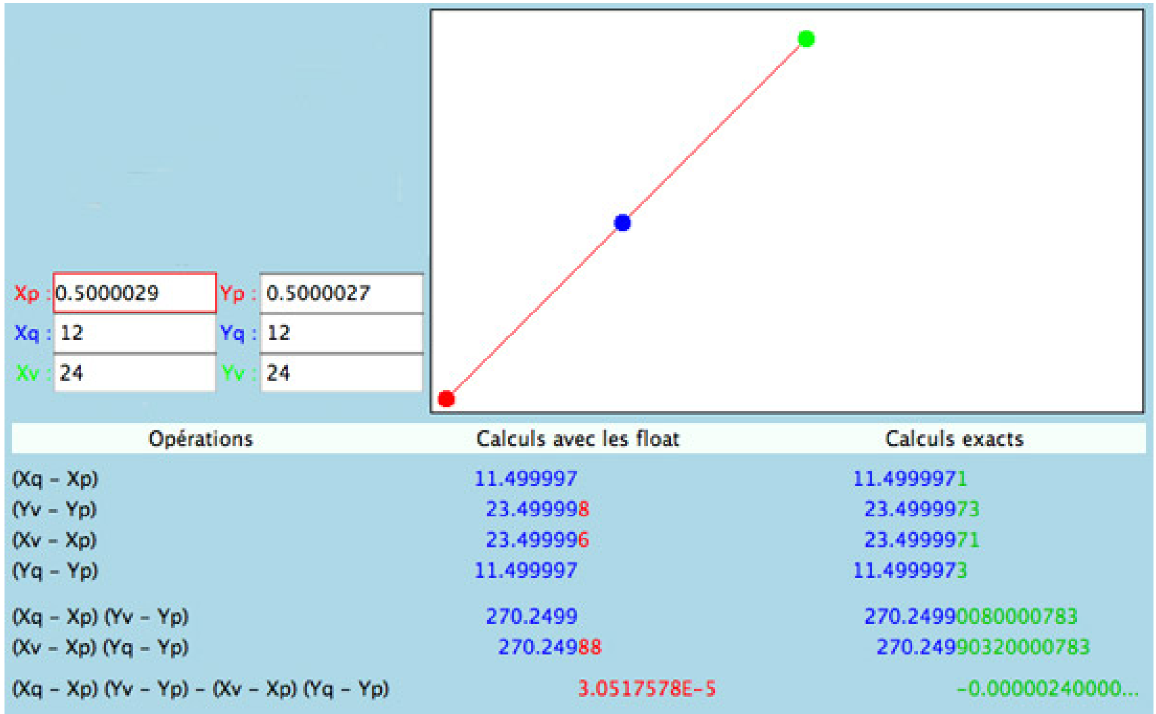


A close look at that example

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

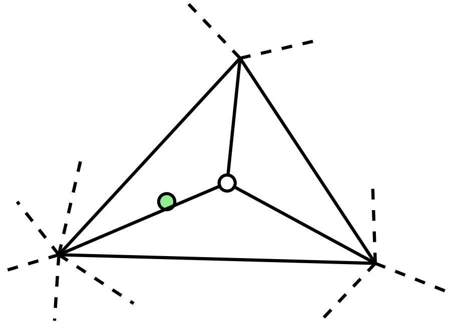


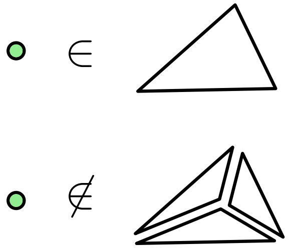
```
Float xp,yp,xq,yq,xr,yr;

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	270.24990080000783
(Xv – Xp) (Yq – Yp)	270.24988	270.24990320000783
(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.

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.

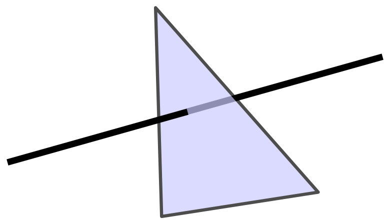
Decisions are made by evaluating signs of polynomial in the input and can be filtered.

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.

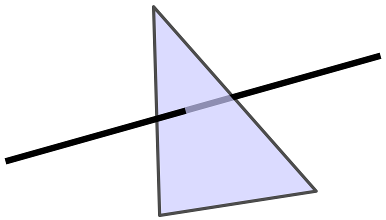
Decisions are made by evaluating signs of polynomial in the input and can be filtered.

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.

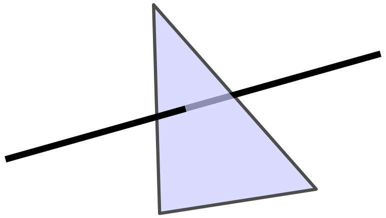
Decisions are made by evaluating signs of polynomial in the input and can be filtered.

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.

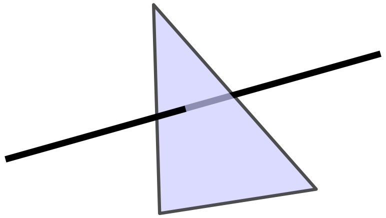
Decisions are made by evaluating signs of polynomial in the input and can be filtered.

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