Elements of computational geometry

CAD & Computational Geometry

Book

- Recommended book : M. de Berg, O. Cheong , M. van Keveld, M. Overmars , Computational Geometry, 3rd ed. 2008, Springer-Verlag

  Available in electronic format at the library !

Introduction

- Example : Problem of urgent thirst

  One needs to proceed to the closest bar.

  We need a map subdividing the local area in small regions for which the closest bar is indicated

  - What is the shape of the regions ?
  - How to build the map ?
  - How to determine in which region we are located?
  - ...

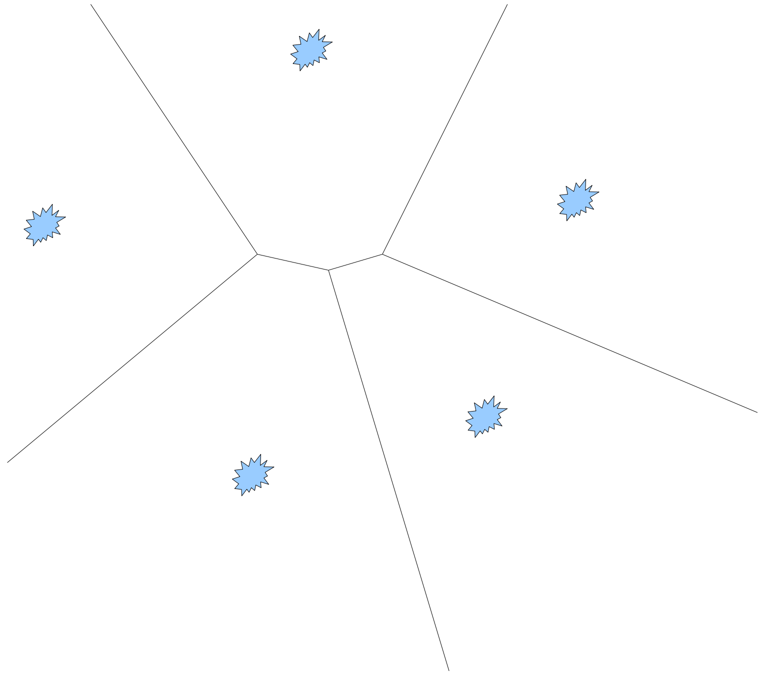
Introduction

Problem of urgent thirst

Solution : Voronoi diagram.

Useful in many, many other applications

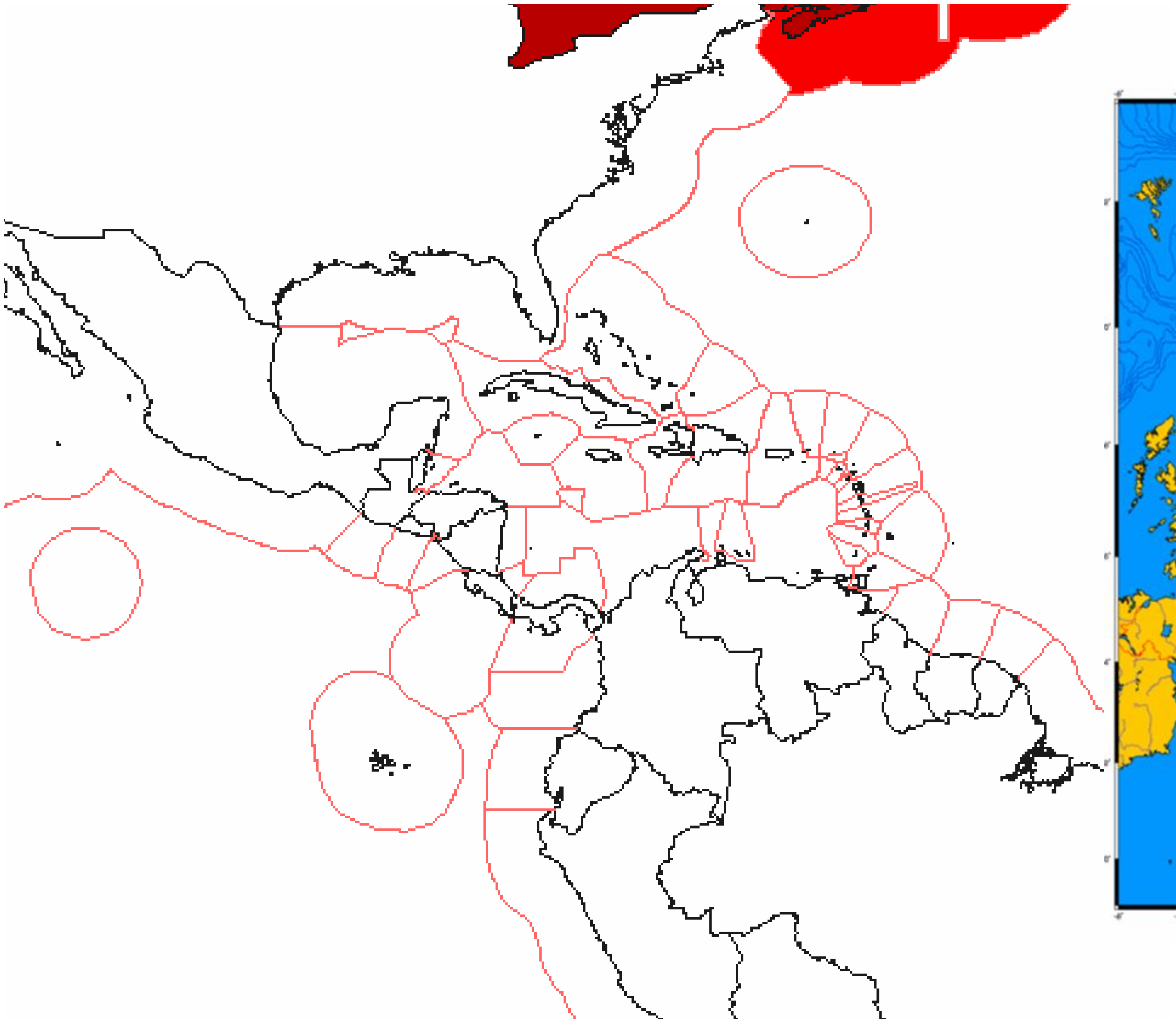
- Determination of Exclusive Economic Zones (in part)
- Robot path planning
- Crystal growth
- etc...

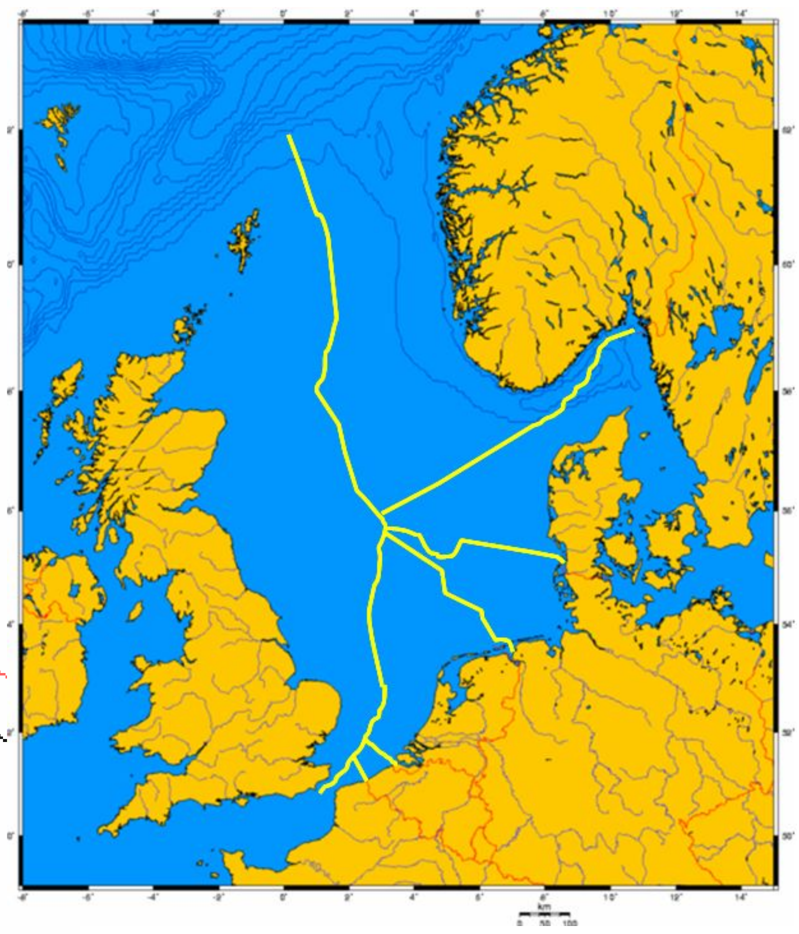


CAD & Computational Geometry

Introduction

- Exclusive Economic Zones...





Introduction

- Another example
  One has two maps at our disposal
  - One describes the buildings (including bars !)
  - The other the ways (routes, paths and the like)

  To plan a journey, both maps have to be combined. It amounts to localize entities from a map onto the other map, compute intersections etc...

  These are common problems found in GIS, “Geographic Information Systems” such as Google maps, openstreetmaps...

Outline of CG

- Introduction
  - Some notions of algorithmic complexity
  - One example : convex hull of a set of points in 2D
  - Basic data structures
- Line intersections
- Polygon triangulation
- Delaunay triangulation
- Search structures
- Voronoi diagrams

Introduction

Algorithmic Complexity

- Notions of algorithmic complexity

  Following definition is given by D. Knuth (1976)

  - Big O notation :
    $g(n) \in O(f(n))$ iff $\exists C > 0$ such that $g(n) \leq Cf(n) \forall n \geq n_0$
    Functions at most as big as $Cf(n)$:
    upper bound to the complexity
  - Capital Omega notation:
    $g(n) \in \Omega(f(n))$ iff $\exists C > 0$ such that $g(n) \geq Cf(n) \forall n \geq n_0$
    Functions at least as big as $Cf(n)$:
    Lower bound of the complexity

CAD & Computational Geometry

Algorithmic Complexity

- Small thêta notation:
  $g(n) \in \theta(f(n))$ iff $\exists C_1, C_2 > 0$ such that
  $C_1 f(n) \leq g(n) \leq C_2 f(n) \; \forall n \geq n_0$
  Functions with the same order of magnitude

  This is the concept one uses to qualify an algorithm as optimal with a theoretical result
- Small o notation:
  $g(n) \in o(f(n))$ iff $\forall C > 0 \; , \; g(n) \leq Cf(n) \; \forall n \geq n_0$
  Functions at most as big as $Cf(n)$ not including $\theta(f(n))$, for all C... (including $C \rightarrow 0$)

Algorithmic Complexity

The complexity may concern :

- Execution time T
- Maximal Memory use M

There exists two “types” of measures of complexity

- On the “worst case” (limit = theoretical performance)

  One gets an upper bound on the performance
- On an “average” case (practical performance)

  Problem : definition of a typical use case for the algorithm

Algorithmic Complexity

- On the “worst case”

  This is generally a result that one can get theoretically by analyzing the algorithm without considering any practical distribution on the input date

- On the “average” case

  Theoretical results are relatively scarce :

  - Problem to define the “average case”

  - Tremendous mathematical difficulties even if the distribution considered in the average case is simple

Algorithmic Complexity

- Unfortunately (or fortunately), it is clear that the measure of the complexity on the average case is (or can be) better than that on the worst case.
  - An algorithm of bad theoretical complexity may be good at use depending on the type of data used as input.
  - It remains very important to *test*... and the theoretical complexity says nothing about the coefficient of proportionality … e.g. some algorithms may have better locality (use of local data), hence perform much better on recent CPU architectures (less cache misses) than older but theoretically better algorithms.

Algorithmic Complexity

- One may as well determine a complexity with respect to the output of the algorithm (the solution to the problem instead of the initial data)

  Sometimes, the complexity bound is depending on this output size

  One says that the algorithm has a complexity that is "output sensitive"

Algorithmic Complexity

Classification of geometrical algorithms

- Deterministic algorithms

  For a given input, their execution is always following the same path. The result is obviously identical.

- Randomized algorithms

  For a given set of input data, the algorithm may execute using different paths, but the result will remain the same (and not depend on the path)...

  Advantage : complexity of an average case even if the input data is unfavorable.

  But: beware of floating point computations which are dependent on the execution path even if theoretical results should be the same.

Algorithmic Complexity

- Incremental algorithms

  Those allow to build the optimal solution while adding primitives one by one, in the order given by the user. There is no need to know the whole set of input data before starting.

- Dynamic algorithms

  Allow adding primitives as well as the deletion thereof, while keeping the whole solution valid at every time.

- Algorithm that do not fall into these categories have to take the complete input data at once. In case of a modification, the whole construction has to be started again.

  - Randomized algorithms are often of this type !

Algorithmic Complexity

- Algorithms that are efficient in C.G.

  Often conceived using classical paradigms

  - “Divide and conquer”
  - Recursion
  - ...
  - Cf classical books on algorithms (e.g. D. Knuth)

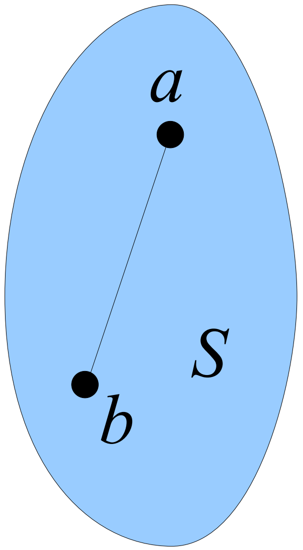
  Use of the geometrical characteristic of the problem

  - Line sweeping algorithms
    - In the plane, one simulate the advance of an imaginary line…
  - ...

