# Reconnaissance faciale par triangulation de Delaunay

Chaumeron Pauline

Candidate 19321

Session 2021

# Triangulation de Delaunay

## Théorie

Dans quelle mesure la triangulation de Delaunay peut-elle s'appliquer à la reconnaissance faciale ?

Objectifs :

- Implémenter l'algorithme naïf et l'algorithme par insertion pour obtenir la triangulation de Delaunay d'un ensemble de points.
- Implémenter l'algorithme permettant de passer du diagramme de Voronoï à la triangulation de Delaunay.
- Évaluer le pourcentage de ressemblance d'un même visage sur deux photos différentes selon des critères prédéfinis.
- Reconnaître un visage parmi différentes images.

# Plan de l'exposé

# Triangulation de Delaunay

## Théorie

**Définition**

Une triangulation d'un ensemble de points est un ensemble de triangles ne se recouvrant pas, dont l'union est l'enveloppe convexe de l'ensemble.

**Définition**

La triangulation de Delaunay d'un ensemble de points est telle qu'aucun des points ne se trouve à l'intérieur du cercle circonscrit d'un des triangles.

# Triangulation de Delaunay

## Théorie

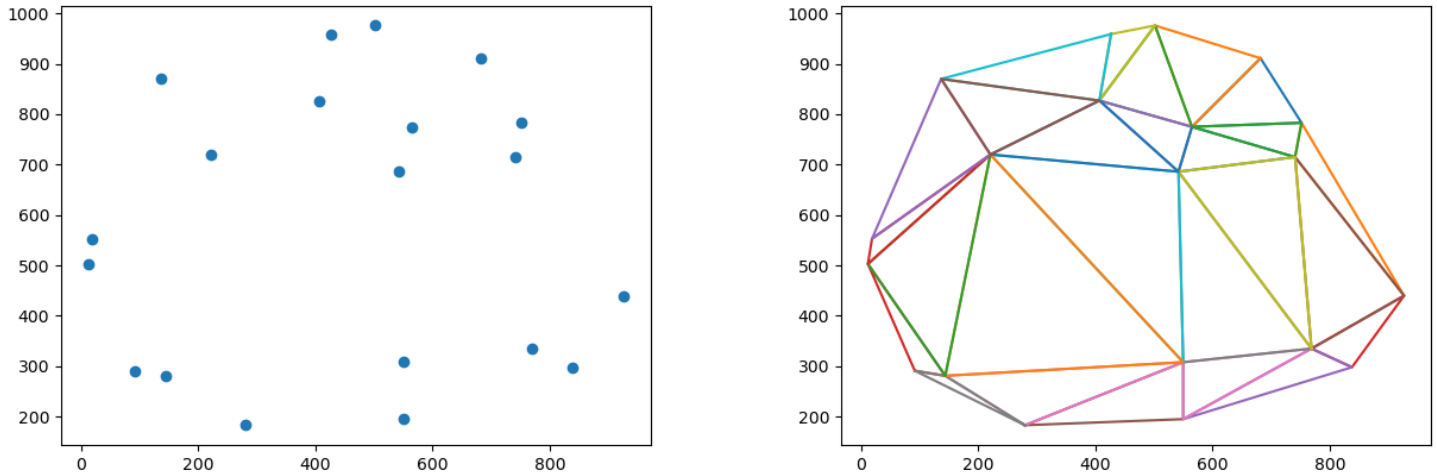



FIGURE – Triangulation d'un ensemble de 20 points

# Triangulation de Delaunay et implémentation

## Algorithme naïf

- Algorithme le plus simple à coder.
- Pas efficace pour un grand nombre de points.
- Complexité : $O(n^4)$.

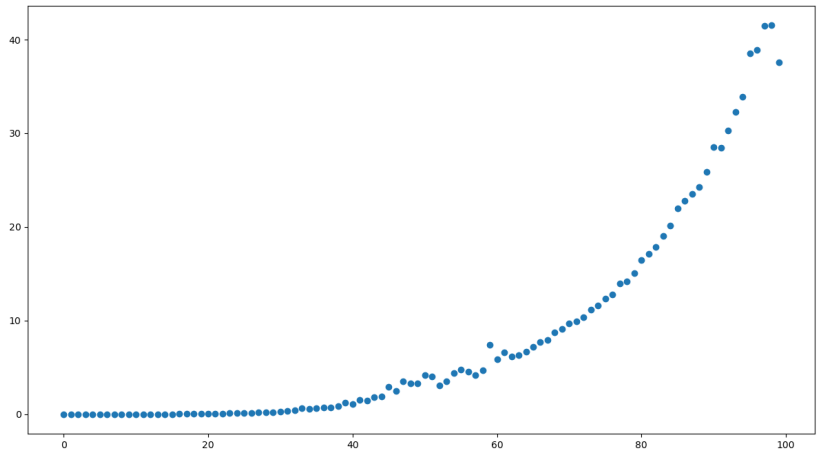



FIGURE – Temps de calcul de la triangulation d'un ensemble de points par l'algorithme naïf en fonction de sa taille

# Triangulation de Delaunay et implémentation

## Algorithme naïf

```
1  def triangulation(l):
2      triangles = touslestriplets(l)
3      tri = []
4      valide = verif(l)
5      for i in range(len(triangles)):
6          if valide[i]:
7              tri.append(triangles[i])
8      return tri
```

# Triangulation de Delaunay et implémentation

## Algorithme naïf

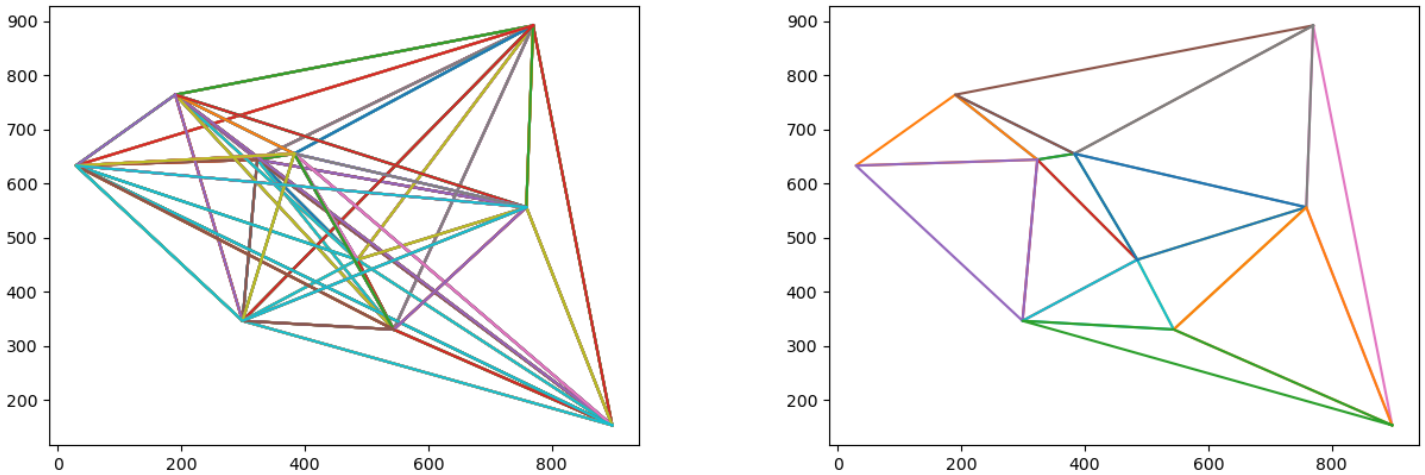



FIGURE – Triangulation par l'algorithme naïf

# Triangulation de Delaunay et implémentation

## Algorithme par insertion

- Prendre un grand triangle qui entoure l'ensemble à trianguler, insérer les points un à un.
- Complexité : $O(n^2)$.

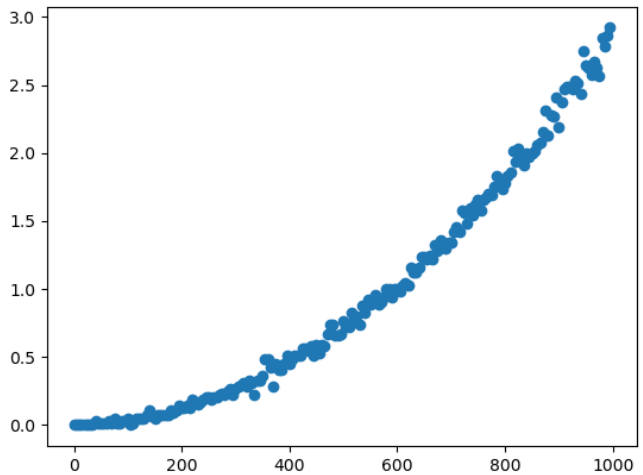



FIGURE – Temps de calcul de la triangulation par insertion d'un ensemble de points en fonction de sa taille

# Triangulation de Delaunay et implémentation

## Algorithme par insertion

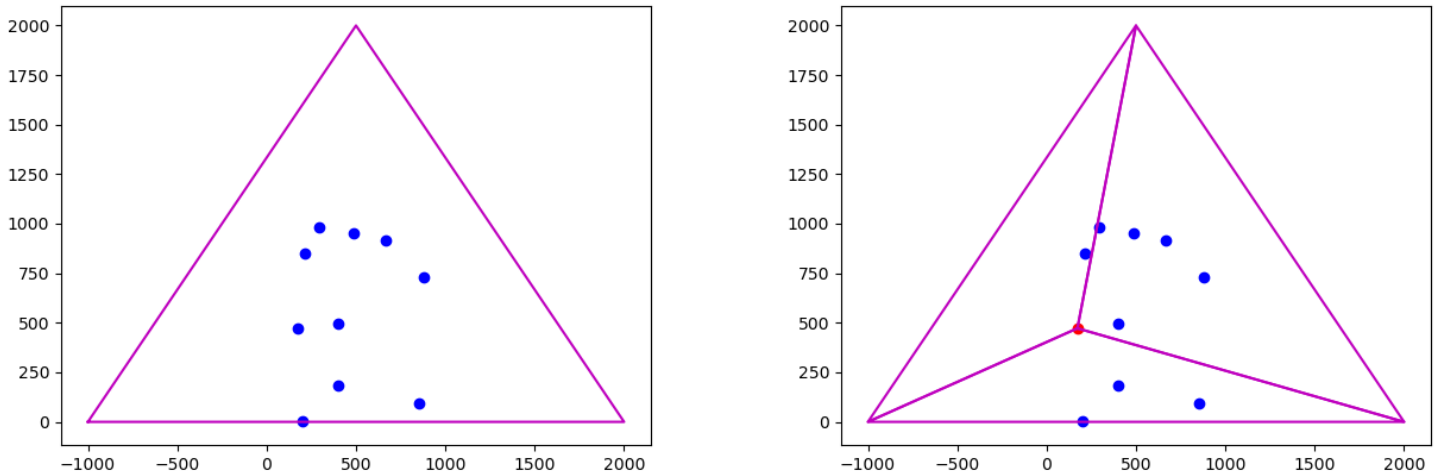



FIGURE – Triangulation par l'algorithme par insertion

# Triangulation de Delaunay et implémentation

## Algorithme par insertion

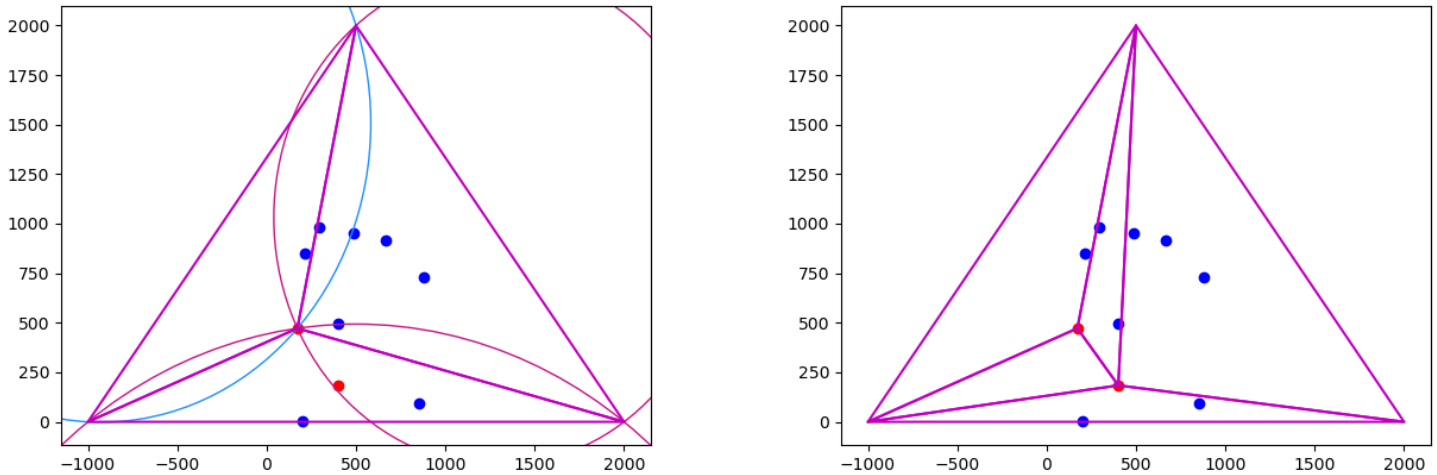



FIGURE – Triangulation par l'algorithme par insertion

# Triangulation de Delaunay et implémentation

## Algorithme par insertion

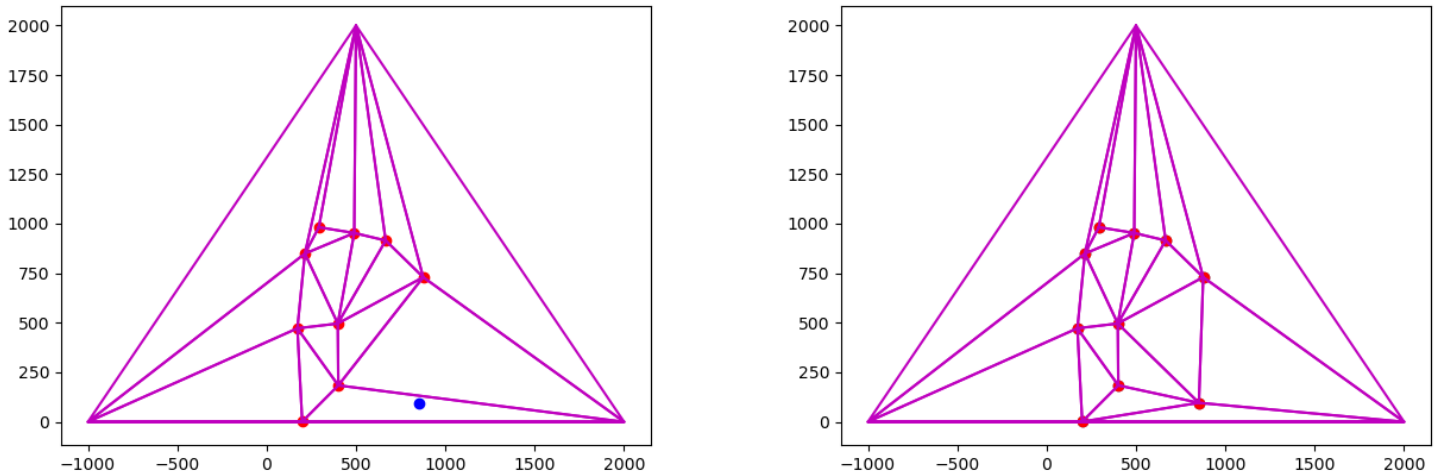



FIGURE – Triangulation par l'algorithme par insertion

# Triangulation de Delaunay et implémentation

## Algorithme par insertion

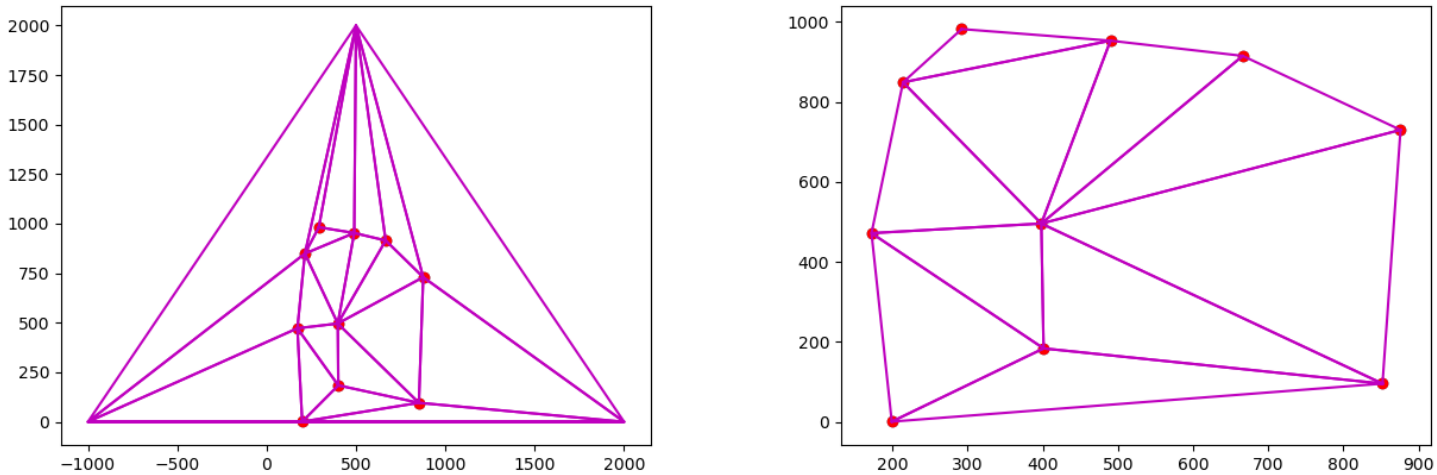



FIGURE – Triangulation par l'algorithme par insertion

# Triangulation de Delaunay et implémentation

## Algorithme par insertion

```
centre, rayon = cercles[i]
if centre == 'impossible':