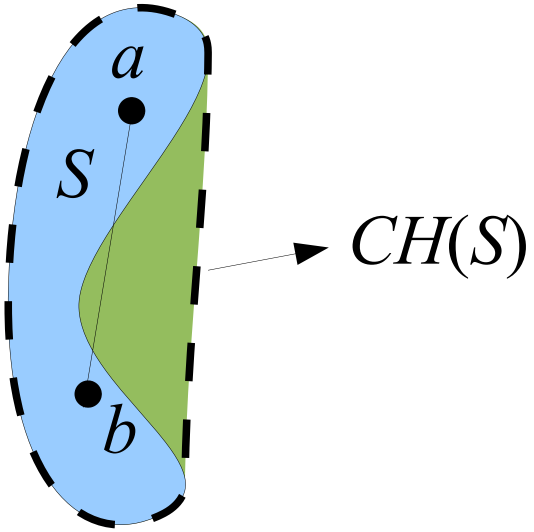
Convex hull in 2D

Convexity...

- A set S is convex <u>iff</u> for every couple of points $a, b \in S$, the line segment $\overline{ab}$ is completely contained in S.
- The *convex hull* of a set S is the smallest convex set $CH(S)$ that contains S.
  It is also the intersection of every convex set containing S.



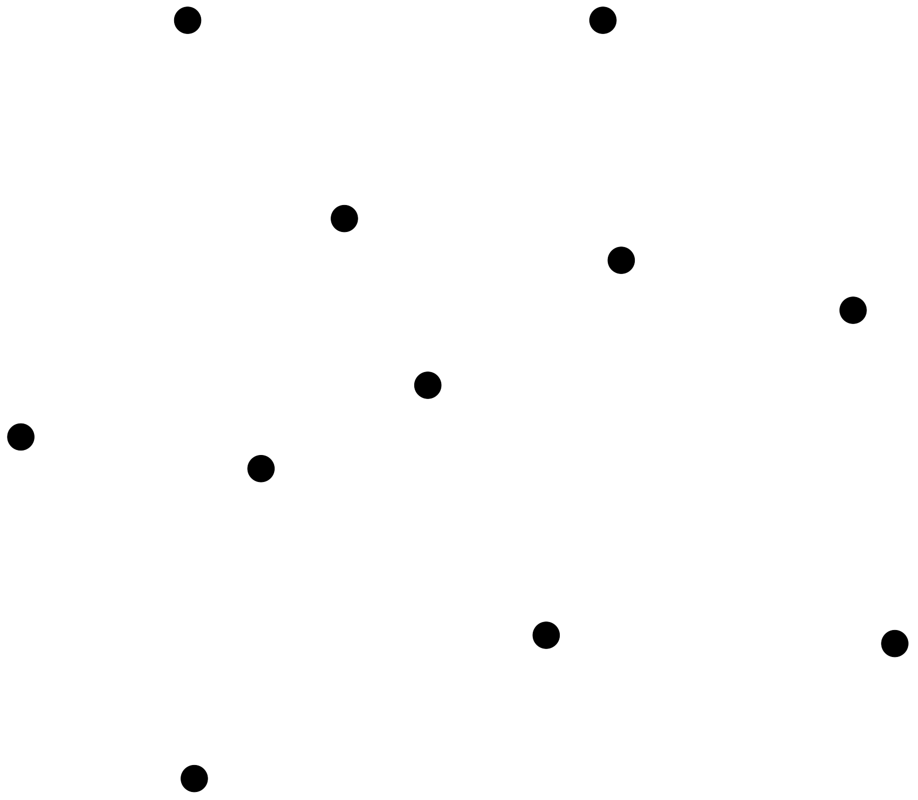
S is a convex set



S isn't a convex set
$CH(S)$ is a convex set

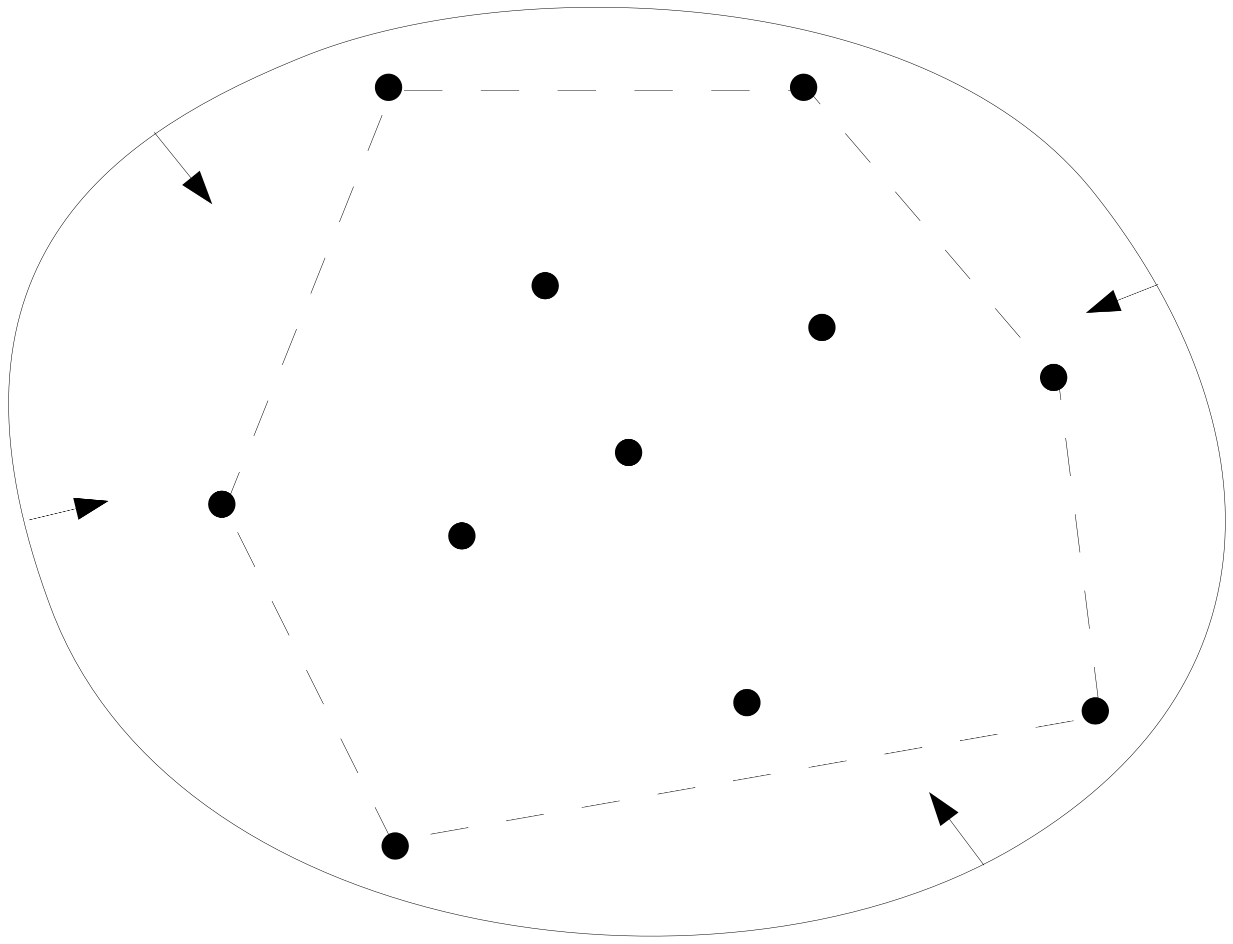
Convex hull of a set of points

- Let P a set of n points in the euclidean plane.
  One wish to compute the convex hull of this set..



Convex hull of a set of points

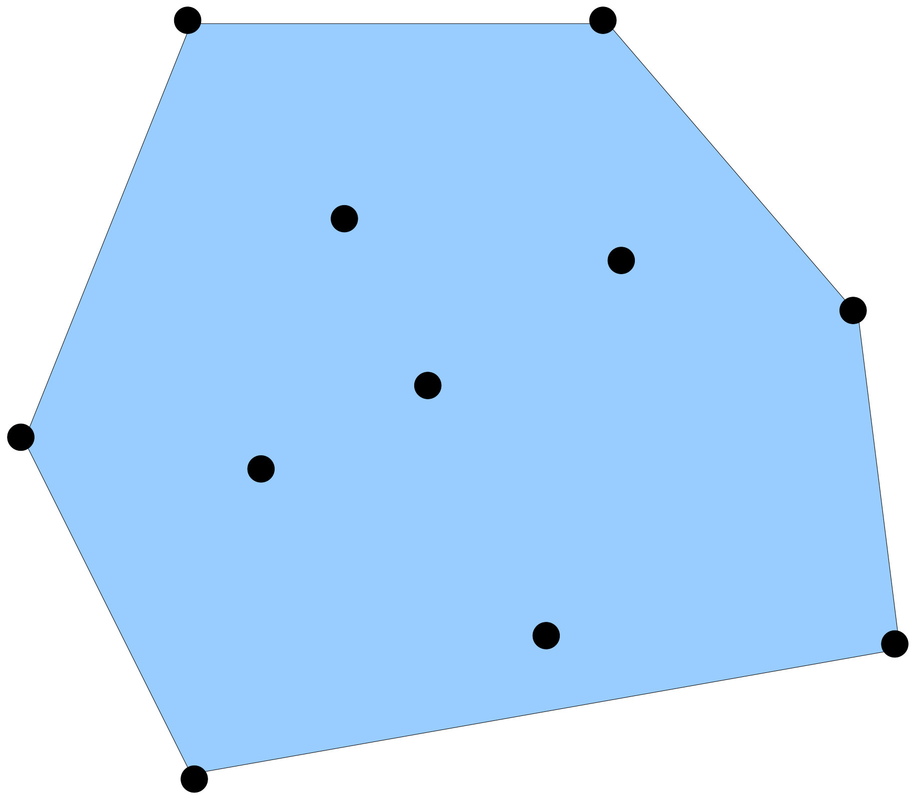
It is the smallest convex (closed) polygon for which the vertices belong to P and which contains every point of P.



CAD & Computational Geometry

Convex hull of a set of points

- The previous definition is equivalent to the original definition (but it must be proven ...)

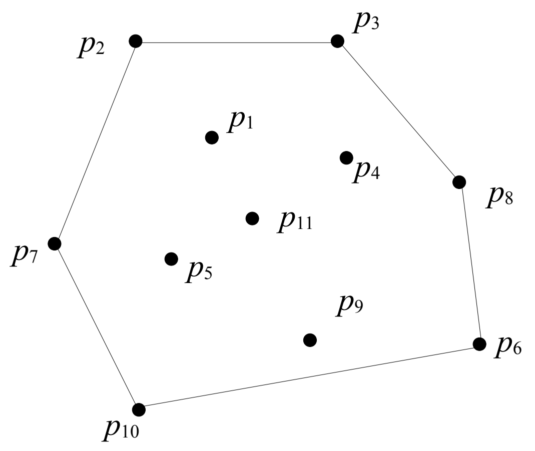


Convex hull of a set of points

- How to *compute* the convex hull ?

What means *compute* ?
One has to define the input data and the output (expected result)

For instance, one could list the vertices of the polygon in a clockwise manner.



Input = Set of points P in any order

$p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}$

Output = Convex Hull $CH(P)$

a clockwise list of points :

$p_7, p_2, p_3, p_8, p_6, p_{10}$

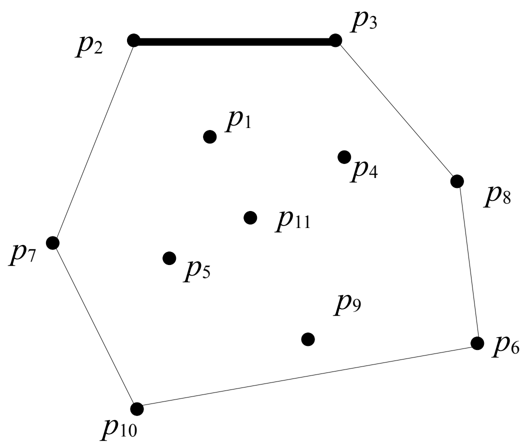
Convex hull of a set of points

- How to define the algorithm

  The general definition of the convex hull is not very useful in this case : the intersection of every convex set containing P involves an infinite number of operations.

  We will rather observe that the convex hull has a special structure ***in this case*** : it is a convex polygon whose vertices belong to P.

- In particular, lets consider one of the sides of the polygon...

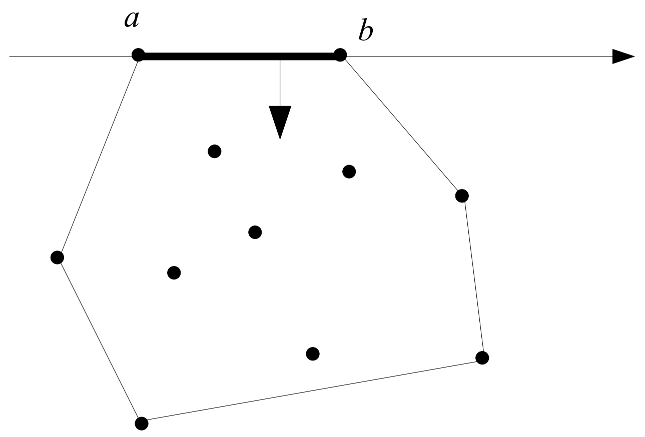


Convex hull of a set of points

For one of the edges of the polygon $CH(P)$

- The extremities belong to P.
- If we follow the line going from a to b so that $CH(P)$ is on the "right" of us, then all other points of P are also on the right.
- If one takes any oriented line ab and find that all other points of P are on the right, then ab belongs to the convex hull $CH(P)$.

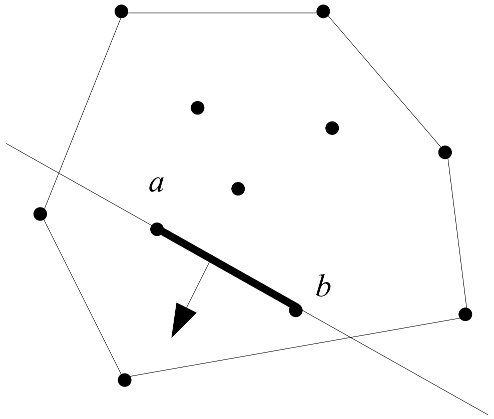
  Only then one can start thinking about writing an *algorithm* !