  for pt_tri in tri[i]:
  if pt_tri not in p_illegal:
    p_illegal.append(pt_tri)
    c_ill.append(i)
  elif rayon - distance(centre, point)> 10**-5:
    for pt_tri in tri[i]:
      if pt_tri not in p_illegal:
        p_illegal.append(pt_tri)
        if tri[i] not in tri_ill:
          tri_ill.append(tri[i])
          c_ill.append(i)
```

# Diagramme de Voronoï

## Théorie

### Définition

Le diagramme de Voronoï est un découpage du plan en cellules à partir d'un ensemble discret de points appelés germes. Ce diagramme isole chacune des germes dans une cellule tel que chaque point d'une cellule est plus près de cette germe qu'aucun autre point du plan.

# Diagramme de Voronoï

## Théorie

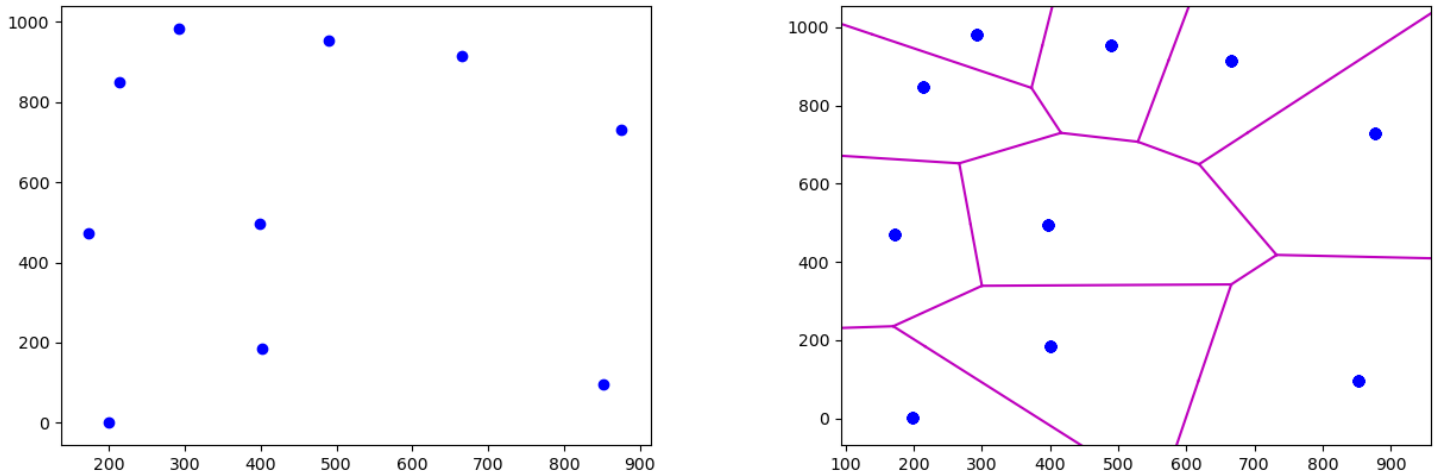



FIGURE – Diagramme de Voronoï

# Diagramme de Voronoï

## Algorithme de Fortune

- Balayement de l'ensemble par une droite verticale se déplacant de la gauche vers la droite.
- Créer le diagramme au fur et à mesure.
- Complexité : O(nlog(n))