Convex hull of a set of points

(Slow) algorithm:

```
ConvexHullSlow($P$, $L$)
Intput : a set of points $P$ in the euclidean plane
Output : an ordered list of points $L$ of the vertices defining the polygon CH($P$) in a clockwise order
{
 $E = Ø$ // set of edges
 For every pair of points $(a, b) \in P \times P$ with $a \neq b$
 {
  $valid = true$
  For every point of $p \in P$, $p \neq a$ et $p \neq b$
  {
   If $p$ is on the left of $ab$ then set $valid = false$, and exit the loop.
  }
  If ($valid = true$) then apend ($a,b$) to $E$.
 }
 Build an ordered list $L$ of vertices from the
                    unordered set of edges $E$.
}
```

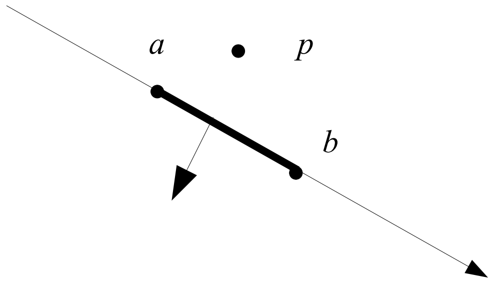


Convex hull of a set of points

- Explanations

  The operation

```
...
    If $p$ is on the left of $ab$ Then ...
...
```



is a ***predicate*** : it is a basic operation necessary for the expected exectution of the algorithm. In the sequel of the course, we expect such operations are available. Here, it is obvious that this predicated can be computed in a ***constant time*** (i.e. independent from n)

- It means that the asymptotic behavior (when n becomes large) is not affected by the predicate's own complexity.

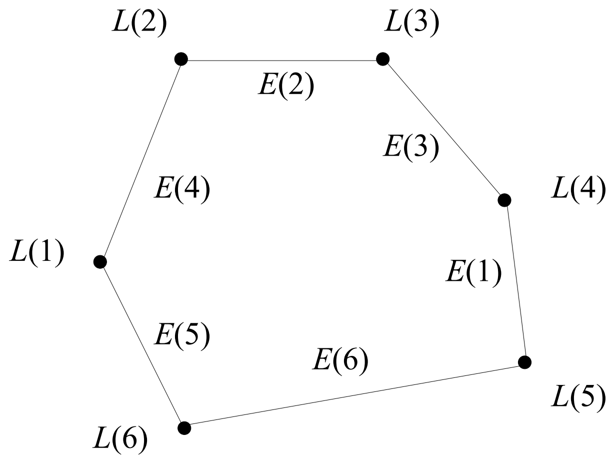
Convex hull of a set of points

- Explanations

  …

  Build an ordered list L of vertices from the unordered set of edges E.

  …
- is a non trivial operation.
  Here is an example of implementation:



```
BuildOrderedList($E$, $L$)
Input : Unordered list of edges $E$
Output : a clockwise ordered list of the vertices of the polygon formed by E
{
  Take the first element of $E$ : $E(1)$
  Add the starting point of E(1) , and the destination into $L$ ; Delete $E(1)$ from the list.
  While E contains more than one element
  {
    Fine the element $E(i)$ for which the starting point is equal to the last element of $L$.
    Add the destination of E(i) into L ; delete E(i) from the list E ;
  }
}
```

The complexity is proportional to n^2 , but it is possible to make this better (in $n \log n$) by an adequate initial sort.

Convex hull of a set of points

Complexity analysis

Quite easy in this case ...

There are $n^2 - n$ pair of points to be tested

- For each pair, the predicate is checked against the $n - 2$ other points.

Thus, there are $(n^2 - n)(n - 2) = n^3 - 3n^2 + 2n$ tests to be made.

When n is huge, the n^3 term becomes dominant. We say that the algorithm takes a time in $\mathrm{O}(n^3)$ to execute.

We did not take the final sort into account. In the most stupid implementation, it takes $\mathrm{O}(n^2)$ to execute.

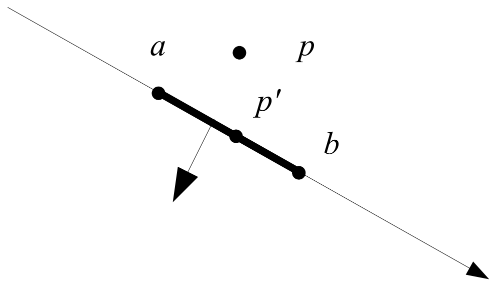
- It does not affect the global complexity which remains at $\mathrm{O}(n^3)$.

Generally speaking, and algorithm in $\mathrm{O}(n^3)$ is way too costly for any practical use if n is significant.

Convex hull of a set of points

A more thorough analysis of the predicate

- A point is not always *to the right* or *to the left* of a line. It can be *on* the line.



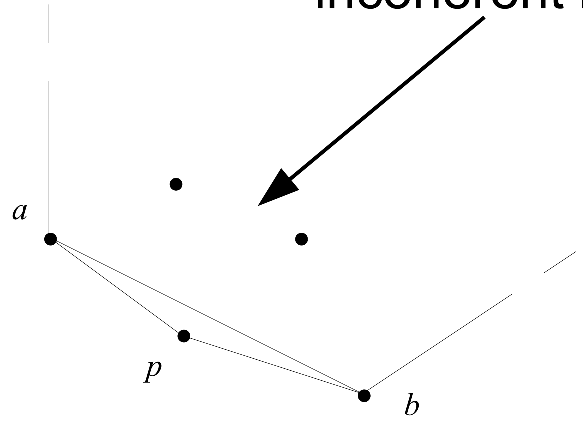
- It is a degenerate case because the points are not in general position. Otherwise said, some are collinear.

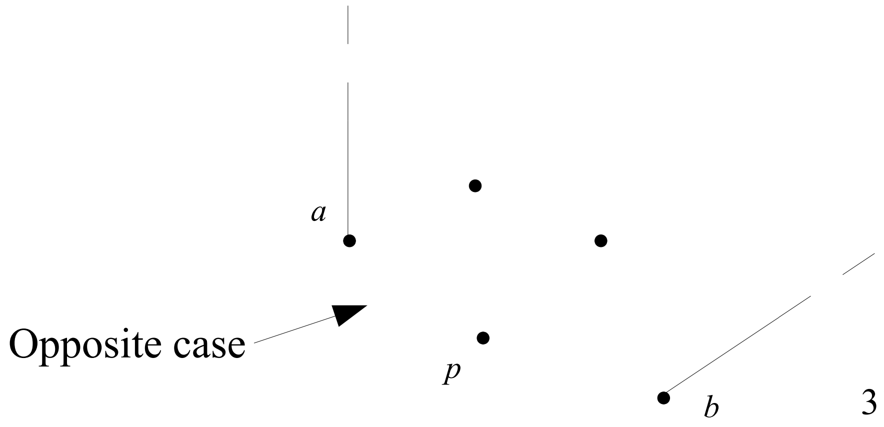
- Lets reformulate the test : an oriented segment ab is an edge of $CH(P)$ iff every other point in P is strictly on the right of ab, or if they are *on* the open segment ab.

  It makes sense, because a point in the middle (on the segment) could be considered as being part of convex hull, but this would lead to a topologically bigger convex hull (even though it has the same geometry), so the right thing to do is to avoid this, and keep only the longest edge corresponding to the collinear points.

Convex hull of a set of points

- A more thorough analysis of the predicate
  - We supposed that the numerical test is accurate (exact). However, with floating point coordinates, it is not the case. Sometimes the result given by the predicate is not exact, nor coherent.
    Let's consider the case of three points a,b et p almost collinear. The algorithm will test, among others, (a,b), (a,p) and (p,b).
    It is possible that nasty rounding errors lead to the test saying that, **at the same time**, p is to the right w.r. to (a,b), b is to the right w.r. to (a,p) and a is to the right w.r. to (p,b).
  - It is obviously geometrically impossible and will give an incoherent result ...





Convex hull of a set of points

- First algorithm
  - Is fundamentally correct ...

    if one supposes that the predicate is **exact** !!

  But :
  - It is particularly **slow** - $O(n^3)$
  - It has some issues with **degenerate cases**
  - Is **not robust** if the predicate is not exact (because of the multiple ways the predicate is used to asses the same results)

  It is obvious that better algorithms do exist.

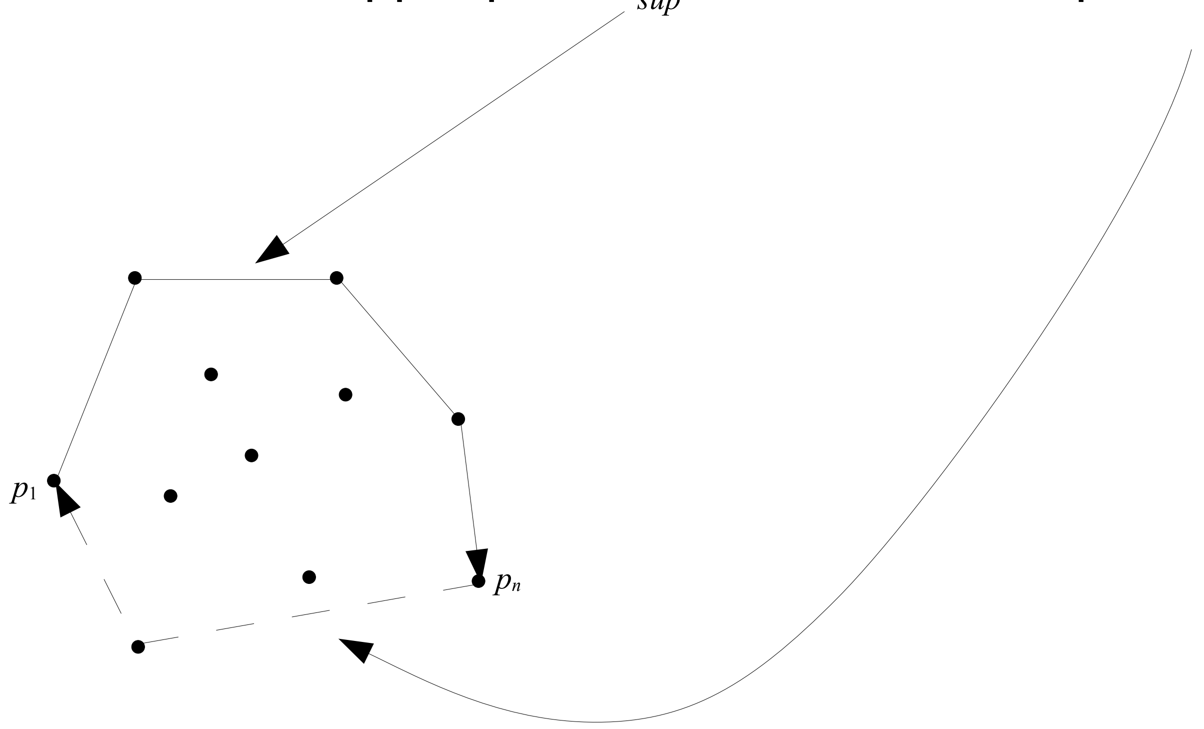
Convex hull of a set of points

- Graham’s algorithm

Graham, R.L. (1972). An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. *Information Processing Letters* 1, 132-133

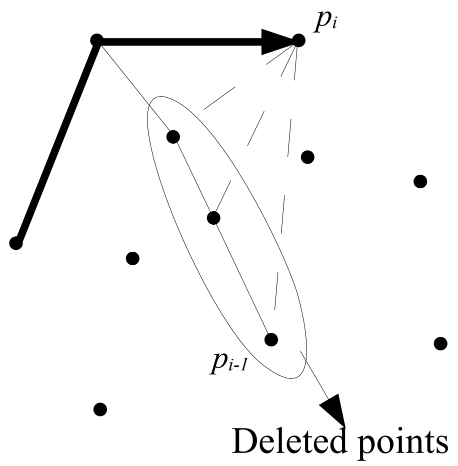
Convex hull of a set of points

- Algorithm
  - One classifies points from right to left
  - Then, one adds points of P one by one, keeping the solution up to date after each addition
  - The convex hull is computed in two steps
    - The upper part L_{sup} , then the lower part L_{inf}



Convex hull of a set of points

- The delicate step is the update of the convex hull after each new point insertion p_i .
  - From $p_1, \dots , p_{i-1}$ one wants $p_1 , \dots , p_\mathrm{i}$
  - There is one hint: when “walkin” a convex polygon in clowise fashion, every turn is a “right turn” at each vertex. Therefore :



- p_i est obviously the last point of L_{sup}, hence it is inserted
- One checks the 3 last points of L_{sup},
  - If the turn is a “right turn”, end here.
  - Otherwise, one has to delete the beforelast point of L_{sup}, and recheck the three last points – until there is a right turn (or only two points in L_{sup})
- Same idea for L_{inf}

Convex hull of a set of points

Algorithm :

```
ConvexeHull($P$, $L$)
Input : a set of points P in the euclidean plane
Output : an ordered list $L$ of points of CH($P$) , in the the clockwise order
{
  Sort all the points of P in increasing $x$ order
  Insert $p_1$ et $p_2$ in this order in $L_{sup}$
  For $i$=3 to $n$
  {
    Ad $p_i$ to $L_{sup}$
    While
          $L_{sup}$ contains more than 2 points
          AND the 3 last points do not “turn right”
    {
     Delete the beforelast point of Lsup
    }
  }
```