# Diagramme de Voronoï

## Algorithme de Fortune

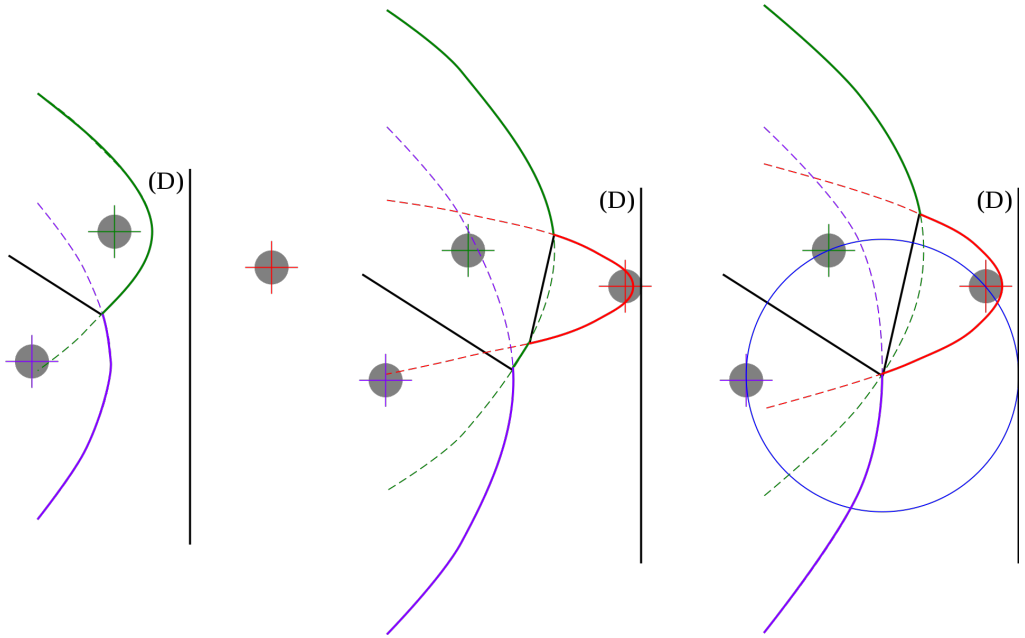



FIGURE – Types d'évènements

Par Cdang — Travail personnel, CC BY-SA 4.0,

https ://commons.wikimedia.org/w/index.php ?curid=38121712

# Diagramme de Voronoï

## Algorithme de passage

**Propriété**

On peut obtenir la triangulation de Delaunay d'un ensemble de points à partir de son diagramme de Voronoï. En effet, si deux cellules de Voronoï sont adjacentes on peut relier les deux germes et ainsi créer un segment valide de la triangulation de Delaunay.

# Diagramme de Voronoï

## Algorithme de passage

- Il s'agit de trouver les cellules adjacentes et ainsi de créer les segments valides pour la triangulation.
- Complexité :

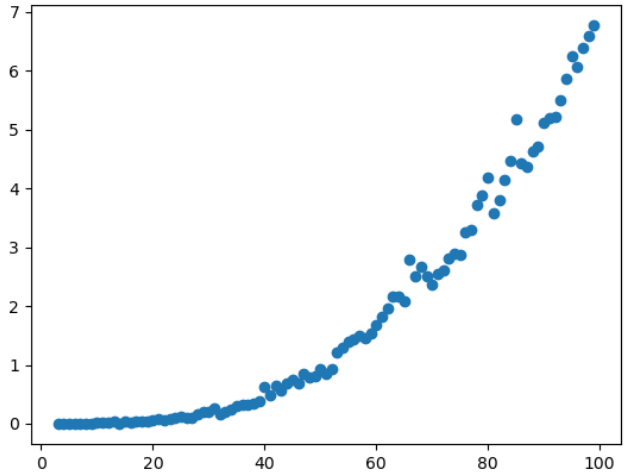



FIGURE – Temps de calcul du passage de Voronoï vers Delaunay d'un ensemble de points en fonction de sa taille

# Triangulation de Delaunay et implémentation

## Algorithme de passage

```
1  if coupe([points[i], points[j]], segment) and
2  test_scal([points[i], points[j]], segment) :
3      if not([points[i], points[j]]) in delaunay:
4          delaunay.append([points[i],points[j]])
5
6  a1, a2 = (y2 - y1)/(x2 - x1), (y4 - y3)/(x4 - x3)
7      if a1 == a2: return False
8      else :
9          b1 = y1 - a1 * x1
10         b2 = y3 - a2 * x3
11         xinter = (b2 - b1) / (a1 - a2)
12         return ((xinter < max(x1, x2)) and
13                         (xinter > min(x1, x2)))
```

# Diagramme de Voronoï

## Algorithme de passage

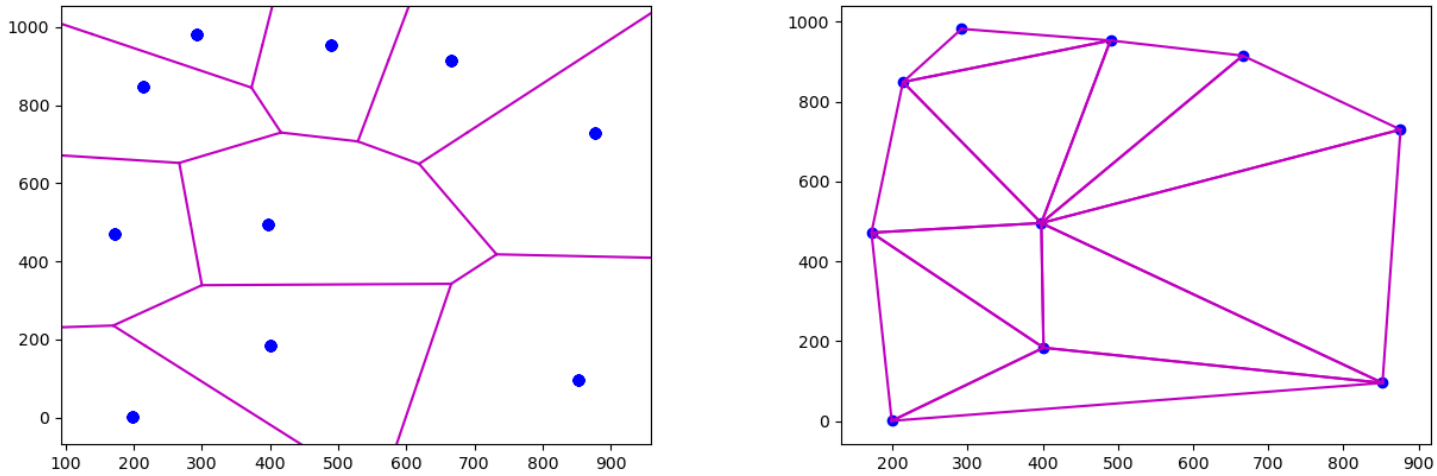



FIGURE – Diagramme de Voronoï et triangulation de Delaunay d'un ensemble de points

# Lien avec la reconnaissance faciale

## Principe

- On relève les points particuliers : 15 ici.
- Ensuite on s'intéresse à des grandeurs caractéristiques ne changeant pas trop sur un visage.
- L'écart entre les yeux, la taille des yeux, la largeur du nez et l'aire du polygone obtenu par triangulation.

# Lien avec la reconnaissance faciale

## Implémentation

```
1  def aire_t(t):
2      """aire = 1/2 * abs(det u, v)"""
3      a, b, c = t
4      xa, ya = a
5      xb, yb = b
6      xc, yc = c
7      return((1/2) * abs((xb - xa) * (yc - ya)
8                         - (xc - xa) * (yb - ya)))
```

# Lien avec la reconnaissance faciale

## Implémentation

On définit le taux de ressemblance comme suit :

$$\text{taux} = \frac{yeux1 + yeux2 + ecart + nez1 + nez2 + aire}{6}$$

où les grandeurs correspondent aux quotients des mesures de chaque image multipliés par cent.