>>>see next

Convex hull of a set of points

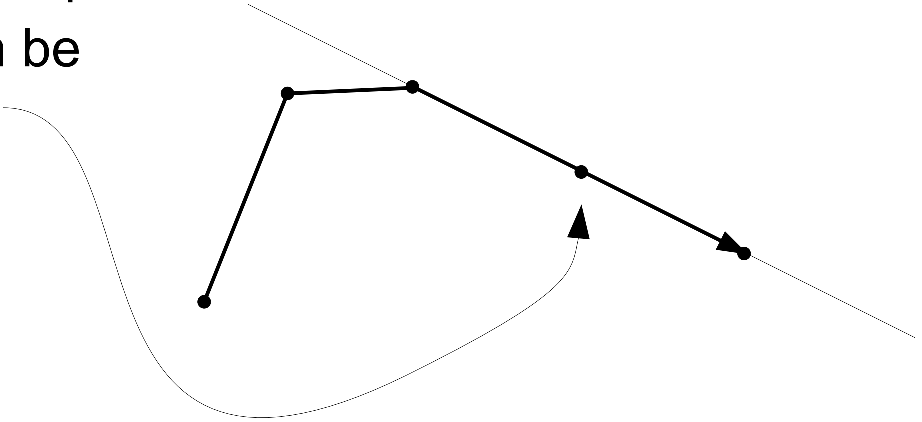
Algorithm (sequel):

```
Insert $p_n$ et $p_{n-1}$ in this order in the list $L_{inf}$
For $i=n-2$  to 1
{
  Add $p_i$ to $L_{inf}$
  While
          $L_{inf}$ contains more than 2 points
          AND the 3 last points do not “turn right”
  {
    Delete the beforelast point of $L_{inf}$
  }
}
Delete the first and last point of  $L_{inf}$
Concatenate $L_{sup}$ then $L_{inf}$ in this order into  $L$.
}
```

CAD & Computational Geometry

Convex hull of a set of points

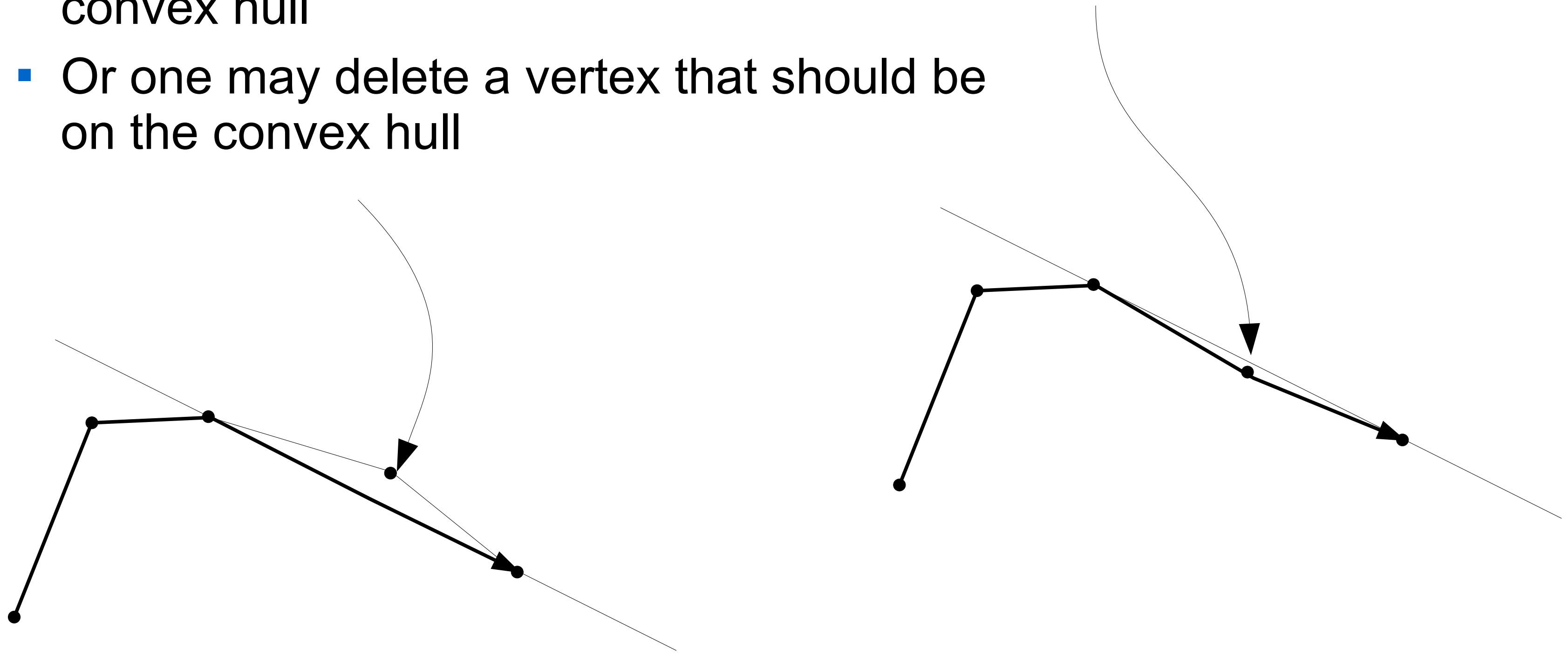
- What happens when there are points with the same x coordinate?
  - Need to sort this out. If same x , compare with y !
    The is called a *lexicographical* sort.
- What happens when 3 points are collinear ?
  - One shoud keep only “external” points
    The “right turn” test must then be considered **false**.
  - With these small changes the algorithm handles special cases correctly.



CAD & Computational Geometry

Convex hull of a set of points

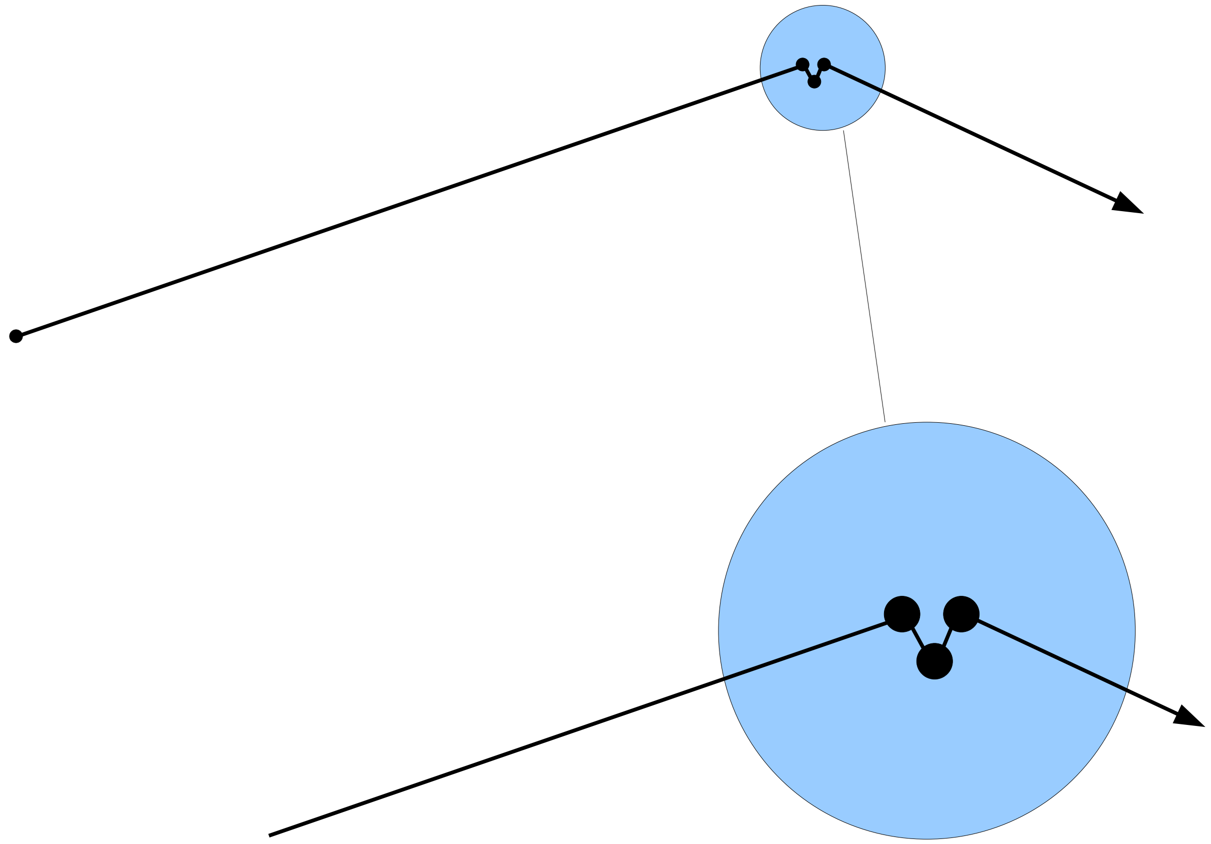
- Now, what happens if rounding errors are such that the “right turn” test is wrong ?
  - One may fail to delete a vertex that is somewhat inside the convex hull
  - Or one may delete a vertex that should be on the convex hull



  - In every case, the (approximate) convex hull that is computed here is coherent : a closed polygon, described in clockwise order, where every turn is a “right turn” – as seen from the computer’s limited arithmetic capabilities.

Convex hull of a set of points

- There also exists the case where a sharp left turn is seen as a right turn when the three points are very close to each other
  - Solution : avoid this by merging the very close points (buy rounding)



Convex hull of a set of points

- Analysis of the algorithm and complexity
  - For L_{sup} : (identical results for L_{inf})
    - The main loop is executed n-3 times
    - The predicate is evaluated an unknown number of time at each iteration of the main loop
    - However, the TOTAL sum of vertices withdrawn from L_{sup} for all iterations cannot exceed n !
    - As a consequence, this part of the algorithm is linear in $O(n)$.
  - Merging the two lists is also in $O(n)$
  - The sorting at the beginning is usually done in $O(n \log n)$
  - As a consequence, the average complexity is $O(n \log n)$.
- It is not optimal. The optimal complexity is $O(n \log h)$, h being the size of the output (number of vertices of the convex polygon (an output sensitive algorithm !)

  Chan, Timothy M. (1996). "Optimal output-sensitive convex hull algorithms in two and three dimensions". Discrete & Computational Geometry. 16 (4): 361–368. doi:10.1007/BF02712873.

Convex hull of a set of points

- How to design a geometrical algorithm
  - First understand the geometrical nature of the problem, avoiding other distractions that may bring complexity (special and degenerated cases, etc...)
  - Then, once a “good” algorithm has been devised, bring in the special cases. It is always better to modify the algorithm so that it handles directly those special cases, instead of, well, having explicit special cases in the code.
  - Then comes the implementation. One needs robust predicates, and the algorithm may be again adjusted so that it handles nicely cases where the predicates are failing (because of rounding).
    - Cf . second algorithm for the convex hull.

Convex hull of a set of points

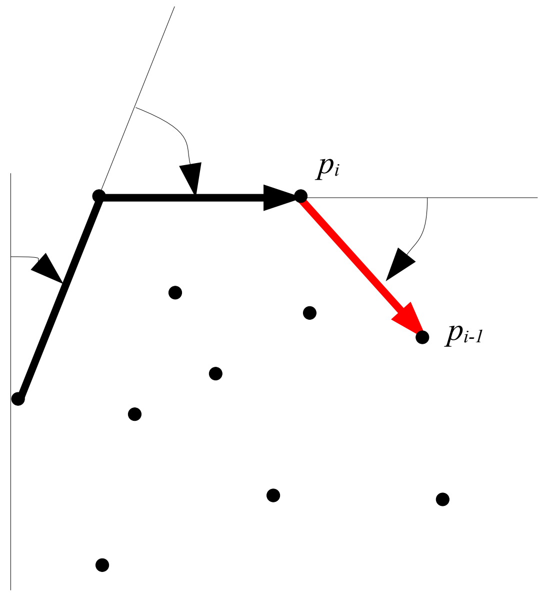
- Your turn : analysis of Jarvis's algorithm

Jarvis, R. A. (1973). "On the identification of the convex hull of a finite set of points in the plane". *Information Processing Letters* 2: 18–21.

  - Complexity ?
  - Nature (incremental or not, randomized ...)
  - Robustness ?

Convex hull of a set of points

- Jarvis Algorithm : From Pi, sweep through all other vertices and find the one having the smallest angle.



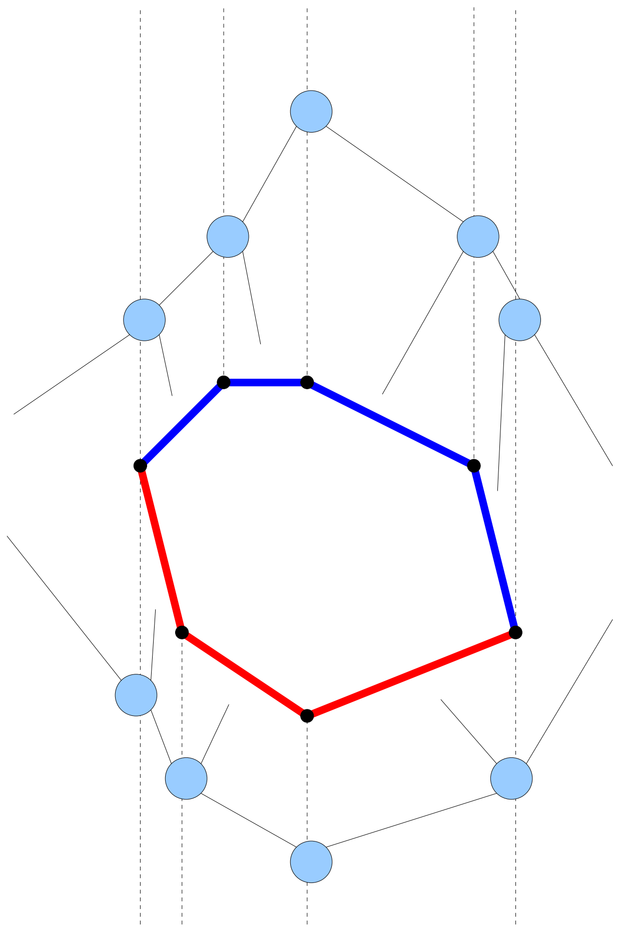
Convex hull of a set of points

- Now, what about an *incremental* algorithm?
  - That means we want to be able to insert a given point P_i , knowing the convex hull of the i-1 preceding points.
    - That is the first step toward *dynamic algorithms*, allowing the deletion a point at any time...
  - There exists an efficient implementation :
    - Idea : being able to locate efficiently the new point so that the update of the convex hull is also efficient.

CAD & Computational Geometry

Convex hull of a set of points

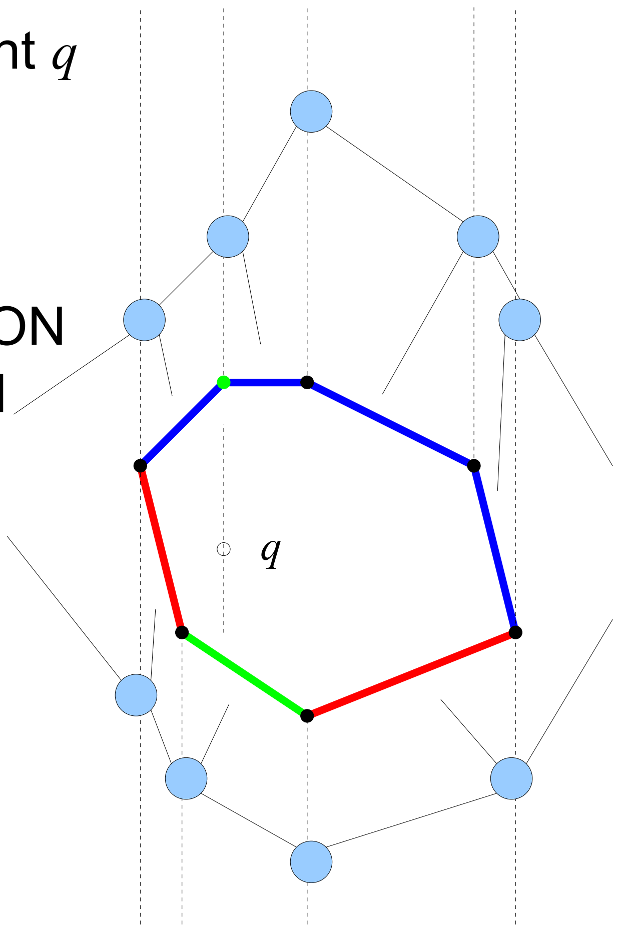
- One stores the upper and lower branch of the convex hull into balanced binary trees...



Convex hull of a set of points

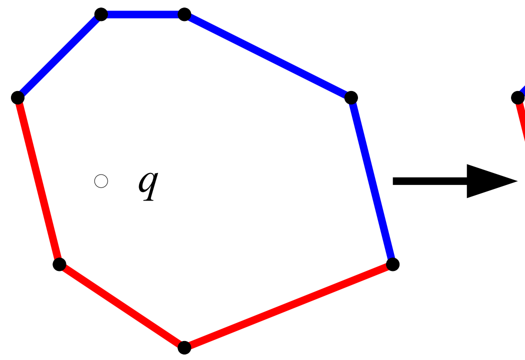
- One stores the upper and lower branch of the convex hull into balanced binary trees...
  - It is then easy to fin which segment immediately above and below the point q that is to be inserted.

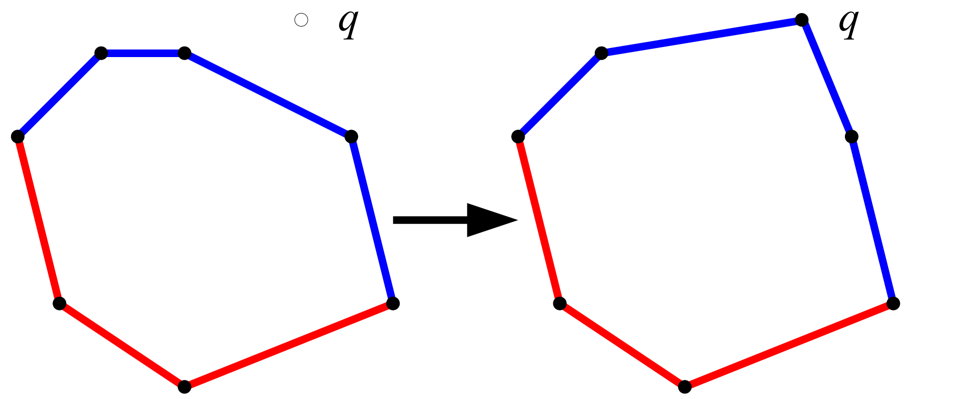
    4 cases :
    - No such elements : OUT
    - If q is on one of these elements : ON
    - If q is above the element "inf" and below the element sup : IN
    - Otherwise : OUT
  - Depending on these cases, one will insert the point q into the convex hull (or not).



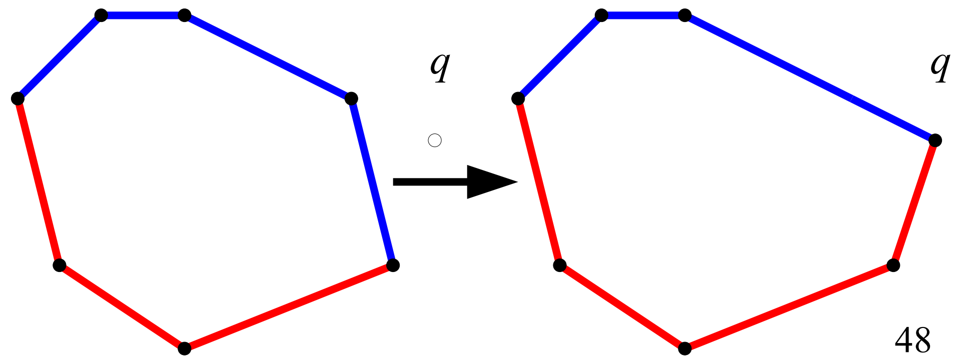
Convex hull of a set of points

- Depending on the case ...
  - Case IN : q is put aside





  - Case OUT and above : Insertion in the superior chain
  - Case OUT and below : insertion in the inferior chain



  - Case OUT and neither above or below : insertion in both chains (at the extremity)

Convex hull of a set of points

- Insertion in one of the chains (here the superior chain)
  - Find the edge e (of the vertex v) for which the superior extension contains q

    w is the vertex to the left of e or v

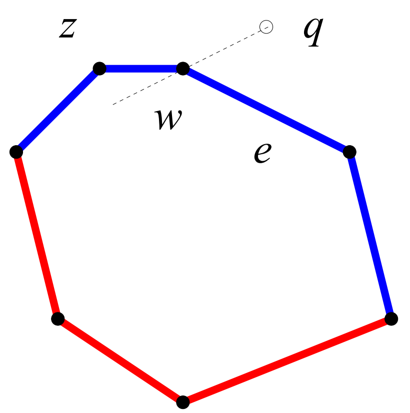
    z is the vertex to the left of w
  - While orientation(q,w,z) is in clockwise order (or collinear)

    Delete w

    w=z

    z= right neighbor of w
  - Same thing on the other side (to the right)...
  - Add q inside the chain.

    (NB. For lisibility, I do not talk about special cases, e.g. at the extremities...)



CAD & Computational Geometry

Convex hull of a set of points

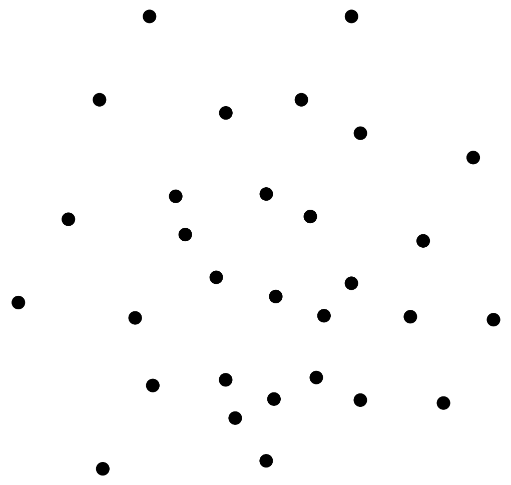
- Complexity analysis (left as an exercise)

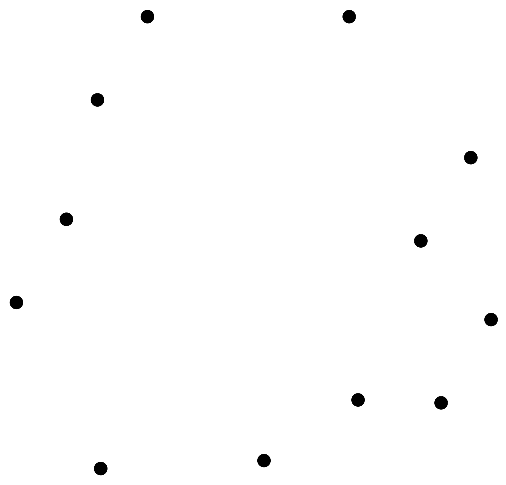
Convex hull of a set of points

- Tools : balanced binary tree (e.g. Red and Black tree)
- Implementation available e.g. in the STL in C++ …
- This is a dynamic data structure with optimal complexity bounds concerning insertion and deletion (logarithmic...)

Convex hull of a set of points

- What happens when the data have a peculiar structure ?
  - One algorithm, which performs less well than optimal algorithms (on the worse case) may be well above for such special data
- Sometimes, testing is the only way to choose the right algorithm for the right data.


This case is common


This one is very rare (worse case)

Line intersections

Line intersections

Line intersections

Classical problem in GIS

Example :

One has access to several independent maps :

- Forestry
- Wildlife
- etc...

One wishes to know every homogeneous zones for every tuple of characteristics :

- e.g. deciduous forest & deer
  pine forest & deer
  pine forest & bears … and so forth.

Line intersections

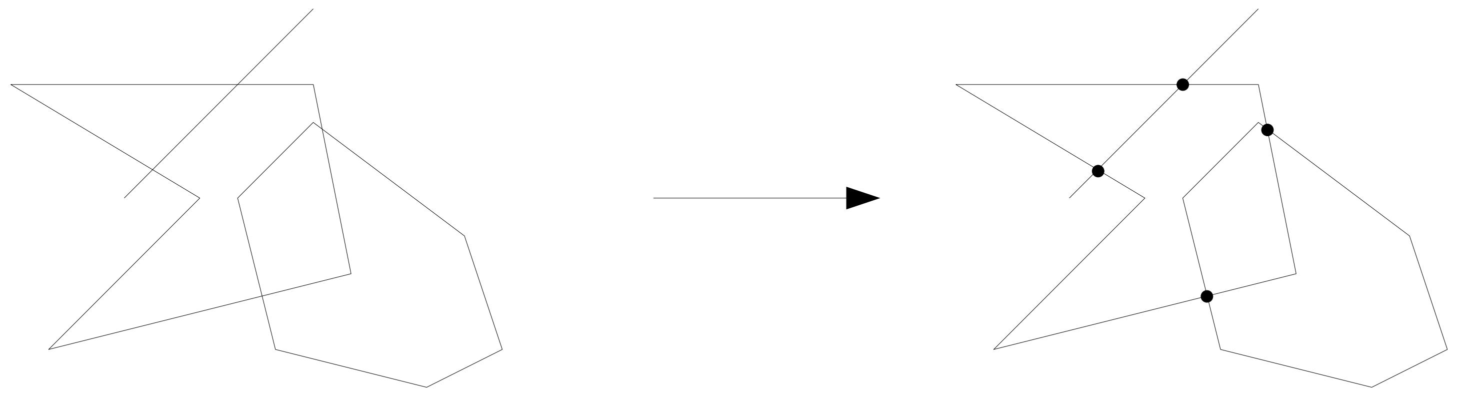
- Types of data that are contained in maps :
  - Towns (as points or polygons)
  - Rivers, highways (networks, graphs)
  - Wooden areas, or of a given climate (polygons)
- One needs to combine 2 or more of these maps, therefore compute intersections
  - Two maps with networks → points (bridges ?)
  - Two maps with areas (zones) → new polygons
  - Any combination...
- One common task (elementary task) is the intersection between two sets of line segments.

Line intersections

- Line segment intersection

  Input data : Two sets of line segments

  Output : Every intersection of segments of both sets
- On may, without loss of generality, merge the two sets and compute the intersections from inside one lone set..
  - It is easy to find the intersection between elements of the same initial set : this may be filtered afterwards.