# Lien avec la reconnaissance faciale

## Implémentation

```
1  def tauxderessemblance(p1, p2):
2      t1 = triangulation_bis(p1)
3      t2 = triangulation_bis(p2)
4      moy = 0
5      yeux_nez = yeuxnez(p1, p2)
6      compare = comparairetot(t1, t2)
7      m = len(yeux_nez)
8      for j in range(m):
9          moy += yeux_nez[j]
10     moy += compare
11     return (moy / (1 + m))
```

# Lien avec la reconnaissance faciale

## Test

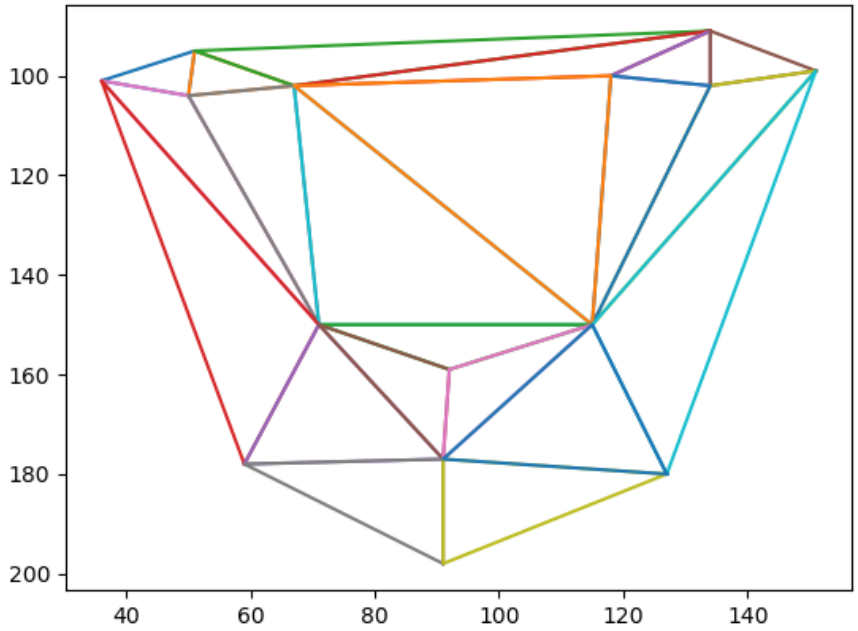



FIGURE – Visage et triangulation

# Lien avec la reconnaissance faciale

## Test

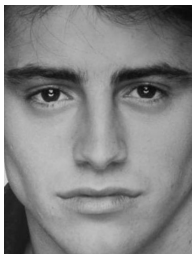
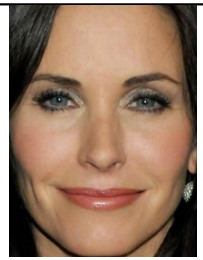
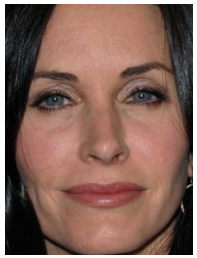
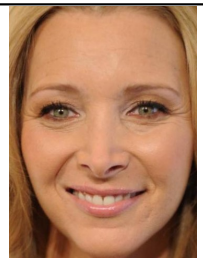
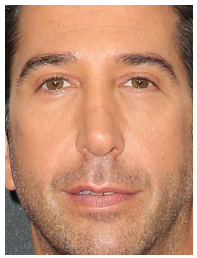
| | | | | | | |
|---|---|---|---|---|---|---|
| couples de photos |   |   |   |   |   |   |
| taux de ressemblance | 90,1% | 94,2% | 91,5% | 91,8% | 90,9% | 93,7% |

TABLE – Ressemblance des couples d'images

# Lien avec la reconnaissance faciale

## Test

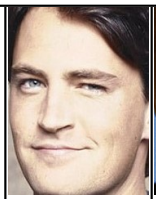
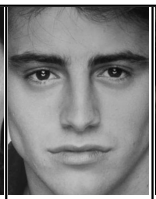
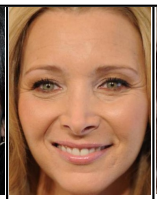
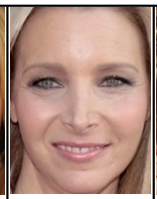
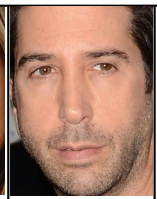
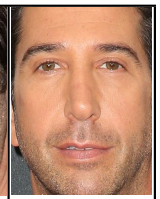
| photos |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  | 78,2 | 78,6 | 74,4 | 79,3 | 78,9 | 79,8 | 91,8 | 100 | 80,1 | 77,6 | 79,7 | 82,9 |

TABLE – Taux (en %) de ressemblance

# Conclusion

- Bonne exposition, visages ressemblants, peu d'expression.
- Plutôt rapide, semble fonctionner pour un ensemble de peu de personnes.
- Nécessité de trouver d'autres grandeurs de comparaisons.

# Algorithme de Delaunay naïf

```
1  import matplotlib.pyplot as plt
2  import math
3  import numpy as np
4  from random import *
5
6  def ensemble_point(n) :
7      x = [randint(0, 1000) for i in range(n)]
8      y = [randint(0, 1000) for i in range(n)]
9      return ([[x[i], y[i]] for i in range(n)])
10
11 def tracer_triangle(l):
12     for tri in l :
13         xa, ya = tri[0]
14         xb, yb = tri[1]
15         xc, yc = tri[2]
16         plt.plot([xa, xb, xc, xa], [ya, yb, yc, ya])
17     plt.show()
```

# Algorithme de Delaunay naïf

```
1  def distance(a, b):
2      """renvoie la distance entre deux points"""
3      xa, ya = a
4      xb, yb = b
5      return (((xa - xb) ** 2 + (ya - yb) ** 2) ** 1/2)
6
7  def mediatrice(a, b):
8      """renvoie le coeff directeur et l'ord. à l'or., ou 'vertical'"""
9      xa, ya = a
10     xb, yb = b
11     if xa == xb :
12         return (0, (ya + yb) / 2)
13     elif ya == yb :
14         return ('vertical', (xa + xb) / 2)
15     else :
16         pente = - (xb - xa) / (yb - ya)
17         ordalor = (yb + ya) / 2 - pente * (xb + xa) / 2
18         return (pente, ordalor)
```

# Algorithme de Delaunay naïf

```
1  def cerclecirc(triangle):
2      """renvoie centre, rayon et points du triangle initial"""
3      a, b, c = triangle
4      medab = mediatrice(a, b)
5      medac = mediatrice(a, c)
6      pentmab, ordmab = medab
7      pentmac, ordmac = medac
8      if pentmab == pentmac :
9          return ('impossible', 'infini', (a, b, c))
10     elif pentmab == 'vertical':
11         centre = (ordmab, pentmac * ordmab + ordmac)
12     elif pentmac == 'vertical':
13         centre = (ordmac, pentmab * ordmac + ordmab)
14     else :
15         x = (ordmac - ordmab) / (pentmab - pentmac)
16         y = pentmab * (ordmac - ordmab) / (pentmab - pentmac) + ordmab
17         centre = (x, y)
18     return (centre, distance(centre, a), (a, b, c))
```

# Algorithme de Delaunay naïf

```
1  def touslescercles(l):
2      """liste de points -> liste de ((centre, rayon), triangle)"""
3      cercles = []
4      triangles = touslestriplets(l)
5      for triangle in triangles :
6          cercles.append(cerclecirc(triangle))
7      return cercles
```

# Algorithme de Delaunay naïf

```
1  def verif(l):
2      """liste de points -> liste de bool d'indices ceux des triangles"""
3      rayon = 0
4      cercles = touslescercles(l)
5      n = len(cercles)
6      valide = [True for i in range(n)]
7      for i in range(len(cercles)):
8          for element in l :
9              if element not in cercles[i][2] :
10                 if cercles[i][0] == 'impossible':
11                     valide[i] = False
12                 elif distance(element, cercles[i][0]) < cercles[i][1] :
13                     valide[i] = False
14     return valide
```

# Algorithme de Delaunay naïf

```
1  def touslestriplets(l):
2      """liste de points -> liste de tous les triangles"""
3      n = len(l)
4      L = []
5      for i in range(n - 2):
6          for j in range(i + 1, n - 1):
7              for k in range(j + 1, n):
8                  L.append((l[i], l[j], l[k]))
9      return L
```

# Algorithme de Delaunay naïf

```
1  def triangulation(l):
2      """liste de points -> liste des triangles"""
3      triangles = touslestriplets(l)
4      tri = []
5      valide = verif(l)
6      for i in range(len(triangles)):
7          if valide[i]:
8              tri.append(triangles[i])
9      return tri
```

# Algorithme de Delaunay par insertion

```
1  def ensemble_point(n) :
2      x = [randint(0, 1000) for i in range(n)]
3      y = [randint(0, 1000) for i in range(n)]
4      return ([(x[i], y[i]) for i in range(n)])
5
6  grand_triangle = [(-1000, 0), (500, 2000), (2000, 0)]
7
8  def distance(a, b):
9      """renvoie la distance entre deux points"""
10     xa, ya = a
11     xb, yb = b
12     return (((xa - xb) ** 2 + (ya - yb) ** 2) ** (1/2))
```

# Algorithme de Delaunay par insertion

```
1  def cerclecirc(triangle):
2      """renvoie le centre, le rayon du cercle circonscrit"""
3      a, b, c = triangle
4      medab = mediatrice(a, b)
5      medac = mediatrice(a, c)
6      pentmab, ordmab = medab
7      pentmac, ordmac = medac
8      if pentmab == pentmac :
9          return ('impossible', 'infini')
10     elif pentmab == 'vertical':
11         centre = (ordmab, pentmac * ordmab + ordmac)
12     elif pentmac == 'vertical':
13         centre = (ordmac, pentmab * ordmac + ordmab)
14     else :
15         x = (ordmac - ordmab) / (pentmab - pentmac)
16         y = pentmab * (ordmac - ordmab) / (pentmab - pentmac) + ordmab
17         centre = (x, y)
18     return (centre, distance(centre, a))
```

# Algorithme de Delaunay par insertion

```
1  def triangulation_bis(l):
2      n = len(l)
3      cercles = [cerclecirc(grand_triangle)]
4      tri = [grand_triangle]
5      for k in range(n):
6          point = l[k]
7          p_illegal = []
8          c_ill = []
9          tri_ill = []
10         for i in range(len(cercles)):
11             centre, rayon = cercles[i]
12             a, b, c = tri[i]
13             if centre == 'impossible':
14                 for pt_tri in tri[i]:
15                     if pt_tri not in p_illegal:
16                         p_illegal.append(pt_tri)
17                 c_ill.append(i)
```

# Algorithme de Delaunay par insertion

```
            elif rayon - distance(centre, point)> 10 ** -8:
                for pt_tri in tri[i]:
                    if pt_tri not in p_illegal:
                        p_illegal.append(pt_tri)
                if tri[i] not in tri_ill:
                    tri_ill.append(tri[i])
                c_ill.append(i)
        if c_ill != []:
            for w in inversel(c_ill):
                tri.pop(w)
                cercles.pop(w)
        for t in tri_ill:
            enleve(t, tri)
```

# Algorithme de Delaunay par insertion

```
        for i in range(len(p_illegal)):
            if i < (len(p_illegal) - 1):
                for j in range(i+1, len(p_illegal)):
                    adj = nb_arrete_adja(p_illegal[i], p_illegal[j],
                    tri_ill)
                    if adj == 1:
                        cercles.append(cerclecirc([point, p_illegal[i],
                        p_illegal[j]]))
                        tri += [[point, p_illegal[i], p_illegal[j]]]
    a, b, c = grand_triangle
    tribis = []
    for m in range(len(tri)):
        if a not in tri[m] and not b in tri[m] and not c in tri[m]:
            tribis.append(tri[m])
    return (tribis)
```

# Algorithme de Delaunay par insertion

```
1  def inversel(l):
2      if len(l) == 1:
3          return l
4      else:
5          n = l.pop()
6          return ([n] + (inversel(l)))
7
8  def nb_arrete_adja(A,B,liste):
9      c = 0
10     for points in liste :
11         if A in points and B in points:
12             c += 1
13     return c
```

# Algorithme de Voronoï

```
1  ## Ce code provient du site:
2  https://github.com/jansonh/Voronoi/blob/master/Voronoi.py
3  ## Je n'ai pas implémenté cet algorithme
4
5  import random
6  import math
7  import heapq
8  import itertools
9
10 class Point:
11     x = 0.0
12     y = 0.0
13
14     def __init__(self, x, y):
15         self.x = x
16         self.y = y
```

# Algorithme de Voronoï

```
1   class Event:
2       x = 0.0
3       p = None
4       a = None
5       valid = True
6
7       def __init__(self, x, p, a):
8           self.x = x
9           self.p = p
10          self.a = a
11          self.valid = True
```

# Algorithme de Voronoï

```
1   class Arc:
2       p = None
3       pprev = None
4       pnext = None
5       e = None
6       s0 = None
7       s1 = None
8
9       def __init__(self, p, a=None, b=None):
10          self.p = p
11          self.pprev = a
12          self.pnext = b
13          self.e = None
14          self.s0 = None
15          self.s1 = None
```

# Algorithme de Voronoï

```
1   class Segment:
2       start = None
3       end = None
4       done = False
5
6       def __init__(self, p):
7           self.start = p
8           self.end = None
9           self.done = False
10
11      def finish(self, p):
12          if self.done: return
13          self.end = p
14          self.done = True
```

# Algorithme de Voronoï

```
1   class PriorityQueue:
2       def __init__(self):
3           self.pq = []
4           self.entry_finder = {}
5           self.counter = itertools.count()
6
7       def push(self, item):
8           # check for duplicate
9           if item in self.entry_finder: return
10          count = next(self.counter)
11          entry = [item.x, count, item]
12          self.entry_finder[item] = entry
13          heapq.heappush(self.pq, entry)
```