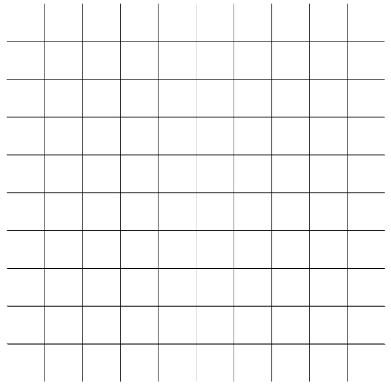
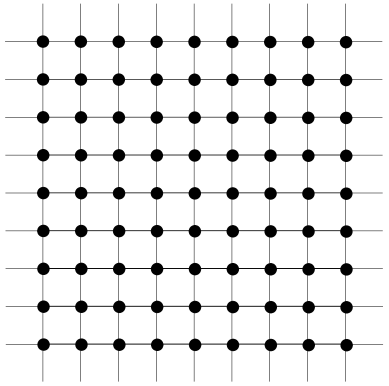
Line intersections

- Algorithm 1
  - Brute force : take each pair of segment, and check if they intersect each other

    $O(n^2)$!!!
  - In some sense, this is optimal if a “high” number of segments do intersect in many places (also in $O(n^2)$)
    - The algorithms computing this are necessary in $\Omega(n^2)$

example :

$2n$ segments
n^2 intersections

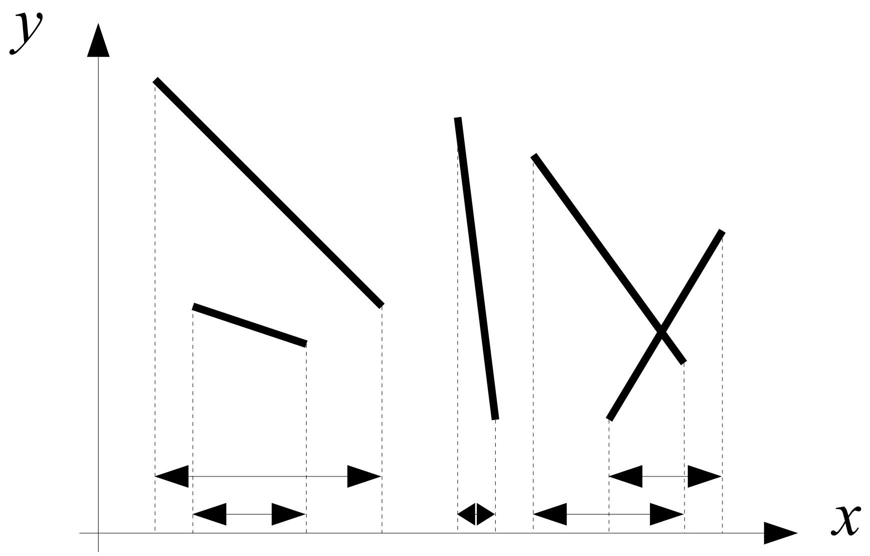



Line intersections

- In practice, the number of intersections is generally not in $O(n^2)$. The algorithme is therefore sub-optimal *in that case.*
- One need to design a better algorithm
  - Ideally, $O(f(n,I))$ with $f(n,I)$ better than $O(n^2)$ when I is $O(n)$
  - Here, the expected complexity is depending on the input data (i.e. the cardinality of the input set, n) , but also on the cardinality of the output set (here, number of intersections I)
  - This is a case of algorithm that is sensitive on the output data – this was not the case with the convex hull, for which the cardinality of the output data was at most that of the input data.
- How to avoid testing every pair of segments ?
  - Make use of the geometric nature of the problem !

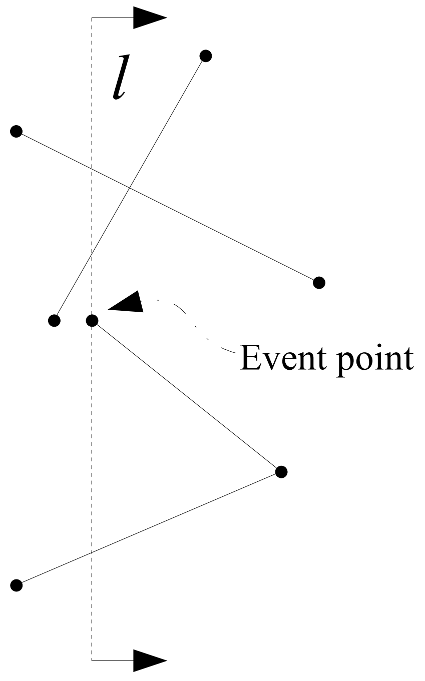
Line intersections

- Let $S=\{s_{1,}s_{2,}s_{3,}\ldots,s_n\}$, the set of all line segments.
- One may test only segments which have a non disjoint projection onto the x axis



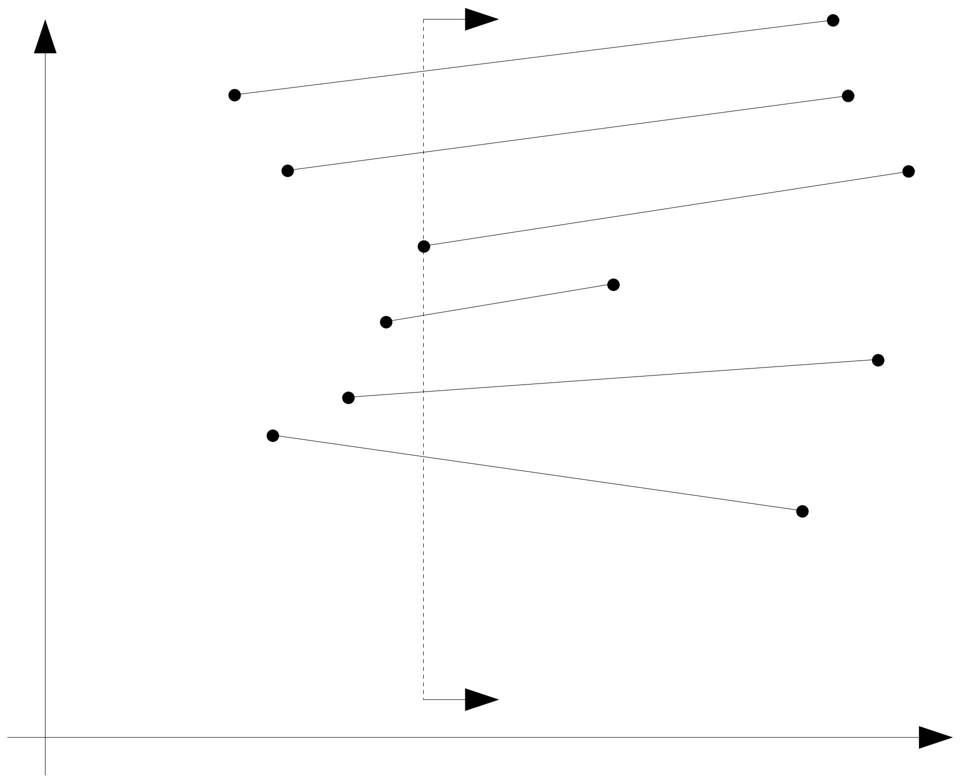
Line intersections

- In order to detect such pairs of segments that are non x disjoint, lets define an imaginary line l sweeping the domain from left to right
  - The **status** of the line l is the set of segments that intersect it
    - It changes as the line moves
    - For each **event** ; the status is updated
      It is the only time we "do" something : add a segment, perform some intersection tests, remove a segment.
    - If an event corresponds to a left extremity, the we add the segment; we have to test the intersection with all the segments that are already present
    - If an event is a "right" extremity, the segment has to be deleted.



Line intersections

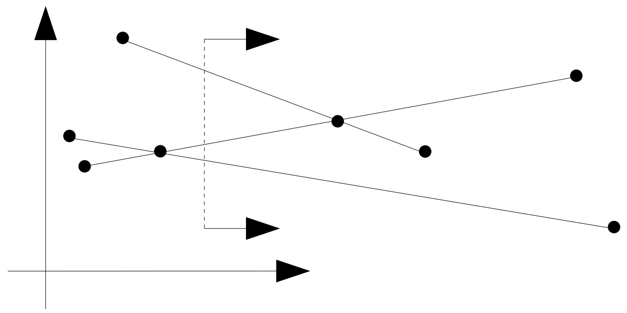
- One tests all segments present in the status. Is that optimal ?
  - No !



Here, a quadratic number of pairs of segments are tested ...

Line intersections

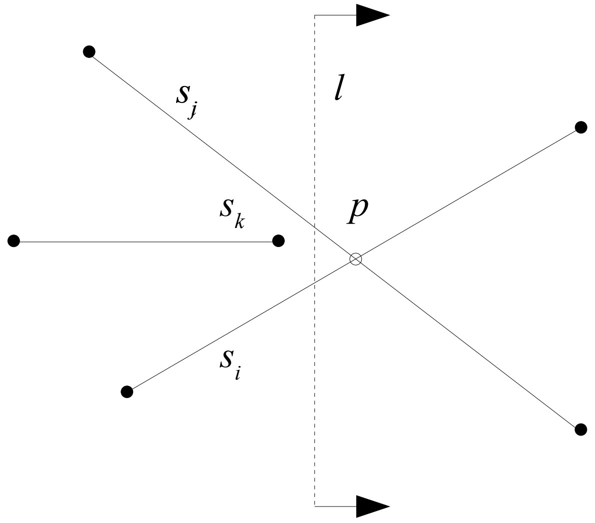
- Lets sort the segments in the status, from bottom to top
  - This allows to check whether two segments are close in the vertical direction
  - Obviously, one tests only adjacent segments in this setting.
  - When a segment is added in the status, it is tested only against the one immediately “below”, and the one “above”
  - If an event is an intersection, one needs to swap the segments... and test those with the new neighbors



  - If an event amounts to delete one segment from the status, then the segments above and below have to be tested as they become neighbors
- Therefore, every “new” adjacency is tested for a potential intersection.

Line intersections

- Does it work ?
  - One need to check that every intersection p can be computed when an event is processed.
    - Amounts to show that s_i abd s_j become adjacent before the event p is processed.

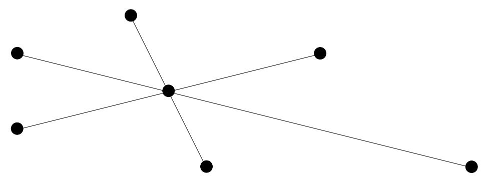


Here, s_i and s_j become adjacent when s_k is deleted from the status

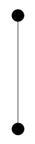
  - It is the case, because both segments are adjacent when p is processed ; but are not adjacent at the beginning (before even one of s_i or s_j has been added). There exist at least one event leading to the the new adjacency.

Line intersections

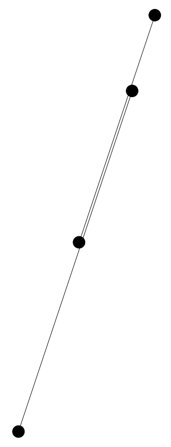
- In principle, the algorithm works, without taking care of the degenerated and special cases.
  - Intersections of 3 segments or more at the same place



  - Vertical segments



  - Overlapping segments



Line intersections

- « Invariant » of the algorithm : every intersection point to the left of the imaginary line have been correctly computed and processed.

Line intersections

- Data structures
  - The events list F
    - Sorted lexicographically in function of the coordinates of the point → the special case of vertical segments is automatically solved. Same structure as in the case of the convex hull.
    - Each event has a specific “nature”
      - “Left” point of a segment → segment added in the status
      - “Right” point of a segment → segment deleted of the status
      - Intersection point → swap the two segments
  - The status T
    - Segments are ordered along the line l
    - Allows to efficiently look for segments that are contiguous to a given event (e.g. sorted along y)
    - Difficulty : the access key (y coordinate) changes when the line moves …

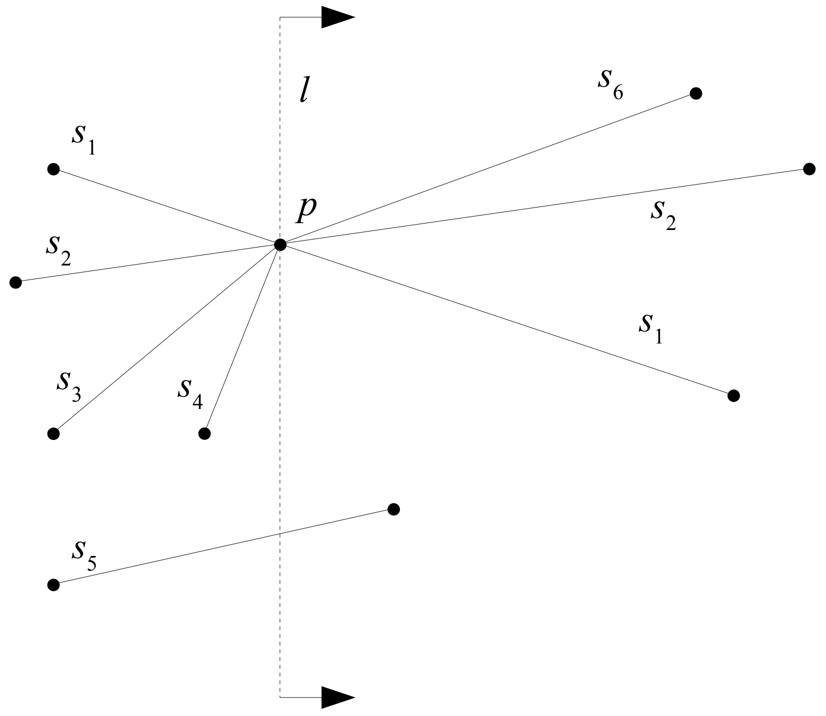
Line intersections

- The Algorithms :

```
**FindIntersections**($S$)
Input : Set of all line segments in the euclidean plane
Output : Set of all intersection points, with links to the associated line segments
{
  Initialize an empty events list $F$
  Insert the enpoints of all segments from $S$, into $F$.If the point is a left point, then a link to
the segment is attached to it .
  Initialize and empty Status $T$ .
  While  $F$ is not empty
  {
    Find the next point  $p$ in $F$, delete it from  $F$.
    **ProcessEvent**($p$,$T$)  // function call
  }
}
```

Line intersections

- ProcessEvent(p, T) must be able to respond correctly to such degenerate cases :