# Algorithme de Voronoï

```
1       def remove_entry(self, item):
2           entry = self.entry_finder.pop(item)
3           entry[-1] = 'Removed'
4
5       def pop(self):
6           while self.pq:
7               priority, count, item = heapq.heappop(self.pq)
8               if item is not 'Removed':
9                   del self.entry_finder[item]
10                  return item
11          raise KeyError('pop from an empty priority queue')
```

# Algorithme de Voronoï

```
1       def top(self):
2           while self.pq:
3               priority, count, item = heapq.heappop(self.pq)
4               if item is not 'Removed':
5                   del self.entry_finder[item]
6                   self.push(item)
7                   return item
8           raise KeyError('top from an empty priority queue')
9
10      def empty(self):
11          return not self.pq
```

# Algorithme de Voronoï

```
1  class Voronoi:
2      def __init__(self, points):
3          self.output = [] # list of line segment
4          self.arc = None  # binary tree for parabola arcs
5          self.points = PriorityQueue() # site events
6          self.event = PriorityQueue() # circle events
7          # bounding box
8          self.x0 = -50.0
9          self.x1 = -50.0
10         self.y0 = 550.0
11         self.y1 = 550.0
```

# Algorithme de Voronoï

```
1         for pts in points:
2             point = Point(pts[0], pts[1])
3             self.points.push(point)
4             # keep track of bounding box size
5             if point.x < self.x0: self.x0 = point.x
6             if point.y < self.y0: self.y0 = point.y
7             if point.x > self.x1: self.x1 = point.x
8             if point.y > self.y1: self.y1 = point.y
9         # add margins to the bounding box
10        dx = (self.x1 - self.x0 + 1) / 5.0
11        dy = (self.y1 - self.y0 + 1) / 5.0
12        self.x0 = self.x0 - dx
13        self.x1 = self.x1 + dx
14        self.y0 = self.y0 - dy
15        self.y1 = self.y1 + dy
```

# Algorithme de Voronoï

```
1       def process(self):
2           while not self.points.empty():
3               if not self.event.empty() and
4               (self.event.top().x <= self.points.top().x):
5                   self.process_event() # handle circle event
6               else:
7                   self.process_point() # handle site event
8
9           # after all points, process remaining circle events
10          while not self.event.empty():
11              self.process_event()
12          self.finish_edges()
13
14      def process_point(self):
15          # get next event from site pq
16          p = self.points.pop()
17          # add new arc (parabola)
18          self.arc_insert(p)
```

# Algorithme de Voronoï

```
1        def process_event(self):
2        e = self.event.pop()
3        if e.valid:
4            s = Segment(e.p)
5            self.output.append(s)
6            a = e.a
7            if a.pprev is not None:
8                a.pprev.pnext = a.pnext
9                a.pprev.s1 = s
10           if a.pnext is not None:
11               a.pnext.pprev = a.pprev
12               a.pnext.s0 = s
13           if a.s0 is not None: a.s0.finish(e.p)
14           if a.s1 is not None: a.s1.finish(e.p)
15           if a.pprev is not None:self.check_circle_event(a.pprev,e.x)
16           if a.pnext is not None:self.check_circle_event(a.pnext,e.x)
```

# Algorithme de Voronoï

```
1       def arc_insert(self, p):
2           # print('arc')
3           if self.arc is None:
4               self.arc = Arc(p)
5           else:
6               # find the current arcs at p.y
7               i = self.arc
8               while i is not None:
9                   flag, z = self.intersect(p, i)
10                  if flag:
11                      # new parabola intersects arc i
12                      flag, zz = self.intersect(p, i.pnext)
13                      if (i.pnext is not None) and (not flag):
14                          i.pnext.pprev = Arc(i.p, i, i.pnext)
15                          i.pnext = i.pnext.pprev
16                      else:
17                          i.pnext = Arc(i.p, i)
18                      i.pnext.s1 = i.s1
```

# Algorithme de Voronoï

```
1                    # add p between i and i.pnext
2                    i.pnext.pprev = Arc(p, i, i.pnext)
3                    i.pnext = i.pnext.pprev
4                    i = i.pnext # now i points to the new arc
5                    # add new half-edges connected to i's endpoints
6                    seg = Segment(z)
7                    self.output.append(seg)
8                    i.pprev.s1 = i.s0 = seg
9                    seg = Segment(z)
10                   self.output.append(seg)
11                   i.pnext.s0 = i.s1 = seg
12                   # check for new circle events around the new arc
13                   self.check_circle_event(i, p.x)
14                   self.check_circle_event(i.pprev, p.x)
15                   self.check_circle_event(i.pnext, p.x)
16
17                   return
```

# Algorithme de Voronoï

```
1                i = i.pnext
2
3            # if p never intersects an arc, append it to the list
4            i = self.arc
5            while i.pnext is not None:
6                i = i.pnext
7            i.pnext = Arc(p, i)
8
9            # insert new segment between p and i
10           x = self.x0
11           y = (i.pnext.p.y + i.p.y) / 2.0;
12           start = Point(x, y)
13
14           seg = Segment(start)
15           i.s1 = i.pnext.s0 = seg
16           self.output.append(seg)
```

# Algorithme de Voronoï

```
1       def check_circle_event(self, i, x0):
2           # print('cercle')
3           # look for a new circle event for arc i
4           if (i.e is not None) and (i.e.x  != self.x0):
5               i.e.valid = False
6           i.e = None
7
8           if (i.pprev is None) or (i.pnext is None): return
9
10          flag, x, o = self.circle(i.pprev.p, i.p, i.pnext.p)
11          if flag and (x > self.x0):
12              i.e = Event(x, o, i)
13              self.event.push(i.e)
```

# Algorithme de Voronoï

```
1       def circle(self, a, b, c):
2           # check if bc is a "right turn" from ab
3           if ((b.x - a.x)*(c.y - a.y) - (c.x - a.x)*(b.y - a.y)) > 0:
4           return False, None, None
5           # Joseph O'Rourke, Computational Geometry in C (2nd ed.) p.189
6           A = b.x - a.x
7           B = b.y - a.y
8           C = c.x - a.x
9           D = c.y - a.y
10          E = A*(a.x + b.x) + B*(a.y + b.y)
11          F = C*(a.x + c.x) + D*(a.y + c.y)
12          G = 2*(A*(c.y - b.y) - B*(c.x - b.x))
13          if (G == 0): return False, None, None # Points are co-linear
14          ox = 1.0 * (D*E - B*F) / G
15          oy = 1.0 * (A*F - C*E) / G
16          x = ox + math.sqrt((a.x-ox)**2 + (a.y-oy)**2)
17          o = Point(ox, oy)
18          return True, x, o
```

# Algorithme de Voronoï

```
1       def intersect(self, p, i):
2           # check whether a new parabola at point p intersect with arc i
3           if (i is None): return False, None
4           if (i.p.x == p.x): return False, None
5           a = 0.0
6           b = 0.0
7           if i.pprev is not None:
8               a = (self.intersection(i.pprev.p, i.p, 1.0*p.x)).y
9           if i.pnext is not None:
10              b = (self.intersection(i.p, i.pnext.p, 1.0*p.x)).y
11          if (((i.pprev is None) or (a <= p.y)) and ((i.pnext is None)
12          or (p.y <= b))):
13              py = p.y
14              px = 1.0 * ((i.p.x)**2 + (i.p.y-py)**2 - p.x**2)
15              / (2*i.p.x - 2*p.x)
16              res = Point(px, py)
17              return True, res
18          return False, None
```