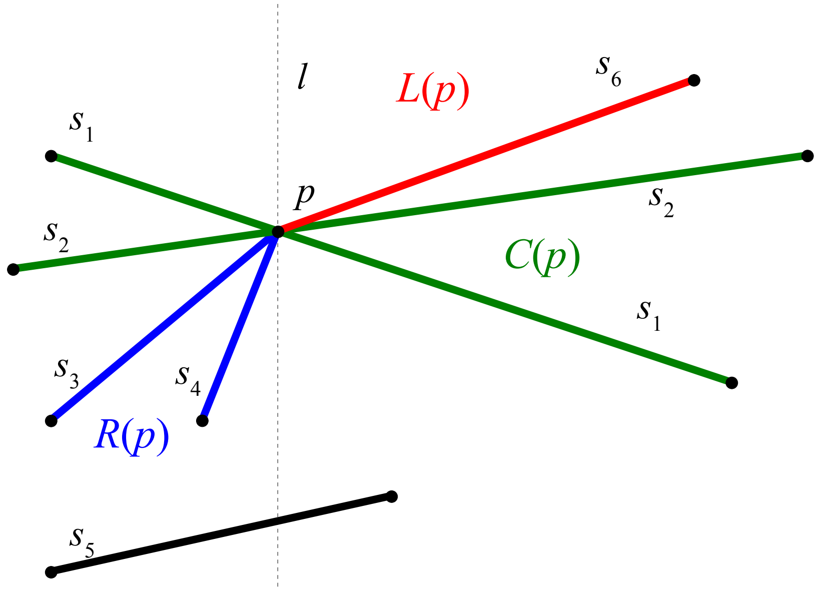


$T=\{s_5, s_4, s_3, s_2, s_1\} \rightarrow T=\{s_5, s_1, s_2, s_6\}$

CAD & Computational Geometry

Line intersections

```
ProcessEvent($p$,$T$)
{
  Let $L(p)$ the set of all segments from which the left endpoint is $p$ (those are known and stored with $p$)
  Find every segment in $T$ that contain $p$ : those are adjacent in $T$.
  Let $R(p)$ be the subset of this set for which p is the right endpoint
      $C(p)$ be the subset that contain $p$ (i.e. $p$ is in the interior of the segment, not at the end)
  If Union($L(p)$,$R(p)$,$C(p)$) contains more than one segment
    { $p$ is an intersection, link $L(p)$,$R(p)$ and $C(p)$ to it }
  Delete all segments belonging to Union($R(p)$,$C(p)$) from $T$
  Insert all segments belonging to Union($L(p)$,$C(p)$) in $T$ :
      the new ordering should correspond to the ordering when $l$ is just at the right of $p$.
      If there is a vertical segment, it comes last.
      Note : the ordering among segments in $C(p)$ is reversed...
  If Union($L(p)$,$C(p)$) is empty
  {
    Let $s_u$ and $s_d$ the neighbors above and below $p$ in $T$
    FindEvent($s_u$, $s_d$, $p$)
  }
  Else
  {
    Let $s'$ the highest segment in Union($L(p)$,$C(p)$)
    Let $s_u$ the above neighboring segment to $s'$ in $T$
    If $s_u$ exists FindEvent($s_u$, $s'$, $p$)
    Let $s''$ the lowest segment in Union($L(p)$,$C(p)$)
    Let $s_d$ the below neighboring segment in to $s''$ in $T$
    If $s_d$ exists FindEvent($s''$, $s_d$, $p$)
  }
}
```



CAD & Computational Geometry

Line intersections

```
**FindEvent**($s_1$, $s_2$, $p$)
{
 If $s_1$ and $s_2$ do intersect to the right of the imaginary line $l$, or on the line but above $p$
 And if the intersection is not already present in $F$
   Insert the intersection as a new event in $F$. Attach the lines to the new entry.
}
```

Line intersections

- Analysis of the algorithm

  Does it find all intersections ?

  - Poof by induction on the priority of the events
    - One suppose that all events with higher priority than p in the file are correctly processed.
    - 1st case : p is an extremity of one of the segments
      It has been inserted into F at the beginning and therefore it is the event file, with $L(p)$, and $R(p)$ et $C(p)$ are in T when this event is processed.
    - 2nd case : p is an intersection, need to prove that p has been introduced in F some time before.
      - Here, all segments that are concerned have p in their interior (not at the extremities : this is case 1)

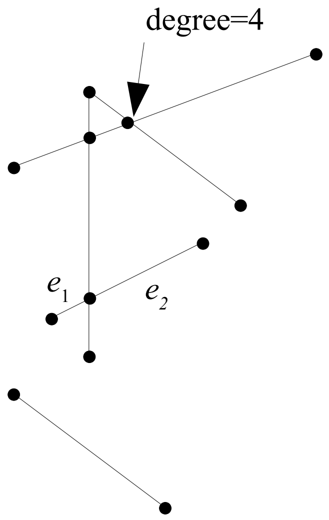
      Let s_i and s_j two neighboring segments in T. The previous proof (slide 63) allows to make sure that these segments become neighbors at one point, and are tested for intersection and p computed at a certain event q with higher priority than p.

Line intersections

- Performance (in execution time) of the algorithm
  - It can be proved that $t=O((n+k)\log n)$ with k = size of the output
    - Building the event list : $O(n \log n)$
    - Processing of each event
      - Insertions / deletions in F : $\log n$ each
      - Insertions / deletions in T : $\log n$ each (worst case), but there are less : let $m(p)=\mathrm{Card}(\mathrm{Union}(L(p), R(p), C(p)))$

    Let m be the sum of all $m(p)$, globally one have $O(m \log n)$

    Since $m=O(n+k)$, k being the output size (segments + intersections) one gets an $O((n+k)\log n)$ complexity.
  - A stronger result can be proven : $T=O((n+I)\log n)$ with I = number of intersection → considerations on planar graphs
    - m is bounded by the sum of the degrees of each vertex
    - Each edge contribute to the degree to maximum two vertices, so m is bound by $2n_e$ (number of edges of the graph). n_v (number of vertices) is at most $2n+I$. Because of the Euler relation on a planar graph, $n_e = O(n_v)$, QED.



Line intersections

- Euler relation … for planar graphs
  - Every face in the graph is bound by at least three edges
  - Each edge bounds at most two distinct faces
  - Therefore, $n_f \leq 2n_e/3$
  - Euler relation : $n_v - n_e + n_f = r$ with $r \geq 2$
    - r depends on the topological structure (nb of holes etc.). Here, this is a constant.

      One has therefore $n_e = O(n_v)$.

Line intersections

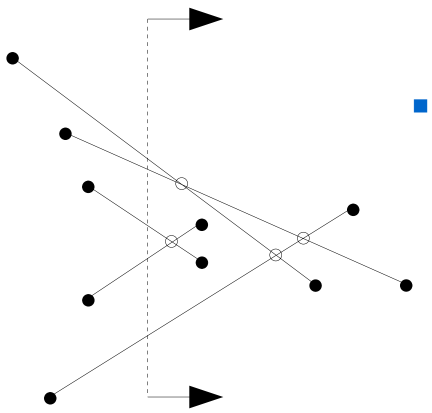
- Memory performance of the algorithm
  - T stores a most n segments , in a binary tree $\rightarrow O(n)$
  - The list F stores at most $2n + I$ events $\rightarrow O(n + I)$
  - So, finally, $m{=}O(n{+}I)$

    If $I = O(n^2)$, it is not very efficient. One does not always need to store all intersections at once. One may process them one after another (without storage), in that case it's catastrophic.

It is possible to do better :

  - Make sure that F contains only events (intersections) corresponding to segments that are adjacent in T.
  - Therefore, as soon as two segments are no more adjacent in T, the corresponding event must be removed from F.
    - Such an event may be added and removed multiple times before being processed.

      However, it never happens more than n times at all, so it does not affect the bounds of the execution time $t{=}O((n{+}I) \log n)$
    - The list F therefore contains only $O(n)$ elements at any given time.



CAD & Computational Geometry

Line intersections

Briefly said :

- It is possible to compute intersections in a time $t=O((n+I) \log n)$ and a memory footprint $m=O(n)$

  This algorithm dates back to 1979 (with a later modification to keep memory imprint in $O(n)$)

  J. L. Bentley and T. A. Ottmann , *Algorithms for reporting and counting geometric intersections*, IEEE Trans. Comput., C-28:643–647, 1979

- Is that optimal ?

  No ! ... case where $I=O(n^2)$: $t=O(n^2 \log n)$, or it is possible to achieve the same result in $O(n^2)$ - using the naive brute force algorithm !!!

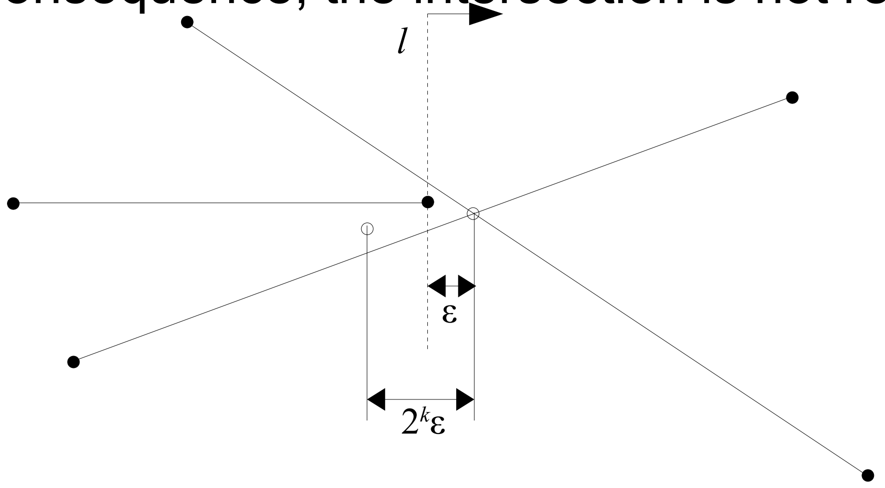
  - The theoretical lower bound is $T=\Omega(n \log n + k)$ – and there exists a deterministic algorithm since (only) 1995

    I. J. Balaban. *An optimal algorithm for finding segment intersections*. In Proc. 11th Annu. ACM Sympos. Comput. Geom., pages 211–219, 1995.

CAD & Computational Geometry

Line intersections

- Robustness ?
  - As such, the algorithm is not robust because of the floating point computations of intersections in finite precision...
  - It is possible that the (inaccurate) computation of an intersection gives a point that is slightly to the left of l , but that this point has never been reported before (had never been in F.)
  - As a consequence, the intersection is not reported...



Line intersections

- Solutions for robustness
  - Work with exact input data (e.g. integer coordinates), and compute intersections exactly as rational numbers
    - May slow down quite heavily the computation
  - Increase the precision of the numerical evaluations
    - A naïve implementation (that is, as it is shown here) of Bentley & Ottmann's algorithme impose computations on $5n$ bits for a result accurate on n bits

      Boissonat & Preparata have shown that one may use only $2n$ bits for the same final n bits by carefully arranging computations.

      See Boissonat, J.-D.; Preparata, F. P. (2000), *Robust plane sweep for intersecting segments*, SIAM Journal on Computing 29 (5): 1401–1421

CAD & Computational Geometry

Floating point issues

- Numerical errors
  - Floating point calculations are (most of the time) inaccurate
  - Analysis of the rounding done during operations in floating point :

$$x \ominus y = (x - y)(1 + \delta_1) \ , \ |\delta_1| \leq 2\varepsilon$$
$$x \oplus y = (x + y)(1 + \delta_2) \ , \ |\delta_2| \leq 2\varepsilon$$
$$x \otimes y = (xy)(1 + \delta_3) \ , \ |\delta_3| \leq \varepsilon$$

  - Mathematically equivalent calculations but expressed differently give distinct results
  - An example with $x^2 - y^2 = (x + y)(x - y)$

CAD & Computational Geometry

Floating point issues

- Error made with the expression $(x+y)(x-y)$

$$
\begin{aligned}
(x\oplus y)\otimes(x\ominus y) &= (x-y)(1+\delta_1)(x+y)(1+\delta_2)(1+\delta_3)\\
&= (x+y)(x-y)(1+\delta_1+\delta_2+\delta_3+\delta_1\delta_2+\delta_2\delta_3+\delta_1\delta_3+\delta_1\delta_2\delta_3)\\
&\approx 5\varepsilon
\end{aligned}
$$

- No catastrophic increase of the relative error
- Error made with the expression x^2-y^2

$$
\begin{aligned}
(x\otimes x)\ominus(y\otimes y) &= [x^2(1+\delta_1)-y^2(1+\delta_2)](1+\delta_3)\\
&= ((x^2-y^2)(1+\delta_1)+(\delta_1-\delta_2)y^2)(1+\delta_3)\\
&= ((x^2-y^2)(1+\delta_1+\delta_3+\boxed{(\delta_1-\delta_2)y^2}+\delta_1\delta_3+(\delta_1-\delta_2)y^2\delta_3))
\end{aligned}
$$

- When x is close to y, the error can be of the order of magnitude of the calculated result...