# Algorithme de Voronoï

```
1        def intersection(self, p0, p1, l):
2         p = p0
3         if (p0.x == p1.x): py = (p0.y + p1.y) / 2.0
4         elif (p1.x == l): py = p1.y
5         elif (p0.x == l):
6             py = p0.y
7             p = p1
8         else:
9             z0 = 2.0 * (p0.x - l)
10            z1 = 2.0 * (p1.x - l)
11            a = 1.0/z0 - 1.0/z1;
12            b = -2.0 * (p0.y/z0 - p1.y/z1)
13            c = 1.0 * (p0.y**2 + p0.x**2 - l**2) / z0 - 1.0 *
14            (p1.y**2 + p1.x**2 - l**2) / z1
15            py = 1.0 * (-b-math.sqrt(b*b - 4*a*c)) / (2*a)
16        px = 1.0 * (p.x**2 + (p.y-py)**2 - l**2) / (2*p.x-2*l)
17        res = Point(px, py)
18        return res
```

# Algorithme de Voronoï

```
1       def finish_edges(self):
2           l = self.x1 + (self.x1 - self.x0) + (self.y1 - self.y0)
3           i = self.arc
4           while i.pnext is not None:
5               if i.s1 is not None:
6                   p = self.intersection(i.p, i.pnext.p, l*2.0)
7                   i.s1.finish(p)
8               i = i.pnext
9
10      def print_output(self):
11          it = 0
12          for o in self.output:
13              it = it + 1
14              p0 = o.start
15              p1 = o.end
16              print (p0.x, p0.y, p1.x, p1.y)
```

# Algorithme de Voronoï

```
1        def get_output(self):
2            res = []
3            for o in self.output:
4                p0 = o.start
5                p1 = o.end
6                res.append((p0.x, p0.y, p1.x, p1.y))
7            return res
```

# Algorithme de passage de Voronoï vers Delaunay

```
1  def test_scal(s1, s2):
2      a1, b1 = s1
3      xa2, ya2, xb2, yb2 = s2
4      xa1, ya1 = a1
5      xb1, yb1 = b1
6      v1 = (xa1 - xb1, ya1 - yb1)
7      v2 = (xa2 - xb2, ya2 - yb2)
8      return (abs(v1[0] * v2[0] + v1[1] * v2[1]) < 10 ** (-8))
```

# Algorithme de passage de Voronoï vers Delaunay

```
1  def coupe(s1, s2):
2      point1, point2 = s1
3      x3, y3, x4, y4 = s2
4      x1, y1 = point1
5      x2, y2 = point2
6      v1 = (abs(x1 - x2) < 10 ** (-5))
7      v2 = (abs(x3 - x4) < 10 ** (-5))
8      if  v1 == True and v2 == True:
9          return False
10     else:
11         if v2:
12             xinter = x3
13             a1 = (y2 - y1) / (x2 - x1)
14             b1 = y1 - a1 * x1
15             yinter = a1 * xinter + b1
16             return ((yinter < max(y1, y2)) and (yinter > min(y1, y2))
17             and (yinter < max(y3, y4)) and (yinter > min(y3, y4)))
```

# Algorithme de passage de Voronoï vers Delaunay

```
1         elif v1:
2             xinter = x1
3             a2 = (y4 - y3) / (x4 - x3)
4             b2 = y3 - a2 * x3
5             yinter = a2 * xinter + b2
6             return ((yinter < max(y1, y2)) and (yinter > min(y1, y2))
7             and (yinter < max(y3, y4)) and (yinter > min(y3, y4)))
8         else:
9             a1 = (y2 - y1) / (x2 - x1)
10            a2 = (y4 - y3) / (x4 - x3)
11            if a1 == a2:
12                return False
13            else :
14                b1 = y1 - a1 * x1
15                b2 = y3 - a2 * x3
16                xinter = (b2 - b1) / (a1 - a2)
17                return ((xinter < max(x1, x2)) and
18                (xinter > min(x1, x2)))
```

# Algorithme de passage de Voronoï vers Delaunay

```
1  def voronoïversdelaunay(points, segments):
2      delaunay = []
3      for i in range(len(points) - 1):
4          for j in range(i + 1, len(points)):
5              k  = 0
6              for segment in segments:
7                  if coupe([points[i], points[j]], segment) and
8                  test_scal([points[i], points[j]], segment) :
9                      if not([points[i], points[j]]) in delaunay:
10                         delaunay.append([points[i], points[j]])
11     return delaunay
```

# Algorithme de test sur des photos

```
1  def aire_t(t):
2      """aire = 1/2 * abs(det u, v)"""
3      a, b, c = t
4      xa, ya = a
5      xb, yb = b
6      xc, yc = c
7      return((1/2) * abs((xb - xa) * (yc - ya) - (xc - xa) * (yb - ya)))
8
9  def aires(t):
10     aire = []
11     for i in range(len(t)):
12         aire.append(aire_t(t[i]))
13     return aire
```

# Algorithme de test sur des photos

```
1   def yeuxnez(p1, p2):
2       yg1 = distance(p1[0], p1[3])
3       yg2 = distance(p2[0], p2[3])
4       yd1 = distance(p1[4], p1[7])
5       yd2 = distance(p2[4], p2[7])
6       ecart1 = distance(p1[3], p1[4])
7       ecart2 = distance(p2[3], p2[4])
8       nezg1 = distance(p1[8], p1[10])
9       nezd1 = distance(p1[9], p1[10])
10      nezg2 = distance(p2[8], p2[10])
11      nezd2 = distance(p2[9], p2[10])
12      l1 = [yg1, yd1, ecart1, nezg1, nezd1]
13      l2 = [yg2, yd2, ecart2, nezg2, nezd2]
14      comparaison = []
15      for i in range(len (l1)):
16          if l1[i] > l2[i]: comparaison.append (100 * (l2[i] / l1[i]))
17          else: comparaison.append(100 * (l1[i] / l2[i]))
18      return comparaison
```

# Algorithme de test sur des photos

```
1  def ecart_yeuxnez(p1, p2):
2      y = yeuxnez(p1, p2)
3      m = len(y)
4      moy = 0
5      for j in range(m):
6          moy += y[j]
7      return(moy / m)
8
9  def comparairetot(t1, t2):
10     aires1 = aires(t1)
11     aires2 = aires(t2)
12     a1, a2 = 0, 0
13     for i in range(len(t1)):
14         a1 += aires1[i]
15         a2 += aires2[i]
16     if a1 > a2 : return (a2 / a1) * 100
17     else : return (a1 / a2) * 100
```

# Algorithme de test sur des photos

```
1  def tauxderessemblance(p1, p2):
2      t1 = triangulation_bis(p1)
3      t2 = triangulation_bis(p2)
4      moy = 0
5      yeux_nez = yeuxnez(p1, p2)
6      compare = comparairetot(t1, t2)
7      m = len(yeux_nez)
8      for j in range(m):
9          moy += yeux_nez[j]
10     moy += compare
11     return (moy / (1 + m))
```

# Algorithme de test sur des photos

```
1  def trouverphoto(p):
2      t = 0
3      iplusproche = 0
4      for i in range(len(photos)):
5          if photos[i] != p:
6              if tauxderessemblance(p, photos[i]) > t:
7                  iplusproche = i
8                  t = tauxderessemblance(p, photos[i])
9      return (iplusproche)
```