Floating point issues

- Some useful rules (not a comprehensive list !)
  - Prefer $(x+y)(x-y)$ to x^2-y^2
    - Lagrange form more accurate than horner's scheme …
  - E. g. sum of many terms $S=\sum_{j=1}^{N} X[j]$
    - Naive algorithm :

```
S=0;
for (j=1;j<=N;j++){ S=S+X[j] ; }
return S ;
```

involves an error $\approx N\varepsilon$

- Kahan's summation algorithm

```
S=X[1];C=0
for (j=2;j<=N;j++)
 { Y=X[j]-C; T=S+Y; C=(T-S)-Y; S=T }
return S ;
```

involves an error $\approx 2\varepsilon$

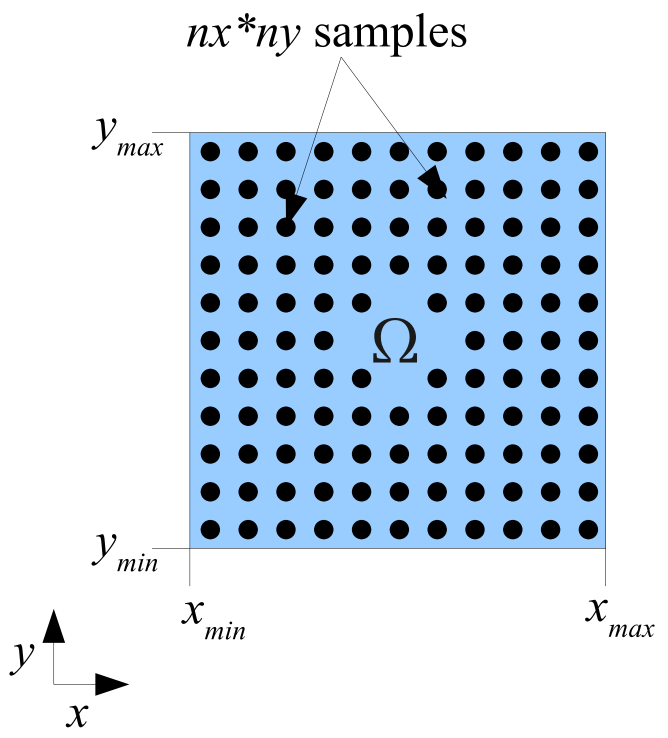
Floating point issues

- Example of catastrophic rounding
  - Computation of an integral :

$$S=\int_{\Omega} f(x,y)\,dxdy \text{ with } f(x,y)=x^2+y^2$$

$$S\approx\sum_{i=0}^{nx-1}\sum_{j=0}^{ny-1} f(x(i),y(j))\,det\,J$$

$$dx=(x_{max}-x_{min})/nx$$
$$dy=(y_{max}-y_{min})/ny$$
$$x(i)=x_{min}+i\,dx+dx/2$$
$$y(j)=y_{min}+j\,dy+dy/2$$
$$det\,J=dx\,dy$$



CAD & Computational Geometry

Floating point issues

- Computations made with the following parameters:

  $x_{min} = y_{min} = 0.0$; $x_{max} = y_{max} = 1.0$; $nx = ny = 10$, $\mathbf{S_{exact}=2/3}$

  1) Single precision floating point numbers
  2) Double precision floating point numbers
  3) Quad precision floating point numbers
  4) Single precision floating points numbers with Kahan's summation algorithm

```
bechet@yakusa:floating_error$ ./test 10
1) sum (float  )=0.66500002145767211914
2) sum (double )=0.66500000000000025757
3) sum (ldouble)=0.66499999999999999997
4) sum (kahan  )=0.66499999999999999997
```

Note : Program should be compiled without optimization !

Floating point issues

```
bechet@yakusa:floating_error$ ./test 100
1) sum (float  )=0.66664981842041015625
2) sum (double )=0.66665000000000051994
3) sum (ldouble)=0.66665000000000000093
4) sum (kahan  )=0.66664993762969970703

bechet@yakusa:floating_error$ ./test 1000
1) sum (float  )=0.66668075323104858398
2) sum (double )=0.66666649999999805232
3) sum (ldouble)=0.66666649999999999982
4) sum (kahan  )=0.66666668653488159180
```

CAD & Computational Geometry

Floating point issues

```
bechet@yakusa:floating_error$ ./test 10000
1) sum (float  )=0.36836880445480346680
2) sum (double )=0.66666666499985449690
3) sum (ldouble)=0.66666666499999997623
4) sum (kahan  )=0.66666567325592041020

bechet@yakusa:floating_error$ ./test 100000
1) sum (float  )=0.00390625000000000000
2) sum (double )=0.66666666665538221181
3) sum (ldouble)=0.66666666665000055245
4) sum (kahan  )=0.66666662693023681641
```

Compensated summation algorithm coming from :

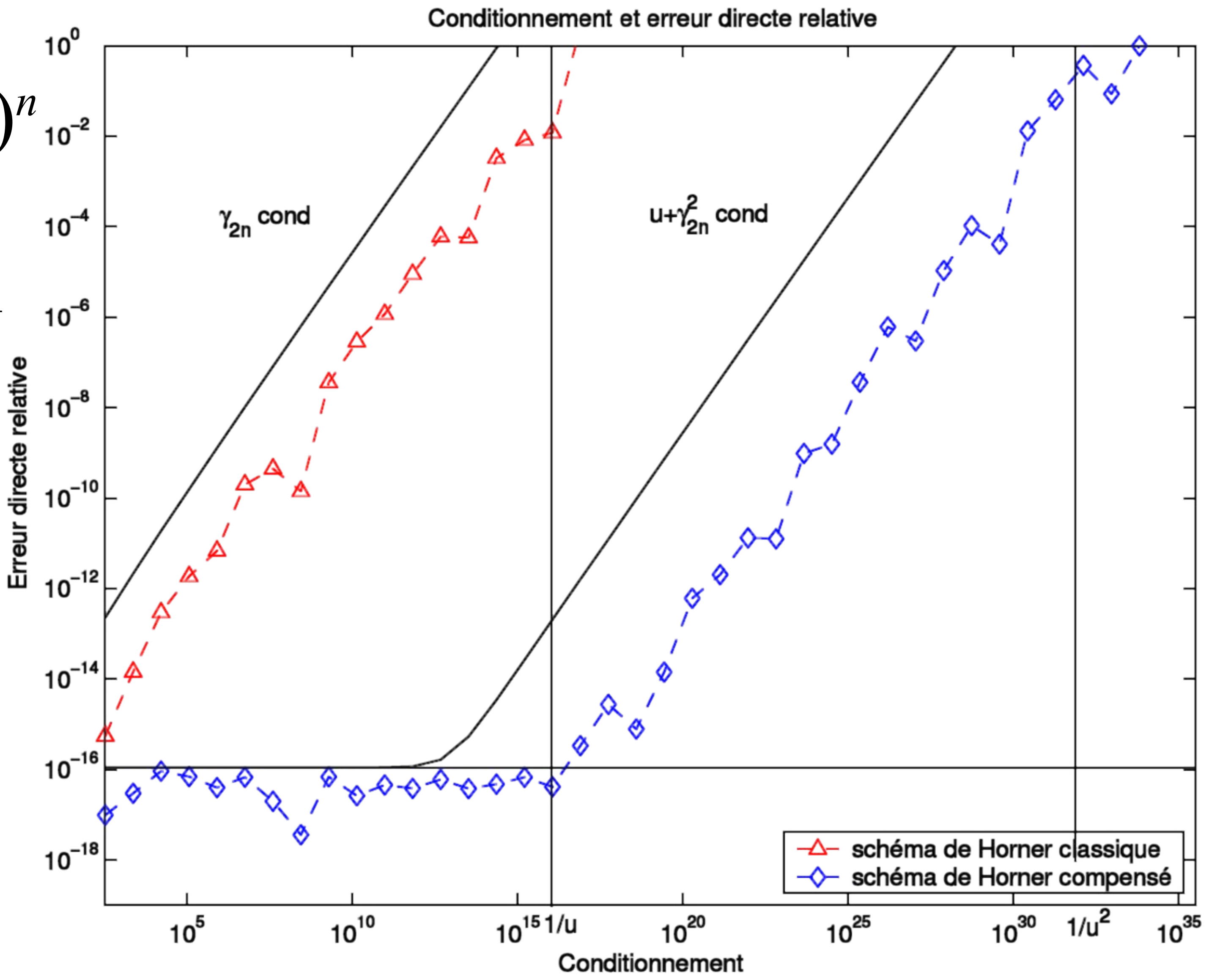
William Kahan. Further remarks on reducing truncation errors. Comm. ACM, 8(1):40,1965.

CAD & Computational Geometry

Floating point issues

$y=p(x)=(1-x)^n$
for $x=1.333$
and $2<n<41$

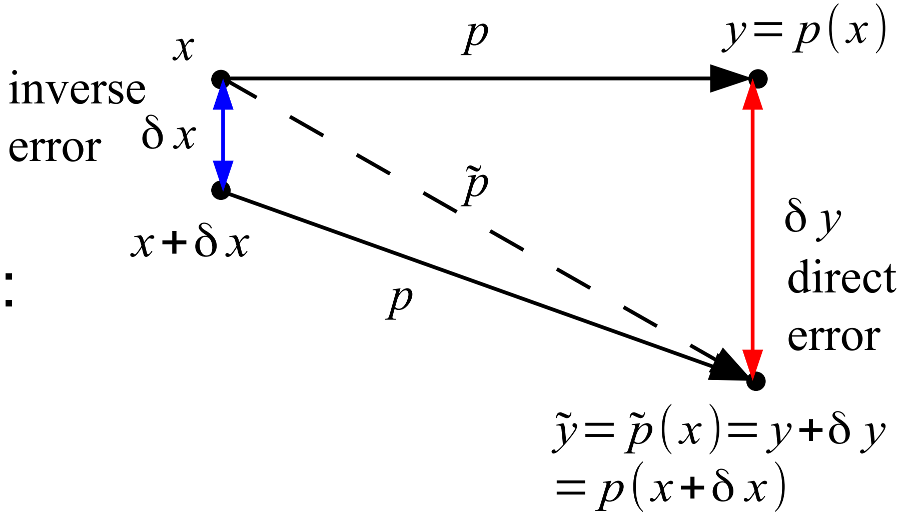
S. Graillat, Ph. Langlois, N. Louvet Compensated Horner Scheme Research Report RR2005-04, LP2A, University of Perpignan, France, july 2005



Floating point issues

- Definition of the condition number of a numerical expression
  - Ratio of the direct error to the inverse error

$$K(P,x)=\lim_{\varepsilon\to 0}\sup_{\delta x\in D(\epsilon)}\left(\frac{|\delta y|}{|\delta x|}\right)$$



  - For a polynomial under monomial form:

$$K(P,x)=\frac{\sum_{i=0}^{n}|a_i||x|^i}{\left|\sum_{i=0}^{n}a_i x^i\right|}$$

Floating point issues

- There are various compensated algorithms to carry out calculations on floating point numbers.
  - Ex. Kahan summation, Compensated Horner scheme ...
- In general, they allow to have similar results as when using internal floating point with a precision twice that of the input data, following by a final rounding.
- See references available on the course's website for the compensated Horner scheme.

Line intersections

- Line sweep algorithms

  An useful paradigm in many planar CG problems

  - Segment intersection
  - Voronoï diagrams
  - Polygon triangulation

  M. I. Shamos and D. Hoey. *Geometric intersection problems*. In Proc. 17th Annu. IEEE Sympos. Found. Comput. Sci., pages 208–215, 1976.

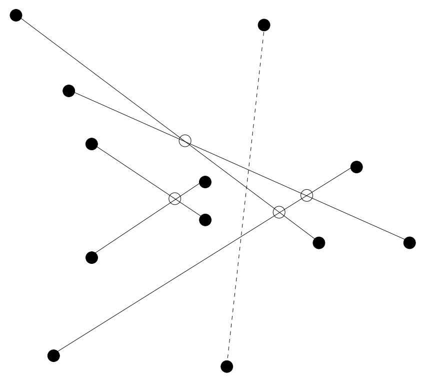
  D. T. Lee and F. P. Preparata. *Location of a point in a planar subdivision and its applications*. SIAM J. Comput., 6:594–606, 1977.

  and previous reference (J. L. Bentley and T. A. Ottmann)

Line intersections

- Exercise :

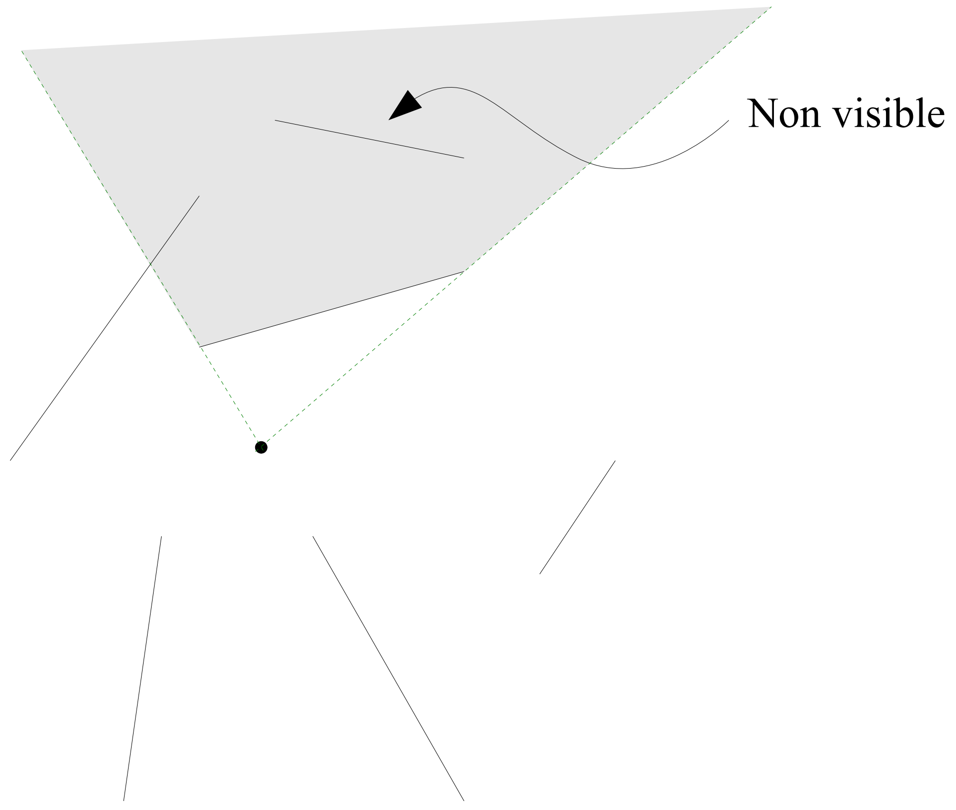
  Find an incremental algorithm for the intersection of segments ...



CAD & Computational Geometry

Sweep line paradigm

- Two other exercises
  - Find in $t=O(n\log n)$ the segments (disjoint segments) that are visible from one point..



Sweep line paradigm

- Link a set of n disjoint triangles
  - Every segment joints two triangles
  - The segments should not intersect, except at extremities) and should not intersect triangles (connected at exactly one point)